Imperial College London
Department of Computing

SymbexNet: Checking Network Protocol Implementations using Symbolic Execution

JaeSeung Song

February 2013

Supervised by Professor Kin K. Leung and Dr. Peter Pietzuch

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Abstract

The implementations of network protocols, such as DNS, DHCP and Zeroconf, are prone to flaws, security vulnerabilities and interoperability issues caused by ambiguous requirements in protocol specifications. Detecting such problems is not easy because (i) many bugs manifest themselves only after prolonged operation; (ii) the state space of complex protocol implementations is large; and (iii) problems often require additional information about correct behaviour from specifications.

This thesis presents a novel approach to detect various types of flaws in network protocol implementations by combining symbolic execution and rule-based packet matching. The core idea behind our approach is to generate automatically high-coverage test input packets for a network protocol implementation. For this, the protocol implementation is run using a symbolic execution engine to obtain test input packets. These packets are then used to detect potential violations of rules that constrain permitted input and output packets and were derived from the protocol specification. We propose a technique that repeatedly performs symbolic execution on selected test input packets to achieve broad and deep exploration of the implementation state space. In addition, we use the generated test packets to check interoperability between different implementations of the same network protocol.

We present a system based on these techniques, SYMBEXNET, and show that it can automatically generate test input packets that achieve high source code coverage and discover various bugs. We evaluate SYMBEXNET on multiple implementations of two network protocols: Zeroconf, a service discovery protocol, and DHCP, a network configuration protocol. SYMBEXNET is able to discover non-trivial bugs as well as interoperability problems, most of which have been confirmed by the developers.

Acknowledgements

I would like to express my deeply-felt thanks to my first supervisor Professor Kin K. Leung for his warm encouragement, thoughtful guidance and support throughout the entire research. I would also like to thank to my second supervisor Dr. Peter Pietzuch for his patience, motivation, enthusiasm, and immense knowledge, without whom this work would not have been possible. I appreciate his tolerance and his time spent on advising me and evaluating my work. I would also like to thank Dr. Cristian Cadar for his assistance and the technical discussions on verification techniques including symbolic execution. He has given me a lot of inspiration and suggestions during my PhD as the adjunct supervisor.

I wish to thank my colleagues and friends with whom I had many interesting discussions, in particular Sungwoo Jeon, Dr. Athanasios Gkelias, Dr. Cheolsoo Park, Dr. Haseong Kim, Dr. David Looney and Jaehwa Lee.

Finally, and most importantly, I would like to thank my wife Jongsook who has always been beside me, showing great patience and sharing my concerns throughout these years. I thank my parents and parents-in-law for their faith in me and allowing me to be as ambitious as I wanted to be. To my beloved daughter Jiwon and my precious son Jiho, I would like to express my thanks for being such good kids, always cheering me up.

Contents

1. Introduction **1**

1.1. Motivation . . . 1

1.1.1. Where do protocol implementation errors come from? . . . 2

1.1.2. What effect can protocol implementation errors have? . . . 2

1.2. Current Techniques for Checking Protocol Implementations . . . 3

1.3. Research Statement . . . 4

1.4. Summary of Contributions . . . 6

1.5. Thesis Outline . . . 6

2. Background **9**

2.1. Network Protocols . . . 9

2.2. Network Verification . . . 15

2.2.1. Goals of Network Verification . . . 15

2.2.2. Verification Approaches . . . 16

2.3. An Example of Model Checking . . . 20

2.3.1. Self-Configuration of Base Stations . . . 21

2.3.2. Discussion . . . 22

2.4. Network Testing . . . 23

2.4.1. Testing Techniques . . . 23

2.4.2. Symbolic Execution . . . 24

2.5. Summary . . . 28

3. Checking Network Protocol Implementations **29**

3.1. Requirements . . . 30

3.2. SYMBEXNET Architecture . . . 31

3.3. Symbolic Execution and Exploration . . . 33

3.3.1. Symbolic Marking . . . 34

3.3.2. Symbolic Packet Injection . . . 35

3.4. Single Packet Exchange Symbolic Execution . . . 36

3.4.1. Test Packet Generation . . . 36

3.4.2. Test Packet Replay . . . 36

3.5. Multi Packet Exchange Symbolic Execution . . . 38
3.5.1. Problem of Symbolic Exploration . . . 38
3.5.2. Generation of Packet Sequences . . . 40
3.5.3. Formal Model . . . 42
3.5.4. Code Exploration . . . 43
3.5.5. Replaying Sequences of Test Packets . . . 45
3.6. SYMBEXNET Implementation . . . 45
3.6.1. Protocol Specific Components . . . 45
3.6.2. Common Components . . . 46
3.7. Summary . . . 47

4. Rule-based Verifiable Specifications **49**
4.1. Packet Analysis Techniques . . . 50
4.1.1. Overview of Packet Analysis Techniques . . . 50
4.1.2. Rule Matching . . . 51
4.2. Packet Rule Validation . . . 52
4.3. Rule Extraction . . . 53
4.4. Packet Rule Description Language . . . 55
4.4.1. Types of Packet Rule . . . 56
4.4.2. Grammar of Packet Rule Language . . . 56
4.4.3. Rule and Operators . . . 57
4.5. Examples . . . 60
4.6. Rule Implementation and Validation . . . 61
4.7. Related Work on Rule-based Analysis . . . 63
4.8. Summary . . . 64

5. Symbolic Interoperability Testing **65**
5.1. Overview . . . 65
5.2. Symbolic Interoperability Checking . . . 67
5.2.1. Overview . . . 68
5.2.2. Formal Model of Interoperability Relation . . . 70
5.2.3. Symbolic Interoperability Testing . . . 71
5.3. Generation and Execution of Interoperability Test Packets . . . 72
5.4. Rule-based Interoperability Checking . . . 74
5.4.1. Interoperability Rules . . . 74
5.4.2. Features of IOT Rules . . . 75
5.4.3. Interoperability Decision Criteria . . . 76
5.5. Related Work . . . 77
5.6. Summary . . . 78

6. Evaluation **79**
6.1. Overview . . . 79
6.2. Rule Derivation . . . 81
6.3. Experimental Set-up . . . 82
6.4. Single Packet Exchange Symbolic Execution . . . 83
6.4.1. Test Packet Generation . . . 84
6.4.2. Line Coverage Results . . . 87
6.4.3. Discovered Implementation Errors . . . 88
6.5. Multi Packet Exchange Symbolic Execution . . . 90
6.5.1. Generation of Test Sequences . . . 90
6.5.2. Line Coverage Results . . . 92
6.5.3. Source Code Analysis . . . 92
6.6. Symbolic Interoperability Testing . . . 95
6.6.1. Interoperability Rule Derivation . . . 95
6.6.2. Interoperability Test Packet Generation . . . 96
6.6.3. Detected Interoperability Errors . . . 97
6.7. Discovered Implementation Errors . . . 99
6.8. Summary . . . 102

7. Conclusions **105**
7.1. Future Work . . . 106

Bibliography **109**

A. Evaluation Results for Case Study **124**

B. Packet Rules **127**
B.1. Packet Rules for the Zeroconf Protocol . . . 127
B.2. Packet Rules for the DHCP Protocol . . . 132

List of Tables

4.1. Main syntax of regular expressions 52
4.2. The keywords for describing the requirements in IETF standard documents [Bra97]. 54
4.3. Input test packets used in Figure 4.5. 63

6.1. The summary of the network daemons tested using SYMBEXNET. 80
6.2. The results of the rule derivation from the specifications. 81
6.3. A set of network services created for the Zeroconf experiments. 83
6.4. Summary of symbolic execution for all combinations of packet fields 86
6.5. Line coverage of each daemon for generated test packets. 88
6.6. Generated sequences of packets for DHCP. 91
6.7. Results after applying MPE-SE to the DHCP daemons. 91
6.8. Comparison of source code coverage measurements for the `udhcp` and `isc-dhcp` servers. 93
6.9. A complete list of all detected bugs for each class 99
6.10. Detected inconsistencies through interoperability testing for Zeroconf daemons. . 101

List of Figures

1.1. Typical network service development, from protocol specification to implementation 2
1.2. An example of network implementation errors occurring in a modern office environment . . . 3

2.1. The relationship between protocol, specification and implementation . . . 11
2.2. Message format for Zeroconf . . . 12
2.3. Simplified state transition diagram for Zeroconf . . . 13
2.4. Message format for DHCP . . . 14
2.5. Simplified state transition diagram for DHCP . . . 14
2.6. The life cycle of network protocol software development and applicable formal verification techniques. . . . 16
2.7. Conceptual view of system verification using model checking . . . 18
2.8. Examples of collision and confusion in PCI assignment . . . 21
2.9. Source code that handles a received input packet. . . . 25
2.10. The corresponding symbolic execution tree to Figure 2.9 . . . 26
2.11. Overview of symbolic execution and the structure of KLEE. . . . 27

3.1. SYMBEXNET system architecture. . . . 32
3.2. Original and symbolic packets. . . . 34
3.3. Marking modes for a symbolic input packet. . . . 35
3.4. An example of network implementation source code. . . . 36
3.5. The replay process for a test input packet p. . . . 37
3.6. An example C program for a state machine in a traditional network protocol implementation. . . . 39
3.7. High-level overview of multi packet exchange symbolic execution. . . . 41
3.8. Pseudocode for MPE-SE. . . . 41
3.9. The program exploration method for a) SPE-SE and b) MPE-SE. . . . 44

4.1. A generic network monitoring system that is comprised of two main components: a pre-processor and a detection engine. . . . 51
4.2. Procedure for packet rule validation. . . . 53
4.3. Packet rule grammar written in as extended BNF. . . . 56

4.4. High level packet rule automaton. 62
4.5. An example of rule automaton. 62

5.1. (a) Conformance testing ensures that an implementation complies with its protocol specifications; (b) Interoperability testing ensures that different implementations work as expected 66
5.2. Typical configuration for interoperability testing in industry. 69
5.3. Comparison of (a) common IOT and (b) SYMBEXNET. 69
5.4. Overview of SYMBEXNET SIOT system architecture and its steps for checking interoperability for implementations of IUT-1 and IUT-2 the same protocol specification. 71
5.5. Approach for generating interoperability test packets for protocol X. 72
5.6. A configuration for cross replay in SIOT. 73
5.7. Packet stream and stream identifiers. 75
5.8. An IOT model (left) and decision criteria (right) used in SIOT. 76

6.1. Number of completed paths and generated test packets with various symbolic execution timeout values for `Bonjour` (a and c) and `udhcp` (b and d), while progressively marking more fields as symbolic. 84
6.2. Number of generated test packet comparisons for Zeroconf and DHCP. 85
6.3. Number of generated test packets for pairs of fields in Zeroconf (subset of all combinations for each daemon) 86
6.4. Number of generated test packets for pairs of fields in DHCP (subset of all combinations for each daemon) 87
6.5. Code fragment from the `Bonjour` daemon implementation that sends UDP response packets. 89
6.6. Code fragment from the `main` function in `udhcp`. 94
6.7. Examples of derived interoperability rules for Zeroconf and DHCP. 96
6.8. Venn diagrams illustrating the results of checking the generated response packets for Zeroconf and DHCP. 97

A.1. Verification and simulation results for two deployed eNBs 125

CHAPTER 1

Introduction

IMPLEMENTATIONS of network protocols commonly used today, such as DNS [Pau87a], Zeroconf [SC05] and DHCP [Dro97] are often prone to implementation errors [ME04, SMCP11]. The complexity of these network protocols makes errors difficult to detect. Ambiguities in protocol specifications can cause different interpretations by developers, leading to bugs and interoperability errors in the corresponding implementations of network services, even for well-studied and mature protocols. Such flaws are often hard to detect because they may only be triggered by complex sequences of events that occur only after long execution as part of a production network [LGW+08]. For example, DNS server implementations that are vulnerable to DNS cache poisoning attacks [Dan08] are difficult to detect because the vulnerability only exhibits itself in specific scenarios. Unlike other software errors that can be detected before deployment, the impact of the vulnerabilities can be severe and the cost of fixing them is high. Although a large body of work has focused on techniques for finding software errors, these techniques have significant weaknesses because (i) many bugs manifest themselves only after prolonged operation; (ii) the state space of complex protocol implementations is large; and (iii) detection often requires additional information from protocol specifications about correct behaviour.

1.1 Motivation

In order to understand the problems of network protocol implementations and motivate research toward new techniques for detecting implementation errors, it is necessary to consider real examples that show how ambiguities in protocol specifications can cause various flaws and interoperability problems. In the following, we introduce two examples that provide a justification for our research throughout the rest of the thesis.

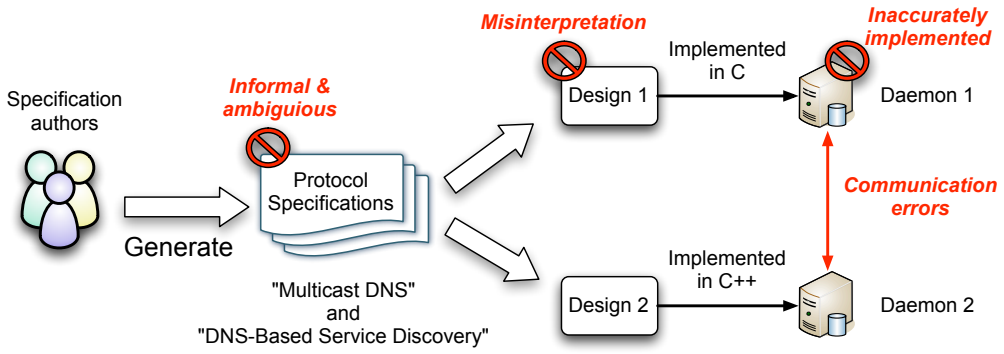


Figure 1.1: Typical network service development, from protocol specification to implementation

1.1.1 Where do protocol implementation errors come from?

Network protocols are typically described in specifications that contain the information required to produce a protocol implementation. Protocol specifications are usually written in informal languages, and therefore may be vague, incomplete or fail to address important properties of the protocol implementation. These specifications are referred to by multiple manufacturers and lead to different protocol implementations. A misinterpretation of a specification by one of manufacturer can cause implementation errors. Many errors are not detected until the service is in real use. Therefore, there is a strong need for developing a method to minimise the gap between protocol specifications and their implementations.

Figure 1.1 shows the typical development process of a network service in which protocol specifications are referenced by manufacturers to provide a network protocol implementation to end users. Let us consider *Zeroconf* [CK10] network, a service discovery protocol. The protocol is described in several standardised documents such as the "Multicast DNS" [CK10] and "DNS-Based Service Discovery" [Stu10] specifications. Then specifications are used by multiple vendors to create different implementations such as `Bonjour` [App] by Apple and `Avahi` [Ava] by the Avahi Project. Different interpretations of requirements in the specifications can cause errors, which may interfere with the proper communications between Zeroconf implementations.

1.1.2 What effect can protocol implementation errors have?

In order to avoid the complex task of configuring all parameters of a network service, most network-capable devices and services available on the market today support features that allow them to configure their parameters automatically. For example, devices and services follow procedures to select a unique service name and an Internet Protocol (IP) address in the network so that conflicts do not occur [CK10].

Figure 1.2 shows a common office environment with networked devices that can automatically con-

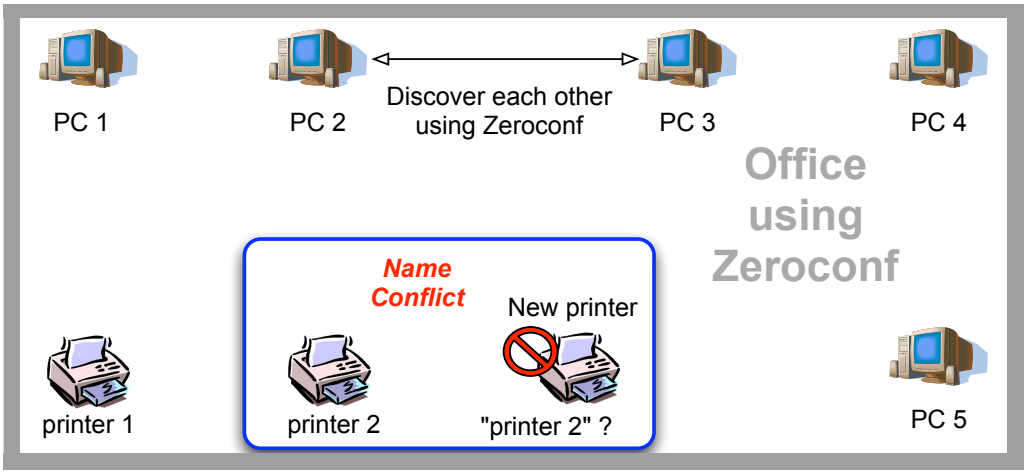


Figure 1.2: An example of network implementation errors occurring in a modern office environment

figure their network parameters using Zeroconf. According to the Zeroconf specification [CK10], any network device that selects a conflict name must immediately reset its name with a new name and restart its registration process. Here, let us assume that a new printer, which does not fully comply with the required procedure for resolving a conflict of devices having the same name, is installed in the office with the name of "printer 2". The new printer tries to resolve the conflict but fails. It may incorrectly select the same name or not manage to register itself on the network.

1.2 Current Techniques for Checking Protocol Implementations

Testing and *verification* can both be used to enhance the correctness of network protocol implementations [Ray82, Smi96]. Testing, which is a commonly used technique, involves generating test cases, which are used to check the protocol implementation. The two most common testing methods of network protocol implementations are *conformance testing* and *interoperability testing* [APRS93]: conformance testing determines whether an implementation behaves as described in the protocol specification, while interoperability testing is concerned with whether two implementations can interoperate as intended. A common way of doing these tests is to generate test input packets either manually or via random testing [MRW03]. Unfortunately, as the complexity of network services continues to increase, these methods become less effective; for example, randomly generated test packets often cannot explore source code efficiently and thus miss rare errors that are hard to detect.

Unlike testing, *verification* can be used to show that the implementation complies with its protocol specifications. *Model checking* [CGL94] is an automatic formal verification technique for the analysis of systems. It can be used to detect certain properties such as deadlock in concurrent systems, invariants and user defined assertions in the code. Model checking exhaustively checks programs,

but it is limited by the size of the state space that it can handle [SMP10]. *Static analysis* [WD01] is another popular technique that can be used to detect flaws in network protocol implementations. While static analysis benefits from high code coverage [WD01, ULF08], it is often too imprecise to guarantee properties that depend on accurate information about execution state. Although there has been much research on formal verification of network protocols [RA00, ISK06], such approaches cannot guarantee the correctness of the actual implementation because they check an abstract model of the protocol rather than its implementation.

Symbolic execution [Kin76] is a popular technique for generating high-coverage test cases through exhaustively exploring all execution paths of a system. It combines the strengths of both testing and verification. The core idea behind symbolic execution is to use *symbolic variables*, which are initially allowed to have any value instead of using concrete data. Program variables are represented as symbolic expressions. Symbolic execution explores a large number of potential execution paths and generates test data for all traversed paths. Thus, it provides developers with high quality test cases generated after exhaustive exploration. Symbolic execution has been successfully used to find bugs in a broad range of applications ranging from libraries to operating systems [CDE08, CGP+06, GKS05, GLM08a]. Symbolic execution has been used to check network protocol implementations [CGP+06, SMCP11], but prior work (1) considered only single input packets and (2) focused on generic errors, ignoring protocol semantics.

Consider the following concrete example. In a `dhcpd` daemon, each connection from a client has internally an associated state machine that handles input packets from the client and generates required responses. Symbolic execution on an input packet from the client can generate the possible input packets necessary to explore various states of that machine. Therefore, the generated input packets can be used to check whether the daemon is correctly implemented to process the input packets that it can receive. After receiving an input packet, the daemon expects to exchange more packets in an exact order to provide specific functionality. For example, the `dhcpd` daemon may assign an IP address to the client only after it has sent a packet offering an available IP address followed by having received a packet that requested the offered IP address. A single generated input packet using symbolic execution cannot effectively check whether multiple packets are correctly handled as specified in the specification.

1.3 Research Statement

The focus of this thesis is to check the correctness of network protocol implementations. It proposes a novel effective technique designed to check complex network protocol implementations against their specifications and to discover various types of implementation flaws. As a result, the technique enables developers to reduce the gap between specifications and implementations of network protocols.

To achieve this, we overcome three challenges: (1) due to the size of complex network protocol implementations, symbolic execution must explore a potentially exponential number of paths in the source code — an effect called *path explosion* [CDE08]. The total amount of state and the number of constraints grow during symbolic execution, and this limits the size of programs that can be executed symbolically; (2) by default, symbolic execution can only detect low-level programming errors such as null pointer references. In order to detect deviations from protocol specification, we need a mechanism that can detect violations in observed input and output packets; and (3) performing symbolic execution on a single input packet generates test packets that are inadequate for exploring the deep paths reached only after handling exchanged multiple packets.

In this thesis, we introduce a technique that automatically checks a network protocol implementation against its specification and discovers various types of errors that are hard to detect manually. As input, our technique takes the C source code of a network implementation and a set of rules derived manually from the protocol specifications, which define correctness and security violations. Developers specify a set of rules in a high-level packet description language that states invalid patterns in the sequence of packets exchanged between a client and a server. It then uses an expressive automata-based approach for the detection of rule violations.

Using symbolic execution, our technique generates an exhaustive set of input packets that achieve broad and deep exploration of the program state space. We execute the implementation on these test packets and check whether the implementation correctly handles them according to the packet rules. If an implementation is supposed to exchange multiple packets with a client in order to reach a certain program state, our technique repeatedly performs symbolic execution with additional packets. The output packets in response to input packets show externally visible behaviour of the implementation. Considering the observed behaviour according to the given set of rules enables us to check the network protocol implementation.

Furthermore, our technique uses the generated test input packets to test interoperability between different implementations of the same network protocol specification. Once a set of test input packets are generated from a network implementation using these test packets, they are executed against all available implementations. Analysing the observed output packets from these implementations explores the interoperability between these implementations. Since they implement the same protocol specification and are tested under the same conditions with the same test input packets, they are meant to behave in the same way, i.e. the response packets must be consistent.

We developed a system based on this technique, SYMBEXNET, that checks network protocol implementations, and empirically evaluated it with multiple network protocol implementations. We are able to generate high-coverage test input packets avoiding the path explosion problem and construct concrete input packets that can detect low-level errors leading to crashes. We also find hard to detect errors that lead to incorrect protocol behaviour, such as generating unintended response packets for test inputs, by using the rules derived from the protocol specifications of the target

implementations. Our experiments reveal that multiple implementations of the same protocol can behave differently resulting in interoperability problems.

1.4 Summary of Contributions

The main contributions of this thesis are as follows:

1. A novel checking technique that uses symbolic execution combined with rule-based packet analysis to generate automatically high-coverage test packets from the source code of network protocol implementations in order to detect various errors in these implementations.
2. Two enhancements to symbolic execution so that it can explore efficiently various network protocol implementations avoiding the path explosion problem. The enhancements provide both broad and deep exploration of the state space resulting in high source code coverage of the network protocol implementations.
3. A new interoperability testing technique that generates test packets to check interoperability between multiple implementations of the same protocol using symbolic execution. The behaviour of these implementations for each test packet is compared to verify that they are interoperable with each other.
4. Our experience implementing this approach in a system, SYMBEXNET, together with an evaluation that shows various real-world flaws in multiple implementations of two protocols: Zeroconf and DHCP.

1.5 Thesis Outline

The rest of the thesis is organised as follows.

Chapter 2 provides some preliminary definitions that are used throughout the thesis and the background that is required to understand our work. In order to have a better understanding of the benefits and limitations of formal network verification, it also includes a case study applying model checking to a network protocol managing configuration parameters.

Chapter 3 describes an overview of the proposed SYMBEXNET architecture. It also provides a description of how SYMBEXNET explores the state space of a given network protocol implementation and generates high quality test input packets using symbolic execution. It introduces two methods of performing symbolic execution, which can generate test input packets achieving broad and deep exploration of the program state space. For checking network protocol implementations, deciding which parts of input packets to mark as symbolic is an important consideration in order to

generate effective test packets. The chapter proposes a way to avoid path explosion by introducing a symbolic marking strategy, which marks individual packet fields as symbolic instead of an entire input packet.

Chapter 4 describes verifiable specification that can be derived from network protocol specifications. A packet rule description language that allows the verification of the externally visible behaviour is also introduced.

Chapter 5 presents a method for checking the interoperability of multiple network implementations of the same protocol specification. It shows how test packets generated using symbolic execution can be used for interoperability testing. The chapter also describes a method for the detection of interoperability problems through an extension of our rule description language.

Chapter 6 evaluates the SYMBEXNET approach. After describing the implementations details of SYMBEXNET and the experimental set-up for testing, it discusses the application of SYMBEXNET to two standard network protocols, Zeroconf and DHCP. It reports the results from checking multiple available implementations of these two protocols.

Finally, Chapter 7 provides discussion of future work and concludes.

CHAPTER 2

Background

THE purpose of this chapter is to provide definitions and the necessary background to understand the concepts that are used in this thesis. Formal verification and testing are widely used techniques to improve the dependability of network protocol implementations. Although they can help detect various network bugs, they suffer from their own limitations. Symbolic execution can be seen as a technique that takes advantages of formal verification and testing in order to generate high quality test cases that can effectively check software implementations.

This chapter is structured as follows: first, the definitions of network protocol, specification and implementation are provided followed by examples of network protocols in Section 2.1. After that, various techniques used to verify network protocols are described in Section 2.2. A case study applying model checking to a network protocol managing configuration parameters is showed in Section 2.3, discussing the benefits and limitations of using formal verification techniques to verify network protocols. The chapter finishes with Section 2.4 that discusses different kinds of testing techniques for checking network protocols. In particular, Section 2.4.2 describes the details of symbolic execution in terms of test case generation.

2.1 Network Protocols

A network can be defined as a collection of entities interconnected by communication technologies that enable the exchange of information [Tan02]. The communicating entities require an agreement to exchange information and such agreements are called network protocols. Therefore, throughout this thesis, we will use the following definition for the term "network protocol":

Definition 2.1 (Network protocol)**.** *A network protocol is an agreement between two or more network entities to communicate by exchanging information over a network.*

A network protocol consists of a set of network entities that play a specific role in the protocol and defines a set of rules which govern the exchange of messages through interactions of the par-

ticipants over the interface between them [For03]. Network entities use well-defined formats for exchanging messages, and each message contains information that leads to a defined response of the receiver.

Usually, a message is divided into small pieces, called packets. Although in the literature there has been several different definitions of a *packet*, here the term can be defined as follows [For03]:

Definition 2.2 (Packet)**.** *A packet is a sequence of binary octets of data exchanged over a connection between network entities.*

A packet comprises of two main parts: the header and the payload. The header usually contains network layer protocol data carried by the packet; for instance, in the case of an IP packet, the header includes the length of the packet, a sequence number, a protocol type and the source and destination IP addresses. The payload, which is also called the body, carries the actual user data or data from upper layer protocols that the packet is delivering to the destination. For example, in a DNS packet, the header information includes the source and destination IP address and the payload contains DNS messages, such as `QNAME` and `QTYPE`. Different network protocols use different packet formats.

Network protocols can be roughly categorised into two groups [Ste93]: *stateless* and *stateful* protocols. In stateless protocols, an implementation simply handles a request independently from any previous request so that the communication occurs in independent pairs of requests and responses. On the other hand, in stateful protocols, the implementation maintains the connection with communicating network entities and exchanges a series of packets until the entire transaction is over, meaning that the implementation is aware of previous requests.

For example, in the Hypertext Transfer Protocol (HTTP/1.0) [BLFF96], a web server generates a response for a request from a client and does not maintain state about the client. Whereas, in the Simple Mail Transfer Protocol (SMTP) [Kle01], a mail server handles a sequence of messages from the client. In this type of protocol, a server expects a specific packet sequence and executes certain functions only when it receives an expected packet from the client.

Consider two network entities at one layer, A and B, that are connected by a channel. In order to complete a certain task, such as finding a printer, A and B must exchange packets through the established channel based on rules provided by the network protocol between them. Each packet delivers specific information, and the task is completed after exchanging a sequence of packets in an exact order. The series of exchanged packets between these two instances is called *a packet stream*, and it can be defined as follows:

Definition 2.3 (Packet Stream)**.** *A packet stream (or a sequence of packets) on a connection between network entities communicating with each other is a series of consecutive packets on that connection.*

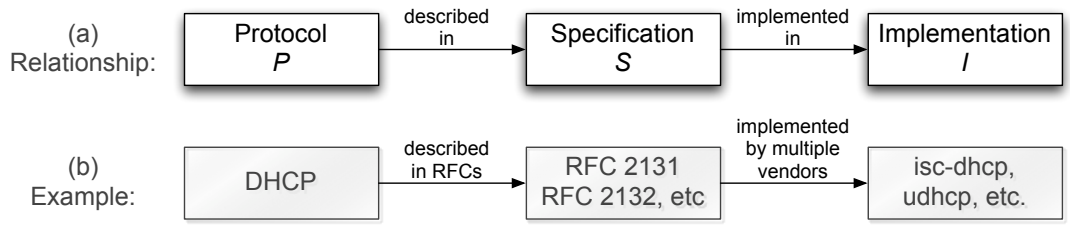


Figure 2.1: The relationship between protocol, specification and implementation

When a network protocol is designed, all the information regarding methods, behaviour and packet formats are described in documents, which form the *protocol specification*, to be referenced by developers of a *protocol implementation*. Figure 2.1 illustrates the relationship between protocol, specification and implementation. When the requirements of a protocol P are specified, they are described in a protocol specification S and the specification is implemented in I. For example, the Dynamic Host Configuration Protocol (DHCP) is a network configuration protocol for devices on TCP/IP networks and described in several protocol specifications [Dro97, AD97]. The specifications are implemented in several products such as `isc-dhcp` [Int] and `udhcp` [DR02].

In the domain of network communication protocols, a protocol specification is a description of the intended and required behaviour of its participants [BS80, For03]. The nature of the network communication, i.e. the exchange of actual data and the intended behaviors, are usually defined by a standard specification such as Request For Comments (RFC). The goal of protocol specifications is to furnish a set of logical rules for exchanging information between network entities communicating with each other [Yua88]. A protocol specification is defined by several pieces of information, such as a general description of the services that it provides, a list of the types and formats of messages exchanged between the entities and rules governing the reaction of each entity to messages from other entities.

Protocol specifications are usually implemented into a software program by protocol participants, which communicate with one another [BS80]. In general, a protocol specification is assumed a complete and self-contained document, thus the specification must be implemented exactly as expected and intended. As individual vendors implement the same specification, there typically exist multiple implementations providing the same functions to users. For the purpose of this thesis, we will use the following definition of protocol implementation.

Definition 2.4 (Protocol implementation)**.** *A protocol implementation is a network enabled software developed based on requirements specified in a standard protocol specification.*

In Unix and other operating systems, there are special system applications that execute in the background, which are called "*network daemons*". Network daemons are often started at system boot time and await network requests for the functions that they provide. Many network daemons are implemented based on standard protocol specifications.

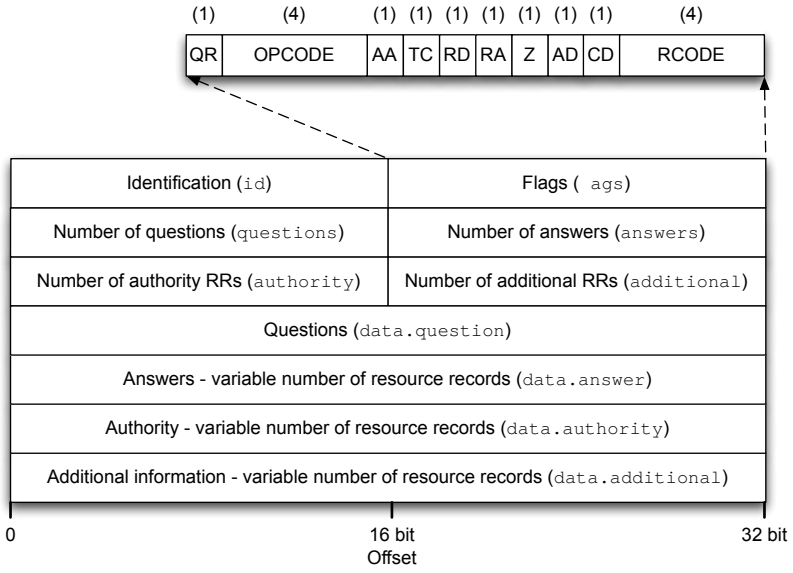


Figure 2.2: Message format for Zeroconf

We describe two examples of network protocols; *Zeroconf* and *DHCP*. Both are widely used and implemented by many different vendors. They are used throughout the thesis to demonstrate the various problems posed and addressed by our approach.

Example 1 — Zeroconf protocol. Zeroconf [CK10] is a network discovery protocol that enables devices on an IP network to configure themselves and their services automatically and be discovered without manual intervention. Zeroconf is a serverless implementation of the DNS naming function built on top of standard DNS and uses the same format of a DNS packet.

Figure 2.2 shows the message format for DNS. The format describes different types of DNS messages, which are processed based on the information of each field. The format has a 12-byte fixed-length header in addition to a variable data part reserved for question, answer, authority and additional DNS information. In DNS packets, the fields deliver several key control flags. For example, a 16-bit identification field in a query packet is copied to the response packet by the daemon. It is then used by a device that sends the query to identify the corresponding response packet.

The Zeroconf protocol is defined as part of two RFC specifications: multicast DNS (MDNS) [CK10] and DNS-based Service Discovery (DNS-SD) [Stu10]. The MDNS RFC covers basic behaviour such as probing, announcements and responses of Zeroconf; the DNS-SD RFC describes the structure of resource records and service discovery mechanisms.

Zeroconf supports service discovery to allow applications to find a particular service name or all instances of a given service type. When the Zeroconf daemon receives a DNS query packet for a given type or name, it responds with any services matching the query. Figure 2.3 illustrates

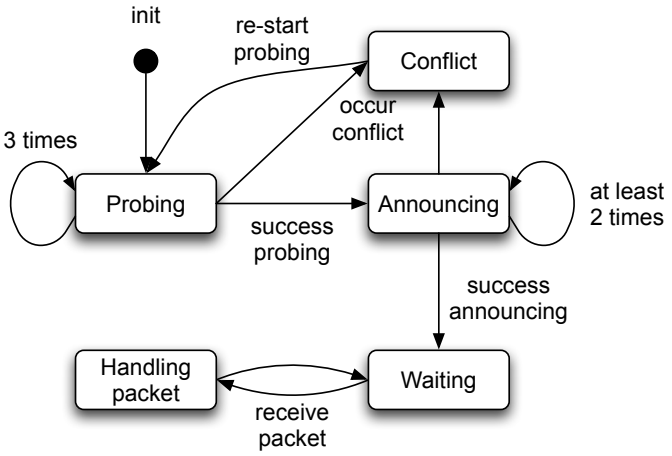


Figure 2.3: Simplified state transition diagram for Zeroconf

the finite state machine that specifies how Zeroconf handles service registration and discovery. In Zeroconf, a new network service, such as a file server or printer, is added as follows. In order to register a new network service, a device selects a service instance name. It then sends a DNS packet registering a new service to its local Zeroconf daemon. This causes the Zeroconf daemon to send out a broadcasting DNS packet three times to the network in order to probe if the service name already exists. If there is no response, the daemon starts to send a broadcast DNS packet announcing the new service at least two times. After the successful announcement, the daemon waits for request packets and replies if it receives a query packet requesting a service running on the daemon.

Example 2 — DHCP. In this thesis, we also use DHCP (Dynamic Host Configuration Protocol) [Dro97], a standard mechanism to obtain configuration parameters. Network devices that are connected to IP networks must be configured before they can communicate with other hosts. DHCP allows a server to assign network configuration parameters dynamically, especially the IP address, to clients. DHCP has eight types of packets, such as `DHCPDISCOVERY` and `DHCPOFFER`. They share the same format but can be distinguished based the values of certain fields in the packets.

DHCP is standardised in the RFC 2131 Dynamic Host Configuration Protocol [Dro97]. The DHCP-RFC describes the behaviour of a dynamic configuration service framework that passes configuration information to hosts on a TCP/IP network. The RFC includes the description of client-server interaction for allocating a network address and the structure of message types used in the protocol. Since the protocol is based on the Bootstrap Protocol (BOOTP) [CG85], it also describes methods maintaining the compatibility between DHCP and BOOTP.

Figure 2.4 shows the format of a DHCP packet. The first 12-bytes of the DHCP packet are used to deliver basic information about messages and client types, such as hardware type and address

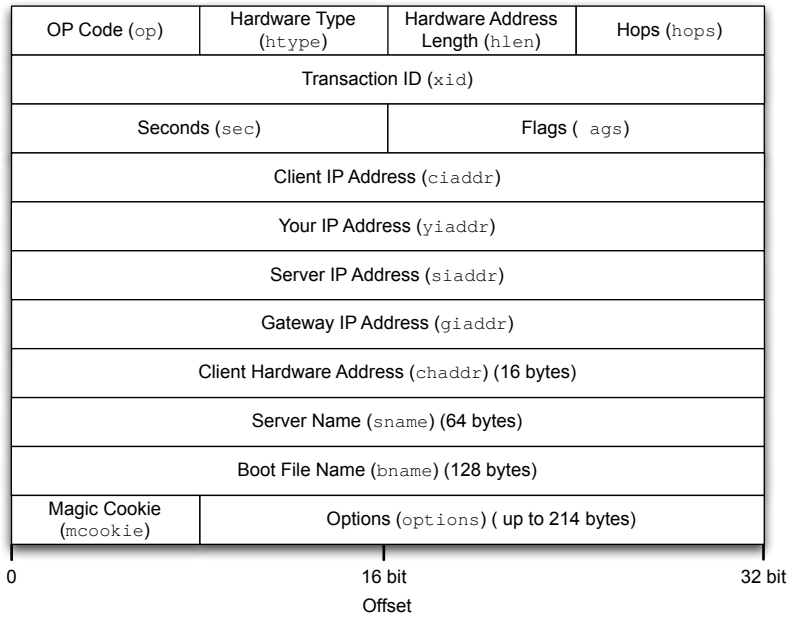


Figure 2.4: Message format for DHCP

length. After that, the format has various fields for IP addresses that are needed to provide an available IP address to clients. For example, the address to be assigned to a client is stored in the assigned IP address field (`yiaddr`). Options contains a variable list of data records passing configuration information to clients.

Figure 2.5 illustrates a simplified state transition diagram of a DHCP server handling an IP address allocation request from a client. When a DHCP-enabled client is connected to the network, the client sends a broadcast query packet (`DISCOVERY`) requesting an IP address from a DHCP server. Any DHCP server that receives the query may send a packet (`OFFER`) offering an available IP address. The client responds to the packet by sending a broadcast response packet (`REQUEST`)

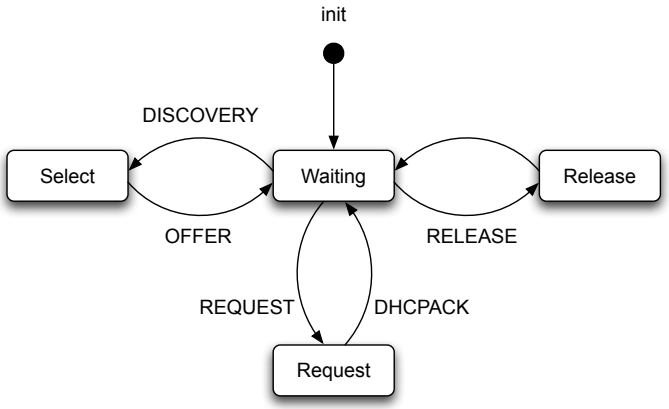


Figure 2.5: Simplified state transition diagram for DHCP

accepting the offered IP address. The server responds to the request packet with an acknowledgement packet (`DHCPACK`), thus completing the assignment process. Before leaving the network, the client terminates the leased IP address by sending a packet that requests to release the address to the DHCP server (`RELEASE`). The server then returns the client's IP address to the available address pool.

2.2 Network Verification

As explained in the previous section, in order to provide the intended functions, network entities communicating with each other must exchange a series of packets as described in the protocol specification. Usually system verification is understood as the process of checking that an object, i.e. design or implementation, meets its specification. In the same way, network protocols can be verified through checking a protocol design or implementation against its specification.

Software programs implementing network protocols have their specific characteristics that are different from other types of software and these characteristics make it difficult to apply verification techniques. For example, since a network protocol defines rules for communication between network entities, interactions with other entities needs to be modeled and verified. Constructing a protocol model within an environment often results in a model having complex states, thereby making it difficult and expensive to explore.

Although the verification of a protocol design can detect errors at an early stage, the implementation may still contain errors because of human mistakes, incorrect implementations and wrong interpretations of a specification. Therefore, verifying a protocol implementation is important to guarantee that the protocol provides the desired behaviour specified in its protocol specification. Prior verification research work [SMP10, Zav08, ISK06] has focused on the design rather than the implementation. We focus on implementation verification to examine the correctness of implementations, i.e. whether they behave as specified in their protocol specification without errors.

The purpose of this section is to provide a general overview of verification methods and related research. After presenting the main goals of network verification in the next section, we discuss various verification approaches in Section 2.2.2.

2.2.1 Goals of Network Verification

In this section, we describe the goals of network verification that have been used in verifying network protocols. *Liveness* and *safety* are the main properties that need to be verified in order to guarantee the correctness of a given implementation. *Liveness* is a property that a correct function is provided during execution, i.e. a program eventually enters a desired state, while *safety* is a

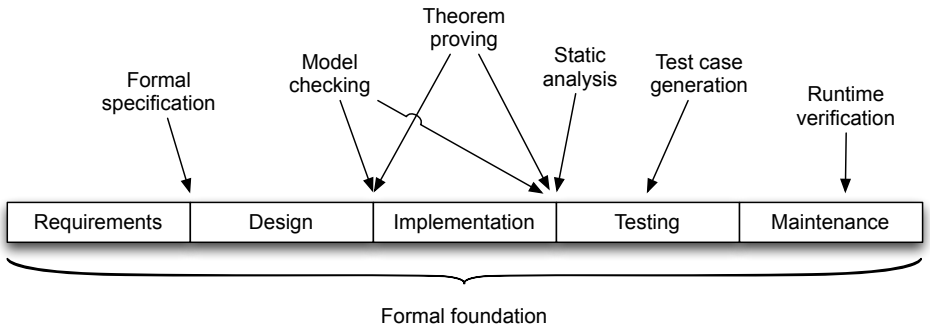


Figure 2.6: The life cycle of network protocol software development and applicable formal verification techniques.

property that some unintended function is not provided during execution, i.e. a program never enters an undesired state.

Since we try to verify protocol implementations that are able to send and receive network packets, we define the verification goal for these two properties as follows:

Definition 2.5 (Liveness property)**.** *A network protocol implementation that receives a packet eventually enters a desirable state such as correct handling of input packets.*

Definition 2.6 (Safety property)**.** *A network protocol implementation that receives a packet never enters an undesired state such as unexpected termination.*

2.2.2 Verification Approaches

The idea of using a method verifying the correctness of programs via a formal proof on an abstract model, called *formal verification*, is well explored. Examples include the processing of input packets in a protocol implementation [SGE00], the correct control of information flow in a routing protocol [Fea04, FB05, ISK06, AW01, WBLS09], the reliable data transfer in a transport protocol [DKNP06, DKN+10], the selecting of parameters in a configuration protocol [Fea04] and the handling of incorrect packets [Sch81, BOG02]. Researchers have explored various approaches for the verification of network protocols. Formal verification techniques provide a proof that an implementation satisfies its verification goals. In the following, we show when network verification techniques can be used in the development of network protocols.

Figure 2.6 shows how various formal verification techniques are used as part of the protocol development, ranging from specification to maintenance. *Automated Theorem Proving* considers the relationship between a specification and its implementation as a theorem in logic. It is therefore applied during the design and implementation phases. In *Model Checking*, the specification is described in the form of a logic formula, and the truth is determined with regard to an abstract model provided by an implementation. Therefore, it can also be applied during the design and implementation phases. In the case of *Static Analysis*, the runtime behaviour of a protocol implementation

is predicted based on a direct analysis of the source code. Therefore, it usually occurs during the implementation phase. Research on network verification aims to increase the ability of formal verification techniques to be applied earlier (*Formal specification*), to cover the entire protocol development cycle (*Formal foundation*), or to verify dynamically the behaviour of a target system at runtime (*Runtime verification*). Next we provide explanations of each technique in detail.

Automated Theorem Proving

Automated Theorem Proving (ATP) [Sla74, CW96] deals with the development of programs that show that some statement is a logical consequence of a set of statements. In other words, ATP proves that an implementation satisfies a specification by mathematical reasoning. The relationship between a specification and its implementation is considered as a theorem in logic. The proof can use axioms and assumptions provided by the implementation. The increase in the number and types of theorem provers is evidence for the growing interest in theorem proving.

There are theorem provers that can be used to verify network protocols. *Isabelle* [Pau93] is a generic theorem prover, designed for interactive reasoning in a variety of formal theories. The main proof method of Isabelle is a higher-order version of resolution, based on higher-order unification. Isabelle has been used to formalise numerous theorems from mathematics and computer science and to prove the correctness of security protocols. *Simplify* [DNS05] is an automatic theorem prover and combines decision procedures for several important theories, and also employs a matcher to reason about quantifiers. It uses two techniques, error context reporting and error localisation, for helping the user to determine the reason that a conjecture is false. However, Simplify lacks support for important programming language constructs such as pointers, structures and unions. *Prototype Verification System* (PVS) [ORS92] is a comprehensive interactive tool for specification and verification, combining an expressive specification language with an integrated suite of tools for theorem proving and model checking. The specification language of PVS is a typed higher-order logic with a richly expressive type system with predicate subtypes and dependent types. It has been applied to a wide range of verification tasks, such as security algorithms, real-time and hybrid systems and distributed algorithms.

To verify network protocols, high order logic is used to reason about possible protocol executions. Due to certain limitations, such as a lack of automation (there is still a need for human intervention), theorem proving techniques are mainly used to describe protocols with formal languages or semantics, thus making the designed protocol specification verifiable [WBLS09,GS05]. In contrast to ATP, our technique provides an automated method to check the implementation.

Model Checking

Model Checking [CGL94, CW96] is a technique that explores all possible states of a target system in a systematic manner. Figure 2.7 shows an overview of system verification using model checking. In order to apply model checking to a protocol, two inputs are required: a verification model and

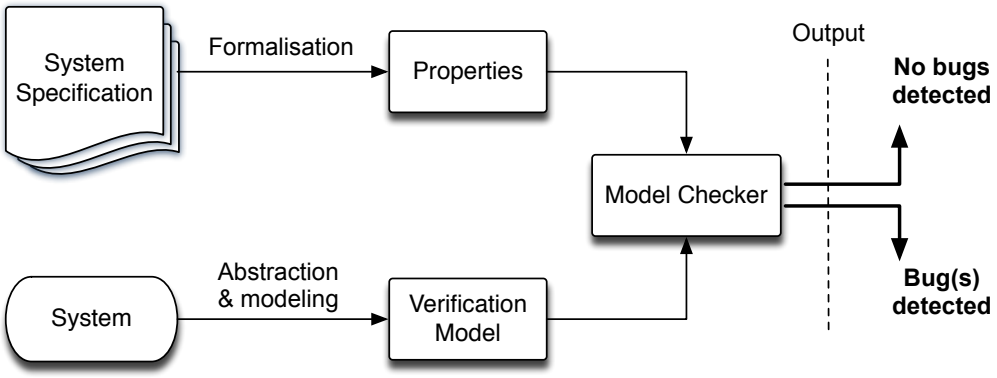


Figure 2.7: Conceptual view of system verification using model checking

a logical property. The verification model represents an abstraction of the given protocol and the property prescribes what the protocol should do and what it should not do. A model checker, the software tool that performs the model checking, examines all possible states to check whether they satisfy the given property. If a state violates the property, the model checker reports an error and provides a counterexample that indicates how the model could reach the undesired state.

For network protocol verification, the protocol specification is described in the form of a logic formula, and the truth is determined with regard to an abstract model provided by an implementation. An implementation is typically modeled as a finite-state machine that consists of nodes and edges representing states of a system and possible transitions, respectively. Model checking then explores all possible states and checks the logical correctness of the implementation.

Although model checking has been successfully applied to many network protocols [ISK06], it also has several weaknesses. For example, it suffers from the state-space explosion problem [SMP10]. Although several extensions have tried to address this issue by using partial order reduction [FG05], program slicing model reduction [DHH$^+$06], decomposition into components and/or sub-layers [AL03], an assume-guarantee approach [FQ03], it still remains a difficult issue that needs to be tackled. In addition, protocol specifications need to be formalised and protocol models need to be written manually.

Static Analysis

Static Analysis [WD01, AHM$^+$08] has been used for decades to improve software quality and to detect bugs. Static analysis techniques work on the program source code and take the specification of the property to be checked as input. This specification is written in a given specification language. Since static analysis is carried out on the source code, it is possible to check correctness immediately after the code was written. Static analysis also systematically explores all possible execution paths in a program at compile time and thus achieves good source code coverage.

In the domain of network verification, *static analysis* is used to predict (detect) errors from

an implementation of network protocols without execution [Fea04, XZM+05]. Depending on the technique employed, static analysis can detect various flaws from possible programming errors [BPS00, EGHT94] to user-defined properties such as the reachability between end hosts [Hol02, XZM+05]. Once an implementation is analysed, the technique reports that the implementation is safe, i.e. the implementation satisfies the specified property or gives an error trace that violates that property. However, static analysis also has several drawbacks, for instance, a trade-off between scalability and precision, high false positive rates, etc. In comparison, our approach is more precise than static analysis because it does not abstract the implementation under test and thus does not suffer from the inherent imprecision of static analysis.

Formal Foundation

Another approach is to apply formal verification techniques during all the development phases of a system and provide a formal foundation of network protocols. In this approach, architectures or protocols are specified with formal languages or semantics, making the designed system correct by verification.

Karsten et al. [KKPB07] focus on formalising communication behaviour from a set of basic axioms to verify functional correctness of a protocol. Their framework can be used to analyse network protocols formally based on structural properties, and also to derive working prototype implementations of these protocols. They axiomatically specify basic interworking concepts such as the deliverability of messages and show that their approach can assist in the proofs of correctness of communication protocols, such as TCP over NAT, DNS and Hierachical Mobile IP.

The work on *metarouting* [GS05] addresses the issue that there is a shortage of routing protocols that meet the needs of network engineers. The work defines routing protocols using a high-level and declarative language and enforces a clean separation of protocol mechanisms from routing policy. The proposed solution allows for the construction of routing algebras and has correctness conditions that can be derived automatically for a new routing algebra.

Anduo et al. [WBLS09] propose the design and implementation of a *declarative network verifier (DNV)*. In their work, network protocol specifications are expressed using a declarative logic-based query language, which can be mapped automatically to logical axioms. These axioms are then directly used in theorem provers to validate protocol correctness. Since their specification can be executed as an implementation, the approach bridges specification, verification and implementation. However, the method can only be applied to the implementations of declarative networking protocols.

Melange proposed by Madhavapeddy et al. [MHD+07] introduces a framework that comprises a domain specific language, which is called the Meta Packet Language (MPL), a compiler and a suite of libraries. MPL is used to describe the format of network protocols. The specification of a protocol in MPL are then used by the compiler to generate code and required interfaces for

the protocol. Their approach, since based on type-safe language, helps eliminate many errors and provides additional benefits, such as automatic garbage collection. However, their framework cannot detect implementation flaws caused by misunderstood requirements.

Although formal foundation provides a unified framework, it is not yet fully mature and only applicable to specific network domains such as declarative networking.

Runtime verification

This is a verification technique that adopts mathematical language to formalise requirement specifications, such as temporal behaviour, events or actions. According to these formal properties, a verifier that continually performs monitoring and analysis rigorously checks them against target implementations at runtime. If any violation is found, the verifier identifies the error.

For example, monitoring-oriented programming (MOP) [CR05] is a runtime formal model checking framework for software systems. The MOP framework automatically generates observers monitoring the system for given formal properties. The monitoring observers can discover if the specified properties are violated, which is hard to discover by ordinary testing. Based on automatic code generation and program instrumentation, MOP provides support for applying runtime monitoring and recovery mechanisms in software development to improve reliability. CrystalBall [YKKK09] is a runtime model checker that can predict and prevent inconsistencies in deployed distributed systems. CrystalBall can periodically collect snapshots of neighbouring hosts and construct a global state of a distributed system. If bugs or inconsistent states are found, it steers the execution away from them.

Continuous runtime verification can improve overall robustness of a system and find bugs that can only be detected after prolonged operation. However, overall performance overheads are difficult to avoid in these approaches [SSRL07, SLS05, YKKK09].

2.3 An Example of Model Checking

As described, *formal verification* techniques based on model-checking have been successfully used for the verification of hardware designs, communication protocols and safety-critical systems. Such approaches explore the entire state space of a system to ensure that there are no violations of correctness properties. For network operators, formal verification can provide correctness assurances for autonomous networks throughout the design and operation. Therefore, we describe a case study that applies model checking to a network management protocol in order to understand the advantages and disadvantages of formal verification for network protocol implementations.

In this section, we performs model checking on a specific management protocol, namely the self-configuration protocol of cellular base stations. We show how it can benefit from model checking

and describe the problems that make it difficult to apply formal verification techniques to real-world network protocol implementations [SMP10].

2.3.1 Self-Configuration of Base Stations

As a case study, we describe a protocol to configure network parameters automatically in a network. We show how formal verification techniques can be adopted to analyse the protocol. A *self-organising network* (SON) is an autonomous network specified by the 3GPP that configures, optimises, heals and protects itself to help operators reduce operational expenses [3GP08]. In a SON, *evolved Node B* (eNBs) and *Home Node B* (HNBs)—also called Femtocells or home base stations—are cellular base stations that cover macro and small (home-range) areas, respectively. A *cell* is the basic geographic unit covered by a base station. A cell has a unique *physical-layer cell identifier* (PCI) as a basic configuration parameter that identifies the cell.

PCI assignment protocol. In a SON, the PCI is an essential configuration parameter for a cell and corresponds to a unique combination of one orthogonal and one pseudo-random sequence for data encoding. Only 504 unique PCIs are supported because of compatibility with legacy base stations [3GP08]. When a new base station is deployed, a PCI needs to be selected.

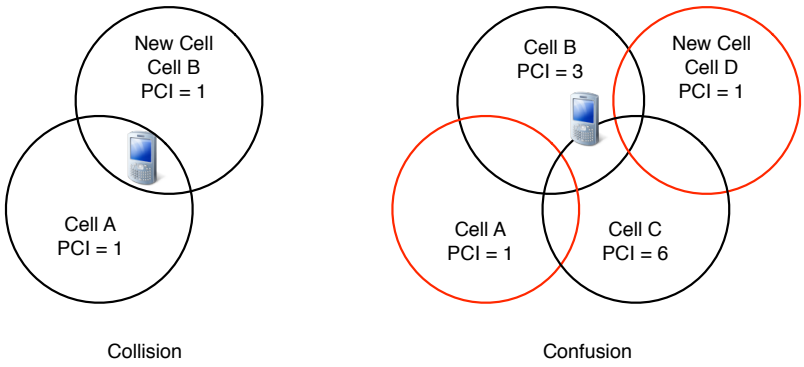


Figure 2.8: Examples of collision and confusion in PCI assignment

Figure 2.8 illustrates the problems of PCI collision and confusion. If the PCI of Cell B is equal to 1, there is a collision between Cells A and B because the PCI of Cell A is also 1. If the PCI of a new Cell D is chosen to be 1, it leads to confusion of Cells B and C with A. This would cause hand-over procedures from B to the new Cell D to fail. To achieve collision- and confusion-free assignments, the 3GPP proposed a PCI selection protocol with several optional extensions.

According to the SON specifications [3GP08], an automated PCI selection algorithm should fulfil the following two requirements regardless of deployment strategies:

- **collision-free:** a PCI should be unique in the area that the cell covers;

- **confusion-free:** a cell should not have neighbouring cells with identical PCIs.

To achieve collision- and confusion-free assignments, the 3GPP proposed a PCI selection algorithm that is divided into four main steps: (1) a base station tries to get a valid range of PCIs; (2) it performs neighbour discovery; (3) it exchanges a neighbour relation table that contains information about neighbours of neighbours with its neighbours; and (4) it selects a random PCI from the list of candidate PCIs.

Modelling and evaluation. To assess the satisfaction of the collision- and confusion-free properties, we adopt model-checking as a rigorous method to evaluate a PCI selection algorithm. We use the SPIN model-checker [Hol97] because it supports the verification of asynchronous and distributed process system. It uses the *PROcess Meta LAnguage* (PROMELA) to describe the algorithm and its assignment policies. PROMELA can be translated into a C program for efficient verification using SPIN. It supports the specification of safety properties in *linear temporal logic* (LTL). The semantics of LTL provides temporal modal operators that can make statements about properties that are globally true or eventually true. This is sufficient to describe collision- and confusion-free PCI assignments.

We consider an abstract model of the PCI assignment problem. We only model the configuration procedure and do not consider failure of processes for simplicity. We adopt this simple model to illustrate the problem and investigate the feasibility of formal verification. We use SPIN to exhaustively search the state space for collision or confusion violations of the PCI selection algorithm described previously. Overall, both our verification and simulation results provide evidence that collisions and confusions cannot be avoided in the concurrent PCI selection, which is not addressed by the 3GPP specification. However, the scalability of model-checking becomes limited even in this simple scenario. Our detailed evaluation results are described in Appendix A.

2.3.2 Discussion

Our evaluation of verification and simulation has yielded several insights. Both verification and simulation are useful for detecting hidden design flaws and finding correctness violations. Simulation is easier to understand and perform. It scales better because it avoids the problem of state space explosion. However, simulation has lower state space coverage and only tests scenarios under predefined conditions, thus may not discover infrequently occurring problems. On the other hand, formal verification techniques are more rigorous when assessing required properties. However, modelling and verification of a complex network is challenging and requires a certain level of abstraction. The problem of state space explosion is the main obstacle when applying formal verification using model-checking. Even our simple deployment model with simplifying assumptions has only limited scalability.

Our comparison of simulation and verification using model-checking reveals their respective strengths. Verification is more rigorous when discovering incorrect behaviour but its limited scalability precludes its use in non-trivial deployment scenarios. Therefore, many developers use *protocol testing* as a mainstream method for checking network protocols. We discuss the details of testing techniques for network protocols in the rest section.

2.4 Network Testing

In the domain of network protocols, implementations are tested to ensure that they will work correctly as described in their specifications and are properly interworking with other entities as intended without any errors. Therefore, research on communication protocol testing has mainly focused on finding these errors. Since the generation of high quality test inputs is an important factor in finding bugs, many researchers on network testing have been trying to develop test case generation techniques. In this section, we will describe various techniques that are used to generate test inputs and check protocol implementations to find errors.

2.4.1 Testing Techniques

The goal of network testing is to achieve high code coverage, which means achieving as much statement and branch coverage of the target protocol implementation as possible. Test inputs with high code coverage improve the dependability of the target implementation.

Conformance and interoperability testing. *Conformance testing* is to check the conformance of the implementation of a protocol to its specifications. Since conformance testing is considered a necessary step before operation, it has been studied and formalised [DSU90, Tre99, BFN+05].

Different from conformance testing, in which the focus is to check an implementation against its protocol specification, interoperability testing checks if two or more implementations correctly implement a protocol specification necessary to ensure successful interaction before network services. In general, interoperability testing is the process of testing implementations from multiple vendors by interacting in such a manner as to exercise the network protocol under test.

Although both testing techniques aim to check different aspects of network protocol implementations, they both aim to check whether the implementation is fully conformant with a specification. Therefore, many testing techniques use protocol specifications as a main source of generating high quality test inputs. Typically, the specification is translated into a formal model, such as a finite state machine or use cases, and an algorithm is used to generate test cases from that model [FJJV97, NFLTJ06]. However, the problem with test cases generated from the specification

is that they often do not cover all possible execution paths of the implementation. In addition, expert knowledge of a specific protocol specification is required when the model is derived.

Random testing. Random testing [BM83] or fuzz testing [MFS90, FM00] is a simple testing technique, which uses randomly generated test cases to check whether a target program works as intended. It is one of the most commonly used techniques for network testing. The main advantages of this technique are its ease of implementation and the low overhead in choosing inputs. However, it is also well-known that random testing is not effective for detecting bugs that only occur when a given program receives specific inputs and they provide low code coverage [FM00]. Some fuzz testing techniques provide grammars to enable the testers to add protocol specific knowledge [God07]. Although fuzz testing techniques support effective bug finding methods, they also suffer from the limitations of random testing.

Grammar-based testing. Fuzz testing is limited to test programs dealing highly-structured inputs such as compilers and interpreters. Since these programs have multiple stages for handling complex inputs, such as lexing and parsing, fuzz testing struggles to explore a program beyond these early stages. Research on grammar-based testing has been practically used for such programs since the 1970s [Han70, SB99]. More recently, attention has turned towards applying the testing to network protocols [HK07, HWC+10]. Methods used to generate test inputs can be roughly classified into two categories: random [BM83, Ait02, FM00] and exhaustive [GLM08b, LS06] generation. The first uses randomly modifying well-formed input packets, while the second exhaustively generates all possible test inputs from a formal model. However, since test inputs are generated from simplified (or abstracted) formal specifications, not many new test inputs can be generated. In addition, by checking a program with the test inputs generated from an abstracted specification, it is easy to miss errors.

2.4.2 Symbolic Execution

To address the limitations of random testing techniques, several researchers have proposed automated test case generation techniques that improve code coverage. Test case generation is considered as a production of sets of specific input values that can be used for testing a program. The goal is to produce test cases that achieve high source coverage.

Symbolic execution is a popular technique for generating high-coverage test cases and finding implementation flaws. The core idea behind symbolic execution [Kin76] is to use *symbolic values* as input values, instead of actual data, and to represent values of program variables as symbolic expressions. Consequently, the output values computed by a program are a function of the symbolic input values, resulting in the creation of a symbolic execution tree. The tree is a representation of a

```
1  int handle_input (packet p) {

3      ... parsing input packet p ...

5      if (flags == 0x01)
6         return QUERY; /*query*/

8      if (flags == 0xf0)
9         abort(); /*error*/

11     return RESPONSE; /*response*/
12 }
```

Figure 2.9: Source code that handles a received input packet.

program, which identifies the decision points and the assignments associated with each conditional branch.

Symbolic execution maintains symbolic state. Two pieces of information are associated with each state: a symbolic map (*SM*) and a path condition (*PC*). The *SM* associates symbolic values with program variables, and the *PC* is a first order quantifier free boolean formula involving relations between input variables, which expresses the conditions necessary for reaching that state. During execution, a list of constraints on the inputs is accumulated to *PC*s by the set of symbolic representations of each condition predicate along the path. In order to explain the main concepts used in symbolic execution, we provide an example below.

Let us consider the code fragment in Figure 2.9, which processes a received input packet based on its `flags` value. It contains a statement `abort()` when the value of `flags` is equal to `0xf0`. The function returns `QUERY` when the `flags` value is `0x01` and, otherwise, returns `RESPONSE`. Symbolic execution starts from the first line of the source code and proceeds, branch by branch, to the end of the program.

Initially, the *PC* is `true` and `flags` has symbolic value `X`. At every conditional statement, the *PC* is updated with assumptions about the inputs to select between alternative paths. For example, after the execution of the first conditional statement in line 5, *PC'* is created, initialised to *PC* $\wedge$ `X` $\neq$ `0x01` and *PC* is updated to *PC* $\wedge$ `X` $=$ `0x01` ("if" branch). If the path condition becomes *false*, symbolic execution does not continue for that path and terminates. At the end of the symbolic execution of a path, the current *PC* is solved and the output variables are represented as the *test inputs*.

Figure 2.10 shows the corresponding symbolic execution tree. Initially, `flags` has a symbolic field value `x` that can be any value. At each branch point that is related to `flags`, the value for `flags` is

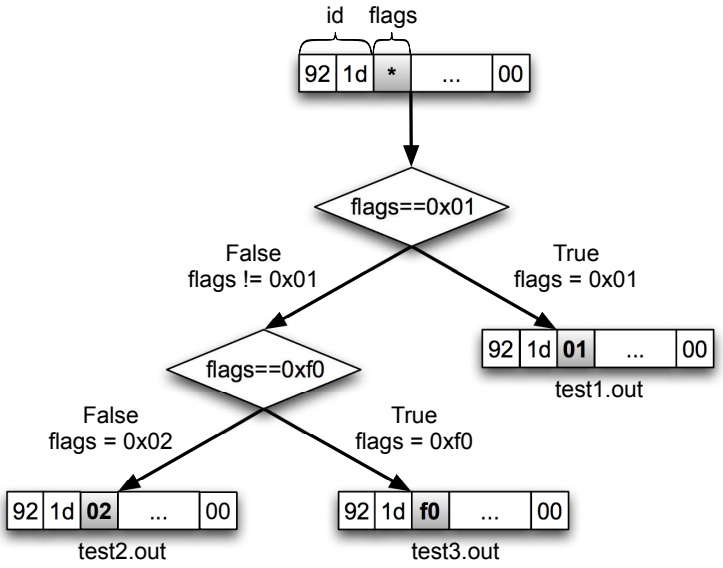


Figure 2.10: The corresponding symbolic execution tree to Figure 2.9

updated with assumptions about the symbolic field. For example, after the execution of the second `if` statement (line 7), both alternatives of the `if` statement update the `flags` value accordingly and one of them leads to the `abort()` error.

The idea has provided a broad and useful spectrum of tools to programmers for verifying the correctness of programs. In the beginning, symbolic execution was used to verify simple sequential programs [Kin76, BEL75, How77]. However, there were a number of challenges that had to be solved [Dil90]. Recently, symbolic execution has received renewed attention with increased computational power and new advanced algorithmic developments. For example, some proposals [ZC10, HBBR10] use symbolic execution in conjunction with static analysis to automatically synthesize executions to reproduce bugs that occurred in the field, without incurring the overhead of execution tracing. Symbolic execution is also used to compute a behavioural characterisation of program changes and check for logical differences and similarities between two versions of a program [PDEP08, CFF+06, PYRK11]. More recently, symbolic execution has evolved to provide the basis for energy-aware programming through platform-specific energy profiles [HEKSP11].

To date, symbolic execution has already been applied to check network protocol implementations. The input packets from the network can be used as symbolic values. Symbolically executing the implementation with these inputs generate test packets that result in a high degree of coverage [CGP+06, SMCP11]. However, prior work (1) typically considered a single packet, starting exploration from the initial program state, and (2) looked only at generic errors.

We use KLEE [CDE08], a symbolic execution tool for C programs capable of automatically generating high-coverage test cases to find low-level bugs. KLEE, at its core, is an interpreter for

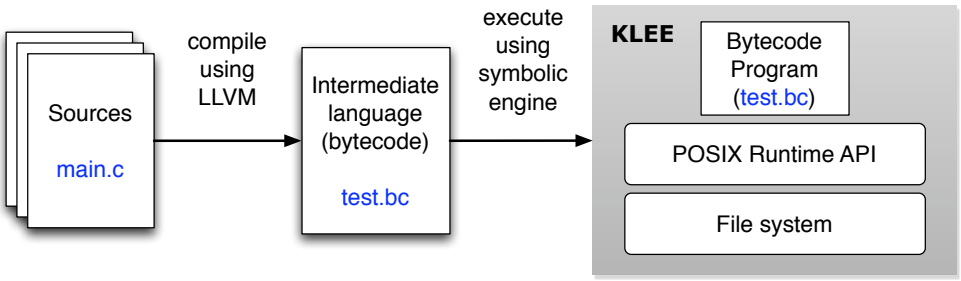


Figure 2.11: Overview of symbolic execution and the structure of KLEE.

an intermediate language (i.e. bytecode). Thus, as shown in Figure 2.11, KLEE requires the source code to be compiled into a bytecode (instead of a native executable) using the LLVM compiler [LA04]. KLEE then interprets the bytecode and functions as a hybrid between an operating system and an interpreter by providing environment models and redirecting system calls to these models. KLEE understands the semantics of POSIX runtime APIs such as open, read and write.

There are several challenges associated with the use of symbolic execution. First, it suffers from the exponential number of paths in the code—an effect called *path explosion*. The total amount of state and number of constraints grow during symbolic execution, and this limits the size of the programs that can be executed symbolically. Recently, several techniques, such as interleaving symbolic execution with random testing [MS07] and guiding path exploration heuristically [CGP$^+$06,GLM08a], have tried to address this challenge.

Second, the number of iterations of a loop is difficult to decide. Especially, when the number of iterations depends on symbolic variables, loops cause an explosion in the number of execution paths to be explored. In order to solve this problem, various techniques have been introduced, such as the use of simple rules to automatically guess an input constraint defining the number of iterations of input-dependent loops [GL11], introducing new symbolic variables that represent the number of times each loop in a program executes [SPMS09], and the use of simple abstraction heuristics [CTS08]. Since most network protocol implementations process multiple incoming network packets using loops, this challenge need to be addressed in order to apply symbolic execution to network protocol implementations.

The third problem concerns interactions with the environment, such as data reads from the file system. In order to execute a program, the program needs to symbolically invoke all libraries and drivers symbolically. However, such symbolic invocations require extensive amounts of memory and computation power. The latest generation of tools tackle this problem by providing custom models and libraries that help explore all possible legal interactions with the target program's environment [CDE08, BHK$^+$07]. Since network protocol implementations usually interact with the environment and also need to interact with other network entities, this is an important requirement for a technique checking network protocol implementations.

2.5 Summary

This chapter has outlined research relevant to the design and implementation of a network verification system. We began with the definition of several terms, such as network protocol, network specification and network verification, which are required to understand a technique checking network protocol implementations. We then provided an overview of various formal verification techniques, which can be used to check implementations. The survey showed the benefits of formal verification for network protocols. However, some of the limitations of formal verification techniques, such as the state space explosion problem, motivate the need for a new approach that can effectively check network protocol implementations while avoiding such limitations. In particular, we mentioned the limitations through our case study applying model checking to a network management protocol. Since we believe symbolic execution is a technique that is most suitable, we provided a detailed overview of symbolic execution, especially focusing on test case generation and its limitations.

CHAPTER 3

Checking Network Protocol Implementations

IN this chapter, we propose a new approach for checking network protocol implementations via symbolic execution and rule-based packet analysis. Since network protocol implementations are fundamentally different from other software programs (e.g. they typically need complex packet exchanges), we devise two exploration methods for performing symbolic execution on protocol implementations. To provide broad and deep exploration of the state space of a target implementation, we generate test input packets that achieve high source code coverage. We then replay a set of generated input packets and observe potential violations of rules derived from the protocol specification, which will be introduced in Chapter 4. We implement this approach as part of a practical network protocol checking system called SYMBEXNET. It first symbolically executes a protocol implementation to generate high coverage input packets and then automatically checks a set of rules constraining permitted input and output packets.

We demonstrate how our approach generates effectively test input packets for protocol implementations using symbolic execution. Before describing the architecture of SYMBEXNET, we discuss the requirements of using symbolic execution for checking network protocol implementations in Section 3.1, followed by a description of the SYMBEXNET system architecture in Section 3.2. In Section 3.3, we explain how test input packets are generated from protocol implementations using symbolic execution. Two methods of performing symbolic execution that provides broad and deep exploration of the state space are introduced in Sections 3.4 and 3.5, respectively. Our prototype implementation of SYMBEXNET is discussed in Section 3.6. The chapter finishes with a discussion and summary in Section 3.7.

3.1 Requirements

Although symbolic execution has improved considerably in terms of performance and scalability, there remain problems when it is applied to network implementations. The main challenges are:

Challenge 1 (C1): Deciding which bytes to mark as symbolic. The behaviour of a network implementation is determined by the input packets that it receives from other entities. For example, a DNS server receives UDP query packets from clients and replies with UDP response packets after having resolved the DNS names in the query packets. Thus, symbolic execution, when checking network protocol implementations, must consider the input packets from the network as symbolic values. Since symbolic execution executes an implementation with these symbolic inputs and generates test input packets, deciding which bytes to mark as symbolic has a big impact on the quality of generated test cases [AGT08, BCE08].

For large and complex network protocol implementations, it is infeasible to mark the complete packet as symbolic because this would result in too many paths to be explored during symbolic execution. As the size of a symbolic input increases, maintaining and solving symbolic constraints associated with the inputs along the paths becomes expensive. When checking network protocol implementations, deciding which input packets and bytes to mark as symbolic is an important factor in order to generate good test packets while avoiding for path explosion problem.

Challenge 2 (C2): Handling of multiple packet exchange. Many network protocol flaws occur after handling specific sequences of packets, not just a single packet. If a network protocol implementation is executed symbolically with a single input packet, the generated test packets cannot detect flaws caused by these specific sequences of packets. Therefore, utilising symbolic execution to cover not only a single symbolic packet but also a sequence of symbolic packets is an important challenge in order to detect complex errors appearing only after multiple packets.

Challenge 3 (C3): Detecting incorrect behaviour. By default, performing symbolic execution on network protocol implementations detects only low-level generic errors, e.g. out-of-bound accesses, division by zero, assertion failure etc. [CDE08]. However, there are additional errors that are caused by an incorrect implementation of the specification. These errors are difficult to detect using symbolic execution, because they require protocol-specific knowledge.

In order to detect these errors, developers can use pre-defined inline assertions. These assertions specify undesired behaviour or errors in terms of internal program variables, environment variables and external system states. For example, if a protocol specification states that the time to live (TTL) field value can never be zero, a developer can add "`assert(TTL != 0)`" at the proper point in

the program. However, adding such inline assertions is expensive and requires manual effort.

Challenge 4 (C4): Assuring interoperability. If flaws are caused by misunderstandings of requirements in a protocol specification, they cannot be detected easily because developers believe them to be correct behaviour. These types of flaws can be found if there are other implementations faithfully implement the same protocol based on a correct understanding of the specification. By comparing the output packets in response to the same input packet, multiple implementations for the same protocol specification can be tested and compared under the same conditions. This approach can discover interoperability problems.

3.2 SYMBEXNET Architecture

Our goal is to determine the compliance of a network protocol implementation with its protocol specification and the interoperability with other network entities. Our approach combines the techniques of testing, verification and symbolic execution. It is simple to use yet rigorous enough to discover non-trivial bugs, providing an automated method to check protocol implementations.

At a high level, our approach performs *test packet generation*, *test packet replay* and *validation of packet rules* in order to check a network protocol implementation and its interoperability. It uses symbolic execution on a given network protocol implementation to generate test input packets. We propose symbolic exploration methods that can explore broad and deep program state spaces so that the generated packets can achieve high code coverage. During *test packet replay*, we consider the implementation as a "black-box". It processes the set of generated test packets, and we observe the output packets generated validating them for compliance against to the protocol specification.

The SYMBEXNET architecture is shown in Figure 3.1. To check a network protocol implementation, SYMBEXNET takes two inputs, the specification and the source code. The specification and source code are used to derive a set of packet rules and generate test input packets, respectively. Any violations detected are reported as the output of the protocol checking.

When verifying a network protocol implementation with SYMBEXNET, there are five steps, as labeled in the figure. First, a set of rules describing the behavioural properties are derived from the protocol specification by developers (❶). SYMBEXNET symbolically executes the network protocol implementation to obtain a set of test input packets that result in high code coverage when processed by the network implementation (❷). After that, it executes the set of test packets under controlled conditions and observes the output packets generated by the network implementation (❸), which are validated for compliance against the protocol specification (❹) or interoperability checking (❺). Each step is described in more detail below:

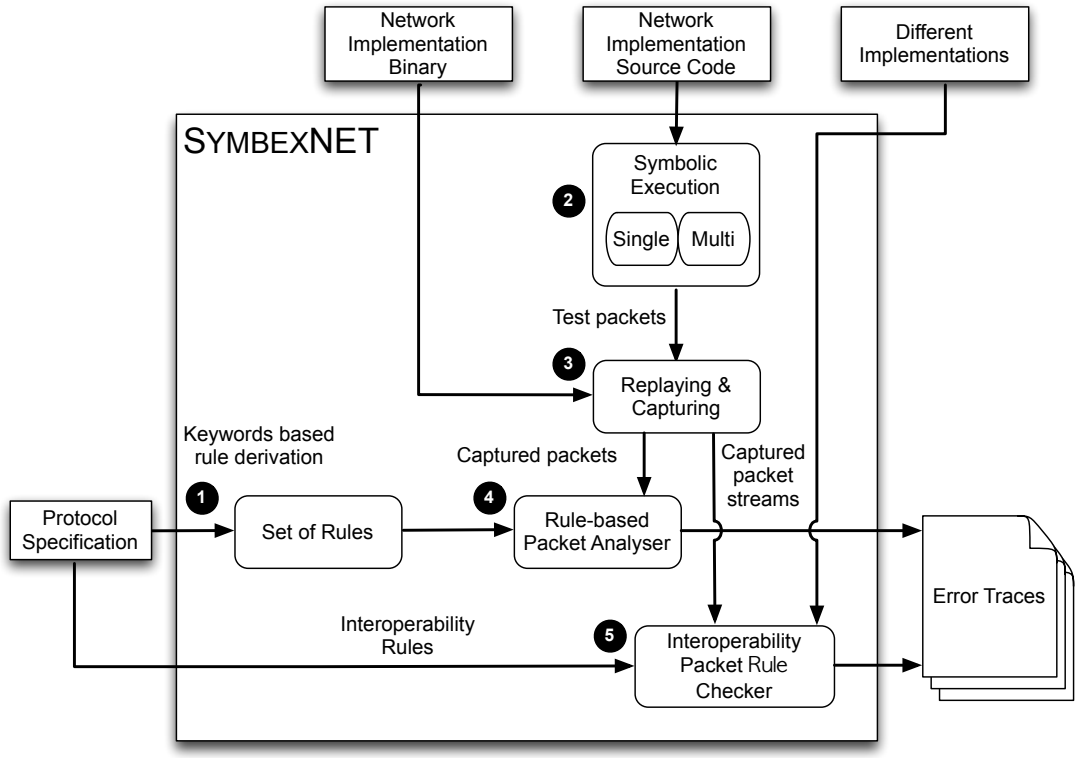


Figure 3.1: SYMBEXNET system architecture.

1. **Creation of packet rules** (cf. Section 4.3). The first step is to develop a rule-based verifiable specification from a protocol specification. The requirements describing behavioural properties of the protocol are extracted from the protocol specification and expressed in terms of the desired input-output behaviour (i.e. the set of packets). SYMBEXNET provides a packet rule language to describe correct sequences of packets.

2. **Generation of test packets** (cf. Sections 3.4 and 3.5). To validate as many packet rules as possible, SYMBEXNET requires a good set of test packets with high code coverage. It uses symbolic execution to explore a large number of code paths in the network protocol implementation and, based on this, synthesises a set of test input packets. To satisfy the challenge C1 requirement (*deciding which bytes to mark as symbolic*), SYMBEXNET runs the implementation on a symbolic input packet by repeatedly marking parts of the packet as symbolic.

   Generated test packets can achieve a broad and deep exploration of the state space of the protocol implementation. To explore state space broadly, SYMBEXNET applies symbolic execution on a single symbolic input packet and generates test input packets (cf. Section 3.4). For deep exploration, it repeatedly performs symbolic execution on selected symbolic input packets to generate sequences of test input packets (cf. Section 3.5). This method satisfies

the challenge C2 (*handling of multiple packet exchange*).

3. **Replay of test packets** (cf. Section 3.4.2). The generated test packets are replayed on the original network implementation. Each test packet is sent to the implementation in a controlled environment, and the output packets generated by the implementation are recorded by SYMBEXNET, together with the input packets, as a packet stream.

4. **Validation of packet rules** (cf. Section 4.6). The challenge C3 (*detecting incorrect behaviour*) can be overcome by means of packet rule validation. In this step, the captured input and output packets from the previous step are validated against the packet rules from step 1. SYMBEXNET translates the packet rules into a set of non-deterministic finite automata (NFAs). Through analysing all captured replay packets against each NFA, SYMBEXNET detects rule violations. The rules can also check interoperability between packet streams from multiple implementations of the same protocol.

5. **Checking for Interoperability** (cf. Sections 5.3 and 4.4.1). In this step, SYMBEXNET provides a solution for the challenge C4 (*assuring interoperability*). To check if multiple implementations of the same protocol specification are interoperable, SYMBEXNET applies steps 2 and 3 to the implementations and generates packet streams for each implementation. Generated packet streams obtained from an implementation are replayed not only on the original implementation but also across implementations. These two steps are performed under the same conditions, thereby the behaviour of an implementation for a given input can be compared with others.

   For checking interoperability, SYMBEXNET uses interoperability rules that normalise a captured packet stream. The normalised packets from a stream of an implementation are compared to check for inconsistencies. For each inconsistency, SYMBEXNET reports an interoperability error: the test input value that led to the interoperability error and the behaviour of implementation for the given test input value.

3.3 Symbolic Execution and Exploration

SYMBEXNET uses *symbolic execution* to check the compliance and the interoperability of network protocol implementations. In both cases, SYMBEXNET executes the network protocol implementation symbolically by marking sets of bytes in an input network packet as symbolic variables. Symbolic execution then explores all (or as many as possible in a given time budget) code paths in the network implementation that are related to a symbolic variable. To do that, SYMBEXNET (1) marks packet fields as symbolic variables and (2) injects the symbolic packets into the network protocol implementation.

3.3.1 Symbolic Marking

As explained in Section 2.4.2, a set of bytes of an input packet need to be marked as symbolic. An open challenge is to decide which fields to mark as symbolic while avoiding the path explosion problem. Marking the complete input packet as symbolic would result in too many paths, and most of these paths would not increase code coverage because they would refer to invalid packets, which are normally discarded by an implementation.

Typically a network packet consists of multiple fields, which are part of the packet header. Most protocol implementations contain logic for handling these fields. Therefore, SYMBEXNET uses these fields as symbolic variables instead of entire input packets.

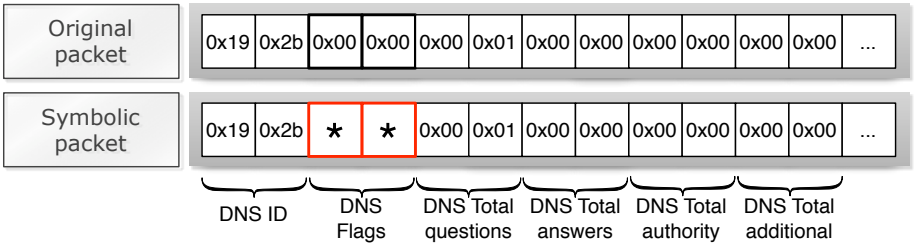


Figure 3.2: Original and symbolic packets.

For example, Figure 3.2 shows how an input packet is processed as a symbolic packet. Let us consider that the *flags* field is marked as symbolic. When SYMBEXNET receives an input packet that needs to be handled as a symbolic packet (*original packet* in the figure), it marks the *flags* field as symbolic and replaces the concrete value of this field within the packet with symbolic values while keeping the other fields unchanged.

Since the size of the symbolic fields is small compare to the whole packet, fewer symbolic constraints are maintained during the execution, thereby alleviating path explosion problem. It is important to be strategic and only mark packet fields symbolic that are likely to result in high coverage gains. Our technique allows developers to mark any combinations of packet fields as symbolic. Comparing the number of generated test packets for different combinations of packet fields help understand the sensitivity of each field in the implementation. In the experiments in Section 6.4.1, we consider all combinations of fields in DNS packets, starting with one symbolic field, and then progressively advancing to more fields, with good results.

We propose two alternative interfaces to developers for symbolic marking: *field mode* and *custom mode*. Figure 3.3 shows how these two modes can be used to mark packets as symbolic. In field mode, developers decide which fields to mark as symbolic. This requires SYMBEXNET to have access to a description of the offsets and lengths of packet fields for a given protocol specification. In custom mode, symbolic bytes in a symbolic packet are specified as pairs of offsets and lengths. Thus, developers select which bytes of the input packet are marked as symbolic. In this mode,

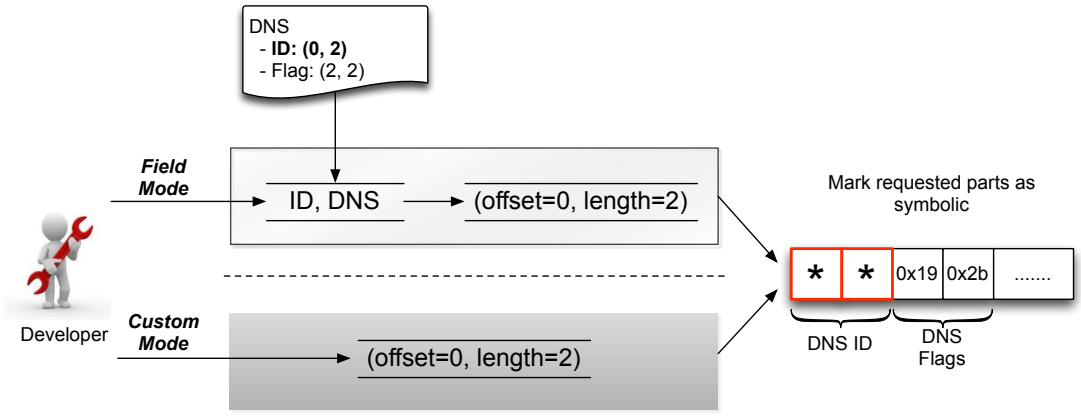


Figure 3.3: Marking modes for a symbolic input packet.

SYMBEXNET does not require any information about the packet format.

3.3.2 Symbolic Packet Injection

To perform symbolic execution, symbolic packets must be injected into the network and processed by the implementation under testing. An important issue is to decide how to inject symbolic packets without requiring major changes to the network protocol implementation. When an implementation receives a given input packet, the packet is intercepted by SYMBEXNET, which marks all or parts of it as symbolic and starts the symbolic execution process using its symbolic execution engine.

Since symbolic packets are delivered over the network, the location in the source code, of which the implementation receives and processes input packets must be identified. Most C network protocol implementations use the standard socket API to receive input packets on a given port (e.g. port 5353 for mDNS).

Figure 3.4 shows an example of source code from a network protocol implementation that processes incoming packets from the network. Lines highlighted with gray are handcrafted to identify symbolic input packets. The implementation follows a standard way of network programming: it creates a socket with the `socket()` system call, binds the socket to an address consisting of a port number using the `bind()` system call and receives data using the `recvmsg()` system call. When the implementation receives a symbolic input packet, it marks the packet as symbolic variable using the `klee_make_symbolic()` function.

When SYMBEXNET executes an implementation symbolically on symbolic inputs, it explores the paths corresponding to various input packets. How to explore the paths is another important factor for the quality of generated test packets. If a method explores the search space of a program effectively, it covers many possible execution paths in the program. In order to explore the implementation effectively, we present two symbolic exploration methods, in the next section.

```
1  if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < -1) {
2    avahi_warn(``socket() failed'');
3  }

5  server_addr.sin_family = AF_INET;
6  server_addr.sin_port = htons(AVAHI_MDNS_PORT); // 5353
7  av = bind(fd, &server_addr, sizeof(server_addr);

9  len = recvmsg(fd, &mdns_msg, 0); // receive mdns message

11 if(memcmp(mdns_msg, ``symbolic_packet'', 15) == 0) {
12  /* discover symbolic packet and mark it as symbolic */
13    char* sym_buf;
14    klee_make_symbolic(sym_buf, sizeof(sym_buf), ``symbolic_variable'');
15 }
```

Figure 3.4: An example of network implementation source code.

3.4 Single Packet Exchange Symbolic Execution

In *single packet exchange symbolic execution* (SPE-SE), SYMBEXNET performs symbolic execution on a single input packet specified as a symbolic input. Since SPE-SE performs symbolic execution to a single symbolic input packet and explores program code paths associated with that packet, it is well suited for stateless network protocol implementations that treat each request independently from previous requests. The checking process of SYMBEXNET using SPE-SE is composed of two tasks: *test packet generation* and *test packet replay*.

3.4.1 Test Packet Generation

In order to run an implementation symbolically, SYMBEXNET sends a symbolic input packet to the network. When the implementation receives the symbolic packet, it intercepts the packet and marks given packet fields as symbolic. When encountering branches that depend on symbolic variables, symbolic execution automatically generates alternative values and executes paths for these values. Symbolic execution then explores all possible execution paths corresponding to the various input packets. At the end of each execution path, SYMBEXNET generates a concrete test packet that is stored on a local disk.

3.4.2 Test Packet Replay

Generated test packets are then executed (“replayed”) using the original network protocol implementation. The replay process automatically executes an unmodified version of the implementation

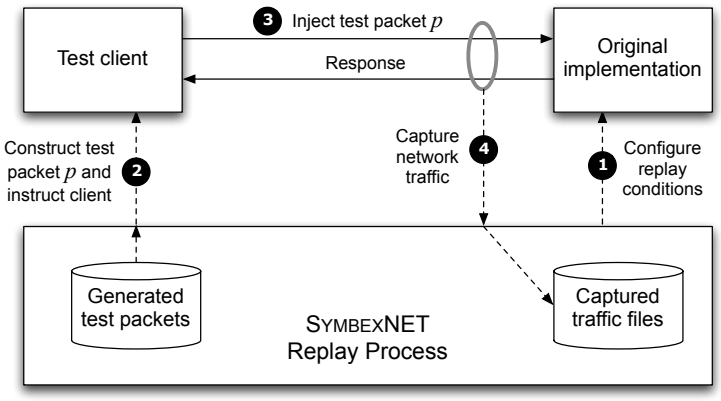


Figure 3.5: The replay process for a test input packet p.

on all of the test packets generated by SYMBEXNET.

Figure 3.5 illustrates how SYMBEXNET performs replay. The process comprises the following four steps, as labeled in the figure: it configures the replay conditions and runs the unmodified native implementation (❶); it runs a test client (❷), it coordinates a test client to inject a test packet (❸); and it captures the network traffic (❹).

Configure the replay conditions (❶). SYMBEXNET executes the unmodified implementation under the same conditions under which the test packets were generated (e.g. using the same configuration parameters) so that any violations detected during symbolic execution can be confirmed. At the beginning of the replay process, SYMBEXNET starts the implementation and configures the replay network. For example, in the case of the Zeroconf protocol, this includes registering the same services, opening the same port numbers, etc.

Construct test packets during the replay stage (❷). When the unmodified implementation executes in the configured environment, SYMBEXNET selects one of test cases to construct a test input packet. The value of the individual test case provides detailed information on how to construct a test input packet. SYMBEXNET uses this information to create input packets that have concrete values derived from the test case. For example, if a test case is for the `flags` field and its value is `0x82`, constructing a test input packet p replaces the value of the `flags` field with `0x82`. SYMBEXNET communicates to the client information required to send the constructed packet p to the implementation (e.g. destination IP and destination port).

Inject a test packet to the network (❸). The client establishes a communication channel to the implementation. After the channel was established, the client injects the test packet p into the test network. SYMBEXNET controls the client so that it can send the test packet when the implementation is ready to receive it, such as after completing service registrations. Replayed packets causing crashes of the implementation are reported during the replay process.

Capture network traffic (❹). To validate the network protocol implementation, SYMBEXNET records all network traffic, i.e. input and output packets generated by the implementation and clients during the replay. The captured traffic is used as one of inputs in the next step, rule validation (see Chapter 4). If a network protocol implementation shows unintended behaviour for a test packet, rule validation detects this based on the recorded network traffic.

3.5 Multi Packet Exchange Symbolic Execution

The method described in the previous section is useful to check stateless network protocol implementations because they process each input packet as an independent request. However, it is not suitable to check stateful network protocol implementations because the generated test packets cannot explore code execution paths that can be reached only after receiving more than one input packet. In order to overcome this limitation, we develop multi packet exchange symbolic execution (MPE-SE). Unlike SPE-SE, MPE-SE performs symbolic execution on multiple symbolic packets instead of a single symbolic packet. It can thereby generate all possible test packet sequences that a target implementation can receive.

3.5.1 Problem of Symbolic Exploration

Next we illustrate why SPE-SE cannot explore deep code paths using the example in Figure 3.6. The source code shows a finite state machine in its simplest form, which is used in many network protocol implementations, such as DHCP, to handle a series of incoming packets based on its previously received packets.

The protocol has an initial state, `ST_START`, and takes three successive input packets $p1$, $p2$ and $p3$, which match to the states, `ST_1`, `ST_2` and `ST_END`, respectively. In order to check the code, if we mark the first input packet p as symbolic, symbolic execution explores the statements located within the `switch` statement in line 8. Since we mark $p1$, the first received packet, as symbolic, symbolic execution generates the test input packets that start with `0`, `1`, `2` and `3` for `ST_1`, `ST_2`, `ST_3` and `ST_END`, respectively. Note that the `switch` statement checks the first byte of the input packet. These test packets are all possible inputs that the program is expected to receive and broadly explore the statements for the corresponding state. For example, the test input packet starting with `0` explores the statements between lines 10 to 14.

When symbolic execution reaches to line 19, it generates a test packet that starts with `11` because the `if` statement accesses the second byte of the input packet. However, the statements in lines 22 and 23 (highlighted with gray in the figure) cannot be explored because of the condition comparing the value of the `first_flag`. The value of `first_flag` becomes `true` only after processing the

```
1  enum state {ST_1, ST_2, ST_3, ST_END};
2  boolean first_flag = False;

4  while (current_state != ST_END)
5  {
6    input = get_input();

8    switch (input[0])
9    {
10     case ST_1:
11     /* Do something with input and set first_flag to True */
12       handle_first_pkt(&input);
13       first_flag = True;
14       break;

16     case ST_2:
17       if (input[1]==0)
18         handle_query(&input);
19       else if ((input[1]==1) && first_flag) {
20       /* The following routines are not explored */
21       /* if we only mark the first arrived packet as symbolic */
22       handle_requested(&input);
23       abort(); /* error */
24       }
25       break;

27     case ST_3:
28     /* Do something different with input and set current_state */
29       handle_final_packet(&input);
30       break;

32     /* ... etc ... */
33   }
34 }
```

Figure 3.6: An example C program for a state machine in a traditional network protocol implementation.

first packet, and consequently sending a test packet starting with `11` cannot explore the source code located within the `if` statement.

Similar to this example, when receiving an input packet, many network protocol implementations make decisions based not only on information that is contained in the packet but also from previously received packets. Therefore, SPE-SE cannot explore the code paths that depend on a sequence of received packets.

3.5.2 Generation of Packet Sequences

To explore the statements that were left unexplored in the previous example, SYMBEXNET introduces the concept of *multi-packet exchange symbolic execution* (MPE-SE). MPE-SE explores all possible sequences of packets using multiple symbolic input packets.

Since MPE-SE handles multiple symbolic packets and generates a set of packet sequences, it needs to store the symbolic packet that is currently being processed and the generated packet sequences. When SYMBEXNET checks a network protocol implementation, it maintains: (1) a *symbolic index* indicating the sequence number of an input packet that is to be executed symbolically and (2) a *symbolic sequence tree* containing all generated sequences of test packets.

The *symbolic sequence tree* characterises the set of sequential input packets generated during symbolic execution. In the tree, nodes represent program states, and edges between nodes represent transitions between states. Each transition is labeled with the input packet required to effect the state change.

Initially the symbolic index is set to one and the symbolic sequence tree only has a root node. When SYMBEXNET encounters a branch while executing the implementation symbolically, it explores all possible values and generates one test input packet for each explored path. Since each input packet triggers a state change, after generating all possible test packets for a given symbolic index, SYMBEXNET adds a node for each test packet to the tree and labels the transition with the packet. For example, if SYMBEXNET generates 10 test packets after symbolically running an implementation with the first symbolic input packet, it adds 10 nodes for these packets to the root of the tree, labeling each transition with the test packet.

Thus, the final symbolic sequence tree contains all possible sequences of input packets that explore the program state. By means of MPE-SE, SYMBEXNET exhaustively explores the code paths of the program and generates a symbolic sequence tree, representing all possible sequences of packets that the program can handle.

To explore code paths efficiently, SYMBEXNET employs a combination of concrete and symbolic execution. It uses the generated sequences of packets, which enable it to explore deep states of the implementation, as input for next round of symbolic execution.

Figure 3.7 gives a high-level overview of MPE-SE. Since MPE-SE repeatedly performs symbolic execution and updates the maintained symbolic information, it has two phases: *execution* and *generation*.

In the execution phase, the implementation behaves normally, i.e. processes all input packets concretely until it receives a symbolic packet. When the implementation receives a symbolic packet, it executes symbolically and generates test packets. In the generation phase, SYMBEXNET updates

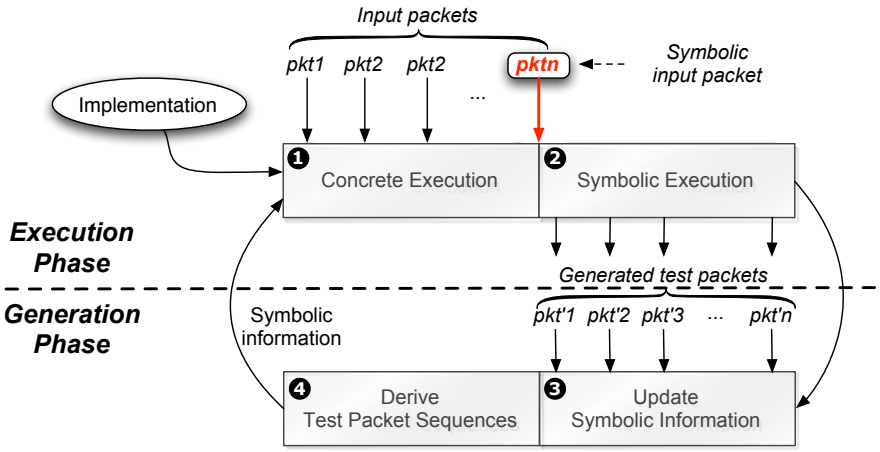


Figure 3.7: High-level overview of multi packet exchange symbolic execution.

Require: symbolic index SI, symbolic sequence tree ST
Ensure: symbolic sequence tree ST
1: $SI = 1$
2: $ST = root$
3: **repeat**
4: temp_paths = find_all_path(ST)
5: **for all** path in temp_paths **do**
6: perform_concrete_execution(path)
7: receive_symbolic_packet()
8: generated_paths = perform_symbolic_execution()
9: add_generated_path(ST, generated_paths)
10: **end for**
11: $SI = SI + 1$
12: **until** check_for_availability(ST)

Figure 3.8: Pseudocode for MPE-SE.

the symbolic index and symbolic sequence tree and derives test packet sequences from them. SYMBEXNET repeats both phases until it cannot find a packet sequence that can explore new program code paths.

The pseudocode for MPE-SE is presented in Algorithm 3.8. It consists of 4 steps, as labeled in Figure 3.7:

1. **Concrete execution.** The protocol implementation concretely processes all input packets that arrive before receiving symbolic packet (line 6). After that, it awaits a symbolic input packet. SYMBEXNET manages the sequence numbers of the received packets and decides when to run the implementation symbolically based on the value of the symbolic index SI.

2. **Symbolic execution.** If the sequence number of an input packet is equal to the value of

the SI, SYMBEXNET marks the input packet as symbolic and executes the implementation symbolically instead of concretely (line 8).

3. **Update symbolic information.** After performing symbolic execution on the implementation, the generated test packets are inserted into the symbolic sequence tree ST and the value of SI is increased by one (line 9 and 11).

4. **Derive test packet sequences.** SYMBEXNET derives all possible packet sequences from the symbolic sequence tree. For each sequence, SYMBEXNET checks whether the next symbolic execution can be performed. If no further symbolic execution is required, SYMBEXNET terminates the process of packet sequence generation. For sequences that allow the implementation to receive the next symbolic packet, SYMBEXNET feeds the selected sequences to the next execution phase, in order to explore more code execution paths (line 12).

3.5.3 Formal Model

We formalise MPE-SE and show how the defined concepts play a key role in generating sequences of packets. A model for MPE-SE is a tuple:

$$M = (\Sigma_S, \Sigma_T, \Sigma_\sigma, \Sigma_{pkt})$$

where Σ_S, Σ_T, Σ_σ and Σ_{pkt} are finite sets of states, transitions, symbolic packets and generated input packets, respectively. Here, Σ_{pkt} is the set of generated test input packets using MPE-SE. Σ_{pkt} can be decomposed as follow: $\Sigma_{pkt} = \Sigma_{pkt_1} \cup \Sigma_{pkt_2} \cdots \cup \Sigma_{pkt_n}$ where Σ_{pkt_n} is the set of generated test packets from the symbolic execution with the symbolic index *n*. Each transition t in the set Σ_T is a 3-tuple, $t = (s_t, e_t, pkt_t)$, where s_t, e_t and pkt_t are the start state, next state and input packet, respectively.

States used in MPE-SE, called *symbolic states*, differ from normal states, on which the usual program execution operates. Symbolic states can store information about symbolic execution: the sequence of packets to reach the current state, the symbolic input packet and the generated test input packets. A symbolic state is a 4-tuple:

$$\eta = (s, \pi, \sigma, \Sigma_{pkt_s})$$

where $s \in S$ is a state; π is a sequence of packets that brings the execution to the state $pkt_1 : pkt_2 : \cdots : pkt_n$ for all $pkt_1, pkt_2, \cdots, pkt_n \in \Sigma_{pkt}$; $\sigma \in \Sigma_\sigma$ is an input packet that needs to be handled symbolically; and Σ_{pkt_s} is a set of generated test input packets after symbolic execution with the given symbolic index.

3.5.4 Code Exploration

Let us revisit the example C program introduced in Figure 3.6, which has statements that cannot be explored by SPE-SE. We now apply MPE-SE to the example to show how the method can explore stateful network protocol implementations effectively.

Initially, SYMBEXNET creates an empty symbolic sequence tree and sets the *symbolic index* to `1`. Since the symbolic index is equal to `1`, when the implementation receives an input packet for the first packet exchange, SYMBEXNET marks the input packet as symbolic and starts to explore the implementation symbolically— the first symbolic execution. After completing the exploration of the implementation symbolically, SYMBEXNET generates the test input packets that start with `0`, `1`, `2` and `3` for `ST_1`, `ST_2`, `ST_3` and `ST_END` and inserts a corresponding four nodes into the symbolic sequence tree. As discussed, the code between lines 22 and 23 cannot be explored using the generated test packets from the first symbolic execution.

After updating the symbolic sequence tree, SYMBEXNET increments the index to `2` and computes all possible packet sequences in the tree starting from the root to each leaf node. Next, SYMBEXNET runs the implementation concretely with each derived sequence where execution reaches line 6 after completing the first packet exchange. For example, if SYMBEXNET runs the implementation with a test packet starting with `ST_1`, the implementation executes the function `handle_first_pkt()` followed by setting `first_flag` to `true` and awaits the next packet exchange.

Since the value of the symbolic index is now `2`, SYMBEXNET marks the second input packet as symbolic and explores the implementation with another symbolic packet—the second symbolic execution. SYMBEXNET generates a test packet starting with `11` when it reaches a symbolic branch condition (line 19). Previously, the test packet with the same value did not explore the statements within the branch because `first_flag` was `false`. However, since the concrete execution of the first packet exchange configured the `first_flag` to `true`, SYMBEXNET can explore lines 22 and 23 successfully.

We have illustrated how SYMBEXNET explores the source code of a network protocol implementation using MPE-SE. During MPE-SE, SYMBEXNET builds concrete test packet sequences that lead to new code paths. Whenever SYMBEXNET finishes a symbolic execution with a given symbolic index, it generates test packets and updates the symbolic sequence tree. SYMBEXNET saves concrete test packet sequences as part of the symbolic sequence tree.

Figure 3.9 compares the program exploration method for (a) SPE-SE and (b) MPE-SE. In SPE-SE, test packets generated from the first symbolic input packet and these packets explore possible execution paths. In MPE-SE, test packets are generated not only from the first symbolic packet, but also from subsequent symbolic packets, which ensure a deeper exploration of execution paths.

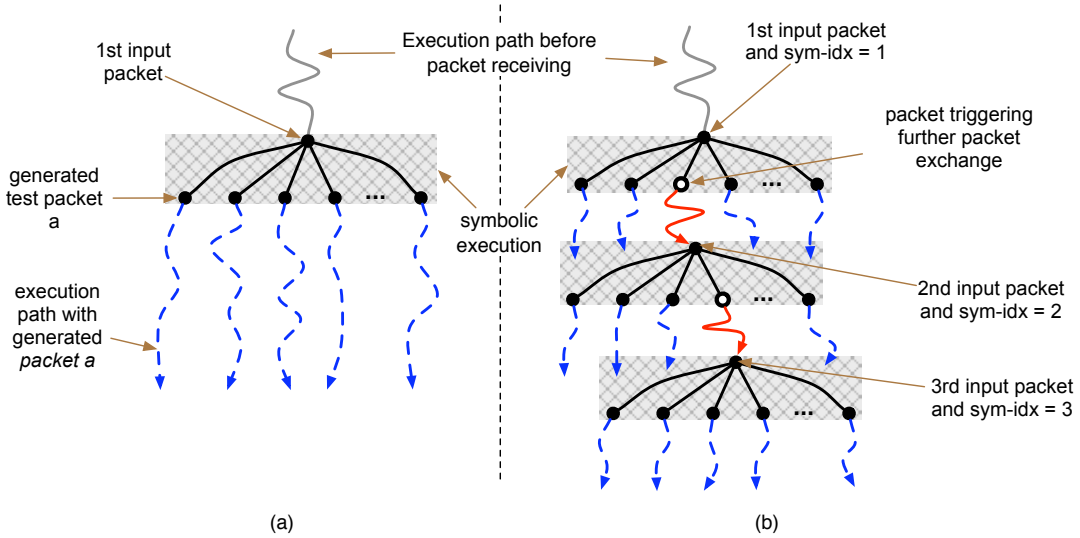


Figure 3.9: The program exploration method for a) SPE-SE and b) MPE-SE.

Symbolic execution availability check. As explained in the previous section, the generated test packets using MPE-SE trigger the next round of symbolic execution. However, not all test packets generated by MPE-SE allow an implementation to receive the next symbolic packet. For these packets, no further rounds of symbolic execution are required. Identifying these test packets can reduce execution time of MPE-SE. For this purpose, SYMBEXNET uses a *symbolic execution availability check*.

Packets that do not trigger symbolic execution are invalid packets. They typically explore functions that validate input packets, thereby making the implementation simply discard these packets. Therefore, these packets can be identified by checking for the existence of response packets.

Identifying these packets while a protocol implementation is executed symbolically is difficult for two reasons. First, symbolic execution runs compiled code similar to an interpreter. Execution takes much longer than running natively. Second, only after spending all the given time budget for symbolic execution, it knows that no response packet was generated.

Therefore, before running symbolic execution with each generated test packets, SYMBEXNET runs the unmodified implementation with the generated test packets and verifies if the implementation can receive a symbolic packet. We refer to this as the *symbolic execution availability check*. If the test packets allow the implementation to receive a symbolic packet, SYMBEXNET terminates the availability check. SYMBEXNET then runs the implementation symbolically with the symbolic packet and generates further test packets.

3.5.5 Replaying Sequences of Test Packets

In SPE-SE, symbolic execution generates test input packets and SYMBEXNET replays the packets on the original version of the implementation. In contrast, MPE-SE uses a sequence tree containing sequences of input packets that SYMBEXNET must replay.

For replay, SYMBEXNET first derives a set of packet sequences from the symbolic sequence tree. Since each node of the symbolic sequence tree represents a generated test packet, the packets from the root to each leaf node of the tree form a packet sequence. Packet sequences are then executed with an unmodified protocol implementation using the same replay method as described in Section 3.4. When injecting the test packets of a packet sequence into the network, SYMBEXNET sends them in the order of their creation. Any replayed sequence of packets causing crashes is recorded during the replay process. All the network traffic between the implementation and clients during replay is also captured to validate the behaviour of the implementation against its specification.

3.6 SYMBEXNET Implementation

In this section, we present our prototype implementation of SYMBEXNET. There are two versions of SYMBEXNET: a regular version (default) that will be used to detect errors violating requirements described in the specification and an enhanced version that finds interoperability problems, which will be introduced in Chapter 5.

In the following, we focus on the regular version, describing the components of the SYMBEXNET system. The components can be classified into two categories: *protocol-specific components* and *common components*.

3.6.1 Protocol Specific Components

Since SYMBEXNET is a system that checks network protocol implementations, it requires components specific to the target network protocol. In SYMBEXNET, three components are classified in this category: an *unmodified implementation*, a *symbolic implementation* and a *test client*. To check a target protocol implementation, developers need to prepare these three components. During the process of generating test packets using symbolic execution, SYMBEXNET uses the *symbolic implementation* and the *test client*. For the replay process, SYMBEXNET uses the *unmodified implementation* together with the *test client*.

Symbolic implementation. Since SYMBEXNET uses symbolic execution for test input generation, an implementation need to be compiled into a bytecode format that can be executed by symbolic execution. For example, a *symbolic implementation* is in a bytecode format compiled using the

LLVM compiler.

Unmodified implementation. In SYMBEXNET, an *unmodified implementation* is an executable binary program compiled with a default compiler, such as GCC. Each generated test case is replayed on the unmodified implementation so that the behaviour of the implementation can be analysed.

Test client. SYMBEXNET also requires a test client that can communicate with the unmodified and symbolic implementations. The client behaves as a network entity to the protocol implementation. It sends symbolic input packets to the symbolic implementation for symbolic execution and generated test input packets to the original implementation for replay.

3.6.2 Common Components

Most components in SYMBEXNET are independent of the targeted protocol implementation. These are the *database*, the *controller*, the *monitor* and the *symbolic execution engine*.

In SYMBEXNET, the *database* stores all information required to check a network protocol implementation, including packet rules, interoperability rules, generated test packets and observed traffic. A *controller* controls protocol-specific components and coordinates their behaviour. It decides when to start the implementation and sends symbolic and test packets to the network. During the replay process, SYMBEXNET treats a protocol implementation as a black-box and inspects traffic entering and leaving the implementation to detect violations. A *monitor* observes network traffic between the implementation and the clients during the replay of each test case. It stores the exchanged traffic into a `.pcap` file, which is used as one of the inputs to the next step.

Symbolic execution engine. A *symbolic execution engine* executes an implementation on symbolic inputs resulting in the creation of test input packets. SYMBEXNET uses KLEE [CDE08] as a symbolic execution engine because it supports C programs, can generate high-coverage tests and can detect low-level bugs. For symbolic execution, the implementation needs to be compiled to LLVM bitcode and is interpreted by KLEE. KLEE executes bitcode but collects symbolic constraints over the symbolic input values at each branch.

Since SYMBEXNET checks the correctness of network protocol implementations, KLEE must support network related system calls such as `socketcall`. Currently, KLEE does not provide networking models, thus cannot support socket system calls. In order to perform symbolic execution on network protocol implementations, we extended KLEE to include a simple networking model. It supports the semantics of most network-related POSIX system calls, such as `socketcall` and `sigaction`, by extending the existing file system model in KLEE. Thus, KLEE can deal with requests to these network-related system calls because the model understands the semantics of the required action in order to generate the desired constraints.

3.7 Summary

In this chapter, we described a technique that can be used to check network protocol implementations. Since our technique uses *symbolic execution*, we started with a discussion of the challenges when applying it to network protocol implementations. We provided an overview of our technique and discussed the five steps required to check the implementation of a network protocol. We then described how our technique generates high-coverage test input packets using symbolic execution with different exploration methods. Based on that, we introduced SYMBEXNET, a practical checking system that realises our technique.

We proposed two different symbolic exploration methods, *single packet exchange symbolic execution* (SPE-SE) and *multi-packet exchange symbolic execution* (MPE-SE). First, the SPE-SE method was described in terms of the generation of test packets using symbolic execution and the replay of generated test packets. After that, we provided a motivating example for MPE-SE, explaining how it can explore deep program execution paths using symbolic execution. Next, we described how it generates sequences of test packets. We also discussed an optimisation method to reduce exploration time of MPE-SE. We illustrated how the generated test packet sequences can be replayed in order to validate protocol implementations. The rest of the chapter focused on the prototype implementation of our technique.

CHAPTER 4

Rule-based Verifiable Specifications

PERFORMING symbolic execution on network protocol implementations can detect low-level generic errors, such as crashes or memory leaks. However, there may also be semantic errors, which expose behaviour not addressed in the specification. If errors do not cause crashes or other obvious behaviour, they become hard to detect using symbolic execution. To discover such errors and then determine the correctness of the implementation, SYMBEXNET checks replayed test packets against packet rules extracted from a protocol specification.

Therefore, an important step in using SYMBEXNET is to make a standard protocol specification, such as an RFC document, verifiable. A verifiable specification allows SYMBEXNET to assess the correct behaviour of a network protocol implementation automatically. We assume that the behaviour of an implementation consists of the output packets that it emits in response to input packets (see Section 3.4.2). We define behavioural violations using a packet rule language that matches incorrect sequences of packets. Our packet rule description language is intended for use by developers of network services and designed based on two requirements: *readability* and *ease of integration with network protocols*.

The packet rule language has four operators (i.e. filter, next, union and iteration) in addition to modifiers that can be used to describe network protocol specific behaviour. Our packet rules are implemented using extended non-deterministic finite automata (NFA). To detect complex packet exchange patterns, our NFAs read captured packet streams and efficiently filter out unrelated intermediate packets. They can also refer to field values of previously detected packets. We do not reason about the internal state of the implementation, which means that packet rules are reusable across different implementations of the same protocol. This allows developers to identify and correct errors of translation and migration between different implementations of a specification.

The rest of this chapter is structured as follows. Next we provide an overview of packet analysis techniques. We then illustrate the process of packet rule validation in Section 4.2. After that we show how rules are derived from specifications in Section 4.3 and introduce our packet rule language in Section 4.4. The chapter finishes with several examples of packet rules (Section 4.5), a

brief introduction of the rule implementation (Section 4.6) and related work on rule-based program analysis in Section 4.7.

4.1 Packet Analysis Techniques

Network traffic data, i.e. input and output packets between network entities, contains considerable information such as the time and duration of communication, the identities of the parties communicating and the type of communication protocol. Since this can be used to understand the behaviour of a protocol implementation, traffic analysis techniques have been used to solve many network management and security problems [BKPR02].

For this purpose, researchers have proposed languages that can formally describe protocol behaviour and packet analysis techniques that can examine network traffic to detect undesired input and output packets in the traffic. However, many of them do not support analysing complex sequence of packets and provide techniques that depend on a specific protocol. In addition, performing network packet analysis at runtime typically requires resources and then degrades performance.

We use a packet analysis technique in SYMBEXNET to detect undesired exchanges of input and output packets, which violate requirements specified in a protocol specification. However, our analysis is performed on stored instead of live network traffic. Before providing details of our packet analysis technique and packet rule language, we give an overview of general packet analysis techniques in the next section. In Section 4.1.2, we show how rules are described and matched in our language.

4.1.1 Overview of Packet Analysis Techniques

Packet analysis techniques investigate the content of data packets exchanged in a network, which includes headers and payloads, and compare it against given rules, for example, network attack rules [Roe99, Pax98]. Based on these rules, the traffic is analysed to determine whether it contains undesired patterns. As a result, malicious network intrusions or denial-of-service attacks can be detected. For example, a *network monitoring system* [MHL94] scans the packet headers and payloads in order to detect a given set of security signatures.

As illustrated in Figure 4.1, the two main architectural components of a network monitoring system are a *preprocessor* and a *detection engine*. The preprocessor prepares input network packets for analysis by the detection engine. For example, it normalises the packets to remove ambiguities, filters unnecessary packets, or reorders packets according to their sequence numbers. Once network packets have been pre-processed, the detection engine examines incoming traffic against known attack patterns. The detection engine performs pattern matching and content inspection using rules

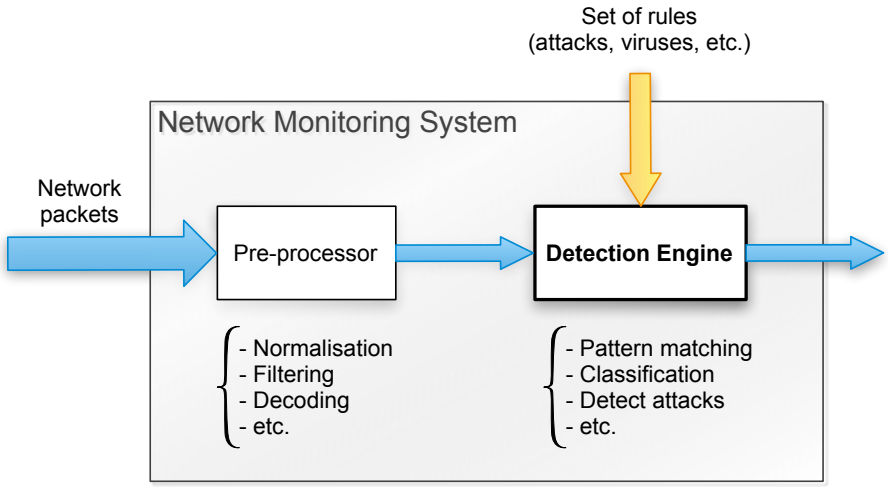


Figure 4.1: A generic network monitoring system that is comprised of two main components: a pre-processor and a detection engine.

describing malicious network traffic.

There have been many studies of packet analysis techniques [Roe99, KVB04, Pax98, ULF08]. For example, Monitor [KVB04] uses network rules to describe network behaviour and identify violations, such as known attacks and threats, by monitoring real-time network traffic. However, it applies verification at the end of the development life-cycle and its rule description language is not expressive enough to describe complex relationships between packets, which are associated with many network errors.

SNORT [Roe99], a popular open source intrusion detection system, performs packet inspection and content searching and matching to detect a variety of network attacks and probes. It contains thousands of rules describing network attack signatures. It is useful to detect attacks in real-time network traffic but cannot verify protocol implementations and struggle to detect implementation specific errors.

In terms of network traffic analysis, our solution is similar but traffic is collected in response to test packets generated using symbolic execution. This traffic and our rule language enable users to detect many hard-to-detect errors.

4.1.2 Rule Matching

In packet analysis, the header and payload of a packet are matched against a set of rules to detect specific patterns of attacks, payloads, viruses, etc. These rules can be derived from different origins, such as protocol specifications [ULF08] and a database of network attacks [Roe99]. Since these sources are usually written in a human language, such as English, they need to be re-written in a

Table 4.1: Main syntax of regular expressions

Syntax	Description	Example
ˆ	Matches the starting position of the input	ˆXY matches the input starts with XY.
*	Matches the preceding element zero or more times	X*Y matches "XY", "XXY", "XYY", etc.
.	Matches a single character	X.Y matches "XZY", "XAY", etc.
[ ]	Matches a single character within the brackets	$[XY]$ matches "X" or "Y".
\| | The choice OR operator | X\|Y matches "X" or "Y". |
+ | Matches the preceding element one or more times | XY+ matches "XY", "XYY", "XYYY", etc. |

formal language converting into a machine readable form. Traditionally, rules have been expressed as exact string match consisting of known patterns of interest. Regular expressions replace these fixed string patterns providing superior expressive power and flexibility [Roe99, YCD+06].

A regular expression gives a flexible means for describing a set of strings without explicitly enumerating all of them. Table 4.1 lists the syntax of regular expressions. Once rules are expressed as regular expressions, they can be converted into finite automata for pattern matching. Most implementations of pattern matching can be classified into two categories: Deterministic Finite Automata (DFAs) [KDY+06, SEJK08] and Nondeterministic Finite Automata (NFAs) [BSMV06].

A DFA is a finite state machine that recognises a regular expression, therefore accepting or rejecting finite input strings or symbols. When performing pattern matching, DFAs provide efficient matching because they transit deterministically from one state to another when reading a character. A NFA is also a finite state machine, but it operates nondeterministically, transferring from one state to several possible states. Although a NFA can be simulated with a DFA, NFA provide flexibility in terms of transitions of states over the same input character.

4.2 Packet Rule Validation

In this section, we give an overview of our approach for packet rule validation (PRV). PRV in SYMBEXNET is the process of validating captured packets with packet rules. It divides into three parts: rule extraction, packet filtering and packet rule validation.

The three steps are illustrated in Figure 4.2. At first, PRV needs to extract packet rules from the protocol specification (❶). Next, for each input packet stream, PRV performs packet preprocessing tasks such as filtering unnecessary packets (❷). Since monitored packet streams may contain

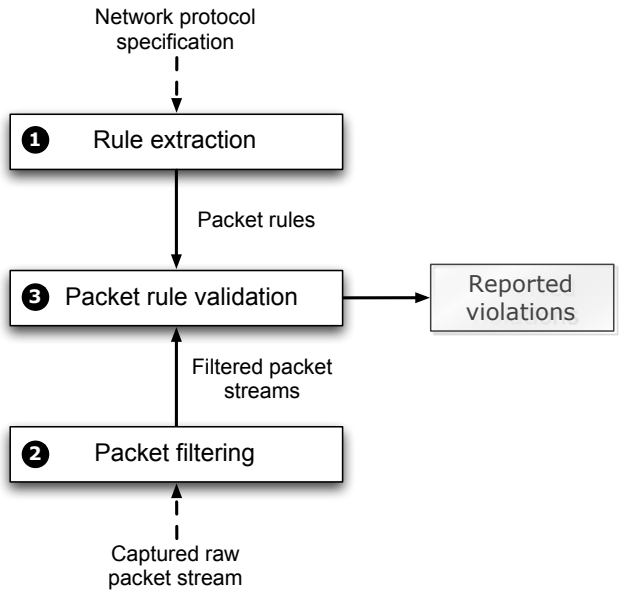


Figure 4.2: Procedure for packet rule validation.

incorrect traffic, pre-processing is required to eliminate these packets. PRV can remove incorrect packets based on not only the header information but also the payload.

Finally, PRV validates the filtered packets against the packet rules (❸). The filtered packets and packet rules are used as input to a rule-based packet analyser. The analyser checks the packets against the packet rules. If this detects a violation, the analyser reports the result with an error trace.

4.3 Rule Extraction

Next we describe how network packet rules can be extracted from protocol specifications. A protocol specification is a description of the intended behaviour of network entities and a set of logical rules for exchanging information between the network entities. Therefore, a set of rules can be extracted from the text of a network protocol specification. In many standards documents, words such as “MUST” and “SHOULD” are used to express requirements in the specification [Bra97]. For example, “MUST” has similar meaning to “REQUIRED” or “SHALL” and means that the statement is an absolute requirement. We find that phrases containing these words are good candidates for translation into formal rules.

In particular, there is a specification for “Keywords for use in RFCs to Indicate Requirement Levels” (RFC 2119) [Bra97], which defines keywords that authors of protocol specifications can use for requirements. These include “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHOULD”, “RECOMMENDED”, etc. Table 4.2 lists some keywords included in the RFC 2119.

Table 4.2: The keywords for describing the requirements in IETF standard documents [Bra97].

Keyword	Description
MUST	*"This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification."*
MUST NOT	*"This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification."*
SHOULD	*"This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course."*
SHOULD NOT	*"This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label."*
MAY	*"This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item."*

Consider how sentences containing such keywords can be used to contribute rules. For example, we can find the following requirement related to the "Query ID", which is used to identify a particular query message and a header field of a multicast DNS packet, in the RFC defining the Multicast DNS (mDNS) network protocol [CK10]:

> *"In unicast response messages generated specifically in response to a particular (unicast or multicast) query, the Query ID MUST match the ID from the query message."*

Since the "MUST" keyword is used, this requirement is a mandatory feature for all network daemons that implement this specification. The requirement states how an mDNS daemon has to set the Query ID in a response packet when answering using a unicast packet. If the daemon does not follow this behaviour—for example, by selecting a random value for the ID that does not match the ID from the query—the client may ignore the response packet. Therefore this requirement is a good candidate for translation into a packet rule.

Usually requirements to be included in protocol specifications address how to communicate with external network entities and how to manage internal states such as cache data, network parameters and protocol-specific data. Our packet rules refer to externally observable aspects of packets, thereby can be reused across different implementations of the same protocol. This means that not all phrases from specifications containing special keywords can be translated to rules. In particular,

we cannot translate requirements that do not specify externally observable behaviour into rules. For example, the following requirement from the mDNS specification cannot be described as a rule because it refers to internal state, i.e. registered services maintained by a daemon:

> *"A Multicast DNS Responder MUST NOT answer a Multicast DNS Query if the answer it would give is already included in the Answer Section with an RR TTL at least half the correct value."*

Although selecting requirements based on keywords from protocol specifications requires human intervention, users without protocol-specific knowledge can derive requirements. Once requirements are found, they need to be translated into rules using a packet rule language, as introduced in the next section.

4.4 Packet Rule Description Language

In order to check a protocol implementation for behaviour that violates candidate requirements, SYMBEXNET converts the requirements into packet rules, describing undesired behaviour using a *packet rule description language*. SYMBEXNET applies an approach using packet filters for detecting complex patterns of packet exchanges. Users convert selected requirements into packet rules in a high-level language with four operators, which are translated into low-level automata.

The main purpose of packet rule description languages is to filter intended network packets from a sequence of exchanged packets. There have been many studies and developed tools for such languages in order to check network behaviour. Examples include *Snort* [Roe99], *Monitor* [KVB04] and *Bro* [Pax98]. Most of these languages rely on Berkeley Packet Filter (BPF) [MJ93] rules to describe network behaviour and identify violations. BPF is a mechanism for filtering network packets at the data link layer. It allows a program to specify a filter in the form of a rule and capture packets from the network interface. However, this may be inefficient [IAIK02, BMG99] when describing complex relationships between packets, as associated with many network implementation errors.

SYMBEXNET uses a BPF-style filter language. Compared with other proposals for using BPF filters to analyse network behaviour, our packet rule description language enhances the degree of expressive capability through specific features. In SYMBEXNET, rules describing undesired packet exchange patterns are specified using four operators, which can reference the header field values of previously received packets and compare packets in different streams.

We start with introducing types of packet rules in the next section. After that, the detailed syntax of our language is described in Section 4.4.3.

```
1  [ruleExpr] ::= [packetExpr + ZeroOrMore(pop + packetExpr)]

2  [packetExpr] ::= [pname + {filterExpr}]
3  [pop] ::= [next(;) | union(|) | iteration(+)]
4  [pname] ::= [string]

5  [filterExpr] ::= [filter + ZeroOrMore(fop + filter)]
6  [fop] ::= ['and' | 'or']
7  [filter] ::= [name + op + value]

8  [op] ::= ['==' | '!=' | '<' | '>' | '<=' | '>=']
9  [value] ::= [hex | decimal | name]
10 [name] ::= [string] | [@+string]
```

Figure 4.3: Packet rule grammar written in as extended BNF.

4.4.1 Types of Packet Rule

Requirements stated in a protocol specification mainly describe packet formats and ways for exchanging a series of packets to provide a certain task. Packet rules try to detect undesired packet exchange patterns, which violate the requirements. Detecting violations of requirements is useful when the model of protocol implementations is incomplete or has a large number of states. At high-level, there are two types of requirements:

The first type describes the packet format. Requirements in this category specify necessary fields, permitted values for each field and their semantics. This type of requirements are translated into rules, which detect packets violating defined packet formats or specified field values in the specification. We refer to these as *intrinsic packet rules*. Examples of intrinsic rules include: detecting packets match a specific character that must not be included or detecting packets with a wrong binary bit in a given field.

The second specifies allowed communication behaviour of a network protocol implementation with external network entities. This type of requirements are translated into rules that detect a sequence of packets that is not allowed in the specification. We call these *sequential packet rules*. Examples include: finding an incorrect response packet to a query packet, which contains invalid field values.

4.4.2 Grammar of Packet Rule Language

In this section, we show the grammar used for the packet rule language in SYMBEXNET. We specify the abstract syntax for the packet rules using an extended BNF (EBNF) syntax.

Figure 4.3 lists the grammar for describing packet rules. In the notation, ::= means "is defined to be". If there are a number of options that can be applied to one particular rule, the option

is delimited with the '|' symbol. To provide an unambiguous way of composing packet rule expressions in SYMBEXNET, we use a default precedence of operators, i.e. AND takes precedence over OR.

Packet rule `ruleExpr` (line 1) can be expressed using one or more packets `packetExpr` (line 2) with packet operators `pop` (line 3). Each packet expression in a packet rule is described in terms of its name `pname` (line 4) and the packet filters `filterExpr` (line 5). Packet filters are an unordered collection of one of more filters `filter` (line 7), which are name/value pairs where a `name` (line 10) is a string for the name of a packet field and a `value` (line 9) is a `hex`, `decimal` or `name` value.

Rules are parsed by SYMBEXNET. Parsed rules are then used to build a corresponding NFA (cf. Section 4.6).

4.4.3 Rule and Operators

In this section, we introduce the grammar and syntax of our packet description rule language for describing packet exchange patterns. Our high-level language provides a rich syntax for building descriptive rules, as well as additional modifiers that can enrich the expressiveness of the whole rule expression. The rule language describes violations of packet requirements and consists of expressions of the following form:

$$packetExpr = pkt\{\Sigma_{filters}\}$$

where pkt is the name of a packet and $\Sigma_{filters}$ is a set of packet filter predicates. A packet filter predicate represents the possible values of the corresponding *fields* in packets that match this filter. We have introduced some of the fields that are part of DNS in Figure 2.2 and DHCP in Figure 2.4 (cf. Section 2.1). The set of packet filter predicates are sequences of valid packet filters joined by the logical operators AND/OR. The modifiers ANY and ALL specify that a predicate has to match at least one or all fields, respectively, if multiple fields with the same name exist. Nested field names are divided by dots (.). In addition, we introduce a set of operators and modifiers to express network specific features such as ignoring packets that do not satisfy a given filter condition.

The Filter Operator

The **filter operator** `p` detects packets `p` that satisfy a predicate θ. The predicate θ allows the operator to filter some packets satisfying the predicate and to fail on others. Consider the following packet description that filters a `query` packet:

```
1 query {src_ip != 224.0.0.251 AND flag.QR = 0x00 AND questions != 0x00}
```

This filter matches a query packet (`flag.QR=0x00`) that is not from the multicast IP address 244.0.0.251 and has more than one question block (`questions!=0x00`). It ignores all other packets that do not satisfy these conditions.

The filter operator is suitable for describing intrinsic rules because it can be used to detect a packet that has specific field values. In order to describe sequential rules, which require to specify multiple packets in a rule, we need operators that can be used to connect more than one packet. For this purpose, we introduce three rule operators: *next* (;), *union* (|) and *iteration* (+). Rule expressions can be built recursively using these three operators.

The Next Operator

If there are irrelevant packets on the network, these packets must be ignored. The **next operator** `p1;p2` detects the next occurrence of packet `p2` after packet `p1`, ignoring any intermediate packets that do not satisfy the filter predicates for `p2`. This operator implies that the timestamp of `p1` is guaranteed to be earlier than the timestamp of `p2`.

The Union Operator

There exist cases when the occurrence of one or more packets out of a set of packets needs to be detected. The **union operator** `p1|p2` matches a choice of packets `p1` or `p2`.

The Iteration Operator

When multiple occurrences of the same packet need to be detected, the **iteration operator** $\mathtt{p}_{\theta}{+}n$ detects n consecutive packets `p` that satisfy θ. Packets not satisfying θ are skipped. Since there are requirements that specify multiple occurrences of the same packet during a given time slot, the iteration operator is often used with the *Timeout* operator, which will be described below.

ANY and ALL

The *ANY* and *ALL* modifiers can be used with a packet filter predicate, which represents the possible values of the corresponding *fields* in packets. The ANY modifier takes a set of packet fields as input and evaluates to `True` if a predicate matches any of these fields. The ANY modifier is useful in situations, in which firing of a rule depends on the occurrence of a value in any given packet field at least once.

The ALL modifier takes a set of packet fields as input and requires that a predicate matches all of the fields. The ALL modifier is useful when a rule has to be associated with the occurrence of a value in all packet fields.

Variable binding

Often a desired packet can be detected based on specific field values of other packets received before. In this case, users can use pattern variables to extract field values from received packets. Using the variable binding operator `@`, fields from previously detected packets can be stored and referenced in subsequent filter expressions. If there exist a packet `p` that previously occurred, a field name of the form `@p.field` refers to the field name `field` of the previous packet `p`.

For example, the rule below shows how the variable binding operator @ is used to refer to a specific field value:

```
1 query{src_ip != 224.0.0.251 AND flag.QR = 0x00 AND questions != 0x00} ;
2 resp {dst_ip = @query.src_ip AND flag.QR = 0x80 AND id != @query.id}
```

Consider the packet filter of the `query` packet (line 1). It matches a DNS query packet (`flag.QR = 0x00`) that is not from the multicast IP address `244.0.0.251` and has more than one question block (`questions != 0x00`). The next operator (;) at the end of `query` ignores packets that do not satisfy these filter conditions. In the `resp` packet filter (line 2), two variable bindings are used—$@query.src_ip$ and $query.id$. Both refer to the value of the corresponding fields in the `query` packet. In particular, $@query.src_ip$ is used to detect a packet response to the source IP address of the `query` packet while $query.id$ tries to discover a packet not using the same ID as the `query` packet.

Timeouts

It is important to include time when describing packet sequences because many aspects of a network protocol are driven by timers and timeouts. To describe timing-related requirements, each packet contains a virtual field called `ts` that represents the timestamp at which the packet was received by the target implementation.

Suppose that the previous `Query ID` example has another associated requirement that the response message *MUST* be generated within at most 10 ms. The violation of this rule can be encoded as follows:

```
1 query{src_ip != 224.0.0.251 AND flag.QR = 0x00  AND questions != 0x00} ;
2 resp {dst_ip = @query.src_ip AND flag.QR = 0x80
3   AND ANY data.answer(name = @query.question.name) AND id != @query.id
4   AND ts >= @query.ts+10}
```

After matching a query packet (line 1), the rule discards all response packets that do not answer the question in the query packet within 10 ms (line 4). Most network protocol patterns with a general query/response model can be described using the operators described above.

4.5 Examples

Using the operators and modifiers introduced so far, we show how the behaviour of network protocol implementations can be modeled in SYMBEXNET using the packet rule language. We use a simplified syntax to make the rules readable.

1. *"Detect an ICMP ping packet (ICMP ECHO REQUEST) with TTL equal to 0"* from RFC 792—Internet Control Message Protocol (ICMP) [Pos81a]. Note that the IP protocol type is set to "ICMP" (`1`) to indicate that the packet is to be handled by the remote end system's ICMP client.

```
1 icmp-pkt {protocol_type = 0 AND TTL = 0};
```

2. *"Detect a TCP packet with `TTL` <= 100 destined to web server `155.168.1.20` at port 80"* from RFC 793—Transmission Control Protocol [Pos81b].

```
1 tcp-pkt {dst_ip = 155.168.1.20 AND dest_port = 80 AND TTL <= 100};
```

3. *"Discover any packet that originates from an mDNS server of `155.168.1.10` and contains a 'byte order mark"'* from the mDNS specification [CK10]. The byte order mark is a Unicode character used to signal the endianness of a text file or stream and its code character is `U+FEFF`. In addition, multicast DNS names *MUST NOT* contain a byte order mark.

```
1 mdns-pkt {src_ip = 155.168.1.10 AND src_port = 5353
2   AND ANY dns.data(srv_name = U+FEFF)};
```

4. *"Discover an mDNS answer message that is not directly sent to the client via unicast in response to a received mDNS query with not the mDNS source port number"* from the mDNS specification [CK10]. If the source UDP port in a received mDNS query is not port `5353`, this indicates that the client originating the query is a simple client that does not fully implement all mDNS functionality. In this case, the mDNS server *MUST* send a UDP response directly to the client, via unicast, to the query packet's source IP address and port. Also note that all mDNS messages MUST be sent to the mDNS multicast address

224.0.0.251 and its port 5353 (0x14e9).

```
1 mdns-query {src_port != 0x14e9 AND dest_port = 0x14e9
2   AND dest_ip = 224.0.0.251 AND flags.query = 0x00}  ;
3 mdns-resp {flags.query = 0x80 AND data.srv_name =
4   @mdns-query.data.srv_name AND dest_ip != @mdns-query.src_ip}
```

5. *"Detect any DHCPNAK message that is not sent to the broadcast address 255.255.255.255, upon receiving a DHCPREQUEST message with giaddr equal to zero"* from RFC 2132 [AD97]. In all cases, when giaddr is zero, the server broadcasts DHCPNAK to the broadcast address. DHCP uses UDP port 67 for sending data to the server, and UDP port 68 for data to the client. The xid field is used by the client to match incoming DHCP messages with pending requests.

```
1 dhcp-request {dest_port = 67 AND options.type = 0x01
2   AND giaddr = 0.0.0.0} ; dhcp-nak {dest_port = 68
3   AND xid = @dhcp-request.xid AND options.type = 0x06
4   AND dest_ip != 255.255.255.255}
```

As shown by the above examples, requirements from protocol specifications can be encoded into packet rules using our packet rule description language. The process of deriving rules is relatively easy because specific keywords are used when mandatory requirements are stated in protocol specifications.

4.6 Rule Implementation and Validation

Network packet rules derived from a specification must be validated. Pattern matching [YKL04, ADGI08] is a popular technique for checking packet rules. This section provides an explanation of our method for validating packet rules. Packet rules are verified using non-deterministic finite automata (NFAs). We use an event model that is similar to the one found in event processing systems [SMMP09] because it provides a mechanism for detecting complex event matches through the use of a high-level event query language.

A SYMBEXNET NFA model is defined as a tuple $(S, \Sigma_{pkts}, T, s_0, s_d)$ where

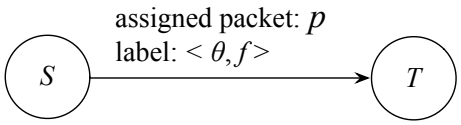


Figure 4.4: High level packet rule automaton.

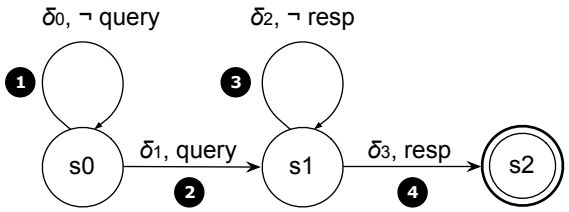


Figure 4.5: An example of rule automaton.

S:	is a finite set of states of the automaton;
Σ_{pkts}:	is an input packet stream into the automaton;
T:	$S \times \Sigma_{pkts} \rightarrow P(S)$ is a transition function. Each transition is labeled with packet filters. $P(S)$ is a power set of S. Equivalently, T can be presented as a relation, i.e. a subset of $(S \times \Sigma_{pkts}) \times S$;
s_0:	$s_0 \in S$ is the start state; and
s_d:	$s_d \in S$ is the set of final states.

An automaton operates as follows. Each NFA state is assigned a name and an input packet. All the outgoing edges of a state read that input packet. Figure 4.4 shows a high level illustration of a packet rule automaton. Suppose an automaton instance is in state S with assigned packet p. Each edge, between states S and T, is labelled with a pair $\langle \theta, f \rangle$ where θ is a predicate and f is a transition function returning the next state T. Let a packet e satisfies predicate $\theta(p, e)$. As a result, the NFA transitions non-deterministically to the next state T, as specified by the transition function f and stores packet p in order to refer back to its field values later.

To show how packet rules are implemented and validated, let us consider an example requirement, "the response packet for a query must have the same ID and data values as the original query". Here the query has the `flag` field with value `0x08`, and the `id` of the response packet must match that of the query packet. This requirement can be expressed through the following rule:

```
1 query{ flag = 0x08 } ; resp { id = @query.id AND data != @query.data }
```

The corresponding automaton for the rule is shown in Figure 4.5. Now we suppose that we have a sequence of received packets in the order $p1$, $p2$, $p3$ and $p4$. The packets are described in Table 4.3.

Table 4.3: Input test packets used in Figure 4.5.

Step	Packet	Before	After	Field values
❶	$p1$	$s0$	$s0$	`id=0x35`, `flag=0x01`, `data=abc`
❷	$p2$	$s0$	$s1$	`id=0x45`, `flag=0x08`, `data=sym`
❸	$p3$	$s1$	$s1$	`id=0x36`, `flag=0x01`, `data=net`
❹	$p4$	$s1$	$s2$	`id=0x45`, `flag=0x01`, `data=sym`

For an incoming packet, the state of the automaton after processing is decided. Initially, the start state $s0$ is active by default. The automaton processes incoming packets as follows:

1. $p1$ **arrives (label ❶).** The automaton checks if it satisfies δ_1, a predicate for the $query$ packet leading to the next state $s1$. $p1$ does not satisfy δ_1 but δ_0, therefore the automaton stays in the same state $s0$.
2. $p2$ **arrives (label ❷).** Since $p2$ satisfies δ_1, the automaton moves to the next state $s1$.
3. $p3$ **arrives (label ❸).** Similarly, the automaton stays in the state $s1$ after receiving $p3$ because it does not satisfy δ_3.
4. $p4$ **arrives (label ❹).** The arrival of $p4$ transitions the automaton to the final state $s2$.

Since here packet rules describe violations of requirements specifying desired packet exchange patterns, arriving at the final accept state means that a given sequence of packets violates the requirement.

4.7 Related Work on Rule-based Analysis

Rule-based analysis is a technique that uses rules to verify programs and has gained traction in the validation of network protocol implementations and the detection of intrusions and vulnerabilities [Roe99, KVB04]. Tools such as Pistachio [ULF08] define network rules, which describe what should happen when an implementation receives a packet, as derived from a specification. Such systems bridge the gap between specifications and implementations, but they achieve only low code coverage and struggle to detect rare errors because their rules are limited to single packet exchanges. SYMBEXNET uses symbolic execution to increase code coverage and provides a high-level packet rule language based on an expressive automata model. While Pistachio's language could be used with SYMBEXNET, our packet rules can describe more complex sequences of packets compared to Pistachio's single input-output patterns. Furthermore, SYMBEXNET can detect interoperability problems, which is not supported by other approaches using rule-based analysis including Pistachio.

Event processing systems can detect complex event patterns using pattern matching techniques, e.g. state automata [BDG+07] or event trees [MSS97]. They use high-level SQL-like query languages, which are designed to support event pattern matching. In these systems, NFAs are the most widely used method to implement queries for detecting occurrences of specific patterns. As automata-based models provide sufficient expressiveness for detecting complex sequences, we use automata to find violations in packet rules. Our packet rule language is similar to the one used by the NEXTCEP system [SMMP09] but is extended with primitives suitable for describing network packet exchange patterns.

4.8 Summary

In this chapter, we described a verifiable protocol specification that assesses the correct behaviour of a network protocol implementation automatically. Since many network protocols state compulsory requirements in their specifications using keywords, we extract a set of rules for a network protocol from its protocol specification. Packet rules defining behavioural violations are expressed using a packet rule language and implemented as non-deterministic finite automata, which match incorrect sequence of packets. Since packet rules capture undesired behaviour of a protocol implementation instead of an entire specification, they can lead to a concise description.

We introduced packet rule operators and their semantics, described by automata that are an extension of traditional automata. We enhanced the language with additional features such as timeouts and variable binding in order to model behaviour occurring in network protocol implementations. This also improves the readability of our packet rules and makes the language well-suited for describing network behaviour, compared to previous languages [SMMP09]. We also gave examples illustrating how invalid patterns of packet exchanges can be stated using our packet rule language. Finally, we provided a mechanism for matching using NFAs.

CHAPTER 5

Symbolic Interoperability Testing

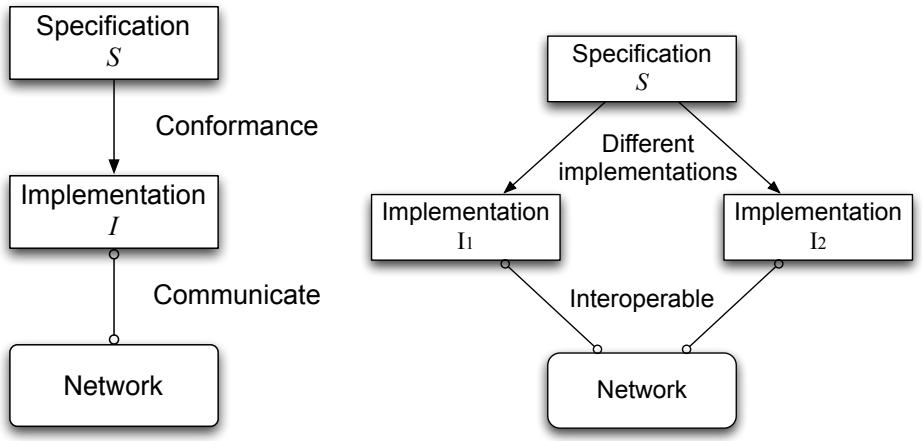
THIS chapter introduces a methodology for checking interoperability between multiple network implementations of the same protocol specification, including a way to derive test input packets for triggering interoperability issues. In the domain of communication networks, *Interoperability Testing* (IOT) is considered as an essential step toward ensuring a correct implementation that operates as specified in the specification [MRW03]. Although many researchers have investigated automated methods for IOT [KSK00, SKKR03, DV08], there still exist many challenges, such as the difficulty in covering all interoperability issues and the necessity of human intervention [KSK00, VRA+07].

In this chapter, we propose a method called *Symbolic Interoperability Testing* (SIOT), which automatically generates test input packets for IOT using symbolic execution and checks interoperability of an implementation using a rule-based packet inspection technique. An advantage of our approach is that we can derive high quality test input packets for IOT automatically from protocol implementations, without deep knowledge of the underlying protocol.

In Section 5.1, we provide an overview and a definition of interoperability testing. After that, the design and implementation of the interoperability checking in SYMBEXNET is presented in Section 5.2. We illustrate how test input packets for IOT are generated and tested with multiple protocol implementations in Section 5.3. Methods for packet filtering and rule-based interoperability checking are discussed in Section 5.4. We conclude this chapter with related work in Section 5.5 and a summary in Section 5.6.

5.1 Overview

Network protocol specifications are usually written by standard organisations such as 3GPP and implemented by different manufacturers. Its standard specifications are the outcome of the agreement between many companies, the compromise of various issues often leads to alternative options and recommendations within a standard [BU91]. Furthermore, standards are typically written in



(a) Conformance Testing (b) Interoperability Testing

Figure 5.1: (a) Conformance testing ensures that an implementation complies with its protocol specifications; (b) Interoperability testing ensures that different implementations work as expected

natural languages, and vendors can easily have different interpretations that lead to divergent implementations. Additionally, the complexity of network protocols makes it difficult to deliver a correct protocol implementation without any interoperability problems.

Two testing techniques can be used to check the compliance of implementations with a network protocol: *conformance testing and interoperability testing*. As shown in Figure 5.1, conformance testing is used to check that the behaviour of an individual implementation complies with its protocol specification. Since conformance testing, which checks an implementation against the specification, is considered a necessary step before operation, it has been studied and formalised [DSU90, Tre99, BFN+05]. Conformance testing can show only the presence of described functionality and cannot assure the correct communication with other network entities. Therefore, interoperability testing, which checks that multiple implementations are interoperable with each other, is desirable to complement conformance testing. The importance of performing both conformance and interoperability testing is well known in the industry [BU91].

Definitions of Interoperability and Interoperability Testing.

Interoperability is a relative term, understood according to its context. For example, ISO/IEC[1] defines interoperability as the capability to communicate among various functional units, and IEEE[2] focuses more on the ability to work among systems. Even within the same domain, different definitions of interoperability exist. ATIS[3] makes the definition more specific on exchanging information while others provide a broad concept of interoperability including the end user perspective.

[1] International Organisation for Standardisation (ISO)/International Electrotechnical Commission (IEC)
[2] Institute of Electrical and Electronics Engineers
[3] Alliance for Telecommunications Industry Solutions

Different from conformance testing, IOT is the process of testing implementations from multiple vendors by interacting in such a manner as to exercise the network protocols under test. The entities under IOT must be set up and synchronised [KSM96]. In particular, IOT is helpful in the early stages of developing a new standard protocol because it can used to evaluate the status of the protocol. Once an implementation is mature, IOT cannot detect new bugs and becomes less valuable [MRW03].

Since we verify network implementations referring to network protocol specifications, the term *interoperability* is defined in this thesis as follows:

Definition 5.1. (Interoperability) *"the capability of the network implementation to interact with other implementations of the same protocol specification."*

We detect interoperability problems of network protocol implementations by comparing their behaviour to test input packets. Thus *Interoperability testing* based on Definition 5.1 is defined as follows:

Definition 5.2. (Interoperability Testing) *"the process of automatically generating a set of interoperability test packets, executing the implementations under test (IUT) with the generated test packets and comparing the behaviour of the implementations to determine whether they are interoperable with each other."*

5.2 Symbolic Interoperability Checking

In this section, we introduce a method for checking the interoperability of protocol implementations using SYMBEXNET. Similar to prior work [KSK00, SKKR03, DV08], we aim to provide an automated method for generating interoperability test packets. However, in order to overcome the limitations of the existing IOT technologies, we provide several new features. First, we derive test packets from the source code so that users can build an interoperability testing model, even when there is no specification for a target protocol. Since test packets are generated from the source code, the method can cover both the control and data parts of a protocol. Most prior work dealt with test case derivation only for the control part of protocols because they focused on observing the interface behaviour. Furthermore, interoperability test packets derived using our approach are the result of running actual code. This enables us to check deeper interoperability behaviour, such as complex sequences of interoperations and behaviour that depends on accurate value information.

In order to generate interoperability test cases automatically while avoiding the above problems, we present a novel interoperability testing method called *Symbolic Interoperability Testing* (SIOT). Unlike conventional IOT approaches using abstract specifications, SIOT derives test cases directly from multiple implementations of a protocol. For test case generation, SIOT uses symbolic execution to explore all possible execution paths of the implementations exhaustively. SIOT checks the

different behaviour of implementations for each test sequence in order to examine if there are any interoperability problems. This is done using rule-based packet analysis that compares input/output packets and detects deviations.

5.2.1 Overview

The four goals of SIOT are (1) *automation*, (2) *flexibility*, (3) *implementation-based checking* and (4) *scalability*.

Automation. The first goal is to provide a mechanism for automatically generating a set of test cases for IOT. Although several IOT methods [KSK00, SKKR03] have targeted automated test case generation, in most cases an abstract model of a protocol, such as finite state machine, is extracted manually from its specifications before generating test cases, then requiring human intervention.

Flexibility. The second goal of *flexibility* is to provide a process that can be flexibly applied to other protocols. The process of building an abstract model from specifications requires expert knowledge of a protocol and therefore, when testing implementations of a new protocol, testers need to be familiar with the protocol, which takes time.

Implementation-based checking. The third goal of SIOT, *implementation-based checking*, is to explore implementations under test and check their interoperability instead of an abstract model. Usually test cases derived from an abstract model of protocol specifications can only check limited interoperability functions that are described explicitly in the specification. Implicitly addressed interoperability features may be missed by these test cases. An important objective of SIOT is to accomplish implementation-based interoperability checking between multiple implementations and not to miss interoperability problems.

Scalability. After deriving test cases for IOT, they are applied to IUTs. In practice, this is performed in an ad-hoc manner by connecting available IUTs together at a central location, at which testing facilities have been provided. Figure 5.2 shows a typical configuration for interoperability testing that interconnect a number of implementations from different vendors in a testing network [MRW03]. The figure illustrates a situation, in which implementation 1 is tested with other implementations 2, 3 and 4.

For example, vendor `A` and `B` may develop implementations `1` and `2` based on the same protocol specification `S`. Interoperability test cases are then executed on both implementations `1` and `2` to examine if they are interoperable with each other. In order to confirm `1` and `2` are interoperable, a test operator has to perform the testing twice. First, test cases are applied to `1` and its corresponding responses are analysed for *the interoperability of 1 with 2*. Next, the operator applies test cases to `2` to check for *interoperability of 2 with 1*.

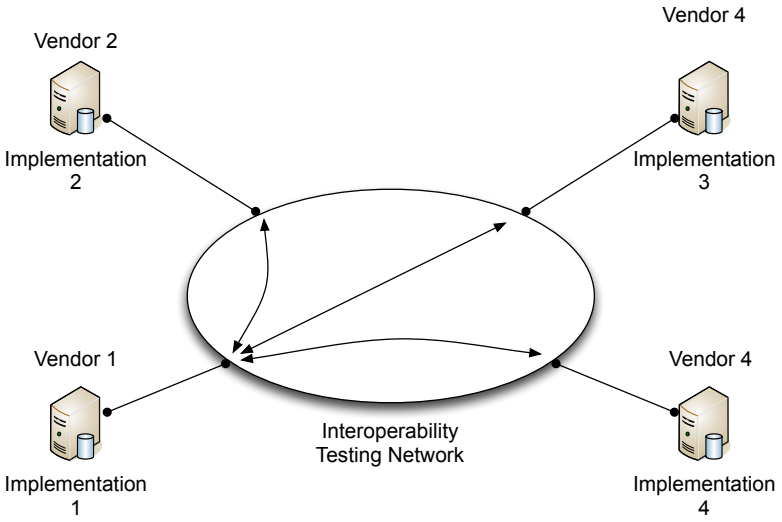


Figure 5.2: Typical configuration for interoperability testing in industry.

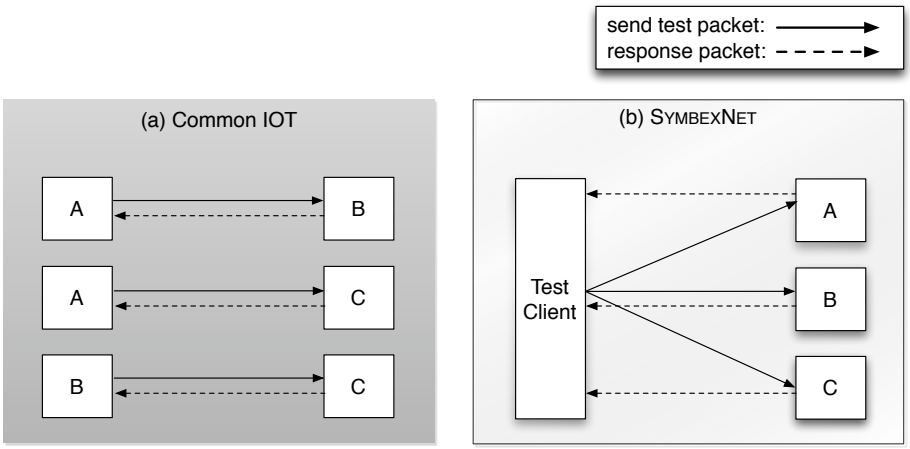


Figure 5.3: Comparison of (a) common IOT and (b) SYMBEXNET.

Since each implementation under test needs to be tested with all other implementations, the number of required tests is $n(n-1)$ where n is the number of IUTs. This gives an important requirement to SIOT for developing an interoperability testing method that scales better than prior methods in terms of the number of tested implementations.

Approach. Now we take a brief look at the approach used in SYMBEXNET to achieve interoperability testing. Similar to other existing IOT approaches [CTCC98, BG01], SIOT integrates white-and black-box testing techniques. It first applies a white-box approach to generate test packets for IOT using symbolic execution, which explore the state space of the implementations. Next, black-box testing is performed by providing a series of generated test input packets to the implementations and observing network traffic to check for interoperability problems.

Current IOT approaches typically use two or more interconnected implementations from different vendors. Figure 5.3 (a) describes such an approach. Each implementation is connected to other implementations under test and examined for its interoperability with them. In this approach, the participation of all vendors implementing the same protocol is an important factor for success. Instead of connecting each implementation to all others, SYMBEXNET uses a test client that performs interoperability testing tasks, as shown in Figure 5.3 (b). Since SYMBEXNET strictly controls the testing environment by ensuring the same network conditions such as order of input and output packets, testing is repeatable, and therefore IOT can be performed independently. This can significantly reduce the costs for IOT and enables to perform IOT more efficiently.

To validate interoperability between target implementations, SIOT provides a checking method that is simple and easy to apply. It exploits the fact that the implementations realise the same protocol specification. Then they are supposed to behave exactly in the same way for any inputs. To check interoperability, SIOT analyses input/output packets and detects behavioral inconsistencies between target implementations. Any inconsistencies are considered a potential interoperability problem and reported.

5.2.2 Formal Model of Interoperability Relation

To define an interoperable relationship formally, let I_1 and I_2 be two different implementations of a network protocol. An *IOT* model is a tuple $M = (\Sigma_I, \Sigma_O, \Sigma_S, \Sigma_T)$ where Σ_I, Σ_O, Σ_S and Σ_T are finite sets of input packets, output packets, states and transitions, respectively. Each transition t in the set Σ_T is a 4-tuple

$$t = (ss_t, se_t, \sigma_t, \omega_t)$$

where ss_t, se_t, σ_t and ω_t are a start state, end state, input packet and output packet, respectively. Here, Σ_I is a set of test input packets generated using symbolic execution for interoperability testing. Since SIOT performs symbolic execution on both I_1 and I_2, Σ_I can be decomposed as follows: $\Sigma_I = \Sigma_{I_1} \cup \Sigma_{I_2}$ where Σ_{I_1} (resp. Σ_{I_2}) is the set of generated test packets from I_1 (resp. I_2).

Each test input packet for IOT is tested with both implementations and the outputs are monitored. Therefore, we can also decompose Σ_O as follows: $\Sigma_O = \Sigma_{O_1} \cup \Sigma_{O_2}$ where Σ_{O_1} (resp. Σ_{O_2}) is the set of output packets from I_1 (resp. I_2). We define $In(\sigma)$ and $Out(\omega_\sigma)$ as a receiving input packet σ and the corresponding response output packet ω, respectively.

Initially, the state machine is in an initial state $s_{init} \in S$. Let us assume that the machine is in state ss_t. Upon receiving an input packet $In(\sigma_t)$, the machine follows the transition t, sending an output packet $Out(\omega_{\sigma,t})$ and moving to state se_t. Since SIOT compares observed outputs to determine if

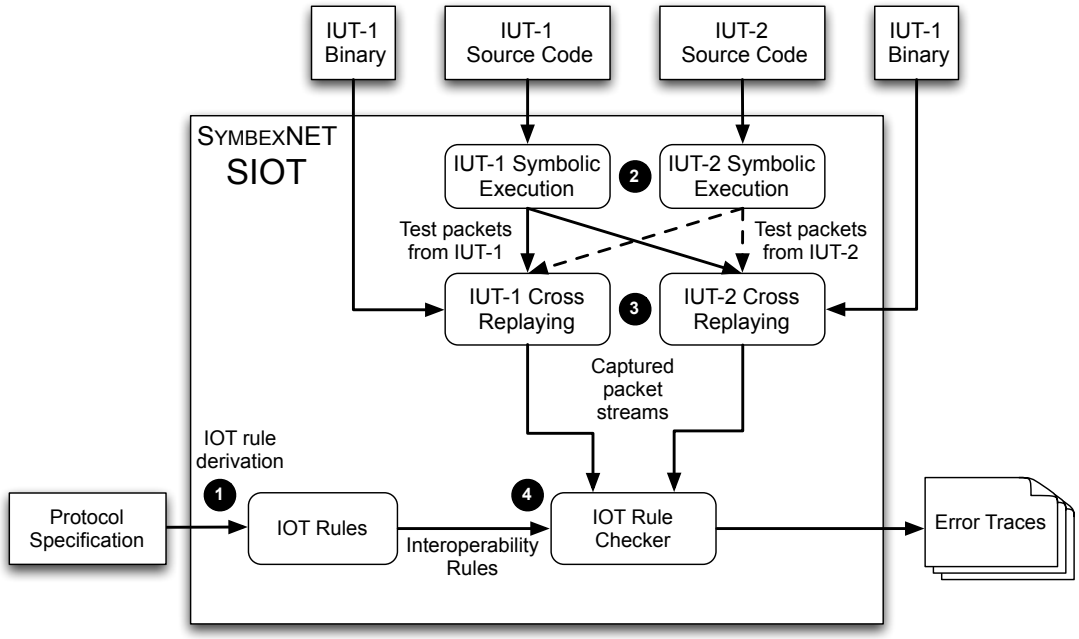


Figure 5.4: Overview of SYMBEXNET SIOT system architecture and its steps for checking interoperability for implementations of IUT-1 and IUT-2 the same protocol specification.

implementations are interoperable, we introduce a formal definition of an *interoperability relation (iot-rel)*. Let $\sigma \in \Sigma_I$, then the interoperability relation between I_1 and I_2 can be defined as follows:

Definition 5.3 (Interoperability relation *iot-rel*)**.** I_1 iot-rel $I_2 \iff \forall \sigma \in \Sigma_I$*, the following conditions are satisfied:*

- $In_{I_1}(\sigma) \rightarrow Out_{I_1}(\omega_\sigma)$ *and the state transition to* se_{I_1}*;*
- $In_{I_2}(\sigma) \rightarrow Out_{I_2}(\delta_\sigma)$ *and the state transition to* se_{I_2}*; and*
- $\omega = \delta$ *and* $se_{I_1} = se_{I_2}$.

Informally, I_1 *iot-rel* I_2 if and only if (*i*) the output packet and the end state of I_1 for a given test packet σ is the same as the state of I_2; or (*ii*) both I_1 and I_2 do not respond for a given test packet σ.

5.2.3 Symbolic Interoperability Testing

Based on the approach introduced above, we explain the main steps of SIOT as shown in Figure 5.4. SIOT takes four steps to check interoperability, as labeled in the figure:

Step (1): Creation of IOT rules. The first step is to derive interoperability rules from a protocol specification. IOT rules are created to compare fields of response packets from different implementations. The packet rule language introduced in Chapter 4 is extended to support the comparison

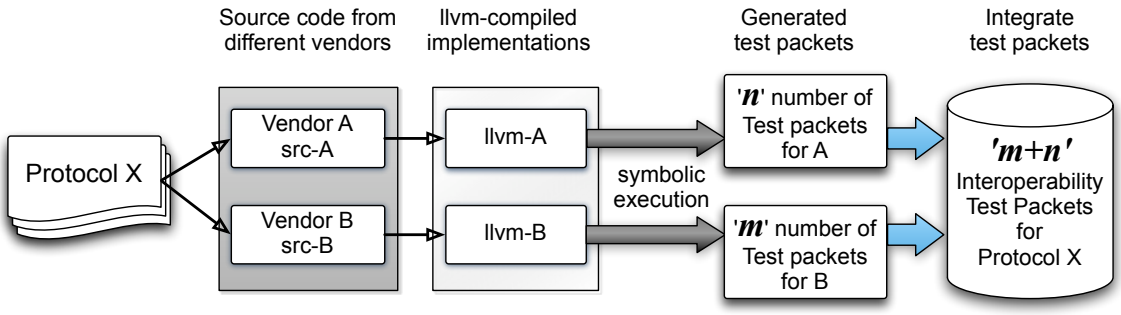


Figure 5.5: Approach for generating interoperability test packets for protocol X.

of response packets from different implementations.

Step (2): Generation of test packets for IOT. To check the interoperability between multiple implementations of the same protocol specification. SIOT relies on a set of test input packets that can detect interoperability problems. Using symbolic execution, as described in Section 3.4, SIOT explores the source code of each IUT and generates test input packets from the implementations, IUT-1 and IUT-2.

Step (3): Cross replay of test packets for IOT. The test packets generated for IOT are replayed not only with their source implementation but also with other implementations of the same protocol. As described in Section 3.4, each test packet is sent to an IUT in the same testing environment. All exchanged input/output packets are recorded.

Step (4): Interoperability checking. Since the implementations are based on the same specification, they are supposed to behave in the same way for inputs with specific field values, i.e. generate the same response packets. In this step, the captured input and output packets of both implementations from the previous step are compared by the IOT rules from step (1). SYMBEXNET translates the IOT rules into NFAs and runs captured input/output packet streams against each NFA to check whether they behave identically. For each divergent behaviour, SIOT reports an interoperability error.

5.3 Generation and Execution of Interoperability Test Packets

In this section, we discuss the generation of test packets for IOT and their cross replay in detail. SIOT is a subsystem of SYMBEXNET and shares many features, such as the generation of test packets from the source code of implementations and the replay of generated test packets with native implementations.

Figure 5.5 shows our approach for generating interoperability test packets. For interoperability checking, SIOT takes as input the source code of multiple implementations of the same protocol

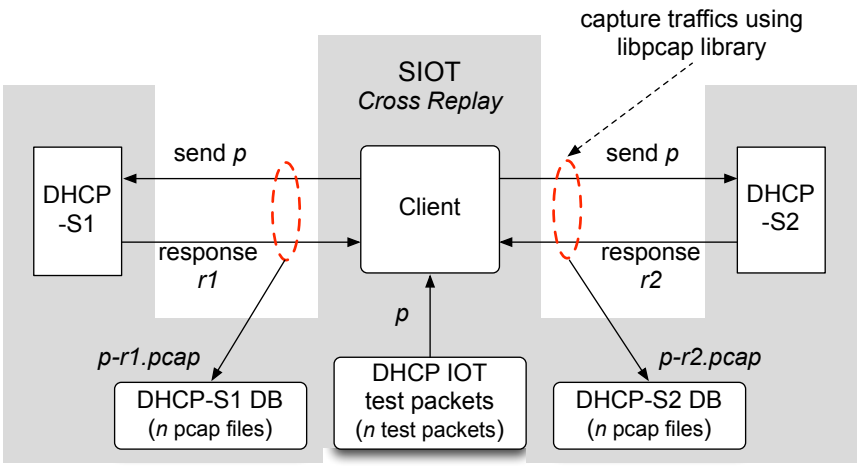


Figure 5.6: A configuration for cross replay in SIOT.

developed by different vendors instead of a single implementation. According to the symbolic execution method described in Section 3.4, SIOT compiles each source code into LLVM bitcode and then explores all code paths in each implementation, generating test packets.

For example, as interoperability testing compares multiple implementations of the same protocol specification, SIOT applies symbolic execution to both `llvm-A` and `llvm-B`. After generating the test packets from one implementation, SIOT prepares the same network environment for test case generation for the others. The generated test packets from each implementation are stored in an internal database maintained by SIOT.

We now explain how generated test packets are used in SIOT to check interoperability. In SIOT, interoperability testing is conducted using a different model. After test input packets are generated from each network protocol implementation, the test packets are replayed with not only their source implementation but also with the other implementations of the same protocol, which we call *cross replay*. The *cross replay* process is the same as the replay process introduced in Section 3.4, running unmodified implementations and capturing all generated network traffic. Since each test input packet is replayed on all IUTs, it generates n packet streams. If each IOT has m test input packets, *cross replay* generates $m \times n$ packets. Since rules that are used for IOT checks observe differences in response packets from different implementations for the same input packet, it is important to perform *cross replay* under the same network conditions, such as using the same port number in the generated test packets.

Figure 5.6 gives an example of SIOT of two DHCP implementations, `DHCP-S1` and `DHCP-S2`. Test packet `p` is one packet of n generated packets for IOT. As described in Section 3.4, SYMBEXNET first starts `DHCP-S1` for replay, sends the test packet `p` to `DHCP-S1` using the `Client` and then waits for the corresponding response `r1`. SIOT captures network traffics (input and output packets) between `DHCP-S1` and `DHCP Client`. After finishing the test replay for `p` with `DHCP-S1`, SIOT

does the same replay with `DHCP-S2`, again recording the traffic. The recorded traffic from both implementations for the same test packets is compared to check for interoperability.

5.4 Rule-based Interoperability Checking

To determine the interoperability between the IUTs, SIOT analyses the observed input and output packets for the test input packets, which are called *packet streams* (cf. Section 2.1). For this analysis, we extend the packet rule language to detect different behaviour among packet streams and introduce rules for checking interoperability.

5.4.1 Interoperability Rules

Interoperability rules detect differences between two or more packet streams by comparing response packets, appearing in each packet stream, for the test input packets. Interoperability rules inherit the syntax of *sequential packet rules* described in Section 4.4.1 but include an identifier that refers to a specific packet stream among multiple streams.

Stream identifier. Since interoperability checking compares packets from more than one stream, the source of each packet needs to be identified in rules for checking interoperability. We introduce *stream identifiers* to enable SIOT to compare fields of output packets from different streams.

A packet filter that is associated with a specific stream has a prefix `S` followed by the number of the stream. For example, `S1.p1.flags` refers to the field name `flags` of packet `p1` from stream `S1`. Packet filters without a stream identifier are used as common filters, which are applied to all streams, while packet filters with a stream identifier are only used to select a packet from the stream specified by the stream identifier.

Figure 5.7 shows how streams are structured and compared in SIOT using stream identifiers. A test client exchanges several packets, $p1$-$p4$, with IUT-A and IUT-B to configure the testing environment, for example, establishing a connection on a specific port number, followed by sending a test packet $p5$ to both IUTs. Upon receiving $p5$, the IUTs respond with $p6$-a for IUT-A and with $p6$-b for IUT-B. All exchanged packets between the client and IUT-A (resp. IUT-B) are captured in stream 1 (resp. stream 2). These two packet streams are then compared to check interoperability using IOT rules. Stream identifiers are used to refer to a specific stream, as follows: `S1.p6.flags = S2.p6.flags`.

IOT rules using stream identifiers. Let us consider a sample interoperability rule that discovers inconsistent response packets from two different packet streams $S1$ and $S2$. The rule matches different response packets $p2$ from streams $S1$ and $S2$, respectively, for a given test packet $p1$:

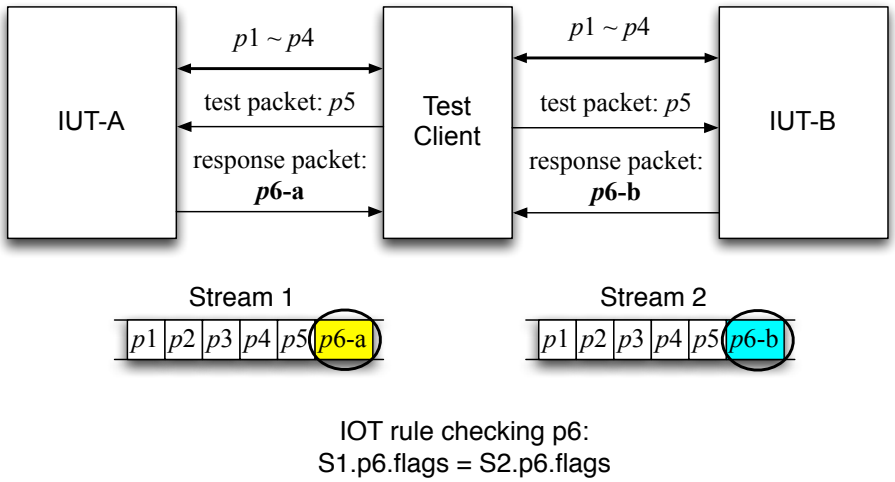


Figure 5.7: Packet stream and stream identifiers.

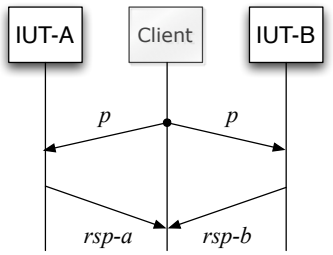
```
1 p1{src_ip != 224.0.0.251 AND flag.QR = 0xff AND questions != 0} ;
2 p2 {dst_ip = @p1.src_ip AND id = @p1.id AND flag.QR = 0x80
3  AND ANY data.answer(name = @p1.question.name) AND S1.data != S2.data}
```

A test query packet $p1$ (line 1) can be identified by the values of the fields `flags` and `src_ip` in the packet. The `next` operator (at the end of line 1) ignores intermediate packets that do not satisfy the filters for packet `p2` (lines 2 and 3). The rule then matches a response packet associated with the test packet from two packet streams `S1` and `S2`. The filter with the stream identifier (line 3) compares the value of the `data` field of the response packet $p2$ from the streams `S1` and `S2`.

5.4.2 Features of IOT Rules

Since SIOT compares field values of response packets when creating a set of IOT rules, developers refer to the packet format defined in a network protocol specification. Developers can build a set of IOT rules that simply compare the value of each field in response packets and detect any differences. However, such rules may not reliably discover interoperability problems due to two reasons: *recommended requirements* and *range of field values*.

Recommended requirements. Some requirements in a specification propose a recommended or optional value for a specific packet field. In this case, one vender may choose to include the requirement while another vendor may decide to ignore it. For example, the mDNS specification recommends to add further identifying information at the end of a resource record name if a name conflict occurs. Since this requirement allows to use any information at the end of a resource



Relationship	PASS	FAIL	Description
$rsp\text{-}a = rsp\text{-}b = \emptyset$	O	X	Both generate no response
$rsp\text{-}a = rsp\text{-}b$	O	X	Responses are identical
$rsp\text{-}a \neq \emptyset \,\&\, rsp\text{-}b = \emptyset$	X	O	One of them does not generate response
$rsp\text{-}a \neq rsp\text{-}b$	X	O	Responses are not identical

Figure 5.8: An IOT model (left) and decision criteria (right) used in SIOT.

record name, Zeroconf daemons implement this feature differently. Avahi appends the symbol `#` followed by an incremented number such as "`R-Name #2`" at the end of the name while Bonjour and JmDNS add an incremented number inside a bracket such as "`R-Name (2)`". Since the values for the record name are different, if IOT rules simply compare the value of the record name field of two packets from these two implementations, an interoperability problem would be reported, even though both implementations are correct.

Range of field values. In network protocol specifications, some packet fields can have a value within acceptable ranges. For example, the DHCP specification allows developers to decide a lease duration between zero and infinity and administrators select a given lease value based on their policy. We found that two commonly used DHCP servers, `isc-dhcp` (7 days) and `udhcp` (10 days), define default lease times differently. Again, a simple comparison rule of DHCP response packets would report this difference as a violation. However, since both values are within the valid range of the lease time, the response packets from both implementations are correct.

The next example shows an IOT rule that handle fields with a value in a certain range:

```
1 query {opts.type = DISCOVER} ;
2 resp {opts.type = OFFER AND S1.lease_time >= 0 AND S2.lease_time >= 0}
```

The rule considers the `resp` packets in both streams, `S1` and `S2`, as correct if the value of the `lease_time` field of both `resp` packets is greater than zero.

5.4.3 Interoperability Decision Criteria

After the comparison, SIOT produces a result showing whether implementations are interoperable with each other. In this section, we explain a detailed decision criteria for reporting IOT results.

First, we define an IOT model that shows the high-level process performed by SIOT. As shown in

the Figure 5.8 (left), the IOT model is comprised of a `Client` that sends test packets and IUTs. The figure also shows decision criteria (right) used in SIOT. As depicted in the IOT model, `Client` sends a test input packet p to both implementations, `IUT-A` and `IUT-B`, and waits for response packets, `rsp-a` and `rsp-b`. Based on these response packets, SIOT reports a result as either `PASS` (i.e. the implementations are interoperable with each other) or `FAIL` (i.e. the implementations are not interoperable). If both `IUT-A` and `IUT-B` generate no response packet (case 1) or identical response packets (case 2), SIOT reports `PASS` as a result. On the other hand, if both implementations generate different response packets or only one of implementations generates a response packet, SIOT reports `FAIL`. When both implementations respond to the test packet, IOT rules compare response packets field by field in order to detect any differences. For each `FAIL` decision, SIOT reports an error trace as well as field information that led to the interoperability failure.

5.5 Related Work

Previous research work on interoperability testing can be classified into two categories: (1) providing general concepts, experiences and formal definitions of interoperability testing and (2) developing methods for automatically deriving interoperability test cases.

The interoperability of many network protocols have been tested in the past to provide various concepts and formalisms for interoperability testing [DSU90, VB94, Tre99, BFN+05]. Moseley et al. [MRW03] present their experience of interoperability testing within the European Telecommunications Standards Institute (ETSI). Formal methods were also considered as an approach to support interoperability testing. Viho et al. [VBT01] extend the concepts of conformance testing to define a framework for interoperability testing and suggest some guidelines for interoperability test case generation.

Similar to this prior research, we provide a concept of interoperability testing that enables performing IOT efficiently (cf. Section 5.2.1) and a formal definition of interoperability relationships (cf. Section 5.2.2). SIOT provides an automated way to generate test packets for interoperability testing and check interoperability problems. Unlike previous interoperability testing approaches, SIOT derives interoperability test cases after exploring a large portion of the program state space using symbolic execution. Such test cases enable developers to check much deeper interoperability behaviour, such as complex sequences of interactions, i.e. behaviour that depends on accurate value information.

Most recent research work has focused on deriving interoperability test cases [MG08, PPP08, DV09]. Although many methods to generate test cases for interoperability testing exist, reachability analysis of finite state machines is a popular approach [KSK00, KC00, HLSG04, VRA+07,

SWSC07]. Kang et al. [KSK00] propose a technique for generating interoperability test suites for communication protocols. They first model an implementation as finite state machines and derive the test suite through analysis of reachable states from a start state. Koné et al. [KC00] use a transition system to generate test sequences for communication systems interworking with other network entities. Their approach automatically computes test patterns through a reachability graph, which is constructed after reachability analysis of the transition system. For approaches that use reachability analysis, the state space explosion problem is a major challenge to be overcome. Koné et al. have focused on minimising the number of entities during modeling. However, minimising the model may sacrifice chances to find bugs because important requirements are often not covered by an abstract model.

In contrast to these approaches, SIOT uses source code, which enables developers to build a model even when there is no formal specification of the target system. Since test cases are generated directly from the source code, they cover both the control and data parts of protocols. Most previous research work deals with test case derivation only for the control part of protocols because it observes input/output packets only at the network interface.

5.6 Summary

In this chapter, we introduced symbolic interoperability testing (SIOT), a part of SYMBEXNET for interoperability checking between different implementations of the same protocol from separate vendors. We started with an overview of conventional testing methods that are used for checking network protocol implementations, *conformance testing* and *interoperability testing*, discussing their differences. We also provided the definitions of interoperability and interoperability testing.

Next, we described SIOT in detail. It is designed with four requirements: *automation*, *flexibility*, *implementation-based checking* and *scalability*. We also provided formal definitions through an *interoperability testing model* and an *interoperable relationship*. SIOT is divided into three steps: (1) automated generation of a set of test input packets; (2) execution of the generated test packets; and (3) performing of IOT rule checking.

As a part of SYMBEXNET, SIOT inherits many features of SYMBEXNET, such as generation of test cases and replay of generated test cases. SIOT takes as input the source code of IOTs and generates test packets for them using symbolic execution. We extended the replay process for IOT so that generated test packets are replayed not only on the original implementation but also across all the other IOTs. We also enhanced the packet rule language by introducing packet stream identifiers, thereby allowing rules to compare packet streams from multiple implementations. Finally, we provided the IOT decision criteria, illustrating how SIOT decides regarding interoperability errors.

CHAPTER 6

Evaluation

TO check a network protocol implementation, SYMBEXNET takes three steps: (1) derivation of packet rules; (2) generation of test input packets; and (3) replay of test packets and validation of the implementation and its interoperability. We applied SYMBEXNET to five real-world network protocol implementations. Our experiments show that SYMBEXNET generates high quality test packets and sequences to check a network protocol implementation as well as its interoperability with other implementations. In particular, SYMBEXNET can avoid the path explosion problem, and the generated packets explore target implementations with high source code coverage.

This chapter is organised as follows. In the next section, we describe the methodology for evaluating SYMBEXNET. Section 6.2 discusses how rules are derived from protocol specifications. The environmental set-up used for the experiments is described in Section 6.3, including details on establishing an isolated network environment. The experimental results on single packet exchange symbolic execution (SPE-SE), multi packet exchange symbolic execution (MPE-SE) and symbolic interoperability testing (SIOT) are presented in Sections 6.4, 6.5 and 6.6, respectively. Finally, we present the detected violations in Section 6.7.

6.1 Overview

The goal of our evaluation is to demonstrate the suitability of SYMBEXNET as an efficient checking tool for finding implementation flaws in real-world network protocol implementations. We applied SYMBEXNET to network daemons implementing the Zeroconf [CK10, Stu10] and the DHCP [Dro97] specifications. A checking system such as SYMBEXNET can be evaluated in terms of the quality of generated test packets and its ability to detect implementation bugs. To show the quality of test input packets, we measure the source code coverage achieved by the generated packets. Bug detection ability is evaluated by validating network protocol implementations against their protocol specifications and detecting implementations flaws. SYMBEXNET discovered 39 unique flaws (22 for Zeroconf and 17 for DHCP). These bugs are due to implementations mistakes

Table 6.1: The summary of the network daemons tested using SYMBEXNET.

Protocol	Daemon	Version	Language	# LOC	# Files
Zeroconf	`Avahi`	0.6.23	C	7K	31
	`Bonjour`	107.6	C	7.9K	10
	`JmDNS`	3.4.1	Java	2K	9
DHCP	`isc-dhcp`	2.0	C	3K	15
	`udhcp`	0.9.9-pre	C	1.2K	12

and ambiguous requirements in the specifications. In our evaluation, we address the following questions:

1. How efficiently does SYMBEXNET derive packet rules from protocol specifications? (Section 6.2);
2. Does SYMBEXNET generate effective test input packets (or sequences) that can achieve broad and deep exploration of the program state space using symbolic execution? (Sections 6.4 and 6.5);
3. Does SYMBEXNET provide an effective way to check interoperability of network daemons? (Section 6.6); and
4. Does SYMBEXNET manage to detect various types of non-trivial implementation bugs? (Section 6.7)

The five network protocol implementations tested are summarised in Table 6.1. We investigate three different implementations of Zeroconf using SYMBEXNET: *Avahi 0.6.23*[1], Apple's *Bonjour 107.6*[2] and *JmDNS 3.4.1*[3]. `Bonjour` has about 8K lines of source code in 10 files for its Linux version and `Avahi` has about 7K lines of source code in 31 files. As `Bonjour` and `Avahi` are the most widely used Zeroconf implementations, we use them for symbolic execution. Nowadays Zeroconf is becoming a vital part of applications on the iOS and Android platforms. `JmDNS` is a Java implementation of Zeroconf and the only available Zeroconf server that can be used on the Android platform because `JmDNS` provides a pure Java library. In order to show the effectiveness of the generated test packets, after we generate test packets from the two C daemons, we use the same packets on all three daemons.

For DHCP, we use two different implementations: *udhcp 0.9.9-pre*[4] and *ISC's DHCP 2.0*[5]. Both `udhcp` and `isc-dhcp` are open source DHCP implementations and their code has been thoroughly

[1] `http://www.avahi.org`
[2] `http://developer.apple.com/opensource`
[3] `http://jmdns.sourceforge.net/`
[4] `http://busybox.net`
[5] `http://www.isc.org/software/dhcp`

Table 6.2: The results of the rule derivation from the specifications.

	Zeroconf: 25 rules		**DHCP: 29 rules**	
Keyword	# Total	# Translated	# Total	# Translated
MUST	79	29	72	8
MUST NOT	29	4	46	15
SHALL (& NOT)	2	0	0	0
Others[1]	0	0	7	7

[1] phrases signifying absolute requirements but without usage of the above keywords.

tested, which makes them suitable for our evaluations.

6.2 Rule Derivation

We manually derive a set of rules from the specifications of the Zeroconf and DHCP protocols by following the process described in Section 4.3. Our rule derivation for Zeroconf and DHCP resulted in a total of 25 and 29 rules, respectively. In Table 6.2, we show the result of the rule derivation from the specifications of both protocols. After becoming familiar with the process of developing verifiable specifications through the experience with the mDNS specification, it took around 3–4 hours to analyse the DHCP specification and to derive the DHCP packet rules.

Zeroconf protocol. As explained in Section 2.1, the Zeroconf protocol is defined in two specifications: multicast DNS (MDNS) and DNS-based Service Discovery (DNS-SD). To obtain a set of packet rules, as defined in Section 4.4, we examine both specifications to find phrases that contain the keywords from Section 4.3. As shown in Table 6.2, we find 110 phrases: 79 phrases with a "MUST" keyword, 29 with "MUST NOT" and 2 with "SHALL/SHALL NOT".

Not all of these phrases can be translated into rules—we translate successfully 29 phrases based on "MUST", 4 phrases based on "MUST NOT" and none of the phrases with "SHALL/SHALL NOT". Some statements are purely informative and some contain environmental requirements such as the interfaces that must be supported. Any phrases referring to the internal state of the daemon, such as the cache maintained by the Zeroconf daemon, have to be ignored. Finally, some phrases that are used to describe the same requirement are combined into a single rule. In total, we obtain a verifiable specification with 25 rules based on 33 valid phrases.

DHCP protocol. After following the same rule derivation procedure, we find 118 phrases in total ("MUST": 72, "MUST NOT": 46 and "SHALL/SHALL NOT": 0). For these 118 phrases, we transform 23 phrases into rules. Most untranslated phrases are related to requirements describing desired behaviour of DHCP clients. Since we are not checking clients but server implementa-

tions, any statements that are not related to the DHCP daemon are ignored. While analysing the specification, we find an additional seven phrases stating absolute requirements without the above keywords. Since these phrases specify how the DHCP daemon must construct response packets, we derive rules from the phrases. The following shows an example phrase.

> *"If the 'giaddr' field is zero and the 'ciaddr' field is nonzero, then the server unicasts DHCPOFFER and DHCPACK messages to the address in 'ciaddr'."*

In summary, we obtain a verifiable specification consisting of 29 rules based on 23 phrases associated with the above keywords and 7 other phrases for the DHCP protocol.

6.3 Experimental Set-up

We conduct our experiments on a 2.4 Ghz Intel Core2 Duo machine with 2 GB of RAM under 32-bit Ubuntu Linux. Usually there exist other network daemons on the network and they generate network traffic that is not related to our experiments. To control network traffic during test packet generation and replay, all experiments are done as part of an isolated test network. We configure the daemons to use the `loopback` (`lo`) interface under Linux allowing daemons to receive only packets from the isolated network.

The general experimental setup used in checking implementations involves two nodes: a network daemon under test and a client. The network daemon is executed, and the client is constrained to communicate with the daemon using our SYMBEXNET verification tool, which is called a test agent. To validate the network daemon, SYMBEXNET captures all network traffic generated by the daemon and clients during the replay on the network interface. For this, SYMBEXNET uses libpcap [Law94], a portable packet capture library.

Zeroconf protocol. When a Zeroconf daemon starts, the daemon typically discovers available services or devices on the network. To emulate a typical environment (i.e. with several network services or devices), the DNS Service Discovery (DNS-SD) client is configured to register six services with the Zeroconf daemon. Table 6.3 shows the set of services that are used in the experiments. These are services that are usually available in a small office environment. We register these services in the same order for each experiment. After registering these services, SYMBEXNET injects a query for browsing a service with the following UNIX `dig` command, which is a command line tool for sending a DNS query:

```
dig -b 127.0.0.1 -p 5353 @224.0.0.251 _http._tcp.local ptr
```

The Zeroconf daemon is instrumented to detect an input packet that is constructed with specific

Table 6.3: A set of network services created for the Zeroconf experiments.

Order	Service name	Type	Port	Additional properties
1	My_music	_daap._tcp	7800	none
2	My_printer	_printer._tcp	5155	pdl=application/postscript
3	My_wiki	_http._tcp	4653	path=/mywiki.html
4	Music_wiki	_http._tcp	6753	path=/musicwiki.html
5	CS_printer	_printer._tcp	5678	pdl=application/postscript
6	Bob_iTunes	_daap._tcp	6543	none

field values (i.e. a unicast packet querying the service "`_http._tcp.local`"). When it receives such an input packet, symbolic execution is started.

DHCP protocol. For the DHCP protocol, we use a DHCP client written in Python to initiate symbolic execution. The client injects a symbolic DHCP query message to the DHCP daemon, which is instrumented to detect the symbolic DHCP packet. When the daemon receives the symbolic DHCP packet, it starts to run symbolically. SYMBEXNET initiates the client with the following command:

```
sym-dhcp-client -s -p 6868
```

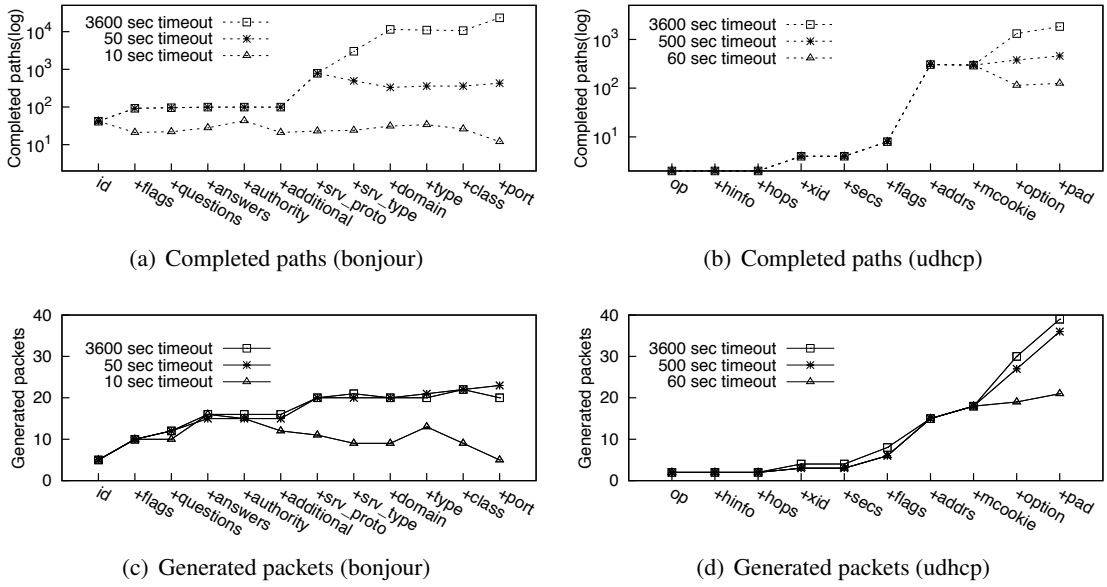
In order to make the DHCP daemon only handle DHCP requests from our DHCP client, we use a customised DHCP port number (`6868`) instead of the well-known port number (`68`). The `-p` flag is used to specify the customised DHCP port number. The `-s` flag is used to indicate to the client to send a symbolic input packet to the DHCP daemon.

6.4 Single Packet Exchange Symbolic Execution

In this section, we describe the experiments that check stateless network protocol implementations (cf. Section 2.1) through generating and replaying test input packets followed by packet rule checking. The experiments were chosen to evaluate the effectiveness of test packets generation. We report the number of generated test input packets and their line coverage and give examples of the detected implementation errors.

In order to avoid the path explosion problem, we try to mark all combinations of fields in given protocol packets, starting with only one field, and then progressively advancing to larger numbers of fields marked symbolic at the same time. The following invocation causes SYMBEXNET to analyse a Multicast DNS daemon:

```
symbexnet-analysis -p zeroconf -d ./daemon/mdns.bc
```



(a) Completed paths (bonjour)

(b) Completed paths (udhcp)

(c) Generated packets (bonjour)

(d) Generated packets (udhcp)

Figure 6.1: Number of completed paths and generated test packets with various symbolic execution timeout values for `Bonjour` (a and c) and `udhcp` (b and d), while progressively marking more fields as symbolic.

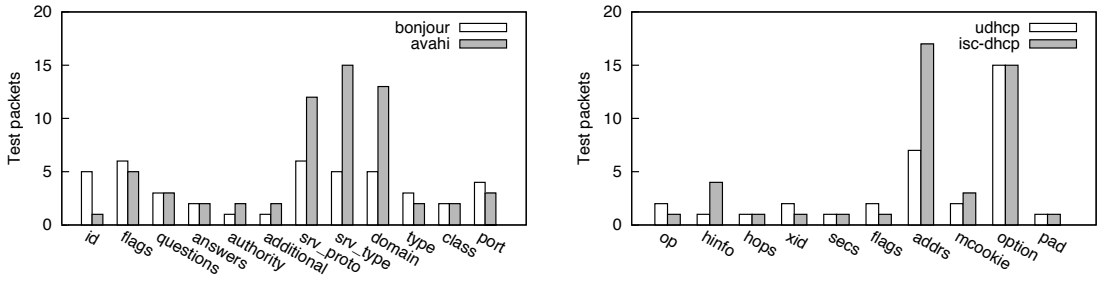
It provides the protocol name with the `-p` option and the file path of the daemon that needs to be executed symbolically with the `-d` option.

6.4.1 Test Packet Generation

First we explore how the amount of symbolic data per input packet affects the number of generated test cases and the code coverage for specific network daemons. Initially, we attempt to mark each field of an input packet as a symbolic variable. In our experiments, DNS and DHCP input packets have 12 and 10 fields, respectively. Each field has a size between 2 bytes (e.g. the `ID` field) to 299 bytes (e.g. the `pad` field). Since the length of these input packets are 512 bytes (DNS) and 5048 bytes (DHCP) in total, it is impractical to mark the entire packet as symbolic and run a daemon symbolically for a long amount of time.

We generate test packets first for the mDNS daemons and then for DHCP daemons. For the mDNS daemons, we start with the `ID` field, as the only symbolic field, run symbolic execution to generate input test packets and then progressively mark more fields as symbolic. As more fields are made symbolic, the number of paths explored by SYMBEXNET increases. By default, one test packet is generated for each path that is explored. To avoid unnecessarily generating a large number of packets, SYMBEXNET configures KLEE to generate only test packets for paths that cover new statements in the code. We also explore different timeout values for the symbolic execution.

Figure 6.1 shows (a) the number of explored paths and (c) generated test packets when we increase



(a) Zeroconf (b) DHCP

Figure 6.2: Number of generated test packet comparisons for Zeroconf and DHCP.

the number of symbolic packet fields and use different timeout values. The results reveal two insights. First, in the case of `Bonjour` and `Avahi`, a 50s timeout value offers a good tradeoff between the time needed to run the experiments and the number of generated test packets. With a 10s timeout, SYMBEXNET generates significantly fewer test packets, but increasing the timeout to 1 hour does not significantly increase the number of generated packets (i.e. SYMBEXNET generates many more paths, but most of them cover the same lines of code). Therefore, we use a 50s timeout value in all of our experiments. In some cases (for example, Figure 6.1(c)), non-determinism in KLEE produces slightly fewer packets for a long timeout value than a short timeout value.

We apply the same timeout selection approach to the `udhcp` daemon. Figures 6.1(b) and 6.1(d) show the number of explored paths and generated test packets for `udhcp`. The results suggest a 500s timeout value for `udhcp` and we use this timeout value in all `udhcp` and `isc-dhcp` experiments. Since the `udhcp` implementation is less complex than the `Bonjour` one, SYMBEXNET can explore `udhcp`'s state space for a longer period of time before exceeding the memory usage limit.

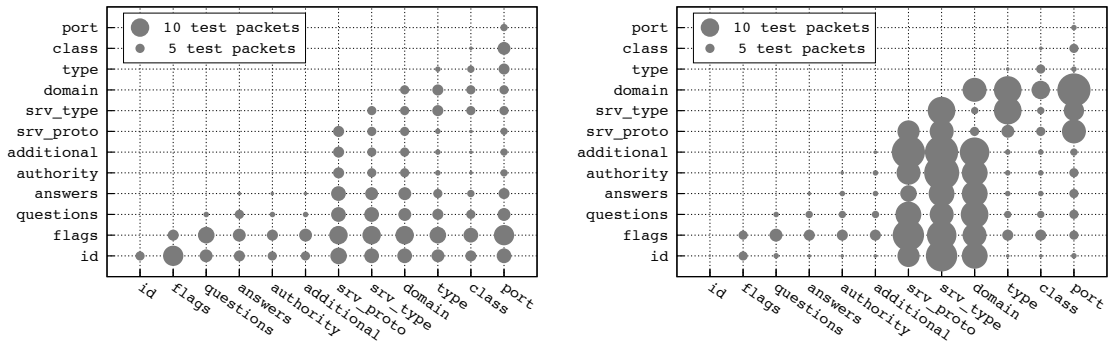
In order to analyse the sensitivity of each field in different implementations of the protocol, we conduct an experiment, in which we mark one field at a time as symbolic and compare the number of test packets generated for Zeroconf (i.e. `Avahi` and `Bonjour`) and DHCP (i.e. `udhcp` and `isc-dhcp`).

Figure 6.2 shows the results. As expected, we obtain similar sets of test packets for the daemons implementing the same protocol. For example, when we mark the `port` field as symbolic, SYMBEXNET generates the `port` values 0, 512 and 5353 for `Avahi` and 0, 2, 5351 and 5353 for `Bonjour`. There are certain fields, such as `srv_proto` and `domain` in Zeroconf, however for which we obtain more test packets for `Avahi` than for `Bonjour`. By examining the code, we discover that the `Avahi` implementation compares the different fields (e.g. domain names) in a complex fashion, i.e. using `if` statements and thus requires more test packets to cover all possible code statements.

Next we try all 4095 (and 1023) possible combinations of symbolic fields for multiple implemen-

Table 6.4: Summary of symbolic execution for all combinations of packet fields

	Description	Avahi	Bonjour	isc-dhcp	udhcp
1	Number of packet fields	12	12	10	10
2	Number of combinations	4095	4095	1023	1023
3	Number of generated test packets	34,047	32,069	16,777	14,271
4	Symbolic execution timeout (seconds)	50	50	500	500
5	Total execution time (hours)	31	22	27	26



(a) Bonjour (b) Avahi

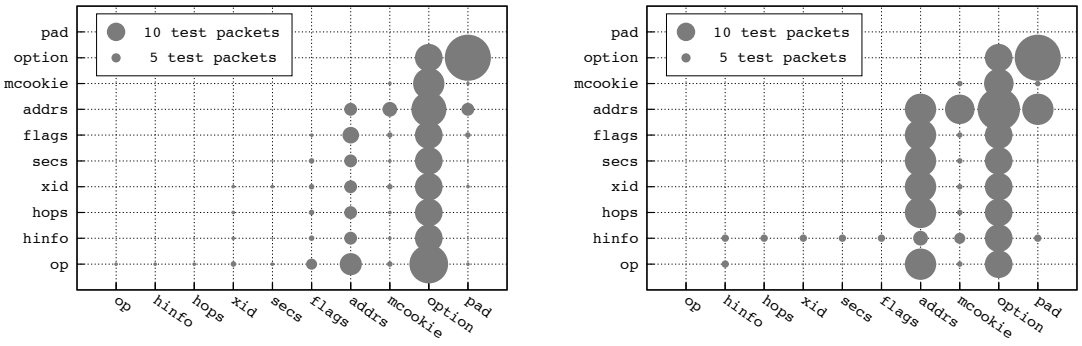
Figure 6.3: Number of generated test packets for pairs of fields in Zeroconf (subset of all combinations for each daemon)

tations of Zeroconf (and DHCP). Obtaining and comparing the number of generated test packets for different packet field combinations helps us understand how each packet field is handled by the implementation.

As shown in Table 6.4, using all combinations of the packet fields, SYMBEXNET generates 32,069 test packets, with a total execution time of around 22 hours for `Bonjour`; and 34,047 test packets in 31 hours for `Avahi`. Each combination is timed out after 50 seconds. For DHCP, SYMBEXNET generates 16,777 test packets in 26 hours for `udhcp` and 14,271 test packets in 27 hours for `isc-dhcp`, with a 500 second timeout value. Overall, symbolic execution generates more test packets for the Zeroconf daemons than for the DHCP daemons. The daemons of the same protocol generate similar numbers of test packets. This implies that the daemons of the same protocol are not that different in the way that they handle input packets.

Comparing the number of generated packets for each field (or pairs of fields) of the daemons implementing the same protocol can show how similar they are in terms of processing an input packet. Figures 6.3 and 6.4 show the number of generated test packets for a subset of these combinations, namely those in which pairs of fields are marked as symbolic. The results show that the number of generated packets depends on which fields are marked as symbolic.

In Figure 6.3(a), when the `flags` field is marked as symbolic, a relatively high number of test



(a) uDHCP

(b) ISC-DHCP

Figure 6.4: Number of generated test packets for pairs of fields in DHCP (subset of all combinations for each daemon)

packets is generated because the `flags` field contains control information that is used extensively by the network daemon to decide how to process a packet. When the `authority` or `additional` fields are marked as symbolic, however, fewer test packets are generated because these fields are only used to specify the number of associated data in the payload part of the packet.

The figure also reveals that implementations of the same protocol specification show similar numbers of generated test packets. For example, Figures 6.4(a) and 6.4(b) show that the circles associated with the `addrs` and `option` fields lead to relatively high number of test packets. These two fields contain information, such as a lease address, a client MAC address and message type, which affect the behaviour of the DHCP server for an input message. Consequently these values are checked by many functions when the daemon handles an input packet.

6.4.2 Line Coverage Results

We use line coverage to measure the quality of test packets generated by symbolic execution. Coverage is measured on the instrumented original binary using the `gcov` tool, which is part of the GNU GCC compiler suite [Lic]. We disable unnecessary compile options, such as debugging features, and exclude from the calculation library files that are not relevant to our experiments.

Table 6.5 shows the line coverage numbers for each daemon. First, we measure the coverage when a daemon is in an idle state. When a daemon is started, it sets up its configuration parameters and environment variables, such as the domain name and the cache size, before waiting for requests. SYMBEXNET injects the test packets generated using symbolic execution and measures coverage. We observe the line coverage that can be obtained from these test packets and compare it with random testing. For this, we prepare randomly generated test packets using the D-ITG (Distributed Internet Traffic Generator) random packet generator [AGE+04]. It is able to produce traffic for various protocols, such as TCP, UDP, ICMP and DNS. Using the generator, we create a similar

Table 6.5: Line coverage of each daemon for generated test packets.

No.	Daemons	# files	total LOC	Baseline (%)[1]	uncovered LOC (%)	Random Test [2]		SYMBEXNET-gen [3]	
						# pkts	Cov.(%)	# pkts	Cov.(%)
1	Avahi	31	7K	29.53	21.75	30K	63.19	34,047	75.25
2	Bonjour	10	7.9K	20.3	28	30K	48.47	32,069	61.5
3	isc-dhcp	15	3K	31.65	30.85	15K	35.59	16,777	66.85
4	udhcp	12	1.2K	31.21	16.43	15K	59.86	14,271	79.15

[1] Baseline shows the coverage when a daemon is in its idle state.
[2] Random shows the coverage with the randomly generated test packets using D-ITG packet generator.
[3] SYMBEXNET-gen shows the coverage with test packets generated by SYMBEXNET.

number of random test packets for Zeroconf (30K) and DHCP (15K). There is a trade-off between the number of test packets and the total test execution time. In order to finish the testing within a reasonable time, such as less than 24 hours, we generate a similar number of test packets instead of generating test packets for a similar period of time.

Zeroconf protocol. First, we explore the source code coverage of the `Bonjour` daemon. On average, the generated test packets by SYMBEXNET cover 61.5% of the code, while the baseline tests that execute the daemon without test packets cover only 20%.

Note that our test scenario cannot cover 28% of the source code: in addition to DNS response/request packets, the daemon accepts service registrations from DNS-SD clients, which are not explored symbolically in our experiments. About 15% of the source code is used to handle such requests; another 13% implement other features, such as cache maintenance and name conflict resolution. We also achieve similar coverage for the `Avahi` daemon of 75.25%.

DHCP protocol. SYMBEXNET generates test packets covering 66.85% and 79.15% for `isc-dhcp` and `udhcp`, respectively. About 30% (`isc-dhcp`) and 15% (`udhcp`) of the source code are not covered by our test scenario because they are mostly related to `BOOTP` packet handling, static IP address allocation and unsupported server configurations. Since `isc-dhcp` contains additional features such as DNS lookup, it achieves lower line coverage than `udhcp`.

6.4.3 Discovered Implementation Errors

Using SYMBEXNET, we apply the generated test packets to all three Zeroconf implementations in order to find violations of our packet rules. Although the generated packets come from the `Avahi` and `Bonjour` implementations, they can be used to test other Zeroconf implementations because they are highly effective test packets containing malformed data and corner cases.

Using single packet exchange symbolic execution (SPE-SE), SYMBEXNET discovers five different

```
1 mDNSexport mStatus mDNSPlatformSendUDP
2 (const mDNS *const m, void *const msg, mDNSu8 *const end,
3 mDNSInterfaceID InterfaceID, mDNSAddr *dst, mDNSIPPort dstPort) {
4     ...
5     assert(m != NULL);
6     assert(end != NULL);
7     assert(dstPort.NotAnInteger != 0);
8     ...
9 }
```

Figure 6.5: Code fragment from the `Bonjour` daemon implementation that sends UDP response packets.

errors: two in the `JmDNS` implementation, one in each `Avahi` and `Bonjour` and two in `Bonjour`. We describe two of these errors, one generic and one semantic bug.

Violation 1 (Generic error): Vulnerability caused by source port number of zero. When SYMBEXNET marks the source port field as symbolic, we obtain test packets with the following four values: 0, 2, 5351 and 5353. All these port numbers are used as well-known ports — e.g. port 5353 is assigned to mDNS. According to the mDNS specification, a query must be sent as a multicast packet from port 5353 or as a unicast packet from a random port number. If the source port in a received query is not 5353, the daemon should consider the packet to be a unicast query and generate a conventional unicast response, for example, by repeating the query ID and sending a response to that source port. Therefore, we expect the daemons simply to reply with a response packet to all port numbers without any errors. However, we detect abort errors in `Bonjour` and `Avahi`. Both errors are caused by the source port number of a query packet.

During the replay process, when SYMBEXNET crafts a packet with the source port of 0, the daemons abort after receiving the packet due to an `assert` statement violation. In the case of `Bonjour`, the daemon calls the `mDNSPlatformSendUDP` function to send a response packet as shown in Figure 6.5. Line 8 causes the daemon to abort. Therefore, sending the crafted packet to a multicast address (224.0.0.251) terminates all `Bonjour` daemons in the network, which have an answer to the query. The crafted packet also aborts `Avahi` daemons. This occurs regardless of the existence of a response packet because the assertion is located in a function that handles any received packets.

We have reported this bug to Apple who confirmed it. The latest version of `Bonjour` as of this writing (*Bonjour 320.5.1*) does not exhibit the problem any more. The bug in `Avahi` was detected and its patch applied to version *0.6.28*.

Violation 2 (Semantic error): Incorrect response for unknown record class. When a Zeroconf daemon receives a query packet asking for a specific service, it must compare three values (*name*, *type* and *class*) against its records. The daemon only responds to a query packet when it has a record with the same values for these three fields. This requirement is stated in the specification:

> *"The record name must match the question name, the record rrtype must match the question qtype unless the qtype is ANY (255) or the rrtype is CNAME (5), and the record rrclass must match the question qclass unless the qclass is ANY (255)"* [CK10]

From the above statement, we derive the following rule:

```
1 query{src_port != 5353 AND dst_port = 5353 AND flag.QR = 0x00} ;
2 resp {dst_port = @query.src_port AND flag.QR = 0x80 AND data.answer(class !=
     'ANY' AND class != @query.question.class)}
```

The `class` field states the value of services that define the protocol type. The normal value is "IN", which refers to the Internet protocol. When SYMBEXNET marks the `class` field as symbolic, we obtain the following two test packets: "IN (Internet)" and "0x00 (unknown type)". Both `Bonjour` and `Avahi` respond only to the query with class value "IN", which is the correct behaviour. However, `JmDNS` incorrectly sends a response even when it receives a query with an unknown class value. This can give incorrect service information to clients, which may in turn send further unnecessary queries.

6.5 Multi Packet Exchange Symbolic Execution

In this section, we evaluate the effectiveness of multi packet exchange symbolic execution (MPE-SE) by analysing line coverage results of test packets generated using MPE-SE. As described in Section 2.1, both Zeroconf and DHCP contain state machines. The state machine of DHCP changes its current state based on input packets. Zeroconf also performs transitions based on timeouts (i.e. the transition occurs when there is no input packet for a given amount of time). Since MPE-SE is for stateful network protocols that exchange multiple packets, we focus our evaluation of MPE-SE on DHCP.

6.5.1 Generation of Test Sequences

As described in Section 2.1, the state machine of DHCP is built based on the life cycle of a dynamically assigned IP address between a DHCP client and the daemon. For the life cycle, the DHCP daemon receives three input packets from the client. To generate a series of test inputs that can cover all possible combinations of input/output packets of the life cycle, SYMBEXNET runs MPE-SE on *three* DHCP input packets.

Table 6.6: Generated sequences of packets for DHCP.

Cycle	Test sequence
1	DHCPREQUEST
2	DHCPDISCOVER – DHCPREQUEST
3	DHCPDISCOVER – DHCPREQUEST – DHCPINFORM

Table 6.7: Results after applying MPE-SE to the DHCP daemons.

No.	Description	udhcp server		isc-dhcp server	
		120 min	60 min	60 min	30 min
1	Total # of test cases	293	248	608	397
2	Total # of test cases from the first packet (Discovery)	41	40	46	32
3	Total # of test cases from the second packet (Request)	217	174	407	198
4	Total # of test cases from the third packet (Release)	35	34	155	167
5	Total # of test paths	286	242	595	385
6	Total elapsed time for MPE-SE (hours)	16.9	7.67	17.8	8.5

In the first MPE-SE cycle, SYMBEXNET marks the DHCPDISCOVER packet, which is the first input packet in the life cycle, as a symbolic packet and generates test packets that are used as input in the second cycle. After performing the same procedure in the second and third cycles, SYMBEXNET generates a set of test input sequences. Table 6.6 shows the three test input sequences generated after performing MPE-SE on each symbolic packet. As shown, the generated test sequences can have either one, two or three input packets, each exploring different execution paths in the DHCP daemon.

Table 6.7 shows results after applying MPE-SE to the udhcp and isc-dhcp daemons with different timeout values. We run both daemons with *three* symbolic DHCP input packets and mark 548 bytes of symbolic input packet as symbolic because the DHCP messages have a maximum length of 548 bytes. The experiment suggests 120 mins (udhcp) and 60 mins (isc-dhcp) of timeout values for each MPE-SE cycle in order to get test input sequences within a day. Otherwise SYMBEXNET generates fewer test sequences or requires too much time for the experiments. Since each MPE-SE cycle is performed based on the test packets generated in the previous cycle, more test packets in earlier cycles increases the total execution time for the whole MPE-SE cycle exponentially. With these timeout values, udhcp and isc-dhcp generate a total of *286* and *595* unique test sequences, respectively.

The symbolic path trees (see Section 3.5) for isc-dhcp and udhcp have depth *three* because MPE-SE is performed on three symbolic input packets. The set of all nodes at each depth represent the generated test cases in each cycle. Test sequences of packets are obtained by finding all paths from the root to the leaves.

Although SYMBEXNET uses smaller timeout values for `isc-dhcp` (60 mins), the number of generated packets for `isc-dhcp` from each MPE-SE cycle is greater than for `udhcp`. After analysing the source code of both daemons, we find that the `isc-dhcp` daemon contains more conditional statements than `udhcp` to handle input packets. Generated test input packets after the first cycle contain all possible DHCP message types as well as invalid packet formats, thereby broadly exploring the source code of the daemons. For both daemons, the second cycle generates a large number of test packets, and fewer packets are part of the third cycle. A large portion of generated packets in the second cycle either transfer the state of both daemons to invalid states or have an invalid packet format. As these packets can be received only after receiving a packet from the first cycle, we find that they can cover source code that can only be reached after exchanging several packets in a specific order.

6.5.2 Line Coverage Results

In this experiment, we investigate line coverage to show how test packets generated using MPE-SE explore the source code. We run generated test sequences of packets on each unmodified network daemon and use `gcov` to measure coverage. To send DHCP packets in a test sequence to the DHCP daemon in the correct order, we use a DHCP client written in Python.

As a baseline, we first measure the coverage achieved by a DHCP conformance test suite that checks the functional correctness of the DHCP daemon. The conformance test [Dro97, Vit] sends a series of DHCP messages to the daemon in the order of `DHCPDISCOVER`, `DHCPREQUEST` and `DHCPRELEASE`. It checks if the daemon responds correctly. Since we perform MPE-SE on three symbolic DHCP input packets and generate test packets, it is reasonable to use the coverage from the conformance test as a baseline.

We then measure the coverage after applying MPE-SE on the first symbolic input packet (first cycle). For the test packets generated from second and third cycles, we also observe the achieved coverage. By comparing the obtained coverage from each cycle with each other and the baseline, we determine to what degree symbolic execution on multiple symbolic input packets affects coverage. We also generate test packets with an extended timeout value in the first cycle and measure the coverage. This enable us to compare the effectiveness of MPE-SE and SPE-SE with a long timeout value.

6.5.3 Source Code Analysis

Table 6.8 shows that the test sequences generated after the second MPE-SE cycle achieve more line coverage than the sequences generated after the first MPE-SE cycle. Since the second cycle occurred after handling the first input packet, it can eliminate several constraints that are associated

Table 6.8: Comparison of source code coverage measurements for the `udhcp` and `isc-dhcp` servers.

Daemon	Description	conformance test	1st MPE-SE	2nd MPE-SE	3rd MPE-SE	1st MPE-SE long	combined
udhcp	Symbolic input size (bytes)	-	548	548	548	548	548
	Elapsed time for single symbolic execution (hours)	-	2	2	2	16.9	-
	Total execution time (hours)	-	2	12.9	2	16.9	-
	Source code lines executed	469	507	536	536	533	558
	% of line coverage	66.52	71.91	76.03	76.03	75.60	79.15
isc-dhcp	Symbolic input size (bytes)	-	548	548	548	548	548
	Elapsed time for single symbolic execution (hours)	-	1	1	1	17.8	-
	Total execution time (hours)	-	1	9.4	7.4	17.8	-
	Source code lines executed	1395	1600	1658	1658	1634	1698
	% of line coverage	59.82	68.61	71.1	71.1	70.07	72.81

with symbolic input packets, thereby enabling a deeper state exploration that can be reached after exchanging several packets in a specific order.

The last column in Table 6.8 shows the accumulated coverage of all experiments. It is greater than that of any individual result. This means that our two symbolic execution methods, MPE-SE and SPE-SE with longer execution times, are complementary because there exist statements that can only be covered by test packets generated by one of the methods. Consequently combining the two methods enable developers to achieve even higher line coverage.

Figure 6.6 shows a fragment from the `udhcp` source code that handles an input DHCP packet based on its message type and the current state of the daemon (with additional comments and some omissions). When a DHCP daemon receives a `DHCPDISCOVER` message (line 6) requesting an IP lease from a client as the first input packet, it reserves an IP address and sends a `DHCPOFFER` message (line 9) to the client. The client accepts the DHCP offer and broadcasts a `DHCPREQUEST` message (line 12) because the `DHCPOFFER` does not contain the IP address. The daemon then simply responds with a `DHCPACK` message (line 21) with the IP address to confirm the allocation.

Before sending a `DHCPOFFER` message, the `udhcp` daemon stores the offered IP address in an internal array (line 9). When the `udhcp` daemon receives a `DHCPREQUEST` message, it first checks the message to see if any IP address has been leased to the given client's hardware address by calling the `find_lease_by_chaddr` function (line 1). If there is an address that matches the client's hardware address, the daemon sends a `DHCPACK` response (lines 19–23). Otherwise a `DHCPNAK` message is sent to the client (lines 25–27). The variable `lease`, assigned in line 1, is used to decide which message the daemon sends to the client when it receives a `DHCPREQUEST` message. The `find_lease_by_chaddr` function returns the first lease that matches a given client's hardware address. If there is no match, the function returns `NULL`. Since a lease is assigned to a

```
1  lease = find_lease_by_chaddr(packet.chaddr);
2  /* find the first lease that matches client's hardware address */
3  /* lease becomes NULL if no match */

5  switch (state[0]) {
6    case DHCPDISCOVER:
7       /* sendOffer function gets an lease address and stores it into
8       an internal array, then sends an offer message  */
9       sendOffer(&packet);
10      break;

12   case DHCPREQUEST:
13      requested = get_option(&packet, DHCP_REQUESTED_IP);
14      server_id = get_option(&packet, DHCP_SERVER_ID);

16      if (lease) {
17         /* For single round symbolic execution, lease becomes always NULL */
18         /* Routines within this if statement are not covered by SPE-SE */
19         if (server_id) {
20           if (requested && requested_align == lease->yiaddr) {
21              sendACK(&packet, lease->yiaddr);
22           }
23         }
24      } else if (requested) {
25         if ((lease = find_lease_by_yiaddr(requested_align))) {
26            sendNAK(&packet);   /* Send NAK message */
27         }
28      }
29      break;
```

Figure 6.6: Code fragment from the `main` function in `udhcp`.

client when the server sends a `DHCPOFFER` message, the variable `lease` can have a value only after the server receives a `DHCPDISCOVER` followed by a `DHCPREQUEST` message.

One of the generated test input sequences after the first MPE-SE cycle is a `DHCPREQUEST` message. During replay, the generated `DHCPREQUEST` message is sent to the daemon as the first input from the client. Since the daemon has not yet assigned an IP address to the client, the `lease` variable becomes `NULL` (in line 1). The `lease` variable can only have a proper value once the daemon has received a `DHCPDISCOVER` message from a client and assigned an IP address to the client. The `DHCPREQUEST` message is supposed to execute the statements (lines 13–29) below the `case DHCPREQUEST` statement. However, the statements highlighted in gray (lines 19–23) cannot be covered by the message because of the `lease` variable.

In contrast, a test sequence generated after the second cycle, `DHCPDISCOVER-DHCPREQUEST`, covers the unexplored statements (lines 19–23). With this test sequence, a client sends a `DHCPDISCOVER` message followed by `DHCPREQUEST`. When the daemon receives the first message, it assigns an IP address to the client and stores the address as a leased address mapped to the client's hardware address. Therefore, once the daemon receives the second message, `DHCPREQUEST`, SYM-

BEXNET can explore the statements (lines 13–29).

Although MPE-SE achieves high source code coverage and enables to explore uncovered execution paths, we could not detect errors from these new covered execution paths.

6.6 Symbolic Interoperability Testing

Response packets from network implementations of the same protocol typically have common packet field values because they refer to the same protocol specification. If they behave differently, i.e. generate different response packets, interoperability problems may occur. In this section, we explore interoperability between multiple implementations of the same protocol by different vendors. The purpose of this evaluation is to show that SYMBEXNET provides an efficient interoperability testing method. For interoperability testing, we use the generated test packets from two implementations of Zeroconf (`Avahi` and `Bonjour`) and DHCP (`udhcp` and `isc-dhcp`). The test packets are replayed across all implementations of the same protocol. Response packets of these implementations are then compared to check their interoperability.

6.6.1 Interoperability Rule Derivation

SYMBEXNET checks whether the daemons under IOT generate consistent response packets for a given test input packet. Any deviations between the daemons, such as generating response packets that have different values for certain fields or only one of them responds to a given test packet, are reported as interoperability bugs. For this purpose, a list of IOT rules are derived from protocol specifications.

When we derive IOT rules, not all the fields of the response packets from the daemons are compared. Packet fields can have values that depend on the time or are within a range. For example, packets may have different `timestamp` fields and the `destination ip address` may be different according to the device. Therefore, we check these fields from specifications and do not derive IOT rules from them.

For the Zeroconf protocol, we prepare *eight* interoperability testing rules for the fields of a Zeroconf packet: `destination port`, `flags`, `number of answers`, `number of authority records`, `number of additional records`, `answer name`, `TTL` and `record class`. The `id` and `ip address` fields are excluded from the rules.

For the DHCP protocol, we generate *nine* interoperability testing rules for the fields composing a DHCP packet: `destination port`, `message type`, `flags`, `ciaddr`, `yiaddr`, `giaddr`, `chaddr`, `options` and `lease time`. The `op` and `xid` fields are also excluded because they depend on the client and the incoming DHCP packet.

```
1 query{src_port != 5353 AND dst_port = 5353 AND flag.QR = 0x00} ;
2 resp {dst_port = @query.src_port AND flag.QR = 0x80 AND Questions != 0
3    AND S1.data.answer.ttl != S2.data.answer.ttl}
```

(a) Zeroconf interoperability rule

```
1 query{src_port != 5353 AND dst_port = 5353 AND flag.QR = 0x00} ;
2 resp {dst_port = @query.src_port AND flag.QR = 0x80 AND Questions != 0
3    AND S1.data.answer.ttl != S2.data.answer.ttl}
```

(b) DHCP interoperability rule

Figure 6.7: Examples of derived interoperability rules for Zeroconf and DHCP.

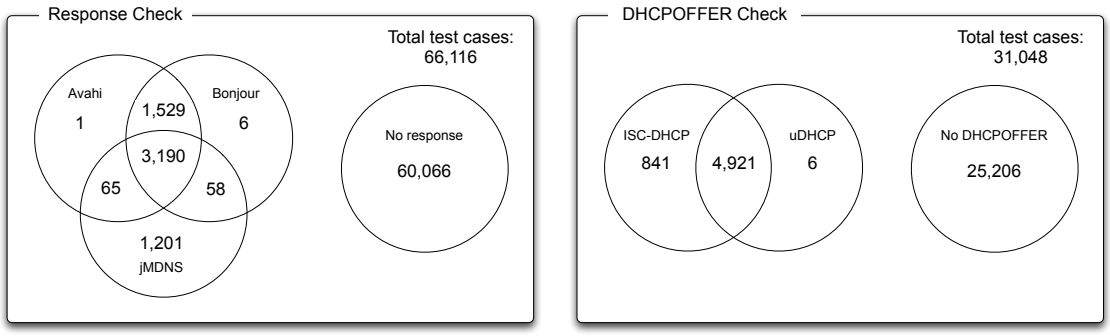
Figure 6.7 gives examples of interoperability rules for both Zeroconf and DHCP. Rules compare two packet streams, i.e. `S1` and `S2`, that store captured input and output packets during replay, each from different daemons, and detect differences. For example, the Zeroconf rule in Figure 6.7 checks the equality of two packet streams in terms of the `TTL` field. The expression starting with `query` (line 1) filters a test query packet, and the statement starting with `resp` (line 2) detects an answer packet for the query. The third line compares the value of the `data.answer.ttl` field of the two packet streams.

6.6.2 Interoperability Test Packet Generation

We generate the input packets for interoperability testing from each implementation by symbolically executing LLVM-compiled implementations. The generated test packets are then replayed against an unmodified daemon as well as other daemons implementing the same protocol specification under the same testing environment. All network traffic generated during the replay phase is captured to check the interoperability between the daemons under testing. We use the test input packets generated in the previous experiments (see Section 6.4.1).

The generated test packets are replayed against multiple implementations of each protocol (i.e. Zeroconf — `Avahi`, `Bonjour` and `JmDNS` and DHCP — `udhcp` and `isc-dhcp`). All input/output packets during the replay phase are stored. We check interoperability between daemons using interoperability rules that compare input/output packets.

Figure 6.8(a) shows the results of SIOT for the three Zeroconf daemons. The daemons ignore 60,066 (out of 66,116) test cases because the test packets are malformed. As all daemons exhibit the same behaviour, these cases do not incur interoperability problems. We find that the daemons show different behaviour in the remaining 6,050 test cases. These test cases include the cases in which (1) all the daemons respond (3,190 test cases); (2) two of them respond; and (3) only one



(a) Zeroconf (b) DHCP

Figure 6.8: Venn diagrams illustrating the results of checking the generated response packets for Zeroconf and DHCP.

daemon sends an answer packet.

Figure 6.8(b) shows the response behaviour of the two DHCP daemons for all test input packets. Both daemons do not generate response packets for 25,206 (out of 31,048) test packets. For test input packets that only one daemon generates a response for, SYMBEXNET reports interoperability bugs.

6.6.3 Detected Interoperability Errors

Using our IOT rules, SYMBEXNET discovered 24 inconsistencies that cause interoperability errors. We describe three of these IOT errors, one from Zeroconf and two from DHCP.

IOT violation 1 (Zeroconf): Incorrect behaviour for packets with inconsistent field sizes. `Bonjour` and `JmDNS` behave incorrectly when they receive a packet claiming to have more data than it actually contains. According to the DNS specification, implementations are expected to verify the size reported in DNS packets. However, the `Bonjour` daemon fails to check the validity of received packets. If `Bonjour` receives a query message with a wrong `AUCOUNT` value, it responds to that packet incorrectly. Note that the `Avahi` daemon ignores the same corrupted packet.

Similarly, SYMBEXNET injects a conflict response packet to test the name conflict function. The correct response includes two answer records (`SRV` and `TXT`). When the `Bonjour` daemon receives a corrupted response message with a wrong `ANCOUNT`, it still processes the packet. Again, the `Avahi` daemon ignores the corrupted response message.

IOT violation 2 (DHCP): Incorrect response to broadcast address.

The DHCP specification states about responding to unicast addresses in Section 4.1:

"If the broadcast bit is not set and 'giaddr' is zero and 'ciaddr' is zero, then the server

> *unicasts DHCPOFFER and DHCPACK messages to the client's hardware address and 'yiaddr' address."* [Dro97]

SYMBEXNET generates a test `DHCPDISCOVER` packet with both `giaddr` and `ciaddr` addresses set to zero and the broadcast bit not set. In this case, the server receiving this `DHCPDISCOVER` packet is supposed to respond with a `DHCPOFFER` message to the client's hardware address and `yiaddr` address. However, if the `isc-dhcp` daemon receives such a test packet, it responds incorrectly with a `DHCPOFFER` message to broadcast address `255.255.255.255`. In contrast, the `udhcp` daemon correctly sends a `DHCPOFFER` message to the client's hardware address and `yiaddr` address.

IOT violation 3 (DHCP): Inconsistency in the response to `DHCPREQUEST` message. The DHCP specification defines the `DHCPNAK` message as follows:

> *"DHCPNAK - Server to client indicating client's notion of network address is incorrect (e.g. client has moved to new subnet) or client's lease as expired [...] If the selected server is unable to satisfy the DHCPREQUEST message (e.g. the requested network address has been allocated), the server SHOULD respond with a DHCPNAK message. "* [Dro97]

If SYMBEXNET marks `chaddr` as symbolic, it generates a `DHCPDISCOVER` message that has a randomly generated address for `chaddr`. Both `udhcp` and `isc-dhcp` correctly respond with a `DHCPOFFER` message to a client if they receive this input. The client then sends a `DHCPREQUEST` message that contains a lease address provided in the `DHCPOFFER` message. However, this time the client uses its correct hardware address because the `DHCPREQUEST` message is not a symbolically generated message but a regular packet that must be handled concretely.

If the daemons receive the `DHCPREQUEST` message requesting the lease address offered to the client, they assume that the requested address is not available because the offered address stored internally is associated with the client's hardware address. In this case, as the above requirement states, the daemon is supposed to send a `DHCPNAK` message.

According to the observed behaviour of both daemons, `isc-dhcp` responds correctly with a `DHCPNAK` message while `udhcp` ignores the `DHCPREQUEST` message. Although this requirement contains the keyword "SHOULD", which can be interpreted as a recommended feature, the implementation of `udhcp` is preferable, otherwise the client has to wait for a timeout before retransmitting a `DHCPREQUEST` message or restarting the initial procedure by sending a `DHCPDISCOVER` message.

Note that the above phrase is not translated to a packet rule because it is associated with state, i.e. the expiration of a client's lease. However, SYMBEXNET detects this issue using IOT.

Table 6.9: A complete list of all detected bugs for each class

Daemon	No.	KLEE(GB)	Rule(SB)	SIOT (IB)	Bug Description
Avahi	1	√			Vulnerability caused by source port number zero
	2		√	√	Generated wrong answer RR fields
	3		√	√	Generated wrong additional RR fields
	4			√	Response to a query with port number 5351
Bonjour	5	√			Source port 0 vulnerability
	6		√	√	Incorrect behaviour for a query (non-zero RCODE)
	7		√	√	Missing records in query packets
	8			√	Query with wrong additional RR is not ignored
JmDNS	9		√	√	Incorrect response for a query with unknown class
	10			√	Missing desired behaviour for OPCODE
	11			√	Wrong TTL value for PTR record
	12			√	Wrong TTL value for TXT record
	13			√	Wrong TTL value for SRV record
	14			√	Response to a query with port number 5351
	15		√	√	Query with non-zero response code is not ignored
	16		√	√	Query with server status request is not ignored
	17		√	√	Query with non-authenticated flag is not ignored
	18		√	√	Query with wrong additional RR is not ignored
	19		√	√	Query with wrong answer RR is not ignored
	20		√	√	Query with unknown class is not ignored
	21		√	√	Generated wrong answer RR fields
	22		√	√	Generated wrong additional RR fields
udhcp	23	√			Out of bound pointer error (`options.c` at line 79)
	24	√			Out of bound pointer error (`options.c` at line 94)
	25	√			Out of bound pointer error (`options.c` at line 99)
	26	√			Out of bound pointer error (`options.c` at line 111)
	27	√			Four bytes read overflow (`dhcpd.c` at line 213)
	28	√			Four bytes read overflow (`dhcpd.c` at line 214)
	29	√			Out of bound pointer error (`dhcpd.c` at line 319)
	30	√			Out of bound pointer error (`serverpacket.c` at line 113)
	31	√			Out of bound pointer error (`serverpacket.c` at line 119)
	32			√	Failed to send `DHCPOFFER` to gateway server
	33			√	Incorrectly generated `DHCPOFFER`
	34			√	Incorrectly ignored `DHCPREQUEST`
	35			√	Incorrect response to unicast address
isc-dhcp	36	√			Out of bound pointer error (`conflex.c` at line 114)
	37	√			Out of bound pointer error (`dhcp.c` at line 205)
	38		√	√	Missing requirement for the broadcast bit
	39		√		Incorrect response to broadcast address
# Total	39	13	15	25	There exist 14 shared bugs

6.7 Discovered Implementation Errors

In this section, we summarise the detected bugs and classify them according to three classes based on the method that is used to discover them: *Generic Bugs (GB)*, *Semantic Bugs (SB)* and *Interoperability Bugs (IB)*.

We evaluate SYMBEXNET with five real-world network protocol implementations. As shown in

the last row of Table 6.9, SYMBEXNET detected 39 unique bugs in the network daemons. Most of these bugs have been confirmed and fixed by the responsible developers. More specifically, SYMBEXNET detected 4 bugs in `Avahi`, 4 bugs in `Bonjour`, 14 bugs in `JmDNS`, 13 bugs in `udhcp` and 4 in `isc-dhcp`. These bugs were found by the checking process of a daemon in SYMBEXNET, i.e. using symbolic execution (GB), rule-based analysis (SB) and interoperability testing (IB). Eleven of the detected bugs using packet rules were also found by interoperability testing. Table 6.9 provides a complete list of all detected bugs for each bug class with their descriptions.

A blind comparison of a field value of response packets causes false positives due to recommended requirements and ranges of field values as described in Section 5.4.2.

These false positivies are prunded by applying derived IOT rules from the DHCP protocol specifications.

Generic Bugs (GB). During symbolic execution, SYMBEXNET detects generic errors. SYMBEXNET found 13 bugs in this category. Most errors are out of bound pointer errors. Two security errors in `Bonjour` and `Avahi` have already been described in Section 6.4.3. For `udhcp`, SYMBEXNET with KLEE was able to reproduce five memory errors that were previously detected by EXE [CGP+06] in addition to four new memory errors.

Semantic Bugs (SB). SYMBEXNET detected 15 bugs classified as semantic bugs. Semantic bugs are difficult to detect because they often require a deep analysis of exchanged packets to verify that the daemon behaves incorrectly. SYMBEXNET uses rule-based packet analysis technique to find these bugs.

Bug #12 in Table 6.9 shows an example of a semantic bug. The bug is related to the `RCODE` field, which is used to get information about the status of a mDNS operation such as a format error (1). The mDNS specification stipulates about RCODE in Section 18.11:

> *"18.11. RCODE (Response Code)*
> *In both multicast query and multicast response messages, the Response Code MUST be zero on transmission. Multicast DNS messages received with non-zero Response Codes MUST be silently ignored."* [CK10]

From the above statement, we derive the following rule:

```
1 query{src_port != 5353 AND dst_port = 5353 AND flag.QR = 0x00
2   AND flag.RCODE != 0x00} ; resp {dst_port = @query.src_port
3   AND flag.QR = 0x80 AND ANY data.answer(name = @query.question.name)}
```

The specification states that an mDNS packet with a non-zero RCODE must be ignored. However, if a `Bonjour` daemon receives a query packet requesting a service with a non-zero RCODE value, it does not ignore the packet but instead returns an answer. When the `Avahi` daemon receives the query packet, it ignores the packet, as expected.

SYMBEXNET prepares a test packet replacing the flags value with `0x02`, which is one of the generated test cases, and sends it to the mDNS daemons. When `Avahi` receives the test packet, it validates the packet by invoking the `avahi_packet_check_valid_mcast` function in `dns.c`. The function has a statement that checks the `RCODE` value as follows:

```
if (flags & AVAHI_DNS_FLAG_RCODE)
    return -1;
```

Therefore, `Avahi` returns `-1` and the packet is ignored. In contrast, `Bonjour` does not check the RCODE value of a received mDNS packet.

Table 6.10: Detected inconsistencies through interoperability testing for Zeroconf daemons.

No.	Description	Test value and its source	Daemons		
			Avahi	Bonjour	JmDNS
1	Source port number with 5351 (assigned for NAT Port Mapping Protocol)	sport=5351 (Bonjour)	O	X	O
2	Query with source port number 0	sport=0 (Avahi &Bonjour)	abort	abort	O
3	Query with non-zero Response code is not ignored	flag=0x0002 (Avahi)	X	O	O
4	Query with server status request is not ignored	flag=0x1000 (Avahi &Bonjour)	X	X	O
5	Query with non-authenticated data flag is not ignored	flag=0x1010 (Bonjour)	X	X	O
6	Query with opcode for inverse query is not ignored	flag=0x0800 (Bonjour)	X	X	O
7	Query with wrong number in Additional RRs	AddRRs=2 (Avahi)	X	O	O
8	Query with wrong number in Answer RRs	AnswerRRs=2 (Avahi &Bonjour)	X	X	O
9	Query with unknown class is not ignored	class=unknown (Avahi &Bonjour)	X	X	O
10	Different number of records in Answer RRs field	any value incurring response	4	1	4
11	Different number of records in Additional RRs field	any value incurring response	0	3	0
12	Destination port number of response packet must not 5353	any value incurring response	$\neq 5353$	$\neq 5353$	5353
13	TTL values of PTR record are different	any value incurring response	10 sec	10 sec	1 hr
14	TTL values of TXT record are different	any value incurring response	10 sec	10 sec	1 hr
15	TTL values of SRV record are different	any value incurring response	10 sec	10 sec	1 hr

Interoperability Bugs (IB). SYMBEXNET found 25 bugs classified in this bug class. Table 6.10 shows the summary of detected inconsistencies for Zeroconf daemons after conducting interoperability testing.

For example, the bug in the sixth row of Table 6.10 is related to the `OPCODE` field, which is originally defined in the DNS specification [Pau87b] as a command given to the DNS server requesting some action, such as a query (`0x00`), an inverse query (`0x01`) or a server status request (`0x02`). The mDNS specification states about `OPCODE` in Section 18.3:

> *"18.3. OPCODE*
> *In both multicast query and multicast response messages, it MUST be zero."* [CK10]

The specification states the values for `OPCODE` but does not describe how to proceed when the value is not zero. Developers can implement a daemon differently to handle a query packet with a non-zero `OPCODE`, either ignoring the packet or the value. Both `Avahi` and `Bonjour` ignore a packet with a non-zero OPCODE. However, `JmDNS` behaves differently from the other two daemons. It replies with a query with the `OPCODE` value of `0x02`. We have reported this bug to the `JmDNS` developers who confirmed it. The latest version of `JmDNS` behaves as the other two daemons.

6.8 Summary

This chapter has presented the experimental evaluation of SYMBEXNET. The first experiment showed the quality of the test packets generated by SYMBEXNET for stateless network protocol implementations. The number of generated test packets and their achieved source coverage were compared with random testing. In general, SYMBEXNET-generated test packets achieve high source coverage and effectively detect non-trivial errors in the tested network protocol implementations. The second experiment showed that the proposed method for stateful network protocol implementations can explore deep code paths that can only be reached after exchanging several packets. The results from the third experiment illustrated that SYMBEXNET's IOT approach can effectively check interoperability between multiple implementations of the same network protocol.

Our experience with using SYMBEXNET has yielded several insights. The majority of detected violations are caused by different interpretations of the same specification. Since ambiguities in the specification may lead to problems such as incorrect functionality, interoperability errors and security problems, it is important to eliminate and detect them from specifications and implementations. By translating textual specifications into verifiable rules, one can eliminate such ambiguities. Since rules only need to be extracted from a specification once, this can be done by domain experts who can resolve ambiguities correctly. Developers can detect ambiguities that lead to different implementations or incorrect functionality by comparing response behaviour for test input packets generated by SYMBEXNET.

Since SYMBEXNET uses a blackbox approach, it cannot reason about the internal state of the network implementation. Any requirements referring to the internal state maintained by the implementation are ignored because they are not visible externally. However, executing multiple

implementations for the same protocol specification and comparing their response behaviour for the same test input packet enable SYMBEXNET to check some parts of functions handling internal states.

We generated test packets from both `Avahi` and `Bonjour` and replayed them against the `JmDNS` daemon, which resulted in the discovery of a range of violations. SYMBEXNET can be used to check network implementations whose source code is not available, as long as appropriate test packets have been generated by other implementations of the same network protocol. Since networks may run legacy daemons without the availability of source code, a behavioral checking technique that are not depend on the availability of source code can be a practical solution to these legacy situations.

The number of generated test packets, the runtime and memory consumption of SYMBEXNET, and the source code coverage achieved are all dependent on the amount of symbolic input in packets. Making all fields in an input packet symbolic is usually not possible because it can lead to path explosion. However, our approach of systematically making all possible combinations of fields symbolic, starting with only one field and gradually making more fields symbolic, seems to achieve a good trade-off between run time and code coverage.

Unlike stateless network protocol implementations, stateful implementations that exchange a series of packets to perform a task are more difficult to explore. MPE-SE, our approach for performing symbolic execution repeatedly on selected symbolic inputs, guides the implementation to reach deep execution paths that can only be explored after exchanging specifically ordered packets. This generates sequences of packet that can achieve broad and deep exploration of program code paths.

In summary, SYMBEXNET generates high quality test suites for checking network protocol implementations. With the rule-based packet inspection technique, SYMBEXNET detects various types of flaws in implementations of network protocols. The generated test packets from multiple implementations of the same network protocol can also be used to check the interoperability between the implementations.

CHAPTER 7

Conclusions

IMPLEMENTATIONS of modern network protocols, such as DNS and DHCP, are prone to flaws and security vulnerabilities caused by ambiguous requirements in protocol specifications. Such problems are hard to detect because they are often triggered by complex sequences of packets that occur only after prolonged operation. Traditional verification and testing approaches have been used to detect problems in network protocol implementations. However, it is difficult to detect complex problems using these techniques. We found symbolic execution to be a powerful technique for generating test inputs that explore a protocol implementation in order to check it against its specification. Although recent symbolic execution techniques improve source code coverage while avoiding computational cost, they are often limited to explore shallow execution states of network protocol implementations because deep states can only be reached after exchanging a series of packets.

In this thesis, we described SYMBEXNET, a practical checking system for network protocol implementations. A set of packet rules derived from a standard protocol specification and a set of input packets using symbolic execution enable SYMBEXNET to discover violations in real-world network daemon implementations as well as interoperability problems. SYMBEXNET leverages exploration methods based on symbolic execution to generate high quality test packets. We found that these test input packets can also be used to check the interoperability of network protocol implementations.

SYMBEXNET combines symbolic execution with automata-based rule checking and is capable of discovering different types of network protocol flaws in implementations. To explore complex packet exchange sequences, we developed an exploration method (MPE-SE) that repeatedly performs symbolic execution on selected test input packets. To check interoperability, we introduced a method that detects behavioural deviations between implementations of the same network protocol specification.

We began with an overview of the formal techniques that can be used to verify network protocols. In order to understand the benefits of formal verification as well as their limitations, we applied one of the techniques, model checking, to a network management protocol that is widely used to

configure cellular 3G base stations. Our case study reveals that verification can rigorously detect incorrect behaviour but is limited to trivial deployment scenarios.

SYMBEXNET enables developers to create a link between specifications and implementations of network protocols. The input to SYMBEXNET is the C source code of a network protocol implementation and a set of rules extracted from the specification that describes invalid patterns of input and output packets. Using symbolic execution, SYMBEXNET generates an exhaustive set of input packets and sequences that can achieve broad and deep exploration of program execution paths. It then replays the test packets against the implementation and uses a rule-based packet analyser to detect rule violations indicating implementation errors.

In order to automatically generate effective interoperability test packets, we introduced symbolic interoperability testing (SIOT), a novel interoperability testing method that: (1) generates a set of test packets for IOT by integrating the generated test packets from each implementation under test using symbolic execution; (2) replays the test packets across multiple implementations and records all input and output network traffic; and (3) compares the behaviour of these implementations to check if they are interoperable. Since the implementations are developed based on the same protocol specification and tested under the same conditions, they are expected to behave in the same manner. For checking interoperability, SYMBEXNET uses an extended rule-based packet analysis technique that is able to compare the behaviour of multiple implementations.

These techniques enable SYMBEXNET to check automatically a network implementation against its protocol specification and discover three different types of violations: *generic errors*, *semantic errors* and *interoperability errors*. We implemented SYMBEXNET and validated it on multiple network protocol implementations of two protocols: the *Zeroconf* network configuration protocol and the *DHCP* protocol. SYMBEXNET successfully detected a total of 39 non-trivial errors in the evaluated implementations. Most of these errors have been confirmed and fixed by the responsible developers.

7.1 Future Work

SYMBEXNET can be extended in a number of directions. First, it can achieve higher code coverage through the development of smart symbolic marking strategies. Next, SYMBEXNET can be enhanced to check mobile applications. Finally, SYMBEXNET can be extended to become a framework providing fully automated network service verification across multiple network hosts.

Enhanced Conformance Testing. To ensure the reliability of network protocols, they must be checked for conformance with their protocol specifications. Usually test packets suites are derived from an abstract model such as a finite state machine (FSM). The implementation is tested for conformance by applying test packets and verifying the corresponding outputs. However, confor-

mance testing suites are not sound and often unable to detect critical errors [Tre94]. In addition, testing suites provide only limited coverage [TCM98].

We propose developing a verification methodology that can be used for generating high-coverage conformance testing suites. The methodology can be viewed as a combination of conformance testing and symbolic execution, thereby providing a means for generating test input cases that enhance conformance testing and can achieve high testing coverage. Similar to SYMBEXNET, the proposed method could generate a conformance test suite by symbolically executing a protocol implementation with carefully selected conformance testing inputs.

Verification Framework for Mobile Applications. Mobile applications are increasingly ubiquitous. However, there are concerns about how third-party mobile applications may misuse or improperly handle privacy-sensitive data. Although many providers and researchers try to provide secure application frameworks for today's mobile devices, this is still an open problem [EGC+10, GE11, SFK+10].

We believe that new verification frameworks are required to cope with these types of security problems. Current mobile application markets distribute applications in a centralised way. Therefore introducing an application verification step could improve significantly the security of mobile devices. While the choice of verification technique remains an open question, symbolic execution could be used to explore all possible code paths of an application and examine whether any privacy-sensitive data could be leaked.

Predictive Runtime Verification. Verification is a rigorous way for discovering incorrect behaviour but may suffer from the state explosion problem. This problem precludes its use in non-trivial deployment scenarios. As a solution, we propose a new verification architecture for autonomous networks that exploits runtime verification [BS01, HR01, KLS+02]. The approach aims to combine the scalability of simulations with the rigour of verification techniques such as model checking and symbolic execution. Within such an architecture, a *runtime verifier* is embedded into network service daemons and continuously monitors and checks the network state against desired properties to provide correctness guarantees.

The main idea is to apply *runtime verification* techniques to verify autonomous networks. Runtime verification applies verification techniques during program execution to detect faults. A runtime verifier periodically checks the correctness of a protocol and implementation at runtime. During normal system operation, runtime verification observes the system's input and output behaviour to verify given properties. Desired correctness properties are checked in potential future states of the network model derived from the protocol. Since this results in a bounded state depth without considering the entire state space, it minimises the number of explored states and thus avoids the state explosion problem. When a potential violation is detected, a fault-avoidance mechanism such as ignoring an input packet causing the violation could be used to influence the operation of the network.

Bibliography

[3GP08] 3GPP TS 32.500. Telecommunication Management; Self-Organizing Networks (SON); Concepts and Requirements (Rel. 8), 2008.

[AD97] S. Alexander and R. Droms. IETF RFC 2132: DHCP Options and BOOTP Vendor Extensions, March 1997.

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[AGE+04] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre. D-itg distributed internet traffic generator. In *Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, QEST '04, pages 316–317, Washington, DC, USA, 2004. IEEE Computer Society.

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[AHM+08] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, September 2008.

[Ait02] Dave Aitel. The advantages of block-based protocol analysis for security testing. Technical report, Immunity Inc., February 2002.

[AL03] Paul C. Attie and David H. Lorenz. Correctness of model-based component composition without state explosion. In *Proceedings of the Workshop on Correctness of Model-based Software Composition European Conference on Object-Oriented Programming (ECOOP) 2003*, 2003.

[App] Apple. Apple's bonjour. `http://developer.apple.com/opensource/`.

[APRS93] Noriyasu Arakawa, Marc Phalippou, Nathalie Risser, and Terunao Soneoka.

Combination of conformance and interoperability testing. In *Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols: Formal Description Techniques, V*, FORTE '92, pages 397–412. North-Holland Publishing Co., 1993.

[Ava] Avahi. Avahi Zeroconf. `http://www.avahi.org`.

[AW01] Rajeev Alur and Bow-Yaw Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 169–181, London, UK, 2001. Springer-Verlag.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 351–366. Springer Berlin / Heidelberg, 2008.

[BDG+07] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1100–1102, New York, NY, USA, 2007. ACM.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Selecta formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, New York, NY, USA, 1975. ACM.

[BFN+05] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 265–276, New York, NY, USA, 2005. ACM.

[BG01] Sami Beydeda and Volker Gruhn. Integrating white- and black-box techniques for class-level regression testing. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 357–362, Washington, DC, USA, 2001. IEEE Computer Society.

[BHK+07] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome,

Pongsin Poosankam, Dawn Song, and Heng Yin. Bitscope: Automatically dissecting malicious binaries. Technical report, In CMU-CS-07-133, 2007.

[BKPR02] Paul Barford, Jeffery Kline, David Plonka, and Amos Ron. A signal analysis of network traffic anomalies. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 71–82, New York, NY, USA, 2002. ACM.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. IETF RFC 1945: Hypertext Transfer Protocol – HTTP/1.0, May 1996.

[BM83] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM System Journal*, 22(3):229–245, September 1983.

[BMG99] Andrew Begel, Steven McCanne, and Susan L. Graham. Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '99, pages 123–134, New York, NY, USA, 1999. ACM.

[BOG02] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49:538–576, July 2002.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30:775–802, June 2000.

[Bra97] S. Bradner. IETF RFC 2119: Key Words for Use in RFCs to Indicate Requirement Levels, 1997.

[BS80] G. Bochmann and C. Sunshine. Formal methods in communication protocol design. *Communications, IEEE Transactions on*, 28(4):624–631, April 1980.

[BS01] Mike Barnett and Wolfram Schulte. Spying on components: A runtime verification technique. In *Proceedings of the Workshop on Specification and Verification of Component- Based Systems OOPSLA 2001*, page 7, 2001.

[BSMV06] Joao Bispo, Ioannis Sourdis, Joao M.P.Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 119–126, December 2006.

[BU91] Barry S. Bosik and M. Ümit Uyar. Finite state machine based formal methods in protocol conformance testing: from theory to implementation. *Computer Networks and ISDN Systems*, 22(1):7–33, August 1991.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CFF+06] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal Parallel Programming*, 34:61–91, February 2006.

[CG85] B. Croft and J. Gilmore. IETF RFC 951: Bootstrap Protocol (BOOTP), September 1985.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16:1512–1542, September 1994.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM.

[CK10] Stuart Cheshire and Marc Krochmal. IETF Internet Draft: Multicast DNS, March 2010.

[CR05] F. Chen and G. Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550. Springer, 2005.

[CTCC98] Huo Yan Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7:250–295, July 1998.

[CTS08] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28:626–643, December 1996.

[Dan08] Dan Kaminsky. Black ops 2008 - its the end of the cache as we know it, Black Hat USA, 2008.

[DHH+06] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2006.

[Dil90] Laura K. Dillon. Using symbolic execution for verification of ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12:643–669, October 1990.

[DKN+10] M. Duflot, M. Kwiatkowska, G. Norman, D. Parker, S. Peyronnet, C. Picaronny, and J. Sproston. Practical applications of probabilistic model checking to communication protocols. In *Handbook of Formal Methods in Industrial Critical Systems*. IEEE Computer Society Press, 2010.

[DKNP06] Marie Duflot, Marta Kwiatkowska, Gethin Norman, and David Parker. A formal analysis of bluetooth device discovery. *International Journal on Software Tools for Technology Transfer*, 8:621–632, October 2006.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52:365–473, May 2005.

[DR02] R. Dill and M. Ramsay. udhcp client/server package (2002). `http://udhcp.busybox.net/`, 2002.

[Dro97] R. Droms. IETF RFC 2131: Dynamic Host Configuration Protocol, March 1997.

[DSU90] A.T. Dahbura, K.K. Sabnani, and M.U. Uyar. Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE*, 78(8):1317 –1326, August 1990.

[DV08] Alexandra Desmoulin and César Viho. Automatic interoperability test case generation based on formal definitions. In *Proceedings of the 12th international conference on Formal Methods for Industrial Critical Systems*, FMICS'07, pages 234–250, Berlin, Heidelberg, 2008. Springer-Verlag.

[DV09] Alexandra Desmoulin and Csar Viho. Formalizing interoperability for test case generation purpose. *International Journal on Software Tools for Technology Transfer (STTT)*, 11:261–267, June 2009.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI'10,

pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '94, pages 87–96, New York, NY, USA, 1994. ACM.

[FB05] N. Feamster and H. Balakrishnan. Correctness properties for Internet routing. In *43rd Annual Allerton Conference Communication, Control, and Computing Monticello, Il*, September 2005.

[Fea04] Nick Feamster. Practical verification techniques for wide-area routing. *SIGCOMM Computer Communication Review*, 34:87–92, January 2004.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[FJJV97] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29:123–146, July 1997.

[FM00] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 59–68, Berkeley, CA, USA, 2000. USENIX Association.

[For03] Behrouz A. Forouzan. *Data Communications and Networking*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[FQ03] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *SPIN'03*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224, Berlin, Heidelberg, 2003. Springer-Verlag.

[GE11] Damianos Gavalas and Daphne Economou. Development platforms for mobile applications: Status and trends. *IEEE Software*, 28(1):77–86, January 2011.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[GL11] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on*

Software Testing and Analysis, ISSTA '11, pages 23–33, New York, NY, USA, 2011. ACM.

[GLM08a] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceeding of Network and Distributed System Security Symposium*. The Internet Society, 2008.

[GLM08b] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[GS05] Timothy G. Griffin and Joäo Luís Sobrinho. Metarouting. *SIGCOMM Computer Communication Review*, 35:1–12, August 2005.

[Han70] K. V. Hanford. Automatic generation of test cases. *IBM System Journal*, 9(4):242–257, December 1970.

[HBBR10] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 143–146, New York, NY, USA, 2010. ACM.

[HEKSP11] Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Seep: exploiting symbolic execution for energy-aware programming. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, pages 4:1–4:5, New York, NY, USA, 2011. ACM.

[HK07] D. Hoffman and N. Kube. Automated testing of SCADA protocols. In *Proceedings of Automated testing of SCADA protocols Symposium*, pages 4.1–4.13, 2007.

[HLSG04] Ruibing Hao, David Lee, Rakesh K. Sinha, and Nancy Griffeth. Integrated system interoperability testing with applications to voip. *IEEE/ACM Transactions on Networking*, 12:823–836, October 2004.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, May 1997.

[Hol02] Gerard J. Holzmann. Uno: Static source code checking for userdefined properties. In *In 6th World Conference on Integrated Design and Process Technology, IDPT 02*, 2002.

[How77] W.E. Howden. Symbolic testing and the dissect symbolic evaluation system.

Software Engineering, IEEE Transactions on, SE-3(4):266 – 278, July 1977.

[HR01] K. Havelund and G. Rosu. Java pathexplorer–A runtime verification tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey, Montreal, Canada*, 2001.

[HWC+10] Daniel Hoffman, Hong-Yi Wang, Mitch Chang, David Ly-Gagnon, Lewis Sobotkiewicz, and Paul Strooper. Two case studies in grammar-based test generation. *Journal of Systems and Software*, 83(12):2369–2378, December 2010.

[IAIK02] S. Ioannidis, K.G. Anagnostakis, J. Ioannidis, and A.D. Keromytis. xpf: packet filtering for low-cost network monitoring. In *High Performance Switching and Routing, 2002. Merging Optical and IP Technologies. Workshop on*, pages 116–120, 2002.

[Int] Internet Software Consortium. ISC DHCP. `http://www.isc.org/software/dhcp`.

[ISK06] S.M.S. Islam, M.H. Sqalli, and S. Khan. Modeling and Formal Verification of DHCP Using SPIN. *International Journal of Computer Science and Applications (IJCSA)*, 3:145–159, June 2006.

[KC00] O. Kon and R. Castanet. Test generation for interworking systems. *Computer Communications*, 23(7):642–652, 2000.

[KDY+06] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 339–350, New York, NY, USA, 2006. ACM.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19:385–394, July 1976.

[KKPB07] Martin Karsten, S. Keshav, Sanjiva Prasad, and Mirza Beg. An axiomatic basis for communication. *SIGCOMM Computer Communication Review*, 37:217–228, August 2007.

[Kle01] J. Klensin. IETF RFC 2821: Simple Mail Transfer Protocol, April 2001.

[KLS+02] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. *Electronic Notes in Theoretical Computer Science*, 70(4):95–111, 2002.

[KSK00] Sungwon Kang, Jaehwi Shin, and Myungchul Kim. Interoperability test suite

derivation for communication protocols. *Computer Networks*, 32:347–364, March 2000.

[KSM96] James D. Kindrick, John A. Sauter, and Robert S. Matthews. Improving conformance and interoperability testing. *StandardView*, 4(1):61–68, March 1996.

[KVB04] Gunjan Khanna, Padma Varadharajan, and Saurabh Bagchi. Self Checking Network Protocols: A Monitor Based Approach. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 18–30, Washington, DC, USA, 2004. IEEE Computer Society.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO'04)*, Washington, DC, USA, March 2004. IEEE Computer Society.

[Law94] Lawrence Berkeley National Labs. libpcap, June 1994.

[LGW+08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008.

[Lic] GNU GPL License. Gcov: Gnu coverage tool. `http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html`.

[LS06] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *Proceedings of the 18th IFIP TC6/WG6.1 international conference on Testing of Communicating Systems*, TestCom'06, pages 19–38. Springer-Verlag, 2006.

[ME04] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 155–168, Berkeley, CA, USA, 2004. USENIX Association.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[MG08] Stephane Maag and Cyril Grepet. Interoperability testing of a manet routing protocol using a node self-similarity approach. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1908–1912, New York, NY, USA, 2008. ACM.

[MHD+07] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. Melange: creating a "functional" internet. *SIGOPS Operating Systems Review*, 41(3):101–114, March 2007.

[MHL94] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *Network, IEEE*, 8(3):26 –41, May 1994.

[MJ93] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 259–269, Berkeley, CA, USA, 1993. USENIX Association.

[MRW03] S. Moseley, S. Randall, and A. Wiles. Experience within etsi of the combined roles of conformance testing and interoperability testing. In *Standardization and Innovation in Information Technology, 2003. The 3rd Conference on*, pages 177–189, October 2003.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[MSS97] Masoud Mansouri-Samani and Morris Sloman. GEM: A Generalized Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[NFLTJ06] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jezequel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32:140–155, March 2006.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, 1992. Springer-Verlag.

[Pau87a] Paul Mockapetris. IETF RFC 1034: Domain Names - Concepts and facilities, November 1987.

[Pau87b] Paul Mockapetris. IETF RFC 1035: Domain Names - Implementation and specification, November 1987.

[Pau93] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *In Logic and Computer Science*, 1993.

[Pax98] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*,

pages 2435–2463, Berkeley, CA, USA, 1998. USENIX Association.

[PDEP08] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[Pos81a] J. Postel. IETF RFC 792: Internet Control Message Protocol, September 1981.

[Pos81b] J. Postel. IETF RFC 793: Transmission Control Protocol, September 1981.

[PPP08] Giorgos Pentafronimos, Spyridon Papastergiou, and Nineta Polemi. Interoperability testing for e-government web services. In *Proceedings of the 2nd international conference on Theory and practice of electronic governance*, ICEGOV '08, pages 316–321, New York, NY, USA, 2008. ACM.

[PYRK11] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.

[RA00] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 156–165, Washington, DC, USA, 2000. IEEE Computer Society.

[Ray82] David Rayner. A system for testing protocol implementations. *Computer Networks (1976)*, 6(6):383–395, 1982.

[Roe99] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[SB99] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 1–13, New York, NY, USA, 1999. ACM.

[SC05] Daniel H Steinberg and Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, December 2005.

[Sch81] Daniel Schwabe. Formal specification and verification of a connection establishment protocol. In *Proceedings of the seventh symposium on Data communications*, SIGCOMM '81, pages 11–26, New York, NY, USA, 1981. ACM.

[SEJK08] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang:

fast and scalable deep packet inspection with extended finite automata. *SIGCOMM Computer Communication Review*, 38:207–218, August 2008.

[SFK+10] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, March 2010.

[SGE00] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):133–166, April 2000.

[SKKR03] Soonuk Seol, Myungchul Kim, Sungwon Kang, and Jiwon Ryu. Fully automated interoperability test suite derivation for communication protocols. *Computer Networks*, 43(6):735–759, 2003.

[Sla74] James R. Slagle. Automated theorem-proving for theories with simplifiers commutativity, and associativity. *J. ACM*, 21:622–642, October 1974.

[SLS05] Usa Sammapun, Insup Lee, and Oleg Sokolsky. Rt-mac: Runtime monitoring and checking of quantitative and probabilistic properties. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '05, pages 147–153, Washington, DC, USA, 2005. IEEE Computer Society.

[SMCP11] JaeSeung Song, Tiejun Ma, C. Cadar, and P. Pietzuch. Rule-based verification of network protocol implementations using symbolic execution. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–8, August 2011.

[Smi96] M. A. S. Smith. Formal verification of communication protocols. In *IFIP TC6/ 6.1 international conference on formal description techniques IX/protocol specification, testing and verification XVI on Formal description techniques IX : theory, application and tools: theory, application and tools*, pages 129–144, London, UK, UK, 1996. Chapman & Hall, Ltd.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed Complex Event Processing with Query Rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[SMP10] JaeSeung Song, Tiejun Ma, and P. Pietzuch. Towards Automated Verification of Autonomous Networks: A Case Study in Self-Configuration. In *8th IEEE International Conference on, Pervasive Computing and Communications Workshops*, April 2010.

[SPMS09] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 225–236, New York, NY, USA, 2009. ACM.

[SSRL07] Oleg Sokolsky, Usa Sammapun, John Regehr, and Insup Lee. Runtime verification for wireless sensor network applications. In *Runtime Verification*, volume 07011 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[Ste93] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[Stu10] Stuart Cheshire, Marc Krochmal and Apple Inc. IETF Internet Draft: DNS-Based Service Discovery, March 2010.

[SWSC07] M. Schmidt, A. Wilde, A. Schulke, and H. Costa. IMS interoperability and conformance aspects [IP Multimedia Systems (IMS) Infrastructure and Services]. *Communications Magazine, IEEE*, 45(3):138 –142, March 2007.

[Tan02] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[TCM98] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test case generation: The optimisation-based approach. In *In International Workshop on Dependable Computing and Its Applications*, pages 169–180, 1998.

[Tre94] Jan Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[Tre99] Jan Tretmans. Testing concurrent systems: A formal approach. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 46–65. Springer-Verlag, 1999.

[ULF08] Octavian Udrea, Cristian Lumezanu, and Jeffrey S. Foster. Rule-based Static Analysis of Network Protocol Implementations. *Information and Computation*, 206:130–157, February 2008.

[VB94] Gert Vermeer and Hans Blik. Interoperability testing: Basis for the acceptance of communication systems. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 315–330, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[VBT01] César Viho, S. Barbin, and Lénaick Tanguy. Towards a formal framework for interoperability testing. In *Proceedings of the IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems*, FORTE '01, pages 53–68, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[Vit] Stas Vitkovsky. dhquery. `http://code.google.com/p/dhquery/`.

[VRA+07] A. Vallejo, J. Ruiz, J. Abella, A. Zaballos, and J.M. Selga. State of the art of ipv6 conformance and interoperability testing. *Communications Magazine, IEEE*, 45(10):140 –146, October 2007.

[WBLS09] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, PADL '09, pages 61–75, Berlin, Heidelberg, 2009. Springer-Verlag.

[WD01] D. Wagner and R. Dean. Intrusion Detection via Static Aanalysis. In *Proceedings IEEE Symposium on Security and Privacy*, pages 156–168, Washington, DC, USA, 2001. IEEE Computer Society.

[XZM+05] G.G. Xie, Jibin Zhan, D.A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2170 – 2183 vol. 3, March 2005.

[YCD+06] Fang Yu, Zhifeng Chen, Yanlei Diao, T.V. Lakshman, and R.H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 93 –102, December 2006.

[YKKK09] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[YKL04] Fang Yu, R.H. Katz, and T.V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 174 – 183, October 2004.

[Yua88] M.C. Yuang. Survey of protocol verification techniques based on finite state machine models. In *Computer Networking Symposium, 1988., Proceedings of the*, pages 164–172, April 1988.

[Zav08] Pamela Zave. Understanding sip through model-checking. In *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, pages 256–279. Springer-Verlag, Berlin, Heidelberg, 2008.

[ZC10] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 321–334, New York, NY, USA, 2010. ACM.

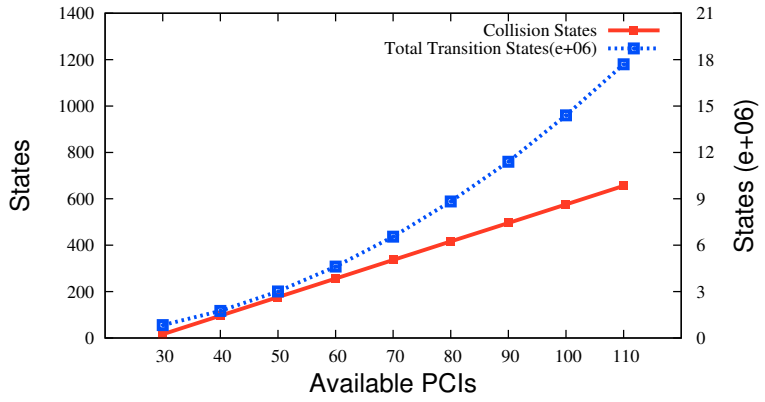
APPENDIX A

Evaluation Results for Case Study

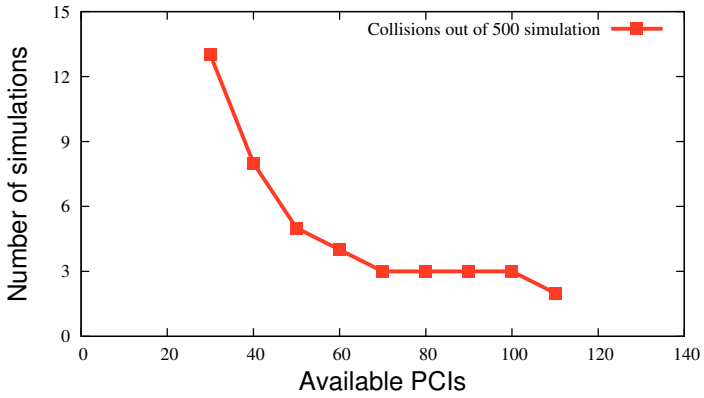
The goal of this experimental evaluation is to evaluate the feasibility of formal verification compared to simulation. We use the SPIN model checker to search the state space for collision or confusion violations of the PCI selection algorithm described in Section 2.3.1. We choose the minimum number of available PCIs to be 30 to ensure enough PCIs for the 5×5 matrix. For the case study scenario, we limit the maximum number of available PCIs to 110 to reduce the number of verifiable states. We use this range of PCIs in the experiments unless otherwise stated. For simulation results, we run each experiment 500 times to ensure a reasonable number of samples.

Single base station scenario. In this scenario, one eNB is deployed in a 5×5 matrix and the other 24 places have pre-deployed eNBs. The self-configuration requirement is that the PCI of the new eNB is collision- and confusion-free in relation to the other cells. We evaluate this scenario according to the PCI assignment algorithm (described in Section 2.3.1) in three cases: (1) PCI LIST: the eNB can only receive a valid list of available PCIs from the OAM; (2) NEIGHBOUR DETECT: the eNB can detect its neighbours' PCIs; and (3) X2: eNBs can exchange neighbour information over the X2 interface.

The verification results show that for (1) PCI LIST, there are 8 collisions and 16 confusions; for (2) NEIGHBOUR DETECT, there are no collisions and 16 confusions; and for (3) X2, there are no collisions and confusions. It shows the potential number of states that the deployed eNB has with collisions and confusions. When the eNB cannot detect neighbours' PCIs and communicate with other eNBs through the X2 interface (PCI LIST), both collisions and confusions may occur. When the eNB is capable of detecting its neighbours' PCIs (NEIGHBOUR DETECT), the collision-free property can be guaranteed but confusions may still happen due to the lack of PCI information about neighbours' neighbours. When the X2 interface is added (X2), there are no collisions or confusions. This result provides evidence for the usefulness of the 3GPP's recommendation of neighbour detection and the X2 interface.



(a) Verification of two concurrent eNBs deployment



(b) Simulation of two concurrent eNBs deployment

Figure A.1: Verification and simulation results for two deployed eNBs

Multiple base station scenario. We consider two eNBs that are deployed concurrently as neighbours. We assume that all optional extensions are implemented by each eNB, which is why we do not expect collisions or confusions.

Figure A.1(a) presents the verification results in terms of potential collisions and the total number of state transitions. The x-axis shows the number of available PCIs, the y-axis shows the collisions (left) and the total state transitions (right). As the number of available PCIs increases, the number of collision states also increases linearly and the total transition states of the model increase faster than the collision states. This indicates that, with more PCIs, the possibility of collisions occurring is lower, but they still exist.

Figure A.1(b) shows the simulation results of the average number of occurred collisions. We can see that, as we increase the number of available PCIs, the number of detected collisions decreases. This implies that when there are more PCIs, the possibility of collisions can be reduced. Simulation

cannot detect all potential collisions when there are many candidate PCIs, while verification can always detect all possible collision states. For assessing the collision-free property, verification is more rigorous.

Overall, both our verification and simulation results provide evidence that collisions and confusions cannot be avoided due to the concurrent PCI selection, which has not been addressed in the 3GPP specification.

APPENDIX B

Packet Rules

This appendix lists the packet rules used in the experiments. The derived packet rules for the Zeroconf and DHCP protocols are listed in Section B.1 and B.2, respectively.

B.1 Packet Rules for the Zeroconf Protocol

The following gives the full list of rules for the Zeroconf protocol.

```
#Rule1:
# If the source UDP port in a received Multicast DNS Query is not port 5353, this
# indicates that the client originating the query is a simple client that does not
# fully implement all of Multicast DNS. In this case, the Multicast DNS Responder
# MUST send a UDP response directly back to the client, via unicast,  to the query
# packet's source IP address and port.
==
pkt1 {pkt1.sport != 0x14e9 AND pkt1.dport = 0x14e9 AND pkt1.flag.qr =  0x00} ;
pkt2 {pkt2.flag.QR = 0x80 AND pkt2.data.qu = @pkt1.data.qu AND pkt2.dport != @pkt1.sport}
MSG: MDNS Responder MUST send a UDP response directly to the query packet's source port.

#Rule2:
# If the source UDP port in a received Multicast DNS Query is not port 5353, this
# indicates that the client originating the query is a  simple client that does not
# fully implement all of Multicast DNS. In this case, the Multicast DNS Responder
# MUST send a UDP response directly back to the client, via unicast, to the query
# packet's source IP address and port.
==
pkt1 {pkt1.sport != 0x14e9 AND pkt1.dport = 0x14e9 AND pkt1.flag.qr = 0x00} ;
pkt2 {pkt2.flag.QR = 0x80 AND pkt2.data.qu = @pkt1.data.qu AND pkt2.d_ip != @pkt1.s_ip}
MSG: MDNS Responder MUST send a UDP response directly to the query packet's source ip.
```

```
# Rule3:
# In multicast responses, including gratuitous multicast responses, the Query ID
# MUST be set to zero on transmission, and MUST be ignored on reception.
==
pkt1 {pkt1.destip = 0xe00000fb AND pkt1.dport = 0x14e9 AND pkt1.flag.QR = 0x80
   AND pkt1.id != 0x00}
MSG: The Query ID of response packet to multicase request MUST be set to zero.

# Rule4:
# In unicast response messages generated specifically in response to a particular
# (unicast or multicast) query, the Query ID MUST match the ID from the query message.
==
pkt1 {pkt1.flag.QR = 0x00 AND pkt1.data.qu.type != 0x00ff} ;
pkt2 {pkt2.destip != 0xe00000fb AND pkt2.flag.QR = 0x80 AND pkt2.data.an ^ @pkt1.data.qu
    AND pkt2.id != @pkt1.id}
MSG: The Query ID of response packet to multicase request MUST match the ID from the
    query message.

# Rule5:
# In response messages for Multicast Domains, the Authoritative Answer bit MUST be
# set to one.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.flag.authoritative != 0x04}
MSG: In response messages, the Authoritative Answer bit MUST be set to one.

# Rule6:
# In multicast response messages, the TC bit MUST be zero on transmission, and MUST
# be ignored on reception.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.flag.truncated != 0x00}
MSG: The TC bit MUST be zero on transmission.

#Rule7:
#  In both multicast query and multicast response messages, the Recursion Available
# bit MUST be zero on transmission.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.flag.recurA != 0x00}
MSG: The Recursion Available bit MUST be zero on transmission.

#Rule8:
#  In both query and response messages, the Zero bit MUST be zero on transmission.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.flag.z != 0x00}
MSG: The Zero bit MUST be zero on transmission.
```

```
#Rule9:
#  In both multicast query and multicast response messages the Authentic Data bit
# MUST be zero on transmission
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.flag.answerAuth != 0x00}
MSG: The Authentic Data bit MUST be zero on transmission.

#Rule10:
# In both multicast query and multicast response messages, the Checking Disabled bit
# MUST be zero on transmission,
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.flag.cd != 0x00}
MSG: The Checking Disabled bit MUST be zero on transmission.

#Rule11:
# In both multicast query and multicast response messages, the Response Code
# MUST be zero on transmission.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.flag.rCode != 0x00}
MSG: The Response Code MUST be zero on transmission.

#Rule12:
# The resource record TTL given in a legacy unicast response SHOULD NOT be greater
# than ten seconds, even if the true TTL of the Multicast DNS resource record is higher.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ;
pkt2 {pkt2.destip != 0xe00000fb AND pkt2.flag.qr = 0x80 AND pkt2.data.an ^ @pkt1.data.qu
    AND pkt2.data.an.ttl > 0x00000a}
MSG: TTL in a legacy unicast response SHOULD be greater than ten seconds.

#Rule13:
#  Multicast query message has to include more than one question.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.questions = 0x00} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu}
MSG: Multicast responder MUST NOT response to malformed query.

# Rule14:
#  Multicast response message has to include more than one answer
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.answerRRs = 0x00}
MSG: Multicast response message MUST include more than one answer.
```

```
# Rule15:
#  Multicast query message has to include more than one question  and data field
# has to contain exact number of questions
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.questions != 0x00 AND pkt1.questions != pkt1.data.qu};
     pkt2 {pkt2.flag.qr = 0x80 AND pkt2.data.an ^ @pkt1.data.qu}
MSG: MDNS responder received a malformed packet that has inconsistence questions field,
    but it replied to the question.

# Rule16:
# Multicast response message has to include more than one answer and data field
# has to contain exact number of answers
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.answerRRs != 0x00 AND pkt2.answerRRs !=
    pkt2.data.an}
MSG: The number of answers in data field MUST same to the answers in header field.

# Rule17:
# Multicast response message has to include more than one Authority RRs  and data
# field has to contain exact number of answers.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.authorRRs != 0x00 AND pkt2.authorRRs !=
    pkt2.data.ns}
MSG: The number of authority records in data field MUST same to the authority bytes in
    header field.

# Rule18:
# Any DNS query for a name ending with ".local." MUST be sent to the mDNS multicast
# address (224.0.0.251 or its IPv6 equivalent FF02::FB).
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff AND pkt1.data.qu.name ^ 'local'
     AND pkt1.destip != 0xe00000fb} ; pkt2 {pkt2.flag.qr = 0x80 AND pkt2.data.an ^ @pkt1.
    data.qu}
MSG: MDNS responder responded to a DNS query for a name ending with local not to be sent
    to the mDNS multicast address.

# Rule19:
# Multicast DNS Responses MUST NOT contain any questions in the Question Section.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ; pkt2 {pkt2.flag.qr = 0x80
    AND pkt2.destip = 0xe00000fb AND pkt2.data.an ^ @pkt1.data.qu
    AND pkt2.questions != 0x00}
MSG: Multicast DNS Responses MUST NOT contain any questions in the Questions field.

#Rule20:
# MDNS Responder MUST not response to a malformed query that contains inconsistence
# questions and questions's data
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.questions != 0x00 AND pkt1.questions != pkt1.data.qu};
pkt2 {pkt2.flag.qr = 0x80 AND pkt2.data.an ^ @pkt1.data.qu}
MSG: MDNS Responder MUST NOT response to a malformed query.
```

```
#Rule21:
# 250ms after the first probing query the host should send a second, then 250ms after
# that a third.
==
BOF ; pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type = 0x00ff} ;
pkt2 {pkt2.flag.qr = 0x00 AND pkt2.data.qu.type = 0x00ff AND pkt2.data.qu = @pkt1.data.qu
     AND pkt2.timestamp > @pkt1.timestamp + 0.5}
MSG: MDNS Responder sends successive proving packets not within 250ms.

#Rule22:
#  For probing query, 250ms after the first query the host should send a second, then
#  250ms after that a third.
==
BOF ; pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type = 0x00ff} ;
pkt2 {pkt2.flag.qr = 0x00 AND pkt2.data.qu = @pkt1.data.qu} ; pkt3 {pkt3.flag.qr = 0x80
    AND pkt3.data.an ^ @pkt1.data.qu}
MSG: MDNS Responder enters announcing packet too late

# Rule23:
# Because of the mDNS multicast rate limiting rules, the first two probes SHOULD be
# sent as "QU" questions with the "unicast response" bit set, to allow a defending
# host to respond immediately via unicast, instead of potentially having to wait before
# replying via multicast.
==
BOF ; pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type = 0x00ff
    AND pkt1.data.qu.class = 0x0001}
MSG: The first two probes SHOULD be sent as QU questions with unicast response.

# Rule24:
# Because of the mDNS multicast rate limiting rules, the first two probes SHOULD
# be sent as "QU" questions with the "unicast response" bit set, to allow a defending
# host to respond immediately via unicast, instead of potentially having to wait before
# replying via multicast.
==
BOF ; pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type = 0x00ff
    AND pkt1.data.qu.class = 0x8001} ; pkt2 {pkt2.flag.qr = 0x00
    AND pkt2.data.qu = @pkt1.data.qu AND pkt2.data.qu.class = 0x0001};
MSG: The first two probes SHOULD be sent as QU questions with unicast response.

# Rule25:
# The "cache flush" bit MUST NOT be set in any resource records in a response packet
# sent in legacy unicast responses to UDP ports other than 5353.
==
pkt1 {pkt1.flag.qr = 0x00 AND pkt1.data.qu.type != 0x00ff} ;
pkt2 {pkt2.destip != 0xe00000fb AND pkt2.dport != 0x14e9 AND pkt2.flag.qr = 0x80
    AND pkt2.data.an ^ @pkt1.data.qu AND pkt2.data.an.flushbit = 0x8000}
MSG: The "cache flush" bit MUST NOT be set in any resource records in a response packet
     sent in legacy unicast responses to UDP ports other than 5353.
```

B.2 Packet Rules for the DHCP Protocol

The following gives the full list of rules for the DHCP protocol.

```
# Rule 1:
# If 'giaddr' is 0x0 in the DHCPREQUEST message, the client is on the same subnet as
# the server.  The server MUST broadcast the DHCPNAK message to the 0xffffffff
# broadcast address because the client may not have a correct network address or
# subnet mask, and the client may not be answering ARP requests
==
pkt1 {pkt1.opts.53 = 0x04 AND pkt1.giaddr = 0x00000000} ; pkt2 {pkt2.xid = @pkt1.xid
    AND pkt2.opts.53 = 0x06  AND pkt2.destip != 0xffffffff}
MSG: DHCPNAK message MUST be broadcasted to the 0xffffffff broadcast address.

# Rule2:
# If 'giaddr' is not 0x0 in the DHCPREQUEST message, the server MUST send the DHCPNAK
# message  to the IP address of the BOOTP relay agent, as recorded in 'giaddr'.
==
pkt1 {pkt1.opts.53 = 0x04 AND pkt1.giaddr != 0x00000000} ; pkt2 {pkt2.xid = @pkt1.xid
    AND pkt2.opts.53 = 0x06 AND pkt2.destip != @pkt1.giaddr}
MSG: DHCPNAK message MUST be broadcasted to the giaddr in the DHCPREQUEST message.

# Rule3:
# OFFER - IP address lease time MUST be provided
==
pkt1 {pkt1.opts.53 = 0x01} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND pkt2.opts.51 = NULL}
MSG: DHCPOFFER message MUST provide IP address lease time


# Rule4:
# ACK - IP address lease time MUST (DHCPREQUEST) be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND pkt2.opts.51 = NULL}
MSG: DHCPACK message MUST provide IP address lease time


# Rule5:
#  OFFER - Server identifier  MUST be provided
==
pkt1 {pkt1.opts.53 = 0x01} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND pkt2.opts.54 = NULL}
MSG: DHCPOFFER message MUST provide sever identifier


# Rule6:
#  ACK - Server identifier MUST be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND pkt2.opts.54 = NULL}
MSG: DHCPACK message MUST provide sever identifier
```

```
# Rule7:
#  NAK - Server identifier MUST be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x06
    AND pkt2.opts.54 = NULL}
MSG: DHCPNAK message MUST provide sever identifier


# Rule8:
#  The server MUST return to the client about the client's network address
==
pkt1 {pkt1.opts.53 = 0x01} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND pkt2.yiaddr = 0x00000000}
MSG: The server MUST return the client's network address to the client


# Rule9:
#  OFFER - Requested IP address MUST NOT be included
==
pkt1 {pkt1.opts.53 = 0x01} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND pkt2.opts.50 != NULL }
MSG: OFFER message MUST NOT provide requested ip address


# Rule10:
#  ACK - Requested IP address MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND pkt2.opts.50 != NULL }
MSG: ACK message MUST NOT provide requested ip address


# Rule11:
#  NAK - Requested IP address  MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x06
    AND pkt2.opts.50 != NULL }
MSG: NACK message MUST NOT provide requested ip address


# Rule12:
#  ACK - IP address lease time  MUST NOT (DHCPINFORM) be provided
==
pkt1 {pkt1.opts.53 = 0x08} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND pkt2.destip = @pkt1.ciaddr AND pkt2.opts.51 != NULL }
MSG: ACK message for DHCPINFORM MUST NOT provide  IP address lease time


# Rule13:
#  NAK - IP address lease time  MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x06
    AND pkt2.opts.51 != NULL }
MSG: NACK message MUST NOT provide  IP address lease time
```

```
# Rule14:
#  NAK - 'file''sname' fields MUST NOT be used
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x06
    AND pkt2.sname != 0x00 }
MSG: NACK message MUST NOT use 'file''sname' fields


# Rule15:
#  OFFER - Parameter request list  MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x01} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND pkt2.opts.55 != NULL }
MSG: OFFER message MUST NOT provide Parameter request list


# Rule16:
#  ACK - Parameter request list MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND pkt2.opts.55 != NULL }
MSG: ACK message MUST NOT provide Parameter request list


#Rule17:
#  NACK - Parameter request list MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x06
    AND pkt2.opts.55 != NULL }
MSG: NACK message MUST NOT provide Parameter request list


# Rule18:
#  OFFER - Client identifier MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x01} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND pkt2.opts.61 != NULL }
MSG: OFFER message MUST NOT provide Client identifier


#Rule19:
#  ACK - Client identifier MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND pkt2.opts.61 != NULL }
MSG: ACK message MUST NOT provide Client identifier


# Rule20:
# OFFER - Maximum message size MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x01} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND pkt2.opts.57 != NULL }
MSG: OFFER message MUST NOT provide Maximum message size
```

```
#Rule21:
#  ACK - Maximum message size MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND pkt2.opts.57 != NULL }
MSG: ACK message MUST NOT provide Maximum message size


#Rule22:
#  NACK - Maximum message size MUST NOT be provided
==
pkt1 {pkt1.opts.53 = 0x03} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x06
    AND pkt2.opts.57 != NULL }
MSG: NACK message MUST NOT provide Maximum message size


#Rule23:
# If the 'giaddr' field is zero and the 'ciaddr' field is nonzero, then the server
    unicasts
# DHCPOFFER messages to the address in 'ciaddr'.
==
pkt1 {pkt1.opts.53 = 0x01 AND pkt1.giaddr = 0x00000000 AND pkt1.ciaddr != 0x00000000} ;
pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02 AND destip != @pkt1.ciaddr }
MSG: OFFER message must send to ciaddr in DISCOVER message from the client


# Rule24:
# If the 'giaddr' field is zero and the 'ciaddr' field is nonzero, then the server
# unicasts DHCPACK messages to the address in 'ciaddr'.
==
pkt1 {pkt1.opts.53 = 0x01 AND pkt1.giaddr = 0x00000000 AND pkt1.ciaddr != 0x00000000} ;
pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05 AND destip != @pkt1.ciaddr }
MSG: DHCPACK message must send to ciaddr in DISCOVER message from the client


# Rule25:
# If 'giaddr' is zero and 'ciaddr' is zero, and the broadcast bit is set, then the server
# broadcasts DHCPOFFER messages to 0xffffffff.
==
pkt1 {pkt1.opts.53 = 0x01 AND pkt1.giaddr = 0x00000000 AND pkt1.ciaddr = 0x00000000
    AND pkt1.flags.br = 0x80} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND destip != 0xffffffff }
MSG: DHCPOFFER message must send to the broadcast address 255.255.255.255


#Rule26:
# If 'giaddr' is zero and 'ciaddr' is zero, and the broadcast bit is set, then the server
# broadcasts DHCPACK messages to 0xffffffff.
==
pkt1 {pkt1.opts.53 = 0x01 AND pkt1.giaddr = 0x00000000 AND pkt1.ciaddr = 0x00000000
    AND pkt1.flags.br = 0x80} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND destip != 0xffffffff }
MSG: DHCPACK message must send to the broadcast address 255.255.255.255
```

```
# Rule27:
# If the broadcast bit is not set and 'giaddr' is zero and 'ciaddr' is zero, then the
# server unicasts DHCPOFFER messages to the client's hardware address and 'yiaddr'
    address.
==
pkt1 {pkt1.opts.53 = 0x01 AND pkt1.giaddr = 0x00000000 AND pkt1.ciaddr = 0x00000000
    AND pkt1.flags.br != 0x80} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x02
    AND destip = 0xffffffff }
MSG: DHCPOFFER message must not send to the broadcast address 255.255.255.255


# Rule28:
# If the broadcast bit is not set and 'giaddr' is zero and 'ciaddr' is zero, then the
# server unicasts DHCPOFFER messages to the client's hardware address and 'yiaddr'
# address.
==
pkt1 {pkt1.opts.53 = 0x01 AND pkt1.giaddr = 0x00000000 AND pkt1.ciaddr = 0x00000000
    AND pkt1.flags.br != 0x80} ; pkt2 {pkt2.xid = @pkt1.xid AND pkt2.opts.53 = 0x05
    AND destip = 0xffffffff }
MSG: DHCPACK message must not send to the broadcast address 255.255.255.255


# Rule29:
# In all cases, when 'giaddr' is zero, the server broadcasts any DHCPNAK messages
# to 0xffffffff.
==
pkt1 {pkt1.opts.53 = 0x01 AND pkt1.giaddr = 0x00000000} ; pkt2 {pkt2.xid = @pkt1.xid
    AND pkt2.opts.53 = 0x06 AND destip != 0xffffffff }
MSG: DHCPNAK message must send to the broadcast address 255.255.255.255
```