This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

SYMBEXNET: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications

JaeSeung Song, *Member, IEEE,* Cristian Cadar, *Member, IEEE,* Peter Pietzuch, *Member, IEEE*

Abstract—Implementations of network protocols, such as DNS, DHCP and Zeroconf, are prone to flaws, security vulnerabilities and interoperability issues caused by developer mistakes and ambiguous requirements in protocol specifications. Detecting such problems is not easy because (i) many bugs manifest themselves only after prolonged operation; (ii) reasoning about semantic errors requires a machine-readable specification; and (iii) the state space of complex protocol implementations is large.

This article presents a novel approach that combines symbolic execution and rule-based specifications to detect various types of flaws in network protocol implementations. The core idea behind our approach is to (1) automatically generate high-coverage test input packets for a network protocol implementation using *single- and multi-packet exchange symbolic execution* (targeting stateless and stateful protocols, respectively) and then (2) use these packets to detect potential violations of manual rules derived from the protocol specification, and check the interoperability of different implementations of the same network protocol. We present a system based on these techniques, SYMBEXNET, and evaluate it 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.

Index Terms—Symbolic Execution, Network Security, Testing, Interoperability Testing.

1 INTRODUCTION

IMPLEMENTATIONS of network protocols used today, such as DNS [36], Zeroconf [44] and DHCP [14] are often prone to errors. Ambiguities in network protocol specifications can cause different interpretations by developers, leading to bugs and interoperability problems in the corresponding implementations of network services. The complexity of network protocols makes errors difficult to detect, even for well-studied and mature protocols: errors may only manifest themselves after complex sequences of network packets [30]. For example, DNS server implementations that are vulnerable to cache poisoning attacks [12] only exhibit problems in specific scenarios. The impact of such vulnerabilities can be severe though, and the cost of fixing them can be high.

Although a large body of work has focused on finding software errors, existing techniques have significant weaknesses when applied to network protocol implementations because (i) many bugs manifest themselves only after prolonged operation in a production network; (ii) reasoning about semantic errors in network protocol implementations requires a machine-readable specification of the intended protocol behaviour; and (iii) the state space of complex network protocol implementations is large.

In this article, we describe SYMBEXNET, a new approach that combines *symbolic execution* [25]—a program analysis technique that can generate inputs that explore multiple paths in a program—with *rule-based specifications* to check automatically a network protocol implementation against its specification and discover various types of errors, which would be hard to detect manually.

SYMBEXNET takes as input the C source code of a network protocol implementation and a set of rules, which define correct and incorrect behaviour. Developers derive rules manually from the protocol specification and express them in a high-level *packet stream language*, which states invalid patterns in the sequence of packets exchanged between a client and a server. The language permits rules to refer to packet header fields and their relationship within a packet stream. Rules can be extracted easily from RFC network protocol specifications [4], thus encoding the externally-visible behaviour of a network protocol in terms of input and output packets.

Using symbolic execution, SYMBEXNET generates an exhaustive set of input packets that achieve a broad and deep exploration of the program state space. To scale up to large protocol implementations, SYMBEXNET mixes concrete and symbolic execution: for a broad exploration, it considers in turn all combinations of fields as symbolic, running each combination for a fixed amount of time. For a deep

- *J. Song is with the Department of Computer and Information Security, Sejong University, Seoul. E-mail: jssong@sejong.ac.kr*
- *C. Cadar and P. Pietzuch are with the Department of Computing, Imperial College London. Email: {c.cadar, prp}@imperial.ac.uk*

This article is partly based on a paper published in the International Conference on Computer Communication Networks (ICCCN 2011) [42]

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

exploration, in order to detect errors that require *complex packet exchanges* to reach a given network protocol state, SYMBEXNET repeatedly performs symbolic execution with additional packets sent in multiple rounds. SYMBEXNET then natively executes the implementation on all generated test packets and checks whether the implementation correctly handles them according to the specification rules.

In addition, SYMBEXNET can use the generated test input packets to check the *interoperability* of different implementations of the same network protocol. After a set of test input packets was generated from a network implementation, SYMBEXNET executes the packets against all available implementations and reports any differences as potential errors.

We empirically evaluate a prototype implementation of SYMBEXNET with multiple network protocol implementations. We are able to generate high-coverage test input packets and detect low-level errors leading to crashes. We 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. Our experiments also reveal that multiple implementations of the same protocol can behave differently resulting in interoperability problems.

In summary, the main contributions of this article are as follows:

1) An approach that uses symbolic execution combined with rule-based network protocol specifications to automatically generate high-coverage test packets and find semantic errors in network protocol implementations. Our approach enhances symbolic execution to allow it to explore network protocol implementations efficiently: by mixing concrete and symbolic execution and generating multiple rounds of test input packets, it is possible to achieve both a broad and a deep exploration of the state space, resulting in high source code coverage of the network protocol implementations.
2) Our experience implementing this approach, together with an experimental evaluation on several real-world network protocol implementations—namely, five network daemons implementing the Zeroconf and the DHCP specifications—where it found 39 unique generic, semantic and interoperability errors (22 for Zeroconf and 17 for DHCP).

The next section provides background information. §3 gives an overview of the SYMBEXNET design, explaining how it explores the state space of network protocol implementations and how it generates high-coverage test input packets using symbolic execution. §4 describes the rules that can be derived from network protocol specifications; §5 examines interoperability testing of different protocol implementations; and §6 presents our experimental results and the errors discovered. The article finishes with a discussion of related work (§7) and conclusions (§8).

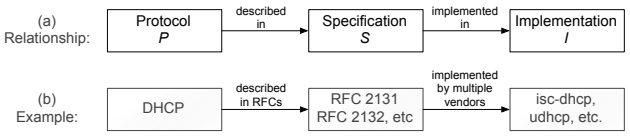


Fig. 1. Relationship between protocol, specification and implementation

2 BACKGROUND

We start by giving some background about the operation of network protocols (§2.1) and the basics of symbolic execution for generating test cases (§2.2).

2.1 Network Protocols

A network can be defined as a collection of entities interconnected by communication technologies that enable the exchange of information [45]. The communicating entities require an agreement to exchange information and such agreements are called *network protocols*. The messages exchanged by these entities are called *packets*, and a sequence of packets is referred to as a *a packet stream*.

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*. In UNIX and other operating systems, implementations of network protocols are called *network daemons*.

Figure 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 which is described in several *Request For Comments (RFC)* documents that form the protocol specification [1], [14]. Several implementations of the specification exist, such as `isc-dhcp` [20] and `udhcp` [13].

We give two examples of network protocols: *Zeroconf* and *DHCP*. Both are widely used and implemented by different vendors. They are used throughout the paper to demonstrate the various problems addressed by our approach.

Zeroconf. Zero-configuration networking [10] 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. It

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

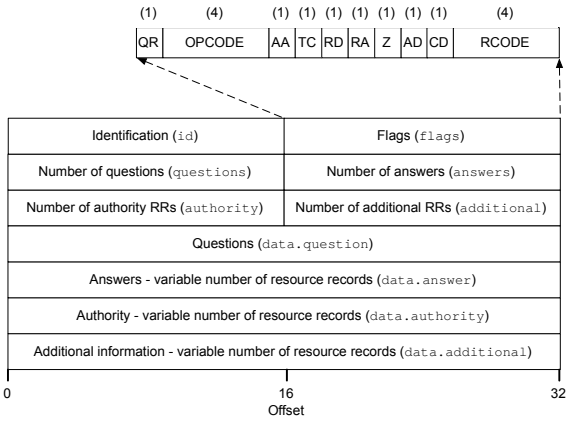


Fig. 2. Packet format for DNS/Zeroconf

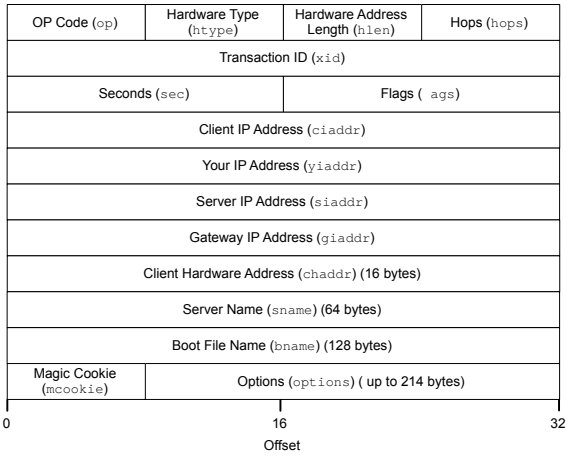


Fig. 3. Packet format for DHCP

is used widely as part of applications on the iOS and Android platforms.

Figure 2 shows the packet format for DNS/Zero-Conf. 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. DNS packets also include fields with control flags.

The Zeroconf protocol is defined as part of two RFC specifications: multicast DNS (mDNS) [10] and DNS-based Service Discovery (DNS-SD) [11]. 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.

In Zeroconf, a new network service, such as a file server or printer, is added as follows. First, 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 broadcasting DNS packet announcing the new service at least twice.

DHCP. The Dynamic Host Configuration Protocol [14] is a standard network protocol 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 on the values of certain fields in the packets.

DHCP is standardised in the RFC 2131 Dynamic Host Configuration Protocol [14]. The DHCP RFC describes the behaviour of a dynamic configuration service framework that passes configuration information to hosts on a TCP/IP network.

Figure 3 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 length. After that, the format has various fields for IP addresses that are needed to provide an available IP address to clients.

When a DHCP-enabled client is connected to the network, the client sends a broadcast query packet (`DHCPDISCOVER`) requesting an IP address from a DHCP server. Any DHCP server that receives the query may send a packet (`DHCPOFFER`) offering an available IP address. The client responds to the packet by sending a broadcast response packet (`DHCPREQUEST`) 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 (`DHCPRELEASE`). The server then returns the client's IP address to the available address pool.

For both protocols, if any step does not complete successfully, there are recovery actions specified by each protocol.

2.2 Symbolic Execution

The core idea behind symbolic execution [7]–[9], [17], [18], [25] is to use *symbolic values* as input, instead of actual data, and to represent values of program variables as symbolic expressions. As the program is executed, any statements that depend on the symbolic inputs are added as symbolic constraints. When execution reaches a branch that depends on the symbolic input, both potential paths are followed separately, adding the constraint that the branch condition is true or false, respectively.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

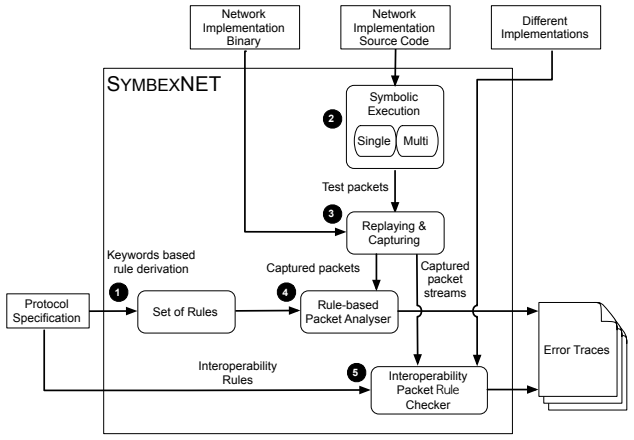


Fig. 4. SYMBEXNET system design

Two pieces of information are associated with each explored path: 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 following that path. When the program terminates or encounters an error, the current *PC* can be solved for concrete values, which form a test case that follows that exact path.

3 SYMBEXNET DESIGN

We propose a new approach for checking network protocol implementations using symbolic execution and rule-based specifications. The main idea is to generate a set of test input packets using symbolic execution to achieve high code coverage and replay them against an implementation, observing potential violations of rules derived from the protocol specification. To overcome the challenge that network protocol implementations often require complex packet exchanges in order to exhibit particular behaviour that should be checked, we devise two exploration methods, *single-* and *multi-packet exchange symbolic execution*, that achieve broad and deep exploration of the state space of a target implementation.

3.1 SYMBEXNET Overview

Our goal is to determine the compliance of a network protocol implementation with its protocol specification and the interoperability with other implementations of the same protocol. Our approach is simple to use, yet rigorous enough to discover non-trivial bugs, providing an automated method to validate protocol implementations.

The SYMBEXNET design is shown in Figure 4. When testing a network protocol implementation with SYMBEXNET, there are five steps, as labeled in the figure:

(1) **Creation of packet rules** (§4.1). The first step is to develop a rule-based specification from a protocol specification. The requirements describing behavioural properties of the protocol are extracted manually 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** (§3.3 and §3.4). To validate as many packet rules as possible, SYMBEXNET generates 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 explore the state space broadly, SYMBEXNET repeatedly marks parts of a packet as symbolic. For deep exploration, it repeatedly performs symbolic execution on selected input packets to generate sequences of test input packets.

(3) **Replay of test packets** (§3.3 and 3.4). The generated test packets are replayed on the original network implementation. Each test packet is sent to the implementation in a controlled network environment, and the output packets generated by the implementation are recorded, together with the input packets, as a packet stream.

(4) **Validation of packet rules** (§4.3). The captured input and output packets from the previous step are validated against the rule-based specification. SYMBEXNET translates the specification rules into a set of non-deterministic finite automata (NFAs). Through analysing all captured replay packets against each NFA, SYMBEXNET detects rule violations.

(5) **Checking for interoperability** (§5). To check if multiple implementations of the same network protocol are interoperable, SYMBEXNET applies steps (2) and (3) to each implementation. The generated packet streams obtained from each implementation are replayed on all implementations. To check for interoperability, SYMBEXNET uses specific interoperability rules to compare the packets streams for discrepancies, while ignoring differences that are not semantically meaningful, such as redundant packets or different packet orderings.

3.2 Symbolic Execution and Exploration

SYMBEXNET executes the network protocol implementation symbolically using the KLEE [7] symbolic execution engine, available at http://klee.llvm.org.

Symbolic data. The first decision regards the granularity at which data is treated as symbolic. Operating at the level of entire input packets may seem the natural choice, but this creates a huge state space,

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

which often causes symbolic execution to get stuck in parsing code, generating mostly invalid packets that are discarded early by an implementation. Instead, we make use of the knowledge available from the protocol specification and operate at the level of packet fields. We consider in turn all combinations of fields as symbolic, running each combination for a fixed amount of time.

Symbolic packet injection. To perform symbolic execution, symbolic packets must be "injected" into the network protocol implementation. An important issue is to decide how to inject symbolic packets without requiring major changes to the implementation. When a running implementation receives a certain real input packet from our purposely-constructed test client (§6.3), the packet is intercepted by SYMBEXNET, which marks all or parts of it as symbolic and starts the symbolic execution process using the symbolic execution engine.

Once symbolic packets are injected, symbolic execution explores as many paths as possible within a given time budget. To maximise the chance of finding errors, our goal is to achieve high coverage of the network implementation. Below, we discuss two exploration strategies for network protocol implementations that generate high-coverage test input packets.

```
1  enum state {ST_1, ST_2, ST_3, ST_END};
2  boolean first_flag = False;
3
4  while (current_state != ST_END)
5  {
6    input = get_input();
7
8    switch (input[0])
9    {
10     case ST_1:
11 /* Do something with input and set first_flag */
12       handle_first_pkt(&input);
13       first_flag = True;
14       break;
15
16     case ST_2:
17       if (input[1]==0)
18         handle_query(&input);
19       else if ((input[1]==1) && first_flag) {
20 /* The following code is not explored by SPE-SE*/
21       handle_requested(&input);
22       abort(); /* error */
23       }
24       break;
25
26     case ST_3:
27 /* Do something with input and set current_state */
28       handle_final_packet(&input);
29       break;
30
31 /* ... etc ... */
32   }
33 }
```

Fig. 5. C program for a state machine in a typical network protocol implementation

3.3 Single-Packet Exchange Symbolic Execution

In *single-packet exchange symbolic execution* (SPE-SE), SYMBEXNET performs symbolic execution on a single symbolic input packet. SPE-SE is well suited for stateless network protocols that treat each request independently. The checking process of SYMBEXNET using SPE-SE is composed of two tasks, *test packet generation* and *test packet replay*:

Test packet generation. To execute a network daemon symbolically, we first compile its source code to LLVM bitcode [27], the low-level language used by the KLEE symbolic execution engine. When the LLVM-compiled daemon starts, it behaves normally and waits for input. SYMBEXNET can then send a specific test input packet to the daemon in order to trigger symbolic execution. When the daemon receives this test packet, SYMBEXNET intercepts the packet and marks specified fields as symbolic.

For example, if the user provides an instruction to mark the `flags` field as symbolic, SYMBEXNET replaces the concrete value of this field within the packet with symbolic values while keeping the other fields concrete. It then uses the symbolic execution engine to explore possible execution paths corresponding to the various input packets having different `flags` values. At the end of each execution path, it stores the concrete test packet for a given path on disk.

Test packet replay. Generated test packets are then executed ("replayed") using the native version of the implementation. This replay process is used to check the behaviour of the implementation against the rule-based specification and to confirm any previously-encountered generic errors. SYMBEXNET executes the 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. When the implementation executes in the configured environment, SYMBEXNET selects a test input packet and then 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.

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 an input to the next step, rule validation (§4).

3.4 Multi-Packet Exchange Symbolic Execution

SPE-SE is not suitable for checking stateful network protocol implementations because the generated test packets cannot explore code execution paths that are only reachable after receiving more than one input packet. In order to overcome this limitation, we propose *multi-packet exchange symbolic execution* (MPE-SE).

Motivating example. The code in Figure 5 shows a simple finite state machine (FSM) implementation, as used in many network protocol implementations,

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

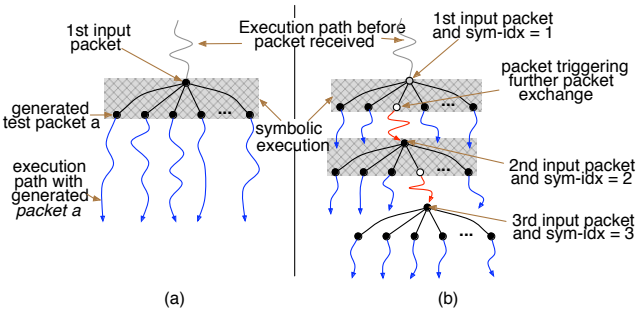


Fig. 6. Program exploration for (a) SPE-SE and (b) MPE-SE

such as DHCP, to handle incoming packets based on previously received packets. When such code is run by SPE-SE with a single symbolic packet, symbolic execution cannot explore the `then` side of the `if` statement on line 19 because the branch condition can become `true` only after processing the first packet. In general, many network protocol implementations make decisions based not only on information contained in the current packet but also in previously received packets.

The obvious solution to this problem is to consider multiple packets as symbolic input. However, this approach is not scalable because it typically increases the program search space and the complexity of generated constraints exponentially with the input size.

To alleviate this problem, MPE-SE uses a combination of concrete and symbolic execution: it uses the currently generated sequences of packets as input for the next round of symbolic execution, which enables it to explore deep states of the implementation more effectively.

When processing packet k, MPE-SE alternates between concrete and symbolic execution as follows:

(1) **Concrete execution.** MPE-SE starts by replaying, in turn, each sequence of $k-1$ concrete packets generated before.
(2) **Symbolic execution.** On each of the paths explored in step (1), if the implementation waits for a k-th packet, MPE-SE marks that packet as symbolic and executes the implementation symbolically. Otherwise, the path is terminated.
(3) **Update sequence tree.** The packets generated at the end of the previous step are inserted into a sequence tree, to be used in step (1) of the next iteration.
(4) **Terminate.** MPE-SE finishes after a predefined number of rounds or when it cannot find a packet sequence that increases source code coverage further.

Figure 6 shows graphically how SPE-SE and MPE-SE compare to each other.

4 Rule-based Specifications

Symbolic execution can automatically detect low-level generic errors, such as buffer overflows or division-by-zero errors. To discover semantic errors, we augment SymbexNet with an expressive rule-based language, which can be used to create network protocol specifications. Therefore, an important step in using SymbexNet is to translate a standard protocol specification, such as an RFC document, into a rule-based specification.

We define behavioural violations using a rule-based language that matches incorrect sequences of packets. We assume that the behaviour of an implementation consists of the output packets that it emits in response to input packets. In this black-box approach, we do not reason about the internal state of the implementation, which means that some parts of the specification cannot be encoded, but has the advantage that 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.

We first show how rules are derived from specifications (§4.1) and then introduce our rule-based packet stream language (§4.2). We finally show how rules are implemented and validated (§4.3).

4.1 Rule Extraction

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 [5]. For example, "MUST" (similarly to "REQUIRED" or "SHALL") means that the statement is an absolute requirement. We find that phrases containing these words are good candidates for translation into formal rules.

Consider how sentences containing such keywords can be used to form 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 mDNS network protocol [10]:

> *"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."*

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 rule.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

Usually requirements to be included in protocol specifications address how to communicate with external network entities and how to manage internal states such as cached data, network parameters and protocol-specific data. Our rules refer to externally observable aspects of packets, thereby can be reused across different implementations of the same protocol. But this means that not all phrases from specifications can be translated 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 [...]"*

4.2 Rule-based Packet Stream Language

Since our rule-based packet stream language is intended for use by developers of network protocol implementations, it is designed based on two requirements: *expressiveness* and *ease of integration* with network protocols. In contrast to existing pattern matching languages [39], it contains domain-specific constructs to refer to packet header fields for specific network protocols. It also includes a set of operators and modifiers to express protocol-specific features such as ignoring packets that do not satisfy a given filter condition.

The language uses packet expressions of the following form:

$$packet\_expression = pkt\{\Sigma_{filters}\}$$

where pkt is a name given to the packet to be matched, 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 introduced some of the fields that are part of DNS and DHCP in Figures 2 and 3, respectively. The set of packet filter predicates are sequences of valid packet filters joined by the logical operators AND/OR. If multiple fields with the same name exist, the modifiers ANY and ALL specify that a predicate has to match at least one or all fields, respectively. Nested field names are separated by dots.

Rule operators. To describe a sequence of exchanged packets, packet expressions are composed using operators. Specificially, rule expressions can be built recursively using three operators: *next* (;), *union* (|) and *iteration* (+):

1) The **next operator** `p1;p2` detects the next occurrence of packet `p2` after `p1`, ignoring any intermediate packets that do not satisfy the filter predicates for `p2`.
2) The **union operator** `p1|p2` matches a choice of packets `p1` or `p2`.
3) The **iteration operator** `p` $+n$ detects n consecutive packets `p`.

Variable binding. 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 following rule 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 the packet filter predicates associated with the `resp` packet.

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` detects a packet response to the source IP address of the `query` packet while `query.id` discovers a packet that does not use the same ID as the `query` packet.

Timeouts. It is important to reason about time when describing packet sequences because many aspects of network protocols are driven by timers. To describe timing-related requirements, each packet contains a virtual field called `ts` that represents the timestamp at which the packet was received. For example,

```
ts >= @query.ts + 150
```

means that a rule matches a response packet with a timestamp that is 150 ms larger than that of the corresponding query packet.

4.3 Rule Implementation and Validation

The rules derived from a protocol specification are validated using *non-deterministic finite automata* (NFAs). We use an approach that is similar to ones found in event processing systems [39]. NFAs provide a mechanism for detecting complex event matches through the use of a high-level event pattern language.

An NFA for a rule from our packet stream language 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. Suppose that an automaton instance is in state S with assigned packet p. Each edge, between states S and T, is labelled with a

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

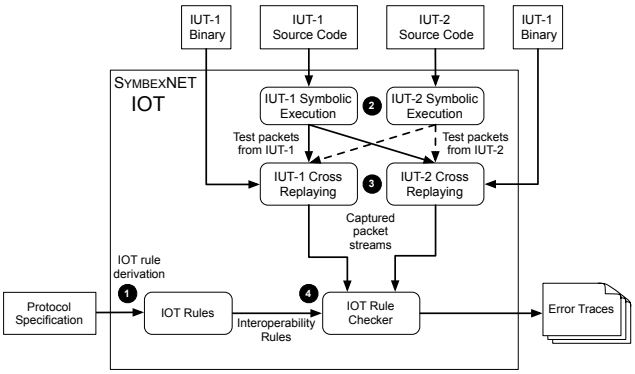


Fig. 7. Overview of checking interoperability for implementations under test IUT-1 and IUT-2 of the same network protocol using SYMBEXNET

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 satisfy 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. The rule has matched successfully after the NFA has reached an accepting state.

5 INTEROPERABILITY TESTING

This section introduces an interoperability testing (IOT) methodology for network protocol implementations. Given two implementations of the same network protocol, IOT performs four steps, as labeled in Figure 7:

(1) **Creation of IOT rules.** The first step is to derive IOT rules for interoperability testing from a protocol specification, which compare response packets from different implementations.
(2) **Generation of test packets.** To check the interoperability between two implementations, IOT relies on a set of high-coverage test input packets that can detect interoperability problems. These packets are generated by applying SPE-SE and MPE-SE to each implementation.
(3) **Cross-replay of test packets.** The test packets generated for each implementation are replayed on both implementations and all exchanged input and output packets are recorded.
(4) **Interoperability checking.** The recorded input and output packets are compared using the IOT rules from step (1). For each divergent behaviour, IOT reports a potential interoperability error.

5.1 Rule-based Interoperability Checking

To perform interoperability checking, we extend the rule-based language with the ability to compare packets from two different packet streams. Rules can include a *stream identifier* that refers to a specific packet stream among multiple 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 `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.

Packet field comparison. Developers can build a set of IOT rules that compare the value of each field in response packets and detect discrepancies. Such rules, however, may not discover interoperability problems robustly because some packet fields are permitted to have any value within an acceptable range. For example, the DHCP specification permits the lease duration to vary between zero and infinity, and administrators select a given value based on their policy. Therefore IOT rules permit to ignore such differences.

Interoperability decision criteria. For a test input packet p sent to both implementations *IUT-A* and *IUT-B*, IOT reports *PASS* as a result if both *IUT-A* and *IUT-B* generate no response packet or equivalent response packets, and *FAIL* otherwise. Equivalence between response packets is determined using the IOT rules created in step (1).

6 EVALUATION

The goal of the evaluation is to demonstrate the suitability of SYMBEXNET as an efficient tool for finding implementation flaws in real-world network protocol implementations. We apply SYMBEXNET to five network protocol implementations and show that it generates high quality sequences of test packets to check the correctness of network protocol implementations, as well as their interoperability with other implementations.

This section is organised as follows. §6.1 describes the objectives and methodology for evaluating SYMBEXNET, and §6.2 discusses how we derived rules from protocol specifications. The environmental setup used for the experiments is described in §6.3. The experimental results on single-packet exchange symbolic execution (SPE-SE), multi-packet exchange symbolic execution (MPE-SE) and interoperability testing (IOT) are presented in §6.4, §6.5 and §6.6, respectively. Finally we discuss the detected violations in §6.7.

6.1 Objectives and Methodology

We apply SYMBEXNET to network daemons implementing the Zeroconf [10], [11] and the DHCP [14] specifications. A 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

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

TABLE 1
Network daemons tested using SYMBEXNET

Protocol	Daemon	Version	Language	# LOC	# Files
Zeroconf	Avahi	0.6.23	C	7,000	31
	Bonjour	107.6	C	7,900	10
	JmDNS	3.4.1	Java	2,000	9
DHCP	isc-dhcp	2.0	C	3,000	15
	udhcp	0.9.9-pre	C	1,200	12

TABLE 2
Rules derived from specifications

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

[1] This number is smaller than the total number of translated rules, as some rules are combined together.
[2] These are phrases signifying absolute requirements but which don't use any of the keywords above.

the source code coverage achieved by the generated packets. The bug detection ability is evaluated by validating network protocol implementations against their protocol specifications and detecting interoperability flaws.

The evaluation addresses the following questions:

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

The five network protocol implementations tested are summarised in Table 1. We investigate three different implementations of Zeroconf: `Avahi` 0.6.23[1], Apple's `Bonjour` 107.6[2] and `JmDNS` 3.4.1.[3] `Avahi` has about 7,000 lines of code in 31 C files, and `Bonjour` has about 8,000 lines of source code in 10 C files for its Linux version. `JmDNS` is a Java implementation of Zeroconf, and currently the only available Zeroconf server that can be used on the Android platform. Since `JmDNS` is written in Java, we cannot run it using SYMBEXNET but we use it for interoperability checking.

For DHCP, we use two different implementations: `udhcp` 0.9.9-pre[4] and ISC's `isc-dhcp` 2.0.[5] Both `udhcp` and `isc-dhcp` are open-source DHCP implementations, and their source code has been thoroughly tested. `udhcp` has about 1,200 lines of code in 12 C files, and `isc-dhcp` has about 3,000 lines of code in 15 C files.

6.2 Rule Derivation

We manually derived a set of rules from the specifications of the Zeroconf and DHCP protocols by following the process described in §4.1. Our rule derivation for Zeroconf and DHCP resulted in a total of 25 and 29 rules, respectively. In Table 2, we show the result of the rule derivation from the specifications of both protocols. After becoming familiar with the process of developing rule-based 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. As explained in §2, 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, we examined both specifications to find phrases that contain the keywords from §4.1. As shown in Table 2, we found 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 translated 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 obtained a specification with 25 rules based on 33 valid phrases.

DHCP. After following the same rule derivation procedure, we found 118 phrases in total (72 with `MUST` and 46 with `MUST NOT`), from which we created 23 rules. Most untranslated phrases are related to requirements describing the behaviour of DHCP clients. Since we do not check clients but server implementations, any statements that are not related to the DHCP daemon are ignored. While analysing the specification, we found an additional seven phrases stating absolute requirements without the above keywords. Since these phrases specify how the DHCP daemon must construct response packets, we derived rules from these phrases too.

6.3 Experimental Set-up

The general experimental set-up involves two nodes: a network daemon to be tested and a test client. We

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

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

extend simple clients written in C (for Zeroconf) and in Python (for DHCP) with the ability to send regular mDNS/DHCP packets, as well as manually crafted packets that instruct SYMBEXNET to mark some fields as symbolic. To emulate a typical environment, the client registers six services with the daemon.

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, which ensures that they receive only packets from an isolated network.

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* [28], a portable packet capture library.

We conduct all experiments on a 2.4 Ghz Intel Core 2 Duo machine with 2 GB of RAM running 32-bit Ubuntu Linux.

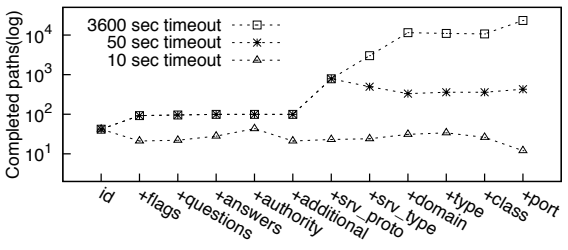
6.4 Single-Packet Exchange Symbolic Execution

Next we describe the experiments that check stateless network protocol implementations using SPE-SE.

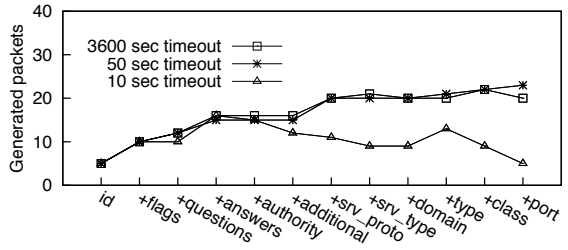
Test packet generation. As discussed in §3.2, an open challenge is to decide how many fields to treat as symbolic. Choosing too few fields may explore too little of the search space, but choosing too many (e.g. all) may result in too many paths, most of which would refer to invalid packets that are discarded early by an implementation. As a result, we treat as symbolic, in turn, all combinations of fields, and for each combination, run symbolic execution with a fixed timeout. To determine an appropriate timeout value for each protocol, we first try a subset of all possible combinations. In particular, we start by making only the first field of a packet symbolic, and then progressively also make subsequent fields symbolic, until all fields are considered.

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, until all 12 fields in the DNS packet are symbolic. 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 source code.

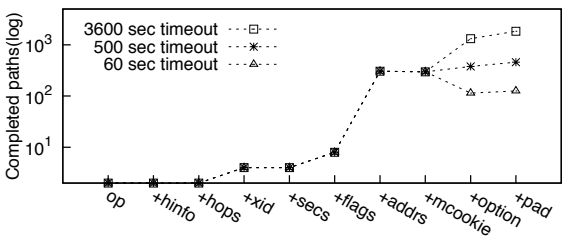
For each combination, we explore different timeout values for the symbolic execution. Figure 8 shows, for `Bonjour`, (a) the number of explored paths and (b) the number of generated test packets as we increase the number of symbolic packet fields and use different timeout values. These results show that a 50 s timeout value offers a good trade-off between running time and the number of generated test packets. With



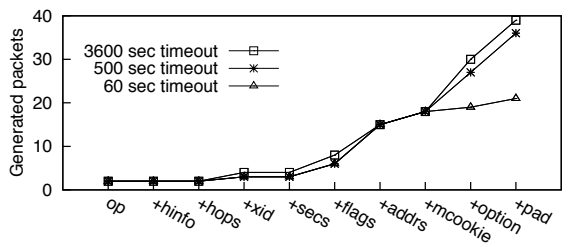
(a) Completed paths (bonjour)



(b) Generated packets (bonjour)



(c) Completed paths (udhcp)



(d) Generated packets (udhcp)

Fig. 8. Number of completed paths and generated test packets with various symbolic execution timeout values for `Bonjour` and `udhcp` with different numbers of symbolic fields

a 10 s timeout, SYMBEXNET generates significantly fewer test packets. Increasing the timeout to one hour, however, does not significantly increase the number of generated packets (i.e. SYMBEXNET generates many more paths but these do not cover additional lines of code). Therefore we use a 50 s timeout value in all of our subsequent `Bonjour` and `Avahi` experiments.

We run similar experiments for the `udhcp` daemon, and present our results in Figures 8(c) and 8(d). (Note that in some cases, the non-determinism in KLEE leads to the generation of slightly fewer packets for a longer timeout value.) Here we find that a 500 s

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

TABLE 3
Line coverage for the daemons using randomly-generated packets and SPE-SE

	Daemon	# files	Total LOC	Random cov.	SPE-SE	
					# pkts	Cov.
1	Avahi	31	7K	63%	34,047	75%
2	Bonjour	10	7.9K	48%	32,069	61.5%
3	isc-dhcp	15	3K	36%	16,777	67%
4	udhcp	12	1.2K	60%	14,271	79%

TABLE 4
Line coverage for the DHCP daemons using MPE-SE

Daemon	Conform. test	1st MPE-SE	2nd MPE-SE	3rd MPE-SE	SPE-SE	Combined
udhcp	67%	72%	76%	76%	76%	79%
isc-dhcp	60%	69%	71%	71%	70%	73%

timeout value offers a good trade-off between the time needed and the number of generated packets, which we therefore use in subsequent experiments.

Next we try all 4095 possible combinations of symbolic fields for multiple implementations of Zeroconf, with a 50 s timeout. Comparing the number of generated test packets for different packet field combinations helps us understand how each packet field is handled by the implementation. For Bonjour, SYMBEXNET generates 32,069 test packets with a total execution time of around 22 hours, while for Avahi 34,047 test packets in around 31 hours.

For DHCP, we similarly try all 1023 possible combinations, with a 500 s timeout. 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 s timeout value.

For both Zeroconf and DHCP, the two daemons generate similar numbers of test packets. This suggests that the daemons of the same protocol are, as expected, not that different in the way that they handle input packets.

Line coverage results. We use line coverage to measure the quality of test packets generated by symbolic execution. Coverage is measured by replaying the generated packets under the gcov tool, which is part of the GNU GCC compiler suite [29]. We exclude library files from the coverage measurements.

Table 3 shows the line coverage results for the four daemons. As a baseline, we also compute for each daemon the coverage of n packets randomly generated using the *Distributed Internet Traffic Generator* (D-ITG) [3], where n is set to the same number of packets as generated by SYMBEXNET (i.e. 32,069 for Bonjour, 34,047 for Avahi, 16,777 for udhcp and 14,271 for isc-dhcp, respectively). The total execution time is 12.3 hours for Bonjour, 13.1 hours for Avahi, 4.6 hours for udhcp and 3.8 hours for isc-dhcp, respectively.

We perform 10 such experimental runs for each daemon, and calculate the average coverage value. Since we perform the random testing with a large number of packets, the average coverage value does not change across runs. Furthermore, the same coverage is achieved even when we compute the cumulative coverage for all 10 experiments (i.e. for a total of $10n$ random packets).

For Bonjour, the randomly-generated packets cover 48% of the code, while SymbexNet covers 61.5%. Note that, fundamentally, 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% implements other features, such as cache maintenance and name conflict resolution.

For Avahi, the coverage difference is similar: the randomly-generated tests achieve 63% coverage, while SYMBEXNET achieves 75%. Around 22% of the code cannot be covered in our test scenario.

For DHCP, SYMBEXNET generates test packets that cover 67% and 79% for isc-dhcp and udhcp, respectively, while the randomly-generated packets only cover 36% and 60%, respectively. About 31% (isc-dhcp) and 16% (udhcp) of the source code cannot be covered in our testing scenario because it relates to BOOTP packet handling, static IP address allocation and unsupported server configurations. Since isc-dhcp is larger, containing additional features such as DNS lookup, SYMBEXNET achieves lower source code coverage than for udhcp.

6.5 Multi-Packet Exchange Symbolic Execution

In this section, we evaluate the effectiveness of multi-packet exchange symbolic execution (MPE-SE). Since MPE-SE is targeted towards stateful network protocols, we focus our evaluation on DHCP.

Test sequence generation. As described in §2, the state machine of DHCP is built around the life cycle of a dynamically assigned IP address between a DHCP client and a daemon, and involves three input packets received by the DHCP daemon from the client.

Consequently, we run both udhcp and isc-dhcp with three symbolic DHCP input packets of size 548 bytes, the maximum length of a DHCP packet. We use 120 mins (udhcp) and 60 mins (isc-dhcp) as timeout values for each MPE-SE round, respectively, in order to generate input sequences within one day. With these timeout values, udhcp and isc-dhcp generate a total of 286 and 595 unique test sequences, respectively.

Line coverage results. As a baseline, we measure the coverage achieved by a DHCP conformance test suite that checks the functional correctness of the DHCP

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

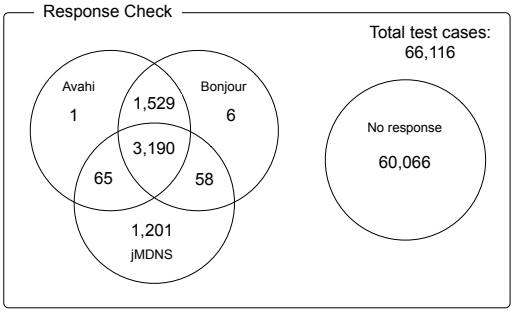
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

daemon [14], [48]. The test suite is rather minimal, but it checks that the daemon behaves correctly in the standard IP assignment scenario discussed in the DHCP specification.

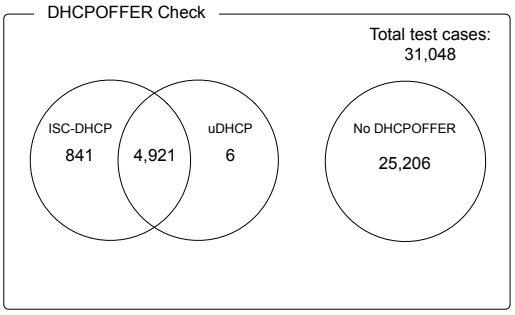
To compare MPE-SE with SPE-SE, we also generate test packets with an extended timeout value in the first round and measure the coverage. We choose a timeout value of 16.9 hours—the same time used to perform MPE-SE on three symbolic packets.

Table 4 shows the coverage results for the conformance test, each MPE-SE round, and SPE-SE. The test packet sequences generated after the second MPE-SE round achieve higher line coverage than the sequences generated after the first MPE-SE round. Since the daemons in our experiments release the session after they receive the third `DHCPRELEASE` packet, coverage between the second and third round of MPE-SE does not increase.

The last column shows the combined coverage of all SYMBEXNET experiments, which is greater than that of any individual experiment. This means that our two symbolic execution methods, MPE-SE and SPE-SE with longer execution times, can be effectively combined to achieve higher overall coverage.



(a) Zeroconf



(b) DHCP

Fig. 9. Venn diagrams with packet numbers when checking responses for Zeroconf and DHCP

6.6 Interoperability Testing

Next we explore the effectiveness of SYMBEXNET for finding interoperability issues between multiple implementations of the same protocol. For this, we use the previously generated test packets from two implementations of Zeroconf (`Avahi` and `Bonjour`) and DHCP (`udhcp` and `isc-dhcp`).

SYMBEXNET checks whether the daemons generate consistent response packets for a given set of test input packet. Any deviations are reported as potential interoperability bugs. To eliminate false positives, a list of IOT rules are derived from the protocol specifications (see §5). We create 8 and 9 interoperability testing rules for Zeroconf and DHCP, respectively.

Figure 9(a) shows the results of interoperability checking for the three Zeroconf daemons. The daemons ignore 60,066 test packets (out of 66,116) because the packets are malformed. As all daemons exhibit the same behaviour, these cases do not incur interoperability problems. On the other hand, there are several cases in which (1) only two of them respond (`Avahi`/`Bonjour`: 1,529 packets; `Avahi`/`JmDNS`: 65 packets; `Bonjour`/`JmDNS`: 58 packets); and (2) only one daemon responds (`Avahi`: 1 packet; `Bonjour`: 6 packets; `JmDNS`: 1,201 packets).

Figure 9(b) shows the response behaviour of the two DHCP daemons for all test input packets. Neither `isc-dhcp` nor `udhcp` generate response packets for 25,206 test packets (out of 31,048). There are 841 cases in which only `isc-dhcp` responds, and 6 cases in which only `udhcp` responds.

Some of these inconsistent cases are caused by recommended requirements and ranges of field values from the protocol specification. Using our packet rules for interoperability testing, we eliminate these inconsistencies and detect 25 genuine problems that indicate interoperability issues, as described below.

6.7 Discovered Implementation Errors

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). These bugs are discovered using generic error checks during symbolic execution (GB), rule-based analysis (SB) and interoperability testing (IB). Eleven of the detected bugs using packet rules are also detected by interoperability testing.

As shown in the last row of Table 5, SYMBEXNET detects 39 unique bugs in the five tested network daemons. Most of these bugs have been confirmed and fixed by the developers. More specifically, SYMBEXNET detects 4 bugs in `Avahi`, 4 bugs in `Bonjour`, 14 bugs in `JmDNS`, 13 bugs in `udhcp` and 4 bugs in `isc-dhcp`. The table provides a complete list of all detected bugs for each bug class with their descriptions. We discuss one error of each type below.

Violation 1 (Generic bug): Vulnerability caused by source port number zero. When SYMBEXNET marks the source port field as symbolic, it generates test packets with the following four values: 0, 2, 5351

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

TABLE 5
List of all detected generic bugs (GB), semantic bugs (SB) and interoperability bugs (IB)

	GB	SB	IB	Bug Description
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
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
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
23	√			Out of bound ptr error (`options.c` at line 79)
24	√			Out of bound ptr error (`options.c` at line 94)
25	√			Out of bound ptr error (`options.c` at line 99)
26	√			Out of bound ptr 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 ptr error (`dhcpd.c` at line 319)
30	√			Out of bound ptr error (`serverpacket.c` at line 113)
31	√			Out of bound ptr 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
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
39	13	15	25	There are 14 shared bugs

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 to reply with a response packet to all port numbers without errors. However, we detect abort errors in `Bonjour` and `Avahi`. Both errors are caused by the source port number of a query packet.

When a packet with source port 0 is received, the daemons abort due to an `assert` statement violation. Therefore sending a crafted packet to a multicast address (224.0.0.251) terminates all `Bonjour` daemons in the network that have an answer to the query. Such a 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 (version 320.5.1) does not exhibit the problem any more. The bug in `Avahi` was detected by the developers, and a patch was applied to version 0.6.28.

Violation 2 (Semantic bug): 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 all 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)"* [10]

From this 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. `JmDNS`, however, 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.

Violation 3 (Interoperability bug): Incorrect response to broadcast address. The DHCP specification states the following about responding to unicast addresses:

> *"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."* [14]

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, when the `isc-dhcp` daemon receives such a test packet, it responds incorrectly with a `DHCPOFFER` message to the broadcast address `255.255.255.255`. In contrast, the

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

`udhcp` daemon correctly sends a `DHCPOFFER` message to the client's hardware address and `yiaddr` address.

7 Related Work

To reason about the correctness of network protocols, prior work has employed a variety of program analysis techniques, such as model checking [15], [21], [35], [41], [43], static analysis [16], [46], [49], theorem proving [50], and refinement checking [2].

While some of these techniques can provide formal correctness guarantees, they typically require substantial manual effort (e.g. building an abstract model in model checking or guiding a theorem prover) or have false positives (e.g. due to imprecision in static analysis). In contrast, our packet matching rules are a lightweight approach for formally specifying network protocol behaviour. They permit symbolic execution to reason precisely about the actual implementations of a network protocol without false positives (but can only make guarantees about the paths that are explored).

Symbolic execution was used in the past to check network server implementations [8], but this considered only single input packets and focused on generic errors, ignoring protocol semantics. In the context of distributed protocols, symbolic execution was used to find semantic bugs via distributed assertions [38], which are extensions of standard C-like assertions that are added at the code level to check predicates on distributed node states. Instead our RFC-derived rules are designed to be independent of the implementation, operating at the level of input/output packet streams.

The idea of mixing concrete and symbolic execution was explored in prior work, e.g. by combining symbolic execution with random testing [32] or with well-formed inputs [18], including entire manual test suites [22], [34]. This is similar to our MPE-SE approach—it exploits the advantages of mixing concrete and symbolic execution, targeting stateful network protocols.

Interoperability testing is an established technique in the context of network protocols [19], [23], [26], [31], [40], [47]. SymbexNet enhances interoperability testing by exploiting high-coverage test cases derived via symbolic execution and employing rules to remove false positives.

Rule-based analysis has seen adoption for the validation of network protocol implementations and the detection of intrusions and vulnerabilities [24], [37]. Tools such as Pistachio [46] 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 rule-based packet stream language. 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 including Pistachio.

Event processing systems can detect complex event patterns using pattern matching techniques, e.g. state automata [6] or event trees [33]. They use high-level query languages that 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 they provide sufficient expressiveness for detecting complex sequences, we also use NFAs to find violations in packet rules. Our rule-based packet stream language is similar to the one used by the NextCEP system [39] but is extended with primitives suitable for describing network packet exchange patterns.

8 Conclusions

We described SymbexNet, a practical approach for checking network protocol implementations. It uses packet rules derived from protocol specifications and input packets generated using symbolic execution to discover violations in real-world network daemon implementations as well as interoperability problems. To explore complex sequences of packet exchanges, SymbexNet uses an exploration technique (MPE-SE) that repeatedly performs symbolic execution on selected test packets. To check interoperability, it observes behavioural differences between implementations of the same network protocol.

We implemented SymbexNet and validated it on multiple implementations of two network protocols: *Zeroconf* and *DHCP*. SymbexNet successfully detected a total of 39 non-trivial errors in the evaluated implementations, most of which have been confirmed and fixed by developers.

Our experience with SymbexNet has yielded several insights. Many of the detected violations are caused by different interpretations of the same specification. Since ambiguities may lead to problems such as incorrect functionality, interoperability errors and security vulnerabilities, it is important to eliminate and detect them from specifications and implementations. By translating textual specifications such as RFCs into rule-based ones, 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. For example, we believe that a large part of current RFCs could be written in such a form, enabling the use of automated techniques such as SymbexNet.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

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 thus 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. When networks run legacy daemons without available source code, such a behavioural checking technique that does not depend on source code can be a practical solution.

Unlike stateless network protocol implementations, stateful implementations that exchange a series of packets to perform a task are more difficult to check. MPE-SE, our approach for performing symbolic execution repeatedly on selected symbolic inputs, allows stateful implementations to reach deep execution paths that can only be explored after exchanging a sequence of specifically ordered packets.

Acknowledgements. The work reported in this article has formed part of the Flexible Networks area of the Core 5 Research Programme of the Virtual Centre of Excellence in Mobile & Personal Communications, Mobile VCE (www.mobilevce.com), and has been jointly funded by Mobile VCEs industrial member companies and the UK Government, via the Engineering and Physical Sciences Research Council.

REFERENCES

[1] S. Alexander and R. Droms, "IETF RFC 2132: DHCP Options and BOOTP Vendor Extensions," Mar. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2132.txt

[2] R. Alur and B.-Y. Wang, "Verifying network protocol implementations by symbolic refinement checking," in *Proc. of the 13th International Conference on Computer Aided Verification (CAV 2001)*, 2001, pp. 169–181.

[3] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre, "D-ITG distributed internet traffic generator," in *Proc. of the 1st International Conference on Quantitative Evaluation (QEST 2004)*, 2004, pp. 316–317.

[4] S. Bradner, "IETF RFC 2026: The Internet Standards Process – Revision 3," Oct. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2026.txt

[5] ——, "IETF RFC 2119: Key Words for Use in RFCs to Indicate Requirement Levels," Mar. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2119.txt

[6] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: a high-performance event processing engine," in *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2007)*, 2007, pp. 1100–1102.

[7] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc.of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*, 2008, pp. 209–224.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. of the 13th ACM Conference on Computer and Communications Security*, 2006, pp. 322–335.

[9] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proc. of the 33rd International Conference on Software Engineering (ICSE 2011)*, 2011, pp. 1066–1071.

[10] S. Cheshire and M. Krochmal, "IETF Internet Draft: Multicast DNS," Mar. 2010. [Online]. Available: http://files.multicastdns.org/

[11] S. Cheshire, M. Krochmal, and Apple Inc., "IETF Internet Draft: DNS-Based Service Discovery," Mar. 2010. [Online]. Available: http://tools.ietf.org/html/draft-cheshire-dnsext-dns-sd-06.txt

[12] Dan Kaminsky, "Black ops 2008 - its the end of the cache as we know it, Black Hat USA," 2008. [Online]. Available: http://www.doxpara.com/DMKBO2K8.ppt

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

[14] R. Droms, "IETF RFC 2131: Dynamic Host Configuration Protocol," Mar. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2131.txt

[15] 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*, 2010.

[16] N. Feamster, "Practical verification techniques for wide-area routing," *SIGCOMM Computer Communication Review*, vol. 34, pp. 87–92, Jan. 2004.

[17] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proc. of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2005)*, 2005, pp. 213–223.

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Network and Distributed System Security Symposium (NDSS 2008)*, Feb. 2008.

[19] R. Hao, D. Lee, R. K. Sinha, and N. Griffeth, "Integrated system interoperability testing with applications to VoIP," *IEEE/ACM Trans. on Networking*, vol. 12, pp. 823–836, Oct. 2004.

[20] Internet Software Consortium, "ISC DHCP," http://www.isc.org/software/dhcp.

[21] S. Islam, M. Sqalli, and S. Khan, "Modeling and Formal Verification of DHCP Using SPIN," *International Journal of Computer Science and Applications (IJCSA)*, vol. 3, pp. 145–159, Jun. 2006.

[22] P. Joshi, K. Sen, and M. Shlimovich, "Predictive testing: amplifying the effectiveness of software testing," in *Proc. of the 6th Joint Meeting on the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2007)*, sep 2007.

[23] S. Kang, J. Shin, and M. Kim, "Interoperability test suite derivation for communication protocols," *Computer Networks*, vol. 32, pp. 347–364, Mar. 2000.

[24] G. Khanna, P. Varadharajan, and S. Bagchi, "Self Checking Network Protocols: A Monitor Based Approach," in *Proc. of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS 2004)*, 2004, pp. 18–30.

[25] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, pp. 385–394, Jul. 1976.

[26] O. Kon and R. Castanet, "Test generation for interworking systems," *Computer Communications*, vol. 23, no. 7, pp. 642–652, 2000.

[27] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of the International Symposium on Code Generation and Optimization (CGO 2004)*, Mar. 2004.

[28] Lawrence Berkeley National Labs, "libpcap," Jun. 1994. [Online]. Available: http://www.tcpdump.org/

[29] G. G. License, "Gcov: Gnu coverage tool," http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html.

[30] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3S: Debugging Deployed Distributed Systems," in *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2008)*, 2008.

[31] S. Maag and C. Grepet, "Interoperability testing of a MANET routing protocol using a node self-similarity approach," in *Proc. of the ACM Symposium on Applied Computing (SAC 2008)*, 2008, pp. 1908–1912.

[32] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proc. of the 29th International Conference on Software Engineering (ICSE 2007)*, 2007, pp. 416–426.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2323977, IEEE Transactions on Software Engineering

[33] M. Mansouri-Samani and M. Sloman, "GEM: A Generalized Event Monitoring Language for Distributed Systems," *Distributed Systems Engineering*, vol. 4, no. 2, pp. 96–108, 1997.

[34] P. D. Marinescu and C. Cadar, "make test-zesti: A Symbolic Execution Solution for Improving Regression Testing," in *Proc. of the 34th International Conference on Software Engineering (ICSE 2012)*, Jun. 2012, pp. 716–726.

[35] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proc. of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004, pp. 155–168.

[36] Paul Mockapetris, "IETF RFC 1034: Domain Names - Concepts and facilities," Nov. 1987. [Online]. Available: http://www.ietf.org/rfc/rfc1034.txt

[37] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. of the 13th USENIX Conference on System Administration (LISA 1999)*, 1999, pp. 229–238.

[38] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010)*, 2010, pp. 186–196.

[39] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch, "Distributed Complex Event Processing with Query Rewriting," in *Proc. of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS 2009)*, 2009, pp. 4:1–4:12.

[40] S. Seol, M. Kim, S. Kang, and J. Ryu, "Fully automated interoperability test suite derivation for communication protocols," *Computer Networks*, vol. 43, no. 6, pp. 735–759, 2003.

[41] A. P. Sistla, V. Gyuris, and E. A. Emerson, "SMC: a symmetry-based model checker for verification of safety and liveness properties," *ACM Trans. on Software Engineering and Methodology*, vol. 9, no. 2, pp. 133–166, Apr. 2000.

[42] J. Song, T. Ma, C. Cadar, and P. Pietzuch, "Rule-based verification of network protocol implementations using symbolic execution," in *Proc. of 20th International Conference on Computer Communications and Networks (ICCCN 2011)*, Aug. 2011, pp. 1–8.

[43] J. Song, T. 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*, Apr. 2010.

[44] D. H. Steinberg and S. Cheshire, *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Dec. 2005.

[45] A. Tanenbaum, *Computer Networks*, 4th ed. Prentice Hall Professional Technical Reference, 2002.

[46] O. Udrea, C. Lumezanu, and J. S. Foster, "Rule-based Static Analysis of Network Protocol Implementations," *Information and Computation*, vol. 206, pp. 130–157, Feb. 2008.

[47] A. Vallejo, J. Ruiz, J. Abella, A. Zaballos, and J. Selga, "State of the art of ipv6 conformance and interoperability testing," *Communications Magazine, IEEE*, vol. 45, no. 10, pp. 140 –146, Oct. 2007.

[48] S. Vitkovsky, "dhquery," http://code.google.com/p/dhquery/.

[49] D. Wagner and R. Dean, "Intrusion Detection via Static Aanalysis," in *Proc. of IEEE Symposium on Security and Privacy*, 2001, pp. 156–168.

[50] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky, "Declarative network verification," in *Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL 2009)*, 2009, pp. 61–75.

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.