PBit - A Pattern Based Testing Framework for Linux Iptables

By

Yong Du
B.Sc., Wuhan University, 1996

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

MASTER OF SCIENCE

in the Department of Computer Science

University of Victoria

Supervisor: Dr. D.M. Hoffman and Dr. P. Walsh

ABSTRACT

Firewall testing is important because firewall faults can lead to security failures. Firewall testing is hard because firewall rules have parameters, producing a huge number of possible parameter combinations. This thesis presents a firewall testing methodology based on test templates, which are parameterized test cases. A firewall testing framework for iptables, the Linux firewall subsystem, has been implemented. Twelve test templates have been created for testing iptables parameters and extensions. A GUI tool is also provided to integrate these test templates with various test generation strategies. The most important of these strategies, pairwise generation, has been investigated in detail. Based on the investigation, we developed an improved pairwise generation algorithm.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my supervisors, Dr. Daniel M. Hoffman and Dr. Peter Walsh, for their strong support and valuable instruction during my graduate program at the University of Victoria. I must also acknowledge NSERC for their financial support, without which my thesis would not have been finished. Finally, I would like to acknowledge my family's endless support.

Chapter 1. Introduction

Linux iptables is the current Linux firewalling subsystem. As Linux is accepted by more users and network security problems become more common and serious, it is becoming popular to use iptables to set up firewalls for computer networks.

Firewall testing is important because a buggy firewall puts security at risk. Firewall testing is also hard because it involves a lot of parameters, which may produce a huge number of possible parameter combinations. Another difficulty of iptables testing is introduced by the open source property of iptables. Iptables, like many other open source applications in Linux, provides great user programmability and extensibility. This means that user defined modules can be added to iptables or the Linux kernel by any user, providing specific functionality that the user wants. Many of these iptables modules have been made available on the Internet to share with other iptables developers and users. It is likely these modules will become popular and widely distributed without having been thoroughly tested. As a result, potential bugs in these iptables modules may lead to serious security holes in the Linux firewalling subsystem, and a huge number of users may be affected.

This thesis research provides a regression testing framework and a test suite for Linux iptables. Since new iptables extensions are coming out frequently, a test application that covers all the current iptables extensions is likely to become obsolete very quickly. For this reason, we are not interested in creating a fixed test suite. Our objective is to create a testing framework with a set of carefully selected test patterns that can be reused for testing any iptables extension. The underlying test methodology is based on test templates, which are parameterized test cases. Each test template is designed to test a subset of iptables functionalities. This subset of functionalities should be carefully selected to achieve reasonable granularity. If the test templates are too finely grained, a large number of test templates may have to be created, too many to be managed effectively. On the other hand, if the granularity is too large, each test template may cover too many functionalities and is thus

hard to evolve and reuse. Since iptables rules are organized by parameters and extensions, and the coupling between these parameters and extensions is very low, we have decided to design and implement one test template for each iptables parameter or extension.

Test templates are associated with test generation strategies, which aim at identifying patterns in a large test set. In our iptables testing framework, we have introduced three test generation strategies: Cartesian product generation, boundary values generation, and pairwise generation. Furthermore, we investigated pairwise generation in detail and found an important property that can improve pairwise generation strategies relying on the order of input parameters. Based on the investigation, we developed an improved algorithm for pairwise generation. Experiments show that the improved algorithm generates test sets better than or at least as good as the original algorithm. In several test scenarios, the improved algorithm generates test sets better than a well-known commercial product.

Twelve test templates have been created in order to demonstrate the practicality and effectiveness of our iptables testing framework. These test templates cover the testing of six iptables parameters and seven iptables extensions. Patterns for designing and implementing test templates have been identified and summarized. These patterns can be quickly followed in order to create test templates for new iptables extensions.

All three of the test generation strategies described above, as well as the twelve test templates for testing iptables, have been implemented in a test tool called PBit (Pattern Based iptables tester). PBit provides GUI dialogue boxes for all test templates and generates test cases based on user input. Test cases are generated according to the associated test generation strategies. Patterns for designing and implementing GUI dialogue boxes are also summarized in order to create GUI dialogue boxes for new test templates.

The contributions of this thesis research are summarized as follows:

1. A testing framework for Linux iptables has been developed. This testing framework is based on test templates, which are parameterized test cases.

2. Twelve test templates have been created for testing iptables parameters and exten-

sions.

3. Pairwise test generation has been investigated in detail. Based on the "Order-Irrelevance" property of pairwise generation, we proposed an improvement of a pairwise generation algorithm. We have implemented both the original algorithm before the improvement and our improved algorithm. A number of experiments have shown that the improved algorithm generates better or at least the same test results as the original algorithm. In a few test scenarios, the improved algorithm generates smaller pairwise tests sets than a commercial tool.

4. A GUI tool called PBit is provided to integrate the twelve iptables test templates with various test generation strategies. Three test generation strategies are available: Cartesian product generation, boundary values generation, and pairwise generation.

5. The PBit testing framework is easy to extend by creating test templates for new iptables extensions.

6. Patterns in abstract test generation are pure mathematical models, and thus are easy to be reused in other software testing systems.

The remainder of the thesis is organized as follows:

- Chapter 2 defines terms and concepts used in our research. Basic definitions of network and firewalls are given, followed by an introduction to the Linux firewalling subsystem. Test templates and several test generation strategies are also explained.

- Chapter 3 introduces related work done by other researchers. We focus on tuple generation, and introduce some network testing approaches. We also discuss socket libraries supporting packet generation.

- Chapter 4 investigates pairwise generation in detail and introduces the Order-Irrelevance property of pairwise generation. An improved pairwise generation algorithm is proposed and some test results are presented.

- Chapter 5 lists the twelve test templates used in PBit for testing iptables. We explain one of these test templates in detail.
- Chapter 6 describes implementation details and the GUI features of PBit. Three GUI dialogue boxes are explained and the design idea of these dialogue boxes is discussed. Some test results by using PBit are also presented.
- Chapter 7 concludes this thesis and describes the future research work.
- Appendix A contains the Java implementation of the improved pairwise generation strategy used in PBit.
- Appendix B contains the test programs for the pairwise generation algorithm shown in Appendix A.
- Appendix C contains the implementation of the Java raw socket library used in PBit.

Chapter 2. Terms and Concepts

This chapter defines the terms and concepts used in our research. We will start with some terms used in the field of computer networks and firewalls in general, followed by those specific to the Linux firewall subsystem. Finally, we will give some definitions related to test generation.

2.1 Network and firewall basics

All Internet traffic is sent in the form of *packets*. A packet contains a header section and a body section. The *header* section contains administrative information of the packet, such as the type, the length, and the checksum. The *body* section contains the data that needs to be transmitted. A *firewall* is a software system, or a hardware device, which restricts access between two networks: the *internal network* and the *external network*. Figure 2.1 shows the structure of real network connections in a firewall system. In this figure, the Internet acts as the external network and a LAN acts as the internal network. A firewall is normally configured to restrict unexpected access coming from the external network, but it may also be set to limit accesses from the internal network to the external network. A packet received by a firewall from the external network is called an *inbound* packet. A packet received by a firewall from the internal network is called an *outbound* packet. An *abstract packet* is the abstract representation of one or more actual packets by specifying the primary attributes of the actual packets. For example, an abstract UDP packet specifies the source IP address, the destination IP address, the source port, and the destination port, representing all UDP packets with the specified four attributes.

2.2 Linux firewall

Netfilter is a software firewalling framework built into the Linux kernel. The operating mechanism of netfilter is based on *packet filters*, which are programs that monitor the header section of each packet as the packet passes by, and determine whether to accept

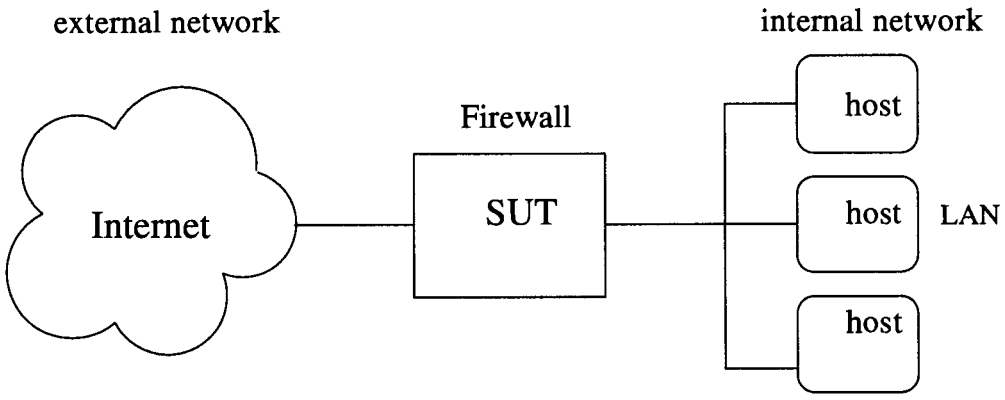

Figure 2.1: Structure of real firewall connections

or reject the packet. Details of packet filtering can be found in [24]. Netfilter maintains a few packet filtering tables, containing rules specifying what to do for each kind of packet.

Iptables is a front-end for netfilter introduced into Linux 2.4 or above. Through iptables, a user can insert or delete rules to or from the packet filtering tables maintained by netfilter, and thus control the operation of the Linux firewalling subsystem. Before Linux 2.4, ipfwadm and ipchains provided functionality similar to iptables.

A *chain* is a list of packet filtering rules maintained by iptables. The netfilter framework has five builtin chains: the PREROUTING chain, the INPUT chain, the FORWARD chain, the OUTPUT chain, and the POSTROUTING chain. The organization of these chains is shown in Figure 2.2. Our research focuses on the FORWARD chain. The FORWARD chain is traversed if the packet is received at one network interface and is going to be sent out through another network interface.

A *rule* in the FORWARD chain usually contains two parts: one part defines the attributes of a matched packet, and the other specifies what to do with a packet if the packet matches. The second part is also called the *target* of the rule, which can be a user-defined chain, or an *extension*, which is a module added to iptables providing extended functionality. An extension used in the first part of a rule is called a *match extension*. An extension used as a target is called a *target extension*. Default targets defined by netfilter are ACCEPT, DROP, and LOG. As an example, the following iptables command adds a rule which will accept

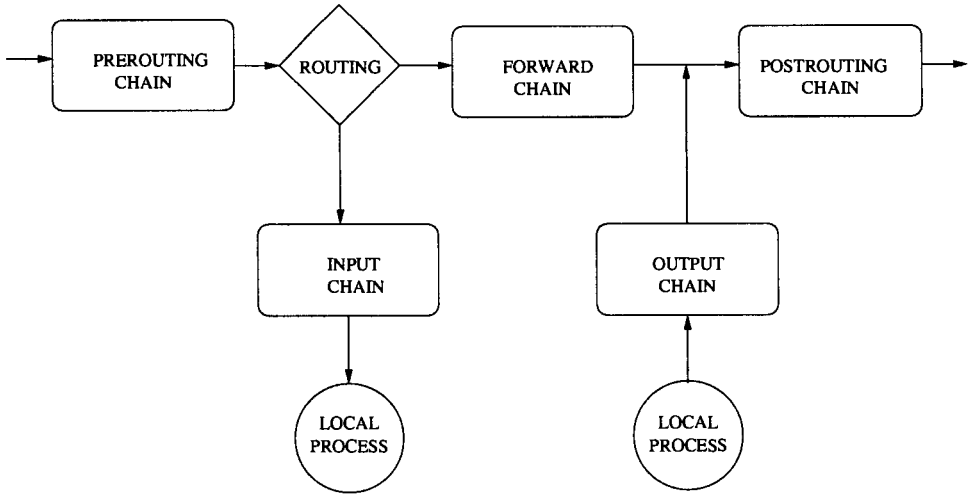

Figure 2.2: Organization of netfilter chains

TCP packets on the FORWARD chain.

```
iptables -A FORWARD -p tcp -j ACCEPT
```

In order to test the FORWARD chain, the System Under Test (SUT) should be connected to the internal and external networks, as illustrated in Figure 2.1. For testing purposes, real internal and external networks are unsuitable because of the following reasons:

- On one hand, having real internal and external networks is impractical since
  - it is costly to configure two networks with multiple hosts, and
  - it is not easy to compare the expected test result with the actual test result.
- On the other hand, it is unnecessary to have both the internal and the external networks since
  - the SUT only cares about the test packets sent outbound and inbound, with no regard to the structures of the internal network and the external network.

Based on the analysis above, we use one Linux box, denoted by the *driver machine*, to function as a host in both the external and the internal networks. Both the SUT and the driver machine have three network interfaces, denoted by eth_0, eth_1, and eth_2. Figure 2.3

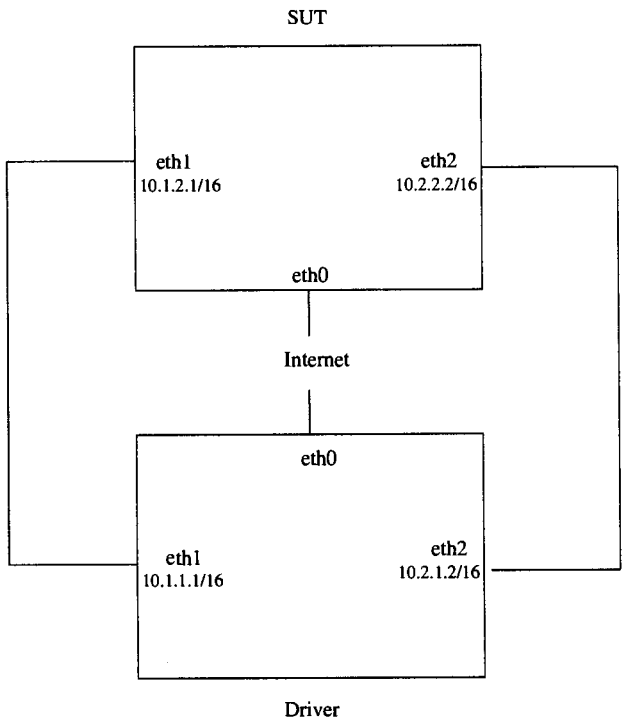

Figure 2.3: Structure of the test system

illustrates this test configuration. Eth_1 of the SUT acts as the interface to the internal network and is connected with eth_1 of the driver machine. Eth_2 of the SUT acts as the interface to the external network and is connected with eth_2 of the driver machine. Eth_0 of the SUT is connected to eth_0 of the driver machine through the Internet and is used to configure iptables rules on the SUT. The driver machine generates outbound packets through its eth_1 interface and inbound packets through its eth_2 interface. With this test configuration, the driver machine is the only place where the tester generates abstract packets, transmits actual packets, and analyzes packets received.

2.3 Test template

A *test template* is a parameterized test case. In this thesis, a test template T is denoted by $T(p_1, p_2, ..., p_n)$ where p_i is the i^{th} parameter of T for $i \in [1, n]$. The set of input values for a parameter p_i is called the *input domain* of p_i. Given a test template $T(p_1, p_2, \ldots, p_n)$ with n correspondent input domains $D_1, D_2, \ldots, D_n$, a *test tuple* of T is an n-tuple $\langle v_1, v_2, \ldots, v_n \rangle$

with $v_i \in D_i$ for $i \in [1, n]$. A set of test tuples is called a *test set*. The test set containing all possible test tuples is called the *test space*. A test case is a test template applied with a test tuple. The process of generating test sets is called *tuple generation*.

The definitions given above may be explained by an example. The following test template has been created in order to test the `--protocol` iptables parameter:

- ProtocolTest(rule-protocol,test-protocol,direction)

The input domains of this test template may be given as:

- rule-protocol: {tcp,udp}
- test-protocol: {tcp,udp,icmp}
- direction: {inbound,outbound}

Based on the input domains given above, the following are two test tuples:

- ⟨udp,udp,inbound⟩
- ⟨udp,tcp,outbound⟩

In the iptables testing, the number of input domains is always less than five. The size of an input domain ranges from 1 to 65535.

The number of test tuples generated for a test template depends on the strategy used in tuple generation. In the next section, we introduce the three tuple generation strategies used in our testing framework.

2.4 Tuple generation strategies

In this section, we introduce three strategies used for tuple generation: Cartesian product generation, boundary values generation, and pairwise generation.

2.4.1 Cartesian product generation

Cartesian product generation is the simplest tuple generation strategy [2, 16]. The test set generated by the Cartesian product strategy is the test space.

Given a test template $T(p_1, p_2, \ldots, p_n)$ and correspondent input domains $D_1, D_2, \ldots, D_n$, the test set S generated by the Cartesian product strategy is the Cartesian product of the n input domains, i.e.,

$$S = \prod_{1 \leq i \leq n} D_i$$

It is obvious that the number of test tuples in S is $|D_1| \times |D_2| \times \ldots \times |D_n|$. Consider the `ProtocolTest` example given in the previous section, the number of test tuples generated by the Cartesian product strategy will be $2 \times 3 \times 2 = 12$.

The Cartesian product strategy generates the test space for exhaustive testing. Sometimes the test space generated is so large that exhaustive testing is impractical. For example, suppose there are 10 input domains and each input domain contains 10 elements. Then Cartesian product generation will generate the test space of size 10^{10}. If it takes 10 milliseconds to execute one test case, then running all the test cases will take more than 10 years. In these cases, more specific generation strategies are used to decrease the size of the test set.

2.4.2 Boundary values generation

Boundary values generation is a common test generation strategy used in software testing [15, 23, 8]. We define the *boundary* of an input domain D, denoted by boundary(D), as the subset of D containing the minimum and the maximum values in D. If D contains two or more elements, boundary(D) will contain two elements. If D contains exactly one element, boundary(D) will contain one element. boundary(D) will be empty if D is empty. Based on this definition, the test set S generated by the boundary values generation strategy is obtained as follows: Given a test template $T(p_1, p_2, \ldots, p_n)$ with n correspondent input domains $D_1, D_2, \ldots, D_n$, S is the Cartesian product of the boundaries of the given n input

domains, i.e.,

$$S = \prod_{1 \leq i \leq n} boundary(D_i)$$

The number of test tuples in S depends on n and the size of the boundary of each input domain. This number is always less than or equal to 2^n. As an example, consider the following three input domains:

- A: [1,10]
- B: [1,100]
- C: [1,25]

It is clear that the size of the test space generated by Cartesian product generation will be $10 \times 100 \times 25 = 25,000$. Using boundary values generation, we will get a test set with only $2 \times 2 \times 2 = 8$ test tuples.

Boundary values generation is simple and relatively easy to implement. This test generation strategy reduces the size of the test set significantly, but the drawback is that it does not work with unordered input domains. Consider the `ProtocolTest` example given in the previous section, it is meaningless to talk about the boundary values of the given three input domains. In the next section, we will introduce a more complicated test generation strategy without this drawback.

2.4.3 Pairwise generation

Pairwise generation, also known as two cover generation or 2-way generation, has been used in various software testing systems. Pairwise generation is efficient because the growth rate of the test set size is logarithmic [30]. Experiments have shown that "most field faults were caused by either incorrect single values or by an interaction of pairs of values" [5]. In [19], Kuhn and Reilly experimented k-cover testing on a browser and Web server; and they found that "the browser and server software were similar in the percentage of errors detected by combinations of degree 2 through 6". These test results indicate that pairwise generation provides sufficient test coverage.

Caller	Type	Market	Callee
Regular	Local	Canada	Regular
Cell phone	Long distance	US	Cell phone
Coin phone	Toll free	Mexico	Pager

Table 2.1: A System Test Scenario (courtesy of A.W. Williams)

Given a test template $T(p_1, p_2, \ldots, p_n)$ with correspondent input domains $D_1, D_2, \ldots, D_n$, a pairwise test set S is a subset of $D_1 \times D_2 \times \ldots \times D_n$ such that for each element x in domain D_i and y in domain D_j, $(i, j \in [1, n]\ and\ i \neq j)$, there is at least one test tuple in S with x in position i and y in position j.

To explain the definition of pairwise generation more clearly, let us consider an example taken from [35]. Suppose a telephone company plans to test its telephone system. Four parameters are identified, with three values in each input domain, as shown in Table 2.1. To test all calling scenarios, Cartesian product generation should be used and $3^4 = 81$ test cases will be generated, corresponding to 81 phone calls. Boundary values generation is not applicable here since none of the input domains can be ordered. The approach used by pairwise generation is to create a test set that covers all pairwise combinations of the input domains. For example, there should be at least one phone call with the cell phone as the caller and the regular phone as the callee, and there should be at least one phone call of type long distance and with Canada as the market, etc. Table 2.2 shows a test set satisfying the pairwise coverage. This test set contains only 9 test tuples, much smaller than the test set generated by the Cartesian product strategy.

For the same `ProtocolTest` example, a pairwise test set may be created containing the following six test tuples:

- ⟨tcp,tcp,inbound⟩
- ⟨tcp,udp,outbound⟩
- ⟨tcp,icmp,inbound⟩
- ⟨udp,tcp,outbound⟩

#	Caller	Type	Market	Callee
1	Regular	Local	Canada	Regular
2	Regular	Long distance	US	Cell phone
3	Regular	Toll free	Mexico	Pager
4	Cell phone	Local	US	Pager
5	Cell phone	Long distance	Mexico	Regular
6	Cell phone	Toll free	Canada	Cell phone
7	Coin phone	Local	Mexico	Cell phone
8	Coin phone	Long distance	Canada	Pager
9	Coin phone	Toll free	US	Regular

Table 2.2: Test configurations for the scenarios in Table 2.1 (courtesy of A.W. Williams)

- ⟨udp,udp,inbound⟩
- ⟨udp,icmp,outbound⟩

From the tester's point of view, pairwise generation is interesting because it can significantly decrease the size of the test set. What makes pairwise generation more interesting is that for a given test template with correspondent input domains, there may be multiple test sets satisfying the pairwise coverage. It is easy to see that the test space is always a pairwise test set, and it is also the pairwise test set with the maximum number of test tuples. As software testers, we expect the pairwise test set to be as small as possible so that less test cases will be executed.

Various pairwise generation algorithms have been proposed, but as we have investigated so far, no algorithm is guaranteed to always generate the minimum pairwise test set. AETG [5] is one of the most well-known commercial test tools using pairwise generation. Stevens and Mendelsohn proposed a few pairwise generation approaches based on covering arrays [30]. In [6], a structure called variable strength covering array was used to generate pairwise test sets. Another pairwise generation algorithm called In-Parameter-Order (IPO) [32] was proposed by Tai and Lei. We will investigate pairwise generation in detail and propose an improved algorithm based on IPO in chapter 4.

We have described enough background knowledge of our research. In the next chapter, we will introduce some related work that have been done by other researchers.

Chapter 3. Related Work

Many researchers have worked on the field of test generation and network testing. The first section introduces related work in the field of tuple generation, and the second section describes related work in network testing.

3.1 Tuple generation

The testing framework for iptables built in this research takes advantage of Roast [8], a testing framework supporting automated testing of Java APIs. Roast supports a few tuple generation strategies, including Cartesian product generation, boundary values generation, and perimeter generation. Each tuple generation strategy is designed as an iterator, which, upon invocation, returns the next available test tuple. Our iptables testing framework reuses the Cartesian product generation strategy and the 1-boundary values generation strategy in Roast. Implementations of the pairwise generation strategies used in our testing framework follow the iterator pattern used in Roast.

Roast does not provide pairwise generation, which is the primary test generation strategy used in the AETG testing tool. AETG stands for Automatic Efficient Test Generation system and is a commercial test tool [5, 4]. AETG focuses on pairwise generation since pairwise test sets are considered powerful enough to reveal potential errors. The underlying algorithm used by AETG works in a greedy fashion and thus is not guaranteed to generate minimum test sets. Tai and Lei have run experiments on AETG and found that in at least two test scenarios, AETG generated larger pairwise test sets than the IPO algorithm [31].

IPO (In-Parameter-Order) is an algorithm for pairwise generation proposed by Tai and Lei [32]. The algorithm is straightforward in principle but tricky to implement. We have implemented the IPO algorithm in Java and experimented with a few test scenarios. The test sets generated by our IPO implementation were close to but always larger than what AETG generated. One problem of IPO is that if the input domains are arranged in different

orders, the sizes of the generated test sets may be different. This problem is investigated in more detail in the next chapter.

Many other test generation methods have been proposed, including constraint-based [9], table-based [3], factor-covering [7], structural [33], and iterative [12]. An analysis of the relationship between test coverage and test reliability can be found in [21].

3.2 Network testing

PROTOS [17] is a test framework for testing implementations of communication protocols using black-box testing. The framework currently has five test-suites for SNMPv1, HTTP-reply, IDAPv3, WAP-WSP-request, and WAP-WMLC respectively. The basic idea of PROTOS is to use a BNF to describe packet types for each protocol and generate concrete test packets from the BNF [17, 13, 26]. The BNF grammar is generated manually, while the concrete test generation is automatic.

The idea of using BNF to generate test tuples is interesting. All iptables rules are specified by a set of key words and values and should be able to be represented using BNF. If the rules for all iptables extensions could be described by a BNF grammar, we could then create a language for iptables rules and write a compiler for the language. In this way, any iptables scripts could be compiled for syntax checking, and executed to generate test tuples for functional testing. This feature is not included in the current version of our iptables testing framework.

Ethertap is software that simulates Ethernet devices. With Ethertap, you can run network experiments that normally require multiple physical Ethernet cards. Rusty Russell, the creator of iptables, has used Ethertap to test iptables. Unfortunately, Ethertap is now an obsolete tool and has been removed from the Linux 2.5.x kernel series. The successor of Ethertap is the TUN/TAP driver [11]. The TUN/TAP drivers simulate point-to-point or Ethernet devices and have been officially included in Linux kernel 2.5.x or above. With the TAP driver, user space applications can write or read Ethernet frames to or from simulated network devices. Our testing framework could run on the SUT without using the driver

machine if we configure the TAP drivers properly.

Some research work has been done in the field of iptables testing. The source code of iptables is incorporated with a default test suite [28], but using this test suite requires extensive knowledge of iptables rules. Another test suite for testing iptables was developed by Prabhakar [14, 25], where test cases are configured at compile time. Adding new test cases or modifying the test configuration requires changes to the C source code.

Tools for network security analysis are introduced in [34], and a thorough analysis of vulnerabilities of firewalls can be found in [18].

3.3 Packet generation

Network testing normally involves packet generation. For testing iptables, we need to generate a variety of test tuples for each iptables extension, and create one or more packets for each test tuple.

Testing iptables extensions requires close control over layer 2, 3, and 4 packet headers. For example, to test the `MAC` extension, the source MAC address of the Ethernet header must be set; to test the `--fragment` parameter, the fragment flags in the IP header must be set; and to test the `TCP` extension, the flags in the TCP header must be set. Our first implementation of packet generation attempted to use the standard Java socket library, which provides stream based network communication and encapsulates low level socket complication. Unfortunately, Java does not support raw sockets and prevents the user from controlling header fields closely, which is our primary concern when generating packets for testing iptables. Our next attempt was the Jpcap socket library [10], which is a Java library supporting raw sockets. After investigating the source code of Jpcap for a while, we found that most of the Jpcap library is useless to our research and it is not easy to reuse the part that is really useful to us. The fact that Jpcap is built on top of the libpcap C raw socket library and uses JNI (Java Native Interface) to glue the C library and the Java application did give us the hint that we could build our own Java raw socket library based on a C raw socket library using JNI.

We decided to use the C raw socket library provided by Durga Prabhakar [25], which is similar to the libnet and libdnet library [29]. The raw socket library uses the facade design pattern to specialize the Linux raw socket interface for Ethernet interfaces [14]. The library also adds a timeout mechanism for receiving packets. A byte array is passed to each write call and is returned by each read call. The caller is responsible for parsing the protocol headers. We built a Java raw socket library based on this C raw socket library using JNI, which is the interface for Java to interact with programs written in other languages [20]. JNI serves as a powerful glue between Java and other native languages, but is sometimes tedious and error-prone to use. Fortunately, Sun Microsystems is thinking of adding raw socket support in Java, which may eventually remove the complexity of using JNI to access raw socket.

In this chapter, we have introduced related work done by other researchers, especially in the field of test generation. In the next chapter, we will focus on the pairwise generation strategy.

Chapter 4. An Improved Pairwise Generation Strategy

We have introduced a few test generation strategies in chapter 2. In this chapter, we focus specifically on the pairwise generation strategy. We investigate an important property of pariwise generation and propose an improvement to the IPO pairwise generation strategy.

4.1 Overview

According to the definition of pairwise generation given in section 2.4.3, when there are only two input domains D_0 and D_1, the only pairwise test set is $D_0 \times D_1$. When there are more than two input domains, there may exist multiple test sets satisfying the pairwise property. As an example, consider the following three input domains:

- D_0: {a, b, c}
- D_1: {1, 2, 3}
- D_2: {x, y}

Then the test space, known as $S_0 = D_0 \times D_1 \times D_2$, is a pairwise test set, and the three subsets of S_0 shown in Table 4.1 are also pairwise test sets. Of the four pairwise test sets, S_0 contains 18 test tuples, S_1 and S_2 both contain 9 test tuples, and S_3 has 12 test tuples. It is easy to see that every pairwise test set for the given three input domains must contain at least 9 test tuples since $D_0 \times D_1$ has 9 elements. We would like to generate test sets like S_1 or S_2 since they satisfy the pairwise property with the minimum number of test tuples and will thus be executed faster. But in practice, it is normally not easy to find the minimum pairwise test sets. In [32], Tai and Lei have proved that the problem of generating the minimum pairwise test sets is NP-complete. Neither the AETG strategy nor the IPO strategy introduced in chapter 2 is guaranteed to generate the minimum test sets. During our investigation of the IPO strategy, we found that it is possible to improve this strategy based on an important property of pairwise generation, which we introduce next.

S1	S2	S3
$\langle a, 1, x \rangle$	$\langle a, 1, y \rangle$	$\langle a, 1, x \rangle$
$\langle a, 2, y \rangle$	$\langle a, 2, x \rangle$	$\langle a, 2, x \rangle$
$\langle a, 3, x \rangle$	$\langle a, 3, y \rangle$	$\langle a, 3, x \rangle$
$\langle b, 1, y \rangle$	$\langle b, 1, x \rangle$	$\langle b, 1, y \rangle$
$\langle b, 2, x \rangle$	$\langle b, 2, y \rangle$	$\langle b, 2, y \rangle$
$\langle b, 3, y \rangle$	$\langle b, 3, x \rangle$	$\langle b, 3, x \rangle$
$\langle c, 1, x \rangle$	$\langle c, 1, y \rangle$	$\langle c, 1, x \rangle$
$\langle c, 2, y \rangle$	$\langle c, 2, x \rangle$	$\langle c, 1, y \rangle$
$\langle c, 3, x \rangle$	$\langle c, 3, y \rangle$	$\langle c, 2, x \rangle$
		$\langle c, 2, y \rangle$
		$\langle c, 3, x \rangle$
		$\langle c, 3, y \rangle$

Table 4.1: Three pairwise test sets

4.2 Order-Irrelevance property of pairwise generation

All test tuples in a test set are of the same size, which is called the *order* of the test set. For a test set S of order n, the *exchange(i,j)* operation on S for $i, j \in [1, n], i \neq j$ is defined as exchanging the i^{th} element and the j^{th} element of all test tuples in S. Two test sets S_1 and S_2 are *equivalent* if S_2 can be obtained by a number of exchanges on S_1. To make it clear, consider the following two test sets:

- A = {⟨a,x,1⟩,⟨b,y,2⟩}
- B = {⟨1,a,x⟩,⟨2,b,y⟩}

Then A and B are equivalent since A can be obtained by two exchanges on B: `exchange(1,2)` followed by `exchange(2,3)`. Pairwise generation has an important property shown in the following theorem.

Theorem 4.1:

Given n input domains $D_1, D_2, \ldots, D_n$. $P_1 = (A_1, A_2, \ldots, A_n)$ and $P_2 = (B_1, B_2, \ldots, B_n)$ are two permutations of $(D_1, D_2, \ldots, D_n)$. Let S_1 be the set of all pairwise test sets of P_1, and S_2 is the set of all pairwise test sets of P_2. Then there is a bijection $f : S_1 \longrightarrow S_2$,

such that $\forall(X \in S_1, Y \in S_2)$, $Y = f(X)$ if and only if X and Y are equivalent.

The property shown by Theorem 4.1 is called the *Order-Irrelevance Property* of pairwise generation. This property implies that pairwise generation algorithms depending on the order of input parameters may be optimized by reordering the input domains. This is true because applying the algorithm on the given parameter order may not generate a minimum pairwise test set. By reordering the input parameters, the test set generated may be minimized. By the Order-Irrelevance Property, the minimized test set generated is equivalent to a test set with the original parameter order. In the next section, we investigate one such algorithm and propose an improvement.

4.3 Improvement of the IPO strategy

Tai and Lei proposed an algorithm called In-Parameter-Order (IPO) for pairwise generation [31]. IPO is a specification-based test generation strategy [1, 22]. Given n input domains, the IPO algorithm constructs the test set in n-1 steps. The first step creates the Cartesian product of the first two input domains. For the i^{th} step where $1 < i < (n-1)$, the algorithm creates $(i+1)$-tuples from i-tuples created in the previous step. A detailed explanation of the IPO algorithm can be found in [32].

IPO uses a greedy algorithm for building test tuples in each step, so it is not guaranteed to generate the minimum pairwise test set. Another problem with IPO is that it builds test tuples using the input parameters in the order of they are given, which may not lead to the smallest test set achievable. Recalling the Order-Irrelevance Property of pairwise generation, it is clear there is potential for improving the IPO algorithm by reordering the input parameters. As an example, consider the following two lists of input parameters:

- $T_1 = (p_1, p_2, p_3, p_4)$
- $T_2 = (p_1, p_3, p_2, p_4)$

where the only difference between T_1 and T_2 is that the second parameter and the third

S_1	S_1'	S_2	S_2'
$\langle 10,20,30,40\rangle$	$\langle 10,30,20,40\rangle$	$\langle 10,30,20,40\rangle$	$\langle 10,20,30,40\rangle$
$\langle 10,21,31,41\rangle$	$\langle 10,31,21,41\rangle$	$\langle 10,31,21,41\rangle$	$\langle 10,21,31,41\rangle$
$\langle 11,20,32,42\rangle$	$\langle 11,32,20,42\rangle$	$\langle 10,32,20,42\rangle$	$\langle 10,20,32,42\rangle$
$\langle 11,21,30,40\rangle$	$\langle 11,30,21,40\rangle$	$\langle 11,30,21,42\rangle$	$\langle 11,21,30,42\rangle$
$\langle 10,21,32,42\rangle$	$\langle 10,32,21,42\rangle$	$\langle 11,31,20,40\rangle$	$\langle 11,20,31,40\rangle$
$\langle 11,20,31,41\rangle$	$\langle 11,31,20,41\rangle$	$\langle 11,32,21,40\rangle$	$\langle 11,21,32,40\rangle$
$\langle 10,20,30,41\rangle$	$\langle 10,30,20,41\rangle$	$\langle 11,30,20,41\rangle$	$\langle 11,20,30,41\rangle$
$\langle 10,20,30,42\rangle$	$\langle 10,30,20,42\rangle$	$\langle 10,31,20,42\rangle$	$\langle 10,20,31,42\rangle$
$\langle 10,20,31,40\rangle$	$\langle 10,31,20,40\rangle$	$\langle 10,32,20,41\rangle$	$\langle 10,20,32,41\rangle$
$\langle 10,20,31,42\rangle$	$\langle 10,31,20,42\rangle$		
$\langle 10,20,32,40\rangle$	$\langle 10,32,20,40\rangle$		
$\langle 10,20,32,41\rangle$	$\langle 10,32,20,41\rangle$		

Table 4.2: Pairwise test sets for T_1 and T_2

parameter are exchanged. The following are four input domains, where D_i is the input domain of p_i for $i = 1, 2, 3, 4$:

- $D_1 = \{10, 11\}$
- $D_2 = \{20, 21\}$
- $D_3 = \{30, 31, 32\}$
- $D_4 = \{40, 41, 42\}$

By invoking our implementation of the IPO algorithm, we generate a pairwise test set S_1 for T_1 and another pairwise test set S_2 for T_2, shown in Table 4.2. We can see the following facts from the table:

1. S_2 with 9 test tuples is smaller than S_1 with 12 test tuples
2. By doing an `exchange(2,3)` operation on S_1, we get a pairwise test set S_1' for T_2 with 12 test tuples
3. By doing an `exchange(2,3)` operation on S_2, we get a pairwise test set S_2' for T_1 with 9 test tuples

```
1) Set minSet = empty set
2) Permute the n input domains by size
3) For each permutation P
4)    Set S = test set generated by invoking IPO on P
5)    If minSet is empty or S is smaller than minSet
6)       Set minSet = S
7) End For
8) Order test tuples in minSet as original input parameters order
9) Return minSet
```

Figure 4.1: Pseudocode for the improved IPO algorithm

Note that reordering two input domains of the same size can never improve the test set generated by IPO. This is because the number of test tuples generated at each step is determined by the size instead of the content of the current input domain. To make this point clear, suppose we have a test template $T_3(p_2, p_1, p_4, p_3)$ with the same input domains defined above, then the test set generated by IPO for T_3 is guaranteed to be of the same size as the test set generated by IPO for T_1 since $|D_1| = |D_2|$ and $|D_3| = |D_4|$.

The analysis above leads us to an improvement of the IPO algorithm by first finding the best ordering of the input parameters and then using IPO on that ordering to generate the test set, which will be the smallest pairwise test set that can be achieved by IPO. Although this idea is exciting, finding the best ordering is hard. Our current solution is to invoke IPO on all orderings of the input parameters by size and keep the minimum test set generated. The pseudocode show in Figure 4.1 summarizes the improved IPO algorithm. The input of the algorithm are the n input domains of a test template with n input parameters, and the output of the algorithm is the minimum pairwise test set that can be achieved by IPO.

A few considerations should be pointed out for the improved IPO (IIPO) algorithm. First, the test tuples in *minSet* at step (7) may not be in the same order as the original order of input parameters, so step (8) reorders each test tuple to be in the original order of input parameters. Second, the time complexity of IIPO is determined by step (3), i.e., the number of permutations of the given n input domains by size. We do not need to consider all $n!$ permutations of the n input domains since we stated earlier that reordering two input

domains of the same size does not help improve the test set generated by IPO. Suppose the n input domains have i different sizes s_1, s_2, ..., s_i, and suppose k_j input domains are of size s_j, for $j \in [1, i]$ (it follows that $k_1 + k_2 + \ldots + k_i = n$), then the number of passes of the for loop in the improved IPO algorithm equals:

$$n!/(k_1! \times k_2! \times \ldots \times k_i!) \qquad \text{(Formula 4.1)}$$

The time complexity of the IPO algorithm is $O(n^2 \times m^5)$ [31], where m is the number of elements in the largest input domain. It follows that the time complexity of the IIPO algorithm is exponential in the worst case.

In the next section, we will introduce our implementation of the IIPO algorithm.

4.4 Implementation and test results

We have implemented both the IPO algorithm and the improved IPO (IIPO) algorithm. Appendix A lists the Java source code of our implementation of the IIPO algorithm. The algorithm is implemented as a Java iterator called `PWIterator`. The interface of this Java class is shown as follows:

```
public class PWIterator implements Iterator {
    public PWIterator(Vector v);
    public boolean hasNext();
    public Object next();
}
```

The whole generation process is completed in the class constructor. Upon instantiation, `PWIterator` generates a minimum pairwise test set that can be achieved by IPO. Each time the `next()` method is called, the next test tuple will be returned, until no more test tuples are available. The `hasNext` method checks if there are more test tuples left.

It is fairly easy to calculate the number of permutations that must be considered in the IIPO algorithm from Formula 4.1, but it is much difficult to enumerate these permutations exactly once each. There are a few linear algorithms solving the problem of permutation

with repetition. One of these algorithms is provided in the C++ Standard Template Library (STL), which populates each permutation in lexicographical ordering. The algorithm we have implemented is proposed by Ruskey [27], which is similar to the algorithm provided in STL but simpler. We note that, although the permutation algorithm is linear, it does not improve the asymptotic complexity of the IIPO algorithm. IIPO is not a practical algorithm when there are a large number of input parameters. In our testing framework, the number of input parameters never exceeds 5, so it is practical to use IIPO in our research.

A program has been created to test the correctness of our IIPO implementation. The test program is implemented in Java and includes four source files, which are given in Appendix B. The test procedure contains two steps: `testPosition` and `testCoverage`. `testPosition` ensures that the i^{th} element of each test tuple is from the i^{th} input domain. `testCoverage` checks that all pairs that should be covered are indeed covered by the test set generated. Figure 4.2 lists the source code for `testCoverage`. We can see that `testCoverage` works by closely following the definition of pairwise generation. For each pair p that needs to be covered, it searches the generated test set for a test tuple that covers p. If a covering test tuple could not be found for p at any step, the test program breaks and returns false, which means the generated test set is not a pairwise test set.

We have set up the following eight test scenarios to test our implementations:

- S1 with 5 input domains
  - 2 domains with 2 values
  - 1 domain with 3 values
  - 2 domains with 4 values
- S2 with 4 input domains
  - 1 domain with 2 values
  - 2 domains with 3 values
  - 1 domain with 4 values
- S3 with 5 input domains
  - 1 domain with 2 values

```
private boolean testCoverage(Vector tuples) {
    boolean pass = true;
    for (int i=0; i≤domainVector.size()-1; i++) {
        for (int j=i+1; j≤domainVector.size(); j++) {
            Vector domain0 = (Vector)domainVector.elementAt(i);
            Vector domain1 = (Vector)domainVector.elementAt(j);
            Vector cpVector = new Vector();
            cpVector.addElement(domain0);
            cpVector.addElement(domain1);
            Vector pairSet = new Vector();
            Iterator cpIter = new CPIterator(cpVector);
            while (cpIter.hasNext()) {
                Vector v = (Vector)cpIter.next();
                pairSet.addElement(v);
            }
            for (int k=0; k≤tuples.size(); k++) {
                Vector tuple = (Vector)tuples.elementAt(k);
                for (int m=0; m≤tuple.size()-1; m++) {
                    for (int n=m+1; n≤tuple.size(); n++) {
                        Vector v = new Vector();
                        v.addElement(tuple.elementAt(m));
                        v.addElement(tuple.elementAt(n));
                        pairSet.remove(v);
                    }
                }
            }
            if (!pairSet.isEmpty()) {
                pass = false;
                System.out.println("FAIL: some pair is not covered");
                for (int m=0; m≤pairSet.size(); m++) {
                    System.out.println(pairSet.elementAt(m));
                }
            }
        }
    }
    return pass;
}
```

Figure 4.2: Code in the test program of the IIPO algorithm

  - 2 domains with 3 values
  - 2 domains with 4 values
- S4 with 7 input domains
  - 1 domain with 2 values
  - 2 domains with 3 values
  - 2 domains with 4 values
  - 2 domains with 5 values
- S5 with 6 input domains
  - 2 domains with 3 values
  - 2 domains with 5 values
  - 2 domains with 7 values
- S6 with 8 input domains
  - 4 domains with 3 values
  - 4 domains with 4 values
- S7 with 6 input domains
  - 1 domain with 7 values
  - 2 domains with 8 values
  - 2 domains with 9 values
  - 1 domain with 11 values
- S8 with 6 input domains
  - 1 domain with 3 values
  - 2 domains with 5 values
  - 1 domain with 6 values
  - 2 domains with 10 values

Table 4.3 shows the sizes of the test sets generated by IPO, AETG, and IIPO for the eight test scenarios given above. All the pairwise test sets generated by IPO and IIPO have been verified by the test program introduced earlier in this chapter. From this table, we see that IIPO generates pairwise test sets smaller than or as large as the test sets generated by IPO

Scenario	S1	S2	S3	S4	S5	S6	S7	S8
IPO	18	13	20	29	54	23	111	102
AETG	16	12	16	25	49	21	102	100
IIPO	16	12	16	25	49	20	106	100

Table 4.3: Comparison of IPO, AETG, and IIPO in eight test scenarios

Scenario	S1	S2	S3	S4	S5	S6	S7	S8
Input domains (N)	5	4	5	7	6	8	6	6
Permutations (N!)	120	24	120	5040	720	40320	720	720
Execution time (ms)	149	19	41	8590	315	74996	24838	9880
Time/permutation (ms)	1.24	0.75	0.34	1.71	0.44	1.86	34.5	13.7

Table 4.4: Execution time in eight test scenarios

in all test scenarios. Except for one test scenario, IIPO generates test sets no larger than AETG.

The disadvantage of using IIPO is that it is not efficient. When the number of input domains is large, the number of permutations to be considered may become too large to be executed efficiently. Table 4.4 shows the time used for each of the eight test scenarios running on a Pentium IV 1.8GHz machine. From the table, we can compute the average time used to invoke the IPO algorithm for each permutation, which ranges from 340 microseconds to 35 milliseconds in the given eight test scenarios and is about 10 milliseconds on average for our 1.8GHz test machine. Then, given the number of input domains, we can estimate the execution time of the IIPO algorithm in the worst case. Table 4.5 lists the estimated execution time for the number of input domains from 5 to 12. From this table, we can see that the performance of the IIPO algorithm will become unacceptable when the number of input domains grows large. Our iptables testing framework uses the IIPO algorithm because the number of input domains is always less than six.

N	5	6	7	8	9	10	11	12
Time	1.2sec	7.2sec	50sec	7min	1hours	10hours	110hours	55days

Table 4.5: Estimated execution time of IIPO for n from 5 to 12

In this chapter, we have introduced the Order-Irrelevance property of pairwise generation. Based on this property, we proposed an improvement of the IPO pairwise generation strategy. In our iptables testing framework, the improved IPO strategy has been integrated with the test templates described in the next chapter.

Chapter 5. Test Template Catalog

In this chapter, we describe twelve test templates designed for testing iptables.

5.1 Overview

The use of test template in our iptables testing framework originates from the Roast framework [8]. Test templates are useful for regulating the test process of the System Under Test (SUT). A test template should be carefully designed to cover a reasonable subset of the functionalities of the SUT. If the test templates are too finely grained, it may result in a large number of test templates, too many to be managed effectively. On the other hand, if the granularity is too large, each test template may cover too many functionalities of the SUT and is thus hard to evolve and reuse.

Since iptables is organized as a set of extensions, we have chosen to design the test templates for iptables by extensions. The general rule is to create one test template for each iptables extension. For each iptables parameter, a test template is also created, except that the `--source` and the `--destination` parameters are integrated in the same test template. In this chapter, we organize test templates as a catalog, where each test template is described with the structure shown in Figure 5.1.

Iptables rule summary
- **Syntax**: the syntax of related iptables rules
- **Semantics**: the meaning of related iptables rules

Test plan
- **Goal**: the design objective of this test template
- **Template**: the prototype of this test template
- **Strategy**: the test generation strategy used
- **Description**: the execution procedure

Figure 5.1: Categories of a test template

For each test tuple executed, the test result may be: (1) success if the expected iptables action is accept and the packet is received, or if the expected iptables action is reject and the packet is not received; or (2) failure if the expected iptables action is accept and the packet is not received, or if the expected iptables action is reject and the packet is received.

The following constant sets will be used throughout this chapter:

- *protocols* = {*tcp,udp,icmp*} is the set of valid protocols
- *directions* = {*inbound,outbound*} is the set of valid transmission directions
- *results* = {*accept,reject*} is the set of possible iptables filtering results

Twelve test templates have been created for testing iptables, as listed in Table 5.1. In order to explain how a test template is organized, we now take the `ProtocolTest` template in section 5.2 as an example and describe it in detail.

We start by summarizing the iptables rule syntax associated with the given test template. The `ProtocolTest` template is designed to test the iptables `--protocol` *proto* parameter, where *proto* is a user specified protocol. A sample iptables rule using this parameter is shown as follows:

```
iptables -A FORWARD --protocol udp -j ACCEPT
```

Most iptables parameters or extension options have abbreviations, which we have omitted in the catalog. As an example, the `--protocol` parameter can be abbreviated as `-p`. Hence, the following iptables rule has the same meaning as the one above:

```
iptables -A FORWARD -p udp -j ACCEPT
```

Following the rule syntax, we give the semantics of the rule. For the `ProtocolTest` template, the related iptables rule described above will accept all packets of the specified protocol. Notice that this rule does not imply the rejection of packets of any other protocol,

Protocol	IP address
In interface	Out interface
Fragment	TCP
UDP	ICMP
MAC	Limit
TOS	Multiport

Table 5.1: Iptables test templates

which depends on the default policy and whether or not there are other iptables rules. When describing test procedure of a test template, we assume that the default iptables policy is always drop and there are no other iptables rules except the ones associated with the given test template. Based on this assumption, the iptables rule given above will accept all packets of protocol UDP and reject all packets of any other protocol. The specified protocol can be any value in the *protocols* set, or the special value *all*. If *all* is specified, this rule will accept packets of all protocols. The `!` option is also allowed and when specified, the effect of the rule will be inverted. For example, the following iptables rule will *reject* all packets of protocol UDP and *accept* all packets of any other protocol. For simplicity, we have omitted the discussion of the `!` option in the test template catalog.

```
iptables -A FORWARD --protocol ! udp -j ACCEPT
```

After having explained the related iptables rules, we describe the test plan of the `ProtocolTest` template. We first give the goal of the test. Whenever possible, we want to test all protocols. It is also expected to test both inbound and outbound packets, and the test procedure should demonstrate both the acceptance scenarios and the rejection scenarios. Second, the test template is given as follows:

`ProtocolTest`(*rule-protocol,packet-protocol*)

There are two input parameters for this test template. The first parameter, *rule-protocol*, is a set of protocols specified by the user and contains possible values for *proto* in the iptables rule. It is required that $\textit{rule-protocol} \subseteq (\textit{protocols} \cup \{\textit{all}\})$. The second parameter, *packet-protocol*, is the set of all valid protocols and it follows that *packet-protocol=protocols*. As

we mentioned earlier, one of the goals of this test template is to test both inbound and outbound packets, so another input parameter should have been added to specify the available packet directions. However, since the direction parameter is almost always considered in all test templates, we have treated it as a default parameter and ignored it when describing a test template. You should be able to see the effect of the direction parameter in the description of the test procedure.

The strategy category is very important for a test template. As we have discussed in chapter 2, different test sets will be generated by applying different test generation strategies. In our current implementation of the iptables testing framework, each test template is associated with exactly one test generation strategy. For example, Cartesian product generation has been associated with the `ProtocolTest` template. If the cardinality of the *rule-protocol* set is n, then by the rule of product, the number of test tuples generated for the `ProtocolTest` template will be:

$$|\textit{rule-protocol}| \times |\textit{packet-protocol}| \times |\textit{directions}| = n \times 3 \times 2 = 6n$$

The description part for a test template describes the test procedure for using the test template. For the `ProtocolTest` template, the idea is to test for each combination of user specified protocol, valid protocol, and valid packet direction.

In the following sections, we will list the twelve test templates designed for iptables one by one.

5.2 Protocol test template

5.2.1 Iptables rule summary

Syntax: `--protocol [!]` *proto*

Semantics: This iptables rule will match all packets of the specified protocol *proto*, which can be any value in *protocols*. If *proto* = *all* the rule will match packets of all protocols.

5.2.2 Test plan

Goal: Test for each $p \in$ *protocols*, inbound and outbound, accept and reject.

Template: `ProtocolTest(rule-protocol,packet-protocol)`

rule-protocol is the set of user specified protocols

packet-protocol is the set of all valid protocols

Strategy: Cartesian product generation. If $|$*rule-protocol*$| = n$, then the number of test tuples generated will be $n \times 3 \times 2 = 6n$.

Description:

for each protocol *rp* $\in$ *rule-protocol*
  set iptables rule to accept packets of protocol *rp*
    for each protocol *pp* $\in$ *packet-protocol*
      for each direction *d* $\in$ *directions*
        send a packet of protocol *pp* in direction *d*
        if *rp* = *pp* or *rp* = *all*
          expected iptables action is accept
        else
          expected iptables action is reject

5.3 IP address test template

5.3.1 Iptables rule summary

Syntax:

`--source [!]` *sip*

`--destination [!]` *dip*

Semantics: This iptables rule will match all IP packets with the specified source or destination addresses. *sip* and *dip* can be any valid IP addresses. If the ! argument is specified, the effect is inverted.

5.3.2 Test plan

Goal: Test for all 1-boundary values implicitly determined by the specified IP address, inbound and outbound, accept and reject.

Template: `IPAddressTest(sip0,sip1,sip2,sip3,dip0,dip1,dip2,dip3)`

sip_0 is the first byte of the user specified source IP address

sip_1 is the second byte of the user specified source IP address

sip_2 is the third byte of the user specified source IP address

sip_3 is the fourth byte of the user specified source IP address

dip_0 is the first byte of the user specified destination IP address

dip_1 is the second byte of the user specified destination IP address

dip_2 is the third byte of the user specified destination IP address

dip_3 is the fourth byte of the user specified destination IP address

Strategy: 1-boundary values generation. For each specified byte value v, we create a set of byte values containing *v*, *(v-1) mod 255*, and *(v+1) mod 255*. These sets are used to construct the source and destination IP addresses of the actual packets. The number of test tuples generated will be $3^4 \times 3^4 \times 2 = 13122$.

Description:

let *sip* be the source IP address $sip_0.sip_1.sip_2.sip_3$

let *dip* be the destination IP address $dip_0.dip_1.dip_2.dip_3$

let S_i be the set $\{(sip_i - 1)\%255, (sip_i + 1)\%255\}$ for $i = 0, 1, 2, 3$

let D_i be the set $\{(dip_i - 1)\%255, (dip_i + 1)\%255\}$ for $i = 0, 1, 2, 3$

set iptables rule to accept IP packets with source IP *sip* and destination IP *dip*

for each source IP $si = si_0.si_1.si_2.si_3$, where $si_i \in (S_i \cup \{sip_i\})$ for i=0,1,2,3

for each destination IP $di = di_0.di_1.di_2.di_3$, where $di_i \in (D_i \cup \{dip_i\})$ for i=0,1,2,3

for each direction $d \in directions$

send an IP packet with source IP *si* and destination IP *di* in direction *d*

if $si = sip$ and $di = dip$

expected iptables action is accept

else

expected iptables action is reject

5.4 In interface test template

5.4.1 Iptables rule summary

Syntax: `--in-interface [!] i`

Semantics: This iptables rule will match all IP packets received at the specified interface *i*. *i* can be the name of any valid interface on the SUT. If the `!` argument is specified, the effect is inverted.

5.4.2 Test plan

Goal: Test for all protocols, inbound and outbound, accept and reject.

Template: `InInterfaceTest(i)`

i is the user specified input interface

Strategy: Cartesian product generation. The number of test tuples generated will be 6.

Description:

set iptables rule to accept packets received at interface *i*

for each protocol $p \in protocols$

for each direction $d \in$ *directions*

send a packet of protocol p in direction d

if $d = i$

expected iptables action is accept

else

expected iptables action is reject

5.5 Out interface test template

5.5.1 Iptables rule summary

Syntax: `--out-interface [!]` *o*

Semantics: This iptables rule will match all IP packets to be sent to the specified interface *o*. *o* can be the name of any valid interface on the SUT. If the `!` argument is specified, the effect is inverted.

5.5.2 Test plan

Goal: Test for all protocols, inbound and outbound, accept and reject.

Template: *OutInterfaceTest(o)*

o is the user specified output interface

Strategy: Cartesian product generation. The number of test tuples generated will be 6.

Description:

set iptables rule to accept packets to be sent to interface *o*

for each protocol $p \in$ *protocols*

for each direction $d \in$ *directions*

send a packet of protocol $p \in$ direction d

if $d \neq o$

expected iptables action is accept

else

expected iptables action is reject

5.6 Fragment test template

5.6.1 Iptables rule summary

Syntax: `[!] --fragment`

Semantics: This iptables rule will match the second and further fragments of fragmented packets. If the `!` argument is specified, the effect is inverted.

5.6.2 Test plan

Goal: Test for all protocols, inbound and outbound, accept and reject.

Template: `FragmentTest()`

Strategy: Cartesian product generation. The number of test tuples generated will be 6.

Description:

set iptables rule to accept fragmented packets

for each protocol $p \in$ *protocols*

for each direction $d \in$ *directions*

send a fragmented packet of protocol $p \in$ direction d

expected iptables action is accept

send a not-fragmented packet of protocol $p \in$ direction d

expected iptables action is reject

5.7 TCP test template

5.7.1 Iptables rule summary

Syntax:

`--protocol tcp --source-port [!] [`p_1`[:` p_2`]]`

`--protocol tcp --destination-port [!] [`p_1`[:` p_2`]]`

Semantics: This iptables rule will match all TCP packets with the source and destination ports in the specified ranges. p_1 and p_2 can be any valid port numbers from 0 to 65535. 0 is assumed if p_1 is omitted and 65535 is assumed if p_2 is omitted.

5.7.2 Test plan

Goal: Test for all 1-boundary values of the specified port ranges, inbound and outbound, accept and reject.

Template: `TCPTest(`sp_1`,` sp_2`,` dp_1`,` dp_2`)`

sp_1 is the lower bound of the user specified source port range

sp_2 is the upper bound of the user specified source port range

dp_1 is the lower bound of the user specified destination port range

dp_2 is the upper bound of the user specified destination port range

Strategy: 1-boundary values generation. The number of test tuples generated will be no more than 72.

Description:

let S_0 be the set of ports from 0 to $(sp_1 - 1)$

let S_1 be the set $[sp_1, sp_2]$

let S_2 be the set of ports from $(sp_2 + 1)$ to 65535

let D_0 be the set of ports from 0 to $(dp_1 - 1)$

let D_1 be the set $[dp_1, dp_2]$

let D_2 be the set of ports from $(dp_2 + 1)$ to 65535

let SB_0 = 1-boundary(S_0)

let SB_1 = 1-boundary(S_1)

let SB_2 = 1-boundary(S_2)

let DB_0 = 1-boundary(D_0)

let DB_1 = 1-boundary(D_1)

let DB_2 = 1-boundary(D_2)

set iptables rule to accept TCP packets with source port in S_1 and

destination port in D_1

for each source port $sp \in (SB_0 \cup SB_1 \cup SB_2)$

for each destination port $dp \in (DB_0 \cup DB_1 \cup DB_2)$

for each direction $d \in$ *directions*

send a TCP packet with source port *sp* and destination port

dp in direction *d*

if $sp \in S_1$ and $dp \in D_1$

expected iptables action is accept

else

expected iptables action is reject

5.8 UDP test template

5.8.1 Iptables rule summary

Syntax:

`--protocol udp --source-port [!]` $[p_1[:p_2]]$

`--protocol udp --destination-port [!]` $[p_1[:p_2]]$

Semantics: This iptables rule will match all UDP packets with the source and destination ports in the specified ranges. p_1 and p_2 can be any valid port numbers from 0 to 65535. 0 is assumed if p_1 is omitted and 65535 is assumed if p_2 is omitted.

5.8.2 Test plan

Goal: Test for all 1-boundary values of the specified port ranges, inbound and outbound, accept and reject.

Template: `UDPTest(sp1, sp2, dp1, dp2)`

sp_1 is the lower bound of the user specified source port range

sp_2 is the upper bound of the user specified source port range

dp_1 is the lower bound of the user specified destination port range

dp_2 is the upper bound of the user specified destination port range

Strategy: 1-boundary values generation. The number of test tuples generated will be no more than 72.

Description:

let S_0 be the set of ports from 0 to $(sp_1 - 1)$

let S_1 be the set $[sp_1, sp_2]$

let S_2 be the set of ports from $(sp_2 + 1)$ to 65535

let D_0 be the set of ports from 0 to $(dp_1 - 1)$

let D_1 be the set $[dp_1, dp_2]$

let D_2 be the set of ports from $(dp_2 + 1)$ to 65535

let SB_0 = 1-boundary(S_0)

let SB_1 = 1-boundary(S_1)

let SB_2 = 1-boundary(S_2)

let DB_0 = 1-boundary(D_0)

let DB_1 = 1-boundary(D_1)

let DB_2 = 1-boundary(D_2)

set iptables rule to accept UDP packets with source port in S_1 and

destination port in D_1

for each source port $sp \in (SB_0 \cup SB_1 \cup SB_2)$

for each destination port $dp \in (DB_0 \cup DB_1 \cup DB_2)$

for each direction $d \in$ *directions*

send a UDP packet with source port *sp* and destination port *dp* in direction *d*

if $sp \in S_1$ and $dp \in D_1$

expected iptables action is accept

else

expected iptables action is reject

5.9 ICMP test template

5.9.1 Iptables rule summary

Syntax: `--protocol icmp --icmp-type [!]` *type*

Semantics: This iptables rule will match all ICMP packets of the specified type *t*. *t* can be any valid ICMP type. If the `!` argument is specified, the effect is inverted.

5.9.2 Test plan

Goal: Test for all ICMP types, inbound and outbound, accept and reject.

Template: `ICMPTest(Types)`

Types is the set of user selected ICMP types

Strategy: Cartesian product generation.

Description:

let *S* be the set of all valid ICMP types

for each ICMP type $t \in$ *Types*

set iptables rule to accept ICMP packets of type *t*

for each ICMP type $s \in S$

for each direction $d \in$ *directions*

send an ICMP packet of type s in direction d

if $s \in$ *Types*

expected iptables action is accept

else

expected iptables action is reject

5.10 MAC test template

5.10.1 Iptables rule summary

Syntax: `--mac-source [!]` *addr*

Semantics: This iptables rule will match all packets with the specified source MAC address *addr*. *addr* must be of the form *XX:XX:XX:XX:XX:XX*, where *XX* is a 2-digit hexadecimal number. If the `!` argument is specified, the effect is inverted.

5.10.2 Test plan

Goal: Test for all 1-boundary values implicitly determined by the specified MAC address, inbound and outbound, accept and reject.

Template: `MACTest(`mac_0`,`mac_1`,`mac_2`,`mac_3`,`mac_4`,`mac_5`)`

mac_0 is the first byte of the given source MAC address

mac_1 is the second byte of the given source MAC address

mac_2 is the third byte of the given source MAC address

mac_3 is the fourth byte of the given source MAC address

mac_4 is the fifth byte of the given source MAC address

mac_5 is the sixth byte of the given source MAC address

Strategy: 1-boundary values generation. The number of test tuples generated will be no more than 128.

Description:

let *mac* be the MAC address mac_0:mac_1:mac_2:mac_3:mac_4:mac_5

let S_i be the set $\{mac_i\text{-}1, mac_i\text{+}1\}$ for i$\in$ [0,5]

set iptables rule to accept packets with source MAC address *mac*

for each MAC address m=m_0:m_1:m_2:m_3:m_4:m_5, $m_i \in (S_i \cup \{ mac_i\})$ for $i \in [0,5]$

for each direction $d \in$ *directions*

send an IP packet with source MAC m in direction d

if $m = mac$

expected iptables action is accept

else

expected iptables action is reject

5.11 Limit test template

5.11.1 Iptables rule summary

Syntax:

`--limit` *r*

`--limit-burst` *b*

Semantics: This iptables rule will match packets of a given protocol at the specified rate *r*. *r* is given as a number with an optional unit suffix /second, /minute, /hour, or /day. *b* is the initial maximum number of packets to match, with default value 5.

5.11.2 Test plan

Goal: Test for each $p \in$ *protocols*, inbound and outbound, accept and reject.

Template: `LimitTest(r,b)`

r is the specified maximum average matching rate

b is the specified initial maximum number of packets to match

Strategy: Cartesian product generation. The number of test tuples generated depends on r and b.

Description: Since the testing process highly depends on r and b, it is difficult to describe the process for arbitrary r and b. The following description assumes r = 1/second and b = 3.

for each protocol $p \in$ *protocols*
  set iptables limit rate for protocol p to be 1/second
  set iptables burst rate for protocol p to be 3
    for each direction $d \in$ *directions*
      send 4 packets of protocol p in direction d
        expected iptables action for the first 3 packets is accept
        expected iptables action for the last packet is reject
      wait for 3 seconds
      send 3 packets of protocol p in direction d
        expected iptables action for the 3 packets is accept
      wait for 1 second
      send a packet of protocol p in direction d
        expected iptables action is accept

5.12 TOS test template

5.12.1 Iptables rule summary

Syntax: `--tos` *t*

Semantics: This iptables rule will match all IP packets of the specified Type Of Service (TOS) field *t*. *t* can be any valid IP TOS value.

5.12.2 Test plan

Goal: Test for all $p \in$ *protocols*, inbound and outbound, accept and reject.

Template: `TOSTest(Types)`

Types is the set of user selected TOS types

Strategy: Cartesian product generation.

Description:

let S be the set of all valid TOS types
for each TOS type $t \in$ *Types*
  set iptables rule to accept IP packets of TOS type t
for each TOS type $s \in S$
  for each protocol $p \in$ *protocols*
    for each direction $d \in$ *directions*
      send a packet of protocol p and with TOS field s in direction d
      if $s \in$ *Types*
        expected iptables action is accept
      else
        expected iptables action is reject

5.13 Multiport test template

5.13.1 Iptables rule summary

Syntax:

`--source-port` $[sp_1[, sp_2[, \ldots[, sp_i]\ldots]$

`--destination-port` $[dp_1[, dp_2[, \ldots[, dp_j]\ldots]$

`--port` $[p_1[, p_2[, \ldots[, p_k]\ldots]$

Semantics: This iptables rule can only be used together with `--protocol` *tcp* or `--protocol` *udp*. This rule will match all packets of the specified protocol with the source port being one of the specified source ports *sp* or one of the specified common ports *p*. The rule will also match all packets of the specified protocol with the destination port being one of the specified destination ports *dp* or one of the specified common ports *p*. Up to 15 ports may be specified for any of the three options, i.e., $i, j, k \leq 15$.

5.13.2 Test plan

Goal: Test for both TCP and UDP protocols, inbound and outbound, accept and reject, achieving pairwise coverage.

Template: `MultiportTest(P,SP,DP,CP)`

P is the set of user specified protocols

SP is the set of user specified source ports

DP is the set of user specified destination ports

CP is the set of user specified common ports

Strategy: pairwise generation

Description:

for each protocol $p \in P$

set iptables rule to accept packets of protocol *p* with source port in *SP*

set iptables rule to accept packets of protocol *p* with destination port in *DP*

set iptables rule to accept packets of protocol *p* with either source or destination port in *CP*

let the input domain for protocol be *protocols*

let the input domain for source port be $(SP \cup DP \cup CP)$

let the input domain for destination port be $(SP \cup DP \cup CP)$

let *S* be the test set returned by pairwise generation on (protocol, source port,

destination port)

for each test tuple $\langle pr, s, d \rangle \in S$

for each direction $d \in directions$

send a packet of protocol pr with source port s and destination port d in direction d

if $pr = p$

if $s \in (SP \cup CP)$ or $d \in (DP \cup CP)$

expected iptables action is accept

else

expected iptables action is reject

else

expected iptables action is reject

In this chapter, we have described the twelve test templates designed for testing iptables. All of these test templates have been implemented in the iptables testing framework, which we will introduce in the next chapter.

Chapter 6. GUI Implementation and Features

In previous chapters, we have introduced the test generation strategies and test templates used in the iptables testing framework, which has been implemented in a test tool called PBit (Pattern Based iptables tester). In this chapter, we describe GUI implementations and features of PBit.

6.1 Overview

PBit is a GUI test tool implemented in Java. Figure 6.1 shows the main window of PBit. The structure of the main window is described in Figure 6.2.

As shown in these figures, test templates are accessed through the **Template** menu. When a test template menu item is selected, the corresponding test dialogue box will be opened, allowing the user to specify input parameters and generate test tuples. Test tuples are then passed to the underlying test template to generate actual test cases. Each time a test tuple is executed, its test result will be automatically added to the **Test Result Area**. When a test case is selected in the **Test Result Area**, detailed information about the generated packet will be presented in the **Packet Details Area**. Each entry in the **Test Result Area** is organized with the following six fields:

Id: an integer index of the test case

Protocol: the protocol of the packet associated with the test case

Sent on: the destination network interface of the SUT

Expected Result: the expected test result

Actual Result: the actual test result

Successful: whether the test is successful or not

The expected result and the actual result are iptables actions and can be either accept or

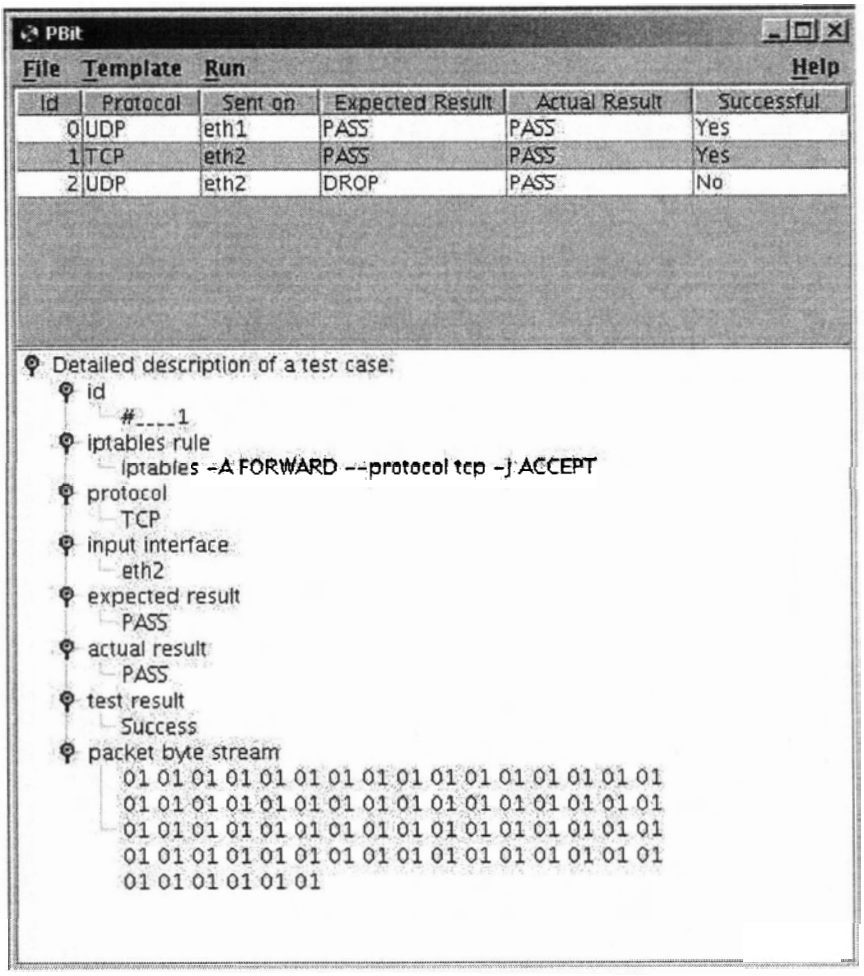

Figure 6.1: The main window of PBit

reject. The test is successful if the expected result and the actual result are the same; otherwise the test is a failure.

The **Help** menu item is used to access the help window, which is shown in Figure 6.3. All other menu items have straightforward meanings except for the **Config** menu item. By selecting the **Config** menu item, the Test Configuration dialogue box will be opened. We will explain this GUI feature in the following section.

Menu Bar
- **File**
  - **Open**: open a test set from a file
  - **Save**: save a test set to a file
  - **Config**: change the test configuration
  - **Exit**: exit PBit
- **Template**
  - **Base Tests**
    - **Protocol Test**: invoke the `ProtocolTest` template
    - **IP Address Test**: invoke the `IPAddressTest` template
    - **In Interface Test**: invoke the `InInterfaceTest` template
    - **Out Interface Test**: invoke the `OutInterfaceTest` template
    - **Fragment Test**: invoke the `FragmentTest` template
  - **TCP Test**: invoke the `TCPTest` template
  - **UDP Test**: invoke the `UDPTest` template
  - **ICMP Test**: invoke the `ICMPTest` template
  - **MAC Test**: invoke the `MACTest` template
  - **Limit Test**: invoke the `LimitTest` template
  - **TOS Test**: invoke the `TOSTest` template
  - **Multiport Test**: invoke the `MultiportTest` template
- **Run**
  - **Run**: run the current test tuples
- **Help**
  - **Help**: open the help manual

Work Space
- **Test Result Area**: list all the test tuples generated
- **Packet Details Area**: show the key fields of the generated packet

Figure 6.2: Structure of the PBit main window

6.2 Test configuration GUI

One of the most significant features of PBit is that the user is able to modify the test configuration at run time. As illustrated in Figure 2.2, both the SUT and the driver machine need three network interfaces to run the tests. As a result, PBit needs to configure the following parameters before running any test:

- **SUT**
  - IP addresses of eth0, eth1, eth2
  - MAC addresses of eth1, eth2

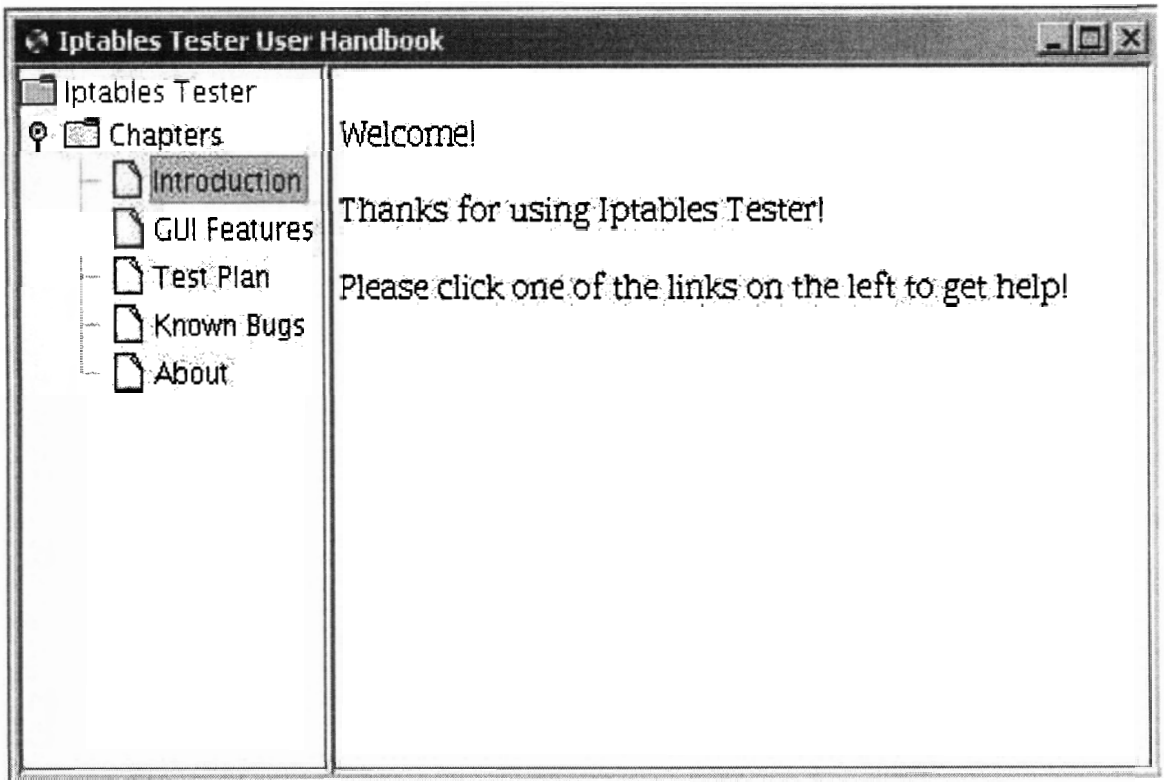

Figure 6.3: The help window of PBit

- **driver machine**
  - IP addresses of eth1, eth2
  - MAC addresses of eth1, eth2

The IP addresses shown in Figure 2.2 are examples of eth1 and eth2 IP values. It is likely that whenever the system hardware environment changes, the values of these parameters will have to be modified and the whole application has to be re-compiled. To make the life of the tester easier, PBit provides a GUI to easily configure these parameters at run time. A screen shot of the test configuration GUI in PBit is shown in Figure 6.4. A MAC address is input as six single bytes in hexadecimal format, and an IP address is input as four single bytes in decimal format, with the range of each byte from 0 to 255. Note that the configuration GUI is used to set test parameters in PBit at run time. The GUI can not be used to set the actual network addresses, which should be configured using some

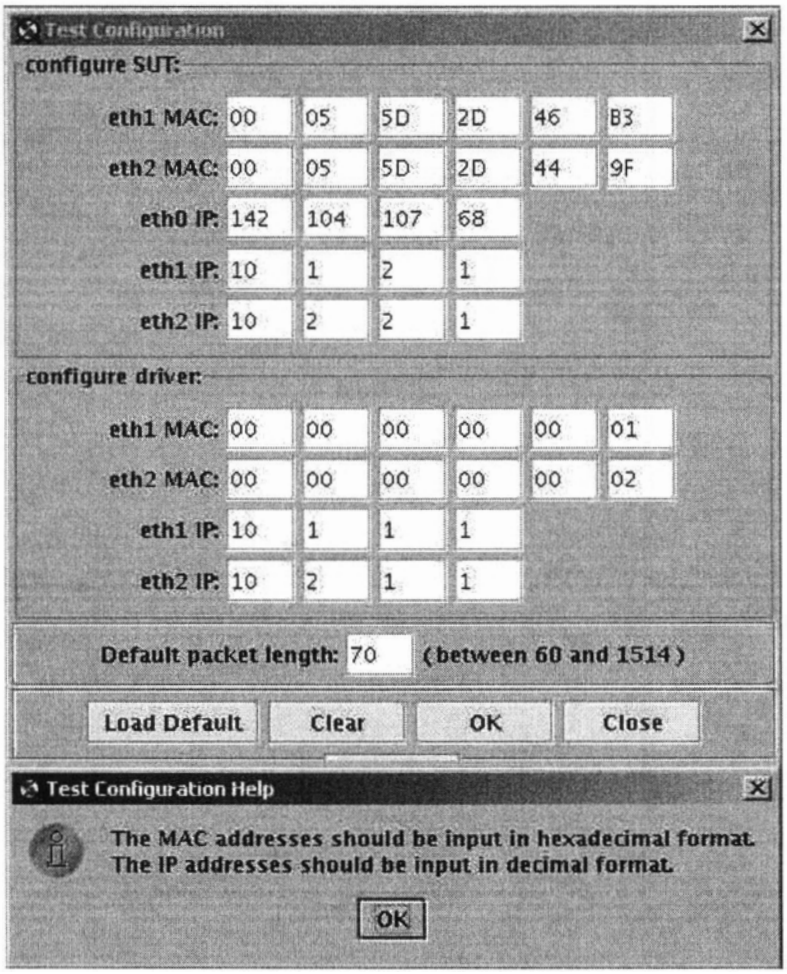

Figure 6.4: The test configuration GUI dialog in PBit

administration tools in the operating system (e.g., *ifconfig* in Linux). The user can also change the length of actual packets generated in the tests. The default packet length is 70 bytes and may range from 60 bytes to 1514 bytes. Once clicked, the **Load Default** button will load default values for all parameters. The **Help** button is used to open a message box showing instructions for using this dialogue box. In the following sections, we will see that **Help** buttons are also available in all test template dialogue boxes.

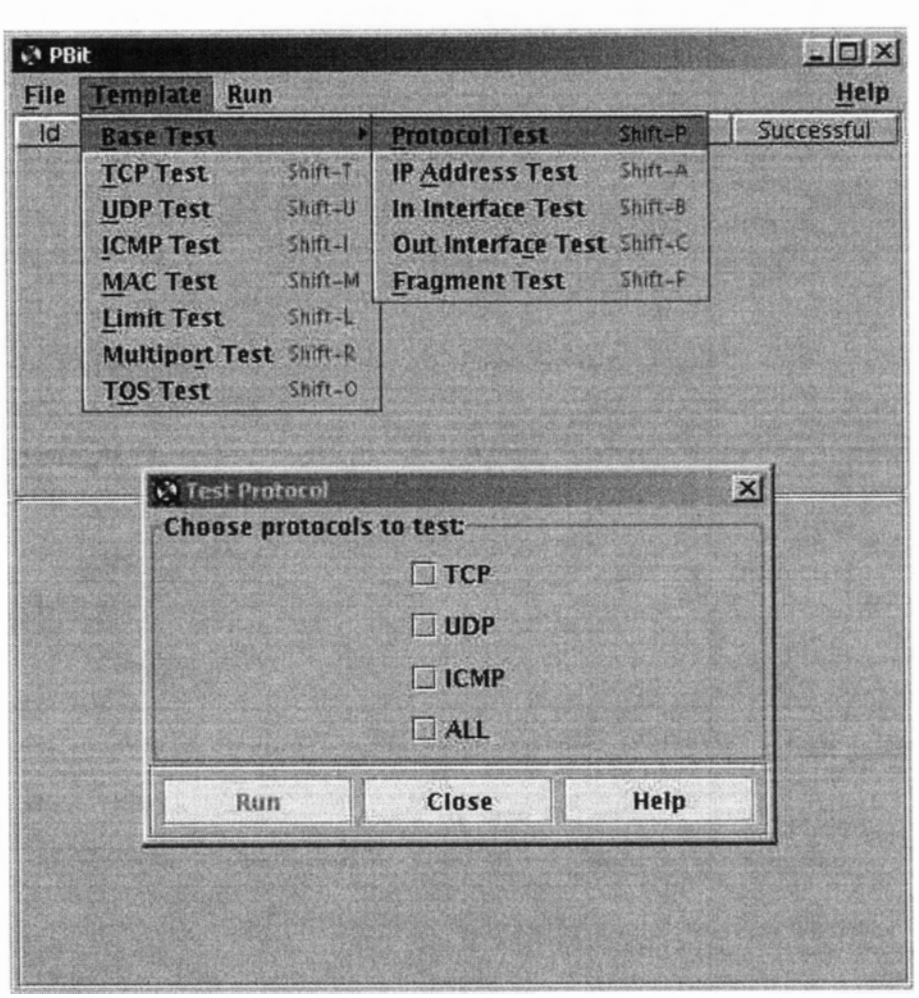

Figure 6.5: The ProtocolTest GUI dialog in PBit

6.3 Test template GUI

Chapter 5 introduced twelve test templates for testing iptables, as listed in Table 5.1. PBit implements all of these test templates and provides GUI interfaces to them. In this section, we explain the GUI interfaces of three test templates: `ProtocolTest`, `UDPTest`, and `MultiportTest`.

6.3.1 ProtocolTest GUI

Figure 6.5 shows the GUI dialogue box for the `ProtocolTest` template. This GUI dialogue box allows the user to select a set of protocols to test the iptables `--protocol`

parameter. The selected protocol set will become the input domain of the first input parameter (*rule-protocol*) in the `ProtocolTest` template described in section 5.2. The available protocols are TCP, UDP, ICMP, and the special value ALL. Note that the user can not control the second input parameter *packet-protocol*, which is always associated with the input domain {TCP, UDP, ICMP}. The Cartesian product generation strategy is used in this test template. When the **Run** button is clicked with at least one protocol selected, the GUI will invoke the underlying implementation of the `ProtocolTest` template by passing the selected set of protocols. The underlying test template will then generate test tuples following the description given in section 5.2. For each test tuple, a packet will be created and sent to the corresponding network interface. Once a packet is sent out, a timer is started in order to collect the test result within a certain amount of time (3 seconds by default). After the test results of all the test cases are collected, a message box will appear to summarize the successful and failed test cases. The results will also be added to the **Test Result Area** of the main window. This GUI interface is easy to understand because the `ProtocolTest` template is simple. We will see a more complicated example in the next section.

6.3.2 UDPTest GUI

Figure 6.6 shows the GUI dialogue box for the `UDPTest` template. This GUI dialogue box allows the user to test the iptables UDP extension with the specified source port range and destination port range. The range of source port is given as two port numbers: the lower bound and the upper bound. Each port number should be input as an integer in the range [0,65535], and the upper bound should be greater than or equal to the lower bound. If the lower bound is not specified, the default value 0 will be used. If the upper bound is not specified, the default value 65535 will be used. When the **Run** button is clicked, the GUI will invoke the underlying implementation of the `UDPTest` template by passing the four integers specified by the user. The test template will then generate test tuples

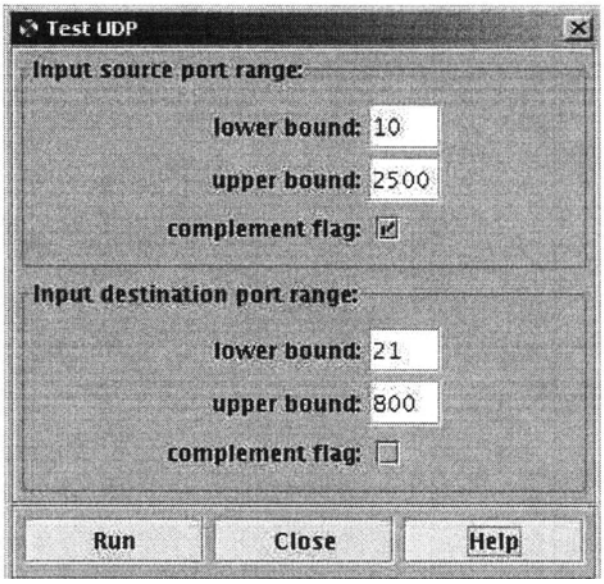

Figure 6.6: The UDPTest GUI dialog in PBit

following the description given in section 5.8. The 1-boundary values generation strategy is used in this test template. Notice that although there are four input parameters in the `UDPTest` template, only two input domains will be created. The first input domain is the set of integer values determined by the actual arguments of the first and the second input parameters. The second input domain is the set of integer values determined by the actual arguments of the third and the fourth input parameters. The procedure of sending actual packets and collecting test results is the same as described in the last section.

We have described two GUI dialogue boxes for test templates: one using Cartesian product generation and the other using boundary values generation. In the next section, we will describe the GUI dialogue box for the `MultiportTest` template, which uses pairwise generation.

6.3.3 MultiportTest GUI

The `MultiportTest` template is the only test template using pairwise generation in the current version of PBit. Figure 6.7 shows the GUI dialogue box of the `MultiportTest`

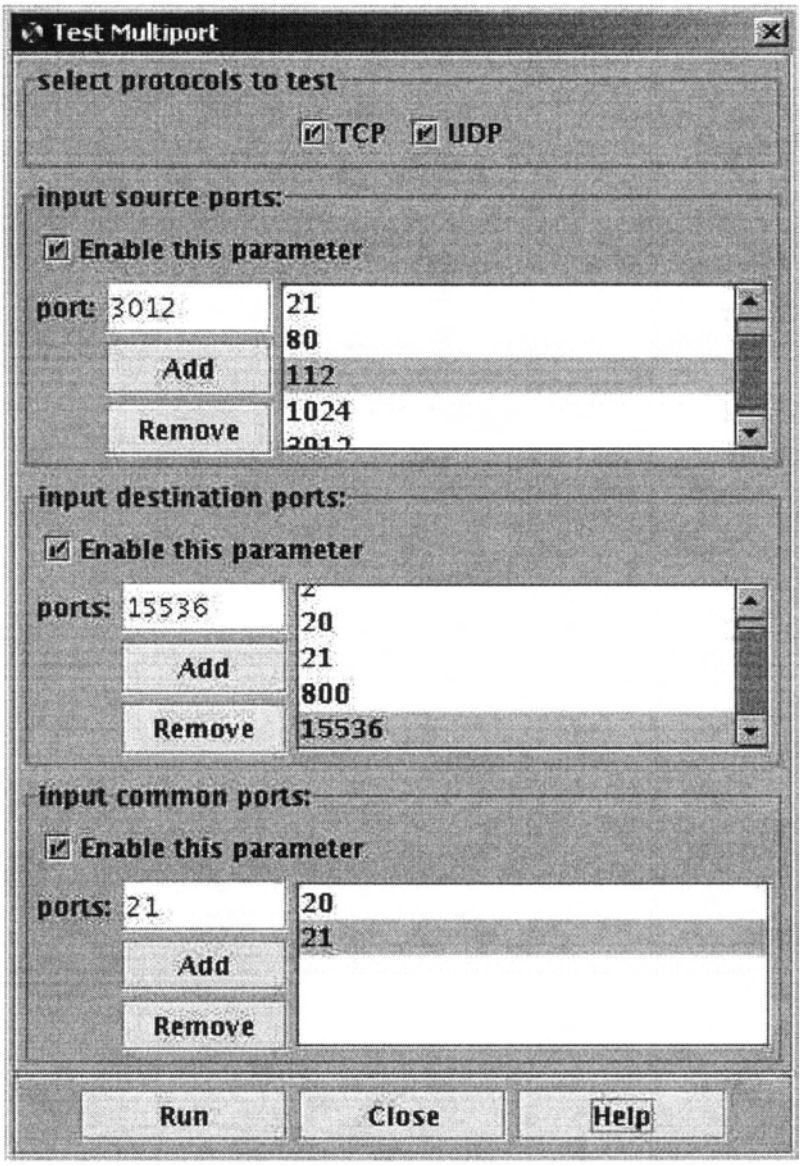

Figure 6.7: The MultiportTest GUI dialog in PBit

template in PBit. This GUI dialogue box allows the user to specify four parameters: the set of protocols, the set of source ports, the set of destination ports, and the set of common ports. The specified protocol set will become the input domain of the first input parameter (*P*) in the `MultiportTest` template described in section 5.13. The available protocols are TCP and UDP. In order to add values to a port set, the user must first enable the corresponding parameter. Each port number should be input as an integer in the range [0,65535].

The **Add** button is used to add a port number to the port set and the **Remove** button is used to remove a port number from the port set. The set of source ports specified by the user will become the input domain of the second input parameter (*SP*) in the test template, and the set of specified destination ports will become the input domain of the third input parameter. The input domain of the last input parameter comes from the set of common ports specified by the user.

Pairwise generation is used in this test template and the idea is to create TCP or UDP packets to cover all pairwise combinations of source ports and destination ports. The procedure of sending actual packets and collecting test results remains the same as the `ProtocolTest` template and the `UDPTest` template.

We have explained three GUI dialogue boxes, each using a different test generation strategy. In the next section, we will explain the design of PBit and how to extend PBit to accommodate new test templates.

6.4 PBit design and extension point

The objective of the PBit testing framework is not just to test the current iptables extensions. Since new iptables extensions are created frequently, a test tool with this objective is likely to become outdated soon. PBit aims at building an iptables testing framework that is easy to extend for testing new iptables extensions. In order to fulfill this purpose, our research has been focused on the following three topics:

- identifying test template patterns
- integrating with a test generation strategy
- building GUI dialogue boxes for test templates

Given a new iptables extension, the first topic intends to identify input parameters of the test template to be created. The designer needs to understand the related iptables rules in order to produce a good test template. The designer should also analyze what kinds of input domains can be associated with each input parameter. Based on the analysis, the designer must decide which input parameters should accept user input while other input parameters should be associated with default input domains preset by the system. For example, the `ProtocolTest` template has been designed to accept the input domain of the first input parameter from the user and fix the input domain of the second input parameter to be the constant set {TCP,UDP,ICMP}. To regulate the creation of new test templates, an abstract class has been created to act as the super class of all concrete test templates. The following code shows the interface of this class:

```
public abstract class TestTemplate {
    public static void config(
            int[] sutMAC1, int[] sutMAC2,
            int[] sutIP0, int[] sutIP1, int[] sutIP2,
            int[] driverMAC1, int[] driverMAC2,
            int[] driverIP1, int[] driverIP2,
            int len
        ) throws IptablesException;
    public static void init() throws IptablesTestException;
}
```

The `config` method is used to configure system parameters on SUT and the driver machine. The `init` method is used to initialize some system variables and network parameters of SUT.

The second topic deals with test generation strategies. PBit has reused the Cartesian product generation strategy and the boundary values generation strategy implemented in the Roast framework [8]. The IIPO algorithm proposed in chapter 4 is also implemented in

Template	Strategy
Protocol	Cartesian product
IP address	1-boundary values
In interface	Cartesian product
Out interface	Cartesian product
Fragment	Cartesian product
TCP	1-boundary values
UDP	1-boundary values
ICMP	Cartesian product
MAC	1-boundary values
Limit	Cartesian product
TOS	Cartesian product
Multiport	pairwise

Table 6.1: Test generation strategies used in PBit

PBit. In principle, multiple test generations strategies may be used for each test template. For example, Cartesian product generation could be permitted in all test templates, and boundary values generation could be used whenever the input domains are ordered. The pairwise generation strategy could also be used in all test templates, but it is only meaningful when the number of input parameters of a test template is greater than two. Since most of the test templates in PBit have two or less parameters, the pairwise generation strategy has not been used widely. For simplicity, the current implementation of PBit fixes the test generation strategy associated with each test template, as listed in Table 6.1. A few factors will affect the selection of the test generation strategy, such as the number of input parameters and the number of elements in each input domains. It is expected that the next version of PBit will integrate multiple test generation strategies with each test template.

The third topic is to build GUI dialogue boxes for test templates. Once a test template has been created and integrated with a test generation strategy, it should be straightforward to build the GUI dialogue box for the test template. To regulate the GUI presentation of the test template dialogue boxes, an abstract class has been created to act as the super class of all test template GUI dialogue boxes. Figure 6.8 shows part of the Java code in this

```
public abstract class AbstractTestDialog extends JDialog {
    // buttons for all test dialogue boxes
    protected JButton runButton;
    protected JButton closeButton;
    protected JButton helpButton;
    // constructor
    public AbstractTestDialog(JFrame frame)
    // public method: start the dialogue box
    public void start()
    // protected method: shared by all sub classes
    protected void closeButtonActionPerformed(ActionEvent evt)
    // protected abstract methods: to be implemented by sub classes
    protected abstract void initFieldsPanel();
    protected abstract void initComponents();
    protected abstract void runButtonActionPerformed(ActionEvent evt);
    protected abstract void helpButtonActionPerformed(ActionEvent evt);
}
```

Figure 6.8: Interface of the `AbstractTestDialog` class

`AbstractTestDialog` class. All concrete test dialogue boxes must contain three buttons: `runButton`, `closeButton`, and `helpButton`. `runButton` is used to invoke the underline test template, `closeButton` is used to close the current dialogue box, and `helpButton` is used to open a message box with instructions on how to use the current dialogue box. All concrete test dialogue box classes must implement the four abstract methods declared in `AbstractTestDialog`. The first two methods, `initFieldsPanel` and `initComponents`, should be used to create and layout GUI components that accept user input, such as the check boxes in the `ProtocolTest` GUI dialogue box and the text fields in the `UDPTest` GUI dialogue box. Next comes the `runButtonActionPerformed` method, which should contain code for invoking the corresponding test template. The last method, `helpButtonActionPerformed`, should be used to create the help message box for the current GUI dialogue box.

According to the discussion above, the process of extending PBit for a new iptables extension follows the steps shown in Figure 6.9.

For each new iptables extension *E*

1. Identify input parameters of *E*
   Determine user configurable input parameters
   Determine input parameters with fixed input domains
   Create a test template *T* for *E* by extending `TestTemplate`
2. Associate a test generation strategy with *T*
3. Create a GUI dialogue box *D* for *T* by extending `AbstractTestDialog`
4. Run test template *T* through *D*

Figure 6.9: Procedure of extending PBit

One of the most difficult parts in PBit is the implementation of raw sockets. The Java raw socket library implemented in PBit builds packets in byte arrays using Java input and out streams. It provides the following four native methods.

```
public class RawSocket
{
    // expect a library with name libRawSocket.so
    static { System.loadLibrary("RawSocket"); }
    public static native int open(String eth);
    public static native void send(int fd, byte[] b);
    public static native byte[] receive(int fd);
    public static native void close(int fd);
}
```

The `open` method accepts the name of a valid network interface and returns a file descriptor, which is a handle of the associated network device. The `send` method sends the specified byte stream to the network interface associated with the given file descriptor, and the `receive` method receives a stream of bytes from the network interface associated with the given file descriptor. Finally, the `close` method closes the network interface associated with the given file descriptor.

By using the Java Native Interface (JNI), this Java raw socket library can be mapped to native C implementations. If the native methods are declared in the file `RawSocket.java` and the native implementation is contained in the file `socketLibrary.c`, then the following procedure should be followed to build the raw socket library:

- **Precondition**: the LD_LIBRARY_PATH environment variable must include the directory where libRawSocket.so will be located.
- **Step 1**: compile `RawSocket.java`
- **Step 2**: create native interface using `javah`
- **Step 3**: compile `socketLibrary.c`
- **Step 4**: create the shared raw socket library

Implementation of the raw socket library used in PBit is given in Appendix C.

PBit focuses on testing the FORWARD chain of iptables. It is not difficult to test other iptables chains, but there is no easy way to fully automate the test procedure. For example, in order to test the INPUT chain, a process *P* must be started on SUT to receive inbound TCP or UDP packets. *P* can only access the body sections of the inbound packets because the header sections will be automatically processed by the system TCP/IP stack. Without the headers, it is not easy to check if the received packets are the same as the expected packets. Furthermore, in order to verify the received packets, *P* will have to notify the driver machine about what packets have been received. Automating the whole procedure would be difficult compared with the testing of the FORWARD chain. The testing of the OUTPUT chain will have similar problems as testing the INPUT chain. Note that all the match rules usable by INPUT and OUTPUT chains can be tested on the FORWARD chain. The only difference is that the source and destination of the packets are different. In the default test suite of iptables, only the FORWARD chain was tested.

As we just mentioned, the twelve test templates in PBit give examples to illustrate the creation of new test templates. In the next section, we will introduce the test results of experimenting with these twelve test templates.

6.5 Test results

The current version of PBit implements twelve test templates that cover six iptables parameters and seven iptables match extensions. The number of test tuples generated for each test template, as well as the test results, are listed in Table 6.2. The following is a summary of the test results:

- No functional error has been found.
- One specification error has been found. The iptables man page specifies that the `--port` option of the `multiport` extension *"match if both the source and the destination ports are equal to each other and to one of the given ports"*. According to our tests, it should say as *"match if either the source port or the destination port equal to one of the given ports"*.

6.6 Advantages of PBit

In chapter 3, we introduced the iptables testing research done by Prabhakar [25]. Based on the discussion in this section, we now summarize the difference between PBit and Prabhakar's iptables testing. First and foremost, Prabhakar aimed at doing a thorough functional testing of the current iptables, while PBit is a testing framework to be used by iptables testers for testing any iptables extension they are interested in. This is why Prabhakar's testing has covered more iptables extensions than what PBit has provided. The iptables extensions that have been covered, as well the test results, are the main components of Prabhakar's iptables testing. In contrast, the twelve test templates implemented in PBit are

Template	Number of test tuples	Functional error	Specification Error
Protocol	18	0	0
IP address	13122	0	0
In interface	6	0	0
Out interface	6	0	0
Fragment	6	0	0
TCP	72	0	0
UDP	72	0	0
ICMP	512	0	0
MAC	128	0	0
Limit	8	0	0
TOS	512	0	0
Multiport	72	0	1
Total	14534	0	1

Table 6.2: PBit test results

not the heart of this testing framework; instead, they are sample implementations that help the users of PBit understand the testing framework. The heart of PBit is the test generation strategies and the patterns used to build test templates, which enables the user to extend PBit for testing other iptables extensions.

The second advantage of PBit over Prabhakar's iptables testing is that PBit allows run-time test configuration. As long as the hardware connection remains the same as in Figure 2.3, PBit can be compiled once and reused everywhere. In Prabhakar's iptables testing, everytime a system parameter changes, the source code has to be modifed to reflect that change and the testing system has to be re-compiled.

The last but not the least advantage of PBit is that it applies test generation strategies in every test template. By using test generation strategies, it is more likely that the generated test set will be small enough to be executed efficiently, and large enough to reveal potential errors of the SUT. The following is a brief summary of the advantages of PBit:

1. **reusable**: The test generation strategies, as well as the test templates, can be reused in other test generation systems.

2. **extensible**: It is easy to add test templates for new iptables extensions.

3. **configurable**: The test configuration can be modified at run-time.

4. **portable**: PBit is implemented mostly in Java and is thus machine independent. Raw socket is implemented by reusing a third-party library written in C and can be replaced by any native program supporting raw socket.

In this chapter, we have introduced PBit, the GUI implementation of our iptables testing framework. We described the GUI features of PBit in detail by giving three examples. Furthermore, we discussed the design idea of PBit and compared it with Prabhakar's iptables testing.

Chapter 7. Conclusion

In this chapter, we summarize the contributions of our research and describe our future work.

7.1 Summary of contributions

We present the contributions of our research as follows:

1. We have introduced a testing framework for Linux iptables. This testing framework is based on test templates, which are parameterized test cases.

2. Twelve test templates have been created for testing iptables parameters and extensions. A GUI tool called PBit is provided to integrate these test templates with various test generation strategies. Three test generation strategies are available: Cartesian product generation, boundary values generation, and pairwise generation.

3. Pairwise test generation has been investigated in detail. Based on the "Order-Irrelevance" property, we proposed an improved IPO algorithm for pairwise generation. We have implemented both the original IPO algorithm and the improved IPO algorithm. A number of experiments have shown that the improved IPO algorithm always generates test sets smaller than or as large as the test sets generated by the original IPO algorithm. In some test scenarios, the improved IPO algorithm generates tests sets smaller than the AETG commercial tool.

4. The PBit testing framework is easy to extend by creating test templates for new iptables extensions.

5. Patterns in abstract test generation are pure mathematical models, and thus are easy to be reused in other software testing systems.

7.2 Future work

Our research on iptables testing has led to a few open topics for our future research work. The following is a brief summary of these topics:

1. Extend the IIPO algorithm for k-cover where k>2.

2. Find a pairwise generation strategy that is guaranteed to produce the minimum pairwise test sets.

3. Investigate iptables rules and try to use BNF to describe the iptables grammar. The objective is to build a compiler that parses and validates iptables scripts, and automatically generates test tuples for testing iptables.

4. Extend the Java programming language to support raw sockets.

Bibliography

[1] S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[3] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of The International Conference on Software Testing, Analysis, and Review (STAR)*, San Diego, USA, October 1998.

[4] D.M. Cohen, S.R. Dalal, M.L. Fredman, , and G.C. Patton. The aetg system: An ppproach to testing based on combinatorial design. *IEEE Trans. Software Eng.*, 23(7):437–443, 1997.

[5] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, SE-13(5):83–89, 1996.

[6] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, and C.J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, USA, May 2003.

[7] S.R. Dalal and C.L. Mallows. Factor-covering designs for testing software. *Technometrics*, 50(3):234–243, 1998.

[8] N. Daley, D. Hoffman, and P. Strooper. A framework for table driven testing of java classes. *Software - Practice and Experience*, 32(5):465–493, 2002.

[9] R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[10] K. Fujii. Jpcap home page. *http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html*.

[11] K. Fujii. TUN/TAP home page. *http://vtun.sourceforge.net/tun/*.

[12] N. Gupta, A.P. Mathur, and M.L. Soffa. UNA based iterative test data generation and its evaluation. In *Proceedings of The 14th IEEE Conference on Automated Software Engineering*, Cocoa Beach, USA, October 1999.

[13] T. Havana and J. Roning. Communication in the software vulnerability process. In *Proceedings of the 15th FIRST Conference on Computer Security Incident Handling*, pages 22–27, Ottawa, Canada, June 2003.

[14] D.M. Hoffman, D. Prabhakar, and P.A. Strooper. Testing iptables. In *Proceedings of Cascon 2003*, Markham, Canada, October 2003.

[15] D.M. Hoffman, P.A. Strooper, and L. White. Boundary values and automated component testing. *Journal of Software Testing, Verification, and Review*, 9(1):3–26, 1999.

[16] W.E. Howden. Functional program testing. *IEEE Transaction on Software Engineering*, SE-6(3):162–169, 1980.

[17] R. Kaksonen. A functional method for assessing protocol implementation security. *VTT Publications 448, Technical Research Centre of Finland*, 2001.

[18] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen. Analysis of vulnerabilities in internet firewalls. *Journal of Computers and Security*, 22(3):214–232, 2003.

[19] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceedings of the 27th NASA/IEEE Software Engineering Workshop*, Greenbelt, MD, USA, December 2002.

[20] S. Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison Wesley, 1999.

[21] Y.K. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. The relationship between test coverage and reliability. *Colorado State University Technical Report CS-94-110*, 1994.

[22] J. McDonald and P. A. Strooper. Translating Object-Z specifications to passive test oracles. In J. Staples, M. Hinchey, and S. Liu, editors, *Proc. International Conference on Formal Engineering Methods*, pages 165–174, Brisbane, Australia, December 1998. IEEE Computer Society Press.

[23] G.J. Meyers. *The Art of Software Testing.* John Wiley and Sons, 1979.

[24] A. Molitor. An architecture for advanced packet filtering. In *Proceedings of The 5th USENIX UNIX Security Symposium*, Salt Lake City, USA, June 1995.

[25] D. Prabhakar. Firewall testing. *Master's Thesis, Department of Computer Science, University of Victoria*, 2003.

[26] J. Roning, M. Laakso, A. Takanen, and A. Kaksonen. PROTOS - systematic approach to eliminate software vulnerabilities. *Invited presentation at Microsoft Research, Seattle, USA*, 2002.

[27] F. Ruskey. Combinatorial object server - information on permutations and combinations of a multiset. *http://www.theory.csc.uvic.ca/ cos/inf/mult/Multiset.html.*

[28] R. Russell and H. Welte. Linux netfilter hacking howto. *http://www.iptables.org*.

[29] M. Schiffman. *Building Open Source Network Security Tools*. Wiley, 2003.

[30] B. Stevens and E. Mendelsohn. Efficient software testing protocols. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Canada, November 1998.

[31] K. C. Tai and Y. Lei. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 254–261, Washington DC, USA, November 1998. IEEE.

[32] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transaction on Software Engineering*, JA-28(1):109–111, 2002.

[33] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of The 13th IEEE Conference on Automated Software Engineering*, Hawaii, USA, October 1998.

[34] G. Vigna, F. Valeur, J. Zhou, and R.A. Kemmerer. Composable tools for network discovery and security analysis. In *Proceedings of ACSAC 02*, Las Vegas, USA, December 2002.

[35] Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of The 13th International Conference on Testing of Communicating Systems (TestCom 2000)*, Ottawa, Canada, August 2000.

APPENDIX A. The Java Implementation of the IIPO Pairwise Generation Strategy

This appendix contains the Java source code of the improved IPO pairwise generation strategy described in chapter 4. The Java implementation of the IIPO strategy includes the following four classes:

- **PWIterator.java**: the implementation of the IIPO strategy
- **IntegerDomain.java**: a helper class that represents an input domain containing integers
- **Pair.java**: a helper class that represents a pair of elements
- **Element.java**: a helper class that represents an integer value in an input domain

---

A-1: PWIterator.java

```
package twocover;

import java.util.*;

public class PWIterator implements Iterator {

  private Vector domainVector = new Vector();
  private Vector minTupleSet = null;
  private int[] minOrder = null;
  private Vector tupleSet = new Vector();
  private Vector pairSet = new Vector();
  private int tupleIndex = 0;

  /**
   * Construct the test tuple set from the given input domains.
   * @param domainVector The set of input domains.
```

```
 * @pre domainVector not null but may be empty
 * @pre each domain in domainVector not null and not empty
 * @pre each element in a domain not null and supports equals
 * @post tupleSet is empty if domainVector empty
 * @post tupleSet is constructed if domainVector not empty
 */
public PWIterator(Vector v)
{
  int n = v.size();
  IntegerDomain[] domains = new IntegerDomain[n];
  domains[0] = new IntegerDomain(n);

  for(int i=1; i<n; i++) {
    IntegerDomain d = domains[i-1];
    domains[i] = new IntegerDomain(n, d);
  }

  // construct test set for each permutation of domainVector
  while(domains[0].hasNext()) {
    int[] tuple = new int[n];

    boolean stop = false;
    for(int i=0; i<n; i++) {
      int e = domains[i].getElement();
      for(int j=0; j<i&&!stop; j++) {
        if(e==tuple[j])
          stop = true;
      }
      if(!stop)
        tuple[i] = e;
    }

    if(!stop) {
      // use tuple to construct domainVector
      domainVector = new Vector();
      for(int i=0; i<n; i++) {
        domainVector.addElement(v.elementAt(tuple[i]));
      }

      // construct tupleSet
      tupleSet = new Vector();
      pairSet = new Vector();

      if (domainVector.size() == 1) {
        addOneDomain();
      } else if (domainVector.size() > 1) {
```

```
        // add the first two domains
        addTwoDomains();

        // add the rest domains if available
        for (int i=2; i<domainVector.size(); i++) {
          // get the next domain
          Vector newDomain = (Vector)domainVector.elementAt(i);

          // construct the set of new pairs to be covered
          makePairs(i);

          horizontalGrowth(newDomain, i);
          verticalGrowth(i);
        }

        // remove nulls generated in the last vertical growth
        removeRestNulls();
      }

      // update minTupleSet
      if(minTupleSet==null ||
       tupleSet.size()<minTupleSet.size()) {
        minTupleSet = tupleSet;
        minOrder = tuple;
      }
    }
  }

  // recover the original order
  tupleSet = new Vector();
  for(int i=0; i<minTupleSet.size(); i++) {
    Vector t = (Vector)minTupleSet.elementAt(i);
    Vector newTuple = new Vector(n);
    for(int j=0; j<n; j++) {
      newTuple.addElement(null);
    }

    for(int j=0; j<n; j++) {
      newTuple.setElementAt(t.elementAt(j), minOrder[j]);
    }
    tupleSet.addElement(newTuple);
  }
}

/*
 * Initialize tupleSet for a single domain
```

```
 */
private void addOneDomain()
{
  Vector domain = (Vector)domainVector.elementAt(0);
  for (int i=0; i<domain.size(); i++) {
    Vector newTuple = new Vector();
    newTuple.addElement(domain.elementAt(i));
    tupleSet.addElement(newTuple);
  }
}

/*
 * Initialize tupleSet for the first two domains
 */
private void addTwoDomains()
{
  Vector domain0 = (Vector)domainVector.elementAt(0);
  Vector domain1 = (Vector)domainVector.elementAt(1);

  for (int i=0; i<domain0.size(); i++) {
    for (int j=0; j<domain1.size(); j++) {
      Vector newTuple = new Vector();
      newTuple.addElement(domain0.elementAt(i));
      newTuple.addElement(domain1.elementAt(j));
      tupleSet.addElement(newTuple);
    }
  }
}

/*
 * Horizontal growth of the tupleSet
 */
private void horizontalGrowth(Vector newDomain, int index)
{
  if (tupleSet.size() <= newDomain.size()) {
    // add an element from the new domain to each tuple
    addElementDirectly(newDomain, tupleSet.size());
  } else {
    // add an element from the new domain
    addElementDirectly(newDomain, newDomain.size());

    // add an element that covers maximum pairs
    for (int i=newDomain.size(); i<tupleSet.size(); i++) {
      Vector tuple = (Vector)tupleSet.elementAt(i);

      // add an element that covers maximum pairs
```

```
      int maxValue = -1;
      int maxIndex = -1;
      for (int k=0; k<newDomain.size(); k++) {
        Element e = new Element(newDomain.elementAt(k), index);
        int n = pairCount(tuple, e);
        if (n > maxValue) {
          maxValue = n;
          maxIndex = k;
        }
      }
      tuple.addElement(newDomain.elementAt(maxIndex));
      handleNulls(tuple);

      // remove from pairSet pairs covered by the
      // extended tupleSet
      updatePairs(tuple);
    }
  }
}

/*
 * Vertical growth of the tupleSet
 */
private void verticalGrowth(int i)
{
  Vector tempTupleSet = new Vector();

  L:
  for (int j=0; j<pairSet.size(); j++) {
    Pair p = (Pair)pairSet.elementAt(j);

    Element firstElement = p.getFirstElement();
    int firstIndex = firstElement.getIndex();
    Element secondElement = p.getSecondElement();
    int secondIndex = secondElement.getIndex();

    // find a tuple with null at firstIndex and
    // secondElement at secondIndex
    for (int k=0; k<tempTupleSet.size(); k++) {
      Vector currentTuple =
       (Vector)tempTupleSet.elementAt(k);
      Object e1 = currentTuple.elementAt(firstIndex);
      Object e2 = currentTuple.elementAt(secondIndex);
      if(e1==null&&e2.equals(secondElement.getValue())) {
        currentTuple.setElementAt
         (firstElement.getValue(), firstIndex);
```

```
          continue L;
        }
      }

      // if not found, add a new tuple of the form
      // (...firstElement...secondElement)
      Vector v = createTupleWithNull(p);
      tempTupleSet.addElement(v);
    }

    // add elements in tempTupleSet to tupleSet
    for (int j=0; j<tempTupleSet.size(); j++) {
      Vector tuple = (Vector)tempTupleSet.elementAt(j);
      tupleSet.addElement(tuple);
      updatePairs(tuple);
    }

    if (pairSet.size() != 0)
      System.out.println(pairSet);
  }

  private void addElementDirectly(Vector newDomain, int bound)
  {
    for (int i=0; i<bound; i++) {
      Vector tuple = (Vector)tupleSet.elementAt(i);
      tuple.addElement(newDomain.elementAt(i));

      // check if removing nulls can remove uncovered pairs
      handleNulls(tuple);
      // remove pairs covered by the extended tupleSet
      updatePairs(tuple);
    }
  }

  /*
   * Check if removing nulls can remove uncovered pairs.
   */
  private void handleNulls(Vector tuple)
  {
    int size = tuple.size();
    Object ele = tuple.elementAt(size-1);
    for (int k=0; k<size-1; k++) {
      if (tuple.elementAt(k) == null) {
        Object e = getRightElement(k, ele);
        if (e != null)
          tuple.setElementAt(e, k);
```

```
    }
  }
}

private Object getRightElement(int index, Object e)
{
  Iterator iter = pairSet.iterator();
  while (iter.hasNext()) {
    Pair p = (Pair)iter.next();
    Element firstElement = p.getFirstElement();
    Element secondElement = p.getSecondElement();
    if (firstElement.getIndex() == index
      && secondElement.getValue().equals(e)) {
      return firstElement.getValue();
    }
  }
  return null;
}

/*
 * Replace nulls in all test cases by a randomly chosen
 * element in the correspondent domain.
 */
private void removeRestNulls()
{
  for (int j=0; j<tupleSet.size(); j++) {
    Vector tuple = (Vector)tupleSet.elementAt(j);
    for (int i=0; i<tuple.size(); i++) {
      if (tuple.elementAt(i) == null) {
        // add an element randomly
        Vector domain = (Vector)domainVector.elementAt(i);
        int random = (int)(Math.random()*(domain.size()-1));
        tuple.setElementAt(domain.elementAt(random), i);
      }
    }
  }
}

/*
 * Create a new test tuple that covers a given pair
 */
private Vector createTupleWithNull(Pair p)
{
  Vector tuple = new Vector();

  Element firstElement = p.getFirstElement();
```

```
  Element secondElement = p.getSecondElement();
  int firstIndex = firstElement.getIndex();
  int secondIndex = secondElement.getIndex();

  for (int i=0; i<secondIndex; i++) {
    if (i == firstIndex)
      tuple.addElement(firstElement.getValue());
    else
      tuple.addElement(null);
  }

  tuple.addElement(secondElement.getValue());

  return tuple;
}

/*
 * Count the number of pairs covered by adding an element
 */
private int pairCount(Vector tuple, Element e)
{
  int n = 0;

  for (int i=0; i<tuple.size(); i++) {
    if (tuple.elementAt(i) != null) {
      Element ele = new Element(tuple.elementAt(i), i);
      Pair p = new Pair(ele, e);
      if (pairSet.contains(p))
        n++;
    } else {
      Vector domain = (Vector)domainVector.elementAt(i);
      for (int j=0; j<domain.size(); j++) {
        Element ele = new Element(domain.elementAt(j), i);
        Pair p = new Pair(ele, e);
        if (pairSet.contains(p)) {
          n++;
          break;
        }
      }
    }
  }

  return n;
}

/*
```

```
 * Remove pairs covered by the updated tupleSet from pairSet
 */
private void updatePairs(Vector tuple)
{
  int size = tuple.size() - 1;
  for(int j=0; j<size; j++) {
    Object e = tuple.elementAt(j);
    if (e != null) {
      Element e1 = new Element(e, j);
      Element e2 = new Element(tuple.elementAt(size), size);
      Pair p = new Pair(e1, e2);
      pairSet.removeElement(p);
    }
  }
}

/*
 * Add pairs generated by a new domain to the pairSet
 */
private void makePairs(int index)
{
  Vector newDomain = (Vector)domainVector.elementAt(index);
  for (int i=0; i<index; i++) {
    Vector domain = (Vector)domainVector.elementAt(i);
    for (int j=0; j<domain.size(); j++) {
      Element e1 = new Element(domain.elementAt(j), i);
      for (int k=0; k<newDomain.size(); k++) {
        Element e2 = new Element(newDomain.elementAt(k), index);
        Pair p = new Pair(e1, e2);
        pairSet.addElement(p);
      }
    }
  }
}

/**
 * Get the next test tuple.
 * @return Object the next test tuple.
 */
public Object next() throws NoSuchElementException
{
  if (!hasNext()) {
    throw new NoSuchElementException();
  }

  Vector tuple = (Vector)tupleSet.elementAt(tupleIndex++);
```

```
    return tuple;
  }

  /**
   * Check if there are more test tuples.
   * @return boolean true if more test tuples are available.
   */
  public boolean hasNext()
  {
    return (tupleIndex < tupleSet.size());
  }

  /**
   * Remove the next test tuple.
   */
  public void remove() throws UnsupportedOperationException
  {
    throw new UnsupportedOperationException();
  }

}
```

A-2: IntegerDomain.java

```
package twocover;

import java.util.*;

/**
 * This class represents an integer domain that, given
 * an integer n, contains integers from 0 to n-1.
 *
 * To give all permutations of n integers, we create n
 * IntegerDomain objects, each containing n integers 0
 * to n-1. These n objects are organized as a linked list
 * and each object has an index to * indicate which value
 * in this domain should be visited next. Except for the
 * first domain, when the index of a domain hits the upper
 * bound, the index will be reset to 0 and the previous
 * domain is called to increase its index. When the index
 * of the first domain hits the upper bound, the permutation
 * is done.
 */
```

```
public class IntegerDomain {

  // a set of integers
  private int[] intSet;

  // the size of the domain
  private int size;

  // the index of the current element in the domain
  private int index;

  // check if the first domain has been traversed
  private boolean finished;

  // the previous integer domain
  private IntegerDomain prevDomain;

  // the next integer domain
  private IntegerDomain nextDomain;

  /**
   * Initialize an integer domain.
   * @param n The size of the domain.
   */
  public IntegerDomain(int n)
  {
    initIntSet(n);

    prevDomain = null;
    nextDomain = null;
  }

  /**
   * Initialize an integer domain.
   * @param n The size of the domain.
   * @param d The previous integer domain.
   */
  public IntegerDomain(int n, IntegerDomain d)
  {
    initIntSet(n);

    prevDomain = d;
    nextDomain = null;
    d.nextDomain = this;
  }
```

```
/**
 * Get the current element in the domain.
 * @return The current element in the domain.
 */
public int getElement()
{
  int e = intSet[index];

  if (isLastDomain()) {
    index++;
    if (index >= size) {
      index = 0;
      if (isFirstDomain())
        finished = true;
      else
        prevDomain.increaseIndex();
    }
  }

  return e;
}

/**
 * Get the current element in the domain.
 * @return The current element in the domain.
 */
public void increaseIndex()
{
  index++;

  if (index >= size) {
    index = 0;
    if (isFirstDomain())
      finished = true;
    else
      prevDomain.increaseIndex();
  }
}

/**
 * Check if the domain is the first domain in
 * a domain list.
 * @return true if the domain is the first domain,
 * otherwise false.
 */
public boolean isFirstDomain()
```

```
  {
    return prevDomain==null;
  }

  /**
   * Check if the domain is the last domain in a
   * domain list.
   * @return true if the domain is the last domain,
   * otherwise false.
   */
  public boolean isLastDomain()
  {
    return nextDomain==null;
  }

  /**
   * Check if the domain has elements left.
   * @return true if the domain has elements left,
   * otherwise false.
   */
  public boolean hasNext()
  {
    return !finished;
  }

  /*
   * Initialize the integer set.
   * @param n The size of the domain.
   */
  private void initIntSet(int n)
  {
    intSet = new int[n];
    for (int i=0; i<n; i++)
      intSet[i] = i;

    size = n;
    index = 0;
    finished = false;
  }

}
```

A-3: Pair.java

```
package twocover;

/**
 * This class represents a pair of elements, where
 * each element is from an input domain. This class
 * is used to do pairwise checking.
 */
public class Pair {

  private Element x;
  private Element y;

  /**
   * Constructor: Initialize this pair.
   * @param x The first element of this pair.
   * @param y The second element of this pair.
   */
  public Pair(Element x, Element y)
  {
    this.x = x;
    this.y = y;
  }

  /**
   * Get the first element of this pair.
   * @return The first element of this pair.
   */
  public Element getFirstElement()
  {
    return x;
  }

  /**
   * Get the second element of this pair.
   * @return The second element of this pair.
   */
  public Element getSecondElement()
  {
    return y;
  }

  /**
   * Check if the given object is equal to this pair.
   * @param obj The given object to compare to.
   * @return true if the given object is equal to this pair.
```

```
   */
  public boolean equals(Object obj)
  {
    Pair p = (Pair)obj;

    if (x.equals(p.x) && y.equals(p.y))
      return true;
    return false;
  }

  /**
   * Get the string representation of this pair.
   * @return The string representation of this pair.
   */
  public String toString()
  {
    return ("<" + x.toString() + "," + y.toString() + ">");
  }

}
```

A-4: Element.java

```
package twocover;

/**
 * This class represents an element in an input domain.
 * The element is specified as a pair:
 *   (the value of the element, the index in the input domain)
 */
public class Element {

  private Object value;
  private int index;

  /**
   * Constructor: Initialize an element.
   * @param value The actual object value of this element.
   * @param index The index of this element in an input domain.
   */
  public Element(Object value, int index)
  {
    this.value = value;
```

```
    this.index = index;
  }

  /**
   * Get the value of this element.
   * @return The object value of this element.
   */
  public Object getValue()
  {
    return value;
  }

  /**
   * Get the index of this element.
   * @return The index of this element.
   */
  public int getIndex()
  {
    return index;
  }

  /**
   * Check if two elements are equal.
   * @param obj An object to compare to.
   * @return true if the given object is equal to this element.
   */
  public boolean equals(Object obj)
  {
    Element e = (Element)obj;

    if (this.value.equals(e.value) && this.index == e.index)
      return true;
    return false;
  }

  /**
   * Get the string representation of this element.
   * @return The string representation of this element.
   */
  public String toString()
  {
    return value.toString();
  }

}
```

APPENDIX B. Test Program of the IIPO Implementation

This appendix contains test programs for pairwise generation. The test programs are written in Java and include four classes:

- **PWTester.java**: the test program doing the pairwise checking
- **Generate.java**: a helper class that generates pairwise test sets
- **Generate1.java**: a sample sub class of Generate.java that can be used to generate pairwise test sets for five hard-coded input domains.
- **Driver.java**: the main program

B-1: PWTester.java

```
import java.util.*;
import roast.*;
import twocover.*;

/**
 * This class tests if a given test set is a pairwise
 * test set.
 */
public class PWTester {

  private Vector domainVector = new Vector();

  /**
   * Constructor: initialize the input domains.
   * @param domainVector The set of input domains.
   */
  public PWTester(Vector domainVector)
  {
    this.domainVector = domainVector;
  }

  /**
   * Test if the given test set is a pairwise test set.
   * @return true if the given test set is pairwise.
   */
  public boolean test(Vector tuples)
```

```
{
  return (testPosition(tuples) && testCoverage(tuples));
}

/*
 * Test if all test tuples are in the correct order.
 * @return true if all test tuples are in the correct order.
 */
private boolean testPosition(Vector tuples)
{
  boolean pass = true;

  // check the order of each test tuple
  for (int i=0; i<tuples.size(); i++) {
    Vector tuple = (Vector)tuples.elementAt(i);
    for (int j=0; j<tuple.size(); j++) {
      Object e = tuple.elementAt(j);
      Vector domain = (Vector)domainVector.elementAt(j);
      if (!domain.contains(e)) {
        pass = false;
        System.out.println("FAIL: element at " + j +
         " of test tuple " + tuple +
         " is not from the right domain");
      }
    }
  }

  return pass;
}

/*
 * Test if the given test set covers all pairs.
 * @return true if the given test set covers all pairs.
 */
private boolean testCoverage(Vector tuples)
{
  boolean pass = true;
  for (int i=0; i<domainVector.size()-1; i++) {
    for (int j=i+1; j<domainVector.size(); j++) {
      Vector domain0 = (Vector)domainVector.elementAt(i);
      Vector domain1 = (Vector)domainVector.elementAt(j);
      Vector cpVector = new Vector();
      cpVector.addElement(domain0);
      cpVector.addElement(domain1);

      Vector pairSet = new Vector();
```

```
        Iterator cpIter = new CPIterator(cpVector);
        while (cpIter.hasNext()) {
          Vector v = (Vector)cpIter.next();
          pairSet.addElement(v);
        }

        for (int k=0; k<tuples.size(); k++) {
          Vector tuple = (Vector)tuples.elementAt(k);
          for (int m=0; m<tuple.size()-1; m++) {
            for (int n=m+1; n<tuple.size(); n++) {
              Vector v = new Vector();
              v.addElement(tuple.elementAt(m));
              v.addElement(tuple.elementAt(n));
              pairSet.remove(v);
            }
          }
        }

        if (!pairSet.isEmpty()) {
          pass = false;
          System.out.println("FAIL: some pair is not covered");
          for (int m=0; m<pairSet.size(); m++) {
            System.out.println("\t" + pairSet.elementAt(m));
          }
        }
      }
    }

    return pass;
  }

}
```

B-2: Generate.java

```
import java.util.*;
import roast.*;
import twocover.*;

/**
 * This class is used to generate pairwise test sets
 * for integer input domains. An integer input domain
 * is determined by its index and size. For example,
```

```
 * an integer input domain with index 5 and size 4
 * is the vector <51,52,53,54>.
 */
public abstract class Generate {

  /**
   * Generate a pairwise test set.
   * @param debug The debug flag.
   */
  public void generate(boolean debug)
  {
    Vector domainVector = loadDomain();
    Iterator iter = new PWIterator(domainVector);
    int count = 0;
    while (iter.hasNext()) {
      Vector v = (Vector)iter.next();
      count++;

      if(debug) {
        System.out.print("(");
        for (int i=0; i<v.size(); i++) {
          Integer e = (Integer)v.elementAt(i);
          System.out.print(" " + e.intValue() + " ");
        }
        System.out.println(")");
      }
    }
    System.out.println("number of test cases="+count);
    System.out.println();
  }

  /**
   * Add an input domain to the set of input domains.
   * @param v The set of input domains.
   * @param index The index of the new input domain.
   * @param size The size of the new input domain.
   */
  public void addDomain(Vector v, int index, int size)
  {
    VectorDomain d = new VectorDomain();
    for(int i=0; i<size; i++)
      d.addElement(new Integer(index*10 + i));
    v.addElement(d);
  }

  /**
```

```
   * Create a set of input domains.
   * @return A set of input domains.
   */
  public abstract Vector loadDomain();

}
```

B-3: Generate1.java

```
import java.util.*;
import roast.*;
import twocover.*;

/**
 * This is a sample sub class of Generate that creates
 * a set of five input domains.
 */
public class Generate1 extends Generate {

  /**
   * Create a set of input domains.
   * @return A set of input domains.
   */
  public Vector loadDomain()
  {
    System.out.println("Generate1: 5 domains");
    System.out.println("\t2 2-value domain");
    System.out.println("\t1 3-value domain");
    System.out.println("\t2 4-value domain");

    Vector domainVector = new Vector();

    // 2 2-value domains
    for(int i=0; i<2; i++)
      addDomain(domainVector, i+1, 2);

    // 1 3-value domains
    for(int i=2; i<3; i++)
      addDomain(domainVector, i+1, 3);

    // 2 4-value domains
    for(int i=3; i<5; i++)
      addDomain(domainVector, i+1, 4);
```

```
    return domainVector;
  }

}
```

B-4: Driver.java

```
import java.util.*;
import twocover.*;

/**
 * This is a sample driver program that uses Generate1
 * to generate and test a pairwise test set for a set
 * of five input domains.
 */
public class Driver {

  public static void main(String args[])
  {
    Vector domainVector = (new Generate1()).loadDomain();
    Vector tuples = new Vector();
    Iterator iter = new PWIterator(domainVector);

    // generate the pairwise test set
    while (iter.hasNext()) {
      Vector v = (Vector)iter.next();
      tuples.addElement(v);
    }

    // test the pairwise test set
    PWTester tester = new PWTester(domainVector);
    System.out.println("********** start **********");
    long start = System.currentTimeMillis();
    boolean pass = tester.test(tuples);
    long end = System.currentTimeMillis();
    System.out.println("Tuples generated: "+tuples.size());
    System.out.println("Test result: " +
     (pass==true ? "PASS" : "FAIL"));
    System.out.println("Time taken in testing: " +
     (end-start) + " milliseconds");
    System.out.println("********** stop **********");
  }
}
```

APPENDIX C. Implementation of the Java raw socket library in PBit

This appendix contains the source code for the raw socket library used in PBit. The following four files are included:

- **RawSocket.java**: declares the interface of the Java raw socket library
- **socketLibrary.h**: declares the interface of the C raw socket library
- **socketLibrary.c**: contains implementation of the C raw socket library
- **Makefile**: a make file for building the raw socket library

---

C-1: RawSocket.java

```
import java.io.*;

/**
 * This class provides JNI interfaces for accessing
 * corresponding native raw socket functions.
 *
 * See socketLibrary.c for native function implementations.
 */
public class RawSocket {

  // expect a library with name libRawSocket.so
  static { System.loadLibrary("RawSocket"); }

  public static native int open(String eth);
  public static native void send(int fd, byte[] b);
  public static native byte[] receive(int fd);
  public static native void close(int fd);

}
```

C-2: socketLibrary.h

```
#ifndef SOCKETLIBRARY_H
```

```
#define SOCKETLIBRARY_H

#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <net/if.h>
#include <net/if_arp.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/udp.h>
#include <linux/tcp.h>
#include <linux/if_packet.h>

/*************************************************************
* socketLibrary.h
*
* This C socket library has been modified from the socket
* library created by Durga Prabhakar.
*************************************************************/

/*
 * Open a raw socket with protocol ETH_P_ALL and returns its
 * file descriptor.
 */
int ss_socket();

/*
 * Bind raw socket fd to ethernet interface eth.
 * preconditions:
 *    - fd is a open file descriptor returned by ss_socket
 *    - eth is of the form ethN where N is an integer
 */
void ss_bind(int fd,char* eth);

/* Send a raw ethernet packet.
 * preconditions:
 * - fd is a open file descriptor returned by ss_socket
 * - fd is bound to an available ethernet interface
 * - buf points to at least len bytes where len in [64..1514]
 */
void ss_send(int fd,unsigned char* buf,int len);

/*
 * Receive next packet and return the number of bytes
 * read or -1 on error.
 * Note: because ss_socket uses ETH_P_ALL packets of any
```

```
 * protocol are captured. The packet read is contained in p.
 *
 * preconditions:
 *    - fd is a open file descriptor returned by ss_socket
 *    - fd is bound to an available ethernet interface
 *    - p points to at least 1514 bytes
 *    - t is the timeout in seconds, default t is 3 seconds
 */
int ss_receive(int fd,unsigned char* p,int t);

/*
 * Close socket fd.
 * preconditions: fd is a open file descriptor returned
 * by ss_socket
 */
void ss_close(int fd);

#endif
```

C-3: socketLibrary.c

```
/************************************************************
* socketLibrary.c
*
* This file contains native raw socket implementation needed
* by the Java raw socket library.
*
* This C socket library has been modified from the socket
* library created by Durga Prabhakar.
************************************************************/

#include <errno.h>
#include "socketLibrary.h"
#include "RawSocket.h"

#define DEFAULT_TIMEOUT 3

static int debug = 0;
static unsigned char buf[1514];


//========== local helper functions for socket
```

```
/*
 * Return sockaddr_ll whose ifindex = i's index and
 * family = AF_PACKET.
 */
struct sockaddr_ll getLinkLayerAddress(int fd,char* i)
{
  struct ifreq ifr;
  struct sockaddr_ll interfaceAddr;

  memset(&interfaceAddr,0,sizeof(interfaceAddr));
  memset(&ifr,0,sizeof(ifr));
  memcpy(&ifr.ifr_name,i,IFNAMSIZ);
  ioctl(fd,SIOCGIFINDEX,&ifr);

  interfaceAddr.sll_ifindex = ifr.ifr_ifindex;
  interfaceAddr.sll_family = AF_PACKET;
  return(interfaceAddr);
}

/*
 * Set the network interface to be in promiscuous mode.
 */
void setPromiscuousMode(int fd,char*i)
{
  struct ifreq ifr;
  struct packet_mreq mreq;

  memset(&ifr,0,sizeof(ifr));
  memcpy(&ifr.ifr_name,i,IFNAMSIZ);
  ioctl(fd,SIOCGIFINDEX,&ifr);

  memset(&mreq,0,sizeof(mreq));
  mreq.mr_ifindex = ifr.ifr_ifindex;
  mreq.mr_type = PACKET_MR_PROMISC;
  mreq.mr_alen = 6;

  setsockopt(fd,SOL_PACKET,PACKET_ADD_MEMBERSHIP,
   (void*)&mreq, (socklen_t)sizeof(mreq));
}


//========== local functions for socket operations

/*
 * Open a socket and bind it to the given interface.
 */
```

```
int ss_socket()
{
  int fd;
  if((fd=socket(PF_PACKET,SOCK_RAW,htons(ETH_P_ALL)))==-1)
    perror("Error creating packet socket\n");
  return(fd);
}

/*
 * Bind a socket file descriptor to a given interface.
 */
void ss_bind(int fd,char* i)
{
  struct sockaddr_ll interfaceAddr = getLinkLayerAddress(fd,i);
  if(bind(fd,(struct sockaddr*) &
   interfaceAddr,sizeof(interfaceAddr))<0)
    perror("Error binding socket\n");
  setPromiscuousMode(fd,i);
}

/*
 * Send a byte array out of the given interface.
 */
void ss_send(int fd,unsigned char* buf,int len)
{
  if(send(fd,buf,len,0) < 1)
    perror("Error while sending\n");
}

/*
 * Receive a byte array from the given interface.
 */
int ss_receive(int fd,unsigned char* p,int t)
{
  int s, n = 0;
  fd_set fdRead;
  struct timeval tv;

  // default timeout is 3 seconds
  if(t<0)  t = DEFAULT_TIMEOUT;

  FD_ZERO(&fdRead);
  FD_SET(fd, &fdRead);
  tv.tv_usec = 0;
  tv.tv_sec = t;
```

```
  if((s = select(fd+1,&fdRead,NULL,NULL,&tv))<0)
    perror("Error in select\n");
  else{
    if(s>0 && FD_ISSET(fd,&fdRead)) {
      if((n=read(fd,p,1514)) < 0)
        perror("Error while receiving\n");
    }
  }

  return n;
}

/*
 * Close a socket bound to a given interface.
 */
void ss_close(int fd)
{
  close(fd);
}


//========== native implementation of the Java raw
//========== socket library

/*
 * Open a socket and bind it to the given interface.
 */
JNIEXPORT jint JNICALL Java_RawSocket_open
  (JNIEnv *env, jclass jc, jstring js)
{
  char buf[10];
  int fd = ss_socket();
  const char *s = (*env)->GetStringUTFChars(env,js,0);
  strcpy(buf, s);
  ss_bind(fd, buf);

  if(debug) {
    printf("open interface: %s\n",buf);
    printf("fd: %d\n",fd);
  }

  return fd;
}

/*
 * Send a byte array to the given interface.
```

```
 */
JNIEXPORT void JNICALL Java_RawSocket_send
  (JNIEnv *env, jclass jc, jint ji, jbyteArray jba)
{
  int i;
  jsize size = (*env)->GetArrayLength(env,jba);
  jbyte *body = (*env)->GetByteArrayElements(env,jba,0);
  unsigned char frame[1514];

  if(debug) printf("%d bytes sent to eth1 in C:\n", size);

  for(i=0; i<size; i++) {
    frame[i] = body[i];
    if(debug) printf("%02x ", frame[i]);
  }
  if(debug) printf("\n");

  ss_send(ji, (unsigned char*)frame, size);

  (*env)->ReleaseByteArrayElements(env, jba, body, 0);
}

/*
 * Receive a byte array from the given interface.
 */
JNIEXPORT jbyteArray JNICALL Java_RawSocket_receive
  (JNIEnv *env, jclass jc, jint fd)
{
  int i;
  int n = 0;
  jbyteArray jba;
  jbyte *jb;
  unsigned char packet[1514];

  n = ss_receive(fd, packet, DEFAULT_TIMEOUT);

  if(debug) {
    printf("%d bytes received at eth2 in C:\n", n);
    for (i=0; i<n; i++) {
      printf("%02x ", packet[i]);
    }
    printf("\n");
  }

  if(debug)  printf("bytes recieved: %d\n", n);
```

```
  jb = (jbyte*)malloc(sizeof(jbyte)*n);

  jba = (*env)->NewByteArray(env, n);
  for(i=0; i<n; i++)
    jb[i] = (jbyte)packet[i];

  (*env)->SetByteArrayRegion(env, jba, 0, n, jb);

  return jba;
}

/*
 * Close a socket bound to the given interface.
 */
JNIEXPORT void JNICALL Java_RawSocket_close
  (JNIEnv *env, jclass jc, jint ji)
{
  ss_close(ji);
}
```

C-4: Makefile

```
# This is the make file for building the raw socket library
#
# precondition:
# LD_LIBRARY_PATH includes the directory containing libRawSocket.so

JAVAINC = /usr/java/j2sdk/include
JAVALINUXINC = /usr/java/j2sdk/include/linux

all: libRawSocket.so

libRawSocket.so: socketLibrary.o
  gcc -shared -o libRawSocket.so socketLibrary.o

socketLibrary.o: socketLibrary.c socketLibrary.h RawSocket.h
  gcc -I\$(JAVAINC) -I\$(JAVALINUXINC) -c socketLibrary.c

RawSocket.h: RawSocket.class
  javah -jni RawSocket

RawSocket.class: RawSocket.java
  javac RawSocket.java
```

```
clean:
  rm -f libRawSocket.so socketLibrary.o *.class RawSocket.h
```