Strip-Miner: Automatic Bug Detection in Large Software Codebases with a Low False Positive Rate

Fahad Ibrar

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Matthew Hicks, Chair
Ali R. Butt
Francisco Servant

April 16, 2020
Blacksburg, Virginia

Keywords: Software Engineering, Software/Program Verification, Automatic Bug Detection

Strip-Miner: Automatic Bug Detection in Large Software Codebases with a Low False Positive Rate

Fahad Ibrar

(ABSTRACT)

There are a number of techniques for automatic bug detection, most of them have a high false positive rate when used in practice. This work proposes an approach, named Strip-Miner, that combines simple dependency analysis of code with a data mining technique "frequent itemset mining" to reduce the false positive rate. We adopt a two phase approach 1) finding the potential bugs and 2) filtering the false positive ones. In the first phase we extract code elements and dependencies among them using static analysis and frequent itemset mining to find programming patterns where a deviation from these patterns is considered as a potential bug. In the second phase, we use the extracted dependencies to build dependency chains between program elements in a programming pattern and a lack of such a chain potentially makes a bug false positive.

Our evaluation on a set of 7 benchmarks consisting of large software code including OpenSSL, PostgreSQL, Git, FFMPEG, SQLite, Binutils and Putty shows that combining simple dependency analysis with pattern mining can significantly decrease the number of generated bugs. Using our approach we are able to reduce the number of generated bugs by up to 99.9% with a false positive rate of 65.19% and true positive rate of 34.18% on average as compared to an earlier frequent itemset mining based approach "PR-Miner".

Strip-Miner: Automatic Bug Detection in Large Software Codebases with a Low False Positive Rate

Fahad Ibrar

(GENERAL AUDIENCE ABSTRACT)

Every software code has bugs in it that can change its expected behavior. There have been a lot of efforts to automate the process of bug detection but most of the techniques proposed have a high rate of false alarms. Some of these techniques leverage the information available in software code to extract programming patterns that can be used to find potential bugs. Although such an approach has proved to be fruitful for finding bugs but large number of false alarms makes it almost useless in software development.

The elements present in a software code have relationships among them formally known as dependencies and the process of finding them is known as dependency analysis. There is a technique known as market basket analysis used by large retailers to find association between items. It works by looking for combinations of items that occur together frequently in transactions. Similarly, in a software code combinations of elements that occur together, can be used to find association between them. This technique is formally known as frequent itemset mining in the data mining domain. This work proposes an approach, named Strip-Miner, that combines dependency analysis with frequent itemset mining to reduce the rate of false alarms. We adopt a two phase approach 1)finding the potential bugs in code and 2)filtering the false alarms. In the first phase we extract code elements and dependencies among them and use frequent itemset mining to find programming patterns where a deviation from these patterns is considered as a potential bug. In the second phase, we use the

extracted dependencies to build dependency chains between program elements present in a programming pattern and lack of such a chain is an indication of false alarm.

Our evaluation on a set of 7 benchmarks consisting of large software code including version control systems, database management systems, software security libraries and utility software like media players shows that combining simple dependency analysis with frequent itemset mining can significantly decrease the rate of false alarms. Using our approach we are able to reduce the number of generated bugs by up to 99.9% with a false alarms rate of 65.19% and real bugs rate of 34.18% on average as compared to an earlier frequent itemset mining based approach "PR-Miner".

Dedication

Dedicated to my late father who enabled me to achieve my goals!

Acknowledgments

First of all I want to thank my parents who have always supported me. I would like to acknowledge all the sacrifices that they made for me to be successful in life.

I want to thank my advisor Prof. Matthew Hicks for his guidance and support. I would like to acknowledge the time and energy he spent to help me be a better version of myself.

I want to thank Prof. Francisco Servant and Prof. Ali R. Butt for serving on my thesis committee and for teaching me amazing courses.

I want to thank all of my teachers and mentors at FAST-NUCES, ITU, Punjab College and DPS who enabled me to achieve my dreams.

I want to thank my roommates and close friends Usman and Mohammad to help me go through tough times. I would also like to thank my lab-mates and friends Gagan, Steve and Mike who helped me to solve both technical and non-technical problems.

I want to thank Prof. Eli Tilevich, Prof. Clif Shaffer and Ms. Sharon Potter to help me with the logistics of graduate school.

In the end I want to thank Virginia Tech and its whole community for providing me an opportunity to learn and showcase my skills.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software code is prone to bugs that can change its expected behavior. Despite a large investment in software development and testing there are still a number of bugs present in wild that go undetected by traditional software testing techniques. These bugs can cost billions of dollars and can highly affect the software users[1]. In the recent years, there have been a lot of efforts both in industry and academia on devising robust techniques for rigorous software testing. But despite the fact that more human and financial resources have been allocated for it there are still no guarantees available to make a software bug free. A large body of research these days is on devising such bug detection approaches that minimize the need of a human in the process. This can not only reduce the cost of software testing but can help to minimize the uncertainties introduced by human factors.

There are many bug detection techniques presented to find bugs in software code. Some of these techniques are fully automatic i.e. they don't need any human expert in loop to find bugs. Though these techniques have been able to remove human expert from loop but they suffer a problem of high false positive rate. On the other hand there are techniques that work by using the bug patterns encoded by experts to find bugs. Such techniques have been proved to find bugs with greater accuracy with a very small to zero false positive rate. But they have limitations in terms that they need a human expert to encode bug patterns which in some cases is specific to a certain codebase. As a result these techniques only find bugs with known patterns and cannot find any other bugs, which limit their scalability and

generality.

For pattern encoding based techniques researchers have used two types of approaches. In first type a software developer or any person working on a software can write a static checker that check certain property in the code[2]. If code does not follow the property then it will be reported as a bug. In the other type, there is some kind of bug specification language in which a developer can specify a bug which is essentially a behavior that code should not show in execution[3][4][5][6][7][8][9][10][11]. These techniques take the bug specification and find such behavior in code.

Recently, researchers have been focusing more on developing techniques that are fully automatic to fulfill the bigger dream of automated software engineering and to eliminate cost of using human expert in loop[12][13][14][15][16][17][18]. But at the same time they have a bigger challenge of reducing false positive rate. Although, some of these techniques have been able to find real bugs but they still suffer from an intrinsic problem of high false positive rate. Especially in the case of static analysis techniques the false positive rate is so high that it makes these techniques nearly useless in real software development and testing. Software developers working on a software project has limited amount of time because of strict deadlines. In such a scenario, developers are hesitant to use any of such tools because the cost of using a tool is much higher as compared to the benefits or in other words number of true positive bugs detected.

There have been a lot of research to leverage the knowledge available in code bases. People have been trying to use machine learning, deep learning and data mining based approaches to find patterns in the code with an assumption that violation of these patterns can produce bugs in the code. Machine learning and deep learning based techniques suffer from a problem of absence of enough data therefore they are not been able to produce any useful results. Whereas, data mining based techniques are more promising in this scenario and they can be

scaled with increase in the amount of data.

In this paper, we present a hybrid approach that combines both data mining based bug detection and manual pattern encoding. In current literature all of the techniques focus on encoding the bug patterns itself to detect bugs. **But none of the techniques focused on identifying and encoding false positive patterns. Our hypothesis is, if we can identify these patterns, they we can be used to remove false positives**. Such a technique can be more useful because reducing false positive rate can make these tools more appealing to be used by software developers. Furthermore, it can cut the cost of developer's time for going through all the reported bugs and eliminating the one that are false positives.

There is an approach named PR-Miner[16], presented by Li et al. PR-Miner is based on a data mining based approach named frequent itemset mining. It mines programming pattern in a code and use them to generate programming rules. These rules are used to find potential bugs that are essentially the violations of the rules in the code. PR-Miner has been able to find bugs but suffers from the problem of high false positive rate which can go as high as 98% for some benchmarks. By doing a detailed study of the false positives, we observe that they follow some patterns. Furthermore, the patterns can be generalized into three different categories and encoded to identify false positive bugs.

Most of the false positives are produced because of the absence of dependency check between items in a rule e.g. if there is a rule $X \implies Y$ then there should be a dependency between X and Y. But in most of the rules generated by PR-Miner, no dependency exists, hence bugs identified by them are mostly false positives. In code these dependencies can be very complex. The challenge is how we can encode them to identify the false positive bugs. Instead of using a heavy weight dependency detection approach that take large computation resources, we introduce a light weight approach which has a negligible effect on the performance as compared to PR-Miner but in return its able to decrease the number of generated potential

bugs by 95.9% with a false positive rate of 65.19% and true positive rate of 34.81% on average.

Our approach for dependency extraction is based on simple heuristics that if combined can describe complex dependencies. We formulate these heuristics by manually inspecting the false positives and true positive bugs. The heuristics capture such dependencies that hold true in most of the cases hence can identify a potential true positive among a set of both true positives and false positives. As in all of pattern mining based approaches, false positive rate is usually very high so in that case if we can identify true positives then we can mark the remaining as false positives.

Following are our contributions in this work.

(1) It's the first ever detailed study of false positives generated by PR-Miner. In current literature, researchers have been focusing on finding novel and more efficient techniques but little attention has been given to study the false positives themselves.

(2) We provide an updated implementation of PR-Miner approach using LLVM which is a more powerful and emerging industry standard compiler framework. All of the previous approaches have been using GCC for static analysis. But due to C/C++ language restriction of GCC such approaches are not applicable for other languages. On the other hand using LLVM intermediate representation make our technique independent of any programming language.

(3) We introduce a simple technique to capture dependencies among code elements which can extract dependencies while parsing source code.

(4) We provide a detailed comparison analysis of our technique with PR-Miner.

Chapter 2

Review of Literature

2.1 Dynamic Analysis Based Approaches

Ernst et al present a tool named Daikon that can dynamically detect likely invariant in a program[19]. It runs a program and observe the values that it computes. Using the computed values it can report the properties that are true over the executions. Csallner et al present a variant inference tool that can infer consistent invariant for interfaces and overridden methods. Their tool try to solve the internal consistencies that can arise in function overriding while using Daikon[20]. Gabel et al present a technique named Javert that can learn temporal properties from execution traces of a program[12]. The key idea behind Javert is to learn less complex and generic properties that can make up for more complex one. Pradel et al present a dynamic analysis based technique to infer the program specifications[15]. It works by analyzing the method traces of multiple related objects. They also present an automatic protocol conformance checker that combines dynamic analysis to infer multi-object protocols and a static checker of API usage constraint[14]. Newsome et al propose dynamic taint analysis for automatic detection of overwrite attacks[21]. This approach does not need source code or special compilation for the monitored program, and works on commodity software.

2.2 Bug Detection Using Specifications

Legunsen et al present a study on the effectiveness of finding bugs using available java API specifications[22]. They suggest that these API specifications can find bugs in programs but their false positive rate is very high. Hallem et al design a flexible, extension language for specifying analyses and an algorithm for executing the extensions[3]. The language, allows the users to specify a broad class of analyses in terms that resemble the intuitive description of the rules. Their system, executes these analyses using a context-sensitive, interprocedural analysis. Sivakorn et al present a black-box testing framework for analyzing SSL/TLS hostname verification implementations, which is based on automata learning algorithms[7]. It utilizes a number of certificate templates, in order to test different rules from the corresponding specification. Livshits et al propose a static analysis approach based on a scalable and precise points-to analysis[4]. It translate user-provided specifications of vulnerabilities into static analyzers. Chen et al encode security properties as pushdown automaton and uses model checking to verify if any of the properties are being violated in a program[10]. Srivastava et al propose an approach that takes multiple implementation of an API, definitions of security checks and security-sensitive events, and compute security policies enforced by a check before and after an API call[23]. It reports the missing checks by comparing the policies across different implementations. Gu et al presents a tool named IMChecker to detect API misuse in C code[9]. It uses a domain specific language to specify API usage constraint along with constraint-directed static analysis to detect the misuses. Jana et al presents a tool named EPEX, it uses under-constrained symbolic execution with a fallible function and all of its callers functions and error specifications as an input and report bugs introduced by incorrect error handling[8].

2.3 Data Mining Based Programming Rules Extraction

Li et al presents an approach named PR-Miner to infer programming rules from a program[16]. It uses FPClose a frequent itemset mining algorithm to mine frequent patterns in a program and use them to infer programming rules. Bian et al presents an approach similar to PR-Miner named NAR-Miner[18]. It also uses FPClose to mine frequent pattern in a program but infer negative association rules only.

2.4 Static Analysis Based Approaches

Cova et al present an approach to find vulnerabilities in executable binaries using static analysis with symbolic execution[24]. Murali et al present a Bayesian framework that can learn probabilistic specifications from large, unstructured code corpora, and then use these specifications to statically detect anomalous, hence likely buggy, program behavior[25]. Saha et al propose an approach to find resource release omission faults in systems software[26]. It focuses on the error-handling code of each function. Yamaguchi et al propose a method to expose missing checks in source code[27]. It proceeds by statically tainting source code and identifying anomalous or missing conditions linked to security-critical objects. Viega et al describe a tool for statically scanning security-critical C source code for vulnerabilities[2]. Cheng et al develop a tool to automatically identify evidentiary data in the permanent storage of an Android device[28]. It first pre-computes an App Evidence Database (AED) via static analysis of a number of apps and then, matches the files on a smartphone's permanent storage against the AED to identify the files that could store evidentiary data. Tillmann et al uses symbolic execution to infer human understandable specifications of methods in software[29]. Kang et al presents a tool named APEx that uses under-constrained symbolic execution to

automatically infer error specification of API functions[30]. It leverages API usage pattern to infer specifications in C code. Jovanovic presents a tool named Pixy which can detect cross-site scripting vulnerabilities in PHP scripts[31]. It use flow sensitive, inter-procedural and context sensitive data flow analysis to discover vulnerable points in a program.

Chapter 3

Background

Our technique partially based on the approach proposed by PR-Miner which uses frequent itemset mining to infer programming rules. In this section we describe how frequent itemset mining works and how PR-Miner uses that technique to generate programming rules that can be used to find bugs in code.

3.1 Frequent Itemset Mining

Frequent itemset mining is a well known approach in data mining which has application in various domain including shopping where this approach is being used to analyse customer's purchase items and recommend items based on that, biotechnology where it is being used to find DNA sequences. In general, this technique work on large databases where it can find patterns of occurrence or co-occurrence of items. A database consists of itemsets that contain different items. Frequent itemset mining can find frequently occurring patterns in a database, formally called frequent sub-itemsets. Frequency of occurrence of a an sub-itemset is called support count and every sub-itemset should occur at least a minimum number of times i.e. support threshold to be considered as frequent. All the itemsets supporting a particular sub-itemset are called supporting itemsets. For example if we have a database D of itemsets as follows

$D = \{\{a, b, c, e\}, \{b, c, d, e\}, \{b, c, f, e\}, \{c, d, f, g\}\}$

If minimum support threshold is 3 then there are following frequent sub-itemsets with their respective supports

$\{b:3\}$, $\{c:4\}$, $\{e:3\}$, $\{b,c:3\}$, $\{b,e:3\}$, $\{c,e:3\}$ $\{b,c,e:3\}$

If we take example of sub-itemset $\{b,c,e\}$, it has a support count of 3 with supporting itemsets $\{a,b,c,e\}$, $\{b,c,d,e\}$ and $\{b,c,f,e\}$

There are different algorithms available including Apriori, FP-Growth, FP-Close etc. for frequent itemsets mining. First two of these generate all possible frequent sub-itemsets and their number can be exponential. But on the other hand FP-Close only generates closed frequent sub-itemsets. A sub-itemset is closed if none of its super sub-itemsets have same support. For instance if we consider above itemsets database D, the sub-itemsets $\{b\},\{e\},\{b,c\},\{b,e\}$ are not closed because there exists a super sub-itemset $\{b,c,e\}$ with same support i.e. 3. Whereas, sub-itemset $\{c\}$ is closed because none of its super sub-itemset has same support. Therefore, the total number of sub-itemsets generated is 2 which includes $\{c\}$ and $\{b,c,e\}$. FP-Close can greatly increase the performance of sub-itemsets mining by decreasing the space requirement and processing time.

Frequent sub-itemsets generated by FP-Close are used to generate association rule. An association rule is an implication of the form $X \implies Y$, which means if an item X is present in an itemset then an item Y should also be present in the itemset. There are a lot of rules that can be generated from frequent sub-itemsets but most of them are not very useful in other words they only show a week relationship because the items in such a rule only co-occur together a small number of time. Therefore, there is a threshold name confidence that is used to only generate rules that contain items occurring together more than a particular threshold. Confidence for a rule $X \implies Y$ is a measure of how often items in Y appear in an itemset I that also contains items in X. It can be calculated by

the formula $support(I)/support(X)$, where $support(I)$ and $support(X)$ are total number of itemsets supporting I and X respectively. Rules that generated using closed frequent itemsets are closed rules which means they can subsume other rules. For example if we consider the itemsets database D, there is a closed frequent sub-itemset $\{b, c, e\}$ with support 3 generated by FP-Close. From this sub-itemset following six closed rules can be generated, shown with their respective confidence

$\{b\} \implies \{c, e\}, confidence = 3/3 = 1$

$\{c\} \implies \{b, e\}, confidence = 3/3 = 1$

$\{e\} \implies \{b, c\}, confidence = 3/3 = 1$

$\{b, c\} \implies \{e\}, confidence = 3/3 = 1$

$\{b, e\} \implies \{c\}, confidence = 3/4 = 0.75$

$\{c, e\} \implies \{b\}, confidence = 3/3 = 1$

All the remaining rules are subsumed by the above rules. For instance rule $\{c\} \implies \{e\}$, is subsumed by the above rule $\{c\} \implies \{b, e\}$. The generated closed rule with high confidence are then used to find relationship between items.

3.2 PR-Miner

The idea of PR-Miner is, software code follow certain implicit rules like if there is a call to function *malloc* then there should also be a call to function *free*. There can be large number of such rules which makes it hard to write these rules manually. Therefore, PR-Miner proposes to use frequent itemset mining to extract these rules automatically. It works by parsing the source code and generating itemsets out of it. An itemset consists of all the elements

in a function scope. All of these extracted itemsets collectively make an itemset database. PR-Miner uses frequent itemset mining algorithm FP-Close to mine all the closed frequent sub-itemsets from the database, mined frequent sub-itemsets are then used to generate rules.

Rules can be generated from a frequent sub-itemset I by dividing the items in the set into two parts such that for a rule $X \implies Y$, X is a subset of of I and Y is $I - X$. Then for each generated rule calculate the confidence and discard the rules with confidence less than confidence threshold. For a given closed frequent sub-itemset total number of possible rules can be up to $2^n - 2$ where n is the number of items in the rule[16]. Examining, all of these rules to see if they have a confidence value greater than or equal to confidence threshold is not very efficient in terms of space and time complexity. Because, in some cases total number of items in a sub-itemsets can be more than 20[16]. To solve this issue PR-Miner proposes to examine only closed rules because all other rules can be subsumed by closed rules (as described in section 2.1). Furthermore, PR-Miner introduces a condensed format to represent all the rules generated from a closed frequent sub-itemset I. The format is as follows

$$I : s|\{C_1 : s_1|s_1 > s\}...\{C_m : s_m|s_m > s\}$$

Where $C_1...C_m$ are subsets of I with supports $s_1...s_m$ respectively. The support of all of these subsets is different from the support s of I. The confidence of each rule represented by this format can be calculated as, for a rule $X \implies Y$ extracted from the format, if X is a subset C_i of I then confidence is s/s_i whereas in all other cases confidence is 1. PR-Miner proposes that the problem of rule generation can be reduced to finding all of the subsets C_i that has a support greater than I. To solve this problem it proposes an algorithm, which first sort all of the frequent sub-itemset by descending order of their support. Then it finds all the common subsets in each pair of closed frequent sub-itemsets. It achieve this task by

applying FP-Close on closed frequent sub-itemsets with minimum support of 2. A subset C_i can belong to multiple sub-itemsets, therefore the support s_i of C_i is equal to the support of its super sub-itemset that have maximum support. For each subset C_i it iterates on all of its supporting sub-itemsets and assign C_i to those sub-itemsets I (in condensed format) that have a support s less than s_i.

Chapter 4

Approach

4.1 Overview

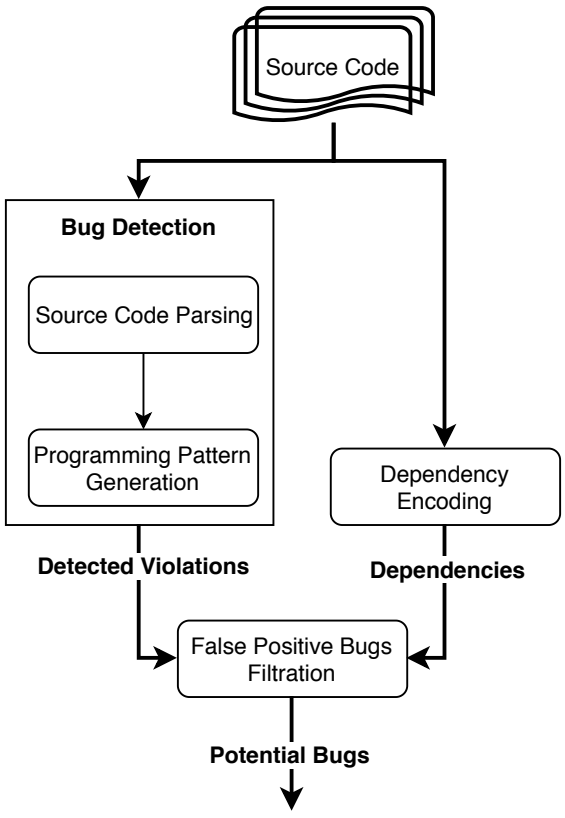

Figure 4.1: Overview of our Approach: Strip-Miner.

Our approach consists of two phases. In the first phase, we make use of PR-Miner idea to extract programming rules and using those rules we find bugs in source code. In the second

phase we use pattern encoding idea to encode dependencies present in code and using those dependencies we filter false positive bugs. An overview of our approach is shown in figure 4.1.

4.2 Bug Detection

In this phase, we use frequent itemsets mining to extract programming patterns from source code and using those patterns we can find potential bugs. For using frequent itemsets mining we need to prepare data in more structured format i.e. an itemsets database, that can be fed to the mining algorithm. For converting source code elements into itemsets we need to parse the code. Once frequent sub-itemsets are generated from parsed code, they are used to generate programming rules. The rules are ranked according to their confidence values and violation to these rules are considered as candidate set of potential bugs. Details of these steps are provided as follows.

4.2.1 Source Code Parsing

There are couple of different ways that can be used to parse source code. Most of the earlier approaches[16][18] use GCC for parsing the source code. These approaches parse the source code without compiling it which results in the introduction of those code elements that are part of dead code. In other words, when we compile source code the binaries generated include only live code which actually run on a machine. Therefore, the analysis of code present in compiled binaries is closer to dynamic analysis hence making the analysis more precise and accurate. But the challenge here is, a compiled binary does not include any source code information i.e. actual variable names and data types. To perform our analysis

we need to have this information of source code. To overcome this challenge we use LLVM intermediate representation(IR) which has source code information embedded in it. LLVM IR provides a hierarchical structure, in the hierarchy the top level entity is called a module which contains global variables and functions. A module can be parsed using LLVM API to extract the instructions present in functions. These instructions contain source code information including variable names, data types etc. and all the elements present in a function scope makes an itemset.

While parsing the source code, there is a challenge that we need to determine what source code element should be added to an itemset. One approach can be to add all the elements in a function scope. But adding all the elements can generates a lot of noise that results into uninteresting code patterns. For instance, almost all the functions has some kind of primitive type variables like *int* or *char* adding these variables do not give any unique information but they can increase the required computation. Therefore, we only include all the variables of non-primitive data types (structs in the case of C), all the fields of an object that are being set, all the function calls and data type of their parameters and return value. Furthermore, instead of using variables name we use data type name with a reason that in different functions, variables belonging to different data types can have same name, it can lead to misleading patterns. Another issue that can arise is different code elements can have same name like there can be a function named *next* to get next token in a string parser and there can also be a field named *next* in a linked list. To avoid such ambiguities we use prefixes to append with each of element types like for a function we added a prefix "F-" in front of the function name, for a data type we add a prefix "T-" in front of data type name and for a field of a data type we add a prefix "T-struct.<struct-name>." in front of it. In last step parsed elements are hashed to numbers before adding them to their respective itemsets.

4.2.2 Programming Patterns Generation

A programming pattern can be defined as a group of code elements that occur together. Every function in code has a specific set of elements that are encoded in the form of an itemset. We can call this itemset a pattern of that function. There can be as many patterns as the number of functions in a software but these patterns are not generalized enough, in other words these patterns are specific to their corresponding functions. Although, these patterns do provide some information but they are not interesting because they cannot be generalized across the whole software. Therefore, we need some approach to extract generalized patterns.

We use frequent itemset mining to extract programming patterns from code. Itemset database created by parsing source code along with minimum support is given as an input to the frequent itemset mining algorithm FP-Close to generate close frequent itemsets. For generating programming rules we use *ClosedRules* algorithm proposed by PR-Miner[16]. As a last step the rules are ranked based on their confidence score and violation of these rules are candidate potential bugs in the code.

4.3 False Positive Bugs Filtration

```
git-1.5.3.5/builtin-config.c:

69   static int get_value(const char* key_, const char* regex_)
70   {
       ......
81   local = repo_config = xstrdup(git_path("config"));
       ......
124  free(key);
```

Figure 4.2: A rule generated by PR-Miner in Git code. According to the rule if there is a call to *git_path* then there should also be a call to *free*.

Rules generated by itemset mining algorithms are solely based on the co-occurrence of items present in itemset database. Such a technique that only consider co-occurrence can work perfectly in problems like market basket analysis where only factor considered is, what items a customer purchases together. But in the case of more complex data like software code there can be factors other than just co-occurrence of code elements and ignoring them can result in false positive bugs. For example, there is a rule in git code shown in figure 4.2, according to this rule if there is a function call to *git_path* then there should also be a call to *free*. But on examining the code we found that there is no relation between the two and they can be used independently. This rule has multiple violations which are essentially false positives. There are large number of such rules generated by frequent itemset mining.

```
git-1.5.3.5/builtin-init-db.c:

90  DIR *subdir = opendir(template);
      .....
102 closedir(subdir);
```

Figure 4.3: A rule generated by PR-Miner in Git code. According to the rule if there is a call to *opendir* then there should also be a call to *closedir*.

To eliminate such type of false positives we can use dependency information present among code elements. For instance if we take example of code in figure 4.3, it shows a rule present in Git code. According to this rule if there is a function call to function *opendir* to open a directory structure then there should also be a call to function *closedir* to close the corresponding directory. A failure to closing the directory can cause a bug in software. These function calls occur together, if we look at these calls we can see that they have a data dependency i.e. *opendir* return a *struct* of type *DIR* with variable name *subdir* and this variable is later used in the code to call the function *closedir*. We can leverage such dependency information to eliminate the false positive bugs. But the challenge here is how to encode dependencies such that they do not make the approach specific to a certain code base or

a certain set of bugs. After careful analysis of generated rules we observe that encoding a small set of basic dependencies, that if combined can represent complex dependencies, can be used to filter out false positives. Details of these dependencies along with their extraction and encoding are discussed in section 4.4. In addition to false positives generated by lack of

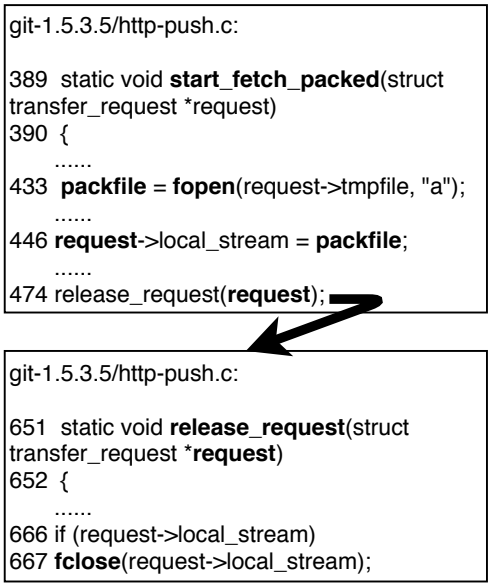

Figure 4.4: A call to *fopen* is present in *start_fetch_packed* function whereas a corresponding call to *fclose* present in a callee "*release_request*" of *start_fetch_packed*.

dependency information we observe that there are some violations for rules because those rules span over multiple functions. As PR-Miner reports violation based on intra-procedural analysis therefore all those rules that spans over multiple functions are reported as violations. For example there is a rule $fopen \implies fclose$ in Git code which says that if there is a call to a function *fopen* to open a file then there should also be a call to a function *fclose* to close that file. There is a violation reported for the rule that says $fclose$ is missing in a function having a call to $fopen$. The violation function *start_fetch_packed* is shown in figure 4.4. In the code, on line 433 *fopen* function is called that assign a file descriptor to variable *packfile*. Later in the code on line 446, the same file descriptor is assigned to a field *local_stream* belonging to a struct variable named *request*. On line 474, the variable

request is passed to another function *release_request* as parameter. There is no function call to function *fclose* in the scope of *start_fetch_packed*. But if we look at the code of function *release_request* (also shown in figure 4.4), on line 667 there is a function call to *fclose* that take field *local_stream* of request variable as a parameter. This function call closes the file being opened in *start_fetch_packed*. We can eliminate such false positives by doing inter-procedural checking discussed in section 4.5.

4.4 Dependency Extraction & Encoding

In this work, we are considering three types of dependencies, 1)function parameters dependency, 2)function return type dependency and 3)object state and usage dependency, that are extracted along with code parsing.

4.4.1 Function Parameters Dependency

This is a very simple dependency, according to it if there is a function call in a function scope then all of its parameters should be present in the same scope. It means if a rule is $X \implies Y$, where X is a function and Y is its parameter then this is valid rule by the virtue of the dependency. But on the other hand if X is a parameter and Y is the function then the rule $X \implies Y$ not necessarily be always true because a variable of same type as parameter can exist independent of the function. For instance there is a rule generated in the first phase of approach i.e. $DIR \implies closedir$. One of the functions violate this rule. On inspecting the function we see that the function has DIR passed as parameter to read the directory structure. Although, it required to close the directory opened for reading but, the function *closedir* not necessarily needs to be present in that function, its possible that the caller is

closing the directory. So encoding function parameters dependency can eliminate such false positives. For encoding this dependency, we extract all the function calls along with their parameters and store them in a dependency map for the respective function.

4.4.2 Function Return Type Dependency

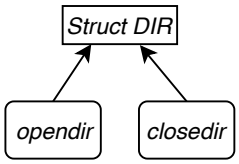

Figure 4.5: Multi level dependency present in Git Code

This dependency says that if there is a function call in a function scope then a variable of type as its return type is assumed to be present in the scope. More formally if there is a rule $X \implies Y$, where X is a function call and Y is a variable of type same as the function's return type then such a rule is always considered to be true. On contrary, a rule in which X is the variable of type same as return type and Y is function call itself then it does not need to be true. An example of such a rule is present in Git i.e. $opendir \implies DIR$, where opendir is a function that returns a variable of type DIR. If there is a function call to *opendir* then a variable of type DIR should be present in the function scope. On the other hand a variable of type DIR can occur independent of *opendir*. By combining this dependency with function parameters dependency we can encode more complex dependencies. For instance, if there is a rule $opendir \implies closedir$, then it can be described by combining the dependencies $opendir \implies DIR$ and $closedir \implies DIR$ as shown in figure 4.5. For encoding function return type dependency, we extract the return type of each function and store it in a dependency map for the function.

4.4.3 Object State and Usage Dependency

```
git-1.5.3.5/builtin-unpack-objects.c:

63  z_stream stream;
    ......
70  stream.next_in = fill(1);
71  stream.avail_in = len;
72  inflateInit(&stream);
```

Figure 4.6: A rule present in Git code. According to the rule the fields *next_in* and *avail_in* of variable *stream* should be set before passing it to function *inflateInit* as a parameter.

State of an object is defined by the values of its fields at any point in the program. If there is an object passed to a function as parameter and some of its fields are being set before calling the function then it is possible that the execution of function depends on the value of those fields. We can encode this dependency to capture the relation between the function call and object's fields. For instance, in figure 4.6 there is a code snippet present in git code. In this snippet, a function *inflateInit* is called, which takes a parameter of type *z_stream*. Here we can see that the fields *next_in* and *avail_in* of *z_stream* are set before calling the function. The function *inflateInit* is member function of a well known compression library named **zlib**[32]. According to the documentation of function *inflateInit*, it is required to set the fields *next_in* and *avail_in* before calling it. There are many other examples which involves such a dependency and by encoding them we can filter true positive bugs from false positives. For encoding this dependency, we extract all the code statements, present before the function call in the function scope, that set a field of an object passed as a parameter to function and store them in a dependency map for the function.

Our approach extract these dependencies for each code element and store them in a hashmap. For function parameter dependency, we extract the function name and all of its parameter from function call instructions in LLVM IR and store them in the map where key is the function name. Same is the case with function return type dependency. However, for object

state and usage dependency we extract all the instructions that set fields of a struct object and if that object is being used as a parameter in a function call then all fields that are being set before function call are added in function dependency list.

4.5 Inter-procedural Checking

We perform inter-procedural checking by storing all the callers and callees of a function. It can be divided into backward checking and forward checking. In backward checking, our approach check all the caller paths of the violation function. If it finds the missing elements in all of the caller paths then the violation is false positive. On the other hand forward checking is performed by checking the callee paths of a violation function. If any of the path has the missing elements then the violation is false positive.

4.6 Dependency Based False Positives Filtration

Algorithm: *DependencyCheck(X , Y , DepGraph)*
Input: X = set of items in LHS of a rule,
Y = set of items in RHS of a rule
$DepGraph$ = *Dependency graph*
Output: True if dependency exists False otherwise
1: $X_{depitems} \leftarrow \{i | i \epsilon DepGraph(X)\}$
2: **if** $X_{depitems} \cap Y$ is not $\emptyset$
3: **return** True
4: **else**
5: $Y_{depitems} \leftarrow \{i | i \epsilon DepGraph(Y)\}$
6: **if** $X_{depitems} \cap Y_{depitems}$ is not $\emptyset$
7: **return** True
8: **else**
9: **return** False

Figure 4.7: Data dependency based algorithm to remove false positive violations.

For removing false positive bugs using data dependency, we use the algorithm shown in figure 4.7. The function *DependencyCheck* is given left hand side i.e. X of a rule, right hand

side i.e. Y of a rule and a dependency graph *DepGraph* containing all the dependencies extracted from code. In graph, the vertices indicate the items in code and the edges indicate the dependencies between them. The algorithm first extract all the items present in the dependency graph of the items in X, in line 1. In line 2, it checks if there is an item of Y present in the dependency graph of X. If an item exists then function returns true indicating that there is a dependency between left and right sides of the rule. Otherwise it extract all the elements in the dependency graph of Y and check if there is any common item in dependency graph of X and Y. If there exists a common item then the function returns true otherwise false, line 4-9.

4.7 Inter-procedural Checking Based False Positives Filtration

```
Algorithm: ForwardCheck(I, CalleeList)
Input: I = Item need to find
       CalleeList = Callees of function
Output: True if found I in a callee False otherwise
1: for callee in CalleeList
2:   if callee has I
3:     return True
4:   else
5:     ForwardCheck(I, callee.CalleeList)
6: return False
```

Figure 4.8: Forward inter-procedural checking algorithm.

As described in section 4.5, inter-procedural checking can be of two types. First one is forward or callee checking, the algorithm in figure 4.8 show the implementation algorithm for it. For each violation function *f*, the function *ForwardCheck* is called on the right hand side items of violated rule. It recursively check all the callees of *f*. If it finds the item in any of the callee then it returns true otherwise it returns false. Second is backward or caller checking, figure 4.9 shows the algorithm for it. Backward checking similar to forward

```
Algorithm: BackwardCheck(I, CallerList)
Input: I = Item need to find
       CallerList = Callers of function
Output: True if found I in all callers False otherwise
1: if CallerList is empty
2:    return False
3: for caller in CallerList
4:    if caller has I
5:       return True
6:    else
7:       if (ForwardCheck(I, caller.CalleeList) ∨
         BackwardCheck(I, caller.CallerList)) is False
8:          return False
9: return True
```

Figure 4.9: Backward inter-procedural checking algorithm.

checking with a difference that in addition to check all the callers of a function f recursively, it also perform forward checking of functions in caller path (line 5). The reason for forward check is, it is possible that the item I is present in a one of the callees of a caller.

Chapter 5

Evaluation

5.1 Experimental Setup

We implemented our approach as prototype system to detect bugs in real world software. For evaluation of our approach we gather a list of 7 benchmark software shown in Table 5.1. These benchmarks are selected because they are widely used in evaluation of earlier approaches for bug detection [18][33][16]. In addition to this, these benchmark belongs to several different domains therefore it can help to generalize our results. We use an implementation of PR-Miner(implemented as part of our approach) as baseline for comparison. There are several reason for choosing PR-Miner as baseline, firstly our approach is an extension of existing PR-Miner approach therefore we can evaluate the performance gain of our approach. Furthermore, PR-Miner is the only approach that uses frequent itemset mining to extract programming rule and detect bugs.

Software	**Version**
Git	1.5.3.5
Postgres	8.0.1
Openssl	1.0.2
FFMPEG	4.2
SQLite	3.26.0
Binutils	2.33.1
Putty	0.73

Table 5.1: Details of Benchmarks Used.

Our approach takes three input parameters namely *min_sup*, *conf_threshold* and *dup_threshold* where *dup_threshold* is an optional parameter and it is used in case when we have multiple binaries that can have same function to remove those rules which have repeated function in their supporting itemsets. For our experimentation we set the value as 15, 90 and 5 for *min_sup*, *conf_threshold* and *dup_threshold* respectively. Furthermore, in compilation step of our approach we set the compiler optimization to "O0" because all other higher optimization can result in the loss of source code elements information in LLVM IR. For comparison of our approach with PR-Miner we only consider top 60 violations (same as [16]). We compare our approach with PR-Miner on the basis of total violations reported, false positive violations and true positive violations.

Benchmark	**PR-Miner**	**Strip-Miner**	**% Decrease**
Git	5682	24	99.5
PostgreSQL	2394	362	84.9
OpenSSL	1944	208	89.3
FFMPEG	8441	42	99.5
SQLite	845	6	99.2
Binutils	9806	15	99.9
Putty	1163	13	98.9

Table 5.2: **Decrease in generated number of bugs:** On running the PR-Miner and Strip-Miner with support threshold of 15 and confidence threshold of 90% we observed that Strip-Miner is able to decrease the number of generated bugs greatly.

5.2 Research Questions

We answer the following research questions through our experimentation.

1. Can including the dependency information and inter-procedural checking decrease the number of generated bugs?

2. Can including the dependency information and inter-procedural checking decrease the number false positives?

3. Can including the dependency information and inter-procedural checking increase the number true positive?

4. Can including the dependency information and inter-procedural checking find new bugs?

Figure 5.1: **Comparison of False Positives:** On comparison of top 60 generated bugs (for benchmarks which have less than 60 bugs generated all of them are used) ranked according to confidence, we observed that the percentage of false positives generated are less in the case of Strip-Miner.

5.2.1 RQ1: Decrease in the Number of Generated Bugs

Our approach is able to decrease the number of generated bugs as compared to PR-Miner for all the benchmarks. As shown in table 5.2, Strip-Miner is able to decrease the bugs by up to 99.9% maximum for Binutils with an average decrease of 95.9%. What does it show is

employing the dependency information and inter-procedural checking can greatly decrease the number of generated bugs that need to checked by a developer to mark as true positives or false positives. Therefore, making use of our approach can greatly increase the efficiency of a developer and provides greater usability as compared to original PR-Miner approach.

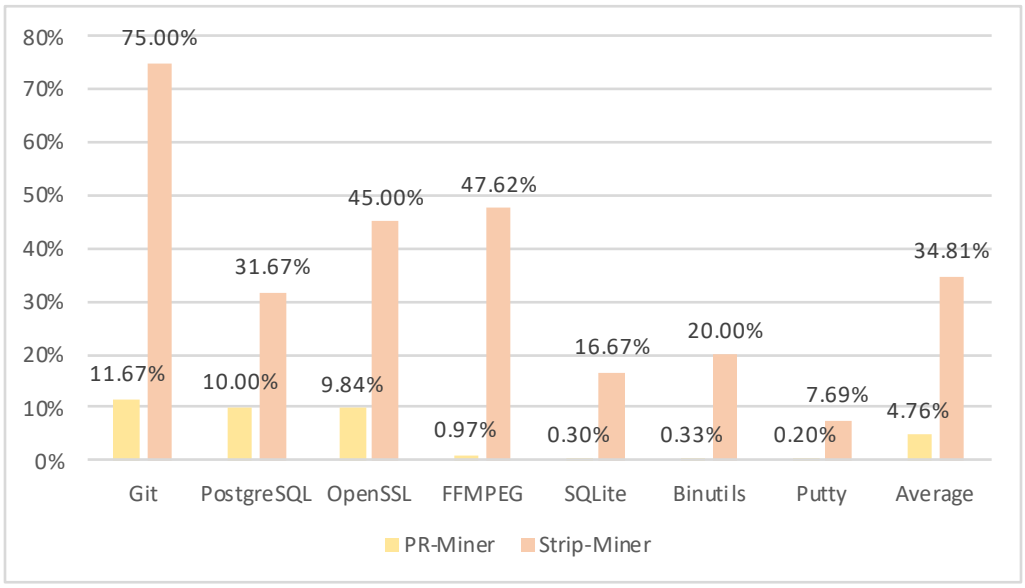

Figure 5.2: **Comparison of True Positives:** On comparison of top 60 generated bugs (for benchmarks which have less than 60 bugs generated all of them are used) ranked according to confidence, we observed that the percentage of true positives generated are greater in the case of Strip-Miner.

5.2.2 RQ2: Decrease in the Number of False Positives Bugs

To answer this research question, we selected the top 60 bugs ranked according to the confidence score and in cases where the total number bugs are less than 60 we use all of the generated bugs. If we look at figure 5.1, Strip-Miner is generating 65.19% false positive bugs on average as compared to PR-Miner which have 95.50% false positives. In the case of Git, our approach has shown the lowest number of false positives with only 25% of the bugs as false positives.

5.2.3 RQ3: Increase in the Number of True Positives

To answer this question, we use the same setup as RQ2. In figure 5.2, we can see that Strip-Miner generates 34.81% of true positive bugs on average whereas PR-Miner only generates 4.76% true positives. Our approach generates a maximum of 75% and PR-Miner generates a maximum of 11.67% true positives both in the case of Git. PR-Miner gets as bad as generating only 0.20% true positive bugs in the case of Putty where Strip-Miner generate 7.69% true positives.

5.2.4 RQ4: Finding New Bugs

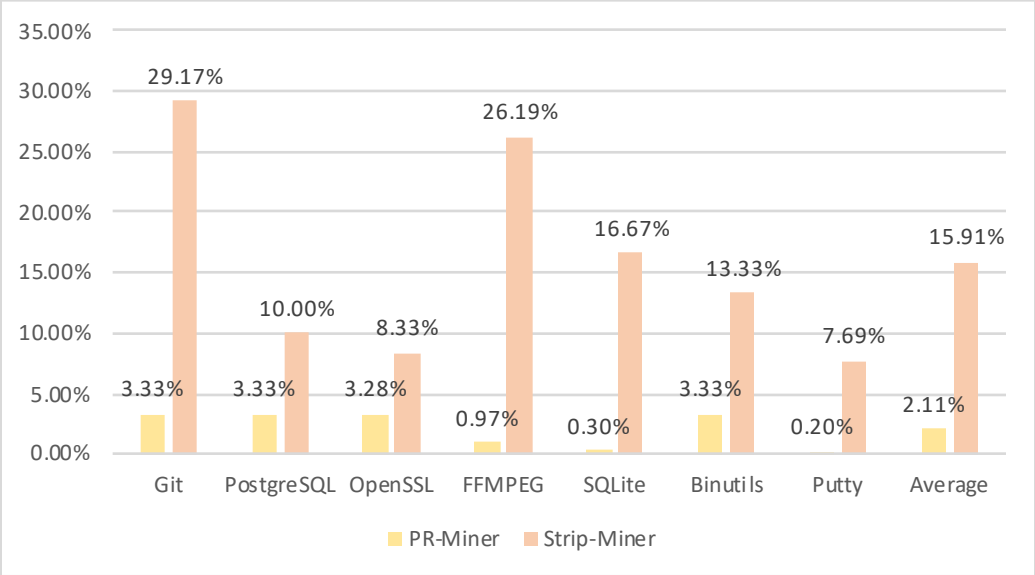

Figure 5.3: **Comparison of Unique True Positives:** On comparison of top 60 generated bugs (for benchmarks which have less than 60 bugs generated all of them are used) ranked according to confidence, we observed that the percentage of unique true positives generated are greater in the case of Strip-Miner.

On analyzing the top 60 generated bugs we observe that the percentage of bugs generated by Strip-Miner that are unique is much greater than PR-Miner. In figure 5.3, we can see that Strip-Miner generates on average 15.91% unique true positives as compared to PR-Miner

that only generates 2.11%. The maximum percentage of unique true positives generated by Strip-Miner is 29.17% whereas the maximum percentage of those generated by PR-Miner is 3.33% both in the case of Git. It is evident that Strip-Miner outperforms PR-Miner for generating previously unseen bugs.

5.2.5 Results Discussion

The results of our experimentation shows that our approach outperforms PR-Miner for both decreasing the false positive rate and increasing the true positive rate. However, we observed one false negative produced by our approach in OpenSSL. The reason for this false negative is our dependency set is not complete. The dependency present between the elements of the false negative is not covered by any of the dependencies. Therefore, there is a need to enhance our dependency set to capture all the dependencies which is our immediate future work.

Chapter 6

Discussion

6.1 Threats to Validity

6.1.1 Internal

The dependencies extracted by our approach do not cover all the relationships among code elements. As a result it is possible that there are some other dependencies present in code, if considered can further decrease the false positive rate. Whereas, in the filtration step it is possible that there are some false negatives because there is no dependency present among the elements present in corresponding rule with respect to the dependency graph. As our approach is fully extensible therefore any dependencies discovered in future can easily be added to dependency set. It is possible that at some point all the possible dependencies are added to the set resulting into removal of all the false positives and false negatives.

6.1.2 External

A possible external threat is the generalization of our results produced in our evaluation. To minimize such threat we have collected the test benchmarks from several different domains spanning across version control systems, DBMS, security libraries and simple utility software like media players. There is a significant improvement in performance of bug detection across

all of these domains which showcase the fact that our approach is generalized enough to work on any software independent of its domain.

6.2 Future Work

Following are several future research directions for our work.

6.2.1 Devising New Dependencies

In our work, we have extracted only three different types of dependencies among code elements and it does not cover all the dependencies that might be present among the elements. Therefore, a future research direction can be to extract other types of dependencies and potentially reducing the false positive rate of bug detection to zero.

6.2.2 Using Dynamic Analysis Traces

In this work, we have extracted program elements and dependencies among them using static analysis. Although, our approach works on software code compiled into LLVM IR but its execution might have different dependencies among elements that can only be extracted from execution traces. Therefore, an immediate future work is to replace static analysis with dynamic analysis to extract programming patterns and dependencies among code elements.

6.2.3 Exploring Other Approaches

Another, interesting future direction can be to explore and do experimentation on other bug detection approaches in an effort to improve their accuracy. We believe that there is

large space for improvement in existing techniques that if filled can enable the production of industrial scale bug detection tools.

6.2.4 Automatic Patch Generation

In order to enhance the usability of our approach an important future work is to automatically generate the patches using detected bugs and their corresponding rules. We believe that the data generated by our approach have enough information available that can be used to localize a bug (from reported bugs) and generate a patch(using the rule that is being violated).

Chapter 7

Conclusions

We present a new approach named Strip-Miner for automatic detection of bugs in code with significant reduction in the number of generated rules and corresponding false positive rate and increase in true positive rate. Our approach leverages simple source code dependencies along with frequent itemsets mining to detect bugs with greater accuracy. We evaluate it using a diverse set of benchmarks collected from different domains. Strip-Miner have high usability in real world software development as compared to older approaches. Experiments show that it is able to reduce the number of generated bugs by up to 99.9%, with a false negative rate of 65.19% and true positive rate of 34.81% on average. In general these experiments show that combining data mining based bug detection approaches with dependency encoding can be useful to improve the usability enormously.

Bibliography

[1] *Software Errors Cost U.S. Economy $59.5 Billion Annually; NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2004 (accessed November 25, 2019).

[2] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267. IEEE, 2000.

[3] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *ACM SIGPLAN Notices*, volume 37, pages 69–82. ACM, 2002.

[4] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[5] Dirk Beyer, Adam J Chlipala, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The blast query language for software verification. In *International Static Analysis Symposium*, pages 2–18. Springer, 2004.

[6] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 2019.

[7] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls imple-

mentations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538. IEEE, 2017.

[8] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 345–362, 2016.

[9] Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Yu Jiang, Ming Gu, and Jiaguang Sun. Vetting api usages in c programs with imchecker. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 91–94. IEEE, 2019.

[10] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, 2002.

[11] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722, 2016.

[12] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349. ACM, 2008.

[13] Michael Pradel. Dynamically inferring, refining, and checking api usage protocols.

[14] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. Statically checking api protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 925–935. IEEE, 2012.

[15] Michael Pradel and Thomas R Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 371–382. IEEE Computer Society, 2009.

[16] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[17] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 752–762, 2017.

[18] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. Nar-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 411–422. ACM, 2018.

[19] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[20] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface invariants. In *Proceedings of the 28th international conference on Software engineering*, pages 861–864. ACM, 2006.

[21] James NewSom. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, network and distributed system security symposium conference proceedings: 2005. In *http://www. isoc. org/isoc/conferences/ndss/05/proceedings/papers/taintcheck. pdf*, 2005.

[22] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 602–613. IEEE, 2016.

[23] Varun Srivastava, Michael D Bond, Kathryn S McKinley, and Vitaly Shmatikov. A security policy oracle: Detecting security holes using multiple api implementations. *ACM SIGPLAN Notices*, 46(6):343–354, 2011.

[24] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 269–278. IEEE, 2006.

[25] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 151–162. ACM, 2017.

[26] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[27] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510. ACM, 2013.

[28] Chris Chao-Chun Cheng, Chen Shi, Neil Zhenqiang Gong, and Yong Guan. Evihunter: Identifying digital evidence in the permanent storage of android devices via static anal-

ysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1338–1350. ACM, 2018.

[29] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In *International Conference on Formal Engineering Methods*, pages 717–736. Springer, 2006.

[30] Yuan Kang, Baishakhi Ray, and Suman Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 472–482, 2016.

[31] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[32] Jean loup Gailly and Mark Adler. *zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library*, 2017 (accessed November 11, 2019).

[33] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, pages 60–71. IEEE Press, 2019.