Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software

Ray-Yaung Chang
EECS Department
Case Western Reserve University
Cleveland, OH 44106
1-216-368-4231
ray-yaung.chang@case.edu

Andy Podgurski
EECS Department
Case Western Reserve University
Cleveland, OH 44106
1-216-368-6884
andy@eecs.case.edu

Jiong Yang
EECS Department
Case Western Reserve University
Cleveland, OH 44106
1-216-368-6197
jiong.yang@case.edu

ABSTRACT
Neglected conditions are an important but difficult-to-find class of software defects. This paper presents a novel approach to revealing neglected conditions that integrates static program analysis and advanced data mining techniques to discover implicit conditional rules in a code base and to discover rule violations that indicate neglected conditions. The approach requires the user to indicate minimal constraints on the context of the rules to be sought, rather than specific rule templates. To permit this generality, rules are modeled as *graph minors* of program dependence graphs, and both frequent itemset mining and frequent subgraph mining algorithms are employed to identify candidate rules. We report the results of an empirical evaluation of the approach in which it was used to discover conditional rules and neglected conditions in ~25,000 lines of source code.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *Statistical method.* D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids*

General Terms
Experimentation, Reliability, Verification

Keywords
Automatic defect detection, mining software repositories, program dependences, frequent itemset mining, frequent subgraph mining

1. INTRODUCTION
Neglected conditions, also known as *missing paths*, *missing conditions*, and *missing cases*, have long been known to be an important and difficult-to-find class of software defects [2][18][31][39]. Well known defect classification schemes such as Orthogonal Defect Classification [21] include a category or subcategory for them. Other research suggests that neglected conditions may be even more important than is generally indicated in the literature. Engler *et al* found hundreds of bugs involving neglected conditions in code for major operating systems such as *Linux* and *OpenBSD* [7]. Raghavan *et al* applied the semantic-graph differencing tool *Dex* to samples of patches to the *GCC* C-compiler and found that 38% of the *Apache* patches and 44% of the *GCC* patches involved inserting conditional selection statements and that 31% of the *Apache* patches and 32% of the *GCC* patches involved altering existing if conditions [34]. In a manual analysis of 167 bugs fixes applied in the *Firefox* web browser project [32], which we conducted during a preliminary investigation of neglected conditions (see Section 2 below), 109 (66%) of the fixes involved correcting neglected conditions. These rather startling numbers suggest that neglected conditions are not just one of several categories of software defects but rather a major cause of reported bugs in software.

The significance of neglected conditions is further magnified by the deployment of software in the hostile environment of the Internet, because attackers searching for vulnerabilities in software routinely explore its ability to handle unusual inputs [29]. *Buffer overflow* vulnerabilities, which have been the most common type of security vulnerability in software [12], result from failure to check whether inputs are too large to fit in an input buffer. Many other types of security vulnerabilities, such as those that enable *SQL injection* attacks and *cross-site scripting*, are due to failures to validate inputs properly [17] and to other types of neglected conditions. For example, a search we did of the National Vulnerability Database [33] with the query "not handle" returned 322 vulnerabilities, many of which are not buffer overflow vulnerabilities but are clearly the result of neglected conditions. It is likely that many of the other vulnerabilities described in the database are also due to neglected conditions.

Many neglected conditions can be prevented by the use of requirements elicitation and analysis techniques that are intended to ensure completeness of a requirements specification, such as *viewpoint analysis* [38]. However, many other neglected conditions are *not* traceable to shortcomings of requirements engineering, because they involve design or implementation issues that do not correspond directly to requirements. A familiar example is failing to check that a pointer or object-reference is non-null before it is dereferenced.

When a program fails to handle conditions that are explicitly mentioned in its requirements specification or design document, it is often straightforward to reveal those defects by either careful code inspection [9] or specification-based testing [36]. However,

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

conditions that are neglected in a program's requirements, design, *and* code are notoriously hard to detect. The essential difficulty in discovering such neglected conditions is the *lack of local clues* about the omissions. Because of this, neglected conditions constitute a significant "blind spot" in the principal approaches to software validation – code inspection, software testing, and model checking.

Code inspection is useful for revealing neglected conditions, because the difference in perspective between someone inspecting code and the programmer who wrote it may permit the inspector to recognize neglected conditions that the programmer is oblivious to. However, empirical studies of code inspection indicate that, although it is clearly beneficial, it still fails to reveal many defects [6][30]. *Boundary value testing* [31] and *special value testing* are software testing techniques that are intended to reveal certain types of neglected conditions. Not surprisingly, they are not particularly effective for revealing neglected conditions that do not involve *recognized* boundary values or special values. *Structural coverage* techniques [31] are useful for revealing a neglected condition only if tests created to satisfy the coverage criterion in use are likely to cause the condition to be true where it should be handled. In principle, *mutation testing* [2] is capable of revealing any neglected condition. However it suffers from a number of practical problems that seriously limit its usefulness, such as the enormous number of mutants that may be generated, the difficulty of creating test data to kill a high proportion of mutants, and the uncertain relationship between simple mutations and more complex defects [46]. Software model checking [43] establishes whether a finite-state model of a program satisfies a set of properties (a specification) for all possible input sequences. It is therefore capable of revealing a condition that is neglected in a program and its model, if this omission causes the model to violate some property in the specification. However, model checking is of no help in revealing violations of properties that are neglected in the specification.

Engler *et al* [7] have shown that many software defects, including certain neglected conditions, can be found by using compiler extensions called *checkers* (written in the *Metal* language [4]) to match *rule templates*, derived from knowledge of typical programming errors, against a code base. The rule templates are sufficiently general to match system-specific programming rules that are not known in advance. An especially simple example of a rule template is "<*a*> must be paired with <*b*>". In cases where there is uncertainty about whether a possible rule must always hold, Engler *et al*'s approach uses statistical evidence, based on how many times the rule holds and how many times it does not hold, to rank candidates for manual inspection. We believe that this idea – that the frequency with which a programming pattern occurs in a code base is a good indicator of how likely the pattern is to represent correct programming practice – is essential to discovering neglected conditions statically. It suggests that dispersed fragments of information about conditions that should be addressed in certain contexts can be exploited to compensate for the lack of local clues about neglected conditions that makes them so difficult to detect with standard validation techniques. The main issue with using Engler *et al*'s approach to discover neglected conditions (or other defects) is the completeness or generality of the set of rule templates. A neglected condition will be discovered only if it violates a rule that matches an existing template. Engler's *et al*'s approach relies on developers to supply rule templates and corresponding checkers. Naturally, developers cannot do so for rules whose essential form they are not aware of.

Li and Zhou presented an approach to finding implicit programming rules and rule violations that is based on frequent itemset mining and does not require specification of rule templates [25]. However, their approach does not consider relevant constraints between rule elements and so apparently will identify a set of elements that frequently appear together in functions as a possible rule without other evidence that the elements are semantically related.

We present a new approach to the detection of neglected conditions in software that builds upon the idea that vital clues about neglected conditions are often distributed throughout a project code base (or even multiple code bases). Our approach is intended to discover a wide variety of programming rules and violations of them without requiring developers to supply specific rule templates or checkers. Instead, developers indicate minimal constraints on the kind of rule violations they wish to find (e.g., any neglected condition, or any neglected condition pertaining to a function call), and candidate rules and possible rule violations are identified automatically. As with Engler *et al*'s approach, candidate rules are ranked statistically and must be confirmed manually by developers. Our approach does consider semantically relevant constraints, in the form of program dependences, between elements of potential rules.

We believe that the keys to discovering neglected conditions automatically given only minimal constraints on their form are to (1) employ a very general representation of programs and of programming rules, encompassing a wide variety of meaningful rules, and (2) exploit appropriate data mining technology to discover rules. Accordingly, in our approach program units are represented by *program dependence graphs* (*PDG*s) [11] (entire programs are represented by *system dependence graphs* (*SDG*s) [15]) and potential rules are represented by *graph minors* [37] of PDGs. Because PDG minors represent transitive (direct and indirect) dependences between program statements, they capture essential constraints between rule elements and exclude spurious ones, permitting a wide variety of meaningful rules to be modeled. In this work, PDG minors are used to model *conditional rules* – rules in which particular conditions are handled properly. Rules are found by mining a database of dependence graphs, using a *frequent subgraph mining* algorithm, to find recurring graph minors, on the assumption that the more a programming pattern is used, the more likely it is to be a valid rule. (Frequent itemset mining is also employed in a preliminary step.) After valid rules are discovered and confirmed, the graph database is searched again, using a heuristic graph matching algorithm, to find rule violations corresponding to neglected conditions. To our knowledge, our approach is the first application of frequent subgraph mining to the problem of revealing neglected conditions in software. We have conducted an empirical evaluation of our approach in which it was used successfully to discover conditional rules and apparent neglected conditions in ~25,000 lines of source code.

The contributions of this paper are as follows:

- A general framework for exploiting clues distributed throughout a code base (or code bases) to identify neglected conditions

- A general representation of programming rules in terms of PDG minors
- An algorithm based on frequent subgraph mining and frequent itemset mining for discovering rules
- A heuristic graph-matching algorithm for finding rule violations
- Initial empirical results characterizing the effectiveness of our approach

The remainder of the paper is organized as follows: Section 2 presents the results of our preliminary study of *Firefox* defects; Section 3 presents background on dependence graphs and data mining; Section 4 presents the specifics of our approach; Section 5 reports on the empirical evaluation of the approach; Section 6 surveys related work; Section 7 summarizes and discusses future work.

2. PRELIMINARY STUDY

To supplement previous work pertaining to the prevalence and form of missing condition defects (see Section 1), we conducted an empirical study of bug fixes in the *Firefox* project [32]. We first retrieved descriptions of all fixed bugs in versions 1.0 and 1.5 of *Firefox* that were reported between July 1, 2005 and March 31, 2006 whose severity level was *block*, *critical*, *major*, or *normal*, and whose status was *resolved*, *verified*, or *closed*. There were 839 such bugs. We manually inspected the descriptions for the first 300 of these bugs and the fixes applied to them, in order to determine how many bugs involved neglected conditions and to further categorize those bugs. Of these 300 bugs, 63 were ignored because their reports did not include enough information, 15 were ignored because they required very complex fixes involving multiple functions, and 55 were ignored because they involved errors in configuration files, HTML files, etc. Among the remaining 167 bugs, we found that 109 (65%) involved one or more neglected conditions. We further classified the 109 bugs involving neglected conditions, as shown in Table 1. (Note that some fixes involved two neglected conditions of different types.)

Table 1: Classification of neglected conditions

Name	Number	Percent
Missing boolean expression (predicate missing)	28	26%
Missing branch ('else' or 'case' branch missing)	13	12%
Missing conditional statement (block needing control present)	5	5%
Both conditional statement and block needing control missing	86	79%

3. BACKGROUND

In this section we provide background on program dependence graphs and data mining that is needed to understand the remainder of the paper.

3.1 Program Dependence Graphs

A *program dependence graph* (*PDG*) [11] is a labeled directed graph that models dependences between the statements of a program or procedure. Two types of dependences are represented: a statement s_1 is *data dependent* on a statement s_2 if there is a variable x and a control flow path $s_2 P s_1$ from s_2 to s_1 such that x is defined at s_2, used at s_1, and not redefined along the subpath P; s_1 is *control dependent* on s_2 if s_2 is a branch predicate that directly controls whether or not s_1 is executed. Because program dependence graphs capture the essential ordering constraints between program elements, programming rules relating elements that are not necessarily adjacent to one another in a program and that do not necessarily appear in the same textual order wherever the rule occurs can be represented as PDG subgraphs or, as we shall see, PDG minors.

The form of program dependence graph we have used in our experiments is the *system dependence graph* (*SDG*) [15] generated by the *CodeSurfer* static analysis tool [14]. An SDG is a collection of *procedure dependence graphs*, one for each procedure, augmented with special edges linking callers and callees. (Henceforward, we will use the abbreviation PDG to denote a procedure dependence graph.) SDG edges can be classified into (1) data dependence and control dependences edges and (2) inter-procedure edges and intra-procedure edges. SDGs contain various types of nodes such as those representing call-sites, statements, control-points, actual-in/out parameters, formal-in/out parameters, switch statements, etc. Expressions associated with PDG nodes are represented by *abstract syntax trees* (*AST*s). Our experiments indicate that the *CodeSurfer* SDG representation is quite adequate for representing a wide variety of rules, including conditional rules.

3.2 Frequent Itemset and Subgraph Mining

Our approach is based on using data mining techniques, including frequent itemset mining and frequent subgraph mining, to discover programming rules. Given a database of itemsets, *frequent itemset mining* algorithms are designed to find frequent sub-itemsets. A sub-itemset (a subset of an itemset) is considered frequent if the number of its occurrences in the database, denoted as its *support*, is greater than or equal to a specified threshold (called the *minimum support*). Our approach uses the *maximal frequent itemset mining* algorithm *MAFIA* [3] to identify nodes with frequent labels from a set of labeled graphs. Instead of generating a complete set of frequent itemsets, *MAFIA* outputs only maximal frequent itemsets. (A frequent itemset is called maximal if it is not a subset of any other frequent itemset.) In this way, *MAFIA* avoids generating an exponential number of sub-itemsets.

Frequent subgraph mining (*FSM*) techniques can be classified into two categories based on the nature of the input dataset. Techniques in the first category find subgraphs that occur frequently across a collection of graphs [19][20][40][44]. Given a graph dataset $\mathcal{G} = \{G_1, G_2, G_3, \ldots, G_n\}$, these techniques find any subgraph H whose *support* is greater than or equal to a given threshold. The support of a subgraph H is the number of graphs in $\mathcal{G}$ of which H is a subgraph. Techniques in the second category, on the other hand, find subgraphs that occur multiple times in a single, typically larger, graph [16][23][24]. Given an input graph G they find any subgraph H whose number of occurrences in G is greater than or equal to a minimum support threshold. Frequent subgraph mining algorithms that produce a complete set of frequent subgraphs, such as *gSpan* [44] and *FFSM* [19], suffer from the problem that the number of frequent subgraphs found may be exponential. Theoretically, a frequent n-edge subgraph may contain 2^n frequent sub-subgraphs, since each subgraph of a

frequent subgraph is also frequent. However, subgraphs that share the same support may be redundant. Thus, *maximal frequent subgraph mining* algorithms such as *SPIN* [20] and *MARGIN* [40] have been proposed to mine only maximal frequent subgraphs – subgraphs that are not a part of any other frequent subgraphs. Because the subgraph isomorphism problem is NP-complete [13], subgraph mining algorithms can deal only with graphs of limited size. To address the scalability problem, FSM algorithms such as *HSIGRPAH*, *VSIGRAPH* [23], *GREW* [24], and *SUBDUE* [16] have been designed to mine graph patterns from a large sparse graph. To the best of our knowledge, all state-of-the-art FSM algorithms are designed to mine frequent subgraphs rather than frequent graph minors. Therefore, special steps must be taken to employ them with our approach.

4. SPECIFICS OF THE APPROACH

In this section we describe the details of our approach, including how conditional rules are represented, mined, and evaluated by users, how violations are found, and limitations.

4.1 Representing Conditional Rules

Our decision to represent programs and conditional programming rules in terms of system dependence graphs is based on our intuition that a wide variety of rules can be described in terms of data and control dependence relationships among a set of SDG nodes with particular attributes (e.g., parameter types and expression ASTs). Control dependences are obviously critical elements of conditional rules, and the absence of an expected control dependence is a prime indicator of a neglected condition. It is important to note, however, that a violation of a conditional rule may involve a missing data dependence instead of a missing control dependence, as when a critical variable is erroneously omitted from a control predicate. Dependence graphs "abstract away" many nonessential constraints on statement ordering that are implicit in other program representations such as abstract syntax trees. Thus rules represented in terms of dependences can match rule instances in a variety of local contexts exhibiting incidental variations in statement ordering. Other important factors in our decision to adopt SDGs were the availability of *CodeSurfer* and the acceptable costs of SDG generation and analysis.

Our initial idea was to represent conditional rules as procedure dependence graph subgraphs and to discover them by applying a frequent subgraph mining algorithm to a graph database consisting of PDGs. This idea required a number of refinements. One such refinement is needed to address a fact we discovered in our preliminary experiments with this approach: *a particular relationship between two statements in a rule can be realized in different rule instances by different chains of direct control and/or data dependences between the statements.* This is illustrated by an example from the *Apache* HTTP server [1]. The following code fragment illustrates a simple conditional rule, which we will call *R*, requiring that before the function apr_file_dup is called, its second parameter is checked to ensure it is not NULL:

```
if (parent_err) {                          // apr_file_t *parent_err
    rv = apr_file_dup(&attr->parent_out, parent_err, attr->pool);
}
```

Figure 1 shows a dependence graph G_1 representing this rule. Consider now the following slightly more complex code fragment, which is an instance of the same rule. A partial dependence graph G_2 for this code is shown in Figure 2.

```
if (parent_err) {                          // apr_file_t *parent_err
    if (attr->parent_err == NULL)          // apr_procattr_t *attr
        rv = apr_file_dup(&attr->parent_err, parent_err, attr->pool);
    else
        rv = apr_file_dup2(attr->parent_err, parent_err, attr->pool);
}
```

Note that G_1 is not isomorphic to a subgraph of G_2. However, in both graphs there is a chain of control dependences from the second actual parameter in a call of the function apr_file_dup to a test of whether parent_err is NULL. In G_1, the chain consists of two direct control dependences; in G_2, the chain consists of three direct control dependences. To enable such patterns to be recognized in SDGs it is necessary to model the transitivity of program dependences. We do so by representing conditional rules as graph minors [37] of SDGs. A graph *M* is called a *minor* of the graph *G* if *M* is isomorphic to a graph that can be obtained by applying zero or more edge contractions to a subgraph of *G*, that is, by replacing certain paths with edges. Figure 3 depicts a minor of graphs G_1 and G_2 corresponding to the rule *R*.

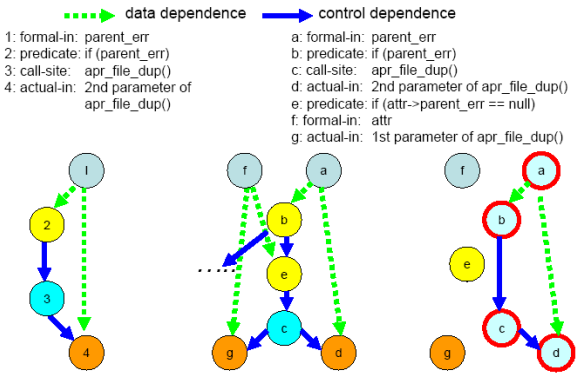

Figure 1: Conditional Rule *R* **Figure 2: Another PDG exhibiting *R*** **Figure 3: Graph minor corresponding to rule *R* consists of the nodes a, b, c, d and the edges shown**

4.2 Mining Conditional Rules

The first step in our approach to discovering conditional rules is to compute a system dependence graph for the code to be analyzed. The next step is to mine the resulting SDG to identify graph minors representing candidate rules. A minor is a candidate if it contains a *control point* (a node representing the branch predicate of a conditional statement), if it occurs at least *t* times in the entire SDG (for a chosen threshold *t*), and if it is maximal in the sense that no proper supergraph of the minor occurs at least *t* times. The key to accomplishing this step is to exploit a frequent subgraph mining algorithm. As mentioned in Section 3.2, several promising algorithms have been proposed recently to find frequent subgraphs in a graph dataset or a single graph [16][19][20][23][24][40][44]. Most of these algorithms work on graphs of very limited size, since subgraph isomorphism is an NP-complete problem [13]. The following issues must be addressed before a FSM algorithm can be used successfully to discover conditional rules from an SDG:

(1) *Dataset size*: The SDG generated for an application with thousands of lines of code may contain hundreds of thousands of

nodes and edges, and even PDGs for individual functions are often larger than can be handled by FSM algorithms that find all frequent subgraphs. Two alternatives are to: (1) use an approximate or heuristic subgraph mining algorithm [23][24], which cannot find all frequent subgraphs, and (2) decompose a graph that is too large to mine into a set of smaller graphs.

(2) *Form of the graph dataset*: It is necessary to choose between FSM algorithms that require input in the form of a single graph, which may be relatively large, and those that require input in the form of a set of smaller graphs.

(3) *Edge direction*: Most FSM algorithms work on undirected graphs. However, SDGs are directed graphs.

(4) *Multigraphs*: To our knowledge, existing FSM algorithms do not work on *multigraphs*, which may have multiple edges between a given pair of nodes. However, SDGs can be multigraphs, because there can be both data and control dependences between a given pair of nodes.

(5) *Irrelevant subgraphs*: For finding programming rules, it is not necessary to consider non-maximal frequent subgraphs. Moreover, for discovering conditional rules, subgraphs that do not contain control points and possibly other nodes of interest are not relevant.

(6) *Node and edge labeling*: FSM algorithms depend on node and edge labels to discover frequent subgraphs. The edges of an SDG are labeled with the type of the dependence (e.g., data or control) they represent. Node labeling is more complex. Nodes in an SDG represent statements or predicates in source code. Seemingly identical statements may have different semantics and different statements may have the same semantics. If nodes with different semantics are assigned the same label, then false positives may result. If nodes with the same semantics are assigned different labels, then false negatives may result.

(7) *Graph minors*: existing FSM algorithms discover only true subgraphs in graph datasets and not graph minors.

(8) *Frequency threshold*: It is necessary to determine how frequent an SDG minor must be to be considered a candidate rule.

We address each of these issues below.

Frequent Subgraph Mining Algorithm: Regarding issues (1) and (2), dataset size and form, we have elected to use a FSM algorithm that finds subgraphs which are frequent across a collection of graphs, because such algorithms in general are more mature. In particular, we employ an algorithm based on *Monkey* [45] to find subgraphs occurring in at least *t* graphs among a collection of graphs, for chosen *t*. The algorithm we use is capable of handling a set of small undirected graphs, each no larger than 254 nodes at present. Regarding issue (3), this algorithm ignores the direction of SDG edges. Our experimental results indicate that this does not strongly affect the discovery of programming rules, apparently because of compensating features of SDGs. For instance, expression nodes are often control dependent on predicate nodes, but predicates nodes are seldom control dependent on expression nodes. Regarding issue (4), the FSM algorithm we employ does not handle multiple edges between the same pair of nodes, so we replace each such edge after the first encountered with a path of length two that connects the edge's original endpoints via a new dummy node. Regarding issue (5), the algorithm avoids generating a complete set of frequent subgraphs by applying some pruning techniques, and only the frequent subgraphs with maximum size are used for further analysis.

Node and Edge Labeling: Regarding issue (6), edge labeling is determined by the type of the edge: data dependence, control dependence, intra-procedure dependence, or inter-procedure dependence. The rules of node labeling are as follows:

- The general rule is that nodes with identical abstract syntax trees (ASTs) are given the same label. However, we ignore variable names in abstract syntax trees and retain the data type of variables. For example, the abstract syntax trees of the following pair of statements are considered to be identical if all of the variables have the same data type:

```
a = b + c;   d = e + f;
```

- A call site node is given the label of the entry node of the callee function. In addition, nodes of actual-in and actual-out parameters are given the label of their corresponding formal-in and formal-out parameters.

- ASTs of some control points are modified to address the fact that certain kinds of predicates are commonly expressed in alternative forms that are semantically equivalent (e.g., `a > b` and `b < a`). In particular, the operators `!=`, `<`, and `<=` are replaced by `==`, `>`, and `>=` , respectively.

Transforming the Data Set: Our strategy for addressing the limitations of the FSM algorithm is to extract procedure dependence graph subgraphs that are small enough to be handled by the algorithm yet collectively are likely to contain most conditional rules of interest. To this end, PDG nodes that are likely to be found in rules of interest, called *candidate nodes*, are identified. Then a *dependence sphere* [35] of limited radius *r* (ignoring edge direction) is generated around each candidate node. Naturally, *control points*, which represent conditional branch predicates, are good candidates to be dependence sphere centers. To focus the search on particular kinds of conditional rules, other types of centers are useful. For example, to find conditional rules involving function calls, *call points* are good candidates. The number of graphs in a data set is limited, if necessary, by randomly sampling candidate nodes with a given label before generating dependence spheres (see Section 5.1).

In preliminary experiments in which we applied our approach to the *Apache* HTTP server [1], we found that dependence spheres of radius 5 had about 150 nodes and 250 edges on average. Although the FSM algorithm we used can handle graphs of this size, the time required for discovering rules in a collection of graphs was unacceptable. We consider it undesirable to reduce the radius further, because some interesting conditional rules may be lost. Thus, it is necessary to prune the dependence spheres more intelligently. We do so in two steps. Since dependence spheres are computed without regard to edge direction, we first remove from a sphere all nodes *n* such that (1) *n* can't be reached from the sphere's center by a directed path and (2) the center can't be reached from *n* by a directed path. We shall call the resulting graphs *partially reduced dependence spheres* (*PRDS*s). We have found that these graphs are still not quite small enough on average to be handled by the frequent subgraph mining algorithm within reasonable time. To further reduce their size, we exploit the fact that if a subgraph *H* occurs in at least *t* graphs of a graph dataset $\mathcal{G}$, then the node set of *H* must be a subset of the node sets of at least *t* graphs of $\mathcal{G}$. In particular, we use the

maximal frequent itemset mining algorithm *MAFIA* [3], which can handle larger data sets than FSM algorithms can, to identify node labels that don't occur in enough node sets to occur in a frequent graph minor. We then remove from each partially reduced dependence sphere all nodes with these labels. We call the resulting graphs *fully reduced dependence spheres* (*FRDS*s).

Further graph transformations are necessary to address issue (7) above, the fact that frequent subgraph mining algorithms search for ordinary subgraphs and not for graph minors. It is not difficult to prove that a directed graph M is a minor of a directed graph G if and only if M is isomorphic to a subgraph of the *transitive closure* of G, which has an edge (m, n) for every pair of distinct nodes m and n such that there is a path from m to n in G. Thus, we can find *frequent minors* of a set $\mathcal{G}$ of fully reduced dependence spheres by finding frequent subgraphs among the set of transitive closures of the elements of $\mathcal{G}$. However, the transitive closure of G has more edges than G, which increases the execution time of the FSM algorithm. To address this, we evaluated several strategies for computing a *near transitive closure* of a fully reduced dependence sphere (see Section 4.1).

Frequency Threshold: Regarding issue (8) above, our initial experiments indicated that our approach is sensitive to the particular frequency thresholds used by the frequent itemset mining algorithm and the frequent subgraph mining algorithm. To address this issue, the thresholds are initially set to high values and if necessary they are iteratively decreased until rules are found or a l is reached. (See Section 5.1.)

Additional Constraints: In some cases, a user may wish to search specifically for conditional rules satisfying certain constraints on nodes or edges, e.g., rules in which a function's return value is checked. When such a constraint involves the presence of a single node with a given type or label, it may be possible to modify the frequent subgraph mining algorithm to discard subgraphs that do not contain such a node, in order to further reduce the computation time. Our current FSM algorithm outputs only frequent subgraphs that contain the candidate node. This modification improves the algorithm's performance significantly.

4.3 User Evaluation of Rules

In this step, a user of our approach employs a tool to review candidate conditional rules identified by frequent subgraph mining to confirm that they are of interest. This tool allows a user to: (1) select candidate rules to examine; (2) display candidate rule instances by highlighting their corresponding source code lines; (3) edit candidate rules to remove irrelevant program elements or dependences (4) automatically merge overlapping rules; (5) select rules the user wants to use in searching for neglected conditions; and (6) indicate one or more *key nodes* of a selected rule R. The set of key nodes for R should be chosen so that violations of R are likely to contain at least one node with the same label as a key node. An example of a key node for a conditional rule involving preconditions or postconditions of a function is a call site for the function. Note that a candidate node that is likely to be missing in violations of a rule (such as a control point in a conditional rule) should not be chosen as a key node.

4.4 Finding Rule Violations

The final step in our approach is finding violations of the conditional rules that were identified by frequent subgraph mining and confirmed (and possibly edited) by the user. Intuitively, a violation of a conditional rule R is a PDG subgraph that is very similar to R but perhaps lacks a control point, control dependence, or data dependence present in R. To find violations of rule R, we first find PDG nodes to which R may pertain and generate a dependence sphere S around each such node. These are nodes with the same label as a user-specified *key node* of R (see Section 4.3). We then employ a heuristic graph matching algorithm to find the graph minor M in S that is most similar to R. If M is similar to but not isomorphic to R then it is reported as a potential violation of R and is examined by the user, with they aid of a tool that highlights its corresponding source code lines, to determine if it is an actual violation.

We now present a simplified description of how our heuristic graph matching algorithm identifies violations. Given a key node k in rule R and node n in sphere S with the same label as k, the algorithm searches for images in S of the remaining nodes in R, as follows.

Initialization: The algorithm computes transitive closures of R and S, denoted by $C(R)$ and $C(S)$.[1] (Each edge that is included in one of these closures to represent a multi-edge path in R or S is labeled "data", "control", or "mixed" according to whether the path contains only data dependence edges, only control dependence edges, or a mixture of the two, respectively.) The algorithm records the fact that n is the image of k, and it initializes a priority queue Q to contain k. Q contains the nodes in R for which corresponding images in S have been found, and it is ordered by distance from k. A set P of processed nodes from R is initialized to contain k. Note that some nodes placed in P may not have images in S.

Traverse nodes in rule *R* and find their images in *S*: This part of the algorithm continues while Q is not empty. On each iteration, the first node s of Q is removed. Let s' be the image of s in S. For each node $t \notin P$ that is adjacent from s in R, a heuristic method is used to try to find an image t' in S that matches t well. If such an image is found, it is recorded and t is added to Q. In any case s is added to P. To find an image for t, the algorithm computes a set $I(t)$ of potential images for t, consisting of all nodes u in S that are adjacent from s' in $C(S)$, have the same label as t, and are not images of nodes in Q and for which (s', u) has the same label as (s, t). The algorithm determines, for each $u \in I(t)$ how many edges incident to or from u in $C(S)$ match edges incident to or from t in $C(R)$. A node u with a maximal number of matching edges is chosen as the image t' of t, provided this number exceeds a chosen threshold. Node t is added to Q and node s is added to P.

Identifying rule violations: The nodes of the minor M are the computed images of the nodes in R that have such images, and the edges of M are the edges of S incident with these nodes. The absence in M of an image for a node in R is treated as an indication that M represents a violation of R, as is the absence in

[1] These are actually not conventional transitive closures. They are computed using a method similar to the CMTC method described in Section 5.1.

M of a dependence that is present in *R*. Similarity between *R* and *M* is measured in terms of the number nodes and/or dependences present in *R* that are not matched in *M*.[2]

4.5 Limitations

Our approach is unlikely to reveal neglected conditions that are not violations of conditional rules that recur in the code base. It is also unlikely to identify a conditional rule whose instances don't give rise to the same dependence graph minor, e.g., because they are structurally dissimilar or because they involve constraints that are not reflected in a dependence graph, such as timing constraints.

5. EMPIRICAL EVALUTION

We have conducted an initial empirical evaluation of our approach, in which it was applied to find conditional rules and rule violations in code from the *Apache* HTTP Server Project [1] and the results were checked manually. The evaluation focused on four principal questions:

- Is the approach able to discover a high proportion of conditional rules in a code base automatically?
- Are the discovered rules are of interest?
- Does the algorithm for detecting rule violations effectively distinguish between rule instances and non-instances?
- Do the rule violations reported by that algorithm actually involve neglected conditions?

A secondary goal of the study was to evaluate the effectiveness of three alternative methods for computing "near" transitive closures of fully reduced dependence spheres (see Section 5.1).

5.1 Experimental Methodology

Nine subprojects of the *Apache* HTTP Server Project (version 2.2.2) were analyzed [1]. These comprised about 400 files and more than 25,000 lines of code. The resulting system dependence graph generated by *CodeSurfer* contained about 500,000 nodes. The version of *CodeSurfer* we employed (version 1.9, patch level 7) could not generate an SDG when additional subprojects were included. Our implementation is written mainly in Java. The evaluation focused on discovery of rules concerning preconditions and postconditions of function calls. This was done because we have observed that a high proportion of neglected conditions in various projects involve violations of such rules and because it was feasible for us to manually inspect the vicinity of call sites to determine if our approach failed to discover conditional rule instances or rule violations. (This took about 1.5 weeks of very intensive work.)

The experimental methodology is detailed in items (1)-(7) below:

(1) We used the *Scheme* API provided by *CodeSurfer* to extract information from the SDG and we imported it into a database for further analysis. Nodes and edges were labeled based on the rules described in Section 4.2.

(2) Only call-site and actual-in/out parameter nodes of functions that were called in at least two different functions were chosen as *candidate nodes*. There were 443 such functions, which we shall call the *subject functions*. In order to eliminate overlapping dependence spheres and thereby reduce the overhead of graph mining, at most one candidate node with a given label *L* was selected per PDG. To further reduce the overhead of FSM, at most ten such nodes in total were selected for *L*. If more than ten PDGs contained *L*, ten of them were chosen at random and a candidate node with label *L* was selected from each of them. Dependence spheres of radius five (containing only intraprocedural edges) were generated around each selected candidate node. Thus for each candidate node label *L*, a set of at most ten dependence spheres, each centered on a node with label *L*, was generated, and this set contained at most one dependence sphere for each PDG containing a candidate node with label *L*.

(3) To generate partially reduced dependence spheres, nodes that were not connected with candidate nodes via certain kinds of paths were removed from dependence spheres. In particular, with call-site and actual-in candidate nodes, we were interested only in the *preconditions* under which functions were called. Hence a node *n* was removed from a sphere if there was no directed path from *n* to the sphere center (candidate node). On the other hand, with actual-out parameters of function calls we were concerned only with validation of call *post-conditions*. Hence, a node *n* was removed from a sphere if it was not reachable from the sphere center via a directed path. Removing such nodes reduced the average size of dependence spheres significantly. The price is that information about dependences between the removed nodes and the remaining nodes is lost.[3] To further reduce their size – producing fully reduced dependence spheres – the maximal frequent itemset mining algorithm *MAFIA* was used to identify and remove nodes with infrequent labels. The frequency threshold for MAFIA was set initially to 80% of the size of a graph dataset and was iteratively decreased, if necessary, until rules were discovered or a lower bound of 50% was reached.

(4) As mentioned in Section 4.2, the performance of our frequent subgraph mining algorithm degrades if transitive closures of fully reduced dependence spheres are computed, because of the increase in edges. To alleviate this effect, three methods were evaluated for computing a *near transitive closure* of an FRDS prior to rule mining (each method adds edges to the FRDS):

- *Simple pure transitive closure* (SPTC): For each pair of distinct nodes *u* and *v* in the FRDS, the original (unreduced) dependence sphere is searched for a directed path from *u* to *v* that includes no other nodes of the FRDS and contains only data dependence edges or only control dependence edges. If such a path is found, a directed edge (*u*, *v*) labeled "data" or "control" is added to the SPTC.

- *Simple mixed transitive closure* (SMTC): An SPTC is first computed for the FRDS, and for those pairs of distinct nodes *u* and *v* that are not connected by a directed edge (*u*, *v*) in the SPTC, the original dependence sphere is searched for a directed *u*-*v* path that contains no other nodes in the FRDS and includes *both* data and control dependence edges. If such a path is found, a directed edge (*u*, *v*) labeled "mixed" is added to the SMTC.

- *Complex mixed transitive closure* (CMTC): The CMTC method first computes the transitive closures of the control

[2] Missing dependences in *M* are actually found by comparing the transitive closures of *R* and *M*.

[3] If the lost dependence information is relevant to a programming rule, it can still be discovered when a dependence sphere is generated for another candidate node.

dependence subgraph and the data dependence subgraph of the original dependence sphere. For each pair of distinct nodes *u* and *v* in the FRDS, the data dependence subgraph is searched for a *u*-*v* path. If such a path is found, a directed edge (*u*, *v*) labeled "data" is added to the CMTC. The control dependence subgraph is also searched for a *u*-*v* path. If one is found, a directed edge (*u*,*v*) labeled "control" is added to the CMTC. If neither a data dependence path nor a control dependence path from *u* to *v* is found, a "mixed" path is sought as in the SMTC method.

With each of these methods rule mining required roughly one day of computation time.

(5) The frequent subgraph mining algorithm mentioned in Section 4.2 was applied to the near transitive closures described in (4) to discover conditional rules. The frequency threshold for the FMS algorithm was initialized to the number of fully reduced dependence spheres and was decremented up to two times if this was necessary to find rules.

(6) We highlighted and examined the source code of all discovered rule instances to determine whether the corresponding rules were actually of interest. If necessary, rules were edited to remove useless program elements (their corresponding dependences are removed automatically). Some overlapping rules were merged automatically. The process of examining and editing mined rules took about three days in total. In addition, all of the call sites for each function called at most 50 times and a random sample of 50 call sites for each other function were examined manually to try to find all actual conditional rules involving a function's preconditions or postconditions (whether or not they were discovered automatically) as well as to find actual rule violations. This process took about 1.5 weeks.

(7) The graph matching algorithm presented in section 3.4 was evaluated in two ways. First, for each valid mined rule *R*, the matching algorithm was used to search for *exact matches* to *R* in the set $\mathcal{G}$ of dependence spheres from which *R* was mined. The key nodes used in the search for *R* were the candidate nodes used to generate $\mathcal{G}$. The second way in which the matching algorithm was evaluated was by using it to search for *violations* of each valid mined rule *R* among the set $\mathcal{G}$ of dependence spheres from which *R* was mined. Again, the key nodes used in the search were the candidate nodes used to generate $\mathcal{G}$. Reported violations were examined manually to determine whether they appeared to be actual neglected conditions. (Because we were not intimately familiar with the *Apache* HTTP server implementation, we could not be certain that apparent conditional rules were always applicable, and hence we could not be certain that apparent neglected conditions were actual ones.) This examination took less than a day.

5.2 Effectiveness of Rule Mining

The experimental results concerning the effectiveness of conditional rule mining are summarized in Table 2. A total of 120 apparent conditional rules were discovered by manual inspection of the 443 subject functions. Automated rule mining produced 138 candidate rules when the SPTC method of computing "near" transitive closures was employed. After these candidates were examined, 28 (20%) of them were judged to be invalid (false positives) and discarded, because they did not appear to correspond to semantically meaningful conditional rules. Thus 110 (80%) of the mined rules were judged to be valid. These were a subset of the 120 rules found manually, comprising 92% of those rules. One major reason that invalid rule candidates were mined is that some statements or predicates with different semantics were given the same label. For example, predicates of the form p == null, where p has a certain type, were given the same label because their abstract syntax trees were identical (recall that variable names were ignored). Investigation of the 10 rules that were found manually but not found automatically revealed that eight of them were due to semantically equivalent statements receiving different labels. The other two missed rules occurred because a critical information flow was propagated transitively via a "mixed" path involving both data and control dependences, which was ignored by the SPTC method. Of the 110 valid rules, 42 required no changes (column (1) in Table 2), 52 required minor changes (column (2)), and 16 required significant changes (column (3)). A change was considered minor if it involved at most two program elements or dependences.

Table 2: Results of automated rule discovery

Approach	Discovered rules				Missed rules	
	Valid rules			Invalid rules	Node labeling	Mixed path
	(1)	(2)	(3)			
SPTC	138				10	
	110			28	8	2
	42	52	16			
SMTC	145				8	
	112			33	8	0
	34	47	31			
CMTC	146				8	
	112			34	8	0
	27	44	41			

Rule mining yielded 112 valid rules with both the SMTC and CMTC methods. These included the 110 rules discovered with the SPTC method plus two rules involving mixed dependence paths. However, there were five more invalid rules that both of these methods produced, due to mixed paths involving inconsistent sequences of data and control dependences. An additional invalid rule, containing an irrelevant indirect control dependence, was produced with the CMTC method. Furthermore, 31 and 41 valid rules discovered with the SMTC and CMTC methods, respectively, required significant modifications.

The above results suggest that rule mining with the SPTC method is sufficient to discover the preponderance of actual conditional rules of the form considered and that a large majority of the rules it discovers are valid. The results also suggest that some additional valid rules can be discovered by employing the SMTC or CMTC methods to address chains of "mixed" dependences, at the cost of additional false positives.

5.3 Effectiveness of Violation Detection

Table 3 summarizes the results of using our heuristic graph matching algorithm to search for each valid mined rule instance *R* in the set of dependence spheres from which *R* was mined. Each search of a sphere *S* for *R* was considered a success if *R* was present in *S* and the search found it or if *R* was not present in *S*

and the search returned no exact match. In all, 595 dependence spheres were searched, and all the searches were successful.

Table 3: Results of using matching algorithm to search for known rules

Dependence spheres	Success	Fail	Precision
595	595	0	100%

Table 4 summarizes the results of using the matching algorithm to search for violations of each valid mined rule instance *R* in the set of dependence spheres from which *R* was mined. The algorithm reported a total of 80 rule violations. After examining the reported violations, we determined that eight of them appeared to be actual neglected conditions. The majority of the false positives (54%) were caused by semantically equivalent statements being given different labels. Our inspection of the source code did not reveal any actual conditional rule violations that were missed by the matching algorithm. However, some rule violations might be missed by our approach when statements with different semantics are given the same label.

Table 4: Rule violations

Rules used to detect violations	112
Reported rule violations	80
Potential neglected conditions	8

The experimental results suggest that the graph matching algorithm used for detecting rule violations accurately distinguishes between instances and non-instances of valid conditional rules of the form we considered and that it detects actual violations of those rules.

5.4 Examples

We present two examples to illustrate our results. The first is a rule that describes the conditions under which the function getpage() is called. It is the result of automatically merging three simpler rules mined using as candidate nodes each actual-in parameter of getpage() and a call site. The rule occurs three times in the source code analyzed. A sample instance of the rule is:

```
APU_DECLARE(apr_status_t) apr_sdbm_fetch(apr_sdbm_t *db,
        …, apr_sdbm_datum_t key) {
    if (db == NULL || bad(key)) return APR_EINVAL;
    …
    if ((status = apr_sdbm_lock(db, … )) != APR_SUCCESS)
        return …;
    if ((status = getpage(db, (exhash(key))) == APR_SUCCESS)
        { … }
    (void) apr_sdbm_unlock(db);
```

The rule requires calling apr_sdbm_lock() before getpage() is called, to acquire an sdbm lock,[4] with the predicate that checks the return value of apr_sdbm_lock() controlling the execution of getpage(). Before apr_sdbm_lock() is called, the formal-in parameter db of apr_sdbm_fetch(), on which the first actual-in parameter of getpage() is data dependent, must be checked to ensure it is not NULL, with the predicate that makes this check controlling the execution of apr_sdbm_lock(). The second actual-in parameter of getpage() is the actual-out parameter of exhash(). The execution of exhash() is controlled by bad(), which is data dependent on formal-in parameter key. A dependence graph representing this rule is shown in Figure 4. The rule involves more than 10 program elements including formal-in parameters, actual-in/out parameters, predicates, call-sites, and expressions.

As a simple example of violation detection, our approach discovered a rule requiring a check that the OS character set is Unicode before apr_conv_utf8_to_ucs2() is called. Using a call-site for apr_conv_utf8_to_ucs2() as a key node, our violation detection algorithm revealed that two call-sites failed to observe the rule and that eight other call-sites did observe it. Interestingly, our approach finds this rule even when the check is written as a macro.

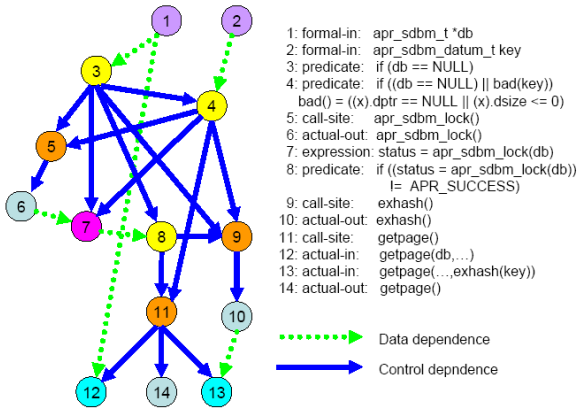

Figure 4: A dependence graph of a rule

5.5 Threats to Validity

The greatest threat to the validity of the study results is, we believe, the fact that the study was limited to a subset of just one project code base. A related issue is the fact that because there were few apparent violations of rules (of the form we considered) present in this code, the violation detection algorithm was exercised less thoroughly than the rule discovery mechanism was. Other threats include the possibilities that: (1) actual conditional rules were overlooked during inspection, (2) mined rules were erroneously classified as valid or invalid, (3) actual violations were overlooked during inspection, and (4) reported violations were misclassified as accurate or spurious. Some mined rules were so subtle that they could easily be overlooked during manual inspection – especially ones involving program elements separated by more than about 15 lines of source code. It is possible that subtle rules were overlooked during both rule mining and manual inspection. Moreover, at most 50 call sites per function were inspected manually.

6. RELATED WORK

Di Fatta *et al* present an approach to fault localization that employs a frequent subtree mining algorithm to extract frequent subtrees from databases consisting of function call trees of successful runs and failing runs, respectively [10]. This information is used to rank functions according to their likelihood of containing a fault. Krinke presented an approach to identifying

[4] Note that another rule discovered by our approach requires using apr_sdbm_lock() and apr_sdbm_unlock() together.

duplicated code in programs by finding maximal similar subgraphs in fine-grained dependence graphs [22].

Li *et al* developed a tool called *PR-Miner*, based on frequent itemset mining, for automatically detecting programming rules and related bugs in large-scale software [25]. *PR-Miner* transforms a function definition into an itemset by hashing program elements to numbers. In this conversion process, similar program elements are mapped to the same number, which is accomplished by treating identifiers with the same data types as identical elements, regardless of their actual names. A frequent itemset mining algorithm called *FPclose* is applied to the itemset database to find programming rules corresponding to sub-itemsets that appear frequently. For efficiency, *PR-Miner* generates only closed rules from a mined pattern. Then, rule violations are detected and ranked by the confidence level of the rules. Because *PR-Miner* does not consider dependences between rule elements, it apparently will identify a set of elements that frequently appear together in functions as a possible rule even if the elements are not related by data flow or control flow. This issue is not discussed in reference [25], and it is not clear from the empirical results described there how frequently false positive violations occur because of it. Li *et al* developed another tool called *CP-Miner*, based on *frequent subsequence mining*, for detecting copy-paste code and related bugs in large scale software [26]. *CP-Miner* transforms a basic block into a sequence of numbers by mapping a statement to a number. The *ColSpan* algorithm is applied to the resulting sequence database to find basic copy-pasted segments. By identifying abnormal mapping of identifiers among copy-paste segments, *CP-Miner* detects copy-paste related bugs, especially those bugs caused by the fact that the programmer forgot to modify identifiers consistently after copy-pasting.

Liu *et al* proposed an approach to detecting software plagiarism by identifying subgraph isomorphism between PDGs from the original program and the plagiarized program, respectively [27]. Livshits *et al* developed a tool called *DynaMine* that combines revision history mining and dynamic analysis techniques to automatically discover coding rules involving correlated method calls and related errors in an application [27]. Wilander *et al* have proposed dependence graphs decorated with type-cast and range information as a general modeling formalism for visualizing security properties and performing static analysis of C code to check several security properties at once [41]. They developed a prototype tool called *GraphMatch* for checking code for integer-input validation flaws. Both correct and incorrect patterns for integer input validation are manually constructed and represented as decorated dependence graphs. Our characterization of programming rules builds upon Wilander *et al*'s representation of security properties.

Williams and Hollingsworth describe a method for using the source code change history of a software project to drive and help to refine the search for bugs [42]. The bug database and source code repository for a project are first manually inspected to determine the types of bugs that have been fixed. Then a bug detector is built. They describe a bug detector that is a function return value checker, which determines if a function's return value is tested before being used. The bug detector uses information about function calls and code changes that was mined from the source code repository to refine the rankings of the warnings it produces.

7. CONCLUSION

The empirical results presented in Section 5 suggest that our approach to rule discovery is feasible and effective for discovering conditional rules involving preconditions and postconditions of function calls and for discovering violations of those rules. The results do not address the approach's effectiveness for discovering other kinds of rules and violations. The experiments also indicated that the frequent subgraph mining algorithm is the performance bottleneck in our approach and that it is critical to control several factors strictly, such as the number of graphs in the graph dataset, the size of graphs, the number of frequent subgraphs, and the size of frequent subgraphs. They also indicate that the effectiveness of the approach is also strongly affected by the node labeling scheme. To address the performance issue, we are currently developing a specialized graph mining algorithm which considers the regular structure of PDGs.

Because the experiments involved only one project and only a moderate amount of code, the results are not conclusive. Hence, we plan to conduct further empirical studies of the approach's effectiveness, using a variety of other code bases. We expect that planned enhancements to the *CodeSurfer* dependence analyzer will make it easier for us to mine large amounts of code, although the most costly aspect of such experiments is manual inspection of code and discovered rules and violations. We also plan to broaden the range of targeted rules, e.g., by choosing other types of candidate nodes such as control-points, expressions, and formal-in parameters. To address node labeling issues, we plan to investigate (1) use of type inference to distinguish semantically distinct expressions having the same ASTs and (2) clustering of semantically similar ASTs.

8. ACKNOWLEDGMENTS

We are grateful to Gramma Technology, Inc. for providing us with *CodeSurfer* and for their technical support.

9. REFERENCES

[1] Apache.org. Apache HTTP Server Project. www.apache.org.

[2] Budd, T.A., DeMillo, R.A., Lipton, R.J., and Sayward, F.G. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. *ACM Symp. on Principles of Prog. Lang.*, 7, pp. 220-233, 1980.

[3] Burdick, D., Calimlim, M. and Gehrke, J. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases, *17th Intl. Conf. on Data Eng. (ICDE '01)*, 2001.

[4] Chelf, B., Engler, D., and Hallem, S. How to write system-specific, static checkers in metal. *2002 ACM Workshop on Program Analysis for Software Tools and Engineering* (Charleston, SC, Nov. 18-19, 2002), 51-60.

[5] Chockler, H., Kupferman, O., and Vardi, M. Coverage metrics for formal verification. *Lecture Notes in Computer Science* 2860, Springer, 2003, pp. 111-125.

[6] Dunsmore, A., Roper, M., and Wood, M. Practical code inspection techniques for object-oriented systems: an experimental comparison. IEEE *Softw* 20, 4 (July 03), 21-29.

[7] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: a general approach to inferring

errors in systems code. *18th ACM Symp. on Operating Systems Principles* (Banff, Canada, Oct. 2001), 57-72.

[8] Engler, D. Meta-level compilation. metacomp.stanford.edu.

[9] Fagan, M. E. Design and code inspections to reduce errors in program development. *IBM Syst. J.* 15, 3, 1976, 258-287.

[10] Fatta, G., Leue, S., and Stegantova, E. Discriminative pattern mining in software fault detection. 3rd Intl. Wkshp. on Software Quality Assurance (Portland, OR, Nov. 2006).

[11] Ferrante, J., Ottenstein, K.J., and Warren, J. D.. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.* 9, 3 (July 1987), 319-349.

[12] Festa, P. Study says buffer overflow is most common security bug. C|Net News.com. news.com.com/2100-1001-233483.html.

[13] Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979

[14] Grammatech. CodeSurfer. www.grammatech.com/products/codesurfer/overview.html.

[15] Grammatech. Dependence graphs and program slicing. www.grammatech.com/research/slicing/slicingWhitepaper

[16] Holder, L.B., Cook, D.J., and Djoko, S. Substructure discovery in the SUBDUE system. *AAAI Workshop on Knowledge Discovery in Databases*, 1994, 169-180.

[17] Howard, M. and LeBlanc, D. *Writing Secure Code*, 2nd edition. Microsoft Press, 2003.

[18] Howden, W. Reliability of the path analysis testing strategy. *IEEE Trans. on Softw. Eng., SE-2,* Sept. 1976, pp. 208-215.

[19] Huan, J., Wang, W., and Prins, J. Efficient mining of frequent subgraphs in the presence of isomorphism. *3rd IEEE Int. Conf. on Data Mining* (Piscataway, NJ, 2003), 549-552.

[20] Huan, J., Wang, W., Prins, J., and Yang, J.. SPIN: mining maximal frequent subgraphs from graph database. *KDD 2004* (2004, Seattle, WA).

[21] IBM. Orthogonal defect classification. Center for Software Eng., www.research.ibm.com/softeng/ODC/ODC.HTM.

[22] Krinke, J. Identifying similar code with program dependence graphs. 8th Working Conf. on Reverse Engineering (Stuttgart, Germany, 2001).

[23] Kuramochi, M. and Karypis, G. *Finding frequent patterns in a large sparse graph.* Technical Report #03-038, Dept. of Computer Sci. and Eng., Univ. of Minnesota.

[24] Kuramochi, M. and Karypis, G. GREW: *A Scalable Frequent Subgraph Discovery Algorithm. Technical Report* #TR 04-024, CSE Dept., Univ. of Minnesota.

[25] Li, Z. and Chou, Y. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. *ESEC-FSE '05* (Lisbon, Portugal, Sept. 05 - 09, 2005).

[26] Li, Z., Lu, S., and Myagmar, S. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng. 32, 3* (Mar. 2006), 176-192.

[27] Liu, C., Yan X., and Han, J: GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. *KDD '06* (Aug. 20 - 23, 2006, Philadelphia, PA, USA.).

[28] Livshits, B. and Zimmermann, T. DynaMine: finding common error patterns by mining software revision histories. *ESEC-FSE '05* (Lisbon, Portugal, Sept. 2005), 296-305.

[29] McClure, S., Scambray, J., and Kurtz, G. *Hacking Exposed: Network Security Secrets and Solutions*, 5th edition, McGraw Hill, 2005.

[30] Meyers, G. J. A controlled experiment in program testing and code walkthroughs/inspections. CACM 21, 9 (Sept. 1978), 760-768

[31] Meyers, G.J. *The Art of Software Testing.* Wiley, 1979.

[32] Mozilla.org. Bugzilla. https://bugzilla.mozilla.org/.

[33] NIST. National Vulnerability Database. http://nvd.nist.gov/.

[34] Raghavan, S., Rohana, R., Leon, D., Podgurski, A., and Augustine, V. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. *20th IEEE Intl. Conf. on Software Maintenance* (Sept. 2004), 188-197.

[35] Renieres, M. and Reiss, S.P. Fault localization with nearest neighbor queries. *18th International Conference on Automated Software Engineering* (Oct. 2003), pp. 30- 39.

[36] Richardson, D.J. and Clarke, L.A. Partition analysis: a method combining testing and verification. *IEEE Trans. on Softw. Eng.*, SE-11 ,12 (Dec. 1985), 1477-1490.

[37] Robertson, N. and Seymour, P.D. Graph Minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B 35 (1)*, 1983, 39-61.

[38] Sommerville, L., Sawyer, P., and Viller, S.. Viewpoints for requirements elicitation: a practical approach. *3rd IEEE Intl. Conf. on Requirements Eng.*, April 1998.

[39] Thayer, T.A., Lipow, M., and Nelson, E.C. *Software reliability study. TRW-SS-76-03*, Redondo Beach, California, TRW, March 1976.

[40] Thomas, L., Valluri, S., and Karlapalem, K. MARGIN: Maximal frequent subgraph mining. *6th IEEE Intl. Conf. on Data Mining* (2006).

[41] Wilander, J. and Fak, P. Rule Matching Security Properties of Code using Dependence Graphs. *1st Intl. Workshop on Code Based Software Security Assessments* (Pittsburgh, PA, November 2005).

[42] Williams C. C. and Hollingsworth, J. K. Automatic mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Trans. of Soft. Eng.* 31 (6), June 2005, 466-480.

[43] Wing, J. M. and Vaziri-Farahani, M. 1995. Model checking software systems: a case study. *3rd ACM Symp. on Found. of Softw. Eng.* (Washington, D.C., Oct. 1995), 128-139.

[44] Yan, X. and Han, J.. gSpan: Graph-Based Substructure Rule Mining. *IEEE Intl. Conf. on Data Mining*, Maebashi City, Japan (2002) 721-723.

[45] Zhang, S., Yang, J., and Cheedella, V. Monkey: Approximate Graph Mining Based on Spanning Trees, to appear in ICDE, April 2007, Istanbul, Turkey.

[46] Zhu, H., Hall, P.A., and May, J.H. Software unit test coverage and adequacy. *ACM Comput. Surv. 29, 4* (Dec. 1997), 366-427.