A New Tool for Grammar-based Test Case Generation

by

Lewis Paul Sobotkiewicz
B.Sc, University of Victoria, 2004

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Lewis Paul Sobotkiewicz, 2008
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
A New Tool for Grammar-based Test Case Generation

by

Lewis Paul Sobotkiewicz
B.Sc, University of Victoria, 2004

Supervisory Committee

Dr. Daniel M. Hoffman, Supervisor
(Department of Computer Science)

Dr. Micaela Serra
(Department of Computer Science)

Dr. Sudhakar Ganti
(Department of Computer Science)
Supervisory Committee

Dr. Daniel M. Hoffman, Supervisor  
(Department of Computer Science)

Dr. Micaela Serra  
(Department of Computer Science)

Dr. Sudhakar Ganti  
(Department of Computer Science)

Abstract

Software testing is a time-consuming and expensive task. To reduce costs, automating many testing steps is desirable. In grammar-based test generation (GBTG), inputs to a system under test are defined by a context-free grammar. The language of the grammar contains all possible test cases. Another approach based on covering arrays (CA) strategically reduces the number of test cases produced. Both GBTG and CA are normally used independently. We show that the two methods are very powerful when used together.

We introduce a notation and derivation algorithm that combines traditional GBTG and CA. We describe YouGen, a new tool for defining and producing such test cases. In order to demonstrate the versatility of YouGen, we describe the methodology and results of a case study testing network firewall behaviour. The thesis of this work is that GBTG and CA can be combined to produce a powerful and practical test case generator.
**Table of Contents**

Supervisory Committee .................................................. ii

Abstract ................................................................. iii

Table of Contents ........................................................ iv

List of Figures ............................................................ viii

1. Introduction ............................................................ 1
   1.1 Solution ............................................................ 1
       1.1.1 What is GBTG ................................................. 2
       1.1.2 Generative CFGs ............................................. 2
       1.1.3 Covering Arrays ............................................. 4
       1.1.4 How can the two be combined? ............................. 4
   1.2 YouGen ............................................................. 5
       1.2.1 What is YouGen ............................................. 5
       1.2.2 Notation for mixed-strength cover specifications ........ 5
       1.2.3 Case study .................................................. 5
   1.3 Thesis Organization ............................................... 7

2. Related Work .......................................................... 8
   2.1 Quality Assurance ............................................... 8
       2.1.1 Source Code Analysis ...................................... 8
           2.1.1.1 Static Code Analysis ................................. 8
           2.1.1.2 Inspections ......................................... 8
           2.1.1.3 Formal Verification ................................. 9
           2.1.1.4 Discussion ......................................... 9
       2.1.2 Testing ..................................................... 9
   2.2 Covering Arrays .................................................. 10
   2.3 Grammar-Based Testing ........................................... 10

3. Mixed-Strength Covering Arrays .................................... 11
   3.1 Covering Arrays ................................................ 11
       3.1.1 Two-cover Example ....................................... 11
       3.1.2 One-cover Example ...................................... 13
8. Conclusions and Future Work ........................................... 59
  8.1 Conclusions .............................................................. 59
  8.2 Future Work .............................................................. 60
    8.2.1 Merge multiple mixed-strength covering arrays ............... 60
    8.2.2 Test additional applications .................................... 60
Bibliography ................................................................. 61
## List of Figures

1.1 Call grammar example ................................................. 3
1.2 Call grammar with a mixed-strength cover. .......................... 6
3.1 A set of domains and a two-cover. ................................. 12
3.2 One-cover of domains from Figure 3.1(a) ............................ 13
3.3 Four domains and a three-cover ................................. 14
3.4 Examples of mixed-strength covering array specifications .......... 16
3.5 Examples of mixed-strength cover specifications with restricted scopes . 17
4.1 Three derivations of the Call grammar .............................. 19
4.2 Derivation tree for Call Grammar ................................. 21
4.3 Bitstring example .................................................. 21
5.1 Derivation tree of grammar from Figure 1.1 ......................... 24
5.2 YouGen output of grammar from Figure 1.1(a) ....................... 25
5.3 Bitstring derivation tree with labelled derivations ................. 27
5.4 Book grammar ..................................................... 28
5.5 Example of Postcode Usage ........................................ 30
5.6 Example of Globalcode Usage ..................................... 32
5.7 Example of Range Terminal Generator ................................ 33
5.8 Example of List Terminal Generator ................................ 33
5.9 Example of a Custom Terminal Generator .......................... 34
6.1 Call graph of grammar translation .................................. 40
6.2 Derivation algorithm in YouGen .................................... 43
6.3 An executable YouGen grammar. ................................... 44
7.1 TCP connection and teardown sequence with possible bad flag positions 47
7.2 TCP Bad Flags Test Grammar ...................................... 50
7.3 Example output of TCP bad flags grammar .................. 51
7.4 Test configuration ........................................... 52
7.5 Bad Flags executor raw output sample ....................... 54
7.6 TCP bad flags executor pseudocode ......................... 55
7.7 Summary of test results ...................................... 56
7.8 Comparison of results of 1-cover and partial 2-cover .......... 58
Chapter 1
Introduction

Production software is known to contain a large number of defects. The cost of these defects is also large. For example, an outage of equipment that automates an assembly line can cost several thousand dollars per minute. A 2002 study by the National Institute of Standards and Technology found that software bugs cost the US economy 60 billion dollars per year, one third of which would be preventable with stronger software testing [19]. There are numerous examples of software errors having a direct impact on the economy. Recently, an error in software models in a popular financial research and analysis firm caused it to give incorrect high ratings to a debt product which threatened billions of dollars of investor money [21].

1.1 Solution

Test generation has been used in a variety of application domains for decades. For smaller systems, hand-crafted test cases are easy enough to develop and maintain. As complex software such as compilers became widespread, developers needed a more efficient and reliable way to increase code coverage. Automated test generation uses software to create the set of test cases.

One of the first application domains to employ test generation was compilers. Because the syntax of most computer languages are context-free, grammars were a logical choice for generating test inputs. The test inputs would be compiled and run, and their outputs compared to the expected outputs for each test case [11, 5, 2, 3].

There are several tools that use test generation to look for bugs in network protocol implementations. The earliest known publication describes a system for testing trivial file transfer protocol (TFTP) implementations, simple network management protocol (SNMP), and others [15]. Implementations of open shortest path first (OSPF) protocols have been tested using similar techniques [24]. Recently, a system for testing
MODBUS protocol implementations has been developed and deployed [4, 12].

1.1.1 What is GBTG

Grammar-Based Test Generation (GBTG) is an approach to test generation that employs context-free grammars to create sets of test cases.

The context-free grammar describes the syntax of the input to the system under test. Using the syntax, a generator tool produces test cases that conform to the syntax of the system under test’s input.

Context-free languages can be used to represent the syntax of most computer languages, as well as network protocols and data serialization formats. A common use of CFGs is the specification of programming language syntax and parsers. Most compilers contains an analytic grammar to verify that a sequence of input symbols matches the definition of the language.

1.1.2 Generative CFGs

GBTG uses generative context-free grammars to produce strings that conform to the syntax of the inputs of the system under test.

With generative context-free grammars, the test space is typically the language of the grammar, $L(G)$. Often, $L(G)$ is so large that it is impractical to run every test case generated. Depending on the nature of the test, executing one test case may be very expensive or time-consuming. Most GBTG tools feature extra-grammatical tags which supply instructions or parameters to the generation algorithm to reduce the number of strings generated in a controlled way.

For example, Figure 1.1(a) shows a grammar that can be used to test VoIP software that is sensitive to the type of operating system. The grammar focuses on combinations of the OS running on the calling phone (CallerOS), the VoIP server (ServerOS), and the called phone (CalleeOS). $L(G)$ is all 12 combinations as shown in Figure 1.1(b).
Call ::= CallerOS ServerOS CalleeOS;
CallerOS ::= 'Macintosh';
CallerOS ::= 'Windows';
ServerOS ::= 'Linux';
ServerOS ::= 'SunOS';
ServerOS ::= 'Windows';
CalleeOS ::= 'Macintosh';
CalleeOS ::= 'Windows';

(a) Call grammar; Call is the start symbol

<table>
<thead>
<tr>
<th>CallerOS</th>
<th>ServerOS</th>
<th>CalleeOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macintosh</td>
<td>Linux</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Macintosh</td>
<td>Linux</td>
<td>Windows</td>
</tr>
<tr>
<td>Macintosh</td>
<td>SunOS</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Macintosh</td>
<td>Windows</td>
<td>Windows</td>
</tr>
<tr>
<td>Windows</td>
<td>Linux</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Windows</td>
<td>Linux</td>
<td>Windows</td>
</tr>
<tr>
<td>Windows</td>
<td>SunOS</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
</tr>
<tr>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
</tr>
</tbody>
</table>

(b) Language of Call grammar

<table>
<thead>
<tr>
<th>CallerOS</th>
<th>ServerOS</th>
<th>ServerOS</th>
<th>CalleeOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macintosh</td>
<td>Linux</td>
<td>Linux</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Macintosh</td>
<td>SunOS</td>
<td>SunOS</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Macintosh</td>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
</tr>
<tr>
<td>Windows</td>
<td>Linux</td>
<td>Windows</td>
<td>Macintosh</td>
</tr>
<tr>
<td>Windows</td>
<td>SunOS</td>
<td>Windows</td>
<td>Windows</td>
</tr>
<tr>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
</tr>
</tbody>
</table>

(c) Domain subsets of size 2

Figure 1.1: Call grammar example
1.1.3 Covering Arrays

Covering array algorithms generate a subset of the cartesian product of the input domains. With covering arrays, the test space is the cartesian product of $N$ finite input domains, where each domain is a parameter to the system under test. Usually, the test space is too large to execute all test cases. The covering array algorithm generates a subset of the cartesian product containing all parameter combinations of a certain type. Covering arrays guarantee that all combinations of parameters of a given size are present in the test set.

From the covering array perspective, Figure 1.1 has three input domains. The cartesian product represents the test space, shown in Figure 1.1(b). The italicized columns give a two-cover.

To verify that the two-cover is correct, look at each domain subset of size 2, as shown in Figure 1.1(c). The two-cover is valid if and only if each row of the domain subsets corresponds to a row in the two-cover.

1.1.4 How can the two be combined?

Covering arrays and GBTG share important similarities. Covering arrays are used for generating subsets of a cartesian product, where context-free grammars are useful for generating subsets of a hierarchically-structured language. Both approaches have seen substantial tool development and successful deployment in industry [11, 8]. Usually they are applied separately. We will show that, when used together, the two can be used to find and investigate bugs more efficiently than using one method alone.

Often the tester is only interested in a subset of the cartesian product; some interactions between system inputs may have more significance to the tester than others. Hence, we employ mixed-strength covering arrays to allow a user to specify which parameters are included in the cover. Mixed-strength covering arrays concentrate combinatorial generation to a specific set of domains.
1.2 YouGen

1.2.1 What is YouGen

YouGen is a new tool that combines grammar-based test generation with mixed-strength covering arrays. The tool combines the two approaches to create smaller test sets from a large test space.

YouGen takes a grammar $G$ as input and produces $L(G)$. By applying restrictions to grammar rules, termed tags, YouGen can generate a subset of $L(G)$ by pruning the number of strings generated.

1.2.2 Notation for mixed-strength cover specifications

YouGen integrates mixed-strength covering arrays using a tag that allows the user to specify a list of mixed-strength cover specifications. A mixed-strength cover is represented as a two-tuple consisting of a list domains to cover (the scope), and the strength of the cover.

For example, a tester using the grammar from Figure 1.1(a) is only interested in interactions between the ClientOS and ServerOS. Figure 1.2(a) shows the Call grammar decorated with a mixed-strength cover specification that will generate all combinations of ClientOS and ServerOS with no coverage guarantees for CalleeOS. Figure 1.2(b) shows the language of the call grammar. The italicized rows show a cover that meets the specification.

1.2.3 Case study

To observe the advantages of mixed-strength covering arrays and grammars, we created a test bench for testing network firewalls. We used a YouGen grammar to generate packets in a Transport Control Protocol (TCP) connection. The grammar allowed us to introduce invalid packets at any point where the connection changes state. By sending these connections through a firewall, we analyzed the firewall’s behaviour upon receipt of the invalid packets at each point. We obtained interesting results from two widely-used firewall products. By changing the mixed-strength cover specification, we could focus the scope of the test cases generated to points in the
\{cov \([(0,1),2)]\} \text{Call ::= CallerOS ServerOS CalleeOS;}
CallerOS ::= 'Macintosh';
CallerOS ::= 'Windows';
ServerOS ::= 'Linux';
ServerOS ::= 'SunOS';
ServerOS ::= 'Windows';
CalleeOS ::= 'Macintosh';
CalleeOS ::= 'Windows';

(a) Call grammar with a mixed-strength cover.

\begin{table}[ht]
\centering
\begin{tabular}{|c|c|c|}
\hline
\text{CallerOS} & \text{ServerOS} & \text{CalleeOS} \\
\hline
1 & Macintosh & Linux & Macintosh \\
2 & Macintosh & Linux & Windows \\
3 & Macintosh & SunOS & Macintosh \\
4 & Macintosh & SunOS & Windows \\
5 & Macintosh & Windows & Macintosh \\
6 & Macintosh & Windows & Windows \\
7 & Windows & Linux & Macintosh \\
8 & Windows & Linux & Windows \\
9 & Windows & SunOS & Macintosh \\
10 & Windows & SunOS & Windows \\
11 & Windows & Windows & Macintosh \\
12 & Windows & Windows & Windows \\
\hline
\end{tabular}
\end{table}

(b) Language of call grammar.

Figure 1.2: Call grammar with a mixed-strength cover.
connection that produced anomalous results.

1.3 Thesis Organization

Chapter 2 describes related work in the field of test generation. Chapter 3 provides an introduction to mixed-strength covering arrays and gives some examples. Chapter 4 contains a brief overview of context-free grammar terminology and gives numerous examples. Chapter 5 describes the main features of YouGen, with emphasis on how YouGen integrates covering arrays. Chapter 6 discusses the technical organization and generation algorithm used in YouGen. Chapter 7 discusses the application YouGen to studying the behaviour of firewalls when they are sent a special family of invalid packets. Finally, Chapter 8, presents the conclusions of this work and discusses some directions for future work.
Chapter 2
Related Work

In this chapter we will contrast source code analysis and testing, the two main software verification approaches. While both are commonly employed during development of large software systems, we will concentrate on two tools used by testers: covering arrays and grammar-based generation.

2.1 Quality Assurance

Quality assurance is a broad discipline in software engineering that attempts to confirm that a piece of software behaves according to its specification. The two fundamental approaches, source code analysis and testing, are widely used in industry as they complement each other considerably.

2.1.1 Source Code Analysis

Source code analysis is a branch of software verification where issues are detected without running the software.

2.1.1.1 Static Code Analysis

Static code analysis is a method of assessing the correctness of computer software without executing any part of that software’s code. Static analysis is usually performed by automated tools that analyze source code and look for programming errors and other weaknesses such as undesirable control flow, data use and redundant code. It should be noted that many compilers, e.g., C/C++, include static analysis features such as warnings [25].

2.1.1.2 Inspections

A source code inspection is a manual analysis of source code performed by a group of software developers [10]. Inspections can be used in any stage of the software de-
velopment cycle. Many organizations perform inspections at the design stage as well as implementation. Typically, they are used to catch mistakes early, and to disseminate knowledge about a system to the group. Human inspection has limitations, as analyzing complex code is itself complex and thus is error-prone [13].

2.1.1.3 Formal Verification

Formal verification of software employs formal methods of mathematics to prove or disprove the correctness of a piece of software [18]. For example, model checkers examine all reachable program states and attempt to find an execution path that violates the program’s specification [9].

2.1.1.4 Discussion

While source code analysis is effective and widely-used, many kinds of defects can be observed only when the software is running in its intended environment. For example, interactions between software and underlying hardware are difficult to predict. Some tools require designers to create complicated models of a program’s behaviour, introducing another source of errors. Thus, the results of source code analysis are questionable [13].

2.1.2 Testing

Testing involves executing the software under test with a predetermined input and expected output. The program’s correctness is determined by comparing its actual output to the expected output. A predetermined input/output pairs is known as a test case. Testers will typically execute numerous test cases, which may be developed manually or using an automated system. Once a set of test cases is created, each one is executed while monitoring the behaviour of the program. Finally, the program’s readiness is determined by evaluating the test case results.

The main drawback of testing is its cost; manually creating test cases is time-consuming, and it is difficult to verify a test case’s effectiveness. Executing and evaluating test results also incurs a cost each time the test cases are run. Therefore, it is advantageous to automate these steps as much as possible [13].
2.2 Covering Arrays

Covering arrays have shown to reduce the cost of test plan development and execution by creating a smaller, more efficient test set [8, 7]. A common utilization is pairwise testing, where all combinations of two parameters are covered. Many approaches have been proposed for generating covering arrays. The common In Parameter Order (IPO) method was first proposed for pairwise testing in [23] and extended to n-way testing in [17]. Many approaches are applicable only when the size of all input domains are equal, which is uncommon in practical testing environments.

2.3 Grammar-Based Testing

The work of Hanford was the earliest known application of CFGs to testing; Hanford generated PL/1 programs for compiler testing [11]. Years later, Bird and Munoz applied GBTG to compiler testing, sort/merge utilities and graphical output applications [1]. They introduced a new technique for test oracles in all these test domains. Burgess utilized grammars for automatically generating test sets for optimizing Fortran compilers [2, 3].

Later, Sirer developed a language called lava to test implementations of the Java Virtual Machine [22]. The system emphasized the advantages of automated test generation over hand-written tests and found previously unknown faults in two commercial JVM implementations.

Much of the later work in GBTG focuses on network protocol testing. Kaksonen created the PROTOS tool for testing the security of Simple Network Management Protocol (SNMP) implementations and later extended it to other protocols [15]. Later research applied GBTG to vulnerability testing of frame-based protocols, specifically Open Shortest Path First [24]. This paper introduced terminology that has been used in much of the subsequent research. Recently, Hoffman and Kube have applied grammar-based testing to SCADA protocols [12].

Most recently, Lammel explored integrating covering arrays with CFGs [16].
Chapter 3
Mixed-Strength Covering Arrays

3.1 Covering Arrays

A covering array is a subset of the cartesian product of non-empty finite domains $D_0, D_1, \ldots, D_{N-1}$. For large domains, the resulting set is usually much smaller than the full cartesian product, and certain specified interactions between inputs are guaranteed.

The strength of a cover refers to the number of domains over which N-way cartesian products are guaranteed to be in the set of resulting domain elements. The number of resulting strings increases with strength; covering at higher strength provides more interactions between the input domain values.

Perhaps the most common form of using covering arrays in the context of testing is pairwise testing, which uses a covering array of strength two. Pairwise testing guarantees that all pairs of inputs are “covered”. That is, all tuples in the cartesian products of any two domains in the input space are present.

3.1.1 Two-cover Example

Figure 3.1 gives an example of a two-cover over three domains. Figure 3.1(a) shows a set of three domains, Figure 3.1(b) shows the cartesian product of these domains, with rows in a two-cover denoted by bold line numbers.
(a) Test domains

<table>
<thead>
<tr>
<th></th>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$D_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>2</td>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>3</td>
<td>$a_0$</td>
<td>$b_1$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>4</td>
<td>$a_0$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>5</td>
<td>$a_0$</td>
<td>$b_2$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>6</td>
<td>$a_0$</td>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>7</td>
<td>$a_1$</td>
<td>$b_0$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>8</td>
<td>$a_1$</td>
<td>$b_0$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>9</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>10</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>11</td>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>12</td>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
</tbody>
</table>

(b) Cartesian product of test domains.

<table>
<thead>
<tr>
<th>Row</th>
<th>$D_0$</th>
<th>$D_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_0$</td>
<td>$b_0$</td>
</tr>
<tr>
<td>4</td>
<td>$a_0$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>6</td>
<td>$a_0$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>8</td>
<td>$a_1$</td>
<td>$b_0$</td>
</tr>
<tr>
<td>9</td>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>11</td>
<td>$a_1$</td>
<td>$b_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row</th>
<th>$D_1$</th>
<th>$D_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$b_0$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>4</td>
<td>$b_0$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>6</td>
<td>$b_1$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>9</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>11</td>
<td>$b_2$</td>
<td>$c_0$</td>
</tr>
<tr>
<td>12</td>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
</tbody>
</table>

(c) Cartesian products $D_0 \times D_1$, $D_1 \times D_2$ and $D_0 \times D_2$ and corresponding row in (b).

Figure 3.1: A set of domains and a two-cover.
To show that the bold rows are a two-cover, we have to find the cartesian products of each pair of domains. In this example, there are three such pairs, i.e., $(D_0, D_1)$, $(D_1, D_2)$, $(D_0, D_2)$.

For each two-tuple, we’ll check to see if it appears in a row with a bold line number in Figure 3.1(b). The first pair, $(a_0, b_0)$, appears in row 1. The next five pairs appear in rows 4, 6, 8, 9, and 11, respectively.

To complete the demonstration, we’ll do the same exercise for the remaining 2-way Cartesian products. Figure 3.1(c) shows the cartesian products of each pair and gives the row number they appear in in Figure 3.1(b).

### 3.1.2 One-cover Example

A one-cover produces every element in each domain at least once, but no combinations between domain elements are guaranteed. A one-cover produces elements in all the domain subsets of size one, which is equivalent to the set of domain values themselves.

\[
\begin{array}{ccc}
D_0 & D_1 & D_2 \\
 a_0 & b_0 & c_0 \\
 a_1 & b_1 & c_1 \\
 a_1 & b_2 & c_1 \\
\end{array}
\]

Figure 3.2: One-cover of domains from Figure 3.1(a)

All values in each of the 3 domains appear at least once in the resulting test set. Because $D_2$ is larger than the other two domains, the values $a_1$ from $D_0$ and $c_1$ from $D_2$ are selected twice. This selection is arbitrary, but one value from each of these smaller domains is necessary to complete a three-tuple.

### 3.1.3 Three-cover Example

A three-cover ensures that all tuples in the cartesian products of any three domains is covered. Figure 3.3 shows a set of four domains, and an example of a three-cover. To verify that Figure 3.3(b) is a three-cover, consider the cartesian products of each group
of three domains. Each row in those cartesian products must appear in Figure 3.3(b).

The full verification has been excluded for brevity.

<table>
<thead>
<tr>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$c_2$</td>
<td></td>
</tr>
</tbody>
</table>

(a) Four domains

<table>
<thead>
<tr>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_0$</td>
<td>$c_1$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_1$</td>
<td>$c_0$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_2$</td>
<td>$c_2$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_2$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_0$</td>
<td>$c_2$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_2$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_1$</td>
<td>$c_0$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_1$</td>
<td>$c_2$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_0$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_2$</td>
<td>$c_0$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_2$</td>
<td>$c_1$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_0$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_2$</td>
<td>$d_1$</td>
</tr>
</tbody>
</table>

(b) A three-cover over domains in (a)

Figure 3.3: Four domains and a three-cover
3.2 Mixed-strength Covering Arrays

While this does guarantee that all interactions of a given strength are covered, this may not be what a tester wants. For example, a tester may want to cover all interactions between two out of three domains and disregard the third. This prunes interactions between inputs that the tester is not interested in and may significantly reduce the size of the test set.

As the strength of the covering array increases, its size also increases. A tester may only be interested in combinations of a specific set of inputs. For large input domains, or tests in which every case is expensive to run, reducing the size of the test set is a big win.

Traditional covering arrays use the same strength across all domains. Mixed-strength covering arrays permit multiple covering array specifications that each apply to a subset of the domains.

- As with traditional covering arrays, mixed-strength covering arrays operate on a list of non-empty, finite domains $D_0, D_1, \ldots, D_{N-1}$.

- An index set is a subset of $[0..N-1]$.

- A coverage specification is a term of the form $(I, n)$ where $I$ is an index set and $n \in [1..|I|]$. $I$ denotes the scope and $n$ the strength of the coverage specification.

- Test set $T \subseteq D_0 \times D_1 \times \ldots \times D_{N-1}$ satisfies coverage specification $(I, n)$ if

$$\forall I_0 \subseteq I, I_0 = \{i_0, i_1, \ldots, i_{n-1}\} \quad \pi_{i_0,i_1,\ldots,i_{n-1}}(T) = D_{i_0} \times D_{i_1} \ldots \times D_{i_{n-1}}$$

where $\pi$ is the projection operator from relational algebra.

Any test set $T$ is a subset of the cartesian product of the input domains and satisfies coverage specification $(I, n)$ if for any subset of $I$ of size $n$ the cartesian products of the domains represented by the indices $\{i_0, i_1, \ldots, i_{n-1}\}$ are in $T$. Users of covering arrays specify a list of one or more coverage specifications $(I, n)$, allowing greater control over the resulting test cases.
Mixed-strength covers can be used to generate traditional covers. Figure 3.4 shows examples of cover specifications that include all domains in their index set.

The big win for testers is the ability to concentrate interactions to a subset of the input domains. Figure 3.5(a) shows an example of a mixed-strength cover where the tester is only interested in combinations between \(D_0\) and \(D_1\) and \(D_2\) and \(D_3\). Combinations between other inputs (e.g. \((D_0, D_2)\)) may be present, but are not guaranteed. For example, the combination \(\langle a_0, c_2 \rangle\) does not exist in the test set.

In Figure 3.5(a), there are two coverage specifications: \((\{0, 1\}, 2)\) and \((\{2, 3\}, 2)\). \(\{0, 1\}\) and \(\{2, 3\}\) are the index sets of the respective specifications and 2 is the strength of both. The first specification has only one subset of \(I\) of size 2: \(\{0, 1\}\). All elements in
the cartesian product $D_0 \times D_1$ must be in the test set. The second specification’s index set also has just one subset of size 2: $(2, 3)$. Similarly, all elements in the cartesian product $D_2 \times D_3$ must be in the test set. Figure 3.5(b) shows these cartesian products and gives the line numbers they correspond to in the test set $T$. Since all elements appear in $T$, Figure 3.5(a) is a mixed-strength cover over the domains from Figure 3.3.

(a) Test set satisfying ($\{0, 1\}, 2$) and ($\{2, 3\}, 2$) over domains from Figure 3.3

<table>
<thead>
<tr>
<th>Line</th>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>2</td>
<td>$a_0$</td>
<td>$b_1$</td>
<td>$c_0$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>3</td>
<td>$a_0$</td>
<td>$b_2$</td>
<td>$c_1$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>4</td>
<td>$a_1$</td>
<td>$b_0$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>5</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_2$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>6</td>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_2$</td>
<td>$d_1$</td>
</tr>
</tbody>
</table>

(b) $D_0 \times D_1$ and $D_2 \times D_3$

<table>
<thead>
<tr>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_1$</td>
<td>$c_0$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_0$</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_0$</td>
<td>$d_1$</td>
</tr>
</tbody>
</table>

(c) Test set satisfying ($\{0, 3\}, 2$) and ($\{1\}, 1$) over domains from Figure 3.3

Figure 3.5: Examples of mixed-strength cover specifications with restricted scopes
Chapter 4
Context-Free Grammars

A context-free grammar (CFG) is a formal specification of a context-free language, whose terms can be written regardless of the context in which they occur [14].

A grammar $G = \langle V, T, R, S \rangle$ where $V$ is a set of variables and $S \in V$ is the start symbol. $T$ is a set of terminals, the components of a string in a language. $R$ is a set of rules that specify how variables may be rewritten. A rule has the general structure $head ::= body$; where $head$ is in $V$ and each element in $body$ is an element in $V$ or $T$. During a grammar derivation step, the head is rewritten by the body.

Figure 1.1 is an example of a grammar. $V = \{ \text{Call, CallerOS, ServerOS, CalleeOS} \}$. $T = \{ \text{'Macintosh', 'Linux', 'Windows', 'SunOS'} \}$. The variable Call is the start symbol $S$. The first line defines a rule which transforms variable Call into CallerOS ServerOS CalleeOS.

4.1 Derivations

When using a context-free grammar for generation, a derivation results in a string in the language defined by the grammar. This section introduces the reader to the aspects of context-free grammar derivation.

4.1.1 Sentential Forms

A sentential form is a sequence of terminals and variables. Specifically, a sentential form is a list of elements $s_0 s_1 \ldots s_{n-1}$ from $V$ and $T$.

4.1.2 Derivation Step

A derivation step is a single application of a rule, where a $head$ variable is rewritten as $body$. The first line of Figure 4.1(a) shows Call being transformed into CallerOS ServerOS CalleeOS as defined by the first rule in the grammar. The second line
Call ⇒ CallerOS ServerOS CalleeOS
  ⇒ Macintosh ServerOS CalleeOS
  ⇒ Macintosh Linux CalleeOS
  ⇒ Macintosh Linux Macintosh

(a) Leftmost Derivation of Macintosh Linux Macintosh

Call ⇒ CallerOS ServerOS CalleeOS
  ⇒ Macintosh ServerOS CalleeOS
  ⇒ Macintosh Linux CalleeOS
  ⇒ Macintosh Linux Windows

(b) Leftmost Derivation of Macintosh Linux Windows

Call ⇒ CallerOS ServerOS CalleeOS
  ⇒ CallerOS ServerOS Windows
  ⇒ CallerOS Linux Windows
  ⇒ Macintosh Linux Windows

(c) Rightmost Derivation of Macintosh Linux Windows

Figure 4.1: Three derivations of the Call grammar

shows another derivation step, where one of the CallerOS ServerOS CalleeOS variables is transformed. In this case, CallerOS is selected and is transformed into 'Macintosh' according to the rule Call ::= CallerOS ServerOS CalleeOS;

4.1.3 Derivation

A derivation is the sequence of derivation steps used to transform the start symbol S into a sentential form consisting only of terminals, or a string in the language. Figure 4.1 shows three examples of derivations.
4.1.4 Language

The language \( L(G) \) of a grammar is the set of all strings that can be derived by the grammar. The Call grammar has twelve strings, shown in Figure 1.1(b).

4.1.5 Leftmost and Rightmost Derivations

When performing a derivation, it is common to have a list of elements \( s_0 s_1 \ldots s_{n-1} \) in a sentential form. To perform the next derivation step, one of the variables in this list must be rewritten. There are no rules in context-free grammar theory that specify which element in the list is to be rewritten. This implies that it’s possible for two strings to be derived several different ways. For example, Figure 4.1(c) shows a different derivation of the same string as Figure 4.1(b). In a lefmost derivation, the leftmost variable in a sentential form is always selected for replacement. Similarly, in a rightmost derivation, the rightmost variable is always selected for replacement.

4.1.6 Derivation Tree

A derivation tree is a useful representation of the derivation procedure. Derivation trees are useful for demonstrating how a context-free grammar is translated from a root rule to a resulting string. Each node is a sentential form, the root is the start symbol, and the leaf nodes are complete strings in the language. The path from the root node to a leaf represents a derivation. Figure 4.2 is an example of a derivation tree for 4.1(a) and (b).

4.1.7 Recursive Grammars

A recursive grammar is a grammar with an infinite number of derivations. Figure 4.3(a) depicts a recursive grammar. Bitstring is a directly recursive grammar because it appears in both the head and the body of a rule. Figure 4.3(b) shows a partial derivation tree of Bitstring.
a. Bitstring ::= Bit;
b. Bitstring ::= Bit Bitstring;
c. Bit ::= '0';
d. Bit ::= '1';

(a) Bitstring grammar; Bitstring is the start symbol.

(b) Bitstring derivation tree

Figure 4.2: Derivation tree for Call Grammar

Figure 4.3: Bitstring example
Chapter 5
YouGen Requirements

This section provides a behavioural description of YouGen’s features.

5.1 Basic Syntax

YouGen grammars are specified by writing a series of rules, each of which consists of a head and a body portion, and optional tags. The syntax for rules in YouGen is $\text{head} ::= \text{body}$; where head is a variable name and body is a list of one or more variables or terminals. The topmost rule in the grammar is considered the start symbol.

If the number sign appears anywhere in the grammar file, the rest of the line is treated as a comment and ignored.

Figures 1.1(a), 4.3(a), and 5.4 are examples of YouGen grammar syntax.

5.2 Derivation and Output Formats

For every string derived from a grammar, YouGen will print one line containing all terminals, each separated by a space. Figure 5.2 shows the output when generating the grammar from Figure 1.1(a).

5.3 About Tags

A grammar developer may want to reduce the number of test cases generated by the system. For many applications, $L(G)$ will be too large to use in practice.

YouGen uses tags, extra-grammatical notations used to prune the derivation tree in specific ways. Tags are attributes attached to rules which affect the derivations using that rule.
5.4 Untagged Grammars

YouGen will produce $L(G)$ by rewriting rules beginning with the start symbol. YouGen will replace the head of a rule with its body and derive each element in the body from left to right. If the same head appears more than once in a grammar file, YouGen will first select the topmost occurrence of head in the grammar. Figure 1.1(a) is an example of a context-free grammar with no tags. Figure 5.1 shows the derivation tree, and Figure 5.2 shows YouGen’s output for this grammar.

5.5 Tag Syntax

A Tag is specified by \{ tagName tagArguments \} written immediately before the grammar rule. A rule can have zero or more tags. A specific tagName may be used multiple times per rule, but except for the cov tag, YouGen will only consider the first occurrence. The cov tag is intended to support multiple definitions to allow for mixed-strength cover specifications.

5.5.1 Count Tag

- Syntax. \{ count N \}

  The count tag takes a single nonnegative integer argument.

- Semantics. N states the maximum number of strings to generate from the tagged rule. If N is greater than the number of possible derivations of that rule, then the tag will have no effect, and derivation of the rule will stop when all possibilities are produced. Derivation terminates after N or fewer strings have been produced by the tagged rule.

- Example. If the tag \{ count 2 \} is applied to the grammar in Figure 1.1(a), YouGen would produce only the first two rows of Figure 5.2.

5.5.2 Coverage Tag

- Syntax. \{ cov (I, n) \}

  Where I is the index set and n is the strength of the cover.
Figure 5.1: Derivation tree of grammar from Figure 1.1
Currently, YouGen supports covers of strength 1, 2, 3, and $N$, where $N$ is the number of rules in the body of the tagged rule.

- **Semantics.** The `cov` tag is used to apply a mixed-strength coverage specification to the tagged rule. The index set is a comma-separated, increasing list of indices of rules in the body of the tagged rule over which the mixed-strength cover will be applied. The index set is encased in square brackets.

- **Example.** If the tag `{cov ([0,1,2],2)}` is applied to the grammar from Figure 1.1(a), YouGen will produce a two-cover over all three domains.

- **Example.** When the tags `{cov ([0,2],2)}` and `{cov ([1],1)}` are applied to the first rule of the grammar in Figure 1.1(a), all combinations of CallerOS and CalleeOS are produced, as well as all elements of ServerOS.

### 5.5.3 Depth Tag

- **Syntax.** `{depth $N$}`

- **Semantics.** Derivation terminates when $N$ levels of the subtree of the tagged rule are traversed.
• **Example.** Figure 4.3(b) shows the derivation tree for the grammar in Figure 4.3(a) with the tag \{depth 2\} applied.

### 5.5.4 *Rdepth Tag*

The recursion depth tag limits the number of recursive derivations of a rule.

- **Syntax.** \{rdepth $N$\}

- **Semantics.** Derivation terminates after the tagged rule appears $N$ times within the subtree of the tagged rule.

- **Example.** Figure 4.3(a) shows a recursive grammar that generates sequences of ones and zeroes. The number of derivations is infinite because the grammar is recursive. If the tag \{rdepth 2\} is applied to rule $b$. of Figure 4.3(a), 14 strings are generated. As recursion depth increases linearly, the number of strings produced increases exponentially. Figure 5.3 shows a derivation tree of Bitstring with recursion depth limited to two. The arcs in the tree are labelled with the rule that provided the derivation steps. Because rule $b$ is limited to two recursive derivations, that rule will appear no more than twice during a derivation.
Figure 5.3: Bitstring derivation tree with labelled derivations
5.6 Covers and recursion depth

Used together, the recursion depth and coverage tags are powerful.

Figure 5.4 shows a recursive grammar that generates simple markup documents. Suppose the focus of a test is bugs that arise when the first and last chapters of a markup document generated by Figure 5.4 have a different number of sections. The simplest way to reduce the number of derivations is with the count tag. If the rule on line 1 of Figure 5.4 is tagged with \{count 3\}, only three books will be generated. The third chapter will have from one to three sections. Because Sections is a recursive variable, increasing the count tag’s value will yield more sections in the third chapter.

1. Book ::= ’<BOOK>’ Chapters ’</BOOK>’;

2. Chapters ::= Chapter Chapter Chapter;
3. Chapter ::= ’<CHAPTER>’ Sections ’</CHAPTER>’;

4. Sections ::= Section;
5. Sections ::= Section Sections;
6. Section ::= ’<SECTION>’ ’Section contents . . .’ ’</SECTION>’;

Figure 5.4: Book grammar

The recursion depth tag can spread sections more evenly across the chapters. If the tag \{rdepth 2\} is applied to the rule on line 5 of Figure 5.4, each chapter will have a maximum of three sections.

The combinations of chapter sizes can be controlled using the coverage tag. If the tags \{cov ([0,2],2)\} and \{cov ([1],1)\} is applied to the rule on line 2 of Figure 5.4, all combinations of chapter sizes between the first and last chapters, and all three sizes for the second chapter will be generated.
5.7 Embedded code

YouGen provides a hook for grammar developers to add arbitrary code that can be used to modify the strings derived by YouGen.

5.7.1 postcode

A postcode block can be declared once per rule.

- **Syntax.** \{postcode arbitrary_python_code_block\}

  Because Python’s language syntax depends on whitespace, special consideration must be given to the placement of lines in postcode blocks. If a postcode block spans multiple lines, an indentation of one tab is required.

  Because tags are terminated with a right-curly-brace, any that appear in the postcode block must be escaped by a backslash. The backslash must be escaped as well. When the grammar is translated to Python, the backslashes are removed from the output.

- **Semantics.** YouGen will execute a postcode block directly after a string from the tagged rule has been derived. YouGen passes one parameter, s, to all postcode code blocks. The parameter s is a list containing the derivation of this rule and its child rules. The list can contain two data types: strings, which contain the literal values of terminals, and lists, which contain the list s derived from each grammar variable in rule’s body.

- **Example.** Figure 5.5(a) shows an example of a grammar annotated with postcode. The first two lines are a simple postcode routine that prints the contents of s upon each derivation of the start symbol, twoBit. Similarly, a postcode block for rule A ::= ’0’; prints the contents of s after that rule is derived. Part (b) shows YouGen’s output when executing this grammar. Because A is the first rule in the grammar to be derived, its postcode block is executed first. Rule B’s postcode comes next. Because all rules in the body of TwoBit have been derived, TwoBit’s postcode block is executed. At this point, YouGen prints its flattened output and begins the next derivation of the grammar.
TwoBit ::= A B;
{postcode print "\tDerived " + str(s) }
A ::= '0';
{postcode print "\tFrom A ::= '0': " + str(s) }
A ::= '1';
{postcode print "\tFrom A ::= '1': " + str(s) }
B ::= '0';
{postcode print "\tFrom B ::= '0': " + str(s) }
B ::= '1';
{postcode print "\tFrom B ::= '1': " + str(s) }

(a) Grammar with Postcode

From A ::= '0': ['0']
From B ::= '0': ['0']
Derived [['0'], ['0']]
0 0
From B ::= '1': ['1']
Derived [['0'], ['1']]
0 1
From A ::= '1': ['1']
From B ::= '0': ['0']
Derived [['1'], ['0']]
1 0
From B ::= '1': ['1']
Derived [['1'], ['1']]
1 1

(b) YouGen output for grammar from (a)

Figure 5.5: Example of Postcode Usage
5.7.2 Globalcode

Global code can be used to define library functions or global variables that may be used in postcode blocks. Global code is specified by a globalcode block at the top of the grammar file, and may be specified only once.

- **Syntax.** \{globalcode arbitrary_python_code_block\}

- **Semantics.** The globalcode block will be interpreted once when YouGen is starting up and prior to grammar derivation.

- **Example.** Figure 5.6 shows an example of a grammar annotated with globalcode. The top four lines declare a globalcode block that initializes a global counter record of the number of derivations. The postcode of the start symbol TwoBit increments the counter and prints its value. Part (b) shows the complete output of this grammar.

5.8 Terminal Generators

Some grammar variables contain long sequences of terminal alternatives. For example, a variable that produces all lower-case letters in the Roman alphabet would require a 26 separate rules to express with a grammar. Terminal Generators are shortcuts to expressing these types of sequences.

Terminal Generators may appear only in the body of a rule.

5.8.1 Range

- **Syntax.** `Range(start, skip, count)`

- **Semantics.** Generates count integers from start, incrementing the value by skip after each derivation.

- **Example.** Figure 5.7 shows a grammar that generates the integers 0 to 9.
Variables that need to be shared should be in Python’s `globals()` dictionary.

```python
globals()['numDerived'] = 0
}
```

```python
globals()['numDerived'] += 1
print '\tDerived',globals()['numDerived'],'strings'
}
```

```python
TwoBit ::= A B;
A ::= '0';
A ::= '1';
B ::= '0';
B ::= '1';
```

(a) Grammar with Globalcode

(b) YouGen output for grammar from (a)

Figure 5.6: Example of Globalcode Usage
S ::= Range(0, 1, 10);

Figure 5.7: Example of Range Terminal Generator

5.8.2 List

- **Syntax.** List(item1, item2, ..., itemN)

  Each item is a string enclosed in single quotes.

- **Semantics.** Generates each item given from left to right.

- **Example.** Figure 5.8 gives a grammar that generates six words.

  S ::= List('hello', 'world', 'you', 'are', 'my', 'sunshine');

  Figure 5.8: Example of List Terminal Generator

5.8.3 File

- **Syntax.** File(fileName)

  fileName is a path to an input file, which must contain a list of strings delimited by newlines.

- **Semantics.**

  For each line in fileName, one string is produced.

5.8.4 Custom Terminal Generators

Grammar developers can create their own terminal generators by creating a sub-class of YouGen’s `Literal` class inside the `globalcode` section. Custom terminal generators can be used multiple times in the grammar. The class must provide two methods:

- `__init__(self, paramList)`: constructor. paramList is a list containing all parameters specified for this terminal generator in the rule body.
• `generate(self)`: a generator function that is called when the rule containing the terminal generator is derived.

Figure 5.9 shows a grammar that utilizes a custom terminal generator to generate all lower-case letters in the Roman alphabet.

```python
{globalcode
class Lowercase(Terminal):
    def __init__(self, paramList):
        # Create a list of lowercase ASCII characters.
        self.slist = [chr(i) for i in range(97, 123)]

    def generate(self):
        # Produce each value in the list.
        for c in self.slist:
            yield c

Alphabet ::= Lowercase();
```

Figure 5.9: Example of a Custom Terminal Generator

## 5.9 Invocation

Running a grammar with YouGen is a two-step process, similar to the compile and run phases of programming languages such as Java. The grammar file must be translated into an executable Python program. When the Python program is run, strings in the language are generated.

For simplicity, YouGen provides a script that combines the two phases. The script’s single parameter provides the name of the grammar file to process. The file is translated into Python and saved to disk. The Python executable is run, with output sent to the console.
Chapter 6
YouGen Design and Implementation

This section describes the structure and implementation of YouGen. YouGen’s two phases are composed of a set of modules, each of which compartmentalize a related group of functions and classes. The following sections provide an overview of each module and lists the number of classes, functions, and lines of code. A line of code is considered any line that contains executable statements. Specifically, comments and blank lines are excluded.

6.1 Grammar Parser and Code Generation

YouGen modules are divided into two distinct groups: one group houses structures and routines for translating a grammar to Python, while the other contains algorithms and libraries for use during grammar runtime.

6.1.1 Modules

The translation process utilizes the following modules:

- Translator Module: converts a YouGen grammar to an executable program.
- Lexical Analysis Module: identifies and returns the sequence of tokens read from a grammar file.
- Parser Module: recognizes sequences of tokens and builds grammar data structures.
- Semantic Analysis Module: performs non-syntactic error checking.
- Code Generator Module: translates a grammar from YouGen’s internal structure to an executable Python program.
• Rule Database Module: stores the internal structure of YouGen’s grammar rules.

6.1.1.1 Translator Module
Lines of code: 30
Classes: 0
Functions: 1
In a similar fashion to compiled computer languages, YouGen grammars must be translated into an executable format before they can be run. The translator module is responsible for managing the steps involved in this process.

6.1.1.2 Lexical Analysis Module
Lines of code: 68
Classes: 4
Functions: 0
The lexical analysis module recognizes text tokens within a grammar file. The module uses a series of regular expressions to test sequences of characters for known patterns, then returns the sequence of tokens and some associated metadata. Metadata such as line numbers is helpful when generating friendly error messages.

6.1.1.3 Parser Module
Lines of code: 195
Classes: 3
Functions: 3
This module contains classes that interpret grammar elements (rule, tag, terminal generator, etc.) from sequences of tokens read by the lexical analyzer. Each class implements a recursive descent parser for the two main syntactic elements in a grammar file: rules and tags. Each parser returns a list of one or more objects recognized. If an invalid sequence of tokens is found, an error is generated.

The parser is the only module that interacts directly with the filesystem. It contains functions to load and save a grammar from a disk file.
6.1.1.4 Semantic Analysis Module

Lines of code: 35
Classes: 1
Functions: 5

This module checks for typos and errors in grammar body definitions. This module checks that all grammar variables that appear in the body of a rule also appear as the head of a rule.

6.1.1.5 Code Generator Module

Lines of code: 112
Classes: 0
Functions: 5

The code generator is responsible for turning the internal representation of a grammar into an executable Python program. The program contains all rules and embedded code, and some stock bootstrapping code to begin generation. The module converts grammar objects into Python code fragments and assembles them into a complete, syntactically correct program. Each rule is represented as a structure containing the head, a list of grammar variables and terminals in the body, and a set of tags.

Terminals are represented as classes containing a constructor and a generator function which returns one or more possible terminal values.

The set of tags is represented by a map of tag names to class instances that implement the behaviour of a tag. The class used depends on the tag. For example, if the grammar developer uses the cov tag, the code generator will produce code that creates an instance of YouGen.tags.cov and passes it the parameters given in the grammar file.

6.1.1.6 Tags Module

Lines of code: 102
Classes: 9
Functions: 0
This module implements the behaviour of YouGen’s tags. Each tag is defined in a class with three methods, each of which corresponds to a point in rule derivation where tag code needs to be maintained.

- **preRule**: Called before the first derivation of a rule. When a rule is about to be rewritten, the `preRule` code is invoked to check if the rule should be derived or not. For example, the `depth` tag uses this hook to check the depth of the tree and determine if derivation of the rule should continue.

- **pre**: Called before derivation of every string from a rule. This hook is similar to the `preRule`, except that it is called once for every new string that results from rule derivation. It determines if derivation should continue. It is generally used to terminate derivation before all strings from a rule have been generated. For example, the `count` tag uses this hook to check if the maximum number of strings has been reached.

- **post**: Called directly after derivation of a string. This hook is used to maintain or update the state of a tag. For example, the `count` tag must increment a counter whenever a new string is produced.

### 6.1.1.7 Rule Database Module

Lines of code: 20
Classes: 1
Functions: 1

The Rule Database Module contains the data structure used to store grammar rules at both translation time and execution time.

The Rule class is the only class in the module. It contains the attributes and metadata associated with a grammar rule:

- **lhs**: a string containing the name of the left-hand-side, or head of this rule.

- **rhs**: an ordered list of variables and terminals in the right-hand-side, or body of this rule. Variables and terminals are differentiated within the list by their data
type. Variables are identified by a string containing the name of the variable. Alternatively, terminals are identified by a reference to a terminal generator object.

- **postcode**: a reference to the postcode function invoked after this rule is derived. If no postcode tag is given for a rule, an empty function is used.

- **tags**: a hash that maps the name of a tag to an instance of the class that implements the tag’s behaviour.

### 6.1.2 Translation Procedure

Figure 6.1 shows the call graph of the Translator Module. Each line of the graph shows the name of a function. Child functions are shown tabbed to the right. Functions at the same tab location are executed sequentially from the top down. For example, **main** first calls **loadFile**, which in turn calls **open**, **read**, and **close**, in that order. The four main tasks performed by this module are shown in italics on the left.

The **main** module’s first task is to load the grammar file from disk. Because grammar files are normally quite small, the entire file is read into memory. Afterwards, the **Parsing** phase uses a recursive descent parser to read the rule database from the grammar file. The grammar parser uses a lexical analyzer to retrieve tokens in the grammar file. If a tag is encountered, a separate parser and lexical analyzer is used to read the tag and its parameter. This separation is needed because the syntax of tokens found inside a tag conflicts with the syntax of tokens found in the grammar proper. When an entire rule has been processed, the rule and all its contents are added to the rule database. If the end of file is reached, the entire rule database is returned.

At this point, the grammar is checked for semantic correctness. Common errors or typos are found at this stage. Some processing of the rule database is performed as well. For example, embedded code text is extracted from the **tags** hash and placed into a separate variable. Syntactically, embedded code is specified in the same way
main
    loadFile
    open
    read
    close

**Lexical Analysis**

GrammarParser.__init__
    LexicalAnalyzer.__init__
    LexicalAnalyzer.getNextToken

**Parsing**

GrammarParser.START
    TagParser.__init__
        LexicalAnalyzer.getNextToken
    TagParser.TAG
        TagParser.NAME
        LexicalAnalyzer.getCodeChunk
        TagParser.getNextToken
        TagParser.RB
        LexicalAnalyzer.advanceOffset
    GrammarParser.getNextToken
    GrammarParser.N
    GrammarParser.TAKES
    GrammarParser.RHS
        GrammarParser.TERM
        GrammarParser.CONSTR_T
        GrammarParser.RHS
    GrammarParser.SEMI

**Semantic Analysis**

checkRulesList
    checkRHSs
    checkEmbeddedCode
    checkCovTags

**Code Generation**

codeGen_Result
    codeGen_Header
    codeGen_GlobalCodeBlock
    codeGen_PostcodeBlocks
    codeGen_Terminal
    codeGen_ConstrTerminal
    codeGen_Tag
    codeGen_Footer

Figure 6.1: Call graph of grammar translation
as a tag, but its behaviour and invocation is much different.

The last step in the translation process is to convert the rule database to an executable Python program. The code generator module creates blocks of Python code for each aspect of a grammar: rules, tags, embedded code, and terminal generators. Code is included for loading YouGen’s libraries and beginning derivation at the start symbol.

6.2 Language Generation

After converting a grammar to an executable program, language generation can begin. The executable grammar can be run standalone on the command line. YouGen uses the following modules that serve as libraries during language generation:

- Tags Module: contains the data structures and implementations of YouGen’s tags.
- Covers Module: contains algorithms for generating one, two, three, and n-covers over a set of domains.
- Terminals Module: contains the implementations of YouGen’s terminal generators.
- Runtime Module: contains the primary generation algorithm and a set of utility functions.

6.2.1 Modules

6.2.1.1 Covers Module

Lines of code: 268

Classes: 2

Functions: 5

The covers module contains algorithms for generating covers over a set of domains. YouGen supports coverage of strength one, two, three, and n, where n is the number of elements in the body of the tagged rule. Because the procedure for generating covers varies significantly depending on the strength, this module contains separate
algorithms for each supported strength. When a `cov` tag is used, YouGen selects the required algorithm to use based on the strength value.

### 6.2.1.2 Terminals Module

Lines of code: 38

- Classes: 5
- Functions: 0

The terminals module contains the terminal generator interface and implementations of the three built-in terminal generators.

### 6.2.1.3 Runtime Module

Lines of code: 97

- Classes: 0
- Functions: 3

The Runtime module contains the main generation algorithm. This routine arbitrates the derivation process by interacting with tags and terminal generators, and running embedded code at the appropriate times. When a grammar is executed, the generation algorithm is used to obtain all derivations from the start symbol. This module interacts with all the generation modules to manage the state of tags, retrieve strings from terminals, and use coverage algorithms if requested.

### 6.2.2 Generation Procedure

Pseudocode for the YouGen generation algorithm is shown in Figure 6.2(a). The function `gen` takes a sentential form `S` as input and returns a sequence of strings, representing all strings that can be generated from `S` for grammar `G`. In the pseudocode, `S` is represented as a sequence of terminals and nonterminals, and angle brackets (`<>`) denote sequences. We use the standard sequence operators `head`, `tail`, and `~` for concatenation. We also use `R.lhs` and `R.rhs` to denote the head (a grammar variable) and the body (a sequence of terminals and nonterminals) of a rule `R`, and `+` for string concatenation.
gen(S)

result = <>

if S == <>

return result

if (head S) is a nonterminal

for rule R in G where R.lhs == head S

    for s0 in gen(R.rhs)

        for s1 in gen(tail S)

            result = result →< s0 + s1 >

        yield result

else

    for s1 in gen(tail S)

        result = result →< head S + s1 >

    yield result

(a) Standard generation

7.1 // generate sequence D of size R.rhs, with D[i] = L(R.rhs[i])
7.2 N = size(R.rhs)
7.3 for i in 0..N − 1
7.4 if R.rhs[i] is a terminal
7.5 D[i] = < R.rhs[i] >
7.6 else
7.7 D[i] = gen(< R.rhs[i] >)
7.8 // use a covering array algorithm on D
7.9 for each covering array specification C in the tag for rule R
7.10 result = result → cov(D,C)
7.11 yield result

(b) Covering array generation: replaces lines 7–10 in (a) above

Figure 6.2: Derivation algorithm in YouGen
### 6.3 Generated Code Example

```python
#!/usr/bin/env python
import sys
try:
    import YouGen
    from terminals import *
except ImportError, e:
    print >> sys.stderr, "Fatal Error: Unable to import YouGen libraries!
    print >> sys.stderr, "Check installation!"
    print >> sys.stderr, str(e)
    raise SystemExit, -1

class T0(Literal):
    def __init__(self):
        Literal.__init__(self, '0')

class T1(Literal):
    def __init__(self):
        Literal.__init__(self, '1')

class T2(Literal):
    def __init__(self):
        Literal.__init__(self, '0')

class T3(Literal):
    def __init__(self):
        Literal.__init__(self, '1')

rules = [
    YouGen.Rule('TwoBit', ['A', 'B'], {}),
    YouGen.Rule('A', [T0()], {}),
    YouGen.Rule('A', [T1()], {}),
    YouGen.Rule('B', [T2()], {}),
    YouGen.Rule('B', [T3()], {})
]
YouGen.rules = rules
if __name__ == "__main__":
    try:
        for s in YouGen.gen(["TwoBit"])�
            print YouGen.flatten(s)
    except KeyboardInterrupt:
        print >> sys.stderr, "Interrupted!"
```

Figure 6.3: An executable YouGen grammar.
Chapter 7
TCP Bad Flags Case Study

7.1 Firewall Configuration Problem

Firewalls are used as a gate to filter network traffic according to a set of rules. Configuring firewall rules correctly is very tricky and error-prone.

When designing a firewall ruleset, there is a tradeoff between security and functionality. To enforce maximum security, a ruleset will allow only certain kinds of traffic through. The drawback to this approach is that many user applications will not work. Most manufacturers of commercial, off-the-shelf firewall products err on the site of functionality. Most network applications will work, but some attacks have the potential of making it through the firewall and disrupting equipment on the inside.

Systematically testing a firewall is a difficult problem, particularly because defining the correct behaviour is difficult or impossible. A recent paper on firewall configuration errors showed that IT firewalls in major corporations often use flawed rulesets and are vulnerable to attack [27].

7.2 TCP Connections

7.2.1 TCP Flags

TCP is the most common protocols in use today. It provides reliable, stream-like communication between two endpoints on a network [6, 20]. A TCP session goes through several states during its lifetime. The state of the session is controlled by a set of six boolean flags within each TCP packet. Each flag has a three-letter name and has a specific meaning when set:

- **SYN**: synchronize two endpoints. SYN packets are only set during connection establishment.
- **ACK**: acknowledges the receipt of a packet.
• **FIN**: indicates that the sender has no more data to send, and the connection can be terminated.

• **PSH**: is set when the receiving TCP implementation should pass the data to the application immediately and not be buffered.

• **RST**: resets the connection.

• **URG**: indicates that some data in the packet is urgent and should be examined first. This flag is application-dependent and is rarely used.

Some well-known network attacks involve sending Transmission Control Protocol (TCP) packets with invalid flag combinations. Such packets are easy to create and are known to cause problems on some devices. They are also used to determine the operating system of a target device, which is useful for a later attack.

### 7.2.2 TCP Connection

TCP Connections are established by a process known as the three-way handshake. The solid lines in Figure 7.1 shows the packets involved in creating and destroying TCP connections. A connection is established when a sequence of three packets have been sent. The node that initiates the connection (often termed the *client*) sends a packet with the SYN flag set to another node (often called the *server*). The server responds with a packet with the SYN and ACK flags set. Finally, the client responds with a packet with just the ACK flag set. The connection is now initiated.

Termination of a connection begins when one side sends a packet with the FIN flag set. The other side of the connection must acknowledge the FIN, and may finish sending its data and follow with a corresponding FIN. Finally, the second FIN is acknowledged, and the connection is considered terminated.

Firewalls can filter TCP in two ways: stateless and stateful. Stateless filtering does not keep track of the state of a TCP connection as data passes through it. While having a higher throughput, these filters cannot block attacks that inject invalid state information at a specific point in the TCP session. Stateful firewalls keep track the
Figure 7.1: TCP connection and teardown sequence with possible bad flag positions
state and other information about each TCP connection, enabling them to filter some of these exploits.

### 7.2.3 Bad Flags

There are 64 possible flag combinations, but only a subset of those are ever valid. For example, a packet with both SYN and FIN set is illegal; there is no need to begin and end a connection at the same time. Of the 64 possible flag combinations, 46 combinations are never valid. Depending on the state of the TCP connection, additional combinations may be invalid as well.

Some TCP implementations are vulnerable to certain flag combinations. Vulnerabilities can cause the target device to lose network connectivity or completely reset. Other bad flag combinations are used to detect the target device’s operating system. For example, a Christmas tree packet, a packet with all flags set, are responded to in distinct ways by TCP implementations. These packets may also require more processing time, potentially causing a denial of service.

### 7.3 Test Generation

In most TCP connections, the client resides on the inside of a firewall and the server resides somewhere on the outside, e.g., the internet. A packet that comes from the outside of the firewall to the inside is called an inbound packet, while packets that are generated inside the firewall and travel outside are call outbound packets. TCP bad flag attacks are usually inbound, and may be sent at any point in the TCP connection. Figure 7.1 shows the TCP connection and termination process. Legitimate packets are shown as solid lines, while positions for potential bad flag packets are shown as dotted lines.

#### 7.3.1 The TCP Bad Flags Grammar

We created a grammar that generates a complete TCP connection. The grammar generates all packets necessary to establish and tear down a TCP connection, with bad flags injected at each of the seven positions shown in Figure 7.1. At each of the
seven positions, there are 46 possible bad flag combinations. There are $46^7$ or roughly 435 billion combinations, which would have taken over 138 years to exhaustively execute. A mixed-strength covering array was used to reduce the cartesian product to a manageable size.

Figure 7.2 gives the YouGen grammar that produces each connection. The grammar produces a set of abstract packets; a set of tokens that specify which TCP flags should be set. These are not the binary data sent on a network device; they are instructions to another program about how the binary packets should be created. Figure 7.3 shows the abstract packets for two connections.

Each test case begins with a separator, shown on line 1 in Figure 7.3. The other lines give the properties of an abstract packet. Each abstract packet begins with a name followed by six strings, each specifying the value of a TCP flag. If the string is capitalized, then that flag is set; otherwise it is unset. Next, the symbols \texttt{OUT} and \texttt{IN} give the direction the packet should be sent. Packets marked with \texttt{OUT} are sent in the outbound direction (client to server). Packets marked with \texttt{IN} are sent inbound. Finally, the correct behaviour of the firewall is given as one of \texttt{DROP} or \texttt{ACCEPT}.

\subsection*{7.3.2 Static Bad Flags}

We created a terminal generator to produce all 46 possible bad flag combinations. The generator is used to generate a bad flag packet before and after each legitimate packet. The generator contains a list of conditions for a bad flags packet. For each of the 64 total flag combinations, if the combination satisfies one of the conditions for illegality, it is generated.

\subsection*{7.4 Test Configuration}

Figure 7.4 depicts the test setup. The Test PC is a Dell Precision 390 with an 2.13Ghz Intel Core2 Duo and 2 gigabytes of RAM. The Test PC hosts YouGen and the test software responsible for execution and analysis of the tests. YouGen creates the abstract packets for the executor, which creates the proper binary packets and analyzes the firewall’s behaviour.
\{cov \[(1,3,5,7,9,11,13),1]\}\}

S ::= SEPARATOR BadFlags0 SYN BadFlags1 SYNACK BadFlags2 ACK1
BadFlags3 FINACK1 BadFlags4 FINACK2 BadFlags5 ACK2 BadFlags6;

SEPARATOR ::= '! ! ! ! ! ! ! ! !';
SYN ::= '\n' 'SYN' 'SYN' 'ack' 'fin' 'psh' 'rst' 'urg' 'OUT' 'ACCEPT';
SYNACK ::= '\n' 'SYNACK' 'SYN' 'ACK' 'FIN' 'psh' 'rst' 'urg' 'IN' 'ACCEPT';
ACK1 ::= '\n' 'ACK1' 'syn' 'ACK' 'fin' 'psh' 'rst' 'urg' 'OUT' 'ACCEPT';
FINACK1 ::= '\n' 'FINACK1' 'syn' 'ACK' 'FIN' 'psh' 'rst' 'urg' 'IN' 'ACCEPT';
FINACK2 ::= '\n' 'FINACK2' 'syn' 'ACK' 'FIN' 'psh' 'rst' 'urg' 'OUT' 'ACCEPT';
ACK2 ::= '\n' 'ACK2' 'syn' 'ACK' 'fin' 'psh' 'rst' 'urg' 'IN' 'ACCEPT';

BadFlags0 ::= BadFlag0;
BadFlag0 ::= '\n' 'badFlag0' StaticBadFlags() 'IN' 'DROP';
BadFlags1 ::= '\n' 'badFlag1' Flag1 'IN' 'DROP';
Flag1 ::= StaticBadFlags();
BadFlags2 ::= '\n' 'badFlag2' Flag2 'IN' 'DROP';
Flag2 ::= StaticBadFlags();
BadFlags3 ::= '\n' 'badFlag3' Flag3 'IN' 'DROP';
Flag3 ::= StaticBadFlags();
BadFlags4 ::= '\n' 'badFlag4' Flag4 'IN' 'DROP';
Flag4 ::= StaticBadFlags();
BadFlags5 ::= '\n' 'badFlag5' Flag5 'IN' 'DROP';
Flag5 ::= StaticBadFlags();
BadFlags6 ::= '\n' 'badFlag6' Flag6 'IN' 'DROP';
Flag6 ::= StaticBadFlags();

Figure 7.2: TCP Bad Flags Test Grammar

The test PC has two network interfaces, each masquerading as one side of a TCP connection. The LAN (or client) side resides on the local subnet and sends outbound packets. The WAN (or server) side is on the wide area subnet and sends inbound packets. The Device Under Test (DUT) forwards packets from one subnet to the other.

The executor uses raw sockets to send hand-crafted test packets. Raw sockets are an operating system facility that gives applications the power to create any sequence of bits and send them on a network interface [26]. Often, they are used to insert non-standard packet modifications that that are unavailable when using higher-level interfaces. The executor uses raw sockets to insert ”phony” network addresses into outgoing packets. Such packets eliminate interference from the operating system on the Test PC. To the DUT, packets creating using raw sockets are indistinguishable from packets generated by the kernel. There may be some protocol fields that have different numerical values, but nothing that can impact the handling of TCP packets.
badFlag0 syn ack fin psh rst urg IN DROP
badFlag1 syn ack fin psh rst urg IN DROP
badFlag2 syn ack fin psh rst urg IN DROP
badFlag3 syn ack fin psh rst urg IN DROP
badFlag4 syn ack fin psh rst urg IN DROP
badFlag5 syn ack fin psh rst urg IN DROP
badFlag6 syn ack fin psh rst urg IN DROP

badFlag0 syn ack fin psh rst URG IN DROP
badFlag1 syn ack fin psh rst URG IN DROP
badFlag2 syn ack fin psh rst URG IN DROP
badFlag3 syn ack fin psh rst URG IN DROP
badFlag4 syn ack fin psh rst URG IN DROP
badFlag5 syn ack fin psh rst URG IN DROP
badFlag6 syn ack fin psh rst URG IN DROP

Figure 7.3: Example output of TCP bad flags grammar
Figure 7.4: Test configuration
by the firewall.

7.4.1 Executor Pseudocode

Figure 7.6 shows the pseudocode for the test executor. The executor reads lines from the TCP bad flags grammar output, as shown in Figure 7.3. To avoid packet forwarding conflicts, each connection is given a unique TCP source port, specified by S. For each abstract packet in each connection, the corresponding binary packet is created. The binary packet has different IP addresses, MAC addresses (not shown), and TCP ports depending on whether the packet is inbound or outbound. After the binary packet is sent, the executor waits for 10 milliseconds to determine receipt of the packet on the other interface. If the packet is expected to be dropped and is received, an error is written to the output log. Conversely, if the packet is expected to be forwarded and is not received, an error is written. Before a new connection begins, S is incremented.

7.4.1.1 ARP

ARP is a ubiquitous protocol used to determine the MAC address of a foreign host based on its IP address. Because we are using phoney addresses, we could not rely on the operating system to respond to ARPs, so the executor responds to ARP requests from the DUT and supplies the phoney MAC address of the client or server ports.

7.4.1.2 Output

Figure 7.5 shows the output of the executor’s error log. During the first connection, the DUT issued ARP requests to both the client port and server port, but no incorrect behaviour was observed. All packets that should have been dropped were dropped and all packets that should have been forwarded were forwarded. Similarly, the second connection proceeded as expected. In connection 3, a bad flag packet in position 1 of Figure 7.1 was forwarded. Afterwards, all legitimate packets were unexpectedly dropped.
Connection 1
ARP request for 10.0.0.99 from 10.0.0.1
ARP Reply: 10.0.0.99 is at 01:02:03:04:05:06
ARP request for 192.168.1.99 from 192.168.1.1

Connection 2

Connection 3
!badFlag1: syn ack fin psh RST URG lineNum=32
!Missing packet! SYN ACK fin psh rst urg lineNum=33
!Missing packet! syn ACK fin psh rst urg lineNum=35
!Missing packet! syn ACK FIN psh rst urg lineNum=37
!Missing packet! syn ACK FIN psh rst urg lineNum=39
!Missing packet! syn ACK fin psh rst urg lineNum=41

Figure 7.5: Bad Flags executor raw output sample

7.5 Test Execution

This section discusses the methodology and rationale of our test procedure. We were not explicitly determined to find a critical bug in firewall software; we wanted to explore the software’s behaviour using YouGen. The prospect of finding defects was attractive, but was not the central aim of this test.

7.5.0.3 One-Cover

We started the test by using a one-cover to conduct a broad sweep of all bad flag positions. The one-cover, as exemplified on line 1 of Figure 7.2, generated 46 connections where each possible bad flag was used in one of the 7 positions precisely once. No interesting combinations of bad flags were produced, but this test allowed us to quickly determine the firewall’s handling of all bad flag packets at each of the seven positions.

Figure 7.7 shows the summarized results of the one-cover. For each of the seven bad flag packet positions (labeled 0 - 6), we checked to see if at least one of each of the 46 bad flag combinations was accepted by the firewall. If at least one such packet was accepted, the grid contains a 1, otherwise it contains a 0. Each bad flag
$S = 1025$

for each connection $C$

for each abstract packet $P$

if $P$ is outbound

create binary packet $B$:

IP source address: 192.168.1.104
IP dest address: 10.0.0.99
TCP source port: $S$
TCP dest port: 23

send $B$ to DUT LAN port

listen for $B$’s arrival on test PC WAN port, timing out after 10 ms.

if $B$ arrives

if $P$ contains DROP

write message to error log

else

if $P$ contains ACCEPT

write message to error log

else // $P$ is inbound

create binary packet $B$:

IP source address: 10.0.0.99
IP dest address: 10.0.0.2
TCP source port: 23
TCP dest port: $S$

send $B$ to DUT WAN port

listen for $B$’s arrival on test PC LAN port, timing out after 10 ms.

if $B$ arrives

if $P$ contains DROP

write message to error log

else

if $P$ contains ACCEPT

write message to error log

else

$S++$

Figure 7.6: TCP bad flags executor pseudocode
Figure 7.7: Summary of test results

During the one-cover, 42 of 46 combinations make it through at least once. In the first bad flags position (labeled with 0), bad flag packets are dropped because the firewall doesn’t have an entry in its forwarding database until the first SYN packet is sent. Notably, in position 1, the four combinations:

- \text{syn ack fin psh rst urg}
- \text{syn ack fin PSH rst urg}
- \text{syn ack fin psh rst URG}
- \text{syn ack fin PSH rst URG}

are never forwarded. Also, more bad flags get through at position 1 than any other. In particular, any packet with the RST flag set is forwarded at this position but no other. After a packet with the RST flag set, no additional packets are forwarded, including legitimate packets. Connection 3 in Figure 7.5 demonstrates this behaviour.
7.5.0.4 Full Two-Cover

Unfortunately, running a two-cover over the entire test set would have been impractical due to the computational complexity of YouGen’s two-cover algorithm. To illustrate this, we did some calculations using a series of arbitrary input domains and domain sizes. On the Test PC, a two-cover over seven domains of size 10 took 80 seconds to produce. After increasing the number of domains to 11, generation time rose to 130 seconds. With 15 domains, generation took over 11 minutes. Clearly, seven domains of size 46 would have been prohibitive to generate using YouGen’s two-cover algorithm.

7.5.0.5 Partial Two-Cover

To reduce generation time, we limited the scope of the two-cover to 3 bad flag positions. We also concentrated our tests on the bad flag positions that gave confounding results during the one-cover.

Figure 7.8 shows the differences between the one-cover’s behaviour and the partial two-cover’s behaviour. For each bad flag position, two lines are shown to compare the difference between each run. The results from the one-cover are shown on the first row, and results from the partial two-cover are shown on the bottom.

As with the one-cover, no bad flag packets are forwarded in position 0. Once again, the four combinations that were always dropped during the one-cover are always dropped during the two-cover.

Many more bad flag packets are getting through in positions 2 and 3. This is because during the one-cover, a packet with RST set between the SYN and the SYN, ACK packets would cause all subsequent packets to be dropped. In some connections, packets with RST set occur in positions 2 and 3 without a RST in position 1.

After analyzing the raw output, we could see that RST packets were not invalidating the connection when they are in positions 2 or 3. This result is unexpected, and indicates that the firewall has to deal with RST packets that follow a SYN. Commonly, RST is sent by a server or another firewall to tell a client that no server is available at the requested port.
7.5.1 Linksys Home Firewall Test

We ran the one-cover experiment against a common home firewall and gateway, a Linksys BEFSR41 home router. Once again, no bad flag packets got through in position 0 due to no entry in the forwarding table. In all other positions, every bad flag packet got through. The Linksys performs no filtering of bad flags at all. If legacy equipment is used in the home network, there is potential for abuse.

We could use these results to start a database of firewall forwarding behaviours for the purpose of fingerprinting. We could gather images such as Figure 7.7 and use them as part of firewall behaviour analysis tool. The tool would contain similar images from a variety of firewalls. Using that database, it would be possible to use the tool to guess the type and version of an unknown firewall.
Chapter 8
Conclusions and Future Work

8.1 Conclusions

Testing is an expensive and time-consuming process in software engineering. Automated testing reduces costs by lowering the amount of development time spent on producing and, especially, running tests. Covering arrays and context-free grammars are two methods of automatically creating test cases for a system. Traditionally, the two methods have been applied separately. We demonstrate that covering arrays and context-free grammars can be used together to discover results that are not detectable using either method on its own.

Context-free grammars are formal specifications of the syntax of context-free languages, and are ubiquitous in computer science. In testing, they define the set of input domains for a system.

Test input domains may be very large, making systematic execution of all test cases in the cartesian product impractical. A covering array is a subset of the cartesian product that guarantees interactions between a certain number of input domains are present in the resulting test cases. Covering arrays dramatically reduce the number of test cases and focus testing on system inputs that are most likely to produce a bug.

YouGen is a tool that features a simple syntax for defining context-free grammars and tags, a set of extra-grammatical constructs which alter a grammar’s derivation. By taking advantage of tags and embedded code, grammar developers can alter the results of a grammar’s derivation, effectively extending YouGen grammars to Turing power.

We used YouGen to test the behaviour of the open-source firewall package `iptables` stateful filtering module when presented with TCP packets containing bad flag combinations. We wanted to observe the firewall’s handling of these packets during each phase of a TCP conversation. We used a context-free grammar to generate the TCP
conversations, each containing “normal” packets to advance the state of the TCP conversation interspersed with “bad” packets. Because the number of potential test cases was greater than 400 billion, we used a mixed-strength covering array to target bad flag generation to the three phases of the TCP handshake which we predicted would have a higher likelihood of strange behaviour.

Using the same grammar and executor, we were able to test a COTS firewall’s bad flag handling and found that it does not drop TCP bad flag packets at all. We envision that it would be a simple task to use this platform to test a wide variety of firewalls, or to create a database of firewall fingerprints which could help identify running firewall software as part of a network security product.

From these results, we conclude that grammar-based test generation and mixed-strength covering arrays are a powerful and flexible testing solution.

8.2 Future Work

8.2.1 Merge multiple mixed-strength covering arrays

A mixed-strength covering array generates only those values that appear in the cover specification’s scope; all other domains are filled with arbitrary don’t care values. The results of multiple mixed-strength covering arrays are generated in the order given in the cov tag, and do not overlap. In many situations, sets of results could be combined to reduce the total number of cases generated while satisfying all mixed-strength cover specifications. An algorithm could recognize these situations and transparently merge the results of multiple cover specs.

8.2.2 Test additional applications

We want to apply YouGen to testing additional application domains. In particular, we intend to focus on testing XML parsers, XML document type interpreters and XQuery tools. XML is a common tree-like data serialization format that is has become widespread on the web, in database systems, and in many programming language. YouGen is well-suited to generate XML documents and predetermine how the document should be interpreted.
Bibliography


Appendix A: TCP Bad Flags Grammar

class StaticBadFlags(Terminal):
    # List of bad flag combinations.
    # Source: wallY.sh
    # Each element in the list is a 2-tuple consisting of:
    #   ( set_of_flags_under_consideration, set_of_flags_that_must_be_set)
    # where
    #   set_of_flags_that_must_be_set is a subset of set_of_flags
    # under consideration

    BADCOMBOS = [ ("ACK","FIN"),["FIN"]),
                  ("ACK","PSH"),["PSH"]),
                  ("ACK","URG"),["URG"]),
                  (["FIN","RST"],["FIN","RST"]),
                  (["FIN","FIN"],"FIN","FIN"),
                  (["SYN","FIN"],["SYN","RST"]),
                  (["SYN","RST"],["SYN","RST"]),
                  (["SYN","ACK","FIN","PSH","RST","URG"],
                   ["SYN","ACK","PSH","RST","URG"],
                   (["SYN","FIN","PSH","RST","URG"], []),
                  (["SYN","ACK","FIN","PSH","RST","URG"],
                   ["SYN","ACK","PSH","RST","URG"],
                   (["FIN","PSH","URG"]),
                  (["SYN","ACK","FIN","PSH","RST","URG"],
                   ["SYN","FIN","PSH","URG"]),
                  (["FIN","ACK","FIN","PSH","RST","URG"]),
                  (["SYN","RST","ACK","FIN","URG"]),
                  (["SYN","ACK","FIN","PSH","RST","URG"]),
                  (["SYN","RST","ACK","FIN","URG"])
    ]

    # List of names and corresponding positions of each flag.
    FLAGS = ["SYN","ACK","FIN","PSH","RST","URG"]

def __init__(self, paramList):
    pass

    # Generate all possible bit combinations of a given size.
def __genBits(self, numBits):
        if numBits == 0:
            yield []
            return
        for row in self.__genBits(numBits-1):
            yield [0] + row
        for row in self.__genBits(numBits-1):
            yield [1] + row

    # Get the index of a flag given its string name
def __getPosition(self, flagName):
        return StaticBadFlags.FLAGS.index(flagName)

    # Convert an array of bits to the canonical flag names with the

73
# appropriate case.
# lowercase indicates flag is unset
# uppercase indicates flag is set
def __bitsToFlags(self, bitList):
    l = []
    for i in range(len(bitList)):
        if bitList[i] == 1:
            l.append(StaticBadFlags.FLAGS[i].upper())
        else:
            l.append(StaticBadFlags.FLAGS[i].lower())
    return l

def generate(self):
    # For each possible flag combination,
    for flags in self.__genBits(6):
        # Try to find a bad flag combination that matches.
        for badcomb in StaticBadFlags.BADCOMBOS:
            matches = 0
            # Check that each flag under consideration has
            # the expected value.
            for flag in badcomb[0]:
                # Flag is supposed to be set.
                if flag in badcomb[1]:
                    if flags[self.__getPosition(flag)] == 1:
                        matches += 1
                # Flag is supposed to be unset.
                else:
                    if flags[self.__getPosition(flag)] == 0:
                        matches += 1
            # If all flags under consideration have the
            # expected value,
            # then this flag combination is bad.
            if matches == len(badcomb[0]):
                yield self.__bitsToFlags(flags)
                break

{cov [(3,5,7),2]}  
S ::= SEPARATOR BadFlags0 SYN BadFlags1 SYNACK BadFlags2 ACK1 BadFlags3 FINACK1 BadFlags4 FINACK2 BadFlags5 ACK2 BadFlags6;
SEPARATOR ::= '! ! ! ! ! ! ! ! !';

SYN ::= '\n' 'SYN' 'SYN' 'ack' 'fin' 'psh' 'rst' 'urg' 'OUT' 'ACCEPT';
SYNACK ::= '\n' 'SYNACK' 'SYN' 'ACK' 'fin' 'psh' 'rst' 'urg' 'IN' 'ACCEPT';
ACK1 ::= '\n' 'ACK1' 'syn' 'ACK' 'fin' 'psh' 'rst' 'urg' 'OUT' 'ACCEPT';
FINACK1 ::= '\n' 'FINACK1' 'syn' 'ACK' 'FIN' 'psh' 'rst' 'urg' 'IN' 'ACCEPT';
FINACK2 ::= '\n' 'FINACK2' 'syn' 'ACK' 'FIN' 'psh' 'rst' 'urg' 'OUT' 'ACCEPT';
ACK2 ::= '\n' 'ACK2' 'syn' 'ACK' 'fin' 'psh' 'rst' 'urg' 'IN' 'ACCEPT';

BadFlags0 ::= BadFlag0;
BadFlag0 ::= '\n' 'badFlag0' StaticBadFlags() 'IN' 'DROP';
BadFlags1 ::= '\n' 'badFlag1' Flag1 'IN' 'DROP';
Flag1 ::= StaticBadFlags();

BadFlags2 ::= '\n' 'badFlag2' Flag2 'IN' 'DROP';
Flag2 ::= StaticBadFlags();

BadFlags3 ::= '\n' 'badFlag3' Flag3 'IN' 'DROP';
Flag3 ::= StaticBadFlags();

BadFlags4 ::= '\n' 'badFlag4' Flag4 'IN' 'DROP';
Flag4 ::= StaticBadFlags();

BadFlags5 ::= '\n' 'badFlag5' Flag5 'IN' 'DROP';
Flag5 ::= StaticBadFlags();

BadFlags6 ::= '\n' 'badFlag6' Flag6 'IN' 'DROP';
Flag6 ::= StaticBadFlags();
Appendix B: TCP Bad Flags Executor Source Code

#include <iostream>
#include <map>
#include <string>
#include <fstream>
#include <sys/types.h> // For uint32_t, uint16_t etc..
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h> // For inet_aton() 
#include "ConfigFile.h"
#include "rawSocketToolkit.h"
#include "timevalOperators.h" 

#define PROMISC_MODE 1
#define BUFLEN 1514 
#define _DEBUG 

#define WAIT_SECONDS 0
#define WAIT_MS 10000 
#define RST_HDR_ALL RST_ETH_HDR|RST_ARP_HDR|RST_IP_HDR|RST_ICMP_HDR|RST_UDP_HDR |RST_TCP_HDR 

using namespace std;

struct test_values {
    unsigned char ifout_MAC[6]; // MAC address of LAN side
    uint32_t ifout_IP;
    uint16_t srcPort;

    unsigned char ifinp_MAC[6]; // MAC address of WAN side
    uint32_t ifinp_IP;

    unsigned char dut_MAC[6]; // MAC address of LAN side of DUT. 
    uint32_t dut_IP;

    unsigned char dut_WANMAC[6]; // MAC address of WAN side of DUT. 
    uint32_t dut_WANIP;

    uint16_t dstPort;

    struct timeval tv_wait;

    int outfd, inpfld;
};
```c
void usage(int argc, char *argv[]) {
    cerr << "Usage: " << argv[0] << " <cfgFile> <packetFile>" << endl;
}

// Function prototypes.
void CreateTestPacket(unsigned char *mem, bool syn, bool ack, bool fin,
    bool psh, bool rst, bool urg, bool lanside, struct test_values &testVals);
int ConvertTestProperties(map<string, string> cfgValues,
    struct test_values &testVals);
bool isUpper(string s);
int receiveLoop(int fd, struct test_values &testVals);

int main(int argc, char *argv[]) {
    // A list of configuration parameters that need to be read.
    const string cfgParams[] = {"LANIF", "WANIF", "LANIP", "LANMAC",
        "DUT_LANIP", "DUT_LANMAC", "WANMAC", "WANIP", "DUT_WANMAC", "DUT_WANIP",
        "DSTPORT", "SRCPORT"};

    map<string, string> cfgValues; // Map of cfgParam -> cfgValue
    ifstream inFile;
    unsigned long testNum = 0;
    unsigned long packets = 0;
    unsigned long packetNum = 0;
    unsigned long lineNum = 0;
    struct test_values testVals; // Contains addresses and other information
        // about the test packets.
    testVals.tv_wait.tv_sec = WAIT_SECONDS;
    testVals.tv_wait.tv_usec = WAIT_MS;

    // Packet data.
    unsigned char outpbuf[BUFLEN];

    // Check command-line parameters.
    if(argc != 3) {
        usage(argc, argv);
        return 1;
    }

    // Parse the configuration file
    ConfigFile config(argv[1]);

    // Read in all necessary configuration params, and store them in cfgValues
    for(unsigned int i=0; i<sizeof cfgParams / sizeof *cfgParams; i++) {
        string inVal;
        if( config.readInto(inVal, cfgParams[i]) == false ) {
            cerr << "Missing configuration parameter "
                << cfgParams[i] << "!" << endl;
            return 1;
        }
        cfgValues.insert( pair<string,string>(cfgParams[i], inVal) );
    }

    // Open the abstract packet file.
    if(strcmp(argv[2], "-") != 0) {
```
infile.open(argv[2]);
if( infile.fail() ) {
    cerr << "Unable to open '" << argv[2] << "'!" << endl;
    return 1;
}
} else {
    // TODO: If the command-line parameter is '-', use stdin as the
    // packet file.
}

// Convert configuration values to binary forms and put them in the
// testVals struct.
if(ConvertTestProperties(cfgValues, testVals) != 0) {
    return 1;
}

// Open the input and output devices in promiscuous mode.
testVals.inpfd = rst_open( (char*)cfgValues.find("LANIF")->second.c_str(),
    PROMISC_MODE);
if( testVals.inpfd < 0 ) {
    cerr << "Unable to open '" << cfgValues.find("LANIF")->second
    << "' in promiscuous mode!" << endl;
    return 1;
}

testVals.outfd = rst_open( (char*)cfgValues.find("WANIF")->second.c_str(),
    PROMISC_MODE);
if( testVals.outfd < 0 ) {
    cerr << "Unable to open '" << cfgValues.find("WANIF")->second
    << "' in promiscuous mode!" << endl;
    return 1;
}

#ifdef _DEBUG
    cerr << "Test starting!" << endl;
#endif

// For each abstract packet in the test case file...
while( infile.good() ) {
    string syn, ack, fin, psh, rst, urg, dir, behaviour;
    lineNumber++;
    // Read in an abstract packet, ignoring whitespace.
    infile >> skipws >> syn >> ack >> fin >> psh >> rst >> urg
    >> dir >> behaviour;

    // If we've encountered a separator, increment the dest. port
    // and restart the loop.
    if(syn.compare("") == 0) {
        testVals.srcPort++;
        testNum++;
        cerr << "\nStarting Test #" << testNum << " dstPort="
        << testVals.dstPort << " , srcPort="
        << testVals.srcPort << ", lineNum=" << lineNumber
        << endl;
    } else {
        // TODO: If the command-line parameter is '-', use stdin as the
        // packet file.
    }
continue;

// Create the corresponding binary frame.
CreateTestPacket(outpbuf,
    isUpper(syn),
    isUpper(ack),
    isUpper(fin),
    isUpper(psh),
    isUpper(rst),
    isUpper(urg),
    (dir.compare("OUT") == 0),
    testVals);

#ifdef __DEBUG
    cerr << syn << " " << ack << " " << fin << " " << psh << " "
        << rst << " " << urg << " " << dir << " " << behaviour << endl;
#endif

// If it was a OUT frame, send it on the OUTIF.
if(dir.compare("OUT") == 0) {
    if( rst_send(testVals.outfd, outpbuf, 60) != 0 ) {
        cerr << "Error sending on LAN interface!" << endl;
        return 1;
    }
}

// See if the packet made it to the other side.
packets = receiveLoop(testVals.inpfd, testVals);

// If this frame should have been accepted,
if(behaviour.compare("ACCEPT") == 0) {
    if( packets == 0 ) {
        cerr << "\tFAILURE: Missing packet! #"
            << packetNum "", lineNum="
            " << lineNum<< endl;
    }
} else {
    // If the packet should have been dropped,
    // error if received something.
    if( packets == 1 ) {
        cerr << "\tFAILURE: Packet " << packetNum
            " should have been dropped!"
            "", lineNum=" << lineNum << endl;
    }
}

// If it was a IN frame, send the packet on the INPIF.
else if(dir.compare("IN") == 0) {
    if( rst_send(testVals.inpfd, outpbuf, 60) != 0 ) {
        cerr << "Error sending on WAN interface!" << endl;
        return 1;
    }
}

// See if the packet made it to the other side.
packets = receiveLoop(testVals.outfd, testVals);
// If this frame should have been accepted,
if(behaviour.compare("ACCEPT") == 0) {
    if(packets == 0) {
        cerr << "\tFAILURE: Missing packet! #"
            << packetNum <<", lineNum="
            << lineNum<< endl;
    }
} else {
    // If the packet should have been dropped,
    // error if received something.
    if(packets == 1) {
        cerr << "\tFAILURE: Packet " << packetNum
            << " should have been dropped!"
            <<", lineNum=" << lineNum<< endl;
    }
}
else {
}
}
packetNum++;}

// Clean up.
infile.close();
rst_close(testVals.outfd);
rst_close(testVals.inpfd);
return 0;

// Is a string entirely in uppercase?
bool isUpper(std::string s) {
    for(unsigned int i=0; i < s.length(); i++) {
        if( (s.at(i) < 'A') || (s.at(i) > 'Z') )
            return false;
    }
    return true;
}

void CreateTestPacket(unsigned char *mem, bool syn, bool ack, bool fin,
            bool psh, bool rst, bool urg, bool lanside, struct test_values &testVals) {
struct rst_buf rBuf;
rst_initRbuf(&rBuf, mem, BUFLEN, RST_ETH_HDR|RST_IP_HDR|RST_TCP_HDR);
if(lanside) {
    // If the packet is to be sent on the LAN side.
    rst_initEth(rBuf.ethhdr, testVals.dut_MAC, testVals.ifout_MAC,
                ETH_P_IP);
rst_initIP(rBuf.iphdr, 40, IPPROTO_TCP, testVals.ifout_IP,
                testVals.ifinp_IP);
rst_initTCP(rBuf.tcphdr, testVals.srcPort, testVals.dstPort);
} else {
    // else if the packet is to be sent on the WAN side.
rst_initEth(rBuf.ethhdr, testVals.dut_WANMAC,
testVals.ifinp_MAC, ETH_P_IP);
rst_initIP(rBuf.iphdr, 40, IPPROTO_TCP, testVals.ifinp_IP,
testVals.dut_WANIP);
rst_initTCP(rBuf.tcphdr, testVals.dstPort, testVals.srcPort);

} // Set TCP flags accordingly.
rBuf.tcphdr->syn = syn;
rBuf.tcphdr->ack = ack;
rBuf.tcphdr->fin = fin;
rBuf.tcphdr->psh = psh;
rBuf.tcphdr->rst = rst;
rBuf.tcphdr->urg = urg;
rst_htonp( &rBuf );

int receiveLoop(int fd, struct test_values &testVals) {
    struct rst_buf rbuf;
    int plen;
    unsigned char inpbuf[BUFLEN];
    struct timeval tv_time = testVals.tv_wait;

    // Initialize receive buffer
    rst_initRbuf( &rbuf, inpbuf, BUFLEN, RST_HDR_ALL );

    while((plen = rst_receive( fd, inpbuf, &tv_time)) > 0) {
        tv_time = testVals.tv_wait;
        // A packet was received.
        #ifdef __DEBUG
        cerr << "Packet received! Length: " << plen
             << " New wait time: " << tv_time << endl;
        #endif
        // Do network-to-host conversions
        rst_ntohp( &rbuf );

        switch(rbuf.ethhdr->h_proto) {
        // If we receive an ARP, construct a reply.
        // We can't rely on the kernel to reply for us,
        // since we are using spoofed IP addresses.
        case ETH_P_ARP:
            if(rbuf.arphdr->opCode == ARPOP_REQUEST){
                unsigned char arpreplybuf[BUFLEN];
                struct rst_buf arbuf;

                #ifdef __DEBUG
                char *ipaddr = inet_ntoa((const in_addr&)rbuf.arphdr->targetIP);
                cerr << "\tARP request for " << ipaddr;
                ipaddr = inet_ntoa((const in_addr&)rbuf.arphdr->senderIP);
                cerr << " from " << ipaddr << endl;
                #endif

            }
        
        default:
            // unknown protocol
            break;
        }
    }
}


// Initialize send buffer
rst_initRbuf(&arpbuf, arpreplybuf, BUFLEN,
RST_ETH_HDR|RST_ARP_HDR);
rst_initEth(arpbuf.ethhdr,rbuf.ethhdr->h_source,
testVals.ifinp_MAC, ETH_P_ARP);
rst_initARPreply(arpbuf.arphdr,
testVals.ifinp_MAC, // senderMAC
testVals.ifinp_IP, // senderIP
rbuf.arphdr->senderMAC, // targetMAC
testVals.dut_WANIP); // targetIP

def __DEBUG
ipaddr = inet_ntoa(
(const in_addr&)arpbuf.arphdr->senderIP);
char macstring[18];
rst_MACitoa(arpbuf.arphdr->senderMAC,macstring);
cerr << "ARP Reply: " << ipaddr << " is at "
<< macstring << endl;
#endif

if( rst_send(fd, arpreplybuf, 60) != 0) {
  cerr << "Problem sending ARP reply!"
  << endl;
}
break;

#ifdef __DEBUG
#endif

if(rbuf.iphdr->protocol == IPPROTO_TCP) {
  cerr << "TCP packet from "
  << inet_ntoa((const in_addr&)rbuf.iphdr->saddr)
  << ":" << rbuf.tcphdr->source;

cerr << " to "
  << inet_ntoa((const in_addr&)rbuf.iphdr->daddr)
  << ":" << rbuf.tcphdr->dest << endl;
} else {
  cerr << "ip_proto: " << (int)rbuf.iphdr->protocol
  << endl;
}
#endif

if(rbuf.iphdr->protocol == IPPROTO_TCP) {
  if(fd==testVals.outfd) {
    if((rbuf.tcphdr->dest==testVals.srcPort)
    &&(rbuf.tcphdr->source
    ==testVals.dstPort)) {
      return 1;
    }
  }else if(fd==testVals.inpfd) {
    if((rbuf.tcphdr->dest==testVals.dstPort)
    &&(rbuf.tcphdr->source
    ==testVals.srcPort)) {
      return 1;
    }
if(fd == testVals.outfd) {
    if(rbuf.iphdr->daddr != testVals.ifout_IP) {
        break;
    }
}

default:
    #ifdef _DEBUG
    cerr << "Unknown packet received! eth_proto == "
    << (int)rbuf.ethhdr->h_proto << endl;
    #endif
    break;
}

#ifdef __DEBUG
    cerr << "No packet received!" << endl;
#endif
return 0;

int ConvertTestProperties(map<string, string> cfgValues,
                          struct test_values &testVals)
{
    if( rst_MACatoi((char*)cfgValues.find("LANMAC")->second.c_str(),
                    testVals.ifout_MAC) ) {
        cerr << "Invalid format: LANMAC" << endl;
        return 1;
    }
    if( rst_MACatoi((char*)cfgValues.find("WANMAC")->second.c_str(),
                    testVals.ifinp_MAC) ) {
        cerr << "Invalid format: WANMAC" << endl;
        return 1;
    }
    if( rst_MACatoi((char*)cfgValues.find("DUT_LANMAC")->second.c_str(),
                    testVals.dut_MAC) ) {
        cerr << "Invalid format: DUT_LANMAC" << endl;
        return 1;
    }
    if( rst_MACatoi((char*)cfgValues.find("DUT_WANMAC")->second.c_str(),
                    testVals.dut_WANMAC) ) {
        cerr << "Invalid format: DUT_WANMAC" << endl;
        return 1;
    }
    if( !inet_aton((const char*)cfgValues.find("LANIP")->second.c_str(),
                    (struct in_addr*)&testVals.ifout_IP)) {
        cerr << "Invalid format: LANIP" << endl;
        return 1;
    }
    if( !inet_aton((const char*)cfgValues.find("WANIP")->second.c_str(),
                    (struct in_addr*)&testVals.ifinp_IP)) {
        cerr << "Invalid format: WANIP" << endl;
        return 1;
    }
return 1;
}
if( !inet_aton((const char*)cfgValues.find("DUT_LANIP")\->second.c_str(),
(struct in_addr*)&testVals.dut_IP)) {
    cerr << "Invalid format: DUT_LANIP" << endl;
    return 1;
}
if( !inet_aton((const char*)cfgValues.find("DUT_WANIP")\->second.c_str(),
(struct in_addr*)&testVals.dut_WANIP)) {
    cerr << "Invalid format: DUT_WANIP" << endl;
    return 1;
}

testVals.srcPort = atoi((const char*)
cfgValues.find("SRCPORT")\->second.c_str());
testVals.dstPort = atoi((const char*)
cfgValues.find("DSTPORT")\->second.c_str());
return 0;