TC:Small: Countering Web-Application Attacks via Testing User-Input Validation

Web applications have become the core business components for many companies due to rapid migration towards web-based solutions; therefore, the security of web applications strongly impacts the benefit of the companies and the society. Unfortunately, sophistication and flexibility of web-application development make it easy to leave security holes in web applications for attackers. To defend against these attacks with malicious inputs, a common methodology adopted by web applications is to use validators that rigorously verify user inputs, hereby referred as User-Input Validation (UIV), before allowing them to be further processed. UIV serves as a primary defense in the web application against such attacks by classifying user inputs as *valid* or *invalid*, accepting *valid* inputs, and rejecting *invalid* inputs such as malicious inputs. Due to their importance, a defect-free UIV has become a key means of enhancing a web application's security. However, in practice, developers rarely produce defect-free UIV.

In our proposed research, we address these challenges of UIV testing when white-box or black-box testing is applied, and propose a novel grey-box testing approach that combines white-box and black-box testing in a practical and unique way for testing UIV, specifically tailored for security testing. In our proposed approaches for UIV testing, we will provide effective tool support for both test-input generation and test-result assessment. These approaches cover a compressive spectrum of testing types, ranging from black-box testing, grey-box testing, to white-box testing. To conduct effective UIV testing, these approaches creatively exploit a comprehensive sources of information including regular expressions (for characterizing valid user inputs and attack patterns) and other implementations of the same validator type as the validator under test.

This project has four main objectives: (1) to address the issue of lacking specifications in UIV black-box testing by automatically inferring specifications with multiple-implementation testing, (2) to improve white-box testing of UIV by addressing the issues related to handling complex string operations that often exist in UIV implementations, (3) to improve black-box testing of UIV by using inferred specifications that can help generate *valid* and *invalid* test inputs, and (4) to address respective weaknesses of black-box or white-box testing with an effective grey-box testing approach.

Intellectual Merit. The proposed research will fundamentally advance knowledge and understanding in foundations, methodologies, techniques, and tools in the interdisciplinary field of software security testing. The proposed research explores new approaches with novel applications of regular-expression-based testing, execution-path-based testing, and multiple-implementation testing for UIV testing, as well as addressing unique requirements and challenges posed by different UIV testing types including white-box, black-box, and grey-box testing. The PI is well qualified for the proposed research with significant prior experience in software engineering as well as applying software engineering knowledge on addressing security problems. The PI's industrial relationships will allow the PI to evaluate the proposed research in real industrial development environments besides student development environments.

Broader Impact. The proposed research will gain understanding about UIV testing of Web applications, which should lead to higher security assurance of Web applications. The validation efforts involve students and professionals, promoting teaching, training, and learning of assuring high software security. The activity broadens the participation of underrepresented groups because the participating students will include female and minority students. The research will enhance the infrastructure for teaching and research by providing open source tools and data sets. The benefits of this proposed activity to society are two-fold. First, our work should lead to good practices for UIV testing of web applications in improving web-application quality and security in general. Second, innovations in new testing algorithms and tools tend to propagate quickly across application or task domains where UIVs are used. In the proposed educational activities, the PI will develop learning modules and tools in security testing for students and educators.

Key Words: user-input validation; application security testing; test generation; test oracles.

PROJECT DESCRIPTION

1 Introduction

Web applications have become the core business components for many companies due to rapid migration towards web-based solutions; therefore, the security of web applications strongly impacts the benefit of the companies and the society. Unfortunately, sophistication and flexibility of web-application development make it easy to leave security holes in web applications for attackers. Based on the Symantec Internet Security Threat Report [1], in the second half of 2007, 58% of vulnerabilities in software were related to web applications. Existing technologies such as anti-virus applications and network firewalls offer comparatively secure protection at the host and network levels, but not at the application level [2]. Such technologies can be easily compromised with attacks in the form of malicious inputs sent by the attackers. These attacks may be hard to be detected at host or network levels and have to be handled at the application level. Such attacks are becoming more prominent as these malicious inputs can come from any online user — even authenticated users [3].

To defend against these attacks with malicious inputs, a common methodology adopted by web applications is to use validators that rigorously verify user inputs, hereby referred as User-Input Validation (UIV), before allowing them to be further processed. UIV serves as a primary defense in the web application against such attacks by classifying user inputs as *valid* or *invalid*, accepting *valid* inputs, and rejecting *invalid* inputs such as malicious inputs. Due to their importance, a defect-free UIV has become a key means of enhancing a web application's security. However, in practice, developers rarely produce defect-free UIV. A recent survey conducted by the Open Web Application Security Project [4] found that the top three vulnerabilities of web applications are due to their UIV's improper handling of user inputs. Furthermore, several major common vulnerabilities (such as hidden fields, cross-site scripting, and SQL injection) in web applications are due to the absence or insufficiency of UIV.

Although UIV is critical in web-application security, UIV testing, which is a common methodology for detecting defects in UIV, has received little attention in the last decade. Often UIV testing is conducted using two traditional testing approaches: white-box [5–7] and black-box [8] testing. White-box testing is conducted based on the source code of the UIV, whereas black-box testing is conducted based on UIV specifications without using the source code of the UIV. Both approaches have their own strengths and weaknesses. For example, black-box testing can identify omission faults, i.e., faults induced due to missing logic or code, but cannot identify specific faults (such as extra logic or functionalities) induced by the implementation of the UIV. On the other hand, white-box testing can identify specific implementation faults but cannot detect omission faults. Furthermore, there are various challenges (not addressed by existing approaches) in applying these white-box and black-box testing approaches individually for UIV testing due to specific natures of UIV.

In our proposed research, we address these challenges of UIV testing when **white-box** or **black-box** testing is applied, and propose a novel **grey-box** testing approach that combines white-box and black-box testing in a practical and unique way for testing UIV, specifically tailored for security testing. More specifically, our proposed research addresses the following main technical challenges:

White-box testing in the presence of complex string operations. White-box testing approaches generate test inputs to achieve high code coverage such as branch coverage of UIV code, called user-input validators in short as validators. However, specific natures of validators such as using complex string operations (including complex regular-expression matching) often pose challenges for existing white-box approaches [5–7]. To address these challenges, we propose a new approach that can effectively generate *valid* and *invalid* string inputs to achieve a high code coverage of validators that use complex regular expressions.

Black-box testing in the absence of specifications. Black-box testing approaches require UIV specifications to provide guidance in conducting testing; however, in practice, specifications are often not available for validators. As specifications often do not exist, existing approaches [8] commonly conduct random testing. As a validator directly accepts user inputs, the input space is often large and valid inputs comprise

only a small portion of the entire input space. Therefore, it is challenging for random testing to generate defect-exposing inputs, which often lie at the boundaries of the valid input space.

To address the preceding challenge of lacking specifications, we propose to infer specifications using a new approach called **multiple-implementation testing**, a type of differential testing [9]. In multiple-implementation testing, we collect several other validators that are similar (in UIV functionalities) to the validator under test, and analyze these validators to infer specifications. These other validators can be collected from either open source or proprietary code bases. We will use these inferred specifications to generate both *valid* and *invalid* inputs. Based on specifications, *valid* inputs can be generated relatively easily, but it is challenging to generate *invalid* inputs, which are valuable for detecting defects, especially vulnerabilities (such as failing to guard against some malicious inputs). We will adapt the notion of mutation testing [10] in mutating specifications to generate *invalid* inputs. Mutation testing helps generate *invalid* inputs that lie at the boundaries of the valid input space and thereby can help detect potential defects. We will also use multiple-implementation testing for collecting test oracles for the generated *valid* and *invalid* inputs, and these test oracles can serve as expected outputs during UIV testing.

Grey-box testing in addressing respective weaknesses of white-box and black-box testing. We propose a new grey-box testing approach that combines both white-box and black-box testing approaches. The rationale for our grey-box testing approach is that covering a path in a validator using a white-box approach does not ensure the absence of defects such as vulnerabilities in that path. To address the preceding issue, an ideal approach is to test all feasible paths (white-box) with all possible input partitions (black-box). However, such an ideal approach is not practical to achieve. We propose a novel grey-box testing approach based on the security requirements of validators (e.g., in the form of attack patterns). Our grey-box testing approach combines both white-box and black-box approaches, and improves the chances of detecting vulnerabilities in validators. We will also define new test coverage criteria for the grey-box testing approach.

In our proposed approaches for UIV testing, we will **provide effective tool support for both test-input generation and test-result assessment**. These approaches **cover a compressive spectrum of testing types**, ranging from black-box testing, grey-box testing, to white-box testing. To conduct effective UIV testing, these approaches **creatively exploit a comprehensive sources of information** including regular expressions (for characterizing valid user inputs and attack patterns) and other implementations of the same validator type as the validator under test.

1.1 Our Vision

We believe that **interdisciplinary approaches** of exploiting software engineering knowledge to address security assurance problems (as proposed in this project) are needed to develop effective solutions to improve web-application security.

1.2 Research Objectives

This project has four main objectives: (1) to address the issue of lacking specifications in UIV black-box testing by automatically inferring specifications with multiple-implementation testing, (2) to improve white-box testing of UIV by addressing the issues related to handling complex string operations that often exist in UIV implementations, (3) to improve black-box testing of UIV by using inferred specifications that can help generate *valid* and *invalid* test inputs, and (4) to address respective weaknesses of black-box or white-box testing with an effective grey-box testing approach.

Our work will focus on UIV testing of web applications written in Java and C#. However, our findings will be fundamental and generally applicable to UIV testing of web applications written in other languages such as PHP. We will evaluate our proposed tools on both web applications written by students in the proposed educational activities and industrial-strength web applications collected from the Internet, universities, and industry.

1.3 NSF Merit Criteria

Intellectual merit criteria are addressed throughout the proposal as follows.

- **Advance knowledge and understanding.** The proposed research will fundamentally advance knowledge and understanding in foundations, methodologies, techniques, and tools in the interdisciplinary field of software security testing.
- **PI qualifications.** The PI is well qualified for the proposed research with significant prior experience in software engineering, including test generation [11–25], regression testing [26–33], and mining program execution [34–38] and code repositories [39–52], as well as testing security policies [18, 53–59]. The PI has also conducted research on application security testing [32, 60]. Substantial groundwork from the PI has been laid for the proposed research efforts with preliminary work described in Section 4. The PI's ongoing projects on software testing and analysis have been sponsored by NSF, ARO, NIST, Microsoft Research, IBM, ABB Inc., and other local industrial members of NCSU CACC [61]. His research has produced impact on industry including major testing-tool companies and research labs[1]. These industrial relationships will allow the PI to evaluate the proposed research in real industrial development environments besides student development environments. In particular, this proposed research will be conducted in collaboration with **Microsoft Research** (see the letters from Dr. Schulte, a Research Area Manager at Microsoft Research) in leveraging Pex [62, 63], an automated structural testing tool for .NET developed at Microsoft Research, and **NIST** (see the letters from Dr. Hu, a researcher at NIST) in collecting benchmarks and distributing the proposed tools and research results.
- **Creative and original concepts.** The proposed research explores new approaches with novel applications of regular-expression-based testing, execution-path-based testing, and multiple-implementation testing for UIV testing, as well as addressing unique requirements and challenges posed by different UIV testing types including white-box, black-box, and grey-box testing.

Broad impact criteria are addressed throughout the proposal as follows.

- **Discovery and understanding.** Through our research, we will gain understanding about UIV testing of web applications, which should lead to higher security assurance of web applications. Our validation efforts involve students and professionals, promoting teaching, training, and learning of assuring high software security.
- **Participation.** The preliminary work for the proposed research has been conducted by a female Ph.D. student (Nuo Li), who will continue to participate in the proposed research. The participating students will later include more female or minority students. The PI has an excellent record of mentoring minority students. His research group has accumulatively included six African American students and five female students, and he is an active member of MentorNet, mentoring three minority students. The PI will request REU funding for the summers and, in this way, will involve more underrepresented students.
- **Infrastructure.** The research will enhance the infrastructure for teaching and research by providing open source tools and data sets for use by students and practitioners, and for enhancement by other software engineering or security researchers. The PIs will provide related learning modules for educators. Close collaboration with local government and industry partners (see the letters Microsoft Research and NIST) will be conducted to validate the effectiveness of the proposed tools for UIV testing in real environments. The PI will also collaborate with other industrial partners through the Center for Advanced Computing and Communication (CACC) at NCSU, as many organizations that have the need for web application security testing are partners of the center.
- **Dissemination.** Our research results will be disseminated in software engineering and security conferences, journals, and books in various forms (e.g., papers, tutorials, and book chapters). Additionally, our developed tools and experimental materials will be posted on Codeplex [64] and SourceForge [65] as open source and will be linked from our project web sites. Our learning materials will be posted

[1] `http://www.csc.ncsu.edu/faculty/xie/research.htm#impact`

from our web sites and on the Open Seminar for Software Engineering [66]. The PI's C# test generation tools have been released at the Pex extensions project web [67] at Codeplex as open source and teaching materials on testing have been shared at a collaborative wiki site [68].

- **Benefits to society.** First, our work should lead to good practices for UIV testing of web applications in improving web-application quality and security in general. Second, innovations in new testing algorithms and tools tend to propagate quickly across application or task domains where UIVs are used.

This proposal is organized as follows. Section 2 discusses related work. Section 3 describes the proposed work as four detailed task plans. Section 4 presents evaluation plans and preliminary results. Section 5 presents the proposed education and dissemination activities. Section 6 describes the project schedule and management plan. Section 7 describes the capabilities of the PI, and Section 8 lists the results from prior NSF support.

2 Relationship to Prior Work

Offutt et al. [8] developed a strategy called bypass testing where client-side tests intentionally violate explicit and implicit checks on user inputs. They summarized the types of Web-application UIV black-box testing and classified such tests to three levels. However, they did not provide an approach to generate UIV test inputs, whereas our proposed work focuses on UIV testing at the server side and supports both test generation and test oracles.

Livshits et al. [69] proposed an approach to statically analyze whether a program assigns invalid inputs to parameters of methods that implement dangerous operations, such as operating a database. They analyzed Java byte code to find out such methods, and checked whether the parameters match some vulnerability patterns that are specified by users with a query language. Xu et al. [6] also focused on the parameters of "dangerous operations", but with a dynamic analysis approach. They tracked flow of taint information from untrusted input into parts of the generated outputs to check whether a program uses some invalid characters to generate outputs. Su and Wassermann [70] presented a formal definition of command injection attacks in the context of Web applications, and gave an algorithm for preventing these attacks based on context-free grammars and compiler-parsing techniques. The preceding approaches provide a good way to prevent general user-input attacks, whereas our approaches complement these previous efforts by effectively generating test inputs for UIV testing, being able to detect domain-specific or application-specific UIV vulnerabilities.

Liu et al. [5] constructed a control-flow diagram from source code of a Web application. Based on the control-flow diagram, they identified where validators reside in the diagram, and classified the branches after input points as acceptance branches or rejection branches. Then, they generated test inputs to cover the branches or conditionals for code portions that contain validators. They supposed that if both the acceptance branch and rejection branch after an input point can be covered, the input point is validated. However, they did not classify test inputs as valid or invalid. A rejection branch may be covered by a valid test input (when validators contain defects), and an acceptance branch may be covered by an invalid test input. Different from their approach, our proposed work will generate test inputs in three ways (black-box, grey-box, and white-box testing), and classify the generated test inputs as valid or invalid, and determine test results as passed or failed based on multiple-implementation testing.

In N-version programming [71], different versions of a program are executed in parallel in the same application environment with the same inputs, and then the outputs are collected to a voter. The majority outputs are treated as the correct output. Based on the theory of N-version programming, more than one team can be asked to develop the same program independently to improve the reliability of software operation. In contrast, our multiple-implementation testing approach determines whether a validator is defective based on whether it accepts invalid test inputs. In addition, in N-version programming, test inputs need to be generated to expose different outputs (if any). In previous work [72–74], Knight et al. generated test inputs randomly for their programs under test. In contrast, we generate test inputs for more effectively exposing behavioral

differences based on advanced test generation tools for structural testing.

McKeeman [75] proposed differential testing for testing several implementation of the same functionality, specifically testing different implementations of C compilers. Lämmel and Schulte [76] developed a C#-based test-input generator called Geno and applied it in differential testing of grammar-driven functionality. Groce et al. [77] applied random testing on flash file system software and conducted differential testing on the software and its reference implementation. Evans and Savoia [78] addressed the problem of detecting behavioral differences by generating tests that achieve high structural coverage separately for an old and a new versions of the program under test. Different from previous differential testing approaches, our multiple-implementation testing approach conducts more effective test generation guided by implementation differences. Our proposed approach additionally classifies test inputs and assesses test results in the context of UIV testing of Web applications.

Existing test generation approaches such as Dynamic Symbolic Execution (DSE) [63, 79–81] can effectively generate test inputs for programs with primitive types, such as `integers`. DSE executes the programs under test (such as UIVs) with random inputs, and collects symbolic constraints on inputs from the predicates in branching statements. DSE explores alternate paths by systematically flipping the captured constraints and by generating concrete values (using a constraint solver) that can cover the alternate paths. Our proposed research leverages and improves DSE in conducting UIV testing.

3 Proposed Activities

Our proposed research focuses on defending against web-application attacks by developing new approaches for UIV testing of web applications. Our proposed research significantly enhances UIV testing by addressing several fundamental research questions in UIV testing such as proposing new approaches for automatically inferring specifications and improving white-box and black-box testing approaches. The proposed research consists of the following four tasks.

Task 1. Multiple-implementation testing. We will automatically infer specifications by analyzing other validators of the same type as that of validator under test (i.e., with the same UIV functionalities). We will also use multiple-implementation testing for generating test inputs (and their respective test oracles) that expose the behavioral differences among the different validators of the same type. These difference-exposing test inputs can have good chances in detecting defects in the validator under test.

Task 2. White-box testing. We will improve white-box testing of UIV by developing new approaches for effectively addressing test-generation challenges related to handling complex string operations that often exist in UIV implementations.

Task 3. Black-box testing. We will improve black-box testing of UIV by using inferred specifications to help generate *valid* and *invalid* test inputs. We will adapt mutation testing to systematically generate invalid inputs from the inferred specifications in Task 1.

Task 4. Grey-box testing. We will develop a new grey-box testing approach that combines white-box and black-box approaches. Our grey-box testing approach will combine both approaches in a practical and unique way, and improve the chances of detecting defects in UIVs.

We next describe these four tasks in detail in subsequent sections along with preliminary results for some of the tasks.

3.1 Task 1: Multiple-implementation testing

The objective of Task 1 is to infer specifications by conducting multiple-implementation testing to reduce the effort of providing the specifications manually.

We will infer specifications in the form of regular expressions since UIVs often include complex string related operations that can be conveniently expressed as regular expressions. Sometimes specifications in the form of regular expressions can be quite complex to infer and also insufficient to express some UIV types. For example, consider the credit-card validator type. The inputs such as strings of the credit-card validator type cannot be solely represented as regular expressions because there can be some semantic constraints (such

as checksum on the digits of a credit card) among the characters within the input strings. To address this preceding issue, we will also generate test inputs (and their respective test oracles) along with specifications.

Basic idea of multiple-implementation testing. Inferring regular expressions from the validator under test alone as specifications can suffer from one major issue: the regular expressions implemented by a faulty validator under test can be incorrect and using these regular expressions as specifications in UIV testing is ineffective in detecting faults. To address this issue, we will apply multiple-implementation testing by comparing the behavior of the validator under test with other validators of the same validator type[2] (i.e., with the same UIV functionalities). Multiple-implementation testing is a type of differential testing [9] with more than two implementations of a particular type of software systems (such as validators in our context). Multiple-implementation testing aims to find behavioral differences between the different versions of a software system. A behavioral difference between two versions can be reflected by the difference between the observable outputs produced by the execution of the same test on the two versions. To apply multiple-implementation testing on validators, we will collect different validators of the same type from either open source or proprietary code bases. For example, to collect from open source code bases, we will use source code search engines (such as Krugle [82] or Google Code Search [83]).

If a fault in the validator under test also occurs in only a minority of other validators of the same type, applying multiple-implementation testing is effective to detect this fault. Otherwise, multiple-implementation testing cannot directly help detect this fault. Based on our preliminary investigation, many faults in validators uniquely occur in one validator or a minority of validators of the same type (instead of consistently occurring across many other validators of the same type) and can be detected by multiple-implementation testing. We will conduct more extensive empirical studies on investigating the characteristics and frequency of faults that can occur across many other validators of the same type; the empirical results will provide insights for us to investigate new techniques for dealing with these types of faults. Note that if a fault occurs in a majority of validators (but not all validators) of the same type, multiple-implementation testing would report difference-exposing test input-output pairs for the remaining (correct) validators as fault-exposing pairs for inspection. After developers inspect and determine the reported input-output pairs are in fact expected, the fault in the majority of the validators can be indirectly exposed.

```
public boolean isValid1(String value)
{   String ZIP_REGEX = "[0-9]{5}";
    String PLUS4_OPTIONAL_REGEX = "([ |-]{0,1}[0-9]{4})?";
    Pattern mask    =  Pattern.compile(ZIP_REGEX + PLUS4_OPTIONAL_REGEX);
    Matcher matcher = mask.matcher(value);
    if (!matcher.matches()) return false;
    else return true;
}
public boolean isValid2(String zipcode)
{   boolean isValidZip = zipcode.length() == 5;
    try
    {   int zip = Integer.parseInt(zipcode);
        isValidZip = isValidZip && ((zip >= 0)&& (zip <= 99999));
    } catch (NumberFormatException e)
    {   isValidZip = false;
    }   return isValidZip;
}
public boolean isValid3(String value)
{   Pattern[] PATTERNS = new Pattern[]
{   Pattern.compile("^[0-9][0-9][0-9][0-9][0-9]$"),
    Pattern.compile("^[0-9][0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$"),
    Pattern.compile("^[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]$")
};  for (int i=0; i<PATTERNS.length; i++)
    {   Matcher m = PATTERNS[i].matcher(value);
        if (m.matches()) return true;
    } return false;
}
public boolean isValid4(String s)
{   boolean flag=true;
    String zipTemplate = "[0-9]{5}[ -][0-9]{4}";
    Matcher matcher = Pattern.compile(zipTemplate).matcher(s);
    if (!matcher.matches()) flag=false;
    return flag;
}
```

Figure 1: Four zip-code validators collected from open-source code bases

Inference of specifications as regular expressions from individual validators. This subtask is to infer

[2]We use the notion of *validator type* to refer to the kind of input validation (such as email, zip code, and credit card number) a validator implements.

regular expressions from each validator of the same type and the inferred regular expressions can be compared in the next subtask to derive specifications to be used in our black-box testing (Task 3) to generate both *valid* and *invalid* inputs. We will investigate two types of approaches: static and dynamic analysis approaches. First, we will apply static string analysis approaches [84, 85] on the source code of each validator. In particular, given a string variable at a particular program location in an application, these static string analysis approaches statically analyze the application source code and keep track the constraints accumulated via string-manipulation operations and string-matching conditions, and then build an automaton to represent all the possible strings that can be assigned to the string variable at that particular program location. In our context, we will designate to string analysis approaches the string variable being validated that is on the branch of reflecting the true value of the validator return value.

Second, we will apply Dynamic Symbolic Execution (DSE) [63, 79–81] on testing each validator to infer regular expressions. Our dynamic analysis approaches can address some limitations inherent in our static analysis approaches such as imprecision and over-conservativeness. In addition, our dynamic analysis approaches can infer specifications (e.g., the ones for credit-card validators) that can be represented by only a mixture of regular expressions and other semantic constraints. The key idea of our dynamic analysis approaches is to derive constraints called path conditions on inputs that force the execution of the validator to follow paths that return true values. In principle, if we can enumerate all the paths that return true values, we can conjunct the path conditions for these paths to form the constraints on all the inputs accepted by the validator. But in practice, a validator can often include infinite paths (e.g., due to the presence of loops). We will explore techniques for exploring a sufficient finite set of paths (such as all feasible independent paths [86]) and generalizing constraints for valid inputs from the path conditions for this finite set of paths. We will also explore techniques for separating semantic constraints from constraints that can be represented by regular expressions, among the constraints collected via DSE. We will empirically compare the cost effectiveness of the static and dynamic approaches and explore the integration of the two to address their respective weaknesses.

Comparison of specifications across validators of the same type. One primary issue with inferred regular expressions from multiple implementations is that these inferred regular expressions may not be the consistent (i.e., same) with each other. Without tool support to compare the semantic differences of these regular expressions, it is challenging to manually decide which inferred regular expressions can serve as common specifications. Furthermore, some inferred regular expressions may be incorrect due to defective validators collected from open source code bases or some inferred regular expressions may define different string formats from the validator under test. To address these preceding issues, we will compute the intersection of the input domains of the inferred regular expressions with algorithms in automata [87], which are produced from regular expressions. The resulting intersection represents the smallest input domain that defines the most common specification among the multiple implementations. In addition, we will also generalize the differences among the input domains of the inferred regular expressions, and then report to developers the differences to help the developers decide which regular expression is close to their expected requirements. We will also investigate an alternative approach for deriving common input domains by enumerating concrete string inputs that can be accepted by any of the validators under comparison, and then apply automaton learners [88] or partial order learners [89] to learn automata for accepting all or most of the string inputs. We will empirically investigate the cost effectiveness of (1) the approach for first inferring regular expressions for each validator and then comparing these regular expressions and (2) the approach for first generating concrete string inputs for each validator and then learning automata from these concrete string inputs.

Basic approach for generating test inputs and test oracles. We use the illustrative examples in Figure 1 to explain our approach for generating test inputs (and their respective test oracles) in multiple-implementation testing. Figure 1 shows four validators of the zip-code validator type. Consider that the zip-code validator, `isValid1`, is the validator under test, and the other three validators are collected from open source code bases for applying multiple-implementation testing. The validator under test uses a regular

expression to validate whether a string consists of five or nine digits and includes a defect as the validator accepts an invalid input "`12345|6789`". Without requiring manually provided test inputs or test oracles, our approach will detect this defect by automatically generating such inputs by comparing `isValid1` with the other three validators (`isValid2 - isValid4`). Indeed, this defect is a real one that we found in our preliminary evaluation (described in Section 4).

Our objective is to generate test inputs that expose differences among these validators, hereby referred as difference-exposing test inputs. The rationale for generating these test inputs is that these difference-exposing test inputs can have high chances in detecting defects in the validator under test, as shown in our preliminary results. As a validator returns either `true` or `false`, to expose differences, we will enumerate all feasible paths (all feasible independent paths in the case of having infinite feasible paths) in each validator and classify these paths into two categories: `true` paths and `false` paths. The `true` paths and the `false` paths are the paths that return `true` or `false` values, respectively. We will next conjunct the path conditions for the `true` path of one validator with the `false` path of the other validator, and then solve the resulting constraints to derive difference-exposing test inputs. Along with generating these test inputs, we will generate test oracles for these test inputs based on the majority voting among the actual outputs by executing all validators with these test inputs.

Optimized approach for generating test inputs and test oracles. Our basic approach described in the previous subtask enumerates all feasible paths or independent paths and pairwise compares `true` paths and the `false` paths. Such a process can be too costly in testing time considering that constraint solving is normally quite expensive. To address the preceding issue, we will explore an optimized approach that shall be more efficient than our basic approach. For example, if two explored paths p_1 and p_2 in one validator share the same prefix subpath. If the conjunction of p_1 with another path p'_1 from another validator is not unsatisfiable, and such unsatisfaction is due to one constraint from the prefix subpathh, we can avoid comparing p_2 with p'_1 while knowing their conjunction is not unsatisfiable. When the validators under comparison are amenable to regular-expression inference, we can use the information from the validators' inferred regular expressions to prune out some pairwise comparison or even directly generate difference-exposing test inputs. We will empirically investigate cost effectiveness of these different approaches.

Other open issues and research challenges. In traditional multiple-implementation testing [73, 74, 78], all the implementations should have the same requirements, which enable developers to detect defective implementations based on whether an implementation behaves differently from the other implementations. However, in practice, requirements for different validators of the same type can be slightly different. For example, consider phone-number validators where a phone-number validator requires inputs as nine-digit number either with no separator or separated by a "-"(such as 999-999-9999). Consider that three other phone-number validators require the inputs as nine-digit number with no separator only. Traditional multiple-implementation testing techniques detect the first validator (which requires 999-999-9999) as defective but the first validator is indeed not defective and has different requirements than the other validators. To address the preceding challenge, we will detect input-domain relationships among the different validators of the same type. Such relationships can help developers understand the different requirements of validators. We will also explore automatic inference of format converter from one validator to another one if they have slightly different requirements and these requirements can be reconciled with simple conversion such as the one for removing "-" in one string accepted by one phone-number validator to produce another string accepted by another phone-number validator.

3.2 Task 2: White-box testing

The objective of Task 2 is to improve white-box testing of UIV by effectively addressing the issues related to handling complex string operations and apply our improved white-box testing to detect difference-exposing test inputs in Task 1.

When there are complex string operations in the program under test, the exploration space of Dynamic Symbolic Execution (DSE) [63, 79–81] is often large and the collected symbolic constraints are too complex

to be solved. Therefore, DSE cannot effectively deal with applications such as UIV that take strings as inputs and include complex string operations such as regular expression matching. This task is to addresses this challenge in white-box testing of UIV.

Basic approach of white-box testing. We will develop a new DSE-based white-box approach that can effectively handle string inputs. In particular, we will reduce the exploration space of DSE by replacing complex and generic regular expression matching operations with specific synthesized checking code based on the automaton represented by the regular expression. The checking code of automata can also guide DSE to generate test inputs to cover different string formats defined by regular expressions. Figure 2 shows the overview of our approach that consists of two main components: automaton synthesizer and symbolic execution engine. The input of our approach is a program under test such as a validtor. The outputs of our approach are generated test inputs that can achieve high code coverage of the program under test and high test coverage of the regular expressions (defined in Task 3) contained in the program. When a complex regular expression matching operation appears in the program under test, we will use the automaton synthesizer to transform the operation into a checking code of automata. We next use the synthesized checking code of automata to replace the original operation and use DSE to generate test inputs for the transformed program.

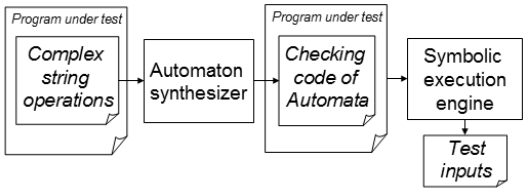

Figure 2: Overview of white-box testing approach

```
01:public bool InputsChecker(string cardNum, string email){
02:  if (!Regex.IsMatch(email,
03:   @"^[\w\-]+@([\w\-]+\.)+[\w\-]+$"))
04:     return false;
05:  if (!Regex.IsMatch(cardNum,
06:     @"^[\d]{4}\-[\d]{4}\-[\d]{4}\-[\d]{4}$"))
07:     return false;
08: string number = ExtractNumbers(cardNum);
09:  if (!IsLuhnValid(number))
10:     return false;
11:  return true;
12:}
```

Figure 3: Code sample of a validator

We next explain the details of our approach using a credit card validator shown in Figure 3. In Lines 2-7 of Figure 3, the validator uses regular expressions to check the formats of `email` and `cardNum`. Existing DSE approaches cannot generate test inputs that explore the `false` branch of Line 2 due to large exploration space inside the `IsMatch` of the `RegEx` class. To address this issue, we will transform the regular expression into a synthesized checked code based on automata and use DSE to explore the checking code of the automata for generating input strings that can explore the `false` branch of Line 2. Figure 4 shows the sample checking code that can be automatically generated by our approach for the `email` regular expression. Figure 5 shows the equivalent automaton of the checking code. After the transformation, DSE can automatically generate acceptable strings such as `00ZZ0Z4V4V4V4VfKfKfKfKfKfKfKfKfK@0.0.0.0` and `0@0.0` for `email` regular expression to explore the `false` branch of Line 2.

Optimized approach of white-box testing. Although we can reduce the exploration space of DSE by using synthesized checking code, the exploration space of DSE may be still be large if the regular expression is complex. In such scenarios, DSE cannot generate acceptable strings for the regular expression by exploring our synthesized checking code. A possible solution to address this issue is to improve the DSE search strategy. Traditionally, DSE iteratively generates test inputs to cover various feasible paths in the synthesized checking code. In particular, DSE flips some branching node from a previous execution to generate a test input for covering a new path. The node to be flipped is decided by a search strategy such as depth-first search. This exploration is quite expensive since there can be an exponential number of paths in the synthesized checking code generated for complex regular expressions. However, often not all paths are needed to be explored to generate test inputs. For example, when we generate difference-exposing test inputs in Task 1, the execution of many branches often cannot help in generating difference-exposing test inputs. Similarly, in the credit-card validator shown in Figure 3, we do not have to explore paths that return `false` value to

cover the `false` branch of Line 2. To make the test generation process efficient, we will develop a new search strategy for DSE to avoid exploring irrelevant paths.

Other open issues and research challenges. One challenge with our preceding white-box testing approaches is that the generated test inputs for the validator under test may not be valid with respect to the entire web application. For example, consider an SSN validator that accepts input strings of the form "XXX-XX-XXXX", where each "X" represents an integer. Consider that we found a defect with an input "234*78*9078". However, sometimes, the web application might not allow the user to enter "*" in the place of "-". In this scenario, the detected defect is a false warning. The primary reason for such false warnings is common to all unit testing approaches such as our preceding white-box testing approaches as these unit testing approaches do not test from the entire system point-of-view. To reduce such false warnings in our white-box testing of UIVs, for each defect-revealing unit-test input, we will automatically check whether such input is feasible from the entire web application point-of-view by identifying the paths from the user input point (where a user enters input) of the web application to the validator under test.

```
01:public class EmailAutomaton{
02: public enum States{STATE0, STATE1, STATE2,
03:                    STATE3, STATE4, STATE5,}
04: States _state = States.STATE5;
05: public bool AcceptChar(char c){
06:  switch (_state){
07:     case States.STATE0:
08:        if (-1 != (".").IndexOf(c)){
09:            _state = States.STATE3;
10:            return true;
11:        }
12:        if (-1 != ("0123456789abcdefghijklmno" +
13:                "pqrstuvwxyzABCDEFGHIJKLMNO" +
14:                "PQRSTUVWXYZ-").IndexOf(c)){
15:            _state = States.STATE0;
16:            return true;
17:        }
18:        return false;
19:     case States.STATE1:
20:         ......
21:  }
22: }
23:}
```

Figure 4: Checking code of the email automaton

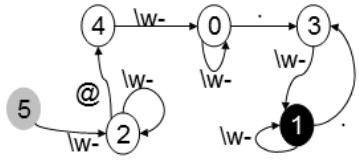

Figure 5: Automaton of the email regular expression

3.3 Task 3: Black-box testing

The objective of Task 3 is to improve black-box testing of UIV by effectively generating both *valid* and *invalid* test inputs from specifications inferred in Task 1.

Generation of valid test inputs using automaton exploration. Given a specification of a validator in the form of a regular expression, we will generate valid test inputs by translating the regular expression into an automaton. Each transformed automaton has one begin state and multiple end states. To generate valid test inputs from the automaton, we will consider the automaton as a graph and use search strategies such as the depth-first strategy to explore the automaton. In particular, we start from the begin state and explore the automaton until we reach an end state. We generate valid test inputs from the visited transitions of each such exploration.

To illustrate our approach, consider the automaton shown in Figure 5 for the e-mail regular expression, where State 5 is the begin state and State 1 is an end state. Using our exploration, we can generate a valid test input "\w@.\w" corresponding the exploration path "5 → 2 → 4 → 0 → 3 → 1". In the preceding valid test input, "\w" indicates any character. In practice, a transformed automaton is often more complex and contains loops such as the "3 → 1 → 3" or "2 → 2" shown in our example automaton. Such loops may result in infinite number of paths in the automaton and a naive exploration strategy can get stuck among those loops, failing to generate any valid test inputs. A simple approach to address the preceding issue is to visit each transition in the automaton only once. For example, when the transition "2 → 2" is already explored, we avoid exploring the same transition again. However, the valid test inputs generated from such a simple approach may cover only a small portion of valid input space. To address this issue, we will use a

bounded-depth-first-exploration strategy. In our approach, when a loop is encountered, we will explore the loop up to a certain bound, such as 10 times. We expect that our bounded exploration strategy can work well in practice due to fact that validators often accept string inputs of bounded length. We will empirically investigate the cost effectiveness of these different strategies.

Coverage criteria for valid test-input generation. To measure the quality of valid test inputs generated from the transformed automaton, we will adapt an existing graph-coverage criterion called independent-path coverage [86]. An independent path is a exploration path in the automaton that explores at least one new transition from the existing exploration paths. The reason for adapting independent-path coverage instead of all-paths coverage is that all-paths coverage could be infeasible due to an infinite number of exploration paths in the transformed automaton. Achieving 100% independent-path coverage ensures that generated valid test inputs cover every state and every transition in the transformed automaton. We will empirically compare the cost (the number of required tests) and effectiveness (fault-detection capability) of state coverage, transition coverage, and independent-path coverage criteria of an automaton (i.e., regular expression).

Generation of invalid test inputs using mutation testing. We will generate invalid test inputs that lie at the boundaries of the valid input space. The rationale behind this technique is that such invalid test inputs can often expose defects as developers often fail to handle boundary scenarios properly. We will use mutation testing [10, 90] to generate such invalid test inputs. In mutation testing, we will generate invalid test inputs based on the mutated regular expressions. To mutate the regular expressions, we will define different types of operations, called mutation operators, on regular expressions. We next present six such example mutation operators.

- MO1: remove the mandatory sets from a regular expression.
- MO2: disorder the sequence of sets in a regular expression.
- MO3: change the repetition time of selecting elements from a set.
- MO4: select elements from complementary sets of sets in a regular expression, especially characters next to the boundary of the input domain (for example, a validator validates whether an input string stands for a number between “1” and “13”, then we generate test inputs “0” and “14”).
- MO5: insert invalid and dangerous characters, such as an empty string, strings starting with a period, and extreme long strings, into a regular expression.
- MO6: insert special patterns, such as some common attack patterns [91], into a regular expression.

We will empirically investigate the costs and benefits of these mutation operators and select an effective subset to use in our test-input generation.

Coverage criteria for invalid test-input generation. To measure how well invalid test inputs are generated using mutation testing, we define a new coverage criterion, called mutation-testing-coverage criterion. To achieve 100% coverage, our mutation-testing-coverage criterion defines that every element in the regular expression has to be mutated at least once and an *invalid* test input has to be generated from such a mutated regular specification.

Other open issues and research challenges. An open issue in our valid test-input generation is how to define the upper bound for our bounded-exploration strategy. To infer the upper bound, we will conduct an empirical study on existing validator types such as email or credit-card validators and investigate their validators’ characteristics. We expect that these upper bound values can also be specific to a validator type as different validator types accept bounded strings of different lengths. Based on the empirical results, we will investigate heuristics for deriving a desirable bound.

We next explain an open issue in invalid test-input generation. The complete input domain of a validator can be broadly divided into two spaces: valid and invalid spaces. We refer to the invalid space as the part of the complete domain excluding the valid space. A mutated regular expression represents a part of the invalid input space. As described earlier, our objective is to generate invalid test inputs that lie at the boundary of the valid input space. However, if we randomly mutate a regular expression and generate invalid test inputs

from such a mutated regular expression, the probability of these generated invalid test inputs to lie at the boundary of the valid test input space is quite low, and thereby the probability of detecting defects can also be low. For example, consider a simple phone-number validator whose specification is represented with a regular expression "[\d]10". The valid input space includes all strings with 10 digits. If we mutate the regular expression as "[\d]5", the invalid test inputs generated from the mutated expression can be easily rejected by the phone-number validator, as the part of the invalid input space is not at the boundary of the valid input space. To address the issue, we will develop a systematic procedure for mutation testing in such a way that the invalid test inputs often lie at the boundaries.

3.4 Task 4: Grey-box testing

The objective of Task 4 is to address respective weaknesses of white-box and black-box testing approaches by developing a grey-box approach.

The rationale behind our grey-box testing approach is that covering a path in a validator using the white-box approach does not ensure the absence of defects in that path. To address this issue, an ideal approach is to test all feasible paths (as in white-box testing) with all possible test inputs or input partitions (as in black-box testing). However, such an ideal approach is not practical to achieve.

Basic approach of grey-box testing. To improve the chances of detecting defects and to increase the confidence in UIV testing, we will develop a practical and unique grey-box testing approach, specifically tailored for security testing. We will first conduct a static analysis of the validator and classify the paths in the validator into two categories: high-priority and low-priority paths. The rationale behind our classification is that not every feasible path in the validator can be of the same importance and *only* a few paths can be of high importance. We classify such paths of high importance as high-priority paths. This classification of paths in the validator into high-priority and low-priority effectively reduces the number of paths to be considered for our grey-box testing approach, thereby making the grey-box testing approach more feasible.

To classify a path as high-priority path, we will use two criteria. First, we identify the paths that accept user inputs and allow them to be further processed by security-sensitive operations such as database operations, and classify those paths as high-priority paths. The rationale behind our criterion is that such paths are often quite important and are relatively low in number due to specific natures of validators. For example, the major purpose of validators is to verify user inputs to detect and reject *invalid* inputs. Therefore, the paths that lead to the acceptance of user inputs shall be relatively low in number. Second, we identify the paths that include loops or recursions whose exit-conditions can be controlled by the user inputs, and classify those paths as high-priority paths. The rationale is that these paths can potentially allow attackers to generate malicious inputs and cause attacks such as denial of service via consuming the system resources such as CPU time or memory. We classify the paths other than high-priority paths as low-priority paths.

In our grey-box testing approach, after we identify high-priority paths, we will use black-box testing to generate various test inputs that exercise each high-priority path multiple times. As the main objective of our grey-box testing approach is to improve the quality of validators to defend against security attacks, in our black-box testing, we will generate test inputs based on known security attacks. We will define partitions of test inputs (e.g., specified with regular expressions or their mixture with semantic constraints) based on known security attacks (hereby referred as *test input partitions*) such as *SQL injection* attacks, and try to generate test inputs from each partition to exercise each high-priority path. Such a combination is a form of pairwise testing [92] (i.e., each pair of test-input partition and high-priority path).

Generation of test inputs in white-box testing to exercise a specific path in the validator is often challenging as we have to solve all constraints in the path condition for that path. Furthermore, in our grey-box testing approach, we try to generate test inputs for the path in such a way that these test inputs also belong to the test input partitions. Therefore, to generate such test inputs, we will combine the constraints collected from the high-priority path (in white-box testing) and the constraints defined by the test input partitions, and solve the conjunction of these constraints together.

Coverage criteria for grey-box testing. As we are the first to propose the grey-box testing approach for

UIV testing, the existing test-coverage criteria for either black-box or white-box testing may not be sufficient to measure how well the validator under test is exercised by our grey-box testing approach. Therefore, we will define a new coverage criterion as a combination of black-box and white-box coverages. For example, consider that there are six high-priority paths and three test input partitions. To achieve 100% coverage, our coverage criterion describes that the generated test inputs should cover all 18 possible combinations of high-priority paths and test input partitions. Achieving 100% coverage using our new coverage criterion can ensure that at least high-priority paths in the validator are resistent from known attacks.

Open issues and research challenges. One major open issue of our grey-box testing approach is how to represent the constraints of test input partitions, suitable for constraint solving. We will adapt regular expression patterns to represent the constraints of test input partitions. For example, consider an SQL injection attack where an SQL query such as "`SELECT * FROM Users WHERE (strUserName= 'test')`" is transformed into "`SELECT * FROM Users WHERE (strUserName = 'test' OR 'A' = 'A')`". As the predicate in the transformed query always evaluates to `true`, the SQL injection attack results in returning all entries in the `Users` table instead of the entries related to the user `test`. In our grey-box testing approach, we will define this SQL injection attack as a regular expression with an expression of the form "`test' OR 'A' = 'A`" in the predicate of the SQL query. An example of such regular expression can be "`\w{4,6}\'\sOR\s\'A\'=\'A`". We will also explore regular expression patterns or broader types of patterns for other attack types such as cross-side scripting.

4 Evaluation and Preliminary Results

We will design and implement a validator-testing system, including multiple research tools, for the proposed work, and evaluate the proposed work by conducting experiments and case studies on the system with the evaluation criteria and plan described below.

Evaluation Criteria. We evaluate the effectiveness of the proposed work based on three main metrics: the number of generated defect-revealing test inputs, real defects detected by the proposed work, the percentage of false positives (false warnings) among the reported defects. The higher number of detected real defects, the higher benefits the proposed work achieves. The lower percentage of false positives, the lower extra manual inspection cost the proposed work requires.

Evaluation Plan. We will use evaluation subjects collected from the following sources: (1) open source Web applications, such as Open Java Forum [93]; (2) Web applications developed in student course projects such as iTrust [94] used in courses being taught by the PI; (3) open source validators, which we will collect through source-code search engines; (4) industrial Web applications or test scenarios from our industrial collaborators such as Microsoft Research (see the Microsoft Research letter) and member companies from NCSU Center for Advanced Computing and Communication (CACC) [61], a membership-based industry/university cooperative research center; and (5) benchmarks collected from the community in collaboration with NIST collaborators (see the NIST letter).

Preliminary Results. We have implemented preliminary prototypes (based on Pex [62, 63]) for multiple-implementation testing and white-box testing approaches.

Multiple-implementation testing. We applied our preliminary prototype for multiple-implementation testing on 49 validators of 6 different validator types including credit card number, email address, phone number, SSN, URL, and zip code. We collected different validators of the same type from Krugle [82] and Google Code Search [83] by giving validator types (e.g., email validator) as keywords. Our preliminary results are quite promising. Our approach generated 2,446 test inputs in total, where each test input was accepted by at least one validator. Out of these test inputs, 92.4% were defect-exposing inputs. These defect-exposing inputs detected 43 validators as defective (including 2 false positives).

White-box testing. We applied our preliminary prototype for white-box testing on 11 validators from 10 open source validators or libraries collected from the Internet. We generated synthesized checking code for 493 regular expressions collected from a popular regular expression library called *RegExLib.com* [95]. The

preliminary results show that our proposed approach achieves higher block coverage than applying DSE without using our synthesized checking code of automata. In addition, our proposed approach also helped improve the test coverage of regular expressions.

We expect that much better results can be produced with the proposed tools, which will incorporate various advanced techniques (proposed in Tasks 1-4) that have not been implemented in our preliminary work.

5 Education and Dissemination

The PI will develop new course modules on security testing and analysis in the courses that he will teach during this project, including CSC 326 Undergraduate Software Engineering, CSC 510 Graduate Software Engineering, and CSC 712 Software Testing and Reliability. The term projects of CSC 326 and CSC 712 have been using iTrust [94], a health care web application that provides role-based access. There validator testing has been currently conducted manually. The PI plans to ask students in these courses to use the tools developed in this proposed research in their term projects. Students in CSC 712 will additionally conduct advanced course projects by expanding the features of the research tools to learn hands-on skills in developing tools for security testing. Students in other courses such as CSC 510 and CSC 516 will also use the proposed research tools for the homework assignments for the topic of security testing.

The PI will publish the results of the proposed work as papers in professional journals, workshops, symposia, conferences, and relevant professional meetings. Typical conferences appropriate for this dissemination include ICSE, FSE, ASE, ISSTA, IEEE S&P, and CCS. In the past, the PI has also established an excellent history in disseminating his research results not only through publications, but also through software releases and technology transfers. For example, the PI served as a mentor for 2008 Java Pathfinder project for Google Summer of Code[3], in helping improve its open source testing features. The PI will make the proposed tools available on SourceForge for other researchers and practitioners to use.

The PI has already carried out collaboration with Microsoft Research on developing new tools [19, 22] upon Pex [62, 63], during the PI's two visits (lasting 2.5 months) to Microsoft Research during summers of 2007 and 2008 and in the Fall semester of 2008 with the involvement of the PI's students. The PI's resulting C# test generation tools [19, 22] have been released at the Pex extensions project web [67] at Codeplex as open source and teaching materials on testing have been shared at a collaborative wiki site [68]. The proposed tools for C# UIV testing will be released at the Pex extensions project web [67] at Codeplex as open source.

In addition, the PI will build a dedicated project web site including an online repository for this project; the materials of the proposed experiments and case studies will be made available in this online repository. Furthermore, the PI will post his educational materials on the Open Seminar in Software Engineering[4] as well.

6 Research Plan and Schedule

Research plan. The work proposed in this project calls for three years of effort. The team consists of two Ph.D. students in addition to the PI. Additional undergraduate students will be supported through NSF REU supplements. The development of the four tasks will be conducted in close collaboration by all participants. In particular, the PI will be primarily responsible for the conception of the design, development, and evaluation of the proposed research, through each year's summer and release time. During each year, one supported graduate student will be primarily responsible for the development and evaluation of the proposed techniques and tools related to test input generation in the proposed research. The other supported graduate student will be primarily responsible for the development and evaluation of the proposed techniques and tools related to test oracles in the proposed research. The two graduate students will collaborate to integrate various components of the tools and conduct evaluation of the developed techniques and tools. The graduate

[3] `http://code.google.com/soc/2008/pathfinder/about.html`

[4] `http://www.openseminar.org/se/`

students will be working under the PI's supervision.

Research schedule. Based on these planned research activities, the time line is scheduled as charted in Figure 6. In addition to the tasks discussed in Section 3, in Year 3 the PI will also perform dissemination activities of our final results to broaden the impact of the work in academia and industry. In particular, as the PI did for previous projects, the PI will integrate our findings in the classes that he teachs, in the form of lectures and projects.

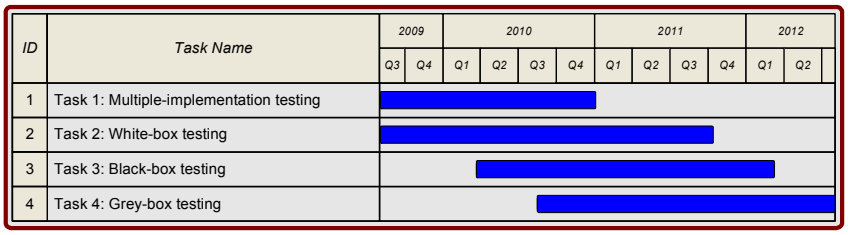

Figure 6: Research task plan

7 Qualifications of the PI for Proposed Work

The PI is an assistant professor in the Department of Computer Science at North Carolina State University. His research expertise is in automated software testing and mining software engineering data. He has developed new approaches on test generation [11–13, 15, 20, 37], test selection [11, 54, 96, 97], test abstraction [14, 34, 36, 98, 99], regression testing [26–29], and testing web services [17, 100], aspect-oriented programs [101–106], and access control policies [18, 53–59]. In addition, he has developed approaches on mining dynamic program information to help avoid bugs in user applications [35], and approaches on mining static program information to help understand code [48, 107], write API client code [39, 44, 45, 47], refactor code [40, 41], and detect code errors [42, 52, 108, 109]. His research has produced impact on industry including major testing-tool companies and research labs[5]. He has conducted research on security application testing [60] and security policy testing [18, 53–59], providing a solid foundation for conducting the proposed work.

8 Results from Prior NSF Support

The PI has one past NSF project and two current NSF projects. The research performed under **NSF CSR grant CNS-0720641** titled "CSR—SMA: Improving Software System Reliability via Mining Properties for Software Verification" (August 2007–July 2008) developed new techniques and tools for mining API properties from static program information generated by analyzing system code repositories. These mined properties can be used for static verification and other software development tasks, and our results have been published in major conferences and workshops [37, 38, 43–52]. The research performed under **NSF CyberTrust grant CNS-0716579** titled "CT-ISG: Collaborative Research: A New Approach to Testing and Verification of Security Policies" (August 2007–July 2010) develops a new approach for testing and verification of security policies, including application-level security policies (such as XACML policies) and network-level security policies (such as firewall policies), and our results have been published in major conferences and workshops (e.g., [25, 32, 58, 59, 110]). The research performed under **NSF SoD grant CCF-0725190** titled "Collaborative Research: SoD-TEAM: Designing Tests for Evolving Software Systems" (January 2008–December 2010) develops techniques for determining the adequacy of a test suite with respect to a set of software changes, and providing automated support for designing and developing test cases that target inadequately-exercised changes. Our results have been published in major conferences and workshops [19, 21–24, 30–33, 45, 111]. None of these three NSF projects has overlap with the proposed work in this proposal. The SoD project provides a strong foundation for the proposed work by providing underlying advanced test generation techniques for multiple-implementation testing. The CyberTrust project provides some groundwork and insights for applying testing techniques in the domain of application security.

[5] `http://www.csc.ncsu.edu/faculty/xie/research.htm#impact`

References

[1] Symantec Corporation. *Symantec Internet Security Threat Report, Trends for July-December 07, Volume XIII*. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiii_04-2008.en-us.pdf.

[2] Yao Wen Huang, Fang Yu, Christian Hang, ChungHung Tsai, D T Lee, and SyYen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, pages 40–52, 2004.

[3] Harold F Tipton and Micki Krause. *Information Security Management Handbook, Sixth Edition*. Auerbach Publications, New York, 2006.

[4] Open Web Application Security Project. *Top 10 2007*. http://www.owasp.org/index.php/Top_10_2007.

[5] Hui Liu and Hee Beng Kuan Tan. Automated verification and test case generation for input validation. In *Proceedings of the 1st International Workshop on Automation of Software Test, at 28th International Conference on Software Engineering*, pages 9–14, 2006.

[6] Wei Xu, Sandeep Bhatkar, and R Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, May 2005. http://seclab.cs.sunysb.edu/seclab/pubs/papers/seclab-05-04.pdf.

[7] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. Technical Report MIT-CSAIL-TR-2008-006, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, February 6, 2008.

[8] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *Proceedings of 15th International Symposium on Software Reliability Engineering*, pages 187–197, 2004.

[9] William M. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[10] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[11] Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES 03)*, volume 2931 of *LNCS*, pages 60–69, October 2003.

[12] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE 04)*, pages 196–205, September 2004.

[13] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005.

[14] Hai Yuan and Tao Xie. Substra: A framework for automatic generation of integration tests. In *Proc. 1st Workshop on Automation of Software Test (AST 2006)*, pages 64–70, May 2006.

[15] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 59–68, September 2006.

[16] Yuanfang Cai, Sunny Huynh, and Tao Xie. A framework and tool supports for testing modularity of software design. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 441–444, November 2007.

[17] Evan Martin, Suranjana Basu, and Tao Xie. Automated testing and response analysis of web services. In *Proc. the IEEE International Conference on Web Services (ICWS 2007), Application Services and Industry Track*, pages 647–654, July 2007.

[18] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proc. 16th International Conference on World Wide Web (WWW 2007)*, pages 667–676, May 2007.

[19] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Method-sequence exploration for automated unit testing of object-oriented programs. In *Presented at Workshop on State-Space Exploration for Automated Testing (SSEAT 2008)*, July 2008.

[20] Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 425–428, November 2007.

[21] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[22] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. Technical Report MSR-TR-2008-123, Microsoft Research, Redmond, WA, September 2008.

[23] Mark Harman, Fayezin Islam, Tao Xie, and Stefan Wappler. Automated test data generation for aspect-oriented programs. In *Proc. 8th International Conference on Aspect-Oriented Software Development (AOSD 2009)*, March 2009.

[24] Prasanth Anbalagan and Tao Xie. Automated generation of pointcut mutants for testing pointcuts in aspectj programs. In *Proc. IEEE International Conference on Software Reliability Engineering (ISSRE 2008)*, November 2008.

[25] JeeHyun Hwang, Tao Xie, Fei Chen, and Alex X. Liu. Systematic structural testing of firewall policies. In *Proc. SRDS*, 2008.

[26] Tao Xie and David Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, October 2005.

[27] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, pages 380–403, July 2006.

[28] Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. Towards a framework for differential unit testing of object-oriented programs. In *Proc. AST*, pages 5–11, May 2007.

[29] Shan-Shan Hou, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. Applying interface-contract mutation in regression testing of component-based software. In *Proc. 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 174–183, October 2007.

[30] Kunal Taneja and Tao Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, 2008.

[31] Alessandro Orso and Tao Xie. BERT: Behavioral regression testing. In *Proc. International Workshop on Dynamic Analysis (WODA 2008)*, July 2008.

[32] Nuo Li, JeeHyun Hwang, and Tao Xie. Multiple-implementation testing for XACML implementations. In *Proc. Workshop on Testing, Analysis and Verification of Web Software (TAV-WEB 2008)*, pages 27–33, July 2008.

[33] Shan-Shan Hou, Lu Zhang, Tao Xie, and Jia-Su Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *Proc. IEEE International Conference on Software Maintenance (ICSM 2008)*, October 2008.

[34] Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods (ICFEM 2004)*, pages 290–305, November 2004.

[35] Amir Michail and Tao Xie. Helping users avoid bugs in GUI applications. In *Proc. 27th International Conference on Software Engineering (ICSE 05)*, pages 107–116, May 2005.

[36] Tao Xie, Evan Martin, and Hai Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *Proc. 28th International Conference on Software Engineering (ICSE 2006), Informal Research Demonstrations*, pages 835–838, May 2006.

[37] Yoonki Song, Suresh Thummalapenta, and Tao Xie. UnitPlus: Assisting developer testing in Eclipse. In *Proc. Eclipse Technology eXchange Workshop at OOPSLA 2007 (ETX 2007)*, October 2007.

[38] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, 17(2):345–371, April 2008.

[39] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proc. 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 54–57, May 2006.

[40] Prasanth Anbalagan and Tao Xie. Automated inference of pointcuts in aspect-oriented refactoring. In *Proc. 29th International Conference on Software Engineering (ICSE 2007)*, pages 127–136, May 2007.

[41] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 377–380, November 2007.

[42] Mithun Acharya, Tao Xie, and Jun Xu. Mining interface specifications for generating checkable robustness properties. In *Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, pages 311–320, November 2006.

[43] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. 30th International Conference on Software Engineering (ICSE 2008)*, pages 461–470, May 2008.

[44] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, pages 25–34, September 2007.

[45] Tao Xie, Mithun Acharya, Suresh Thummalapenta, and Kunal Taneja. Improving software reliability and productivity via mining program source code. In *Proc. the NSF Next Generation Software Program Workshop at IPDPS 2008 (NSFNGS 2008)*, pages 1–5, April 2008.

[46] Suresh Thummalapenta and Tao Xie. NEGWeb: Detecting neglected conditions via mining programming rules from open source code. In *Presented as a Student Poster at International Symposium on Software Testing and Analysis (ISSTA 2008)*, July 2008.

[47] Suresh Thummalapenta and Tao Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 204–213, November 2007.

[48] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots via mining open source repositories on the web. In *Proc. 5th Working Conference on Mining Software Repositories (MSR 2008)*, pages 109–112, May 2008.

[49] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[50] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Locating need-to-translate constant strings for software internationalization. In *Proc. 31th International Conference on Software Engineering (ICSE 2009)*, May 2009.

[51] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as conditional association rules. In *Proc. 31th International Conference on Software Engineering (ICSE 2009)*, May 2009.

[52] Mithun Acharya and Tao Xie. Mining API error-handling specifications from source code. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, March 2009.

[53] Evan Martin, Tao Xie, and Ting Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. ICICS*, pages 139–158, 2006.

[54] Evan Martin and Tao Xie. Inferring access-control policy properties via machine learning. In *Proc. 7th IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, pages 235–238, June 2006.

[55] Evan Martin and Tao Xie. Automated test generation for access control policies. In *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, November 2006.

[56] Evan Martin and Tao Xie. Automated test generation for access control policies via change-impact analysis. In *Proc. 3rd International Workshop on Software Engineering for Secure Systems (SESS 2007)*, pages 5–11, May 2007.

[57] Vincent C. Hu, Evan Martin, JeeHyun Hwang, and Tao Xie. Conformance checking of access control policies specified in XACML. In *Proc. 1st IEEE International Workshop on Security in Software Engineering (IWSSE 2007)*, pages 275–280, July 2007.

[58] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. XEngine: A fast and scalable XACML policy evaluation engine. In *Proc. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*, pages 265–276, June 2008.

[59] Evan Martin, JeeHyun Hwang, Tao Xie, and Vincent Hu. Assessing quality of policy properties in verification of access control policies. In *Proc. Annual Computer Security Applications Conference (ACSAC 2008)*, December 2008.

[60] Yonghee Shin, Laurie Williams, and Tao Xie. SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis. In *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, November 2006.

[61] North Carolina State University Center for Advanced Computing and Communcation (CACC): an NSF Industry/University Cooperative Research Center, Accessed on March 2008. `http://www.cacc.ncsu.edu/`.

[62] Microsoft Research, Pex: Dynamic Analysis and Test Generation for .NET, 2008. `http://research.microsoft.com/Pex/`.

[63] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[64] Microsoft CodePlex open source project hosting web site. `http://www.codeplex.com/`.

[65] Sourceforge open source project hosting web site. `http://sourceforge.net/`.

[66] Open seminar for software engineering. `http://www.openseminar.org/se/`.

[67] Pex extensions. `http://www.codeplex.com/Pex`.

[68] Wiki site for teaching parameterized unit testing/Pex. `http://sites.google.com/site/teachpex/Home`.

[69] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 271–284, 2005.

[70] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM.

[71] Liming Chen and A Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. FTCS*, pages 3–9. IEEE Computer Society, 1978.

[72] John C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[73] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John J. P. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, 1991.

[74] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Transactions on Software Engineering*, 16(2):238–247, 1990.

[75] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[76] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *Proc. TestCom*, pages 19–38, 2006.

[77] Alex Groce, Gerard J. Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Proc. ICSE*, pages 621–631, 2007.

[78] Robert B. Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *Proc. ESEC-FSE*, pages 549–552, 2007.

[79] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

[80] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proc. 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[81] Sen Koushik, Marinov Darko, and Agha Gul. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, Lisbon, Portugal, 2005.

[82] Krugle - code search for developers, 2008. `http://www.krugle.org/`.

[83] Google code search, 2008. `http://www.google.com/codesearch`.

[84] Aske Simon Christensen, Anders Mller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS*, pages 1–18, 2003.

[85] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *Proc. SPIN*, pages 306–324, 2008.

[86] Paul Ammann and Jeff Offutt. Introduction to software testing, 2008. `http://cs.gmu.edu/~offutt/softwaretest/`.

[87] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, 2001.

[88] Anand V. Raman and Jon D. Patrick. The sk-strings method for inferring PFSA. In *Proc. Workshop on Automata Induction, Grammatical Inference and Language Acquisition*, 1997.

[89] Jian Pei, Haixun Wang, Jian Liu, Ke Wang, Jianyong Wang, and Philip Yu. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering*, 18(11):1467–1481, 2006.

[90] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–233, 1980.

[91] Common attack pattern enumeration and classification, 2006. http://capec.mitre.org/.

[92] Rick Kuhn, Yu Lei, Raghu Kacker, and Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, 2008.

[93] Openjf - open java forum, 2006. http://sourceforge.net/projects/openjf/.

[94] Laurie Williams, Andy Meneely, Ben Smith, and Sarah Smith. itrust: Role-based healthcare, Sept 2006. http://agile.csc.ncsu.edu/iTrust/.

[95] Regexlib.com, 2008. http://regexlib.com/.

[96] Tao Xie and David Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *Proc. 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)*, pages 277–287, November 2005.

[97] Tao Xie and David Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.

[98] Tao Xie and David Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems at ACM SIGSOFT 2004/FSE-12 (SAVCBS 2004)*, pages 39–46, October 2004.

[99] Hai Yuan and Tao Xie. Automatic extraction of abstract-object-state machines based on branch coverage. In *Proc. 1st International Workshop on Reverse Engineering To Requirements at WCRE 2005 (RETR 2005)*, pages 5–11, November 2005.

[100] Evan Martin, Suranjana Basu, and Tao Xie. WebSob: A tool for robustness testing of web services. In *Proc. 29th International Conference on Software Engineering (ICSE 2007), Companion Volume, Informal Research Demonstrations*, pages 65–66, May 2007.

[101] Tao Xie and Jianjun Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *Proc. 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 190–201, March 2006.

[102] Tao Xie, Jianjun Zhao, Darko Marinov, and David Notkin. Detecting redundant unit tests for AspectJ programs. In *Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, pages 179–188, November 2006.

[103] Jianjun Zhao, Tao Xie, and Nan Li. Towards regression test selection for aspect-oriented programs. In *Proc. 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP 2006)*, pages 21–26, July 2006.

[104] Prasanth Anbalagan and Tao Xie. APTE: Automated pointcut testing for AspectJ programs. In *Proc. 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP 2006)*, pages 27–32, July 2006.

[105] Prasanth Anbalagan and Tao Xie. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In *Proc. 2nd Workshop on Mutation Analysis (MUTATION 2006)*, pages 51–56, November 2006.

[106] Tao Xie and Jianjun Zhao. Perspectives on automated testing of aspect-oriented programs. In *Proc. 3rd Workshop on Testing Aspect-Oriented Programs (WTAOP 2007)*, pages 7–12, March 2007.

[107] Evan Martin and Tao Xie. Understanding software application interfaces via string analysis. In *Proc. 28th International Conference on Software Engineering (ICSE 2006), Emerging Results Track*, pages 901–904, May 2006.

[108] Mithun Acharya, Tanu Sharma, Jun Xu, and Tao Xie. Effective generation of interface robustness properties for static analysis. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 293–296, September 2006.

[109] Suresh Thummalapenta and Tao Xie. NEGWeb: Static defect detection via searching billions of lines of open source code. Technical Report TR-2007-24, North Carolina State University Department of Computer Science, Raleigh, NC, August 2007.

[110] Vincent Hu, Richard Kuhn, and Tao Xie. Property verification for generic access control models. In *Proc. IEEE/IFIP International Symposium on Trust, Security and Privacy for Pervasive Applications (TSP 2008)*, December 2008.

[111] Lingshuang Shao, Lu Zhang, Tao Xie, Junfeng Zhao, Bing Xie, and Hong Mei. Dynamic availability estimation for service selection based on status identification. In *Proc. IEEE International Conference on Web Services (ICWS 2008), Application Services and Industry Track*, September 2008.