Rule-based Verification of Network Protocol Implementations using Symbolic Execution

JaeSeung Song, Tiejun Ma, Cristian Cadar, Peter Pietzuch
Department of Computing, Imperial College London
London SW7 2AZ, United Kingdom
Email: {jsong, tma, cristic, prp}@doc.ic.ac.uk

***Abstract*—The secure and correct implementation of network protocols for resource discovery, device configuration and network management is complex and error-prone. Protocol specifications contain ambiguities, leading to implementation flaws and security vulnerabilities in network daemons. Such problems are hard to detect because they are often triggered by complex sequences of packets that occur only after prolonged operation.**

The goal of this work is to find semantic bugs in network daemons. Our approach is to replay a set of input packets that result in high source code coverage of the daemon and observe potential violations of rules derived from the protocol specification. We describe SYMNV, a practical verification tool that first symbolically executes a network daemon to generate high-coverage input packets and then checks a set of rules constraining permitted input and output packets. We have applied SYMNV to three different implementations of the Zeroconf protocol and show that it is able to discover non-trivial bugs.

I. INTRODUCTION

Implementations of network protocols such as DNS, Zeroconf and OSPF frequently suffer from security problems and interoperability issues. Ambiguities in protocol specifications such as RFCs [1] can cause different interpretations by developers, even for well-studied and mature protocols. Such errors in network daemons are 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 [2]. For example, DNS server implementations that are vulnerable to DNS cache poisoning attacks [3] are difficult to detect because the vulnerability only exhibits itself in specific scenarios.

In practice, developers attempt to find flaws in network daemons using a combination of manual and random testing [4], code review [5], runtime debugging [6] and static analysis [7]. As the complexity of network services continues to increase, these methods become less effective. Random testing has low code coverage and thus may miss important vulnerabilities. Dynamic tools such as Daikon [8] and Valgrind [9] require test inputs and thus suffer from similar coverage problems. While static analysis of the source code of a network daemon benefits from high code coverage [7], [10], 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 [11], [12], such approaches cannot guarantee the correctness of the actual implementation.

Certain security vulnerabilities and implementation errors are only exposed when a network daemon handles pathological input packets. Recent automated tools for test generation [13] create high coverage test inputs via *symbolic execution* [14], [15]. They run programs on "symbolic" input values and then explore a large number of potential execution paths in order to generate actual test data for all traversed paths. Symbolic execution has been successfully used to find bugs in a wide variety of applications ranging from libraries to network and operating systems code [13], [16], [17].

We propose to execute network daemons on symbolic input packets in order to discover deviations from the protocol specification. To achieve this goal, we overcome two challenges: (1) due to their sizes, it is infeasible to make entire input packets symbolic—we show that good coverage can be achieved by repeatedly making combinations of packet fields symbolic; (2) based on the generated high-coverage input packets, we need to detect incorrect behaviour of the network daemon automatically—we propose a packet rule language that detects violations in observed input and output packets.

In this paper, we describe SYMNV, a verification tool that enables developers to create a link between specifications and implementations of network protocols. SYMNV automatically validates a network daemon against its protocol specification and discovers difficult to find *semantic* bugs. The input to SYMNV is the C source code of a network daemon and a set of rules extracted from the protocol specification that define correctness and security violations. Each rule describes invalid patterns of input and output packets. SYMNV uses symbolic execution to generate an exhaustive set of input packets that yield high source code coverage. It then replays these test packets to the network daemon and uses a rule-based packet analyser to detect rule violations, which indicate implementation errors.

To evaluate the effectiveness of SYMNV, we apply it to three network daemons that implement the *Zeroconf* configuration protocol. Using rules derived from the Zeroconf RFC specifications [18], [19], SYMNV finds two different types of violations in these implementations: *generic errors*, e.g., test packets that cause the daemon to abort and can be used to mount a denial-of-service attack, and *semantic errors*, e.g., test packets that expose incorrect behaviour in the implementation.

In summary, we make the following main contributions:

1) the application of symbolic execution to network daemons to generate high-coverage test packets;
2) the specification of a high-level, *packet rule language*

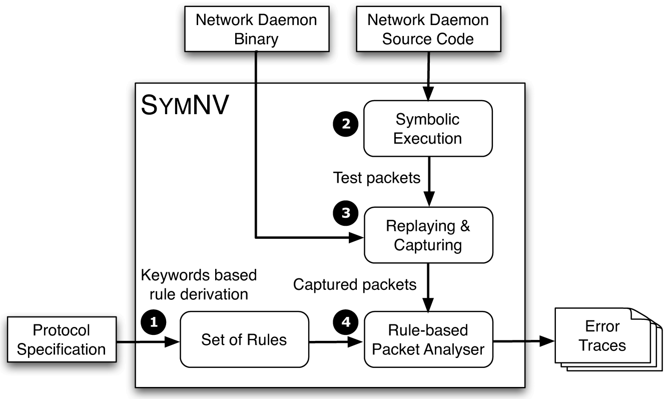


Fig. 1. SYMNV system architecture

using an event automata model for expressing constraints on packet sequences;
3) the implementation of a verification tool, SYMNV, for verifying network protocol implementations; and
4) an evaluation that describes real-world flaws in different implementations of the Zeroconf protocol.

The next section gives an overview of the SYMNV tool. §III describes how to obtain verifiable specifications using our packet rule language. In §IV, we discuss the test generation and replay process using symbolic execution. We then present our evaluation results and the discovered bugs in §V. The paper finishes with a discussion of related work (§VI) and conclusions (§VII).

II. SYMNV OVERVIEW

In this section, we provide an overview of the approach taken by SYMNV, which consists of two parts. First, SYMNV symbolically executes a network daemon to obtain a set of test input packets that result in high code coverage when processed by a network daemon. Second, it replays the set of test packets under controlled conditions and observes the output packets generated by the network daemon, which are validated for compliance against the protocol specification.

The SYMNV architecture is shown in Figure 1. When verifying a network daemon with SYMNV, there are four steps, as labeled in the figure:

1. Creation of packet rules. The first step is to develop a rule-based verifiable specification from a standard protocol specification. SYMNV provides a packet rule language to describe correct sequences of packets. A developer can write packet rules based on the protocol specification, e.g., by translating phrases containing specific words such as "MUST" and "SHOULD" into rules (cf. §III-A).

2. Generation of test packets. To validate as many packet rules as possible, SYMNV needs a good set of test packets that provide high code coverage. It uses *symbolic execution* to explore a large number of code paths in the network daemon and, based on this, synthesises a set of test input packets (cf. §IV-A).

3. Replay of test packets. The generated test packets obtained above are replayed on the original network daemon. Each test packet is sent to the daemon in a controlled environment, and the output packets generated by the daemon in response are recorded by SYMNV together with the input packet (cf. §IV-B).

4. Validation of packet rules. In the final step, the captured input and output packets from the previous step are validated against the packet rules from step 1. SYMNV translates the packet rules into a set of non-deterministic finite automata (NFAs). A *rule-based packet analyser* matches all captured replay packets against each NFA to detect rule violations. For each violation, SYMNV reports an error trace, i.e., the sequence of input and output packets that led to the violation (cf. §III-B and §IV-B).

III. VERIFIABLE SPECIFICATIONS

The first step in using SYMNV is to make a standard protocol specification such as an RFC document verifiable. A verifiable specification allows SYMNV to assess the correct behaviour of a network daemon automatically. We assume that the behaviour of a network daemon constitutes of the output packets that it emits in response to input packets. We define behavioural violations using a packet rule language that matches incorrect sequences of packets. In this black-box approach, we do not reason about the internal state of the network daemon, which means that packet rules are reusable across different daemons implementing the same protocol.

Next we first show how rules are derived from specifications (§III-A) and then introduce our packet rule language to express verifiable specifications (§III-B).

A. Rule Extraction

A set of rules can be extracted from the text of a network protocol specification such as an RFC or IETF standard. In many standards documents, words such as "MUST" and "SHOULD" are used to express requirements in the specification [20]. 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.

Consider how rules can be derived from the sentences in the RFC defining the Multicast DNS (mDNS) network protocol [18]. For example, we can find the following requirement related to the "Query ID" of a multicast DNS packet:

> *"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."*

This requirement says how an mDNS daemon has to set the Query ID for the response packet when it answers using unicast for a given query. If the daemon does not follow this desired 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 message. Therefore this requirement is a good candidate for a rule.

```
1 query{src_ip != 224.0.0.251
2      AND flag.QR = 0x00
3       AND questions != 0x00}
4 ;
5 resp {dst_ip = @query.src_ip
6       AND flag.QR = 0x80
7       AND ANY data.answer(
8              name = @query.question.name)
9       AND id != @query.id}
```

Fig. 2. Example rule for discovering inconsistent Query IDs in DNS packets

In our packet rule language described in the next section, violations of this requirement can be expressed as shown in Figure 2. This rule matches query and response packets satisfying certain predicates. The query and response packets must appear in sequence, as specified by the *next* (;) operator.

Most network protocol patterns with a general query/response model can be described using such rules.

To be reusable across different implementation, all rules must refer to externally observable aspects of packets. Therefore not all phrases from specifications containing these special keywords can be translated to rules. For example, the following requirement from the mDNS specification cannot be described as a rule because it refers to internal state maintained by the daemon that is not visible externally:

> *"A Multicast DNS Responder MUST NOT place records from its cache, which have been learned from other Responders [...]"*

B. Packet Rule Description Language

Since our packet rule description language is intended for use by developers of network services, two design requirements are readability and ease of integration with network protocols. 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 finite set of packet filter predicates. A packet filter predicate represents the possible values of the corresponding *fields* in packets that match this filter. Figure 3 lists some of the fields that are part of a DNS packet and can be referred to in rules. The finite 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 (.).

Consider the packet filter on lines 1–3 of Figure 2.

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 (`questions!=0x00`). It ignores packets that do not satisfy these filter conditions.

Rule operators. Based on such packet definitions, rule expressions can be built recursively using three operators: *next* (;), *union* (|) and *iteration* (+):

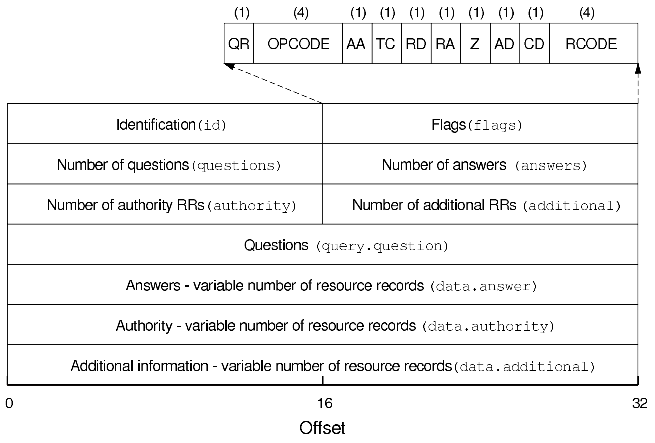

(1)
(4)
(1)
(1)
(1)
(1)
(1)
(1)
(1)
(4)
QR
OPCODE
AA
TC
RD
RA
Z
AD
CD
RCODE
Identification (id)
Flags (flags)
Number of questions (questions)
Number of answers (answers)
Number of authority RRs (authority)
Number of additional RRs (additional)
Questions (query.question)
Answers - variable number of resource records (data.answer)
Authority - variable number of resource records (data.authority)
Additional information - variable number of resource records (data.additional)
0
16
32
Offset

Fig. 3. Example of fields in a DNS packet

1) The **next operator** `Pkt1;Pkt2` detects the next occurrence of `Pkt2` after `Pkt1`, ignoring any intermediate packets that do not satisfy the filter predicates for `Pkt2`.
2) The **union operator** `Pkt1|Pkt2` matches a choice of packets `Pkt1` or `Pkt2`.
3) The **iteration operator** `Pkt +`n detects n consecutive packets `Pkt`.

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. For example, `ts >= @query.ts + 150` means that a rule matches a response packet with a timestamp that is 150 msec larger than that of the corresponding query packet.

Variable binding. Using variable bindings, fields from previously detected packets can be stored and referenced in subsequent filter expressions. A field name of the form `@pkt.field` refers to the field name `field` of a previous packet `pkt`.

Rule implementation. Packet rules are verified using non-deterministic finite automata (NFAs). We use an event model that is similar to the ones found in complex event processing systems [21].

The SYMNV automata operate 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 an automaton instance is in state S with assigned packet p. Each edge, say between states S and T, is labelled by 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 appear such that predicate $\theta(p, e)$ is satisfied. As a result, the NFA transitions non-deterministically to the next state T, as licensed by its transition function f and stores packet p in order to refer back to its field values later.

IV. VERIFICATION OF NETWORK DAEMONS

In this section, we describe how SYMNV generates high-coverage test packets using symbolic execution and replays those packets to discover violations of packet rules.

A. Symbolic Execution of Network Daemons

The use of *symbolic execution* [14] in testing is a well-known approach. The main idea behind symbolic execution is to use as input symbolic values—instead of actual data—and to represent the range of possible values for a given variable as a symbolic expression. One popular application of symbolic execution is the automated generation of test cases that achieve a high degree of source code coverage.

SYMNV executes a network daemon symbolically by marking set 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 daemon that are related to a symbolic variable. When encountering branches that depend on a symbolic value, symbolic execution generates test cases that trigger both execution paths.

SYMNV uses KLEE [13], a symbolic execution tool for C programs capable of automatically generating high-coverage tests and finding low-level bugs.

Marking packet fields as symbolic. Deciding which bytes to mark as symbolic has a big impact on the quality of generated test cases. In most cases, the behaviour of a network daemon is determined by the input packets that it receives from other daemons or clients. For example, a DNS server receives UDP query packets from clients and replies with a UDP response packet after having resolved the DNS name in the query packet to an IP address.

Usually a network packet consists of multiple fields that are part of the packet header and a data part. Most of the source code of daemons contains logic for handling these fields. Therefore, SYMNV treats entire fields as symbolic.

An open challenge is to decide which fields to mark as symbolic. Unfortunately, it is infeasible to mark the complete packet as symbolic because this would result in too many paths that would need to be explored during symbolic execution. Most of these paths would not increase code coverage because they would relate to invalid packets that are normally discarded by a network daemon. Instead, it is important to be strategic and only mark individual packet fields symbolic that are likely to result in the highest coverage gains.

SYMNV allows developers to mark any combinations of packets as symbolic. In our experiments in §V, we try all combinations of fields in DNS packets, starting with one field, and then progressively advancing to larger numbers of fields that are marked symbolic at the same time, with good results.

Injection of symbolic packets. Another important problem is to decide how to inject symbolic packets without requiring major changes to the daemon code. Conceptually, we would like the daemon to receive symbolic packets over the network. To implement this, we simply changed the code that receives an incoming packet to mark certain packet fields as symbolic.

As most C daemon implementations use the standard socket API to receive input packets, we found it easy to make this modification.

B. Generation and Replay of Test Packets

The verification process of SYMNV is composed of three tasks: *test packet generation*, *test packet replay* and *packet rule validation*.

Test packet generation. To run a network daemon symbolically, we first need to compile its source code to LLVM bitcode [22], the low-level language used by the KLEE symbolic execution engine. When the LLVM-compiled daemon starts, it behaves normally and waits for input. SYMNV then sends a specific test input packet to the daemon in order to trigger symbolic execution. When the daemon receives this test packet, it intercepts the packet and marks specified fields as symbolic.

For example, if the user provides an instruction to mark the *flags* field as symbolic, KLEE replaces the concrete value of this field within the packet with symbolic values while keeping the other fields concrete. KLEE then explores all possible execution paths (or as many as possible in a given amount of time) corresponding to the various input packets having different *flags* values. At the end of each execution path, KLEE generates a concrete test packet that is stored on disk.

Test packet replay. Generated test packets are then executed ("replayed") using the original network daemon. The replay process executes an unmodified native version of the daemon on all of the test packets generated by symbolic execution. SYMNV executes the unmodified network daemon under the same conditions under which the test packets were generated (e.g., by using the same configuration parameters). Replayed packets causing crashes are reported during the replay process.

To validate the network daemon, SYMNV captures all network traffic generated by the daemon and clients during the replay. For this, SYMNV uses `libpcap` [23], a portable packet capture library. The captured traffic is stored in a `.pcap` file, which is used as one of inputs to the next step.

Validation of packet rules.

To determine the correctness of the daemon implementation, SYMNV checks the execution of the replayed test packets against the packet rules extracted from the protocol specification. To verify a network daemon using SYMNV mechanically, a developer must provide an executable binary of the network daemon, a set of packet rules, and a test directory that contains all test packets captured during the previous replay phase. For example, the following command instructs SYMNV to verify a Multicast DNS daemon using a verifiable specification and a set of test packets:

```
symnv-validate -exe mdns mdns.rules testcases/
```

Packet rules are verified using non-deterministic finite automata, as described in §III-B. The input to each NFA are the `.pcap` files containing the traffic that was captured during the replay process. If an NFA ends in a violating state, SYMNV reports the violation and generates a trace, which lists the sequence of packets leading to the violation of the corresponding packet rule.

V. Evaluation

The goal of our evaluation is to demonstrate the feasibility of SymNV as an efficient verification tool for finding implementation flaws in real-world network daemons. Using SymNV, we discovered seven flaws in network daemons implementing the Zeroconf network specification [18], [19] caused by implementations mistakes and ambiguous requirements in the specification.

Zeroconf protocol. We center our evaluation around Zeroconf [18], a network discovery protocol that enables devices on an IP network to automatically configure themselves and their services and be discovered without manual intervention. Zeroconf is a serverless implementation of the DNS naming function built on top of standard DNS.

In Zeroconf, a new network service such as a file server or printer is added as follows. A client registers a new network service by selecting a service instance name. It then sends a service registration message to its local Zeroconf daemon. This causes the Zeroconf daemon to broadcast three probing DNS packets to the network, querying if the service name already exists. If there is no response, the daemon announces the service through a DNS announcement packet.

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.

We investigate three different implementations of Zeroconf using SymNV: Apple's *Bonjour 107.6*[1], *Avahi 0.6.23*[2] and *PyZeroconf 0.12*[3]. As Bonjour and Avahi are the most widely used Zeroconf implementations and written in C, we use them for symbolic execution. However, we use the generated test packets on all three daemons.

A. Deriving Rules

The Zeroconf protocol is defined as part of two RFC specifications: multicast DNS (MDNS) [18] and DNS-based Service Discovery (DNS-SD) [19]. 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.

To obtain a set of packet rules, as defined in §III-B, we examined both specifications to find phrases that contain the keywords from §III-A. In total, we found 110 phrases in the specifications: 79 phrases with a "MUST" keyword, 29 with "MUST NOT" and 2 with "SHALL/SHALL NOT".

Not all of these phrases could 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". For example, some statements were purely informative, and some contained 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, had to be ignored too. Finally, some phrases were used together to describe a single requirement. In total, we obtained a verifiable specification consisting of 25 rules based on 33 valid phrases.

[1]http://developer.apple.com/opensource/

[2]http://www.avahi.org

[3]http://www.amk.ca/python/zeroconf

B. Verification of Zeroconf

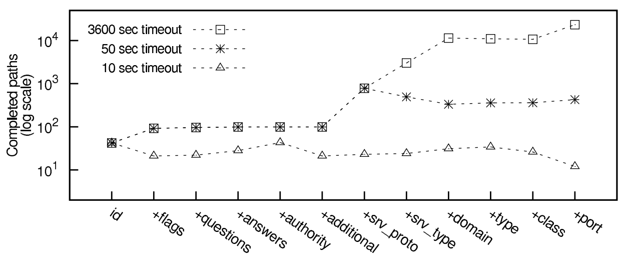
We run our experiments on a 2.4 Ghz Intel Core2 Duo machine with 2 GB of RAM under 32-bit Ubuntu Linux. To control network traffic during test packet generation and replay, all experiments are done as part of an isolated test network. Our experimental scenario involves two nodes: a Zeroconf daemon and a DNS-SD client. The Zeroconf daemon is executed using our SymNV verification tool. To simulate a typical environment, the DNS-SD client registers six services with the Zeroconf daemon. After registering these services, we inject a one-off query using the UNIX `dig` command to begin the symbolic execution of the Zeroconf daemon.

1) Test packet generation: The first step in the test generation process is to decide which fields in the packet to mark as symbolic. In our experiments, each DNS input packet has 11 fields. We start with the `ID` field as the only symbolic field, run KLEE to generate input test packets, and then progressively mark more fields as symbolic, rerunning KLEE. As more fields are made symbolic, the number of paths explored by KLEE increase dramatically. By default, KLEE generates one test packet for each path it explores. To avoid unnecessarily generating a large number of packets, we configured KLEE to generate only test packets for paths which cover new statements in the code. Furthermore, we also explored different timeout values for KLEE to stop the exploration of paths.

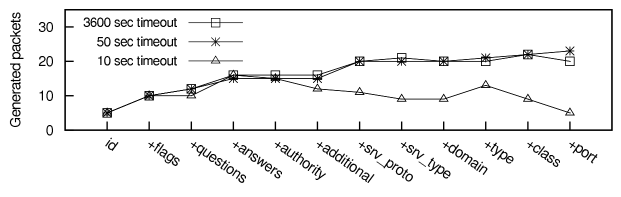
Figure 4 shows the number of explored paths and generated test packets when we increase the number of symbolic packet fields and use different timeout values. These experiments reveal two important insights. First, they suggest that a 50s timeout value offers a good tradeoff between the time needed to run the experiments and the number of generated test packets—with a 10s timeout KLEE generates significantly fewer test packets, but increasing the timeout to 1 hour does not significantly increase the number of generated packets (KLEE 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.

Second, using this approach KLEE generates relatively few test packets overall (under 250). Consequently, we decided to try all 4095 possible combinations of fields to mark as symbolic. Obtaining and comparing the amount of generated test packets for different combinations helped us understand the sensitivity of each field in the implementation. Using all combinations of packet fields, KLEE generated 32,069 test packets, with a total execution time of around 22 hours (each combination was timed out after 50 seconds).

Figure 5 shows 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



(a) Completed paths



(b) Generated packets

Fig. 4. Number of completed paths and generated test packets for accumulated fields with various KLEE timeout values in Bonjour

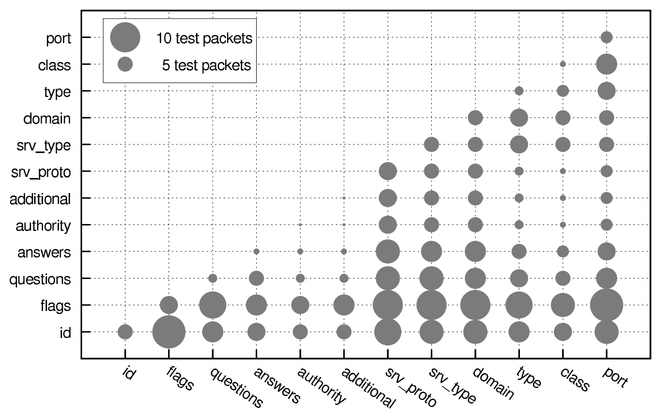


Fig. 5. Number of generated test packets for pairs of fields (subset of all combinations for Bonjour)

number of generated packets depends on which fields are marked as symbolic. For example, when the `flags` field is symbolic, a relatively high number of test packets is generated because the `flags` field contains control information that is used extensively by the network daemon to decide how to handle packets. On the other hand, when the `authority` or `additional` fields are marked as symbolic, fewer test packets are generated because these fields are only involved in simple checks used to decide the validity of a packet.

Finally, in order to analyse the sensitivity of each field in different implementations of the protocol, we run an experiment in which we mark one field at a time as symbolic, and compare the number of test packets generated for both Bonjour and Avahi. Figure 6 shows our results. As expected, since both implementations follow the same protocol, we obtain similar sets of test packets for the two daemons. For example, when we mark the `port` field as symbolic, it results in values 0, 512 and 5353 for Avahi and 0, 2, 5351 and 5353 for Bonjour. However, there are certain fields, such as `srv_proto` and `domain`, for which we obtain significantly more test packets for Avahi than for Bonjour. By examining the code, we discovered that the implementation used by Avahi to compare the different fields (e.g., domain names) is more complex, and requires more test packets to cover all possible code statements.

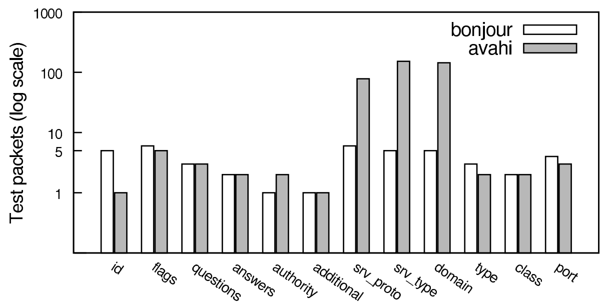


Fig. 6. Number of generated test packets for Avahi and Bonjour

To explore source code coverage, we focus on the Bonjour daemon. It has about 8K lines of source code in 10 files. On average, the generated test packets by SYMNV cover 61% of the code, while the baseline tests that execute the daemon without sending test packets only cover 20%. (We disabled unnecessary compile options and exclude library files from the calculation because they are not related to our experiments; coverage is measured using the `gcov` tool, which is part of the GNU GCC compiler suite.)

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 are used to handle such requests; another 13% implement other features such as cache maintenance and name conflict resolution.

2) Discovered implementation errors: Using SYMNV, we applied 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 source code, they can be used to test other Zeroconf implementations because they are highly effective test packets containing malformed data and corner cases.

SYMNV discovered seven different errors, four in the PyZeroconf implementation, two in both Avahi and Bonjour, and one in both PyZeroconf and Bonjour. We describe three of these errors below.

Violation 1: Vulnerability caused by source port number zero. When we mark 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 well-known; 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 query from a random port number. If the source port

```
1 mStatus mDNSPlatformSendUDP
2 (..., mDNSIPPort dstPort) {
3     ...
4     assert(m != NULL);
5     assert(end != NULL);
6     assert(dstPort.NotAnInteger != 0);
7     ...
8 }
```

Fig. 7. Code fragment from Bonjour daemon leading to abort error

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 these port numbers without any errors. However, we detect assert errors from Bonjour and Avahi. Both errors are caused by the source port number of a query packet.

In order to confirm these errors, we replay the test packets using the original network daemons. During this replay process, when SYMNV replaces the source port with 0 and sends the crafted packet, the daemons abort after receiving the packet. In the case of Bonjour, the daemon calls the `mDNSPlatformSendUDP` function to send a response packet. Line 6 in Figure 7 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 any running Avahi daemons in the network. Avahi daemons are aborted regardless of the existence of an answer because the responsible assertion is located in a function that handles any received packets.

Violation 2: Incorrect response for unknown record class. When the daemon receives a query packet asking for a specific service, it must compare three values—*name*, *type*, and *class*—against its records. The daemon responds to a query packet only 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)"*

From the above statement, we derive the following rule:

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

When we mark 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, PyZeroconf incorrectly sends a response even when it receives a query with an unknown class value.

Violation 3: Probing missing service. The DNS-SD specification requires every service to have a TXT record of the same name as the SRV record. This must be the case even if the service has no additional data to store, resulting in an empty TXT record. In addition, the mDNS specification states that a query for the purpose of probing the uniqueness of a record can be distinguished from a normal query by the fact that the query contains a proposed record in the "authority" section that answers the question in the "question" section. This means that when a client registers a new unique service, probing queries for the service have to include all related records with the same name (i.e. the PTR, SRV and TXT records) in the authority section.

Probing query packets from the Avahi daemon correctly include all related records. However, the Bonjour daemon does not include the TXT record in the authority section and PyZeroconf only includes PTR records without any SRV or TXT records. This behaviour violates a packet rule that matches a probing packet that does not contain all records types (PTR, SRV and TXT) in its authority records field:

```
probing{flag.QR = 0x00
        AND questions != 0x00
        AND auth_rr != 0x00
        AND ALL data.au(type !=
          ['PTR' | 'SRV' | 'TXT'])} +3
```

C. Discussion

Our experience with using SYMNV has yielded several insights. The majority of detected violations are caused by different interpretations of the same specification. Ambiguities in the specification may lead to interoperability problems between daemons. By translating textual specifications into verifiable rules, one can eliminate ambiguities. Since the rules only need to be extracted from a specification once, this can be done by domain experts who can resolve ambiguities correctly.

The number of generated test packets, the runtime and memory consumption of SYMNV, and the coverage achieved in the code are all heavily dependant 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 symbolic all possible combinations of fields, starting with only one symbolic field and incrementally making more fields symbolic seems to provide a good trade-off between runtime and code coverage.

VI. Related Work

As network services become more complex and error-prone, it becomes important to use automated techniques to verify their correctness. *Rule-based analysis* has already gained ground in the validation of network protocol implementations and the detection of intrusions and vulnerabilities [24], [25].

For example, Monitor [25] uses network rules to describe network behaviour and identify violations by monitoring real-time network traffic. However, their rule description language is not expressive enough to describe complex relationships between packets that are associated with many network errors.

Tools such as Pistachio [10] define network rules derived from specifications. Such systems bridge the gap between specifications and their implementation, but they achieve only low code coverage and struggle to detect rare errors. SymNV 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 SymNV, our packet rules can describe more complex sequences of packets compared to Pistachio's single input-output patterns.

Event processing systems can detect complex event patterns using pattern matching techniques, e.g., state automata [26] or event trees [27]. As automata-based models provide sufficient expressiveness for detecting complex sequences, SymNV uses automata to find violations in packet rules. Its packet rule language is similar to the one used by the NEXTCEP system [21] but is extended with primitives to make statements about packet fields.

Symbolic execution is a popular technique for generating high-coverage test cases and finding implementation flaws. Symbolic execution tools such as KLEE have been applied to a variety of application domains [16]. However, without using any high-level semantic rules, these systems have been limited to finding generic errors, such as division by zero or buffer overflows. By combining symbolic execution with rule-based analysis, SymNV has the ability to detect hard-to-find semantic bugs in network protocol implementations.

VII. Conclusions

In this paper, we described SymNV, a practical verification tool for network protocol implementations. SymNV combines symbolic execution with automata-based rule checking. After deriving a set of packet rules from a standard protocol specification, it generates a set of input packets using symbolic execution, and then replays them to discover rule violations in real-world network daemon implementations. We applied SymNV to three implementations of the Zeroconf protocol, and found seven non-trivial errors.

For future work, we plan to extend SymNV in a number of directions. First, we want to achieve higher code coverage through development of smart symbolic marking strategies. We will also investigate generated test cases from several network service daemons and compare them to assess interoperability between daemons in an automated fashion. Finally, we are extending SymNV to a framework providing fully automated network service verification across multiple network hosts. Within the framework, a runtime verifier is embedded into network service daemons and continuously monitors and checks the network state against given desired properties to provide correctness guarantees.

Acknowledgements. We would like to thank Matteo Migliavacca and Paolo Costa for their valuable comments on an earlier version of the paper.

References

[1] S. Bradner, "The Internet Standards Process," RFC 2026, 1996.

[2] 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 Symp. on NSDI*, 2008.

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

[4] R. Hamlet, "Random Testing," in *Encyc. of Soft. Eng.* Wiley, 1994.

[5] J. Remillard, "Source Code Review Systems," *Soft., IEEE*, vol. 22, no. 1, Jan. 2005.

[6] T. Leblanc and J. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. on Compt.*, vol. C-36, no. 4, Apr. 1987.

[7] D. Wagner and R. Dean, "Intrusion Detection via Static Aanalysis," in *Proc. IEEE Symp. on Security and Privacy*, 2001.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, vol. 69, 2007.

[9] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Electr. Notes in Theo. Comp. Sci.*, vol. 89, no. 2, 2003.

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

[11] G. Bochmann and C. Sunshine, "Formal methods in communication protocol design," *Communications, IEEE Transactions on*, vol. 28, no. 4, Apr. 1980.

[12] J. Song, T. Ma, and P. Pietzuch, "Towards Automated Verification of Autonomous Networks: A Case Study in Self-Configuration," in *8th IEEE Int. Conf. on, Pervasive Comp. and Comm. Workshops*, Apr. 2010.

[13] 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 Conf. on Op. Sys. Design and Impl.*, 2008.

[14] J. C. King, "Symbolic Execution and Program Testing," *Comm. ACM*, vol. 19, July 1976.

[15] D. Brand and W. H. Joyner, "Verification of protocols using symbolic execution," *Computer Networks (1976)*, vol. 2, no. 4-5, 1978.

[16] 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 Conf. on Comp. and Comm. Security*, 2006.

[17] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of Network and Distr. System Sec. Symp.*, 2008.

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

[19] Stuart Cheshire, Marc Krochmal and Apple Inc., "DNS-Based Service Discovery," March 2010, work in progress. [Online]. Available: http://tools.ietf.org/html/draft-cheshire-dnsext-dns-sd-06.txt

[20] S. Bradner, "Key Words for Use in RFCs to Indicate Requirement Levels," RFC 2119, 1997.

[21] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch, "Distributed Complex Event Processing with Query Rewriting," in *Proc. of the Third ACM Int. Conf. on Distr. Event-Based Sys.*, 2009.

[22] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of the 2004 Int. Symp. on Code Generation and Optimization (CGO'04)*, March 2004.

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

[24] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. of the 13th USENIX conf. on System admin.*, ser. LISA '99, 1999.

[25] G. Khanna, P. Varadharajan, and S. Bagchi, "Self Checking Network Protocols: A Monitor Based Approach," in *Proc. of the 23rd IEEE Int. Symp. on Reliable Distr. Sys.*, 2004.

[26] 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 *SIGMOD*, 2007.

[27] M. Mansouri-Samani and M. Sloman, "GEM: A Generalized Event Monitoring Language for Distributed Systems," *Distr. Syst. Eng.*, vol. 4, no. 2, 1997.