MiTV: Multiple-Implementation Testing of User-Input Validators for Web Applications

Kunal Taneja¹ Nuo Li¹ Madhuri R. Marri¹ Tao Xie¹ Nikolai Tillmann²

¹Department of Computer Science, North Carolina State University, Raleigh

²Microsoft Research, One Microsoft Way, Redmond

1 {ktaneja, nli3, mrmarri, txie}@ncsu.edu, 2nikolait@microsoft.com

ABSTRACT

User-input validators play an essential role in guarding a web application against application-level attacks. Hence, the security of the web application can be compromised by defective validators. To detect defects in validators, testing is one of the most commonly used methodologies. Testing can be performed by manually writing test inputs and oracles, but this manual process is often laborintensive and ineffective. On the other hand, automated test generators cannot generate test oracles in the absence of specifications, which are often not available in practice. To address this issue in testing validators, we propose a novel approach, called MiTV, that applies Multiple-implementation Testing for Validators, i.e., comparing the behavior of a validator under test with other validators of the same type. These other validators of the same type can be collected from either open or proprietary source code repositories. To show the effectiveness of MiTV, we applied MiTV on 53 different validators (of 6 common types) for web applications. Our results show that MiTV detected real defects in 70% of the validators.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Testing tools

General Terms

Security, Reliability

1. INTRODUCTION

User-input validators are the first barricade that protects a web application from application-level attacks such as buffer overflow, code-injection attack, hidden-field manipulation, and cross-site scripting [5]. When an attacker launches such attacks by sending malicious inputs to a web application, these malicious inputs are identified and filtered by the user-input validators. If the user-input validators are defective, then the validators may not filter these malicious inputs. As a result, the underlying web application may be

*This work is supported in part by NSF grants CNS-0720641, CCF-0725190, CCF-0845272, CNS-0958235, an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'10, September 20–24, 2010, Antwerp, Belgium. Copyright 2010 ACM 978-1-4503-0116-9/10/09 ...\$5.00. vulnerable to attacks. As shown in a recent survey¹, validators are often defective: among the top 10 vulnerabilities of web applications, 6 vulnerabilities are induced by defective validators.

To improve the quality of validators, a common approach is to use manual testing. In manual testing, developers write test inputs and expected test outputs as oracles to check actual test outputs. However, these manually-written test inputs are usually not sufficient in exposing defects because the developers might not be able to construct specific strings that can detect defects in these validators. To improve the defect-detection capability of manuallywritten test inputs, various existing automated approaches [8, 3] can automatically generate test inputs. However, these approaches require specifications to generate test oracles for checking actual test outputs. Writing specifications manually for a validator is often as error prone as implementing the validator. For example, some email validators use regular expressions (RegEx) to validate user inputs. Writing specifications for such validators is equivalent to writing these RegExs. Moreover, inputs for some validators such as credit-card validators cannot be solely represented as RegExs because there can be some semantic constraints (such as checksum on the digits of a credit card) involved in representing these inputs.

To automatically generate test inputs and oracles without requiring specifications, we identify and leverage three main specific characteristics of input validators. *First*, different from other programs, most user-input validators take strings as inputs and return a boolean value to indicate whether an input string passes the checking of a user-input validator². *Second*, the functionalities of a particular type of user-input validators are similar (such as validation of credit card numbers). *Third*, multiple implementations of a validator type³ are commonly available on the Internet.

Based on these characteristics of user-input validators, we propose a novel approach, called MiTV, which applies multiple-implementation testing⁴(MIT) to test a validator. In particular, MiTV compares the behavior of the validator under test with other validators of the same type to provide effective tool support for generating test inputs and test oracles.

MiTV addresses two main technical challenges in applying MIT to validators:

Detecting Behavioral Differences. Existing test-generation techniques [8, 3] generate test inputs for each validator separately. However, generation of test inputs separately for each validator is insufficient to detect behavioral differences between different val-

¹http://www.owasp.org/index.php/Top_10_2007

²In the rest of this paper, we use "a validator accepts an input" to denote "an input string passes the checking of a validator".

 $^{^{3}}$ We use the notion of *validator type* to refer to the type of input validation that a validator implements.

⁴ MIT is an instance of differential testing [9] that is applied on two or more implementations of a software system.

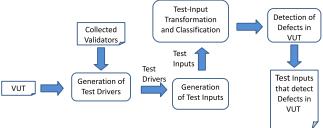


Figure 1: Overview of MiTV

```
public boolean isValid1(String value) {
   String ZIP_REG = "[0-9]{5}";
   String PLUS4_OPTIONAL = "([ -]{0,1}{0-9}{4}?";
   Pattern mask= Pattern.compile(ZIP_REG+PLUS4_OPTIONAL);
   Matcher matcher = mask.matcher(value);
   if(!matcher.matches()) return false;
   else return true; }
```

Figure 2: Zip-code validator under test.

idators [10]. To address the preceding challenge, we design test drivers that guide a test generator to generate tests that reveal different behaviors for each pair of validators of the same type.

Different Validator Requirements. In traditional MIT [6], all the implementations of a software system have the same requirements. Having the same requirements enables developers to detect defective implementations based on whether an implementation behaves differently from (a majority of) the other implementations. However, in practice, requirements for different validators of the same type can be slightly different. To address the preceding challenge, we propose an input-transformation technique for MIT of software systems that have different requirements. Our technique transforms the input strings accepted (and rejected) by a validator v_1 of a validator type to a string in format required by validator v_2 of the same type as v_1 . The transformation facilitates the comparison of behaviors of the validators v_1 and v_2 .

2. APPROACH

In this section, we present details of MiTV and illustrate the approach using an example. Figure 2 shows a zip-code validator and Figure 3 shows three other zip-code validators, all of which are adapted from real validators collected from the Internet. Suppose that isValid1 is the validator under test. isValid1 uses a RegEx to validate whether an input string consists of either five or nine digits. If the string consists of nine digits, there may exist a separator between the first five digits and the last four digits. We want to define that the value of the zip code should be greater than 00501. However, the validator isValid1 is defective since it does not include this restriction on the input values, while the other three validators check for such restriction. Without requiring manually provided test oracles, MiTV detects this defect by comparing isValid1 and the other three validators (isValid2, isValid3, and isValid4 shown in Figure 3). Figure 1 gives an overview of MiTV, which uses three main techniques to effectively detect defective validators.

TEST DRIVER SYNTHESIS. First, MiTV includes the driver-synthesis technique to synthesize drivers for all possible pairs of the validators <code>isValid1</code>, <code>isValid2</code>, <code>isValid3</code>, and <code>isValid4</code>. Figure 4 shows an example test driver. The <code>compare</code> method is a test driver for the <code>isValid1</code> and <code>isValid2</code> validators. Each of the six test drivers (similar to <code>compare</code> but invoking different validator pairs) invokes two zip-code validators in three conditionals to check their behaviors for the same test inputs on both the validators.

GENERATION OF TEST INPUTS. Second, MiTV uses a test generator to automatically generate test inputs for each test driver.

```
public boolean isValid2(String zipcode) {
 boolean isValidZip = zipcode.length() == 5;
  trv{
   int zip = Integer.parseInt(zipcode);
   isValidZip=isValidZip && ((zip>=501)&&(zip<=99999));
  }catch (NumberFormatException e) {
   isValidZip=false;
  }return isValidZip; }
public boolean isValid3(String value) {
  String s = 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]
  for(int i=0; i<PATTERNS.length; i++) {</pre>
     Matcher m = PATTERNS[i].matcher(value);
     if (!m.matches()) return false; }
  if(int.Parse(s) < 501) return false;
return true; }
public boolean isValid4(String s) {
  String zipTemplate="[0-9]{5}[-][0-9]{4}";
  Matcher m = Pattern.compile(zipTemplate).matcher(s);
  if (!m.matches()) return false;
  String[] sArr=s.Contains('-') ? s.Split(new char[] '-'):
                s.Split(new char[] '');
  if(int.Parse(sArr[0]) < 501) return false;
  return true; }
```

Figure 3: Zip-code validators used to test isValid1.

```
public String compare(String s) {
  String case = "outside V1 and V2";
  if(v1.isValid1(s) && !v2.isValid2(s))
    case = "in V1 but outside V2";
  if(!v1.isValid1(s) && v2.isValid2(s))
    case = "in V2 but outside V1";
  if(v1.isValid1(s) && v2.isValid2(s)) case="in V1 & V2";}
```

Figure 4: An example of test driver.

The test generator used by MiTV should work in such a way that the test generator tries to generate test inputs to cover as many branches as possible in the code under test. Therefore, if the test generator can generate a test input that covers a particular branch in a test driver, there exists a test input that satisfies the condition of this branch, and the value of case (in Figure 4) is assigned with a string indicating the condition of this branch. In our implementation, we use Pex [3] as our test generator. Pex is an automatic test-generation tool (developed at Microsoft Research) for .NET programs and generates test inputs (based on path exploration) that achieve high structural coverage such as branch coverage.

CLASSIFICATION. Third, MiTV automatically classifies a test input as valid (invalid) if the test input is accepted (not accepted) by a majority (e.g., more than 50%) of the validators under consideration including both the validator under test and other collected validators. Note that before classifying the inputs as valid or invalid, we use transformation functions to transform the inputs in the format required by one validator to inputs in the format required by another validator. For input transformation, we first match a generated input, say s_i , with a validator, say v_{s_i} , such that s_i is accepted by v_{s_i} . Our intuition is that if the input s_i is accepted by v_{s_i} , s_i is likely to be in the format of v_{s_s} , i.e., if 111-11-1111 is accepted by v_{s_i} , we consider the input 111-11-1111 to be in the format of v_{s_i} . Now, for any other validator v that rejects s_i , s_i is transformed to the format required by v using an input-transformation function. In the zip-code example, an input 12345 generated for isValid2 is transformed into 12345-0000 for the other validators, which require the inputs to contain 9-digit zip code with a separator. Hence, the input 12345 is classified as valid since the transformed input is accepted by all the validators. On the other hand, we determine that a generated test input "00076|0000" is invalid, because

"00076|0000" is accepted by only isValid1 (the other three validators do not accept this input since the value is less than 00501, violating the restriction on the input).

We then detect the validator under test as defective if it accepts an invalid input or rejects a valid input. Thus, in our example, we detect that isValid1 is a defective validator.

3. EVALUATIONS

To evaluate the effectiveness of MiTV, we applied MiTV on 53 validators of 6 different types. Specifically, through our evaluations, we address the following research questions:

- RQ1: How effectively does MiTV classify the generated test inputs as valid or invalid?
- RQ2: How effectively do the classified test inputs detect defective validators?
- RQ3: How high is the defect-detection capability of test inputs generated by MiTV compared to existing testgeneration approaches?

3.1 Subjects

We apply MiTV on 53 validators of 6 different validator types including validation of credit card number, email address, phone number, SSN, URL, and zip code. We collected different validators of the same type from Krugle [2] and Google Code Search [1] by giving validator types (e.g., email validator) as keywords. As we use Pex [3] for input generation, our current implementation can be used for testing validators written in .NET languages such as C#. Some validator implementations that we collected (using code search engines) were implemented in Java. We translated these Java validators into C# using Microsoft Visual Studio. The collected validators are implemented in 160 C# files with 19,929 lines of code (including comments and blank lines) in total. These validators are from a variety of sources. Due to space limit, we do not list here the sources of all of our subject validators, which can be found at our project website [4].

3.2 Setup and Metrics

In our evaluations, we treat each collected validator as the validator under test (VUT) one at a time. In particular, we apply the first two techniques of our approach (i.e., synthesis of test drivers and generation of test inputs) on all the validators at the same time, since they are independent of which validator we treat as the VUT. After finishing the first two techniques, we apply the third technique, i.e., classification of test inputs (using results of the second technique), on each collected validator to detect defects in the validator by treating the validator as the VUT.

Manual Input Classification. To measure the accuracy of testinput classification, we manually analyze all the generated test inputs to verify the results of classification produced by MiTV. To ensure that our manual classification is not biased, we use certain guidelines to determine whether a generated test input is valid or invalid. We make these guidelines by referring to various sources that provide information on the format of a valid input for a validator type. For example, we refer to the RFC 5211⁵ document to make guidelines to determine whether a generated email test input is valid or invalid. The complete set of guidelines that we used are available on our project website [4].

Test Generation Using Pex. To address RQ3, we compare the effectiveness of test inputs generated using MiTV and Pex [3] (as a representative of existing test-generation approaches). We generate a test suite using Pex for each validator and insert oracles in the

Table 1: Generated inputs that are accepted by at least one validator.

Type	N	Inputs				Cov	T(m)
		#Tot	#I	#V	InC		
Credit	11	164	127	37	5%	95%	97
Email	12	275	262	13	2%	88%	107
Phone	7	123	118	5	6%	88%	28
SSN	8	45	31	14	2%	92%	35
URL	6	45	42	3	9%	86%	27
Zip	9	73	63	10	6%	91%	29
Total	53	715	643	72	4%	90%	323

generated test suite by manual analysis of the test inputs. We refer to the preceding approach of using Pex and manual oracles as the "Pex approach" in the rest of this paper.

Input Transformers. For validators (of the same validator type) with different requirements on the input strings, we implemented the input transformation functions. We inferred the requirements of a validator by manually inspecting the generated inputs that were accepted and rejected by the validator. In particular, we wrote input transformers for phone-number, SSN, and zip-code validators. The requirements of validators of the other validator types were the same. It took around 4 hours for the first author to write and test these transformer functions. These transformer functions consist of around 400 C# lines of code.

Metrics. To compare MiTV with other approaches, we use two standard metrics from the Information Retrieval field: Precision and Recall.

3.3 RQ1: Input Classification

In this section, we address the research question RQ1 of whether MiTV is effective in classifying generated test inputs as valid or invalid. Table 1 shows the number of test inputs (generated by MiTV) that were accepted by at least one validator. Column Type shows the validator types. Column N shows the number of validators that we collected. Column #Tot shows the total number of inputs generated by MiTV. Columns #I and #V show the number of inputs that MiTV classified as invalid and valid, respectively. We manually inspected all the test inputs generated by MiTV (as described in Section 3.2) and identified the number of inputs that are incorrectly classified as valid or invalid. Column InC shows the percentage of inputs that are incorrectly classified. Column Cov shows the average branch coverage achieved by the generated test inputs for each validator type. The last column, Column T(m), shows the time taken by Pex (in minutes) to generate these inputs. In total, MiTV generated 715 different test inputs that were accepted by at least one validator. These inputs achieved average branch coverage of 90%. Out of these inputs, MiTV classified 643 as invalid and 72 as valid. MiTV correctly classified 96% of the 715 inputs as valid or invalid. We observed that most of the incorrectly classified inputs are invalid according to their specifications. For example, a zip code containing all zeros is invalid (since the lowest zipcode is 00501^{6}).

3.4 RQ2: Defect Detection

In this section, we address the research question RQ2 of whether the classified test inputs effectively detect defective validators. Initially, we use the manually classified invalid test inputs to detect defective validators. If a validator accepts a manually classified invalid test input, the validator is classified as defective validator. We use these defective validators as golden defective validators to detect the false positives and false negatives of the defective valida-

⁵http://tools.ietf.org/html/rfc5322\#page-16

⁶http://en.allexperts.com/e/z/zi/zip_code.htm

Table 2: Classification of Validators.

Type	vNum	vDef	vFP	vFN	P	R
Credit	11	8	1	0	88%	100%
Email	12	10	1	0	90%	100%
Phone	7	4	0	2	100%	67%
SSN	8	6	0	1	100%	88%
URL	6	4	0	0	100%	100%
Zip	9	7	0	1	100%	88%
Total	53	39	2	4	95%	90%

tors automatically detected by MiTV. Table 2 shows the defective validators detected by MiTV. Column Type shows the validator types. Column vNum shows the number of validators. Column vDef shows the number of defective validators that are detected by MiTV. Columns vFP and vFN respectively show the number of false positives and false negatives produced by MiTV. Columns P and R respectively show the precision and recall of MiTV in detecting defective validators. To measure the effectiveness of our driver synthesis technique and our input transformation technique, we measured the effectiveness of our approach with and without each of the two techniques. Our results showed that MiTV detected 11 more defective validators with the driver synthesis technique than without the technique. On the other hand, using our inputtransformation technique, there was a reduction in the number of false positives (one each for the Zip code and the SSN validators) in finding defective validators. In total, MiTV (with both the driver synthesis and input transformation techniques) was able to detect 39 of the 53 validators as defective with only 2 false positives and 4 false negatives (as shown in Table 2). That is, MiTV achieved a high precision of 95% and a high recall of 90% in detecting defective validators. The complete list of defective validators can be found on our project website [4].

3.5 RQ3: Comparison with Existing Test-Generation Approach

In this section, we address the research question RQ3 of whether MiTV has a higher defect-detection capability compared to an existing test-generation approach. To address RQ3, we compare the effectiveness of our approach in detecting defects in the validators using MiTV compared to the Pex approach. One of the advantages of our approach is the automated generation of test oracles. For the Pex approach, we manually insert test oracles in the generated test suite. Table 3 shows the number of test inputs generated and defective validators detected by the two approaches. Column Typeshows the validator type. Columns N_M and N_P show the number of defect-detecting test inputs (i.e., invalid test inputs accepted by a validator) generated by MiTV and the Pex approach, respectively. Column N_M/N_P shows the ratio of the number of defect-detecting test inputs generated by MiTV to the number of defect-detecting test inputs generated by the Pex approach. Columns D_M and D_P show the number of defective validators detected by MiTV and the Pex approach, respectively. Column R_M (and R_P) shows the recall of MiTV (and the Pex approach) in detecting defective validators. Note that the values in Columns N_M and D_M exclude the false positives generated by MiTV. We observed that MiTV generated 3.8 times more invalid inputs accepted by at least one defective validator. In addition, MiTV detected 48% more defective validators than the Pex approach with 29% increase in recall.

In summary, our results show that MiTV detected defective validators with high a precision of 95% and a recall of 90%, and was significantly (48%) more effective in detecting defective validators when compared to the Pex approach.

Table 3: Comparison of MiTV and the Pex approach.

Type	N_M	N_P	N_M/N_P	D_M	D_P	R_M	R_P
Credit	125	35	11.3	7	4	100%	57%
Email	257	95	36.7	9	4	100%	44%
Phone	115	18	23	4	3	67%	50%
SSN	30	10	5	6	5	86%	71%
URL	38	18	5.4	4	3	100%	75%
Zip	63	11	7.9	7	6	88%	75%
Total	628	187	3.8	37	25	90%	61%

4. RELATED WORK

N-version programming [6] executes different versions of a program in parallel in the same application environment with the same inputs, and then passes the outputs to a voter. The majority outputs are treated as the correct output. While using N-version programming, we need to generate test inputs to cause and observe different outputs (if any). Knight et al. [7] generated test inputs randomly for their programs under test. In contrast, in our approach, we automatically generate test inputs with Pex for our test drivers.

DiffGen [10] detects behavioral differences between two versions of a class. DiffGen uses a test driver similar to the one used by MiTV. Different from DiffGen, our test drivers (for pairwise testing of two input validators) enable a test generator to generate test inputs specifically for each of the following categories: the first validator accepts and the second validator rejects the generated input, the second validator accepts and the first validator rejects the generated input, and both the validators accept the generated input. In contrast, DiffGen's test drivers enable a test generator to generate test inputs for which the two methods under test produce different return values. Hence, the test inputs generated using the test driver proposed in DiffGen may not produce inputs for each of the preceding categories. DiffGen's test drivers would not be as effective in helping detect defects in validators.

5. CONCLUSION

We proposed an approach, called MiTV, that applies multiple-implementation testing for testing an input validator by leveraging different validators of the same type. To generate test inputs that can detect different behaviors among validators of the same type, we synthesize test drivers for each pair of validators of the same type and use a test generator for structural testing to generate test inputs for the synthesized test drivers. We evaluated MiTV using 53 different validators (of 6 common types) collected from the Internet. Evaluation results show that MiTV detected 70% (37 of 53) of the validators as defective with high precision and recall of 95% and 90%, respectively.

6. REFERENCES

- [1] Google code search, http://www.google.com/codesearch.
- $\label{eq:code_search} \textbf{[2]} \ \ \textbf{Krugle-code search for developers}, \ \texttt{http://www.krugle.org/}.$
- [3] Pex and Moles Isolation and White box Unit Testing for .NET. http://research.microsoft.com/Pex/.
- [4] MiTV, https://sites.google.com/site/mitv2009.
- [5] K. Beaver. The importance of input validation, 2006. http://searchsoftwarequality.techtarget.com/tip/0, 289483, sid92 gcil214373, 00.html.
- [6] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. FTCS*, pages 3–9, 1978.
- [7] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE TSE*, 12(1):96–109, 1986.
- [8] H. Liu and H. B. K. Tan. Automated verification and test case generation for input validation. In *Proc. AST*, pages 9–14, 2006.
- [9] W. M. McKeeman. Differential testing for software. Digital Technical Journal of Digital Equipment Corporation, 10(1):100–107, 1998.
- [10] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In Proc. ASE, pages 407–410, 2008.