Enhancing Conformance Testing Using Symbolic Execution for Network Protocols

JaeSeung Song, Member, IEEE, Hyoungshick Kim, Member, IEEE, and Soojin Park, Member, IEEE

Abstract—Security protocols are notoriously difficult to get right, and most go through several iterations before their hidden security vulnerabilities, which are hard to detect, are triggered. To help protocol designers and developers efficiently find non-trivial security bugs, we introduce SYMCONF, a practical conformance testing tool that generates high-coverage test input packets using a conformance test suite and symbolic execution. Our approach can be viewed as the combination of conformance testing and symbolic execution: 1) it first selects symbolic inputs from an existing conformance test suite; 2) it then symbolically executes a network protocol implementation with the symbolic inputs; and 3) it finally generates high-coverage test input packets using a conformance test suite. We demonstrate the feasibility of this methodology by applying SYMCONF to the generation of a stream of high quality test input packets for multiple implementations of two network protocols, the Kerberos Telnet protocol and Dynamic Host Configuration Protocol (DHCP), and discovering non-trivial security bugs in the protocols.

Index Terms—Conformance testing, Kerberos, protocol verification, symbolic execution, Telnet, test packet generation.

I. INTRODUCTION

Security protocols are likely to evolve over time as new environments or technologies are developed. However, it is widely acknowledged that security protocols are hard to analyze by hand, and security bugs are difficult to detect. It is therefore desirable to look for efficient means by which flaws on protocols can be discovered and corrected with minimum effort.

Much of the existing research into testing techniques has focused on exploring protocol testing scenarios as much as possible, and examining the correctness of their implementations [13], [23]. Among the many techniques available, manual, random [1], [12], conformance [21], and symbolic execution [6] testing methods are predominantly used.

However, all of these testing methods also have their own inherent limitations. 1) Manual test case generation, in general, is expensive and slow due to the massive amount of tedious human effort to achieve high code coverage [19], [40]. 2) Random testing is a simple, well-known technique. By nature, however, it is ineffective when inputs needed to reach a given statement are very specific, and if there is only a little chance of randomly finding them in the input space [4], [31]. 3) Conformance testing aims to check the conformance of a protocol to a given specification. However, it does not touch tests beyond the specification of the protocol (e.g. unintended side effects).

4) Symbolic execution-based techniques execute a network implementation to explore all feasible execution paths associated with symbolic inputs. In theory, this approach may provide a neat solution for generating high coverage input test packets. In practice, however, the number of execution paths may be too large as it grows exponentially with the numbers of symbolic inputs or configurations [27].

Manuscript received December 04, 2013; revised May 03, 2014 and August 31, 2014; accepted November 17, 2014. The work of H. Kim was supported in part by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2014-H0301-14-1010) supervised by the NIPA (National IT Industry Promotion Agency) and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (No. 2014R1A1A1003707). The work of S. Park was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No. 2012M3C4A7033348). Associate Editor: S. Shieh.

J. Song is with the Department of Computer and Information Security, Sejong University, Seoul, Korea (e-mail: jssong@sejong.ac.kr).
H. Kim is with the Department of Computer Science and Engineering, Sungkyunkwan University, Suwon, Korea (e-mail: hyoung@skku.edu).
S. Park is with the Graduate School of Management of Technology, Sogang University, Seoul, Korea (e-mail: psjdream@sogang.ac.kr).
Digital Object Identifier 10.1109/TR.2015.2443392
In this paper, we propose SYMCONF (symbolic conformance testing), a testing tool that software developers can use to generate high-coverage test input packets through using an existing conformance test suite as seed input and symbolic execution.

At a high level, SYMCONF combines conformance and symbolic execution testing methods to overcome their individual limitations. Using a symbolic execution based technique together with selected input packets from a conformance test suite, a set of high-coverage test input packets can be generated to cover a very large number of execution states in an efficient manner. They are fundamentally different from other software programs such as Database or Graphic User Interface (GUI) programs (e.g. they typically need various events) because network protocol implementations typically require packet exchanges, and usually change their states based on input-output packets.

The technique using selected input packets makes symbolic execution applicable to test network protocol implementations. SYMCONF enables designers or developers to generate high-quality testing input packets that examine how well a network protocol implementation meets its protocol specifications, and explores all possible execution paths associated with the conformance testing, thereby achieving high-coverage of the tested implementation. In addition, it would still be helpful for developers who want to perform conformance testing of their systems, because SYMCONF uses an existing conformance test suite, and automatically generates high-quality test input packets.

We have designed and built a full implementation of SYMCONF on top of a symbolic execution engine. SYMCONF diagnoses a widely used security protocol implementation, and discovers several security vulnerabilities in the implementation. Our results show that the proposed technique can easily enhance an existing conformance test suite, and is highly efficient in detecting non-trivial bugs.

The main contributions of this paper can be summarised in the following points. 1) We provide a technique that repeatedly changes the points to perform symbolic execution toward deep source code execution paths to generate a high-coverage test suite. 2) We also provide a guideline explaining how to mark symbolic input packets to generate high quality test input packets while avoiding the path explosion problem. 3) And we share experience implementing SYMCONF to test real-world network daemons of widely-used protocols (i.e., the Kerberos Telnet and DHCP protocols). For these two network daemons, our approach achieved 83.2% (for telnet) and 76.3% (for udhcp) code coverage, where the average increased from the baseline tests is 20.6%. The experiments also provide a proof that shows our approach is feasible through finding seven bugs: two new non-trivial security flaws, and five memory errors.

The next section gives the relevant background and motivation for the SYMCONF conformance testing tool. We then present an overview of the SYMCONF process in Section III, followed by our techniques in detail in Section IV. We discuss our experience of using SYMCONF to test two network protocol implementations (i.e., Kerberos telnet and udhcp daemons) and find new security flaws in the tested daemons in Section V, and explore related work in Section VI. Our conclusions and future work are in Section VII.

II. BACKGROUND AND MOTIVATION

We begin with a brief explanation of network protocol testing to motivate SYMCONF, followed by the introduction of symbolic execution, the underlying technique used in SYMCONF.

A. Network Protocol Testing

The implementation of a network protocol should be tested to ensure that its intended functions work correctly as described in the protocol specification, and it properly communicates with other entities. However, for several reasons, including the existence of optional features and configurations of the protocol, and unclear or incomplete descriptions in the protocol specification, there are increased opportunities for multiple interpretations resulting in divergent implementations. Therefore, the research on network protocol testing has been mainly devoted to generating a test suite to capture nearly all specified behaviors in the protocol specification.

A practical solution for this problem is to complement the formulation of protocol standards (conformance testing) with a testing program to verify the interoperability of implementation from different vendors (interoperability testing). Fig. 1 illustrates the relationship between these two types of testing methods. 1) We need to test whether the protocol is correctly implemented by checking the conformance of the implementation to specification; with a sequence of inputs, we verify that it produces outputs that have the behaviors, as expected, in accordance with its specification.

More formally, given a finite state machine $M$ which acts as a specification, and another finite state machine $M'$ which is the alleged implementation that we can observe, we test whether $M'$ produces the same outputs as $M$. If they are the same, the test is successfully passed. Conformance testing has been intensively studied and formalized in several previous studies [2], [8], [42]. 2) We should also consider the interoperability between network entities as an inevitable testing process due to the nature of communication protocols.

However, these testing techniques can only show the presence of described functionalities (i.e., conformance testing) and interworking aspects (i.e., interoperability testing) so that their test cases, which are typically generated from the specification,
often do not cover enough execution paths, and miss detecting critical errors from the design and implementation.

For instance, let us consider a protocol specification for a service daemon having different behaviors according to the type of input packet. Typically, an implementation of such a protocol is structured as shown in Fig. 2. To test the implementation, a tester can generate two input packets: one for checking process_request() (line 7), and the other for exploring process_response() (line 9). The test packets can be used to check whether these two functions comply with associated requirements in the specification. However, the test packets alone do not cover unrelated functions such as parsing_packet() (line 4) and error_handling() (line 11), thus any hidden errors in these functions cannot easily be detected.

B. Symbolic Execution

Automated test case generation techniques have been proposed to overcome the limitations of low code coverage. Using such techniques, we can efficiently produce a set of test inputs for testing a target program. One of the most popular techniques is symbolic execution.

In symbolic execution, symbolic input values are used as input values instead of actual data, and to represent values of program variables as symbolic expressions. A program state is a mapping from program variables to terms built over symbolic input values, a path condition. The path condition is the conjunction of all the guards of all conditional branches that were performed to reach a particular point in an execution path of a program.

This condition means that symbolic execution can be used to examine the program’s behaviors along specific execution paths. Compared with other test case generation techniques, symbolic execution is generally able to generate a smaller amount of test cases that can achieve a high degree of coverage. Moreover, it provides the following advantages:

- Other verification techniques may want to use these particular test values rather than those derived from random generation.

Formally, symbolic execution uses a symbolic state associated with two kinds of information: a symbolic tree (ST), and a path condition (PC).

During execution, a set of constraints related to the inputs is maintained to PCs by the set of symbolic representations of each condition predicate along the path.

Example: To explain the main concepts used in symbolic execution, we provide an example below.

Let us consider the code fragment in Fig. 3, which processes a received input packet based on its flags value. It contains a statement abort() when the value of flags is equal to 0x0f0. The function returns QUERY when the flags value is 0x01, or otherwise returns RESPONSE. Symbolic execution starts from the first line of the source code, and proceeds, branch by branch, to the end of the program.

Initially, the PC is true, and flags has a symbolic value of x. At every conditional statement, the PC is updated with assumptions about the inputs to select alternative paths. For example, after the execution of the first conditional statement in line 5, a PC is created, initialized to PC ∧ x ≠ 0x01, and PC is updated to PC ∧ x = 0x01 (if branch). If the path condition becomes false, the symbolic execution does not continue for that path, but terminates. At the end of the symbolic execution of the path, the current PC is solved, and the output variables are represented as the test inputs.

Fig. 4 shows the corresponding symbolic execution tree. Initially, flags has a symbolic field value of x, which can be any value. At each branch point that is related to flags, the value for flags is updated with assumptions about the symbolic field. For example, after the execution of the second if statement (line 7), both alternatives of the if statement update the flags value accordingly, and one of them leads to the abort() error.

To use symbolic execution for network protocol implementations, we need to consider input packets from another network entity over the network as symbolic packets. A set of bytes of symbolic packets is marked as symbolic variables, and thereby processed symbolically during the execution of its network protocol implementation.

To perform symbolic execution for a particular programming language, a program called symbolic execution tool (also called symbolic interpreter) is required. In the domain of symbolic execution, such a tool is used in various ways to check whether a program is correct. Depending on the implementation of a
In this section, we provide an overview of the architecture taken by SYMCONF. The main goal of SYMCONF is to generate enhanced conformance test suites from existing conformance test suites so that it achieves a high-code coverage, and detects various hidden bugs. For this purpose, SYMCONF symbolically executes a target software program with carefully selected symbolic packets from a given conformance test suite. After that, SYMCONF performs conventional conformance testing on the target software with the enhanced conformance test suite. The behavior of the target during the conformance testing is observed to validate the compliance of the target against its protocol specification. SYMCONF reports any deviations from desired behavior as failures.

Fig. 5 shows the SYMCONF architecture. Traditionally, conformance testing executes a target software program with a given conformance test suite. On the other hand, to perform conformance testing for a target software called Implementation Under Test (IUT), SYMCONF takes two stages as labeled in the figure: 1) performing symbolic conformance testing to generate a high quality conformance test suite, and 2) conducting conventional conformance testing with the enhanced conformance test suite to detect any deviations from the desired behavior.

Stage 1. Performing symbolic conformance testing. To convert a given conformance test suite into a high quality conformance test suite, SYMCONF uses symbolic execution for an IUT. SYMCONF runs the IUT with input packets (called symbolic packets) specifically marked for symbolic execution. Deciding input packets such as the symbolic packet has a significant impact on the quality of generated test packets. SYMCONF selects symbolic packets iteratively from the input packets of the given conformance test suite.

In this stage, SYMCONF first uses concrete execution to warm up the state of a target daemon, then performs symbolic execution to explore various execution paths associated with symbolic packets. Details of the used symbolic execution techniques are described in Section IV. SYMCONF executes the IUT symbolically with the selected symbolic Packets, called symbolic execution mode (SEM), and this enables SYMCONF to explore many uncovered execution paths by the given conformance test suite, which results in generating high quality conformance test input packets. To avoid the path explosion problem, SYMCONF runs the IUT on a symbolic input packet by repeatedly marking parts of the packet as symbolic. SYMCONF distinguishes symbolic input packets from others through using an unused bit in the IP packet header.

Stage 2. Conducting conventional conformance testing. The generated test packets from each symbolic execution are integrated in a SYMCONF conformance test suite. The final SYMCONF conformance test suite therefore contains the existing conformance test suite together with additionally generated test cases from the previous stage. SYMCONF then performs conventional conformance testing on the original IUT using the generated SYMCONF conformance test suite. Each test packet in the suite is sent to the IUT in a controlled environment, and the exchanged input-output traffic is recorded by SYMCONF. The captured input and output packets are validated against the given conformance testing rules to discover bugs.

Based on the type of targeting protocol implementations (whether they are stateless or stateful protocols), SYMCONF can detect bugs triggered by both one and multiple input packets. If a target network daemon implements a stateless protocol, and provides various single input packets as its conformance test suite, SYMCONF can only detect bugs triggered by a single input packet. On the other hand, with a conformance test suite containing sequences of input packets, SYMCONF can be used to discover bugs caused after processing multiple input packets because SYMCONF generates test packets from each symbolic input packet, and combines the test packets into an extended conformance test suite.
IV. SYMBOLIC CONFORMANCE TESTING

In this section, we provide a more detailed description of our methodology used in SYMCONF. Our approach takes advantage of testing and verification techniques through integrating symbolic execution into conformance testing procedures. The approach extends existing conformance test suites to include more high quality test input packets, and achieves high source code coverage. It is easy to use, even for programmers without expert knowledge about the target software, but rigorously checks the target to detect non-trivial bugs.

The extended conformance test suite can be used to explore many execution paths that are not covered by traditional conformance testing techniques.

A. SYMCONF Preparation

The first step toward performing symbolic conformance testing, which results in generating high quality conformance test suites, is to select symbolic packets from a given conformance test suite. A set of selected symbolic packets is then used as an input to the SYMCONF system, together with the source code of a target software program.

Selection of symbolic packets. A conventional conformance test suite consists of a set of packet sequences in which each sequence describes a series of input and expected output packets. Each packet sequence is used to test a different execution path (i.e., program state) of an IUT. From these packet sequences, SYMCONF takes all the input packets and uses them as symbolic input packets (i.e., building a pool of symbolic packets). Later, SYMCONF selects each input packet from the symbolic packet pool in order, and performs symbolic execution with the selected input packet. It can generate high quality test input packets because performing symbolic execution on different input packets enables SYMCONF to start symbolic exploration from different program states. During symbolic conformance testing, SYMCONF keeps track of the order of symbolic packets and sequence information to select a proper symbolic input packet to be used.

Compilation of IUT into Bitcode. Because SYMCONF uses KLEE as its symbolic execution engine, it requires the source code to be compiled into a bitcode (instead of a native executable) using the LLVM compiler [24]. Using KLEE SYMCONF then interprets the compiled bitcode and functions as a hybrid between an operating system and the interpreter by providing environment models, and redirecting system calls to these models.

When the LLVM-compiled IUT starts, it treats all input-output packets concretely until it receives a symbolic packet. SYMCONF instructs a client to send the symbolic input packet to the IUT to run the IUT in symbolic execution mode. When the IUT receives the symbolic packet, it marks certain bytes (i.e., packet fields) of the symbolic input packet as symbolic variables, and starts to run itself symbolically.

For example, let us consider that the IUT has an if statement checking the destination IP address (dest_addr) of the input packet, and SYMCONF wants to mark the dest_addr field as a symbolic variable. When the IUT receives a symbolic packet, it replaces the value of the dest_addr field with a symbolic variable, and then explores all possible execution paths resulting in the generation of test input packets having different dest_addr values.

B. Symbolic Packet Identification and Marking Strategy

When SYMCONF performs symbolic execution on an IUT, symbolic packets are sent to the IUT together with other normal packets.

Identifying and marking symbolic packets are an important issue in SYMCONF because these affect the quality of the generated test cases. This section describes how SYMCONF identifies symbolic packets and marks them as symbolic variables.

Identification of the symbolic packet. As previously mentioned, a conventional conformance test suite consists of sequences of input and output packets. Per each symbolic execution, SYMCONF selects one of the input packets as symbolic. It is important to be able to distinguish the symbolic packet from other normal input packets because SYMCONF runs an IUT symbolically only when it receives a symbolic packet. SYMCONF supports two ways of identifying symbolic packets: symbolic bit, and symbolic keyword. Symbolic bit uses a specific bit in an input packet, while the latter uses a predefined keyword, such as sym-packet, that is embedded into a specific part of an input packet.

1) Symbolic bit. For the purpose of identifying symbolic packets, SYMCONF introduces a new bit, called the symbolic bit (S-bit). The S-bit is a single bit in the packet header to distinguish symbolic packets from normal conformance test input packets. When assigning the S-bit in the IP header of a packet, we can make use of the currently unused IP header bits. According to the RFC 791 Internet Protocol [33], the IP header contains two reserved bits: one in the Type-Of-Service (TOS) byte, and the other in the Flags field. Thus, when SYMCONF sends a symbolic packet to an IUT, it can use one of these two bits as the S-bit to indicate that the packet needs to be treated as a symbolic packet. For example, the Flags field consisting of three bits indicates fragmentation possibilities. The second (DF), and third (MF) bits indicate Don't Fragment, and More Fragment, respectively, while the first bit is unused. Thus, the first bit can be used as the S-bit.

Fig. 6 shows how the S-bit is marked and used in SYMCONF to generate high-quality test packets. When an IUT starts to run, it receives a certain amount of normal conformance testing packets. The IUT treats these normal packets concretely, so that it can explore a specific execution path guided by the packets. When the IUT receives a packet that its S-bit is set to 1, then the IUT marks the packet as symbolic, and starts to run symbolically to generate all possible test input packets.

2) Symbolic keyword. If these reserved bits are used for different purposes in a target IUT, SYMCONF uses a predefined keyword to distinguish symbolic packets. In this case, SYMCONF defines a specific keyword such as sym-packet. When SYMCONF sends a packet that needs to be handled as a symbolic packet, it crafts the packet to include the keyword. The IUT uses the keyword matching-with-position technique to detect symbolic packets. Because symbolic packets are delivered over the network, the location in the
source code, of which the IUT receives and processes input packets, can easily be identified.

Marking symbolic fields Marking the whole part of a symbolic packet to contain symbolic variables generates many more symbolic constraints that need to be maintained during the symbolic execution. Thus, performing symbolic execution with the complete input packet typically results in exploring too many execution paths, thereby causing the path explosion problem.

To solve this problem, SYMCONF marks a set of bytes alone, i.e., marks packet fields of a symbolic input packet as symbolic variables instead of marking the complete input packet. Through the experiment results, we could suggest a marking strategy: 1) divide a packet based on a packet field delimiter, which separates groups of data fields using a control symbol such as . and ; 2) select fields containing multiple meanings such as Flag in protocol specifications; and 3) only perform symbolic execution to the combination of these fields because the value of these fields are referenced a lot within a program.

After performing symbolic execution with the first packet field of the symbolic input packet, SYMCONF applies symbolic execution to the next packet field of the same input packet. SYMCONF iterates this process to all packet fields of the input packet, and generates test input packets. This increases the number of symbolic executionsthat need to be performed on each symbolic input packet. However, the proposed marking strategy can avoid the path explosion problem.

C. Generation and Execution of an Enhanced Conformance Test Suite

Our previous experience shows that, although performing symbolic execution on the first input packet improves testing coverage, many uncovered code regions still exist [41]. That's mainly because the approach cannot handle many branches that are associated with not only symbolic variables but also other constraints (i.e., path conditions). To explore these uncovered regions, SYMCONF uses an existing conformance test suite, and introduces a combined execution technique.

The program has an internal state, and changes the internal state when it processes an input packet. For example, an IUT, after receiving an input packet, parses the header and payload of the packet, and calls several functions. The header field values are then stored in the local variables to bring the IUT into a different state. In addition, these local variables are used in many branches as constraints. Typically, a conformance test suite helps to explore these different states and check functional correctness. The idea behind combined execution is to use conformance testing input packets to bring the program into different states before starting symbolic execution. Because this idea changes the starting point of the symbolic execution, and configures many constraints with proper values, SYMCONF can explore uncovered code paths.

Fig. 7 shows how SYMCONF combines concrete and symbolic execution when it has a conformance test suite having $n$ input packets, $p_1, p_2 \ldots p_n$. SYMCONF iteratively selects a symbolic input packet from the input packets, and performs combined execution per each selected symbolic input packet. Therefore, with the test suite having $n$ input packets, SYMCONF performs $n$ combined executions.

SYMCONF starts its operations with the first test input packet $p_1$, and treats $p_1$ as symbolic packet in the 1st turn of combined execution (label 1).

After performing symbolic execution on $p_1$, SYMCONF generates test input packets and puts them into an enhanced conformance test suite. Next, SYMCONF performs the 2nd turn of combined execution (label 2). This time, SYMCONF handles $p_1$ concretely, and $p_2$ symbolically. This approach enables SYMCONF to explore new execution paths that cannot be covered by symbolic execution with the first symbolic packet $p_1$. SYMCONF applies this process repeatedly to all symbolic input packets starting with the first test packet $p_1$ to the last packet $p_n$.

Formalize the symbolic exploration method.

When SYMCONF tests an IUT, it explores the execution paths of the IUT. Whenever it encounters a branch that is associated with symbolic variables, SYMCONF collects the path conditions, and generates test input values. As explained in Section II-B, within SYMCONF, generating test input packets through performing symbolic execution can be described as a tree whose
edges are generated test packets for a given symbolic packet, and nodes are labeled by symbolic states.

We now formalize the SYMCONF procedures introduced informally in the previous section. Let us consider that there is a given set of conformance test packets $\Sigma_{CT}$ for a program $P$. Because the test suite comprises a sequence of input and output packets, $\Sigma_{CT}$ can be decomposed as $\Sigma_{CT} = \Sigma_{I} \cup \Sigma_{O}$ where $\Sigma_{I}$ is the set of input packets $P_{11}, P_{12}, P_{13}, \ldots, P_{1n}$, and $\Sigma_{O}$ is their corresponding output packets. We also assume that each input packet is composed of $n$ packet fields. SYMCONF uses $\Sigma_{I}$ as symbolic input packets to generate an extended set of test input packets $\Sigma_{eI}$.

Fig. 8 shows the pseudo code for the SYMCONF exploration procedure. SYMCONF takes the two inputs: the symbolic packet field ($SF$) that provides the information of each packet field of a symbolic packet, and the conformance test suite ($\Sigma_{CT}$). After completing the exploration of the IUT, SYMCONF returns an enhanced conformance test suite $\Sigma_{eCT}$ as the output. To perform symbolic execution iteratively on each symbolic input packet and each packet field, SYMCONF internally operates two variables: a symbolic index ($SI$, variable (line 1)), and a symbolic field ($SF$) (line 5).

For each iteration of combined execution, SYMCONF checks that an input packet $p$ is a symbolic packet (lines 6 and 7). If $p$ is a symbolic packet, SYMCONF performs symbolic execution, and generates test packets (lines 7 through 10). Otherwise (i.e., $p$ is a normal conformance testing input packet), SYMCONF performs concrete execution on $p$ (line 12).

D. Performing Conformance Testing With the Enhanced Conformance Test Suite

SYMCONF now has a high quality conformance test suite derived from an IUT. The test packets in the suite are replayed with the original IUT to validate the IUT, and measure the source code coverage.

In this replay process, SYMCONF executes an unmodified version of the IUT, which is compiled with the native compiler, on all the test packets in the enhanced conformance test suite. Each conformance testing is performed under the same conditions under which SYMCONF generated these test packets. For example, before sending a test input packet, SYMCONF sets up the same configuration parameters, and registers the same services to the IUT that were used when the test packets were generated. Marking concrete input packets as symbolic has the potential to generate false positives (e.g., assertion failures) because of non-determinism in the target source code, and errors in the symbolic execution engine. The SYMCONF replay process removes this false positive issue because SYMCONF discovered errors during symbolic execution are all confirmed in the real environment.

KLEE, the symbolic execution engine of SYMCONF, itself provides a replay tool called klee-replay to support the ability to rerun tests outside of KLEE. However, klee-replay only works with test cases having a single input packet. To support the replay process with test cases that contain multiple input packets, SYMCONF provides a tool called SYMCONF-MULTI-REPLAY. The following command instructs SYMCONF to replay a target daemon $targetd$ with the given test case (false positives are then checked using the manual analysis).

```
symconf-multi-replay -d targetd -t test-cases
```

To validate the IUT, SYMCONF captures all exchanged network traffic between the IUT and clients during the replay process. To do this capture, SYMCONF uses libpcap [25], a portable packet capture library. Using libpcap, SYMCONF filters packets on a specific port and interface used during the test case generation. The captured traffic is then stored in a .pcap file, which can be used to validate the IUT through comparing them with the expected input-output packets. This type of rule specifying expected input-output packets is often provided together with a conformance test suite, or can be derived from the protocol specifications as proposed by [41]. During this conventional conformance testing, if any replayed packets cause crashes, SYMCONF reports the related sequence of input and output packets in detail.

V. EVALUATION

We evaluate the feasibility and usefulness of SYMCONF described in Section IV through the conformance testing to find bugs in a real-world network protocol implementation. Using SYMCONF, we explored roughly 83.2% and 76.3% of the source code for the Kerberos telnet daemon, and the udhcp daemon, respectively. We also found two security critical flaws in the implementation of the Kerberos Telnet, and reported these bugs to the developers. The reported bugs have been confirmed by the development team. However, they did not fix the vulnerabilities because this type of bug is an inherent problem with the protocol, and cannot easily be fixed. Note that it requires substantial manual effort to uncover new flaws in popularly used network daemons such as telnetd (http://linux.die.net/man/8/telnetd).

A. Network Protocol Implementations

Kerberos Telnet Protocol: To evaluate SYMCONF, we use the Kerberos Telnet protocol [43]. Kerberos [30] is an authentication protocol using a shared key encryption algorithm, which is based on the Needham and Schroeder trusted third-party authentication model [29], and uses timestamps to prevent re-use of tickets or replay in authentication protocols. Kerberos is a
widely used authentication protocol that is integrated into numerous security applications; a remote login protocol such as Telnet [32], for example, can use Kerberos to securely authenticate users. Telnet is an internet protocol enabling users to run programs remotely through interactive text-based communication sessions. After Telnet was developed in 1969, as one of the first IETF standards, it became very popular. However, due to the lack of authentication, several authentication mechanisms have been proposed. Kerberos versions 4 [3] and 5 [43] were introduced to support authentication mechanisms as well as the Telnet encryption option.

Because of security problems with the Telnet protocol, many users instead use the Secure SHell (SSH) protocol. However, it is not difficult to find users who still want to use Telnet over SSH for various reasons. For those users, Kerberos provides a secure Telnet version.

To discover bugs, and show feasibility, we analyze the Telnet daemon included in Kerberos5 release 1.7, which is specified in RFC2942 [43]. In practice, it is not easy to find bugs in such widely used open network protocol implementations due to a large number of execution states. From the standard specification, we selected a representative authentication procedure as a conformance test suite. This procedure consists of the following four steps (see Fig. 9).

- **Step 1.** The client starts with sending the WILL AUTHENTICATION command to initiate authentication. The daemon replies the message containing the DO AUTHENTICATION command to indicate that the initiation request is successfully accepted.

- **Step 2.** After receiving the message with the DO AUTHENTICATION command, the client then sends the account name that the user wants to be authenticated with the AUTHENTICATION NAME command. If the message from the client is valid, the daemon replies with the message containing the command of AUTHENTICATION REPLY to inform that the authentication is completed successfully. In addition, for secure communication, the daemon can send the WILL ENCRYPTION command similar to the first step negotiation for authentication.

- **Step 3.** In the third step, the client confirms the authentication mechanism by sending the AUTHENTICATION IS command to the daemon, and accepts the request for the encryption using the DO ENCRYPTION command. The daemon then tells the client which cryptographic algorithms for encryption are supported by the daemon by sending the message containing the ENCRYPTION IS command.

- **Step 4.** Finally, the client selects a particular cryptographic algorithm (or a set of algorithms), and sends the message containing the ENCRYPTION REPLY command to provide this information to the daemon. After the negotiation of cryptographic algorithms for encryption has been successfully completed, the daemon sends the message containing the information about the encryption key using the ENCRYPTION KEY ID command.

**DHCP Protocol:** As another application, we also use Dynamic Host Configuration Protocol (DHCP) [11] which is a widely deployed standard protocol to obtain network configuration information to hosts on a TCP/IP network. DHCP allows a server to dynamically assign network configuration parameters including IP addresses to clients. DHCP has eight types of packets, including DHCPDISCOVERY, DHCPOFFER, and DHCPREQUEST.

Fig. 10 illustrates a simplified state transition diagram of a DHCP server handling an IP address allocation request from a client. When a DHCP-enabled client is connected to the network, the client broadcasts a query packet (DHCPDISCOVERY) on the network for finding DHCP servers. When DHCP servers receive the DHCPDISCOVERY message, the servers select an appropriate IP address, and verify that the address is not already in use. The DHCP servers then respond to the client by broadcasting a (DHCPOFFER) message. The (DHCPOFFER) message includes the selected IP address, and information about services that can be configured for the client. The client responds to the message by sending a broadcast response packet (DHCPREQUEST) that specifies the selected IP address, and information about services that can be configured for the client. The server responds to the request packet with an acknowledgement packet.
(DHCPACK), thus completing the assignment process. Before leaving the network, the client notifies the server that the IP address is released by sending a packet (DHCPRELEASE) to release the address to the DHCP server. The server then returns the client’s IP address to the available address pool.

For DHCP, we select the udhcp 0.9.9-pre daemon. The udhcp daemon is an open source DHCP implementation used widely in many embedded devices.

B. SYMCONF Testing Environment and Implementation

We implemented SYMCONF on top of the KLEE symbolic execution engine [4]. Within SYMCONF, we also developed a client written in Python to inject a symbolic packet to invoke the symbolic execution of a target network program.

The experiments were performed on a 2.4 Ghz Intel Core 2 Duo machine with 2 GB RAM, running 32-bit Ubuntu Linux. The network was isolated to control network traffic during testing, i.e., standard conformance testing, test packet generation, and replay of SYMCONF generated packets. The testing scenario consists of two network nodes, a daemon, and a client for Kerberos telnet and udhcp. SYMCONF controls the two nodes, i.e., executing the daemon, indicating for the client to send symbolic packets, etc. To detect critical software bugs, mainly for improper termination, SYMCONF monitors the availability of the target daemon every 10 seconds. In addition, SYMCONF also detects any deviation from expected behavior (i.e., standard packets exchange) through analyzing input-output packets between the daemon and the client.

SYMCONF handles environmental functions such as reading files by using models which are provided by KLEE to simulate the behavior of the functions. KLEE redirects library calls to these models so that it increases code coverage through interacting with the Portable Operating System Interface (POSIX) file system.

However, there are several external library and system calls that SYMCONF cannot understand. To solve this issue, SYMCONF provides separate environment models for these library and system calls. Although SYMCONF cannot analyze these parts, it mitigates the external library call problems.

In what follows, we illustrate the experiments in detail, and introduce discovered bugs.

C. Extended Conformance Test Suite Generation

The first step in performing conformance testing in SYMCONF is to decide symbolic input packets from a given conformance test suite to generate high quality test input packets. In our experiments, the input packets of each conformance test step described in Fig. 9 are used as symbolic input packets.

To know which parts of the packet have to be marked as symbolic, we have applied various marking schemes, e.g., incrementally marking an input packet as symbolic from one byte to the entire bytes, marking each packet field and combinations of all the fields as symbolic. When we marked the entire input packet as symbolic, after generating a small number of test cases, SYMCONF cannot explore further code paths mainly because of multiple invocations of a hash function during parsing and validating procedures. After discarding the field delimiter from symbolic parts, we could work around the problem. Fig. 11 shows the results. Through the experiments, we could derive a smart symbolic marking strategy that allows SYMCONF users to generate high quality test packets efficiently as follows.

1) Divide a packet based on a packet field delimiter, and mark individual packet fields as symbolic variables.
2) Select packet fields containing multiple meanings (e.g., Flag and OPCODE) in protocol specifications, and mark them as symbolic variables.
3) Only perform symbolic execution to the combination of these symbolic packet fields because the values of these fields are referenced a lot within a program.

Based on the derived symbolic marking strategy, we start with the first input packet as a symbolic packet, and mark parts of the symbolic packet as symbolic variables. In the case of the first packet of telnetd, three bytes are marked as symbolic. We then run KLEE to generate all possible test input packets that the Kerberos telnetd can receive as the first input packet. As explained in Section II-B, symbolic execution generates one test case for each path exploration.

To not generate a large number of test cases, we configured KLEE to only generate test cases when it explores new statements. In addition, we instrumented KLEE to explore the daemon for a given timeout duration, i.e., 50 seconds. This timeout value was selected to generate enough test packets while not spending too much time exploring the daemon.
TABLE I
SUMMARY OF SYMBOLIC EXECUTION WITH CONFORMANCE TESTING INPUT PACKETS FOR THE DAMONS (telnetd, and udhcp)

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>telnetd</th>
<th>udhcpd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Total # of test cases from the first packet</td>
<td>8</td>
<td>2,347</td>
</tr>
<tr>
<td></td>
<td>(WILL and DISCOVERY)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Total # of test cases from the second packet</td>
<td>12,426</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>(AUTHENTICATION and REQUEST)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Total # of test cases from the third packet</td>
<td>400</td>
<td>2,004</td>
</tr>
<tr>
<td></td>
<td>(DO and RELEASE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Total # of test cases from the fourth packet</td>
<td>21,096</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(ENCRYPTION)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Total # of test cases</td>
<td>21,555</td>
<td>16,777</td>
</tr>
<tr>
<td>6</td>
<td>Achieved line coverage</td>
<td>83.2%</td>
<td>76.3%</td>
</tr>
</tbody>
</table>

Longer timeout values, such as 2 hours, do not increase the number of generated packets significantly, and often made KLEE run slowly because of memory consumption. To run KLEE, we use the following commands for all of our experiments.

```
klee --libc -uclibc --posix -runtime
    -- optimize -- max - time = 50
    -- only - output - states - covering - new
    -- use - random - path
    -- randomize - fork -- use - batching - search
    -- batch
    -instructions 10000
    -- load = /lib/libutil.so.1
    -- load = /lib/libncurses.so.5
    ./Telnetd.bc -a debug -debug 4567
```

After generating test packets from the first symbolic input packet, we run the same experiment with the remaining symbolic input packets. Every time, we mark parts of the symbolic input packet that are associated with the authentication and the encryption as symbolic, and run KLEE. All other fields, such as Terminal speed, Terminal type, and Display location, are ignored in our experiments.

Table I shows a summary of the symbolic execution. Symbolic execution with the 1st, 2nd, 3rd, and 4th symbolic input packet generates 8, 51, 400, and 21,096 test packet (in total, 21,555 test cases), respectively.

D. Performance Evaluation

The quality of a conformance test suite can be measured through analyzing the source code coverage that is explored by the packets in the suite. To measure the explored source code coverage, we use the gcov coverage tool 4.4.3 (http://gcc.gnu.org/), which is part of the GNU Compiler Collection (GCC) compiler suite. SYMCONF runs the native target daemons (i.e., Kerberos Telnet and udhcp), and instruments the client to send the generated conformance testing input packets. We compiled the daemons without unnecessary compile options, and did not include library files from the measurement, such as libutil, because they are not related to our experiments. According to our experiments, the measured lines of source code coverage for both daemons are about 1.3K (for telnetd) and 1.2K (for udhcp) from seven files. For example, in the case of telnetd, the measured files are authenc.c, slc.c, state.c, system.c, Telnetd.c, termstat.c, and utility.c.

For telnetd, overall, as shown in Fig. 12(c), the extended conformance testing packets generated from the given four input packets using SYMCONF achieve 83.2% code coverage after replay. The baseline tests that replay the daemon with the four given conformance testing packets cover 52.1%. We measured the coverage with the generated test packets from each step, and accumulated the achieved coverage. The generated test packets from the 1st step cover 55.4%, which is a small increase from the baseline test. However, the coverage increases as we replay the daemon with the generated test packets from the later steps, and finally achieves 83.2%. Fig. 12(a) shows the achieved code coverage of the files with generated test packets from each step. About 16% of the telnetd source codes are not explored with our test packets. These uncovered sources are mostly related to unused authentication options, unsupported server configurations, and the loop problem [18], [37], [45] that certain source code segments can be reached only after the loop is unrolled numerous times.

The increased coverage from using the extended conformance test suite is 31.1% in our experiments, and these results show that symbolic execution can be used to enhance existing conformance testing.

For udhcp, as shown in Fig. 12(d), SYMCONF generated conformance testing packets cover 76.3%, while the baseline tests only cover 66.5%. SYMCONF generated test packets cannot cover about 23% of the udhcp source code that are used to handle different types of input packets such as BOOTP packets and ip address allocation mechanisms. Similar to the test with telnetd, SYMCONF achieves increased coverage with the test packets generated from the later steps. Fig. 12(b) describes the coverage of the seven selected files with the test packets generated from each step. The files are selected based on the relevant symbolic input packets.

During the replay phase, every network traffic on the loopback interface is captured via libpcap [25], a portable packet capture library, and stored in a .pcap file. These network traffic files can be used to detect semantic errors as proposed in [41].

E. Bugs Found

Using SYMCONF, we applied the enhanced conformance test suite to the Kerberized Telnet daemon and udhcp daemon. The experiments detected in total seven test cases from the daemons, including two serious security bugs from telnetd, and five memory errors from udhcpd, all of which had been detected previously [5]. In this section, we describe details of the two security bugs.

**Bug 1 (Segmentation fault)**: SYMCONF found a vulnerability in the Kerberized telnet daemon Telnetd that could be used to launch a denial-of-service attack. The vulnerability allows an attacker to send a crafted Telnet packet with the user name replaced by the values of `xff` `/xf0` `/xff` `/xf8` `/x00`. This always causes a segmentation fault of the kerberized telnet daemon.
When we marked the name field of the second conformance testing input packet as a symbolic variable, 
SYMCONF generates 46 test cases. One of the generated test cases is

```
ff fa 25 03 ff f0 ff f8 00
```

During the replay phase with the given test case, as the user name is placed in the Suboption (Authentication option), the first 9 bytes of the crafted packet are replaced with the generated test case.

Within Telnetd, the replaced values of the user name are interpreted as a command because Telnet.h defines \`xff\' as a command option. Therefore, Telnetd understands the input packet with an empty name, and invokes a proper function call that can handle the command followed by the \`xff\' instead of parsing the user name field. The bytes followed by \`xff\' changes the state of Telnetd to TS_IAC as described by the `telrcv()` function in `state.c` as follows.

```
TS_DATA -> TS_IAC(ff)
   -> Sk(f0) -> TS_DATA -> TS_IAC(ff) -> EL(f8)
```

Fig. 13 shows a snippet of the EL (\`xf8\') character results in the execution of the following code excerpt from `state.c`, which is presented in Fig. 13. As this is not valid, an attempt at line 237 to assign proper a Set Local Characters (slc) value to the `ch` variable causes the daemon to trigger a segmentation fault.

The execution of the following command can cause a segmentation fault (replace `ip address` with the address of your test machine).

```
python dos.Telnet_seg_fault.py -h < ipaddress >
```

**Bug 2 (Potential denial-of-service attack):** SYMCONF found four test cases that cause the Telnetd client-daemon to become stuck, which could be potentially used to launch a denial-of-service attack on the server.

RFC 2941 [44], *Telnet Authentication Option*, says that WILL and DO are used only at the beginning of a connection to obtain and grant permissions for future negotiations. Therefore, the first message from the client must be a WILL Authentication Option (\`xff\' \`xfb' \`x25\') message, and the server must respond with a DO Authentication Option (\`xff\' \`xfd' \`x25\') message. Although there is no statement in the specification that states how to handle invalid starting messages, we presume that all
invalid options must be rejected because WILL and DO are the mandatory steps to complete the negotiation procedure.

When we marked the command field of the first conformance testing input packet as symbolic, SYMCONF generated 8 test cases, i.e., 27, 01, 25, 26, 06, FF, 21, and DO. During the replay, when the daemon receives a crafted packet replacing the command field value with the four generated test cases, i.e., Will Encryption Option \(\{xff, xfb, \times 26\}\), Will Binary Transmission \(\{xff, xfb, x00\}\), Will New Environment \(\{xff, xfb, \times 27\}\), and unknown \(\{xff, xfb, xff\}\), it does not reject the messages but responds with related DO options such as Do Encryption Option, Do Binary Transmission, and Do New Environment, respectively.

We observed that, as the client receives a DO option from the server, it again sends a WILL message to the server. This time the server does not respond to the WILL message because it has already sent the DO message previously. (The server avoids sending the same response to the client multiple times.)

As a result, the client and server get stuck in the middle of the negotiation process because the client expects to receive a message, but the server does not send anything.

This bug is not critical, but should be fixed because the current implementation of the Telnet daemon does not handle invalid messages correctly. The bug can be used to launch a denial-of-service attack on the server, exhausting resources with stuck instances of the Telnet daemon. The bug can be fixed easily by making the daemon implementation to reject all invalid initial messages from the client.

VI. RELATED WORK

Several techniques have been studied for checking compliance and generating test cases. An effective way to generate test cases is to use a formal specification given in the form of a finite state machine (FSM). Dorofeeva et al. presented an experimental evaluation of FSM-based conformance testing methods [10]. Endo et al. extended this work with recent methods (SPY, P), and different variations, such as the number of inputs, outputs, and states in FSMs [14]. Other related techniques also aim to produce test cases using various means. In particular, Hu and Ahn proposed an approach for the generation of conformance tests of access control policies through constraint verification [20]. Masood et al. also proposed the use of conformance testing methods to test generation for role-based access control policies [28]. These techniques mainly focus on generating conformance test suites, but require human involvement, and do not consider efficiency in general. On the other hand, our technique is mostly concerned with automatically generating conformance test cases, and having efficiency (i.e., covering as many source code as possible) as a high priority because we want SYMCONF to be usable with a variety of real-world software.

Recently, symbolic execution becomes a popular technique for generating high-coverage test cases, and finding implementation flaws. Symbolic execution tools such as SAGE [17], ESD [47], KLEE [4], and DART [16] have been applied to expose bugs in testing, or infer execution paths in a variety of application domains [5], [46]. These tools could benefit from the reduction in state-space achievable in a system, but they cannot cover all paths in practice. Instead of bounding the search space, Ma et al. proposed a heuristic to target particular lines of interest in a program [27]. We can find comprehensive surveys of symbolic execution elsewhere [6], [34], [38].

Symbolic execution tools such as Hunter [26] also propose a mechanism mixing concrete and symbolic execution to enhance system performance. Such tools perform mixed executions at an instruction level, i.e., only executing symbolic execution on instructions directly associated with symbolic input. They accelerate the overall system performance to a certain degree; however, their mechanism is limited to examining deeper execution paths. On the other hand, SYMCONF performs combined executions at an input packet level, which means that symbolic execution can be applied to selected input packets. This approach enables SYMCONF to examine wider source code regions than the tools using the instruction level mixed execution.

The technique that uses symbolic execution, and is most related to SYMCONF, is [15] by Gaston et al., a technique for using symbolic execution to generate test cases for conformance testing. The technique first derives a finite state machine of Input-Output Symbolic Transition System (IOSTS) from its specifications, which results in a symbolic execution tree. The paths in the symbolic tree denote the set of all the behaviors of the IOSTS system to be tested to confirm a compliance with its specifications. This kind of model-based testing technique [15], [22] enables the developer to test the functional correctness of target software. However, they inherit the most drawbacks of conventional conformance testing such as uncovering corner cases, and missing test cases for critical functions. These drawbacks exist mainly because the technique checks not the target software itself but an abstract model derived from the target software (or derived from its specifications).

Our work tackles such drawbacks through generating test cases directly from the source code using symbolic execution. Applying symbolic execution on the source code allows us to generate test cases for the most corner cases, and not miss any critical functions.

Symbolic execution has been used for many different applications. SymDrive [35] uses symbolic execution to test Linux
and FreeBSD drivers without their devices. FIE [9] provides an extensive platform using symbolic execution to detect bugs in firmware programs for the popular MSP430 family of microcontrollers. Cui et al. [7] proposed a rule-directed symbolic execution technique to efficiently check system rules. Sasnaukas et al. [36] presented a tool named KleeNet to discover bugs in sensor network applications. In particular, KleeNet and SYMCONF share some core ideas of this paper because KleeNet also uses an input packet as symbolic input, and proposes an approach enabling high-coverage testing. Unlike their approach, we focus on designing an enhanced conformance testing framework using symbolic execution with existing conformance test suites.

Our central idea is to repeatedly change the points to perform symbolic execution toward deep source code execution paths to explore uncovered lines of source code with conventional conformance test suites. This approach enables SYMCONF to test network security protocol implementations, and generate a high-coverage test suite.

VII. CONCLUSIONS

In this article, we proposed SYMCONF, a novel system that uses existing conformance test suites as a seed input, and generates high quality test input packets using symbolic execution. From a given existing conformance test suite, SYMCONF selects symbolic input packets to exhaustively explore a target protocol implementation. This approach results in an extended set of conformance test input packets. The SYMCONF generated test packets can cover various program paths that cannot easily be explored by the original conformance test suite so that they can achieve higher source code coverage. SYMCONF then performs a conventional conformance testing with the extended conformance testing packets on the native network implementation to find various bugs including non-trivial security flaws.

We have implemented a conformance testing tool for SYMCONF, and evaluated it on the two network daemons: Kerberos telnet and udhcpd for assigning dynamic IP addresses to devices. The results, i.e., detecting two critical system flaws with 83.2% source code coverage for telnetd, and five previously detected memory bugs with 76.3% source code coverage for udhcpd, show that SYMCONF enhances conventional conformance testing procedures through generating high quality test input packets based on symbolic execution.

In the future, we plan to extend SYMCONF in a number of directions. First, more experiments are required to show that the proposed approach is also applicable to complex network protocol implementations. In addition, the current version of SYMCONF is limited in application to other network protocol implementations. Thus we plan to generalize SYMCONF for various network protocol implementations without requiring any prior expert knowledge. A natural extension of this work is to use goal-directed heuristics to bring SYMCONF to places where there is a high probability of detecting generic type bugs.

We also plan to investigate the possibility of using SYMCONF for interoperability testing. Interoperability testing is generally considered as a complement to conformance testing because conformance testing only shows the compliance of a network protocol implementation to its specification. We will apply symbolic execution to interoperability testing to improve the effectiveness of interoperability testing in finding unknown flaws between network entities. We will continue to add functionalities such as interoperability testing, and publish SYMCONF under an open-source license. At the moment, SYMCONF can only detect generic errors (e.g., segmentation faults). However, to fully support automated symbolic conformance testing, we plan to investigate further mechanisms deriving pre- and post-conditions together with generated test cases.