IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 25, 2018, accepted January 18, 2019, date of publication January 25, 2019, date of current version February 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2895025

SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing

CONGXI SONG, BO YU, XU ZHOU, AND QIANG YANG
College of Computer, National University of Defense Technology, Changsha 410073, China

Corresponding authors: Bo Yu (526905729@qq.com), Xu Zhou (zhouxu@nudt.edu.cn), and Qiang Yang (q.yang@nudt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB0200401, in part by the program for New Century Excellent Talents in University, in part by the National High-level Personnel for Defense Technology Program under Grant 2017-JCJQ-ZQ-013, and in part by the HUNAN Province Science Foundation 2017RS3045.

ABSTRACT In recent years, the fuzzing technology is widely used to detect the software vulnerabilities owing to the coverage improvement in the target program and the easiness of use. However, it is less efficient to fuzz the stateful protocols due to the difficulties like maintaining states and dependencies of messages. To address these challenges, we present SPFuzz, a framework for building flexible, coverage-guided stateful protocol fuzzing. We define a language in SPFuzz to describe the protocol specifications, protocol states transitions and dependencies for generating valuable test cases, maintaining correct messages in session states and handling protocol dependencies by updating message data in time. The SPFuzz adopts a three-level mutation strategy, namely head, content, and sequence mutation strategy to drive the fuzzing process to cover more paths, in conjunction with the method to randomly assign weights to messages and strategies. We use the following metrics to evaluate the performance of SPFuzz and other frameworks upon three protocol implementations, i.e., *Proftpd*, *Oftpd*, and *OpenSSL*, which are three-granularity coverages specifically function, basic block, and edge. In experiments, the SPFuzz framework outperforms the existing stateful protocol fuzzing tool Boofuzz by an average of 69.12% in three granularities coverage tests. This demonstrates that the SPFuzz has the ability to explore more and deeper paths of the target program. We further triggered CVE-2015-0291 in OpenSSL 1.0.2 with the SPFuzz, which proves the validity and utility of our framework.

INDEX TERMS AFL, coverage, fuzzing, network protocol, software security.

I. INTRODUCTION

Network protocols, which describe the format of network messages and how network endpoints communicate between layers, always play a key role in all network-related fields [1]. However, nowadays, they become breakpoints of remote attacks toward systems over the Internet by exploiting numerous vulnerabilities in their implementations. Therefore, it is urgent for us to pay close attention to the security risks of the network protocol. To mitigate such a situation, it is extremely important to build an effective method to discover flaws and weaknesses in network protocols [2]–[4].

In recent years, the fuzzing technology is widely used in protocol vulnerability detecting [5], [37]. During the fuzzing process, the system is fed with random and unexpected data to improve the coverage of target programs [6]. Based on this situation, some fuzzing tools are designed for network protocols with the consideration of protocol features.

The associate editor coordinating the review of this manuscript and approving it for publication was Shenghong Li.

Peachfuzz [7] supports distributed multi-protocol fuzzing. It parses responses and is able to distinguish between data types and vulnerabilities. SNOOZE [8] is a fuzzing tool designed for stateful protocols. It utilizes the Extensible Markup Language (XML) [32] to describe the protocol specifications and generates test cases according to the defined fuzzing scenario. Boofuzz [9] is a Python-based fuzzing framework extended from Sulley [18]. It features maintaining states and identifying failures. The framework implements primitives for describing different types of message fields and mutation libraries for producing test cases. However, all the above fail to handle dependencies.

Leveraging the generation method, Kitagawa designed the Aspfuzz [10] to generate test cases based on the Request For Comments (RFC) [35]. In 2010, Gorbunov designed the AutoFuzz [11] framework, which firstly extracts finite state machines between clients and servers, and then the GMS template to find the valuable message fields. AutoFuzz marks these fields, and does experiment with File Transfer protocol

(FTP) [23]. However, it's unable to process encrypted data. The main perspective of the SecFuzz proposed in 2012 is to define the mutation strategy as three levels, i.e. sequence, payload and field. However, mutation strategies here in each level is too monotonous. The SecFuzz can cope with encryption mechanisms used in Internet Key Exchange protocol (IKE) [36] by searching the client's logs. In 2015, Ruiiter and Poll [12] proposed a method to extract state transition using the Learnlib state machine, and established an input alphabet. The test harness initializes the encryption key by sending ''ClientHello'' message and storing key in response message. This was tested on nine implementations of the Transfer layer security (TLS) [24] protocol. Novickis [6] also took a similar approach upon OpenVPN. Others used statistical and intelligent methods to improve the performance of network protocols fuzzing [1].

Munea *et al.* [2] summed up five criteria for the classification of protocol fuzzing through research in 2015 as follows:

- Intelligence level
- Method of producing test cases
- Method of detecting vulnerabilities in a server
- Ability of storing the previous states of sessions
- Method of sending inputs to a server

According to the above criteria, we investigate recent fuzzing tools and find the following:

- Most of these approaches spend few efforts on the self-learning of protocol specifications.
- Test cases usually tend to be generated by mutating real messages rather than producing inputs based on protocol standards to improve fuzzing efficiency.
- Most of these approaches ignore technologies such as dynamic memory analysis when detecting vulnerabilities in a server because of unavoidable overhead, and analyze vulnerabilities manually.
- The ability to store previous states of sessions is necessary for fuzzing stateful protocols.
- As to the last criteria, one way is keeping the correct order of message sequence while sending inputs, and then mutating message data. The other is mutating both the order and the data.

Researchers should take care of these criteria to tradeoff between the ability to discover vulnerabilities and the overhead caused by tools [10], [13].

A. MAIN CHALLENGE

1) STATES IN THE STATEFUL PROTOCOL FUZZING ARE HARD TO MAINTAIN

Stateful protocol fuzzing is much more complex than stateless fuzzing. For example, the inputs of Hypertext Transfer Protocol (HTTP) [33] are independent, regardless of state retention, while fuzzing a stateful protocol such as FTP requires the fuzzer maintaining a state machine which contains information of how the states are transferred. Then the fuzzer establishes all previous states and keeps the connection based on the state machine to fuzz a target state [15].

2) DEPENDENCIES ARE HARD TO HANDLE

For most of stateful network protocols, there are intra- and inter-message dependencies. For example, the length field or the checksum field is determined by the content, which is a type of intra-message dependencies. The pre-order messages carry the authentication or the encryption information required by the subsequent messages, and this produces dependencies between them, which is a type of inter-message dependencies. These dependencies impact whether we can produce valid test cases when fuzzing.

3) PATH COVERAGE IS HARD TO IMPROVE

Many frameworks generate test cases by formulating some templates according to the specifications of protocols. This method with manual intervention will make a lot of paths not be covered, while hidden vulnerabilities may exist there [14].

B. SPFUZZ FRAMEWORK

In this paper, we propose SPFuzz, a framework that can maintain the session states and the connections of network while fuzzing. Meanwhile it solves the problem of handling message dependencies. What's more, a three-level mutation strategy i.e. head, content and sequence, is used to improve the coverage of fuzzing.

The SPFuzz framework defines a description language to describe the specifications, the state transitions, and dependencies of protocols. The interpreter translates description files into message entities and message sequences that the framework can recognize.

The SPFuzz framework formulates a three-level mutation strategy including head, content and sequence mutation strategy. The three-level strategy combines both the knowledge-driven mutation from the protocol specification and the American Fuzzy Lop(AFL) [17] random full-strategy mutation. What's more, SPFuzz leverages the mutation ability of AFL to generate test cases randomly, which avoids descripting protocol specification with too many details.

Intra- or inter-message dependencies are handled by updating corresponding fields of message.

According to different message characteristics, both the choices of message to be fuzzed and the mutation strategy are given different weights, which produce different selective probabilities for making the fuzzing process traverse more paths and prone to discover vulnerabilities.

The framework schedules the whole fuzzing process. Meanwhile, it communicates with the AFL engine and the target server. Also, it monitors response messages of server and collects traces, all of which are fed back to test cases generation.

C. CONTRIBUTION

We make the following contributions in this work:

1) We define a description language which describes message specification, state transition and dependencies.

2) We design a three-level mutation strategy (head, content and sequence).
3) We implement a framework called SPFuzz to improve coverage with AFL features and weight selection.
4) We conduct experiment, the results show SPFuzz has a higher coverage than Boofuzz and AFL. Also, the SPFuzz framework triggers a known vulnerability.

D. PAPER ORGANISATION

The rest of this paper is organized as follows. Section II gives the background of the fuzzing technology. Section III presents the overview of the SPFuzz framework. The description language is defined in Section IV. Section V introduces the three-level mutation strategy. The scheduling details of SPFuzz is shown in Section VI. Followed by evaluation results are summarized in Section VII. Section VIII discusses some limitations of the current design and the future research direction. Section IX concludes this paper.

II. BACKGROUND

A. FUZZING TECHNOLOGY

Fuzzing is a software testing technique that looks for bugs by feeding random inputs into target programs so as to cover as many code paths as possible. Fuzzing tools can be divided into ''dumb mode'' and ''intelligent mode'' according to whether it understands the characteristics of the target program [27]–[30]. While the input of fuzzing is either generated or mutated, i.e. making new inputs based on a specification or mutating the existing inputs. The second approach is widely adopted in practice.

In some of programs, previous messages received is stored and used to affect the processing of the current message. This is called a ''stateful protocol'' [34]. The stateful network protocol fuzzing is different from the file fuzzing. First, we need to consider the protocol specification to generate valuable inputs; Second, the transmission of messages is stateful, and the input has dependencies; Third, the order of test cases sending to the target may lead to different results. For example, in the FTP protocol [23, Fig. 1], according to the correct order, the first message of sequence is ''USER'', followed by a corresponding ''PASS'' message. After the connection is established, data can be transmitted with a ''STOR'' or ''RETR'' message. If the previous states are not maintained, the subsequent fuzzing is meaningless.

In addition, there are some problems in the stateful network protocol fuzzing process. First, the states of the target should be recorded by a monitor. If a crash happens or the fuzzing tool loses connection with the target, the monitor logs the crash and feeds traces of process back to the fuzzing tool; Second, handling dependencies in messages is a challenge, such as the ''Heartbeat'' message of the TLS protocol [24] whose length field is determined by the length of content field.

Because of above challenges and difficulties, it's worthy to study the efficient fuzzing of stateful network protocols.

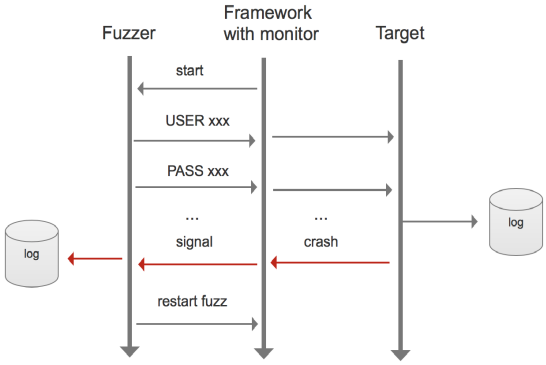


FIGURE 1. The stateful protocol fuzzing process example, FTP.

B. INTRODUCTION OF AFL

American Fuzzy Lop(AFL) [17] is a popular off-the-shelf fuzzing tool. It depends on mutation strategies to achieves high coverage. AFL maintains a queue which stores input files as seeds for mutating each cycle. In addition, AFL uses five strategies to generate a large number of test cases, and instruments the target program with a bitmap to compute coverage. The proposed SPFuzz framework leverages the mutating characteristics of AFL to fuzz the content field of a message.

III. FRAMWORK OVERVIEW

In order to solve the challenges in fuzzing stateful protocols, we design a powerful framework called SPFuzz, which contains not only a description language for specifications of protocols, but also a three-level fuzzing strategy. We implements a framework with functional components to schedule the fuzzing process.

A. MAIN COMPONETS

The overview of SPFuzz is shown in Fig.2, which consists of the following several parts:

1) DESCRIPTION FILE

It covers protocol specification description including the name of messages, the specifications of each field in the message and dependencies, and the protocol state transition description. We define a description language which is designed based on the well-format function of Boofuzz and make some extensions to adapt to our framework.

2) INTERPRETER

The interpreter is responsible for translating the description language, specifically the protocol specification description file into message entities to produce test cases, and the state transition description file into a message sequence.

3) CORE SCHEDULER

The core scheduler schedules the hierarchical fuzzing process using a three-level strategy, namely head, content and

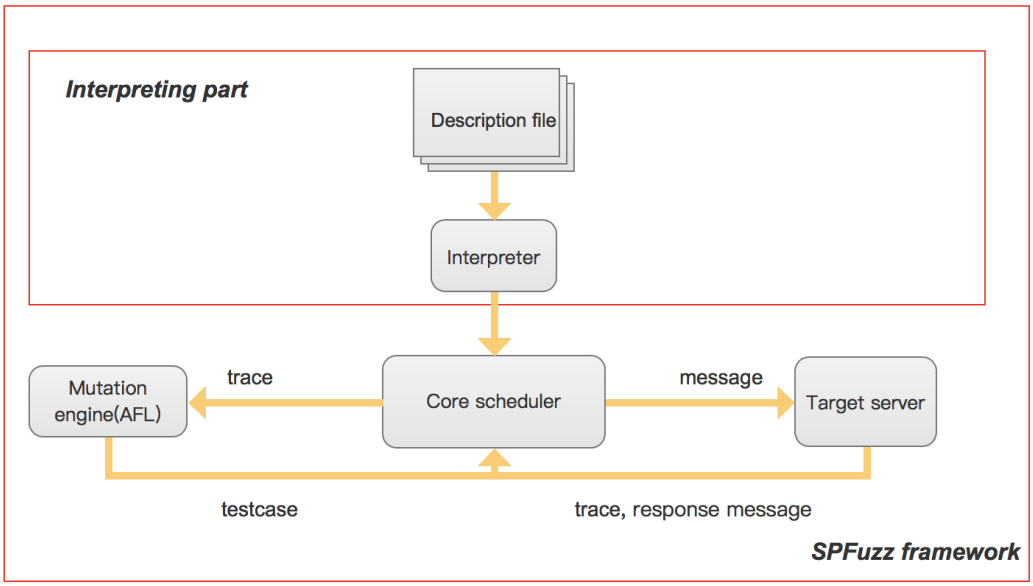


FIGURE 2. The overview of the SPFuzz framework.

sequence. Meanwhile, it is responsible for handling dependencies and communications.

4) MUTATION ENGINE

We leverage AFL as the content mutation engine that produces test cases based on original seeds and feeds test cases back to the core scheduler.

5) TARGET SERVER

The target server receives test cases and generates corresponding responses. These response messages are returned to the core scheduler along with traces of the target process for follow-up analyses.

B. WORK FLOW

The work flow of SPFuzz is shown below:

1) Users define the protocol specification file and the protocol state transition file.
2) The interpreter translates the description files into message entities and a message sequence.
3) The SPFuzz framework starts a round of fuzzing process.
4) The SPFuzz framework randomly selects a message for testing in the current round with the message weight.
5) The SPFuzz framework gets the whole message sequence of the current message which is used to build previous states. And then it updates the strategy weight according to the message traits, e.g. it would to select the head strategy for those messages have many head fields.
6) The SPFuzz framework randomly selects a strategy from three types (head, content and sequence) with strategy weight.
7) The SPFuzz framework performs corresponding strategy mutation on the message and receives test cases from AFL if the content mutation strategy is adopted.
8) The SPFuzz framework formats test cases.
9) The SPFuzz framework constructs the states before the current message according to the message sequence.
10) The SPFuzz framework sends test cases to the target server.
11) The target server sends response messages and traces, and then the framework analyzes and feeds information back to the next round of fuzzing.
12) The SPFuzz framework is reset for the next fuzzing round.

IV. DESCRIPTION LANGUAGE DEFINITION AND TRANSLATION

The SPFuzz framework has to understand the specification of the network protocol being fuzzed including the format of message in protocol and the message sequence to build a session. This knowledge can be obtained from the RFC. Therefore, we define a description language extended from that of Boofuzz. The description language comprises the protocol specification description and the protocol state transition description. Not only the format of the protocol, but also the dependencies intra- and inter-message in the protocol are described in the specification description file. Compared with the description languages of other tools [7]–[9], the SPFuzz framework makes a coarse-grained description and does not define each field of a message, which decreases the manual intervention.

A. PROTOCOL SPECIFICATION DESCRIPTION

Fig.3 illustrates the format of two messages: "USER" and "PORT" of the FTP protocol. A complete message

```
s_initialize("USER",weight=2)
s_block_start("HEAD")
s_string('USER', size=4 ,type='cmd')
s_delim(" ")
s_block_end()
s_block_start("CONTENT")
s_string('ftp')
s_static("\r\n")
s_block_end()
end
s_initialize("PORT",weight=1)
s_block_start("HEAD")
s_string('PORT',size=4 , type='cmd')
s_delim(" ")
s_block_end()
s_block_start("CONTENT")
s_string('192,168,122,1,201,138')
s_static("\r\n")
s_block_end()
end
```

FIGURE 3. An example of the protocol specification description.

description is in three levels. The first level is ''request'', which declares the name and the message weight in protocols. The **message weight** here indicates how important the message is in the protocol. It has influence on the probability of the current message to be fuzzed in all messages of the protocol. The second level is ''block'', which cuts the ''request'' into head field and content field for selecting different mutation strategies. The head block is represented in description file as ''s_block_start(''HEAD'')'' and the content block is represented as ''s_block_start(''CONTENT'')''. The third level is ''primitive'', which is responsible for dividing each field in a head block. The fields that are worthy to fuzz are represented by ''s_string'', while the meaningless fields such as separator and padding are denoted by ''s_delim'' and ''s_static''.

B. PROTOCOL STATE TRANSITION DESCRIPTION

Fig.4 is part of protocol state transition description depicting the process of how to move from initial state to the current state step by step, where ''s_connect'' is the transfer action. The whole sequence of messages is constructed according to the link order.

```
PORT
session.connect(s_get("USER"))
session.connect(s_get("USER"), s_get("PASS"))
session.connect(s_get("PASS"), s_get("SYST"))
session.connect(s_get("SYST"), s_get("PORT"))
```

FIGURE 4. An example of the protocol state transition description.

C. PROTOCOL DEPENDENCY REPRESENTATION

There are two kinds of dependencies in protocol. One is intra-dependency that the dependency between fields within a message, e.g. the length field which is determined by the content field. The other is inter-dependency that the dependency between messages, e.g. in some test cases whose data is controlled by pre-order messages. Sometimes these dependencies affect the generation of valid test cases.

Accordingly, SPFuzz defines two kinds of dependency in protocol specification description.

1) DEPENDENCY REPRESENTATION BETWEEN MESSAGES

The inter-message dependencies refer to these between messages between the pre-order message and the subsequent message, and between the sent message and the response message. The representations are shown as below:

- outDep(type,msg1,field1,msg2,field2)
- resDep(type,msg1,field1,res_type,res)

where ''type'' indicates the type of dependency, ''msg1'' and ''field1'' indicate the message name and field name of pre-order messages, and ''msg2'' and ''field2'' indicate the same information of subsequent messages.

2) DEPENDENCY REPRESENTATION WITHIN A MESSAGE

The intra-message dependency representation is shown as below:

- inDep(type,msg1,field1,field2)

where ''type'' indicates the type of dependency, ''msg1'' indicates the message name, and ''field1'' and ''field2'' represent two dependent fields.

D. TRANSLATION OF DESCRIPTION FILE

As is shown in Fig.5, the two description files are translated by the interpreter into an accessible format for the other parts of the SPFuzz framework. The interpreter extracts the specifications of all messages from the protocol specification file and generates message entities. It also extracts information from the state description file during the state transition process and collects the preamble messages of the current fuzzed message. These preamble messages are uniformly placed in the message sequence for state construction.

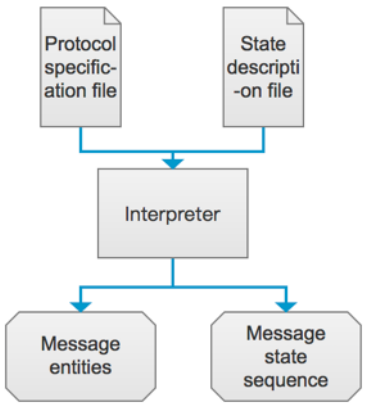


FIGURE 5. The input and output of the interpreter.

V. HIERARCHICAL FUZZ MUTATION STRATEGY SCHEDULING

Considering the internal structure of messages and the connection between them, the mutation strategies for fuzzing thoroughly are designed hierarchically, which covers three levels including head, content and sequence.

A. HEAD MUTATION STRATEGY

The header part adopts a customized strategy according to the semantics and specifications of message header fields shown in the Table 1. After analyzing RFC document manually, we found that header fields can be sorted into the following categories: ''length'', ''marker'', ''version'' and ''command''. Thus the mutation policies could select headers within the normal value range, or with the special value or with the illegal value such as null, zero, a large number and a string whose length is out of bounds.

TABLE 1. Items of head mutation strategy.

Field class	Type of data	Strategy
length	integer	random number
		large number
		zero
		null
marker	fixed integer or string	random value
		special value
		zero
		null
version	integer in range	within-range random number
		out-of-range random number
		zero
		null
command	string in range	within-range random string
		out-of-range random string
		zero
		null

For example, the change within normal range policy will mutate a header of ''USER xxx'', while it may also be mutated into ''eRD2 xxx'' by the special value selection or ''0 xxx'' by the illegal value selection.

B. CONTENT MUTATION STRATEGY

Data in content fields of messages is usually random and messy. Even so, with the help of AFL, the SPFuzz framework could generates test cases for content fields randomly which is used to perform a full-strategy mutation. AFL uses strategies including bitflip, arithmetic, interest, dictionary and havoc, and its engine tends to choose one that discovers new paths. Due to the excellent performance, we can leverage AFL to achieve high coverage on target programs. We specially tested the SPFuzz framework in content mutation mode, which is using the full-strategy mutation of AFL. The result will be explained in Section VII. In addition, there are dependencies between some header fields and content fields, such as length and type fields. The header fields should be updated accordingly after renewed content data.

C. SEQUENCE MUTATION STRATEGY

In order to maintain the structural soundness of messages, we mess up the order of message sequence and this can also improve coverage. The implementation of sequence mutation strategy is inserting the currently selected message into a random position of the correct message sequence to change the direction of state transition.

VI. SPFUZZ SCHEDULING FRAMEWORK IMPLEMENTATION

Fig.6 shows the components of the SPFuzz framework and its workflow. The most important component is the core scheduler which is responsible for global scheduling. The core scheduler communicates with the AFL engine and the target server, and receives description files from the interpreting part. The relationship is illustrated in Fig.2.

A. MUTATION SCHEDULER

The mutation scheduler comprises three schedulers for mutation according to the three-level mutation strategy. Basically, the head mutation scheduler saves values produced by the head mutation strategy (Table 1) into dictionaries. The content mutation scheduler produces test cases got from the AFL engine, and the sequence mutation scheduler distorts the correct message sequence.

The mutation scheduler will select a mutation strategy and a message to be fuzzed randomly, and the random choices are taken with message weight and strategy weight. The message weigh is defined in section IV, and the **strategy weight** indicates which strategy is easier to be selected. After that, it starts the corresponding scheduler for fuzzing.

B. FORMAT HANDLER

The format handler is used to rearrange the format of test cases or heads after mutation. It is necessary to stitch header fields with content fields and make them a integrate test-case, because the mutation scheduler mutates content or head separately.

C. DEPENDENCY HANDLER

The dependency handler deals with dependencies when it receives a complete test case. The handler replaces the header field that violates dependencies with correct data.

Dependencies within message are handled by analyzing whether the relevance of fields is broken, e.g. whether the length of content is updated with a changed content data length. Dependencies between two messages are handled when current test case is depend on previous messages. The handler reads data from a cache, which stores previous messages temporarily to catch relevant data.

D. COMMUNICATION SCHEDULER

The communication scheduler consists of a socket and a traffic state monitor. It is a bridge to send and receive messages from the target server. The scheduler forks a subprocess for AFL continuously that provides test cases which are transmitted back to the mutation scheduler in a pipe way. After being mutated and being formatted, these messages are scheduled and sent to the target server. Then the target server returns response messages and traces of processes, which will be fed back to the mutation scheduler to generate more meaningful test cases. The traffic monitor always listens to the response messages from target. If the response message does not come

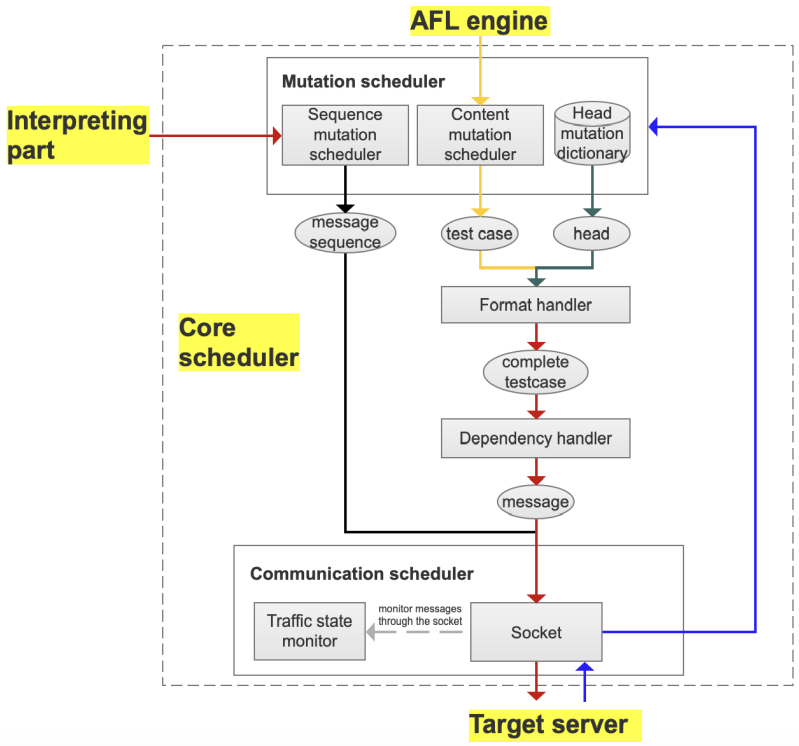


FIGURE 6. The components of SPFuzz framework and how the framework works.

back in time, the current fuzzing round will be interrupted and skipped into next round. This will save more time for for more valuable fuzzing.

VII. EVALUATION

There are two metrics to evaluate the performance of fuzzing tools. One is triggered software crashes, either from known or new vulnerabilities. The other is path coverage of target programs. The path coverage indicates how many paths are covered in target programs. In network protocol fuzzing, no previous work uses coverage-based method, and we take both coverage-based measurement and triggering crashes in this paper. We evaluate the coverage of two implementations of the FTP protocol and the TLS protocol in three granularities, i.e. **the function coverage**, **the basic block coverage** and **the edge coverage**. Besides, we reproduce the CVE-2015-0291 [20] vulnerability of OpenSSL to prove the validity and utility of SPFuzz.

A. EXPERIMENT SETUP

All of our experiments run on a computer with 8GB memory and 64-bit Ubuntu 16.04 LTS. The experiments are carried out on Proftpd 1.3.5 [25], Oftpd 0.3.7 [26] and OpenSSL 1.0.2 [22]. These three target servers are deployed on a lightweight Debian-i386 2.6.32-5-686 virtual machine.

The environment configuration steps are as follows:

1) Deploy the server of protocol on the Debian virtual machine and configure them.
2) Construct the message specification and state transition description file for producing test cases.
3) Start the server to accept the responses and record the traces.
4) Run the framework to fuzz the target.

B. EVALUATING THREE-GRANULARITY COVERAGE

We measure the coverage in three granularities: **the function coverage**, **the basic block coverage** and **the edge coverage**. The function coverage is the proportion of the actually executed functions driven by test cases over the total number of a program. This also pertains to basic blocks for the basic block coverage and edges between two basic blocks for the edge coverage.

In order for that, the static analysis is firstly applied to bin files to obtain the total number of functions, basic blocks and edges of a program with the help of IDA [31], and the dynamic analysis to get the addresses in the trace files, which determines the number of actually executed functions, basic blocks and edges, is adopted as well.

The FTP protocol experiment is carried upon Proftpd and Oftpd. For sake of fairness, we only use "USER", "PASS", "SYST", "PORT", "LIST", "CWD", "STOR", "RETR" and "QUIT" messages to produce test cases in this experiment, and both Boofuzz and SPFuzz initialize messeage sequence based only on the mentioned messages above for testing. The OpenSSL experiment chiefly focuses on handshake messages in TLS. Table 2 shows the experimental

TABLE 2. Three types of coverage produced by Boofuzz, SPFuzz/C and SPFuzz/F on Proftpd, Oftpd and OpenSSL (SPFuzz/C means SPFuzz in content mode with AFL and SPFuzz/F means SPFuzz in full mode with all mutation strategies).

		Proftpd	Oftpd	OpenSSL
Boofuzz	Function coverage	26.56%	28.23%	8.52%
	Basic block coverage	14.86%	26.55%	8.80%
	Edge coverage	7.13%	10.58%	2.05%
SPFuzz/C	Function coverage	29.46%	33.67%	14.13%
	Basic block coverage	17.03%	34.74%	10.67%
	Edge coverage	8.16%	13.73%	2.11%
SPFuzz/F	Function coverage	35.48%	58.50%	17.60%
	Basic block coverage	20.32%	54.20%	14.40%
	Edge coverage	9.00%	15.86%	2.38%

results of three fuzzing tools executing the same message sequence, where the SPFuzz/C indicates the SPFuzz in content mode, which only uses the content mutation strategy based on the AFL engine, and the SPFuzz/F indicates the SPFuzz in full mode with all mutation strategies.

We test three frameworks of three-granularity for two hours, and compare the test results. Since the calculation is based on bin file of target server and not all message types are covered in the test, all coverages keep a relatively low level, while the coverage values of Boofuzz are even lower than the others. The SPFuzz/C leverages AFL and improves the coverage by generating random test cases. However, some paths are still not covered without head mutation and sequence mutation. This is proved by the SPFuzz/F whose function coverage increases to 35.48%, basic block coverage to 20.32% and edge coverage to 9.00% on Proftpd. Meanwhile, the three types of coverage reach 58.5% and 54.20% and 15.58% on Oftpd. The results highlight that the SPFuzz/F has better ability to find more paths on two implementations of FTP. Also, on OpenSSL, SPFuzz/F outperforms Boofuzz by 17.60%, 14.40% and 2.38% in three-granularity coverages.

As the fine-grained edge coverage blends contextual information, we further compare and analyze its change of three server implementations over time, which are shown in Fig. 7, Fig. 8 and Fig. 9.

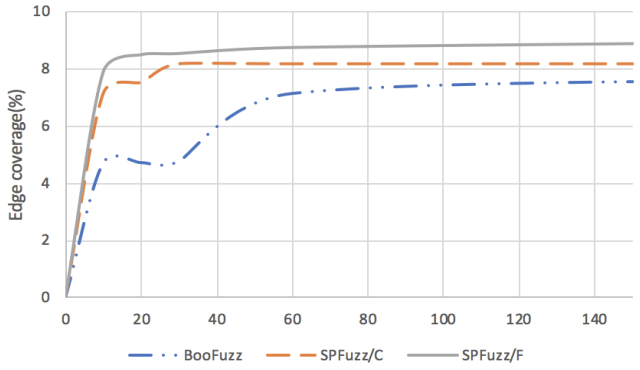


FIGURE 7. The change of edge coverage of Proftpd over time (the horizontal axis indicates time in minutes).

In the Proftpd experiment as shown in Fig. 7, the edge coverage of Boofuzz holds at around 5% in the first 30 minutes, then the value jumps to 7%. In contrast,

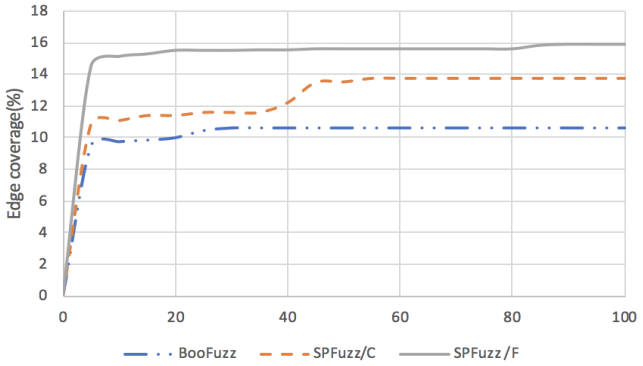


FIGURE 8. The change of edge coverage of Oftpd over time (the horizontal axis indicates time in minutes).

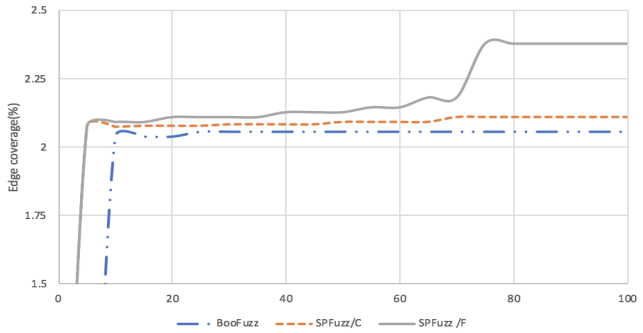


FIGURE 9. The change of edge coverage of OpenSSL over time (the horizontal axis indicates time in minutes).

SPFuzz/C surpasses 8% in 30 minutes due to the randomness of AFL, and SPFuzz/F reaches 9.00%.

In the experiment of Oftpd as shown in Fig. 8, the coverage of Boofuzz keeps at 10.58% after 30 minutes, while the coverage of SPFuzz/C stays at 13.73% after 40 minutes, and the SPFuzz/F strides over 15% rapidly and rises steadily.

Because of the complexity of OpenSSL, all types of coverage of three tools are very low as shown in Fig. 9. Edge coverage of Boofuzz stays at 2.05%. The coverage of SPFuzz/C is 2.11% and the coverage of SPFuzz/F is 2.38%. The jump appears around 70 minutes which indicates some mutations cover serveral new paths.

The mechanism of the Boofuzz framework is to mutate the constructed message sequence according to the mutation

dictionary inside the framework. Therefore, the fuzzing process has to spend a lot of time on traversing the dictionary, and then enters next round of new message. This is monotonous mutation always tries to establish the preamble states covering the same paths and thus is unlikely to find more interesting paths. SPFuzz/C leverages AFL to mutate the content fields of messages. Since AFL cannot maintain the states of protocol fuzzing, we reconstruct the preamble states based on the specific message sequence, and generate the test cases. However, simply mutating the content field content without any knowledge of the protocol leads to numerous meaningless test cases. The SPFuzz/F with three mutation strategies and the ability of handling dependencies overcomes the above drawbacks and achieves better results than the other two.

C. TRIGGERING CVE-2015-0291

1) CVE-2015-0291

The transfer layer security (TLS) [21] protects the security and integrity of sessions in network. Building a handshaking process through exchanging keys and certifications, encrypting data and transferring in application layer is considered secure. The OpenSSL is an implementation of TLS and the CVE-2015-0291 [20] is detected in its 1.0.2 version. This vulnerability causes a denial of service by replacing an invalid signature algorithms extension in the ''ClientHello'' message during a re-negotiation.

2) PROCESS OF TRIGGERING THE VULNERABILITY

We leverage the SPFuzz framework to reproduce CVE-2015-0291 by monitoring handshake protocol in TLS with the help of both the content and the sequence mutation strategies. We describe the specification of handshake messages and select the fuzzable field such as ''client_random'', ''premaster_string'' and ''signature_algorithms''.

During handshaking, the client and the server generate random values which is used to produce the content of subsequent messages. All messages in the handshaking process are required to compute their hash values. The length field of corresponding fields has to keep updating and it is difficult to handle the complex intra- and inter-message dependencies. We overcome these challenges by describing these dependencies in our language, caching the message during handshaking and leveraging the existing library for encryption in TLS. Finally, CVE-2015-0291 is triggered which proves the validity and utility of the SPFuzz framework. However, compared with SPFuzz, BooFuzz fails to trigger the CVE-2015-0291 because it doesn't have ability to handle the dependencies in protocol.

VIII. LIMITATION AND FUTURE WORK

Although the SPFuzz improves program coverage when compared to existing tools, there are some limitations as well as future improvements.

A. MESSAGE SPECIFICATIONS CANNOT COVER ALL SITUATIONS

Currently, the protocol specification description file, the protocol state transition description file and the states construction from RFC have to be processed by hand. This is labor-intensive and is not scalable. It is quite appreciable to make this step automatic and intelligent, e.g. extracting protocol specifications from RFC directly and ensuring the correctness at the same time.

B. APPLICATION IN COMPLEX PROTOCOLS

It is difficult to fuzz proprietary protocols and other complex protocols without protocol specifications and source code. Focusing on these protocols and designing an universal method are the directions for further efforts.

C. ENCRYPTION ISSUES IN THE PROTOCOL

Although the SPFuzz framework triggers a known OpenSSL vulnerability, the encryption problem still challenges stateful protocol fuzzing unless we have deep knowledge of complex encryption algorithms used in protocols.

IX. CONCLUSION

In summary, this paper tries to conquer the challenges of fuzzing stateful protocols which are maintaining the states, handling dependencies, and improving fuzzing coverage. To achieve it, a description language is defined for not only describing the specification of protocol, the state transition of protocol, and dependencies to generate fuzzing test cases and maintain the session states, but also handling intra- and inter-message dependencies. Besides, the three-level strategy (head, content and sequence), and assigning weights to messages and strategies randomly improve the coverage of protocol fuzzing greatly. This helps explore more deeper paths.

Based on the mentioned above, the SPFuzz framework is designed. It schedules the three-level mutation strategy and generates high coverage test cases combining with AFL. The framework can also communicate with AFL and the target server, transmit test cases, collect traces generated by target processes and feed them back to AFL. In experiments, the SPFuzz framework outperforms the existing stateful protocol fuzzing tool Boofuzz by an average of 69.12% in three granularities coverage tests on two implementations of FTP, i.e. Proftpd, Oftpd and OpenSSL. It further triggers CVE-2015-0291 to show the validity and utility.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their insightful comments that helped them improve their work.

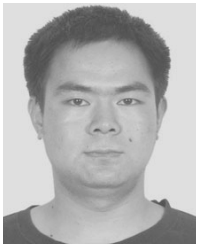
REFERENCES

[1] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, ''Pulsar: Stateful black-box fuzzing of proprietary network protocols,'' in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Dallas, TX, USA: Springer, 2015, pp. 330–347.

[2] T. L. Munea, H. Lim, and T. Shon, "Network protocol fuzz testing for information systems and applications: A survey and taxonomy," *Multimedia Tools Appl.*, vol. 75, no. 22, pp. 14745–14757, 2016.
[3] J. Chen *et al.*, "Iotfuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018. doi: 10.14722/ndss.2018.23166.
[4] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Secur. Privacy*, vol. 3, no. 2, pp. 58–62, Mar./Apr. 2005.
[5] G. Shu, Y. Hsu, and D. Lee, "Detecting communication protocol security flaws by formal fuzz testing and machine learning," in *Proc. Int. Conf. Formal Techn. Netw. Distrib. Syst.* Paris, France: Springer, 2008, pp. 299–304.
[6] T. Novickis, E. Poll, and K. Altan, "Protocol state fuzzing of an OpenVPN," M.S. thesis, Fac. Sci. Master Kerckhoffs Comput. Secur., Radboud Univ. Nijmegen, Nijmegen, The Netherlands, 2016.
[7] M. Eddington. *Peach Fuzzer*. Accessed: Sep. 1, 2017. [Online]. Available: https://www.peach.tech
[8] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a stateful network protocol fuzZEr," in *Proc. Int. Conf. Inf. Secur.* Hangzhou, China: Springer, 2006, pp. 343–358.
[9] J. Pereyda. *Boofuzz: Network Protocol Fuzzing for Humans*. Accessed: Feb. 17, 2017. [Online]. Available: https://boofuzz.readthedocs.io/en/latest/
[10] T. Kitagawa, M. Hanaoka, and K. Kono, "AspFuzz: A state-aware protocol fuzzer based on application-layer protocols," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2010, pp. 202–208.
[11] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," in *Proc. IJCSNS*, vol. 10, no. 8, 2010, p. 239.
[12] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proc. USENIX Secur. Symp.*, 2015, pp. 193–206.
[13] X. Han, Q. Wen, and Z. Zhang, "A mutation-based fuzz testing approach for network protocol vulnerability detection," in *Proc. 2nd Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, Dec. 2012, pp. 1018–1022.
[14] G. Shu, Y. Hsu, and D. Lee, "Detecting communication protocol security flaws by formal fuzz testing and machine learning," in *Proc. Int. Conf. Formal Techn. Netw. Distrib. Syst.* Tokyo, Japan: Springer, 2008, pp. 299–304.
[15] B. Blumbergs and R. Vaarandi, "Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis," in *Proc. Military Commun. Conf. (MILCOM)*, Oct. 2017, pp. 707–712.
[16] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 579–594.
[17] M. Zalewski. *American Fuzzy Lop*. Accessed: Jun. 1, 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/
[18] P. Amini and A. Portnoy. *Sulley Fuzzing Framework*. Accessed: Jun. 12, 2014. [Online]. Available: https://github.com/openrce/sulley
[19] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2017, pp. 1–14.
[20] *CVE-2015-0291*. Accessed: Mar. 19, 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0291
[21] *TLS Protocol*. Accessed: Nov. 4, 2018. [Online]. Available: https://zh.wikipedia.org/wiki/TLS
[22] *OpenSSL 1.0.2*. Accessed: Nov. 15, 2016. [Online]. Available: https://www.openssl.org
[23] J. Postel and J. Reynolds. *File Transfer Protocol (FTP)*. Accessed: Mar. 6, 1985. [Online]. Available: https://www.rfc-editor.org/info/rfc959
[24] E. Rescorla. *Transport Layer Security (TLS) 1.3*. Accessed: Aug. 18, 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446
[25] *ProFTPD*. Accessed: Apr. 9, 2017. [Online]. Available: http://www.proftpd.org
[26] *Oftpd*. Accessed: Mar. 26, 2004. [Online]. Available: http://freshmeat.sourceforge.net/projects/oftpd
[27] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.
[28] Y. Luo, P. Wang, X. Zhou, and K. Lu, "DFTinker: Detecting and fixing double-fetch bugs in an automated way," in *Proc. Int. Conf. Wireless Algorithms, Syst., Appl.* Tianjin, China: Springer, 2018, pp. 780–785.
[29] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Comput. Secur.*, vol. 75, pp. 118–137, Jun. 2018.
[30] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.
[31] *Interactive DisAssembler (IDA)*. Accessed: Apr. 14, 2007. [Online]. Available: https://www.hex-rays.com
[32] *Extensible Markup Language (XML)*. Accessed: Mar. 1, 2014. [Online]. Available: https://en.wikipedia.org/wiki/XML
[33] *Hypertext Transfer Protocol (HTTP)*. Accessed: Jun. 1, 1999. [Online]. Available: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[34] *State (Computer Science)*. Accessed: Jun. 1, 1999. [Online]. Available: https://en.wikipedia.org/wiki/State_(computer_science)
[35] *Requests for Comment (RFC)*. Accessed: Feb. 5, 2004. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Requests_for_comment
[36] *Internet Key Exchange(IKE)*. Accessed: Sep. 26, 2006. [Online]. Available: https://en.wikipedia.org/wiki/Internet_Key_Exchange
[37] L. Kai, P.-F. Wang, G. Li, and Z. Xu, "Untrusted hardware causes double-fetch problems in the I/O memory," *J. Comput. Sci. Technol.*, vol. 33, no. 3, pp. 587–602, 2018.

CONGXI SONG was born in Beijing, China, in 1994. She received the B.E. degree in computer science and technology from Beihang University, Beijing, China, in 2017. She is currently pursuing the M.E. degree in computer science and technology with the National University of Defense Technology, Changsha, China.

Her research interests include software analysis, fuzzing, and symbolic execution.



BO YU was born in Hunan, China, in 1985. He received the M.S. and Ph.D. degrees from the National University of Defense Technology, in 2010 and 2013, respectively, where he is currently a Researcher.

His research interests include information security and network security.

XU ZHOU was born in Shanxi, China, in 1985. He received the Ph.D. degree, majoring in computer science and parallel, from the National University of Defense Technology, Changsha, China, in 2013, where he is currently an Assistant Researcher with the College of Computer.

He has authored papers on PPoPP and several top transactions on parallel. His research interests are computer system and parallel.

QIANG YANG was born in Henan, China, in 1983. He received the Ph.D. degree in computer science from the University of Amsterdam, The Netherlands, in 2014.

He is currently an Assistant Researcher with the National University of Defense Technology. His research interest includes security of network devices and reverse engineering.