AutoFuzz: Automated Network Protocol Fuzzing Framework

Serge Gorbunov and Arnold Rosenbloom
serge.gorbunov@utoronto.ca, arnold@cs.toronto.edu
Department of Mathematical and Computational Sciences,
University of Toronto Mississauga,
Mississauga, Ontario, Canada L5L 1C6

Abstract
Assessing software security involves steps such as code review, risk analysis, penetration testing and fuzzing. During the fuzzing phase, the tester's goal is to find flaws in software by sending unexpected input to the target application and monitoring its behavior. In this paper we introduce the AutoFuzz [1] - extendable, open source framework used for testing network protocol implementations. AutoFuzz is a 'smart', man-in-the-middle, semi-deterministic network protocol fuzzing framework. AutoFuzz learns a protocol implementation by constructing a Finite State Automaton (FSA) which captures the observed communications between a client and a server [5]. In addition, AutoFuzz learns individual message syntax, including fields and probable types, by applying the bioinformatics techniques of [2]. Finally, AutoFuzz can fuzz client or server protocol implementations by intelligently modifying the communication sessions between them using the FSA as a guide. AutoFuzz was applied to a variety of File Transfer Protocol (FTP) server implementations, confirming old and discovering new vulnerabilities.

Key words:
Automated Fuzzing, Software Security, Vulnerability Detection

1. Introduction

1.1 Background

Flaws in the implementations of network protocols are some of the most serious security problems. One such flaw could allow a malicious user to attack vulnerable systems remotely over the Internet. Approximately 85% of all vulnerabilities reported by the National Vulnerability Database [15] in the last 3 years can be exploited remotely.

A fuzzer is a tool used to discover implementation flaws by sending the target implementation unusual inputs in hopes of producing unexpected behavior. A protocol fuzzer can be classified as 'smart' or 'dumb' depending on its knowledge of the network protocol implemented by its targets. A 'dumb' fuzzer sends random inputs to its target. It has no knowledge of the communication protocol implemented by the target. 'Dumb' fuzzers are easy to develop and are immediately applicable to any protocols clients or servers. However, 'dumb' fuzzing is measured to be 50% less effective than 'smart' fuzzing [11]. One example of a 'dumb' fuzzer is ProxyFuzz [17]. ProxyFuzz is a man-in-the-middle non-deterministic network fuzzer. It randomly changes the network traffic [17] between a connected client and server. Fuzzers of the second type, 'smart' fuzzers, have a pre-programmed understanding of the protocol implemented by the targets they fuzz. They typically understand the protocol's state machine, messages syntax and field types and use this to efficiently fuzz deep into target implementation code. Peach is an example of a 'smart' fuzzer [16]. Disadvantages of 'smart' fuzzers include their reliance on the availability of a protocol's specification documents and the degree to which a target implementation conforms to the published specification. In addition, 'smart' fuzzers require manual adaptation to customize them for each new protocol they are to apply to. Therefore, its application to new protocols is labour intensive and tedious.

1.2 Previous Work

A number of attempts have been made to automatically extract protocol specifications for 'smart' fuzzers [2][4][5]. In [5] the automatic extraction of the protocol's specification is based on synthesizing an abstract behavioral model of a protocol implementation. The behavioral model is realized as a Finite State Automaton (FSA) constructed from the recorded conversations between a client and a server. The FSA represents, in a succinct way, the key states and transitions of a protocol implementation and can be used to systematically guide the flaw detection process. The main algorithm proposed in [5] for synthesizing an abstract behavioral model of a protocol implementation is based on passive synthesis with partial FSA reduction. Given a large collection of network traces the algorithm constructs and minimizes a FSA. The construction of a FSA relies on an abstraction function. An abstraction function is a simple function used to map similar messages to a unique abstract representation. For example, SMTP client requests can be

abstracted to their first four characters. That is, messages '*mail from: test@test.com*' and '*mail from: account@test.com*' are abstracted to '*mail*'. Also, SMTP server replies can be abstracted to their first three characters. For example, messages *"550 Permission denied"*, *"221 Bye!"* and *"230 User anonymous logged in"* are abstracted to *"550"*, *"221"* and *"230"* respectively. The tester must supply two abstraction functions, one for the input messages to the target being fuzzed, the other for the output messages. In [4], the authors focus on automated protocol specification extractions by constructing the protocol's FSA and determining message types. However, their technique of FSA construction is substantially different from the technique presented in [5]. Their final system can be used to extract protocol specifications. However, to the best of our knowledge, neither of the systems [4] nor [5] is available publically for future development or research. In [2], the authors try to determine fields of individual protocol messages by using bioinformatics algorithms. In order to determine message fields, similar message samples are aligned using multiple string alignment algorithms and their consensus sequences are analyzed to understand the beginning and the end of fields in the message [2]. Their open-source tool can be used to determine message fields for a collection of protocol messages.

1.3 The New Fuzzing Framework

This paper introduces the AutoFuzz. This open source fuzzing framework is a 'smart', man-in-the-middle fuzzer. For simplicity in the discussion that follows we assume that AutoFuzz is used to fuzz the server side of a network protocol implementation. More specifically, the messages coming from the client to the server are denoted as input messages, and the messages coming from the server to the client are denoted as output messages. However, AutoFuzz can be applied with equal effectiveness to fuzz the client side. First, AutoFuzz extracts specifications of a network protocol implementation from conversations recorded by acting as a man-in-the-middle between server/client sessions. As in [5] AutoFuzz constructs a FSA which captures the sampled conversations, and so, understands the protocol at a high level. AutoFuzz can be extended to understand any protocol by importing appropriate abstraction functions. Then, using the techniques presented in [2], AutoFuzz finds the fields of individual messages. In addition, it derives the type information of the variable data fields of individual messages, and so, understands the protocol at a lower level. More specifically, for each message of the sampled conversations, AutoFuzz associates a Generic Message Sequence (GMS) that is used to capture the syntax information of the message. A GMS is a representation of a message that separates static from variable data fields and associates variable data fields with type and length information. By using GMSs, AutoFuzz eliminates the need for protocol specific fuzzing functions as required by [5]. Fuzzing functions can now be performed on GMS representations instead of individual messages and be based on the derived type or length information of the static or variable data fields. AutoFuzz can also be extended with new fuzzing functions. Finally, AutoFuzz intelligently fuzzes server or client network protocol implementations acting as a man-in-the-middle and using the constructed FSA as a guide during the vulnerability detection process. AutoFuzz was successfully applied to several File Transfer Protocol (FTP) implementations where it found both existing and new vulnerabilities.

2. Framework Overview

2.1 Main Components

The main components of AutoFuzz are **(1) AutoFuzz Graphical User Interface (GUI), (2) Proxy Server, (3) Protocol Specifications Extractor and (4) Fuzzing Engine.** We elaborate on each below.

(1) AutoFuzz GUI allows testers to easily interact with the fuzzer and control its actions. It is constructed using the JAVA Swing library [13]. To visualize a protocol's FSA AutoFuzz uses JUNG graphing library [14].

(2) Proxy Server. AutoFuzz works as a proxy server between a client and a server. It records and modifies the application level traffic to extract protocol specifications and perform fuzzing operations. The proxy server is based on the JAVA Socks server [6], but has been modified to allow direct manipulation of the application level traffic.

Figure 1. AutoFuzz Proxy Model

(3) Protocol Specifications Extractor. The specifications extractor extracts the FSA of a network protocol implementation from a sample of communication sessions between a client and a server. AutoFuzz can understand any application level protocol implementation after appropriate input/output abstraction functions are imported in it. It also extracts GMSs using the algorithm outlined in the Generic Message Sequence Construction section to understand to the syntax of individual messages.

(4) Fuzzing Engine. The fuzzing engine modifies the communication traffic between a server and a client by applying fuzzing functions. We elaborate more on how the traffic is modified in the Fuzzing Algorithm Section. The current set of fuzzing functions contains both deterministic and non-deterministic functions. Deterministic functions insert preprogrammed data into the GMSs such as large strings, maximum/minimum integer values and others. Non-deterministic functions randomly skip static or variable data fields of GMSs, take random transitions in the FSA and insert random data into the GMSs. The fuzzing engine can be extended with new fuzzing functions. All actions during the fuzzing process are recorded in the logs files. This allows testers to determine the state in the communication and the exact input message modifications that were performed during the unexpected application behavior.

2.2 Process Work Flow

The process flow involved in fuzzing using AutoFuzz is presented in Figure 2.

Step 1: Protocol traces are recorded using AutoFuzz's built-in proxy server. The traces can manually be edited by the tester, exported or imported at any point of time.

Step 2: Protocol's behavior model is constructed based on the passive synthesis with partial Finite State Automaton (FSA) reduction proposed in [5].

Step 3: Individual message syntax is extracted and stored in GMS. We extend the use of the abstraction function from [5] to generate clusters of input messages for GMS construction. Hence, each cluster represents a collection of similar input messages. The detailed algorithm is presented in Generic Message Sequence construction section. Intuitively, given the abstraction function for the input messages, similar input messages are clustered together using this abstraction function. Next, sequence alignment algorithms are applied to generate GMS for each cluster. Finally, we traverse the protocol's FSA and associate each transition with the appropriate GMS.

Step 4: Fuzzing functions are applied by modifying live communication sessions between the client and the server. The fuzzing engine is responsible for assigning a fuzzing function. Which fuzzing function is performed is determined by the current state in the FSA, input message and which functions have already been applied. The complete algorithm is presented in the fuzzing algorithm section.

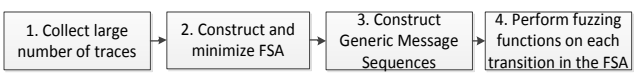

Figure 2. AutoFuzz Fuzzing Processes

3. Generic Message Sequence Construction

We present a complete algorithm used to extract Generic Message Sequences (GMSs). Remember, GMS is a representation of a message that separates static from variable data fields and associates variable data fields with type and length information. A cluster is denoted as a collection of similar messages. *Step 1:* Similar messages are clustered together using a new clustering technique. *Step 2:* Multiple sequence alignment algorithm described in [2] is performed on each cluster. *Step 3:* GMS is constructed for each cluster. *Step 4:* Each transition in the protocol's FSA is associated with the corresponding GMS.

Step1: First, we present a new technique used to cluster similar messages. Remember, that for simplicity, we denote all messages coming from the client to the server as input messages, and all messages coming from the server to the client as output messages.

Define a set of input messages as $I = \{I_1, I_2, \dots, I_n\}$. Let abs_input denote the abstraction function on the input messages. The algorithm returns clusters of similar input messages using the abs_input function. More specifically, for all $1 <= j <= n$, define C_j as follows: $C_j = \{I_z \in \{I_1, I_2, \dots, I_n\} \mid abs\_input(I_j) = abs\_input(I_z)\}$. The algorithm returns $\{ C_j \mid 1 <= j <= n \}$.

Consider the following set of sample input messages of the Simple Mail Transfer Protocol (SMTP). Let $I = \{$"$rcpt\ to: < test@test.com >$",
"$mail\ from: < anonymous@domain.ca >$",
"$rcpt\ to: < test@domain.com >$",
"$mail\ from: < sample@pc.ru >$",
"$mail\ from: test@test.com$"$\}$.

For any input message I_j in I, let $abs\_input(I_j)$ return its first four characters. Applying the algorithm on this example it returns a set of two clusters $\{C_1, C_2\}$ where $C_1 = \{$"$rcpt\ to: < test@test.com >$",
"$rcpt\ to: < test@domain.com >$"$\}$
and
$C_2 = \{$"$mail\ from: < anonymous@domain.ca >$",
"$mail\ from: < sample@pc.ru$",
"$mail\ from: test@test.com$"$\}$.

Step 2: After input messages are clustered we perform multiple sequence alignment algorithm on each cluster proposed in [2]. For each cluster the algorithm returns a list of aligned messages. Alignment of the input messages is performed using the Needleman-Wunsch algorithm [8] based on the progressive alignment technique.

For example, applying the algorithm on the cluster C_2, presented in Figure 3, we obtain three aligned input messages presented in Figure 4. The result is three input messages that have the same length where "-" represents a sequence gap.

m	a	i	l		f	r	o	m	:	<	a	n	o	n	y	m	o	u	s	@	d	o	m	a	i	n	.	c	a	>
m	a	i	l		f	r	o	m	:	<	s	a	m	p	l	e	@	p	c	.	r	u	>							
m	a	i	l		f	r	o	m	:	<	t	e	s	t	@	t	e	s	t	.	c	o	m	>						

Figure 3. Sample SMTP Input Messages

m	a	i	l		f	r	o	m	:	<	a	n	o	n	y	m	o	u	s	-	-	-	-	-	@	d	o	m	a	i	n	.	c	-	a	>
m	a	i	l		f	r	o	m	:	<	-	-	-	-	-	-	-	-	s	a	m	p	l	e	@	-	-	-	-	p	c	.	-	r	u	>
m	a	i	l		f	r	o	m	:	<	-	-	-	-	-	-	t	e	s	-	-	-	-	t	@	-	-	t	e	s	t	.	c	o	m	>

Figure 4. Aligned Sample SMTP Client Requests. "-" represents sequence gap.

Step 3: Next, we construct a Generic Message Sequence (GMS) for each cluster. On the implementation level a GMS is an array list of message blocks, where a block corresponds to either static or variable data field.

First, we identify the beginning and the end of the static and variable data fields. Intuitively the algorithm looks at characters at the same position across all messages and, if all characters are the same, it marks that position as static position in the resulting GMS, otherwise variable position. Consecutive static and variable positions in the GMS are denoted as static and variable data fields, respectively.

More formally, define $\{[m_1 = m_{11}, m_{12}, \ldots, m_{1n}], [m_2 = m_{21}, m_{22}, \ldots, m_{2n}], \ldots, [m_k = m_{k1}, m_{k2}, \ldots, m_{kn}]\}$ as a set of aligned messages where for all $1 <= i <= k$, m_i is an input message and for all $1 <= j <= n$, m_{ij} is its j'th character. Define for all $1 <= i <= n, g[i]$ as the i'th symbol in the GMS. We define $g[i]$ as follows:

$$g[i] = \begin{cases} m_{1i}, if\ \forall j, 1 \le j \le k, m_{1i} = m_{ji} and\ m_{1i}! = '-' \\ \mu, \text{ otherwise} \end{cases}$$

The algorithm returns $GMS = [g[1], g[2], \ldots g[n]]$. Note, "μ" should be replaced by some unique character not seen otherwise in any of the sequences. Consecutive "μ"s in the resulting GMS correspond to variable data fields.

Applying the algorithm on the aligned SMTP input messages presented in Figure 4, we obtain the GMS presented in Figure 5.

m	a	i	l		f	r	o	m	:	<	μ	μ	μ	μ	μ	μ	μ	μ	s	μ	μ	μ	μ	μ	@	μ	μ	μ	μ	μ	μ	.	μ	μ	μ	>

Figure 5. The intermediate GMS obtained from 3 SMTP messages presented in Figure 4.

Next, for each variable data field, identified as consecutive "μ"s in the GMS, we associate the type information by looking over each character at those positions in the aligned sequences and checking which type set they corresponds to. The final GMS applied to our example is presented in Figure 6.

m	a	i	l		f	r	o	m	:	<	£	£	£	£	£	£	£	£	s	£	£	£	£	£	@	£	£	£	£	£	£	.	£	£	£	>

Figure 6. The final GMS obtained from 3 SMTP messages presented in Figure 4. Consecutive "£" s correspond to alpha-numeric variable data fields.

Step 4: Finally, we traverse the protocol's FSA, abstracting each message at a transition and assigning it the corresponding GMS. We now have the protocol's FSA generated from the large sample of network traces where with each transition has a specific GMS assigned.

4. Fuzzing Algorithm

Once the network protocol specifications are extracted by constructing its FSA and GMSs, the fuzzing is started. In addition to the FSA and associated GMSs the fuzzing engine is loaded with an extendable list of fuzzing functions. Initially, the fuzzing engine sets its state to the root of the protocol's FSA. It then monitors the input traffic, making appropriate transitions and applying fuzzing functions. The abstract version of the algorithm is presented in Figure 7.

Note, that the current implementation does not compare the server output messages to the modified responses against the associated transition in the FSA. Ideally, the output messages should be compared to the output messages associated with the transition in the FSA to determine whether a specific type of an unexpected behavior has occurred. For that, the FSA should be aware of the typical negative server responses, such as invalid syntax.

5. Experimental Results

We applied AutoFuzz to extract protocol specification of the File Transfer Protocol (FTP) and fuzz multiple FTP server implementations. This section provides an overview of FTP, describes the setup environment and our findings.

5.1 File Transfer Protocol

File Transfer Protocol (FTP) is an application level protocol used on the Transmission Control Protocol/Internet Protocol (TCP/IP) networks for file exchange. The original specifications of FTP were proposed in 1971 [3], but have been modified many times since then. Most commonly, the FTP is implemented as follows. First, a client connects to the server on port 21, called the control port. The client requests, including the login process, are sent using this socket in ASCII. When the client requests to transfer data, a new socket is

typically opened on port 20 with the server. Port 20 is called the data connection port.
Most client requests to the FTP server consist of a four letter message type followed by the actual message. Commands CWD, RWD, MKD and PWD are the only three letter message type commands [10]. The server responses are also in ASCII with first three digits corresponding to a status code following by an optional message.

5.2 Setup Environment

Step 1: We write and import the abstraction functions for the FTP server implementations. All input messages coming from the client to the server are abstracted to its first four characters, except for the messages beginning with CWD, RWD, MKD and PWD, which are abstracted to its first three characters. All output messages coming from the server to the client are abstracted to its first three characters.
Step 2: We install an FTP server implementation that will be fuzzed, such as Firezilla FTP Server [12].
Step 3: We setup a proxifier to redirect all traffic of Windows ftp.exe client to AutoFuzz proxy server. (Note, since AutoFuzz works as a proxy server between the server and the client, the client connections must be encapsulated in SOCKS5 sessions [7]. One can run a Proxifier [18] on a process to encapsulate its traffic in SOCKS5 protocol and redirect it to a specific SOCKS5 proxy server.)
Step 4: Next, we run AutoFuzz and start its proxy server.
Step 5: We manually connect to the FTP server using ftp.exe client and perform common FTP requests. For example, we connect to the server using different login credentials, download and upload different files, create and remove directories. Each session is identified as a separate network trace. In total, we record 23 network traces.
Step 6: We build the FSA corresponding to the network traces, which is presented in Figure 8. We also construct GMSs and associate them with the appropriate FSA transitions (Figure 9).
Step 7: We start the fuzzing engine. Finally, we run a small FTP client, written in JAVA, to automatically perform multiple sessions with the server and execute various requests, while AutoFuzz automatically follows the fuzzing algorithm presented in Figure 9.

Fuzzing Algorithm Flowchart

Input: Protocol's FSA with each transition associated with a GMS

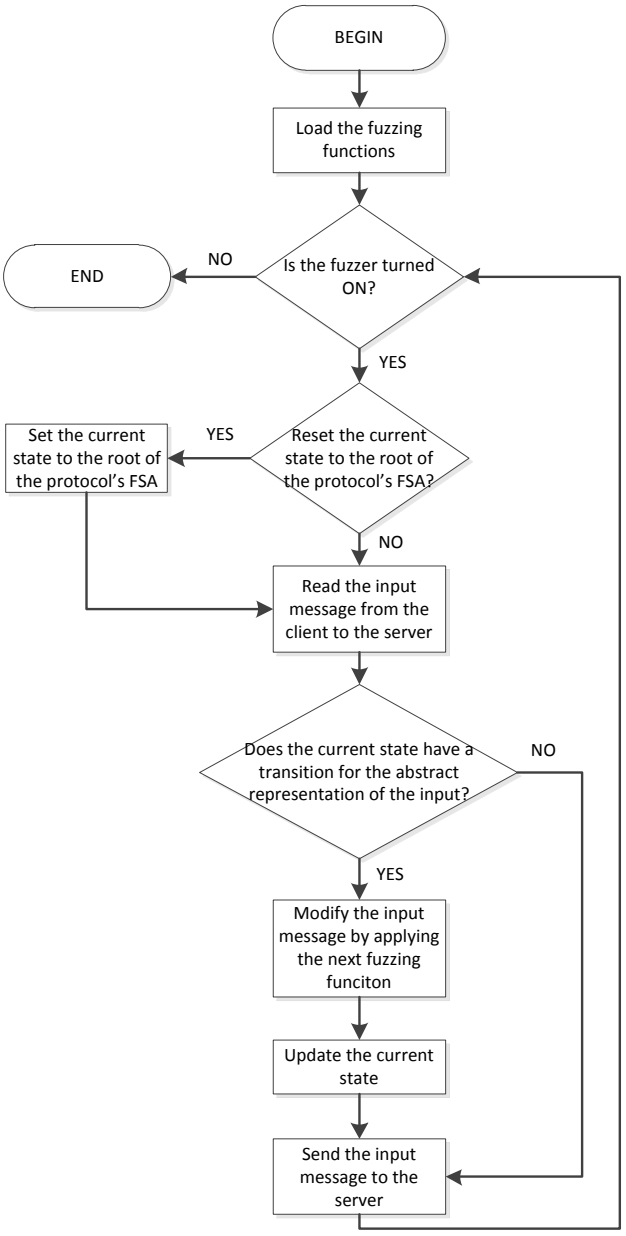

Figure 7. Fuzzing Algorithm Flowchart. The fuzzer is turned on/off by the tester. The tester also sets when the current state should be reset to the root of the protocol's FSA.

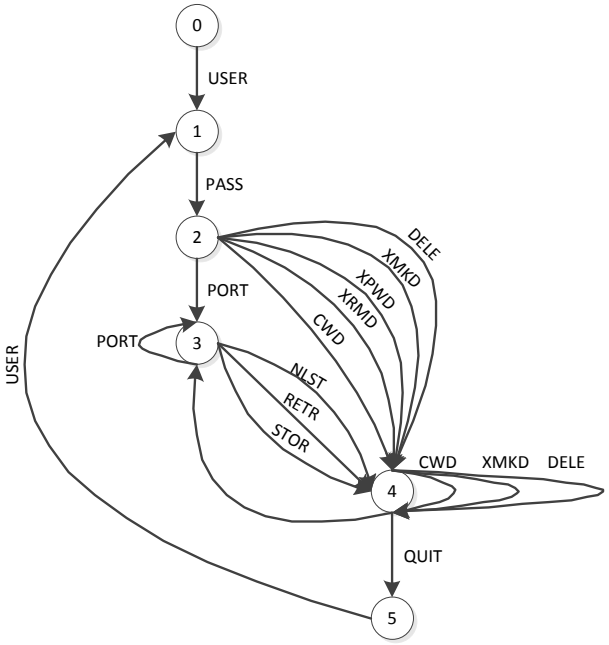

Figure 8. FTP Finite State Automaton constructed from 23 network traces.

State ID	Input Abstract Representation	GMS
0	USER	USER *AAAAAAAAAAAAAAAA*
1	PASS	PASS *AAAAAAAAAAAAAA*
2	PORT	PORT 192,168,192,1*AAAAAAAAAAAAAAAAA*
2	XRMD	XRMD *AAAAAAAAAAA*
2	XMKD	XMKD *AAAAAAAAA*
3	STOR	STOR test.txt 2,1*AAAAA*4,*LLLLLL*
4	QUIT	QUIT *AAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAA*
4	XMKD	XMKD *AAAAAAAAA*
4	DELE	DELE *AAAAAAAAAAAAAAAA*
4	CWD	CWD *AAAAAAAAAA*
5	USER	USER *AAAAAAAAAAAAAAAA*

Figure 9. FTP Generic Message Sequences. Consecutive *'A's* corresponds to a variable data field of any type. Consecutive *'L's* corresponds to a variable data field of a Long Integer.

5.3 Results

We applied AutoFuzz to automatically fuzz three different FTP server implementations: Firezilla FTP Server 0.9.34, Open and Compact FTP Server 1.2 and Wing FTP Server 3.5.2 [12][9][19]. We were unable to find any unexpected behavior instances of Firezilla or Wing FTP servers, but were able to find numerous unexpected behavior instances of Open and Compact FTP Server 1.2. A first set of unexpected behavior instances involves crashing Open and Compact FTP Server by sending arbitrary long strings prior to the authentication on USER, PASS and PORT commands, and sending '\r\n' string prior to or after authentication at any state of the server. The first denial of service attack was already known to the public, while the second attack was new. Another set of unexpected behavior instances involves arbitrary command execution on the server prior to authentication. This attack is even more dangerous since a malicious user does not need to know how to write any shellcode to completely gain control over the server. This attack was also unknown. The developers of Open and Compact FTP Server 1.2 were notified of both vulnerabilities.

6. Conclusion and Future Work

This paper presented a new framework intended to automatically extract specifications of network protocol implementations and test it for implementation flaws. We explained how the framework extracts protocol specifications by learning its behavior model and constructing a corresponding FSA. The framework also extracts individual message syntax allowing abstracting the set of fuzzing functions to apply to any protocol implementation. The framework was applied to multiple FTP server implementations and succeeded in finding old and new vulnerabilities.

There is still a lot of work to be done towards creating a fully automated fuzzing system. Our framework can be extended by incorporating new abstraction and fuzzing functions. It can also be extended by implementing additional fuzzing models. For example, the proxy server can be improved to automatically replay previously recorded traffic. In addition, the framework should be tested on other than ASCII protocol implementations and compared with other fuzzing tools. Different automated solutions aiming to replace the abstraction function should be considered, such as use of similarity scoring techniques of sequence alignment algorithms.

In addition, the framework can be used as a start towards automated honeypot construction. That is, using our framework it is possible to automatically extract protocol specifications which can be incorporated with a separate tool that uses these specifications to mimic real protocol implementations, hence interacting with potential attackers.

References

[1] AutoFuzz: Automated Network Protocol Fuzzing Framework. http://autofuzz.sourceforge.net/

[2] M. A. Beddoe, *Network Protocol Analysis Using Bioinformatics Algorithms*. Available: http://www.4tphi.net/~awalters/PI/pi.pdf.

[3] A.K. Bhushan, "*Request for Comments: 114*", Network Working Group, 1971. Available: http://www.faqs.org/rfcs/rfc114.html.

[4] P. M. Comparetti, G. Wondracek, C. Kruegel, E. Kirda, *"Prospex: Protocol Specification Extraction"*, Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, p.110-125, May 17-20, 2009.

[5] Y. Hsu, G. Shu and D. Lee, "*A Model-based Approach to Security Flaw Detection of Network Protocol Implementation"*, IEEE ICNP, 2008.

[6] JAVA SOCKS Server. http://jsocks.sourceforge.net/.

[7] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas and L. Jones, *"Request for Comments: 1928"*, Network Working Group, 1996. Available: http://www.ietf.org/rfc/rfc1928.txt.

[8] S. B. Needleman and C. D. Wunsch. *"A general method applicable to the search for similarities in the amino acid sequence of two proteins."* Journal of Molecular Biology, 48:444-453, 1970.

[9] Open & Compact FTP Server. http://sourceforge.net/projects/open-ftpd/.

[10] J. Postel and J. Reynolds, "*Request for Comments: 959*", Network Working Group, 1985. Available: http://www.faqs.org/rfcs/rfc959.html.

[11] A.Takanen , J. DeMott and C. Miller, *"Fuzzing for Software Security Testing and Quality Assurance"*, Artech House, Inc., Norwood, MA, 2008

[12] The Firezilla Project. http://filezilla-project.org/.

[13] The JAVA Swing Library. http://java.sun.com/javase/6/docs/api/javax/swing/package-summary.html.

[14] The Java Universal Network/Graph Framework (JUNG). http://jung.sourceforge.net/.

[15] The National Vulnerability Database. http://web.nvd.nist.gov.

[16] The Peach Project. http://peachfuzzer.com/.

[17] The ProxyFuzz Project. http://theartoffuzzing.com/.

[18] Windows Proxifier. http://www.proxifier.com/

[19] Wing FTP Server. http://www.wftpserver.com/.

Serge Gorbunov is working towards B.S. at the University of Toronto Mississauga, specializing in Information Security. He worked at IBM Canada lab as a Software Developer Intern during summer of 2009 and has been a member of the Canadian Honeynet Project since May of 2009.

Arnold Rosenbloom is a Senior Lecturer at the Department of Mathematical and Computational Sciences, University of Toronto at Mississauga, home of the UofT Information Security Program. His interests range from Information Security to Web Programming, from Computational Complexity to first year pedagogy.