Achieving Domain Coverage with Directed Fuzz Testing

0 Abstract
In this paper I present a practical automated approach to achieving domain coverage on inputs of a program using directed fuzz testing. The technique uses a high level state machine description of the software to generate input domain classes. Thereafter, a fuzzer chooses values randomly from the input classes and tests the code using the values. Input classes are created manually according to the software requirements. Further work could automate this process as well by using artificial intelligence and machine learning however. After generating input classes, the process becomes totally automated, which makes this method feasible. The fuzzer also has an output analyzing unit which determines the correctness of the results. The output analyzer obtains the expected output from the program state machine.

1 Introduction
1.1 Software Testing
Software testing is the field of ensuring the quality of software by examining its adherence to the requirements [1]. Typically, software testing is mostly concerned with evaluating the functional quality of software correctness and completeness [2]. However, there exists more software quality characteristics that can be, which are non-functional and can be tested [3]. In essence software testing is the process of selecting test cases, running the software on the test cases, and analyzing the output [1]. Depending on the software model and philosophy, software engineers disagree on when test cases should be developed and how testers should be involved in the software development cycle. Proponents of agile software models argue testing should be considered since the requirements phase. There are two main types of software testing black box testing and white box testing.

Black box testing
Software is tested externally without access to the source code. The test cases are developed based on the software requirements. Black box testing cannot guarantee any sort of code coverage (see 1.2) by the test cases because
the code is a "black box" to the tester. However, it is possible to achieve input
domain coverage with black box testing. Types of black box testing include
integration testing (overlaps), functional testing, beta testing, and fuzz testing (for
more on Integration functional, and beta testing see [12]).

White box testing

Testers maintain access to the source code of the software being tested.
Test cases are developed based on the source code. Test cases can be created
such that different types of code coverage (see 1.2) is accomplished. Moreover,
it is trivial to see that white box testing can also achieve domain coverage.
Examples of white box testing include integration testing, regression testing, and
unit testing [12],

1.2 Test Coverage

Test coverage is one way of assessing the quality of software testing [4].
Test coverage measures the completeness of test cases. There are two main
branches of test coverage: code coverage and input or data coverage.

Code Coverage

Code coverage refers to the amount of code in the program that was
tested or executed by the test cases [5]. Using flow control statements, certain
blocks of code that are ran depending on the input provided to the program.
Therefore, different test cases trigger different pieces of code to execute.
Complete code coverage entails that enough test cases are created to cause the
execution of every single line of code. Code coverage can only be achieved in
white box testing because the test cases must be developed based on the
conditions and flow of the code. There are several types of code coverage (for
more on these types see [6]) including statement coverage, decision coverage,
condition coverage, and path coverage. One advantage of code coverage is that
there is a host of coverage testing tools that can precisely measure it [7, 8, 9, 10,
11].

Input Coverage

Input coverage examines how thoroughly do the test cases cover the
possibilities of input. The total range of possible values a program accepts any
time it waits for input is usually immense. For example, for a program that inputs a single 32 bit integer there are $2^{32}$ possible inputs. If the program accepted two integers the aggregate possible values to input grow exponentially to $2^{64}$. The problem takes on new dimensions of complexity when we consider interleaving different data types for inputs such mixing integers, reals, and strings. It is clear that any modestly complex program would have an input space that is extremely enormous. This leads us to the fact that typically it is absolutely impossible to completely test all possible inputs to a program or in other words to achieve full input coverage [13]. However, it is only with complete input coverage that we can be assured that the program is bug free. Hence regardless of how much testing is done or whether testers are discovering new bugs or not, we cannot rule out the possibility that the software still contains faults. As Dijkstra put it "program testing can be used to show the presence of bugs, but never to show their absence!" Since input coverage is usually impossible, we need a clever method to divide the domain of input into classes where all values in certain class are equivalent, and thus we can achieve domain coverage instead (see 3). Striving for domain coverage is practical because the entire domain of input is divided into an extremely small number of equivalence classes. Ideally, all the data that belongs to one equivalence class is equivalent with respect to the software at hand. Therefore, if the program computes the correct output for a few values in a class, it can be safely assumed that the program will compute the correct output for all the data in the class.

2 Fuzz Testing

2.1 Fuzz Testing

Fuzz testing is a type of black box software testing where input to the program is generated randomly [16]. The software is then tested with the random input and the output of the program is analyzed for failures. Fuzz testing evaluates not only functional qualities of the software such as correctness and completeness, but it usually used to challenge the robustness of a software. Hence in analyzing the software during fuzzing, along with verifying the output the program is checked for crashes, memory leaks, and stability. Fuzz testing
has many advantages including: cost effectiveness, readily automatable, and swiftness in developing test cases and trying them. The downfalls of fuzz testing are the need for randomness, and the disconnection between test cases and the requirements and source code. Since test cases are generated randomly they do not target specific requirement or source code issues.

2.2 Directed Fuzz Testing

In directed fuzz testing, we attempt to get the best of both worlds. In directed fuzz testing, constraints are placed on the pool of possible data to use as input to the program. By directing to the fuzzers to certain interesting inputs, we prevent the fuzzer from choosing trivially undesirable selections [17]. Fuzzing choices are deemed ineffectual if they are equivalent (see 3.1 for definition of equivalency). So in directed fuzz testing certain restrictions are placed on the fuzzers such that it guides the fuzzers to use contributing test cases. There are different methods to constrain the randomness of testing. It has been shown that white box coverage feedback-based directed random testing converges the fastest and reaches the highest overall code coverage [18].

3 Domain Coverage

3.1 Domain Coverage

Domain coverage is coverage observed on a domain of input. As was discussed above, it is practically impossible to test the entire domain. Though, we can achieve domain coverage by partitioning the domain into a few equivalence classes, and ensure that we choose a test case from each class thereby effectively covering the entire domain [14]. An equivalence class is a partition of the entire input domain, in which all the data is equivalent with respect to the program. Equivalence entails that the operations conducted on two equivalent data are the same and the two data share similar properties [15]. Therefore, if the portion of code which deals with one data functions correctly, it is very likely that it will also function correctly on other similar data [14, 15]. The subtle part here is defining the equivalency classes for a domain. Below I present white box and black box equivalence partitions examples. White box makes this task easier since the code can be used to define partition the
domain. The following demonstrates a white box example.

```c
#define x: signed 32-bit integer

x = user_input();
if( x % 2 == 0 ) {
    ...
} else {
    ...
}
```

Figure 1: domain classes for code 1

For this code, it is trivial to have at least two classes one for odd integers and one for even integers. It is also usually a good idea to put the number zero in its own class since zero has unique properties. So in this case we can achieve domain coverage by testing at least three integers such as 4, 0, and -7. A different (and better) partitioning could yield the following classes.

<table>
<thead>
<tr>
<th>even</th>
<th>zero</th>
<th>odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>Positive</td>
<td>Negative</td>
</tr>
<tr>
<td>..., -6, -4, -2</td>
<td>2, 4, 6, ...</td>
<td>..., -5, -3, -1</td>
</tr>
</tbody>
</table>

Figure 2: another class configuration for code 1

In black box testing, we lack the luxury of analyzing the source code. Thus, the equivalence partitions must be inferred from the software requirements or by using certain heuristics. The following illustrates a black box testing example.
The program inputs an unsigned integer $a$. The program inputs an unsigned integer $b$. The program outputs $a/b$. 

Figure 3: software requirements

From the above requirements, we can generate the classes in figure 4.

\[
\begin{array}{|c|c|c|}
\hline
\text{even} & \text{zero} & \text{odd} \\
\hline
\ldots -4, 2, 4 \ldots & 0 & \ldots -3, 1, 3 \ldots \\
\hline
\end{array}
\]

Classes of input domain

Figure 4: domain classes for requirements

3.2 Domain Coverage with Directed Fuzz Testing

In domain coverage-based directed fuzzing, constraints are place on the random generation of test cases based on the input equivalence partitions. In other words, the fuzzer is directed to choose values from a different class after a certain constant limit for the number of values drawn from a single class. By alternating different domain partitions we ensure domain coverage in a feasible manner because the number of partitions is orders of magnitude smaller than the number of possible cases in the entire domain.
4 Methodology

Figure 5: entire testing system
The methodology of the proposed technique is simple and can be summarized in the following steps and the above figure.

- A software requirements or an Request For Comments (RFC) document describing the functionality of the program is provided.
- A set of input domain classes are developed based on the characteristics of the program and provided to the fuzzer. For example, if the program requires any integer input, an integer domain partitions are defined and represented in a text file according to the expected grammar.
- A state machine is constructed manually that represents the requirements, which clearly exhibits the inputs and outputs expected in each state.
- The state machine is converted manually to a text representation according to the predefined grammar, which the state machine interpreter can read. See state machine grammar below.
- The text state machine is read by the interpreter, represented internally, and linked to the domain partitions provided based on the state machine’s edges' inputs and outputs.
- The fuzzer begins selecting values from the defined classes ensuring that it covers all the classes for each domain. The fuzzer then inputs the data to the program and awaits the program output.
- The program output is passed to the output analyzing unit to examine for failures.
- Reports are generated based on the verification of the output analyzer.
Currently this directed fuzz tester is a work in progress. The state machine interpreter has been developed but work is still required to integrate it with the rest of the components.

5 Case Study: Session Initiation Protocol

In Test Lab at UCI headed by Professor Ian Harris, much studying has been conducted about the Session Initiation Protocol or SIP. Consequently, it has been chosen as the first protocol to be used with this tester. SIP is a protocol used to manage communication between IP-based devices. SIP is used to negotiate a conversation to run a media session. After the media session completes, SIP is used once again to negotiate the termination of the conversation. For extensive information on SIP see [20].

![SIP Message Sequence Chart](image)

Figure 7: SIP message sequence chart from a previous work [19]
Figure 8: SIP state machine from a previous work [19]
6 Future Work

Currently efforts are allocated to complete and debug the proposed fuzzing. After the successful completion and testing of the fuzzing, there is an array of additions and insights that can expand on this work, including looking into the automation of the creation of the equivalence partitions by using artificial intelligence and machine learning methods or using heuristics.

7 Summary

It is feasible to achieve domain coverage with the absence of the source code by utilizing the randomness of fuzz testing and guidance of equivalence classes. This black box approach is fast and practical because everything is automated after creating the classes and state machine. State machine design could be tedious, however, it only needs to be done once.

8 References


   http://www.issco.unige.ch/projects/ewg96/node55.html


   http://www.javaranch.com/newsletter/200401/IntroToCodeCoverage.html

   http://www.bullseye.com/coverage.html

   http://www.patersontech.com/TestCoverage/LiveCoverage.aspx

   http://www.bullseye.com/measurementTechnique.html


Source.net/open-source/code-coverage


