Automated Verification and Test Case Generation for Input Validation

Hui Liu
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore 639798
liuh0007@ntu.edu.sg

Hee Beng Kuan Tan
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore 639798
ibktan@ntu.edu.sg

ABSTRACT
Input validation is essential for any software that deals with input from its external environment. It forms a major part of such software that has intensive interaction with its environment. Through the integration of invariant and empirical properties for implementing input validation, this paper proposes a novel approach for the automation of the following tasks from processing the source code of a program: (1) verification of existence of input validation; (2) generation of test cases to test and demonstrate all the input validations; (3) classification of each validation into the various types defined along with its test case generated. All the empirical properties in the theory have been validated statistically based on open source systems. Our evaluation shows that the proposed approach can help in both testing of input validation features and verifying the adequacy of input control.

Categories and Subject Descriptors
D.2.4 **[Software Engineering]**: Software/Program Verification − *validation*; D.2.5 **[Software Engineering]**: Testing and Debugging − *testing tools*.

General Terms
Verification

Keywords
Input validation, software verification, software testing, empirical-based property.

1. INTRODUCTION
Most software including Web applications processes inputs submitted from its external environment. In software, system constraints are usually required to be enforced through input validation − rejecting inputs that do not satisfy the required conditions. For example, to ensure that the outstanding balance of a customer does not exceed its credit limit, an order processing program must reject any delivery of new order placed by the customer that will lead to the exceeding of the limit.

AST'06, May 23, 2006, Shanghai, China.

As stated in the design guidelines for secure Web applications in MSDN, Microsoft's essential online resource for developers [11], proper input validation is one of the strongest measures of defense against today's application attacks. Input validation is a challenging issue and the primary burden of a solution falls on application developers.

Input validation has always been playing an important role in the control and accuracy of external input to software. Coupled with the above-mentioned role on the prevention of application attacks, surely, it is a main concern in Computer Auditing. The auditing of software to verify that sufficient input validation feature is incorporated is an important task in Computer Auditing.

We have discovered some invariant and empirical properties in the control flow between statements for submitting external inputs and statements for raising external effects. Through the use of these properties, this paper proposes a theory and consequently an approach for automated verification and test case generation of input validation from source code.

The paper is organized as follows. Section 2 presents the theory for inferring input validation. Section 3 discusses the use of the theory to generate test cases for testing input validation. Section 4 discusses the proposed approach. Section 5 reports our evaluation. Section 6 compares our work with related work. Section 7 concludes the paper.

2. A THEORY FOR INFERRING INPUT VALIDATION

2.1 Assumption and Terminology
A program that reads user inputs means to do some useful things with the inputs. Let P be the program that reads user inputs and raise some external effect, and G be the control flow graph (CFG) representing P. Each node in G represents a statement in P.

Adapted from the concept of program analysis, we define the term influence as follows: A node n_1 **influences** node n_2, or n_2 is influenced by n_1, if n_1 directly or indirectly affect the values of variables used at n_2.

A node in G at which an input is read in the program is called an **input node** in G. A node in G at which an external effect is raised is called an **effect node** in G. The effect node is defined very general here since the types of effect occurs in a software system varies much depending on the types of the system. When it is applied to a concrete type of system, the definition of effect node

can be further determined. For example, in a database application, any node that updates the database maintained for the application is an effect node.

Assume D is the set of inputs user submitted to the program P before any external effect is raised in P. Let t be the set of input nodes in G that reads data from D (in short, input t). An effect node in G that is influenced by some nodes in t is referred as an effect node on t.

Input t is said to be **validated** both the following cases are possible:

1) Some effect nodes on t are executed: This is called an **acceptance** of input.
2) No effect nodes on t are executed: This is called a **rejection** of input.

A predicate node d in G is called a **candidate prime validation node** (**c-prime validation node**) for t if there is a path from a node in t to d that does not pass through any effect nodes on t and there is a branch of d that satisfies one of the following properties:

1) It always leads to some effect nodes on t.
2) It does not lead to any effect nodes on t unless after passing through another set of input nodes at which the same inputs are submitted.

A branch that satisfies the first property is called an **acceptance branch** at d for t or more generally an **acceptance branch** for t. A branch that satisfies the second property is called a **rejection branch** at d for t or more generally a **rejection branch** for t.

```
   Begin
1  Read 5-digit card No., PIN and the amount of withdrawal M;
   Compute the check digit C through the first 4 digit of the card No.;
2  If C == the 5th digit of the card No. then
3        Validate the card No. and PIN against bank's database;
4        If the validation is confirmed then
5              Get the withdrawal limit and the available balance;
6              If M <=withdrawal limit AND
                  M <=available balance then
7                 Update the database; Dispense cash;
               else
8                 Display withdrawal error message;
               endIf
           else
9             Display invalid card-no/PIN error message;
           endIf
     else
10         Display invalid account error message;
     endIf
11   Eject card; return;
   End
```

Figure 1. A pseudocode for a simplified ATM cash withdrawal program

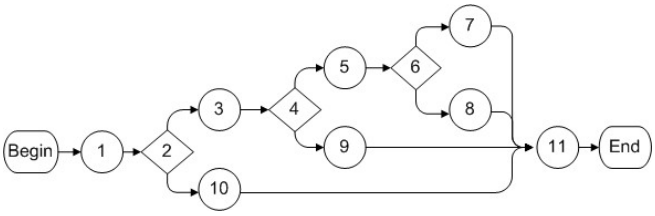

Figure 2. The CFG of the ATM cash withdrawal program

Figure 1 shows a pseudocode for implementing a simplified ATM cash withdrawal program with its basic blocks numbered. Its CFG is shown in Figure 2. Node 1 is an input node in the program. Node 7 is the only effect node in the program. Node 2, 4 and 6 are c-prime validation nodes for node 1. The branch from node 2 to node 10 is a rejection branch at node 2 for node 1. The branch from node 4 to node 9 and the branch from node 6 to node 8 are also rejection branches for node 1. The branch from node 6 to node 7 is an acceptance branch for node 1.

Let d be a c-prime validation node for input t in G. For each path from nodes in t to d that satisfies the following conditions, the sequence of all predicate nodes that influence d in the path according to the order in the path, is called a **candidate validation chain** (**c-validation chain**) at d for t:

1) It does not pass through each of these predicate nodes more than one time.
2) It does not pass through any effect node on t.

Note that d itself is included as the last node in the sequence. Furthermore, each predicate node in the sequence is called a **candidate validation node** (**c-validation node**) for t. For example, for the CFG shown in Figure 2, as discussed earlier, node 2, 4 and 6 are c-prime validation nodes for the input node, node 1. (2) is the only c-validation chain at node 2 for node 1. (2, 4) is the only c-validation chain at node 4 for node 1. (2, 4, 6) is the only c-validation chain at node 6 for node 1.

2.2 Properties for Input Validation

With the terms used in previous section, we generalize an invariant property that is sufficient for implementing input validation in a program. The proof of this property is trivial.

Property 1 – Sufficient Property for Input Validation. If there is a feasible acceptance branch for input t and there is also a feasible rejection branch for input t, then t is validated.

In invariant property 1, if the property stated is not satisfied due to the absence of feasible acceptance branch, input t is said to be **redundant**.

For example, in the CFG of the ATM cash withdrawal program shown in Figure 2, as discussed earlier, there are altogether three rejection branches and one acceptance branch for the input node, node 1. Furthermore, it can be easily verified that all the branches are feasible. Therefore, from invariant property 1, input read at node 1 is validated.

Next, based on the concept of validation chain, we present another invariant property that is sufficient for implementing a decision in a program to accept and/or reject input. The proof of this property is straightforward.

Property 2 – Sufficient Property for Accepting and Rejecting Input. A c-validation chain at a c-prime validation node for input t implements a decision to accept and/or reject t.

For example, for the CFG shown in Figure 2, (2) is a c-validation chain at node 2 for input node 1. (2, 4) is a c-validation chain at node 4 for input node 1. (2, 4, 6) is a c-validation chain at node 6 for input node 1. Each validation chain implements a decision to accept and/or reject input submitted at input node 1. In particular, (2, 4, 6) implements the decision that if the check digit is correct and the card and PIN number are also both correct, then if the withdrawal amount is less than or equal to the withdrawal limit

and available balance, then the cash withdrawal request is accepted and processed accordingly, else, the request is rejected.

As theoretically invariant property 1 may not be necessary for implementing input validation, we cannot base on it to infer whether a program implements input validation. Fortunately, empirically, we have discovered that it is highly probable that the property is a necessary property for implementing input validation. Similarly, as theoretically we cannot base on invariant property 2 to find all decisions in a program to accept and/or reject input, we thus examine it empirically. We have discovered that it is a highly probable that this property is a necessary property for implementing a decision in a program to accept and/or reject input. Next, we shall state the two empirical properties.

Property 1 – Necessary Property for Input Validation. If input t is validated, then it is highly probable that there is a feasible acceptance branch and also a feasible rejection branch for t.

Property 2 – Necessary Property for Accepting and Rejecting Input. Any decision in a program to accept and/or reject input t is implemented by a c-validation chain at a c-prime validation node for t.

By collecting samples from open source systems, we have validated the two properties statistically using hypothesis testing based on binomial test. For empirical property 1, the null hypothesis H_0 states that empirical property 1 holds for less than 99 percent of the cases, and the alternative hypothesis H_1 states that empirical property 1 holds for equal or more than 99 percent of the cases. Based on 0.05 as the type I error probability, if $z > 1.645$, we reject H_0; otherwise, we accept H_0.

We randomly picked up 8 open source projects from different open source websites [4, 14, 15] and included all the server programs in each project for processing. In addition, we randomly examined 201 programs from 80 open source projects downloaded from Sourceforge.net website [15]. Table 1 lists the details of the sample drawn.

Table 1. Samples for testing the empirical properties

	Systems	Empirical Property 1	Empirical Property 2
		# cases	# cases
1	Roomba [15]	16	27
2	Jewels [14]	10	10
3	Smacs [15]	24	36
4	bugtracker [15]	12	26
5	JspShop [4]	29	93
6	JavaLibrary [15]	22	38
7	NMS [4]	12	21
8	studentRecord [4]	24	43
9	randomly selected from Sourceforge.net [15]	201	256
	TOTAL	350	550

As shown in Table 1, the size of the sample for testing empirical property 1 is 350. All the cases in the sample gave affirmative support to the property. The z-score ($\frac{X/n - p}{\sqrt{(p(1-p)/n)}}$) calculated is 1.880 (note that we have p = 0.99 and X = n = 350); hence, we reject H_0 and conclude that the testing gives evidence that empirical property 1 holds for equal or more than 99 percent of the cases at 5 percent level of significance. □

Clearly, we can use invariant property 1 and empirical property 1 together to infer whether a program implements input validation. If the result is affirmative, from invariant property 1, the inference is always correct; otherwise, from empirical property 1, it is highly probable that the result is correct too.

The hypothesis testing for empirical property 2 was conducted in a similar way as the one did for empirical property 1. There are many decisions in a program to accept and/or reject input, and we randomly collected some of the decisions from each program that is previously collected for the test of empirical property 1. As shown in Table 1, the size of the sample for testing empirical property 2 is 550. All the cases in the sample gave affirmative support to the property. The z-score calculated is 2.357; hence, we conclude that the testing gives evidence that empirical property 2 hold for equal or more than 99 percent of the cases at 5 percent level of significance. □

By using invariant property 2 and empirical property 2 together, we can find all decisions in a program to accept and/or reject input. From invariant property 2, all the decisions found are always correct. From empirical property 2, there is no other decision in the program to accept or reject input.

3. TEST CASE GENERATION FOR INPUT VALIDATION

As defined in [16], a **simple predicate** is a Boolean variable or a relational expression possible with one or more NOT (~) operators. In general, a predicate node in a CFG consists of a number of simple predicates composed by Boolean operators.

The objective of testing input validation feature is to exercise all the possible conditions for accepting and rejecting input. From invariant property 2 and empirical property 2 together, it is highly probable that all the decisions in a program to accept and/or reject input t are implemented by and are only implemented by the c-validation chains at each c-prime validation node for t. Therefore, we can approach the test case generation for testing input validation according to one of the following two methods:

1) Branch-Based Input Validation Testing: This aims to cover every possible combination of values of the simple predicates in each c-prime validation node for t that leads to an execution of an acceptance branch or a rejection branch at the c-prime validation node.

2) Condition-Based Input Validation Testing: This aims to cover every possible combination of the following values for each c-validation chain (d_0, d_1, ….., $d_n = d$) at each c-prime validation node d for t, that leads to an execution of an acceptance or a rejection branch at d for t:

   a) For each j, $0 \le j \le n-1$, a combination of values of the simple predicates in d_j that leads to an execution of the branch from d_j to d_{j+1}.

   b) A combination of values of the simple predicates in d that leads to an execution of the acceptance or rejection branch respectively.

For conveniences, we shall call each such combination a **feasible validation condition** for the validation chain that leads to an execution of the branch.

Clearly, the second method subsumes the first method. The choice depends on the time and resources available. Based on invariant and empirical property 2, it is highly probable that all the test cases generated provide a complete coverage for testing input validation.

In the CFG shown in Figure 2, for testing input validation for input read at node 1, branch-based input validation testing aims to cover the following combinations of values of the simple predicates in each c-prime validation node for node 1, that lead to an execution of an acceptance or a rejection branch at the c-prime validation node:

Node 2: "(C == the 5th digit of the card No.) = false" that leads to an execution of the rejection branch from node 2 to node 10.

Node 4: "(the validation is confirmed) = false" that leads to an execution of the rejection branch from node 4 to node 9.

Node 6: "(M <= withdrawal limit) = true" and "(M <= available balance) = true" that leads to an execution of the acceptance branch from node 6 to node 7; and the following three combinations that lead to an execution of the rejection branch from node 6 to node 8: "(M <= withdrawal limit) = true" and "(M <= available balance) = false"; "(M <= withdrawal limit) = false" and "(M <= available balance) = true"; "(M <= withdrawal limit) = false" and "(M <= available balance) = false".

Condition-based input validation testing aims to cover the following feasible validation conditions for each c-validation chain at each c-prime validation node for node 1, that leads to an execution of an acceptance and a rejection branch at the c-prime validation node:

Chain (2) at node, 2: "(C == the 5th digit of the card number) = false" that leads to an execution of the rejection branch from node 2 to node 10.

Chain (2, 4) at node, 4: "(C == the 5th digit of the card number) = true" and "(the validation is confirmed) = false" that leads to an execution of the rejection branch from node 4 to node 9.

Chain (2, 4, 6) at node 6: "(C == the 5th digit of the card number) = true" and "(the validation is confirmed) = true" and "(M <= withdrawal limit) = true" and "(M <= available balance) = true" that leads to an execution of the acceptance branch from node 6 to node 7. The following three feasible validation conditions lead to an execution of the rejection branch from node 6 to node 8: "(C == the 5th digit of the card number) = true" and "(the validation is confirmed) = true" and "(M <= withdrawal limit) = true" and "(M <= available balance) = false"; "(C == the 5th digit of the card number) = true" and "(the validation is confirmed) = true" and "(M <= withdrawal limit) = false" and "(M <= available balance) = true"; "(C == the 5th digit of the card number) = true" and "(the validation is confirmed) = true" and "(M <= withdrawal limit) = false" and "(M <= available balance) = false".

Interestingly, test cases for both methods can be generated by existing automated test case generation technique for branch coverage through transformations. Next, we shall present the automated test case generation for the two methods.

Branch-based Test Case Generation for Input Validation. Let d be a c-prime validation node for input t. A test case to cover a combination of values of the simple predicates in d that leads to an execution of an acceptance or a rejection branch at d for t can be automatically generated as follows:

1) Replace the branch condition of the acceptance or rejection branch respectively with the conjunction of the values of the simple predicates set in the combination.
2) Use a test case generation technique for branch coverage to generate a test case to force through the acceptance or rejection branch respectively.

Condition-based Test Case Generation for Input Validation. Let (d_0, d_1, ….., d_n = d) be a c-validation chain at a c-prime validation node d for input t. A test case to cover a feasible validation condition for the validation chain that leads to an execution of an acceptance or rejection branch at d for t can be generated as follows:

1) For each d_j, $0 \le j \le n-1$, insert nodes immediately before d_j for the following:
2) Assign a Boolean variable, pass(d_j) to "true", to indicate the branch from d_j to d_{j+1} has been followed.
3) Assign each simple predicate in d_j according to its value set in the feasible validation condition.
4) Replace the branch condition of the acceptance or rejection branch respectively with the conjunction of the following predicates:
5) For each simple predicate s in d (note that d_n = d), the predicate, (s = its value set in the feasible validation condition).
6) For each d_j, $0 \le j \le n-1$, the predicate, (pass(d_j) = true).
7) Use a test case generation technique for branch coverage to generate a test case to force through the acceptance or rejection branch respectively.

The condition-based test case generation is based on the fact that following the method, a test case that forces through an acceptance or a rejection branch at d for t will also force through each branch from d_j to d_{j+1}, $0 \le j \le n-1$, with the values of the simple predicates in the predicate node set according to the feasible validation condition. This is due to the setting of simple predicate values in Step 1(b) and the insertion of predicates in Step 2(b) to ensure that these branches are also forced through in the branch condition.

4. THE PROPOSED APPROACH

Section 2 and 3 provide a basis to automatically verify whether input read in a program is validated. The theory also provides a basis to automatically generate test cases to test the input validation feature implemented in a system. It is also highly probable that the set of test cases generated forms a complete set of test cases for testing input validation.

Moreover, we can classify the type of validation performed as follows. Let d be a c-prime validation node for input t in the CFG G of a program. Let (d_0, d_1, ….., d_n = d) be a c-validation chain at d for t. The validation carried out by a simple predicate in a c-validation node d_j ($0 \le j \le n$) can be classified into the following types:

1) Control Check: The predicate is not influenced by any input but by some system predefined variables.
2) Domain Check: The predicate is only influenced by a single input variable. Domain check includes the validation of special strings for security purposes.

3) Intra-Input Check: The predicate is influenced by more than one input variables.

4) Input-Database Check: The predicate is influenced by both input and the data retrieved from system database.

The detailed steps for the proposed approach are described in the following. For each program in a system, first, the CFG G of the program is constructed and the set of input nodes t is identified through syntax analysis. The following steps are then performed:

Step 1: Compute the set P of c-prime validation nodes for t. First, the set E of effect nodes in G that are influenced by some input nodes in t is computed through data flow analysis. Second, for each $e \in E$, identify all the predicate nodes on the paths from t to e and put them in a set D. For each node $d \in D$, verify whether it satisfies the required properties for a c-prime validation node for t according to the definition. In the verification, for properties with regard to a path, it is sufficient to check all the basis paths between the two associated nodes. If the result is affirmative, then d is a c-prime validation node for t; hence it is put in the set P of all the c-prime validation nodes for t.

Step 2: Verify the possibility that input t is validated. Each branch at each c-prime validation node for t is checked for the following condition until the result is affirmative or the end: there is an acceptance branch and there is a rejection branch for t. If the result it is affirmative, from invariant property 1, it is possible that input t is validated and therefore proceeding to Step 3. Otherwise, from empirical property 1, input t is not validated and no further processing is needed.

Step 3: Perform one of the following two options depending on user's choice.

1) Branch-Based Testing: For each c-prime validation node d for t in P, all the acceptance and rejection branches at d for t are computed. Then, for each of these acceptance and rejection branches, for each possible combination of values of the simple predicates in d that leads to an execution of the acceptance or rejection branch respectively, Branch-Based Test Case Generation for Input Validation is applied to generate a test case to cover the combination for executing the branch. Next, the test case is annotated with the branch (with acceptance or rejection indicated) and predicate nodes passed through by executing the test case. And, for each predicate node annotated, each simple predicate in the predicate node is further annotated with its value set in the test case and its type of validation as discussed earlier.

2) Condition-Based Testing: For each c-prime validation node d for t in P, all the acceptance and rejection branches and all the c-validation chains at d for t are computed according to the definition. Then, for each c-validation chain and each acceptance and rejection branch at d for t, for each feasible validation condition v for the c-validation chain to execute the acceptance or rejection branch respectively, Condition-Based Test Case Generation for Input Validation is applied to generate a test case to cover v for executing the acceptance or rejection branch respectively. Next, the test case is annotated with the c-validation chain and the branch (with acceptance or rejection indicated). And, for each c-validation node in the c-validation chain annotated, each simple predicate in the c-validation node is further annotated with its value set v and its type of validation as discussed earlier.

Step 4: Verify whether input t is validated. Based on the test cases generated, if there is a test case to force through an acceptance branch at a c-prime validation node for t and there is also a test case to force through a rejection branch at a c-prime validation node for t, then from invariant property 1, it is concluded that input t is validated. Otherwise, input t is not validated. For the latter case, if no test case is generated to force through any acceptance branch at a c-prime validation node for t, then input t is redundant.

The proposed approach can be applied to any program that processes input and raise effect. It can be used for the verification and testing of input validation implemented in a system. The information on existence of input validation that is computed by the proposed approach gives an overview of the input validation feature implemented. The test cases and the associated annotations generated from the proposed approach show all the details of the input validations carried out. Based on these, testing can be conducted to verify the exact correctness of input validation implemented. This can be carried out for a program or each program in a system. Thus, it can be used in any level of software testing.

5. EVALUATION

5.1 Proof-of-Concept Prototyping

A prototype system has been developed to demonstrate the feasibility of the approach proposed. The prototype system implements the propose

d approach for Web applications that are written in Java. We choose Web applications as the target system because of their popularity, importance and their increasing need of reliability.

WebApps have two ways of validation on user inputs: client-side validation and server-side validation. The importance of server-side validation in WebApps has been emphasized [11, 13] because of the vulnerability of client-side validation; thus we focuses on the analysis of server-side validation, to make sure that any user input used to raise an effect is validated at server-side regardless the validation carried out at client-side. A major type of effect raised in most of the WebApps is updating database; hence, the prototype system is configured to automatically detect the effect on database updating.

The prototype system is implemented as follows. First, we developed a Program Analyzer to analyze the source code of each program and construct its CFG. The Program Analyzer is developed using the program analysis system, Java Architecture for Bytecode Analysis (JABA) [8], which analyzes Java programs at bytecode level. By parsing a class file of a Java program, JABA builds the CFG and collects the control and data flow information for the analyzed program. Next, we developed an Input Validation Analyzer and Test Case Generator (IV-ATCG) to process the CFG, control and data flow information of the analyzed program. The IV-ATCG is implemented in accordance to the four steps described in Section 4. The technique of test data generation for branch coverage proposed in [5] is used to generate test cases. This technique is effective in detecting infeasible paths, and can handle programs with pointers, arrays and function calls.

5.2 Case Studies

5.2.1 Case Study on Verification of Input Validation

To evaluate the use of the proposed approach for the verification and testing of input validation, we conducted case studies on the eight open source systems listed in Table 1 (which are also used for part of the hypotheses testing). We did a general check on the existence of input validation in those systems using the prototype system. In total, our prototype system finds 34 programs out of 150 programs do not have input validation feature, which account for 22.67%. This statistic shows that input validation is often ignored or overlooked by developers; however, the lack of sufficient input validation can cause very serious problems (examples are discussed below).

We then apply the proposed approach in full on two systems: Roomba and Smacs [15]. Roomba is a web-based room booking system for small to medium-sized hotels. The whole system is well designed and implemented. However, in the experiment, the prototype system discovers that 6 out of 16 programs in Roomba do not have input validation feature. An example is shown in Figure 3. The program reads input at line 2, 4, 5 and 6, construct the SQL statement at line 8 and 10, and update database at line 11.

```
* No input validation: Roomba/bookings/saveBooking.jsp */
1  boolean newBooking = false;
2  if ((getParam("newBooking", request).equals("true")))
3   newBooking = true;
4   String id = getParam("id", request);
5   String customerid = getParam("customerid", "0", request);
6   String roomid = getParam("roomid", "0", request);
    …
7   if (newBooking) {
8         sql = "INSERT INTO ROOMBA Bookings(customerid,
    roomid, …) VALUES (" + customerid + ", " +roomid +… + ")";}
9   else {
10        sql = "UPDATE ROOMBABookings SET" +
                    "customerid = " + customerid + ", " + "roomid = "
                     + roomid + … +"WHERE (id = " + id + ")";}
11  DBConnector.executeUpdate(sql);
```

Figure 3. A program without input validation

It is not surprising to see many similar cases as the one shown in Figure 3, in which the program reads user inputs and uses them directly in database insertion or modification without any checking on the input values. Basically, such defect can cause two possible problems. One is that an invalid input can result in Java SQL runtime exception if the value does not comply to its entry type defined in the database schema such as the type and length of the data (e.g., inserting a string into a database where an integer is required); the other one is that it is very easy to be attacked by SQL injection which may cause security issues.

Smacs [15] is a web-based facility for the management of casual staff working in an organization. Our prototype system discovered that 8 out of 24 programs in Smacs do not have input validation feature. An interesting example is shown in Figure 4. In this case, *bean* is an instance of the class *TestingCard* as in line 1. The program checks the input values at line 2 and 4 and set the corresponding field of *bean* if the value is not null. However, although a domain checking is performed on the input, the method *bean.create()* at the line 6 will always be executed regardless of the results of the checking. This program may confuse the users who provide the invalid input; it may also cause the insertion of invalid records into the database.

```
/*No input validation: Smacs/ insertTestingCard.jsp*/
1  <jsp:useBean class="casualstaffhr.TestingCard" id="bean" />
…
2  if ( request.getParameter( "number" ) != null ){
3     bean.setNumber( request.getParameter( "number" ) );
    }
…
4  if ( request.getParameter( "description" ) != null ){
5     bean.setDescription( request.getParameter("description") );
    }
6  bean.create();
```

Figure 4. A program without input validation

Besides the verification of existence of the input validation, we also uses the proposed approach to generate test cases for programs in Roomba and Smacs, for illustrating and verifying the input validation feature implemented. Both the condition-based and branch-based methods are applied and tested. We found that the test cases generated by the branch-based method with annotations sufficiently show the validation features implemented in the systems. The condition-based method tends to generate more test cases for showing the different combinations of conditions.

In summary, through the case study on open source systems, we find that the prototype system can effectively detect programs without input validation. The lack of input validation in many programs could cause serious problems such as program runtime exception, security control, violation of database integrity. We also observe that the test cases with annotation generated by branch-based method are sufficient to illustrate the input validation features implemented.

5.2.2 Case Study on Testing Input Validation

To experience the feasibility of using the proposed approach to aid input validation testing in software development, we conducted a case study on a student project EAuction. EAuction is a web-based auction system that developed by senior computer science students. It provides basic functionalities for hosting online auctions where registered users can both post and bid on items.

For testing the input validation feature in integration testing of the system, we divided the students into two groups: the functional test case design techniques that include testing based on input structure were introduced to the first group; and the proposed testing was introduced to the second group. Interestingly, in addition to the testing of correctness on input validation, we observes that the test cases generated by the condition-based method in the proposed approach also helps to discover complex logic errors among the conditions. It was clear that the group that used the proposed approach tested the input validation more adequately than the other group. We also observed that the group that used the proposed approach generated test cases much systematically and faster than the other group.

6. RELATED WORK

Two techniques for testing of input validation have been proposed [6, 13]. Both of the techniques attempt to violate input specifications in the design of test cases. The technique proposed

in [6] is for input validation analysis and testing of systems that take inputs that can be represented in grammars. The bypass testing proposed in [13] is specifically for testing Web applications. Though the proposed approach shares the objective of verifying and testing the input validation with these techniques, the ways that they address the problem differ vastly from the proposed approach. The proposed approach verifies input validation features and generates test cases for testing these features through analyzing the CFGs of programs that process inputs. As a result, it lands itself as an automated method. The two related techniques are solely based on the analysis of the structure and syntax of the inputs. They are not fully automated.

Software testing is a well research area. Much work has been devoted to structural testing [5, 10, 17], specification based testing [2-3, 9], implementation-based testing [1, 7]. Recently, there has been some interest in using software architecture for code testing [12]. In terms of software testing, the proposed technique shares with the structural technique on the automated generation of test cases. It shares with the specification based testing and software architecture based testing on the use of some kind of model for code testing. In opposing to these approaches, the proposed approach recovers the model automatically from source code for implementing input validation. Based on the model discovered, it automatically generates test cases to test the validation. No specification of model is required. It is for testing a type of features – input validation. So, from the automated test case generation perspective, it can be viewed as a case of feature oriented automated test case generation method. Therefore, it can be viewed as a case on applying feature oriented Software Engineering [18] in automated test case generation.

From the theoretical perspective, the proposed approach is developed through the integration of invariant and empirical properties together as a basis for the automated verification and test case generation for input validation. All the empirical properties have been validation statistically. The use of empirical properties that have been statistically validated is very common in medicine. Though they are some use of empirical properties informally, formal statistical validation on these properties is still not very common in software engineering.

7. CONCLUSION

We have proposed an approach for the automated verification and test case generation for input validation from program source codes. Any affirmative result on the verification of existence of input validation from the proposed approach is always correct. It is highly probable that other result from the verification is also correct. All test cases generated from the proposed approach and all types of input validation reported from the proposed approach are always correct too. It is also highly probable that these test cases and types of input validation have covered all the input validation implemented. The approach can be applied to any program that processes input submitted to raise effect.

In addition to the use in automated software verification and testing, we believe that the proposed approach has a potential to be realized as a computer auditing tool to automatically verify the implementation of input validation in a system from its source code. The tool can also automatically generate test cases from source code to show how input validation is implemented. More exploration on feature-oriented automated software verification and test case generation is also a further research area.

8. ACKNOWLEDGMENTS

We would like to thank Mary Jean Harrold from Georgia Institute of Technology, for sharing the JABA program analysis tool.

9. REFERENCES

[1] Bechini, A. and Tai, K.-C., Design of a toolset for dynamic analysis of concurrent Java programs. in *Proceedings 6th International Workshop on Program Comprehension. IWPC'98, 24-26 June 1998*, (Ischia, Italy, 1998), IEEE Comput. Soc, 190-197.

[2] Cardell-Oliver, R. and Glover, T., A practical and complete algorithm for testing real-time systems. in *Formal Techniques in Real-Time and Fault-Tolerant Systems. 5th International Symposium, FTRTFT'98, 14-18 Sept. 1998*, (Lyngby, Denmark, 1998), Springer-Verlag, 251-261.

[3] Carver, R.H. and Tai, K.-C. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, *24* (6). 471-490.

[4] China Webmaster: http://code.cnzz.cn

[5] Gupta, N., Mathur, A.P. and Soffa, M.L., Generating test data for branch coverage. in *Proceedings of ASE 2000 15th IEEE International Automated Software Engineering Conference, 11-15 Sept. 2000*, (Grenoble, France, 2000), IEEE Comput. Soc, 219-227.

[6] Hayes, J.H. and Offutt, A.J. Increased software reliability through input validation analysis and testing. *Proceedings of the 10th International Symposium on Software Reliability Engineering, ISSRE'99*. 199-209.

[7] Hoffman, D. and Strooper, P., Tools and techniques for Java API testing. in *Proceedings 2000 Australian Software Engineering Conference, 28-29 April 2000*, (Canberra, ACT, Australia, 2000), IEEE Comput. Soc, 235-245.

[8] JABA, http://www.cc.gatech.edu/aristotle/Tools/jaba.html

[9] Mandrioli, D., Morasca, S. and Morzenti, A. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, *12* (4). 365-398.

[10] Mansour, N. and Salame, M. Data generation for path testing. *Software Quality Journal*, *12* (2). 121-136.

[11] MSDN, Design Guidelines for Secure Web Application: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh04.asp

[12] Muccini, H., Inverardi, P. and Bertolino, A. Using software architecture for code testing. *IEEE Transactions on Software Engineering*, *30* (3). 160-171.

[13] Offutt, J., Wu, Y., Du, X. and Huang, H., Bypass testing of Web applications. in *15th International Symposium on Software Reliability Engineering, 2-5 Nov. 2004*, (Saint-Malo, Bretagne, France, 2004), IEEE Comput. Soc, 187-197.

[14] Planet Source Code: http://www.pscode.com

[15] Sourceforge: http://sourceforge.net

[16] Tai, K.-C. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, *22* (8). 552-562.

[17] Taylor, R.N., Levine, D.L. and Kelly, C.D. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, *18* (3). 206-215.

[18] Turner, C.R., Fuggetta, A., Lavazza, L. and Wolf, A.L. A conceptual basis for feature engineering. *Journal of Systems and Software*, *49* (1). 3-1