Wireless Pers Commun
DOI 10.1007/s11277-016-3322-9

Design and Implementation of Fuzzing Framework Based on IoT Applications

Tewodros Legesse Munea[1] · **I. Luk Kim**[2] · **Taeshik Shon**[1]

Abstract Nowadays the most serious security problems are imperfection in the implementations of network protocols. This imperfection can bring a lot of vulnerabilities such as could allow malicious user to attack the systems remotely using the network protocols over the internet. That is why developers value software security phases involving review of code, risk analysis, testing with penetration, and Fuzzing. In case of Fuzz testing, the main aim is to find vulnerabilities in the software/application by sending inputs which are not expected to the target. Then they monitor the situation of the target. Many applications in Internet of things (IoT) (http://en.wikipedia.org/wiki/Internet_of_Things) environments are working with File Transfer Protocol (FTP) based applications. In this study, we present a fuzzing framework, which is applied to test network protocol implementations. It is extendable, man-in-the-middle, smart, and mostly deterministic. Our tool, like AutoFuzz (Gorbunov and Rosenbloom in AutoFuzz: automated network protocol fuzzing framework, Department of Mathematical and Computation Sciences, University of Toronto Mississauga, Canada L5L 1C6, 2010), has the ability to learn a given protocol implementation by building a finite state automaton from records of communication traces between a client and the server. Additionally, this tool has the ability to learn syntax of individual messages at a lower level using the techniques of bioinformatics (Beddoe in Network protocol analysis using bioinformatics algorithms, http://www.4tphi.net/~awalters/PI/pi.pdf). At last, this framework can fuzz a given server protocol specification by changing the communication traces between the server and client. We applied it to multiple implementations of FTP server, with result of finding new and known vulnerabilities.

✉ Taeshik Shon
tsshon@ajou.ac.kr

Tewodros Legesse Munea
teddylegessemunea@yahoo.com

I. Luk Kim
kim1634@purdue.edu

[1] Ajou University, Yeongtong-gu, Suwon-City, Gyeonggi-do 443-749, South Korea

[2] Department of Computer Science, Purdue University, West Lafayette, IA 47907, USA

Published online: 26 April 2016

Keywords Fuzzing · Fuzz-testing · Network protocol fuzzing · Fuzz testing framework on FTP · Fuzzing framework based on IoT applications

1 Introduction

We have been introduced to the word fuzzing (Fuzz testing) for around 20 years but recently it has got attention all over the world. Fuzz-testing is a process of sending semi-valid or malformed data to an application/program on purpose in the intention of finding vulnerabilities, or errors in the target program. Fuzz testing usually gives focus on identifying bugs that can be exploited to permit any malicious user to run their own code.

Fuzzing is a kind of dynamic vulnerability detecting technology; present advanced fuzzing techniques can be divided into two categories. One is white box fuzzing, which combines with static analysis, symbolic execution, concrete execution and other white box testing techniques to produce high code coverage fuzzing. The other is knowledge-based fuzzing, which generates test cases based on format analysis. Outstanding tools such as SPIKE, Peach, and Sulley belong to this class [1].

Fuzzing is a testing technique in software that helps to exploit the number of vulnerabilities in software systems. Semi-random or random inputs are prepared to targeted programs using Fuzz testing. Because the inputs are random, they have high possibility of being incorrect and unexpected inputs to the target programs. The target program will crash or hang during fuzz testing if it does not reject those incorrect inputs. Software systems with no ability to survive fuzz testing will have a high probability of having security flaws because most critical security flaws most of the time lie in insufficient or incorrect program inputs checking. We call Fuzz testing as very quick and costly effective in finding vulnerabilities of the system.

A Fuzzer (Fuzz testing tool) is a tool in a purpose of finding implementation flaws from unusual inputs sent to the target implementation in the aim of generating unexpected behavior. We can classify the network protocol Fuzzer as 'Smart' Fuzzer or 'Dumb' Fuzzer based on the knowledge it have to the target network protocol implementation. A 'Dumb' Fuzzer in general sends inputs which are random to its target. It doesn't understand the target protocol implementation's communication. Since 'Dumb' Fuzzers have been easy in developing, they are applicable to any given protocols servers or clients immediately. The problem with 'Dumb' fuzzing is that they are less effective, around 50 %, from 'Smart' fuzzing when measured [2]. We can take ProxyFuzz [3] as an example of a 'Dumb' Fuzzer. ProxyFuzz is a non-deterministic but man-in-the-middle network protocol Fuzzer. Since it is non-deterministic, it modifies the network traffic [3] randomly between a connected server and client. Secondly, we have 'Smart' Fuzzers, which have a pre-programmed knowledge of the target protocol implementation they are going to fuzz. They distinctly understand the state machine of the protocol, syntax of the messages and specific fields with their types. Using this knowledge, they efficiently fuzz target implementation code deeply. We have Peach as an example for "Smart" Fuzzers. The main disadvantage of "Smart" Fuzzers is they are highly reliable on the availability of documents of protocol specification. Additionally, "Smart" Fuzzers need adaptation manually to make them suitable for every new protocol implementation they are to apply to. Therefore, to apply them to new protocols implementations is tedious and labor intensive.

Fuzzed inputs also called unexpected or abnormal inputs or semi-valid inputs, which are generated from opponent end-point, are fed into the system under test (SUT) and then we monitor the behavior of the SUT for the given input. If there is crash of SUT, we say there is a vulnerability of the software. Network protocol Fuzzers can create invalid inputs in two ways: generating inputs for the target by understanding the protocol specification or mutating the inputs taken from the opponent end-point. Both ways have their advantage and disadvantage and some Fuzzers use both of them to get most vulnerability from the target network protocol [T].

Additionally, some network protocol fuzzing tools store previous conversation between an opponent end-point and SUT to improve the protocol Fuzzer to become much better effective in finding vulnerabilities. Sometimes it is good to consider what happened to some target network protocol can work for others or taking the mutated inputs of previous sessions and apply it for current target [T].

On the other hand, most network protocol fuzzers use ASCII characters in their input creation method in finding vulnerabilities and this has been effective in the process. This is because the inputs we use for the conversation between client (opponent end-point) and server (SUT) are in ASCII format for every request from clients such that request for login. Some researchers have started using Unicode characters especially UTF-8 to find more vulnerabilities. UTF-8 (U from Universal Character Set + Transformation Format—8-bit [4]) is a way of encoding characters in Unicode with the ability to encode all the possible characters which we call them code points. This encoding uses code units with 8-bit and it is variable-length. As other applications, its ASCII compatibility was considered in designing as a backward compatibility. When we consider dominancy for World Wide Web in encoding of characters, UTF-8 has took control. When we count, we found 83.3 % of all Web pages in March 2015 (with most popular East Asian encoding, GB 2312, at 1.3 %). The W3C recommends UTF-8 as default encoding in their main standards (XML and HTML) [4].

In this paper, we consider real-world environments, with a Fuzz testing tool which is smart, mostly deterministic, and man-in-the-middle for a given protocol implementation of network. Additionally, we consider FTP [5] protocol specifications, implementations and the communication between the server and client. This helps us to understand security threats in IoT [6] because many applications in IoT environments are working with FTP based applications. The rest of this paper is organized as follows. In Sect. 2, we briefly discuss the related works. Then, we have Sect. 3 which we discuss about our proposed system's framework and its components. Section 4 describes the steps we follow in the framework for Fuzzing. In the following Sect. 5, we briefly describe constructing message group order (MGO). Section 6 presents the simulation and evaluation of the framework with other researches. In the last section we conclude our work and put our future work.

2 Related Works

Several researches have done to understand the communication format and specification of a given network protocol automatically. We start from "Network Protocol analysis Using Bioinformatics Algorithms" [7]. Here the authors provide a method to utilize algorithms of the bioinformatics field to automate network protocol reverse engineering. They try to align sample messages using multiple string alignment algorithms to determine individual fields in the protocol packets. After that, the aligned messages' consensus sequences are

being examined to have understanding of the start and end of each individual fields of the message in the given packet. They provided an open-source tool which can be used in a collected protocol messages to determine message.

Next, we see "A Model-based Approach to Security Flaw detection of Network Protocol Implementations" [8], they use a technique based on synthesizing an abstract behavioral model of a protocol implementation to extract the specification of the protocol automatically. First they record the conversation between an opponent end-point and SUT, from this conversation the behavioral model is understood as a finite state automaton (FSA). The FSA briefly expresses the transitions and key states of implementation of the protocol and this could be a guide in detecting process of the flaw in a systematic way.

They [8] proposed a model-based algorithm, which is based on passive synthesis with partial FSA reduction, for synthesizing an abstract behavioral model of a protocol implementation to guide input selection. For collected network traces between opponent end-point and SUT, the algorithm constructs and also minimizes a FSA which highly based on a function called abstraction function. If we are given conversation and are required to map of similar messages with a unique abstract representation, we use an abstraction function because it is a simple function.

Additionally we have "Prospex: Protocol Specification Extraction" [9] which the authors first determine message types and then construct the network protocol's FSA because they are focused on automating the protocol specification extraction. Even if their final application/system can be applied for protocol specifications extraction, their FSA construction technique is totally different from the techniques mentioned in [8]. As far as we know, we could not find both [9] and [8] systems to do more research.

Lastly but not least we have "AutoFuzz—Automated network Protocol Fuzzing Framework" [10] which our work is highly based on and it is an open source fuzzing framework for further research and development. This [10] fuzzing framework acts as man-in-the-middle, it is a smart, and designed to fuzz the SUT of the network protocol implementation even though it can be equally applied to fuzz the client side effectively. First of all they denote 'input messages' for all messages originated from the opponent end-point (client) side to the SUT (server) and 'output messages' for messages originated from the server side to the client side. This [10] framework first records the conversation between server and client by acting as a man-in-the-middle, and then from this conversation it automatically extracts specifications of the network protocol implementation.

Both [8] and [10] use the same way in constructing a FSA in capturing the sample conversation as mentioned above which helps them to understand the network protocol implementation at a high level. But, in AutoFuzz [10], they incorporated the necessary abstraction functions which can be extended to highly perceive given network protocol specification. Additionally they [10] can obtain individual messages' fields by using techniques mentioned in [7]. This means, their [10] technique has the ability to understand in detail the network protocol specification at a lower level by obtaining the type and length information of the non-constant data fields of individual messages.

As a matter of fact, AutoFuzz [10] has the technique to store and associate the messages' syntax information from a recorded sample conversation, which is called generic message sequence (GMS). This GMS is a sample format for a message that used to distinguish constant data fields from non-constant data fields and to obsolete the non-constant data fields with length and type information. Because AutoFuzz [10] has the GMS which is used in eliminating the requirement for fuzzing functions which are protocol specific as needed in [8]. Then in place of individual messages GMS

representations would be carried out using Fuzzing functions and this is based on the derived length or type information of the constant or non-constant data fields. The good thing about AutoFuzz is that additional novel fuzzing functions can be extended. As a conclusion, their [10] fuzzer can be implemented to fuzz in both server and client network protocol implementation which they successfully found new and existing vulnerabilities after applying it to three File Transfer Protocol (FTP) servers. This is because their fuzzer acts as a man-in-the-middle and uses already built FSA for vulnerability detection process.

3 Proposed System

As clearly described in Sect. 2, we need to have a fuzzer which has to be smart, can act as a man-in-the-middle with components need to understand the network protocol implementation at both higher level and lower level, and then can fuzz the target protocol implementation. Let's start by discussing the basic elements of this framework which are JAVA SOCKS Server, Proxy Server, Extractor of Protocol Specification, Fuzzing Functions, Fuzzing Engine, and the framework's User Interface.

3.1 Java SOCKS Server

SOCKS (Socket Secure) is an Internet protocol which routes or forwards network data packets in between a server and client through a proxy server. Java SOCKS Server supports both SOCKS4 and SOCKS5 protocols which is entirely written in Java. SOCKS5 differ from SOCKS4 because it gives support for authentication so that authorized users can only access the server [11]. So, our framework implements them and works as SOCKS Server between an opponent end-point and the SUT. It is freely available software with both binary and source code with anyone can modify and distribute it [12]. It is designed to be easily expandable to support different encryption/authentication/authorization methods.

3.2 Proxy Server

The same as Java SOCKS Server, our framework participates as a proxy server in a server and client. Because of this, it records and changes the traffic of application level to obtain target protocol's specifications with the help of Proxy Server and perform fuzzing operations. This proxy server highly depends on the JAVA Socks server. We modified it so that it would let direct manipulation of the application level traffic. Figure 1 describes the architecture of our framework's Proxy model.

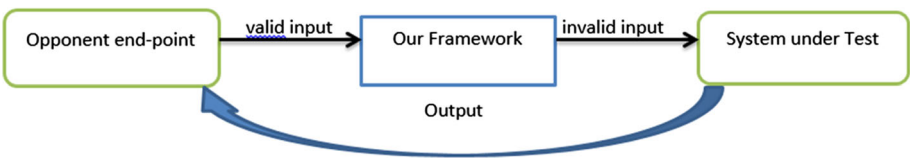

Fig. 1 Proxy server model architecture of our framework

3.3 Extractor of Protocol Specifications

The sample communication session between an opponent end-point and the SUT is given to the extractor of protocol specifications which is used in extracting the FSA of the target network protocol implementation. The framework needs to import appropriate input/output abstraction functions in order to understand and implement any application level protocol implementation. Additionally, using the algorithm mentioned in the Sect. 5, the framework has the ability to extract MGO to understand the individual messages' syntax at lower level.

3.4 Fuzzing Functions

The fuzzing functions set at the present time contain both deterministic and non-deterministic functions. The pre-programed data are inserted into the MGO in deterministic functions. These data can be large amount of strings, big/small integer values and also others. Whereas when we come to non-deterministic functions they randomly skip constant or non-constant data fields of the MGOs. But they suddenly make transitions in the FSA and insert random data into the MGOs.

The framework highly uses format violations in order to discover potential vulnerabilities after understanding the target protocol implementation at both higher and lower level using deterministic functions. It deliberately violates the format or syntax specification of the target program. Let's introduce some of the different kinds of Fuzz testing methods or functions in this framework using real examples of FTP protocols [13].

Insert large string If a message communication has a length limitation in the protocol specification, the framework generates an invalid message that has a string longer than the limitation defined in the protocol specification length. For instance, USER command in FTP is followed by up to 255 Alpha numeric characters; the framework produces messages that have over 255 characters.

Break character restriction rule If a message in a communication command has restriction of characters acceptance, we provide an invalid message which contains the restricted characters. For example, all messages in FTP communication should be printable ASCII characters; the framework purposely provides a message that contains non-printable ASCII characters.

Having format string Nowadays having the vulnerability of format string are common in the software of text-based protocols. The framework uses this chance to generate an invalid message that contains a format string such as "%s" and "%d".

Use Non-ASCII character Most of FTP protocol implementations are based on ASCII characters as their communication, the framework provides an invalid message which contains characters which are not in ASCII.

3.5 Fuzzing Engine

We understood the target network using FSA and MGO; we have fuzzing functions to modify the communication; then using the fuzzing engine we can change the communication traffic between an opponent end-point and the SUT. We can upgrade the fuzzing engine with new fuzzing functions. We have log files which are used to register every detail in the fuzzing process so that this helps us to find the changes and states of the communication between fuzzing actions. Because of this, we can identify which exact

input caused the unexpected output of the application or the target program in the communication. Further explanation on how we modify the traffic communication between the opponent end-point and the SUT is explained latter.

3.6 User Interface of Our Framework

This user interface lets testers interact without difficult with the fuzz testing tool and helps in control its actions. The framework is basically developed by the JAVA Swing library [14] using eclipse. To visualize a protocol FSA, we use JUNG graphing library which is free to use [15].

4 Work Flow Steps

In this section, we present the phases used in the framework to fuzz given target protocol implementation which is presented in Fig. 2.

The flow steps used in the framework is described here. As depicted in Fig. 2, the first step in the process is using the framework's made Proxy Server which is mentioned above to gather a large amount of network protocol implementation traces. We can manually edit, import and export the collected traces at any of time. At the second step, we build and minimize the target Protocol's behavior model using the reduction method mentioned in [8] which is called the passive synthesis with partial FSA. At this step we understand the target protocol implementation at a higher level. At the third step, we want to understand the target protocol at a lower level by building MGO by extracting each and every message's syntax. To build MGO we need to create input messages' clusters. So we modified use of abstraction function method mentioned in [8]. This means, similar input messages are collected in their own cluster. In the next section we present the algorithm of MGO construction with example. In a direct approach, if you have input messages and they are given as input for abstraction function, then we expect the abstraction function to put similar input messages together and cluster them. After that, we need to generate MGO for the created clusters using sequence alignment algorithms. At last, we go through the target protocol's FSA and relate each transition with the right MGO.

Fig. 2 Fuzz testing phases used by our framework

5 Constructing Message Group Order

Message Group Order is an instance or image of a message that is used to separate constant data fields from non-constant data fields and to attach non-constant data fields with length and type information. In this section, we propose an algorithm which is used to obtain MGOs. Let's start by defining a cluster as a collection of similar messages. The construction of MGO is depicted as flowchart in Fig. 3. In phase I: by applying the novel clustering technique, we gather similar messages to cluster. In phase II: On each cluster we apply the multiple sequence alignment algorithm specified in [7]. Then we have phase III: here for each clusters MGO is constructed. In phase IV: for every transition in the protocol's FSA we associate it with the corresponding MGO.

5.1 Phase I

At start, we used a novel technique proposed by [8] to cluster similar messages. Notice that, we call 'input messages' for messages originating from the opponent end-point side to the SUT and 'output messages' for messages originating from the server side to the client side.

Assume for a given input messages we have an abstraction function called *ftp_input*. Let's define a set for given input messages as $M = \{m_1, m_2, m_3, \ldots, m_n\}$. Using the *ftp_input* function, the algorithm mentioned in [10] returns similar input messages' clusters. To put as equation, for every $1 \leq i \leq n$, define C_i as:

$$C_i = \{M_z \in \{M_1, M_2, M_3, \ldots, M_n\} | ftp\_input(M_i) = ftp\_input(M_z)\} \quad (1)$$

This algorithm in (1) returns $\{C_i | 1 \leq i \leq n\}$.

Now let's consider the following input messages/commands in FTP conversation. Assume set

$M = \{$"*open* 127.0.0.1", "*user admin C: rwd*", "*help mdelete*", "*quit admin admin*", "*open* 169.20.6.223", "*user dummy*", "*help disconnect*", "*quit*", "*open* 202.30.24.92",

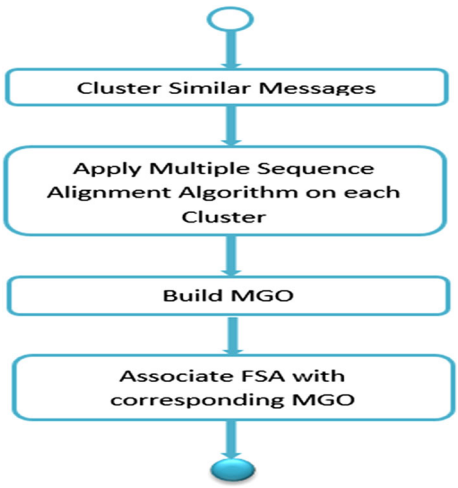

Fig. 3 Flowchart of message group order construction

"*user admin C: r*", "*help aaaaaaaaaaaaaaaaaa*", "*quit TeddyTeddyTeddyTeddy*", "*open* 215.80.0.47", "*user admin C: rw*", "*help remotehelp*", "*quit quit*"}

From the above given input message, say M_i *in* M and *ftp_input*(M_i) returns its first four characters. When we apply the mentioned algorithm on given example input message the function returns a set of four clusters namely $\{C_1, C_2, C_3, C_4\}$ where

$C_1 = \{$"*open* 127.0.0.1", "*open* 169.20.6.223", "*open* 202.30.24.92", "*open* 215.80.0.47"$\}$, $C_2 = \{$"*user admin C: rwd*", "*user dummy*", "*user admin C: r*", "*user admin C: rw*"$\}$, $C_3 = \{$"*help mdelete*", "*help disconnect*", "*help aaaaaaaaaaaaaaaaaaa*", "*help remotehelp*"$\}$, and $C_4 = \{$"*quit admin admin*", "*quit*", "*quit TeddyTeddyTeddyTeddy*", "*quit quit*"$\}$.

5.2 Phase II

The multiple sequence alignment algorithm which proposed in [7] is applied on every cluster once the input messages are clustered. Using this algorithm, list of aligned messages is returned for every cluster. Needleman–Wunsch algorithm [16] is applied to align the input messages based on the progressive alignment technique. Let's take one cluster, say C1, and put it in a format we get as pictured in Fig. 4.

If we apply the algorithm on the above cluster, we get four aligned input messages displayed in Fig. 5. So, after aligning the input messages, we have four input messages with identical length where "-" shows the gap between sequence.

5.3 Phase III

Once we align the input messages, we build MGOs for those clusters. If you consider the above Figures to implement, you will have array list of message blocks. So, MGO will be an array list where the block of the message corresponds to either a constant or non-constant data field.

To separate block of constant or non-constant, we need to recognize the start and the end of constant and non-constant data fields. Naturally, the algorithm checks for characters put at identical position throughout the aligned messages and, if all characters are found to be identical, it points that position as constant position in the resulting MGO, otherwise non-constant position or block. We call series constant and non-constant positions as

o	p	e	n		1	2	7	.	0	.	0	.	1			
o	p	e	n		1	6	9	.	2	0	.	6	.	2	2	3
o	p	e	n		2	0	2	.	3	0	.	2	4	.	9	2
o	p	e	n		2	1	5	.	8	0	.	0	.	4	7	

Fig. 4 FTP sample input message's cluster, C_1

o	p	e	n		1	2	7	.	-	0	.	-	0	.	-	-	1
o	p	e	n		1	6	9	.	2	0	.	-	6	.	2	2	3
o	p	e	n		2	0	2	.	3	0	.	2	4	.	-	9	2
o	p	e	n		2	1	5	.	8	0	.	-	0	.	-	4	7

Fig. 5 FTP sample aligned input message

o	p	e	n		β	β	β	.	β	0	.	β	β	.	β	β	β

Fig. 6 MGO for aligned FTP input message

o	p	e	n		£	£	£	.	£	0	.	£	£	.	£	£	£

Fig. 7 MGO for cluster 1 which aligned in Fig. 6

constant and non-constant data fields respectively. So, the algorithm returns with the following format

$$MGO = [g[1], g[2], g[3], \ldots, g[n]],\ where\ g[i]\ as\ the\ i's\ symbol\ in\ the\ MGO \tag{2}$$

We replace all mentioned unique characters using "β" instead of simple gap sequence. This means, non-constant data fields are represented using consecutive "β" in the resulting MGO. When we apply the mentioned algorithm on the FTP aligned input messages which are presented in Fig. 5, we get the following MGO displayed in Fig. 6.

Once we specified the fields as consecutive "β"s in the MGO for every non-constant data fields, we need to look in detail over each character on the given position from the aligned messages so that we can relate the type information. This means, we need to check to which set type do consecutive "β"s corresponds to. When we apply the type information for consecutive "β"s, we get our final MGO of the aligned messages as displayed in Fig. 7. The type information will be "£" for alpha-numeric non-constant data fields, "L" for long data fields, and so on.

5.4 Phase IV

At last, we go through the target Protocol's FSA, then we abstract every message at a transition and assign it to the corresponding MGO. This means, we got large network traces sample, from this sample we generated the protocol's FSA and every messages transition is assigned to specific MGO.

In the last step, after we make a change for live communication sessions between the opponent end-point and the SUT, we apply the fuzzing functions and these fuzzing functions are assigned by the fuzzing engine. To decide which fuzzing function should be applied for the live communication we need three criteria; the applied fuzzing function, input message, and the current state in the FSA. Additionally list of extendable fuzzing functions are loaded with the fuzzing engine, after the FSA and MGO associated with it. At start, the state of the fuzzing engine is set to the root of the target protocol's FSA. Then it follows up the traffic for any changes. If there is a change in the input traffic, then the fuzzing engine makes the necessary and appropriate change on the fuzzing functions.

6 Simulation and Evaluation

In this section, we present our environment setups, simulation results which we found existing and new vulnerabilities. We implemented the framework to get specification of the FTP. After that we applied it to fuzz some FTP server implementations. We provide an

overview of FTP in terms of command length for server and client conversations. We also discuss the characteristics of our decision using the selected FTP servers.

6.1 File Transfer Protocol

File Transfer Protocol is used for file exchange on the Transmission Control Protocol/ Internet Protocol networks. It is an application level protocol. Next, we will see how FTP implemented most commonly. At start, a client tries to connect to the server using port number 21, which is called the control port. Using this control port socket, the client requests are sent in addition to the login process in the form of ASCII. A new socket port will be opened, typically on port 20, with the server to transfer data when the client requests. This port is said to be data connection port.

Most FTP requests from client to the server are composed of a message type with a four letter following with the actual message. We have some exceptions with three letter message type commands like MKD, CWD, PWD, and RWD [17]. The responses of the server are also in ASCII format with a three digit followed by optional messages. The three digits are status codes about the response.

6.2 Simulation Setup

Phase 1 We have FTP server software like FileZilla FTP server [18], Open FTP server [19] which are installed and ready to implement fuzz testing. These different server implementations are: Open and compact FTP Server 1.2, FileZilla FTP Server 0.9.49, Wing FTP Server 3.6.1, [20] and IndiFTPD FTP Server. The same as AutoFuzz, because our framework included a proxy server, it acts as a proxy server between the server and client. Additionally we needed to redirect all Windows ftp.exe traffic, so that we installed proxifier [21] to redirect the traffic from client to the framework's proxy server. This helps us to trace the communication between the client and the server. But, here the connections of the client should be encapsulated in SOCKS5 sessions. Next, we installed FTP servers, we installed the proxifier to redirect the traffic to the framework's proxy server, and then in the next we started and run the framework so that it will start its proxy server.

Phase 2 Next we started performing the usual FTP requests using ftp.exe client after connecting manually. We tried to separate each session to a different network traces. We used different kinds of login credentials, created and deleted users, created changed and removed directories, uploaded and downloaded different kinds of files. We did sessions of 16 network traces for this simulation. Some of them are displayed in Fig. 8.

Phase 3 Using the abstraction functions mentioned in [8] which we import for the FTP server implementations, we abstracted all input messages which are originating from the client side to the server to the first four characters with exception to those messages starting with three word command (like MKD, CWD, PWD, and RWD) in which they are abstracted to their first three characters. But, the abstraction for the output messages will be to their first three characters. These messages are originating from the server side to the client.

Phase 4 In the next phase, we constructed the FSA which is corresponding to the network traces we recorded above, and displayed in Fig. 9. Additionally we minimized the FSA. Then we need to construct the MGOs and need to associate them with the correct FSA transitions as pictured in Fig. 10.

```
220  ******************************
User (127.0.0.1:(none)):  </Output>
<Input> admin C: rwd </Input>
<Output> 331 Password required for admin
Password: </Output>
<Input> gabriel </Input>
<Output> 230 User admin logged in </Output>
<Input> ls </Input>
<Output> 200 Port Command Successful.
150 Opening Binary mode connection for file list.
changelog.txt
ftpd.conf
ftpd.exe
list.tmp
main.cpp
226 Transfert Complete.
ftp: 56 bytes received in 0.06Seconds 0.97Kbytes/sec. </Output>
<Input> cwd </Input>
<Output> Invalid command. </Output>
<Input> help </Input>
<Output> Commands may be abbreviated.  Commands are:

!               delete          literal         prompt          send
?               debug           ls              put             status
append          dir             mdelete         pwd             trace
ascii           disconnect      mdir            quit            type
bell            get             mget            quote           user
binary          glob            mkdir           recv            verbose
bye             hash            mls             remotehelp
cd              help            mput            rename
close           lcd             open            rmdir </Output>
<Input> get changelog.txt </Input>
<Output> 200 Port Command Successful.
150 Opening binary mode data connection for "changelog.txt" (144 bytes).
226 Finished.
ftp: 144 bytes received in 0.00Seconds 144.00Kbytes/sec. </Output>
<Input> get list.tmp </Input>
<Output> 200 Port Command Successful.
150 Opening binary mode data connection for "list.tmp" (56 bytes).
226 Finished.
ftp: 56 bytes received in 0.00Seconds 56000.00Kbytes/sec. </Output>
<Input> get main.cpp </Input>
<Output> 200 Port Command Successful.
150 Opening binary mode data connection for "main.cpp" (23413 bytes).
226 Finished.
ftp: 23413 bytes received in 0.01Seconds 3344.71Kbytes/sec. </Output>
<Input> ls </Input>
```

Fig. 8 Some sessions of network traces

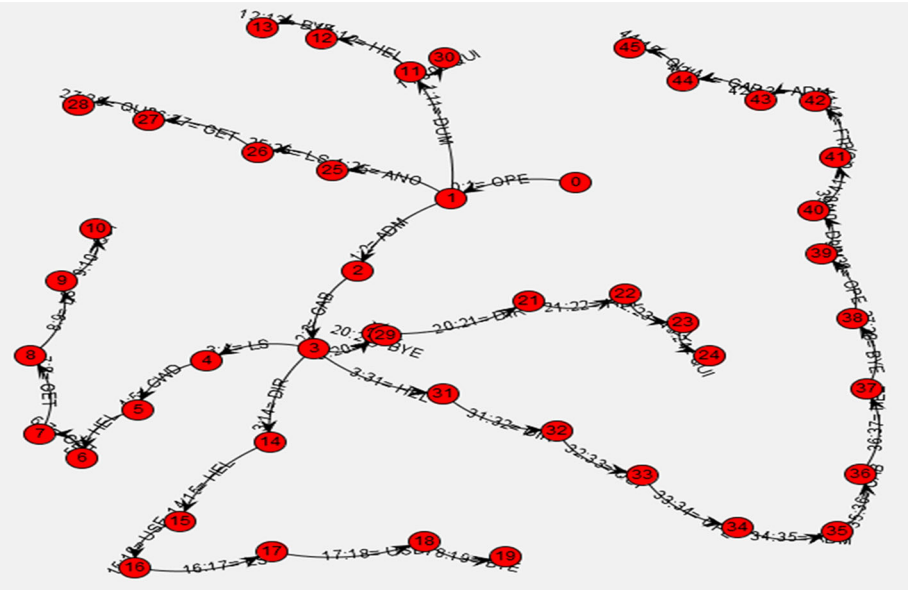

Fig. 9 User interface in building FSA

State ID	Input Sym.	Message Group Order
0	OPE	open AAAAAAAAAAAAA
1	ADM	adminAAAAAAAA
2	GAB	gabriel
3	LS	ls
4	CWD	cwd
5	HEL	helpAAAAAAAAAA
6	GET	get AAAAAAAAAAAAAAAA
7	GET	get AAAAAAAAAAAAAAAA
8	LS	ls
9	QUI	quitAAAAAAAAAA
3	DIR	dir
14	HEL	helpAAAAAAAAAA
15	USE	user AAAAAAAAAAAAAAAAAA C: rAAA
16	LS	ls
17	USE	user AAAAAAAAAAAAAAAAAA C: rAAA
18	BYE	AAAAAAAAAA
3	USE	user AAAAAAAAAAAAAAAAAA C: rAAA
20	DIR	dir
21	HEL	helpAAAAAAAAAA
22	HEL	helpAAAAAAAAAA
20	BYE	AAAAAAAAAA
3	HEL	helpAAAAAAAAAA
31	DIR	dir
32	QUI	quitAAAAAAAAAA
33	OPE	open AAAAAAAAAAAAA
34	ADM	adminAAAAAAAA
35	GAB	gabriel
36	HEL	helpAAAAAAAAAA
37	BYE	AAAAAAAAAA
38	OPE	open AAAAAAAAAAAAA
39	DUM	dummyAAAAAA
40	QUI	quitAAAAAAAAAA
41	FTP	ftp localhost
42	ADM	adminAAAAAAAA

Fig. 10 User interface after MGO is generated for the traces

Phase 5 After all this work we started the fuzzing engine. At last, we used a small FTP client written by AutoFuzz [10] which is written in JAVA. This helps to perform a lot of sessions and execute many requests with the server automatically.

6.3 Simulation Results

We have four FTP server implementations and we applied the framework to fuzz them automatically. Like AutoFuzz, we did find a few unexpected behavior instances sets of one server which is Open and Compact FTP Server 1.2. The first unexpected behavior instances set includes crashing of this server (Open and Compact FTP Server) by sending very long arbitrary strings before to the commands like USER, PASS, and PORT, and then sending String "\r\n" after or before authentication at any state of the server.

Unlike AutoFuzz, we could and did find unexpected behavior instances sets of Wing, IndiFTPD, Filezilla and Open and compact FTP Server. The unexpected behavior instances set include crashing of the above four different Servers by sending very long arbitrary different languages other than English word (non-ASCII characters) before to the commands like USER, OPEN, and PORT, and then sending String "\r\n" at any state of the server as pictured in Figs. 11 and 12.

The first one was already known to the public. The second unexpected behavior instances set, like AutoFuzz found, includes executing arbitrary commands on the server before the authentication and this attack was also known to the public even if the attack was more dangerous. While the third attack was denial of service attack and it is new to the public.

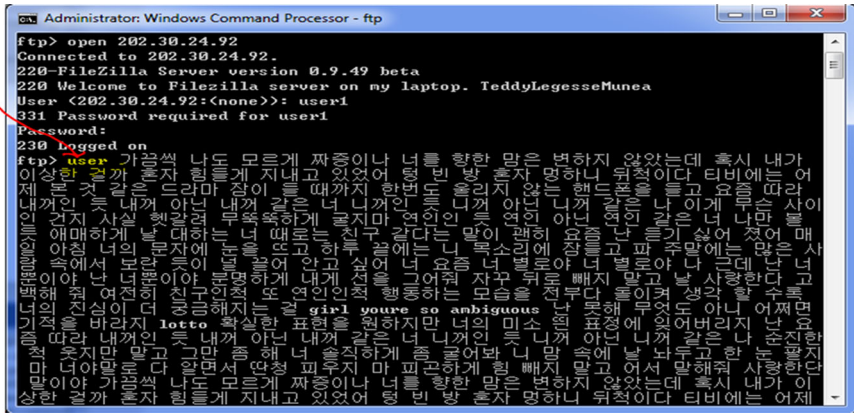

Fig. 11 Sending unexpected characters prior to user command in FileZilla Server

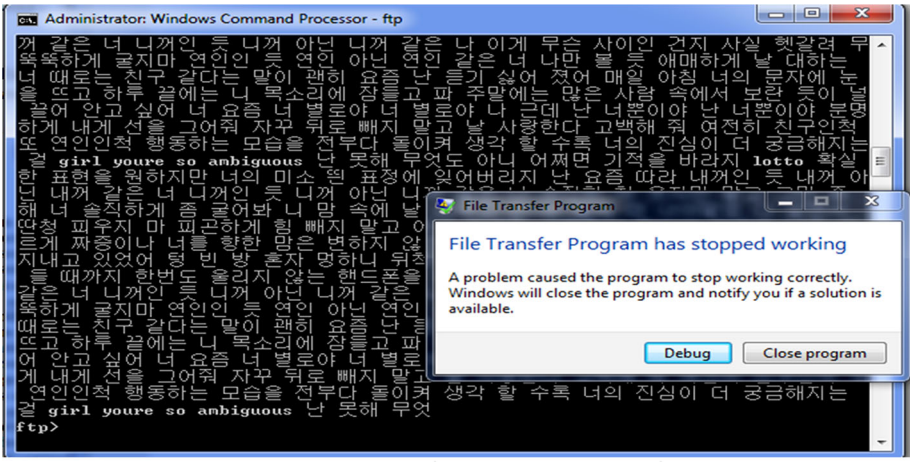

Fig. 12 FileZilla server crashed due to unexpected characters

6.4 Evaluation of our framework to others

We presented a Fuzz Testing framework for network protocol specifications which is applied on four FTP server implementations which also worked on Simple Mail Transfer Protocol and Microsoft MSN protocol. In this section, we compare our research with other related works and also additional criteria are put in Table 1.

The first one we compare is "Network Protocol analysis Using Bioinformatics Algorithms" [7], their whole objective is to find the length and discover the exact position of individual fields in the given protocol packets which is used to identify constant and non-constant data fields and to attach non-constant data fields with the type and length information. The problem with them is, they don't care about the behavioral sequence model of the protocol packets and also the input messages are given as a sample.

The second we evaluate our work is "A Model-based Approach to security Flaw detection of Network Protocol Implementations" [8]. They use two important concepts:

Table 1 Evaluation of our framewok

Criteria	Network protocol analysis using bioinformatics algorithms [7]	A model-based approach to security flaw detection of network protocol implementations [8]	Prospex: protocol specification extraction [9]	AutoFuzz—autmated network protocol fuzzing framework [10]	Design and implementation of fuzzing frame based on IoT applications
Intelligence level on protocol knowledge	Smart	Smart	Smart	Smart	Smart
Level of understanding the target protocol implementation	Lower level	Higher level	Both lower and higher level	Both lower and higher level	Both lower and high level
Input messages	Given as a sample	Socks v5 proxy helps to intercept the message	Given as a sample	Collected automatically from conversation between a client and a server using proxy server	Collected automatically from conversation between a client and a server using java socks server and proxy server
Fuzzing functions	–	Deterministic	Deterministic	Deterministic and non-deterministic	Deterministic and non-deterministic
Used characters	ASCII characters	ASCII characters	ASCII characters	ASCII characters	ASCII and UTF-8 characters
Target protocol applied	HTTP	Microsoft MSN instant messaging (MSNIM) protocol	Simple mail transfer protocol, server message block, session initiation protocol	File transfer protocol, simple mail transfer protocol	File transfer protocol, simple mail transfer protocol, microsoft MSN protocol
Vulnerabilities found protocol implementation servers	–	Gaim (LINUX) and Gaim (Windows)	Agobot C&C	Open and compact FTP server 1.2	Open and compact FTP server 1.2, FileZilla FTP server 0.0.49, wing FTP server 3.6.1, IndiFTPD FTP server

first they obtain the abstract behavior model which is approximated of a SUT protocol implementation, next they use it to direct selection of input for a fault coverage. A large number of input sequences are constructed automatically which we applied for our study. They don't care of individual messages.

The third research our framework evaluated is "Prospex: Protocol Specification Extraction" [9]. They automatically deduce specification of network protocols which are stateful. Additionally, they extract individual messages' format specifications after monitoring the application. They don't mention how they collected the input messages.

Lastly but not least, we compared our study to "AutoFuzz: Automated network Protocol Fuzzing Framework" [10]. Like AutoFuzz, we collect data automatically using Proxy Server. We have additional function like use Non-ASCII characters and break character restriction rule, unlike AutoFuzz, which are helpful in upgrading the fuzzing functions. Additionally, we used two character sets for fuzzing purpose: ASCII and UTF-8.

7 Conclusion

In this study, we presented a modified and better framework with the aim to extract network protocol implementations' specification automatically and to test it for implementation weakness. We started by explaining about, Fuzz testing, UTF-8, then about our framework and its components. Using this framework, we have seen how target protocol specifications are extracted. We need to learn the protocol specification's behavior model and then we need to construct corresponding FSA using the framework. Additionally, the framework is helpful in finding syntax of individual messages by abstracting protocol specification traces. At last this framework was applied to a lot of FTP server implementations with a success result in finding new and already existing vulnerabilities. This result basically shows that there are still the security threats in IoT application.

For future researches, we need to improve the framework with a multiple word to be done. The framework is not fully automated towards applying the fuzzing test on the server. We can add more abstraction functions to improve the fuzzing framework. Nowadays abstraction functions are being replaced with likes of use of similarity scoring techniques of sequence alignment algorithms.

Acknowledgments This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (2015R1A1A1A05001238).

References

1. Han, X., Wen, Q., & Zhang, Z. (2012). *A mutation-based fuzz testing approach for network protocol vulnerability detection*. Beijing University of Posts and Telecommunications, Beijing, 100876, China.
2. Takanen, A., DeMott, J., & Miller, C. (2008). *Fuzzing for software security testing and quality assurance*. Norwood, MA: Artech House Inc.
3. The ProxyFuzz Project. http://theartoffuzzing.com/.
4. http://en.wikipedia.org/wiki/UTF-8.
5. http://en.wikipedia.org/wiki/File_Transfer_Protocol.
6. Internet of Things (IoT). http://en.wikipedia.org/wiki/Internet_of_Things.
7. Beddoe, M. A. (2005). *Network protocol analysis using bioinformatics algorithms*. http://www.4tphi.net/~awalters/PI/pi.pdf.

8. Hsu, Y., Shu, G., & Lee, D. (2008). A model-based approach to security flaw detection of network protocol implementation. In *IEEE ICNP*.
9. Comparetti, P. M., Wondracek, G., Kruegel, C., & Kirda, E. (2009). Prospex: Protocol specification extraction. In *Proceedings of the 2009 30th IEEE symposium on security and privacy* (pp.110–125).
10. Gorbunov, S., & Rosenbloom, R. (2010). *AutoFuzz: Automated network protocol fuzzing framework*. Department of Mathematical and Computation Sciences, University of Toronto Mississauga, Canada L5L 1C6.
11. SOCKS Server http://en.wikipedia.org/wiki/SOCKS.
12. JAVA SOCKS Server. http://jsocks.sourceforge.net/.
13. Kitagawa, T., Hanaoka, M., & Kono, K. (2010). *AspFuzz: A state-aware protocol fuzzer based on application-layer protocols*. Department of Information and Computer Science, Keio University, 3-14-1, Yokohama, Japan.
14. The JAVA Swing Library. http://java.sun.com/javase/6/docs/api/javax/swing/package-summary.html.
15. The Java Universal Network/Graph Framework (JUNG). http://jung.sourceforge.net/.
16. Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology, 48*, 444–453.
17. Postel, J., & Reynolds, J. (1985). *Request for Comments: 959*. Network Working Group. http://www.faqs.org/rfcs/rfc959.html.
18. https://wiki.filezilla-project.org/Character_Set.
19. Open & Compact FTP Server. http://sourceforge.net/projects/open-ftpd/.
20. Wing FTP Server. http://www.wftpserver.com/.
21. Windows Proxifier. http://www.proxifier.com/.

Tewodros Legesse Munea received B.S. degree in Computer Science from Addis Ababa University, Addis Ababa, Ethiopia, in 2008. He is currently pursuing his M.S. degree in Computer Engineering from Ajou University, and is with the Information and Communication Security Lab. His current research interest is Network Protocol Fuzzing.

I. Luk Kim receives M.S. degree in Information Security from Korea University, Seoul, Korea in 2010 and B.S. in Kwangwoon University in 2012. He is pursuing Ph.D. degree in Computer Science from Purdue University, Indiana, USA. His research interests are System/Network Security, Software Engineering, Program Analysis, and Cyber Physical Security.

Taeshik Shon received Ph.D. in Information Security from Korea University, Seoul, Korea and M.S. and B.S. in Computer Engineering from Ajou University, Suwon, Korea. From Aug. 2005 to Feb. 2011, Dr. Shon was a senior engineer in the Convergence S/W Lab, DMC R&D Center of Samsung Electronics Co., Ltd. He is currently assistant professor at the Division of Information and Computer Engineering, College of Information Technology, Ajou University. His research interests include Convergence Platform Security, Mobile Cloud Computing Security, Mobile/Wireless Network Security, WPAN/WSN Security, Anomaly Detection.