Recycling Test Cases to Detect Security Vulnerabilities

João Antunes Nuno Neves
LASIGE, Departamento de Informática,
Faculdade de Ciências da Universidade de Lisboa, Portugal
{jantunes,nuno}@di.fc.ul.pt

***Abstract*—The design of new protocols and features, e.g., in the context of organizations such as the IETF, produces a flow of novel standards and amendments that lead to ever changing implementations. These implementations can be difficult to test for security vulnerabilities because existing tools often lag behind. In the paper, we propose a new methodology that addresses this issue by recycling test cases from several sources, even if aimed at distinct protocols. It resorts to protocol reverse engineering techniques to build parsers that are capable of extracting the relevant payloads from the test cases, and then applies them to new test cases tailored to the particular features that need to be checked. An evaluation with 10 commercial and open-source testing tools and a large set of FTP vulnerabilities shows that our approach is able to get better or equal vulnerability coverage than the original tools. In a more detailed experiment with two fuzzers, our solution showed an improvement of 19% on vulnerability coverage when compared with the two combined fuzzers, being capable of finding 25 additional vulnerabilities.**

***Keywords*-test case generation; vulnerability assessment; protocol reverse engineering**

I. Introduction

Over the years, several black box approaches have been devised to discover security vulnerabilities in software, namely in network facing components such as servers. For example, scanners resort to a database of checking modules to automate the task of identifying the presence of previously reported vulnerabilities [1], [2]. They typically start by determining the type and version of a target server running in a remote machine, and then they perform a pre-defined message exchange to find out if a particular vulnerability exists. Other tools such as fuzzers automatically generate many test cases that include unexpected data, which may cause incorrect behavior in the component (e.g., a crash), allowing the detection of a flaw.

When they appeared, fuzzers mainly used random data to evaluate the component's robustness [3], but eventually they evolved to include more knowledge. Servers are usually protected with a group of validation routines that decide on the correctness of an interaction, letting the computation progress only when the appropriate conditions are present (e.g., non-compliant protocol messages are immediately discarded). Therefore, specialized fuzzers, which understand the component interfaces, were developed and are able to generate valid interactions carrying malicious payloads [4], [5], [2], [6], [7]. The down side of these approaches is that as they become increasingly knowledgeable about the target component, and therefore more effective at finding vulnerabilities, they lose generality and become useless to assess other types of servers.

To address this issue, the paper describes an approach that recycles existing test cases, making it particularly useful in the following scenarios. First, servers implementing newly designed protocols are typically hard to test because of lack of support from the available tools. For instance, the IETF publishes a few hundred new RFCs every year, some of them introducing novel protocols [8]. This fast-paced standard deployment would require a constant upgrade of the testing tools, but what happens in practice is that tools typically lag behind. Second, many application areas are very dynamic, which is usually reflected in protocol specifications that are constantly being updated. For instance, from the publication of the current File Transfer Protocol (FTP) standard [9], the IETF has published 42 related RFCs. When this happens, old test cases are unable to check the new features, and extensions to the protocol may even render them ineffective. Last, servers implementing proprietary (or closed) protocols are difficult to assess because little information is available, and therefore, they end up being evaluated only by their own developers.

Our methodology takes test cases available for a certain protocol and re-uses them to evaluate other protocols or features. It resorts to protocol reverse engineering techniques to build parsers that are capable of extracting the relevant payloads from the test cases, i.e., the malicious parameter data of the messages. These payloads are then applied in new test cases for the target server, by generating malicious interactions that cover the particular features under test. In case the whole server needs to be checked, tests are created to explore the entire state space of the implemented protocol.

To evaluate the methodology, we implemented it in a tool and made a comparison with 10 commercial and open-source testing tools. The experiments were based in analyzing the capabilities of the tools to discover vulnerabilities in FTP servers. A large database was built with all FTP vulnerabilities discussed in security related web sites (currently with 131 vulnerability descriptions and exploits), covering various kinds of problems, from SQL injection to resource exhaustion. The results show that recycling various test cases for the FTP and other protocols, does not diminish the vulnerability detection capability of the automatically generated test cases. In fact, our approach is able to get better

or equal protocol and vulnerability coverage than the original tools. In a more detailed experiment with the Metaploit and DotDotPwn fuzzers, our tool showed an improvement of 19% on vulnerability coverage when compared with the two combined fuzzers, being able to discover 25 additional vulnerabilities.

II. Methodology

The goal of our approach is to produce test cases in an automated way for a particular protocol implementation, i.e., the target server. The solution resorts to test cases from different sources to maximize the richness of the testing payloads, together with a specification of the protocol implemented by the target server to increase the overall coverage of the generated test cases. This allows testing payloads for a particular protocol to be re-used in other kinds of servers, as well as the assessment of added features to newer releases of a server.

The specification of the protocol implemented by the server determines, among other things, the format of the messages (*protocol language*) and the rules for exchanging them with the clients (*protocol state machine*). It also indicates the expected behavior of the server, such as the responses that should be returned. We define a *test case* as a sequence of network messages that check the implementation of a specific protocol feature (e.g., the correctness of directory traversal). The whole set of test cases should, however, cover as much of the protocol implementation as possible. Usually, a test case is composed of a *prelude* of messages that take the server into the desired protocol state, followed by a message that contains the actual test, the *testing message*.

The messages used in the prelude must be carefully constructed. Malformed messages can be dismissed by the validation mechanisms at the server, and therefore the server will not go to the intended protocol state. Messages with wrong parameter data, such as invalid credentials or non-existing path names, will also fail to take the server into the desired state. For these reasons, the prelude of messages must comply to the protocol specification and to the configuration of the target server. The testing message is transmitted to trigger some response from the target server, causing some abnormal behavior in case the server has a flaw. This message is usually a valid protocol request with a special payload, such a frontier value or random data. We call this payload, i.e., the parameter data of the protocol request, the *testing payload*. Since we are looking for security vulnerabilities, the testing payload will correspond to some malicious data.

The main steps of methodology are summarized in Figure 1. The test case generation approach resorts to existing test cases obtained from various sources, potentially for different protocols, in order to get a rich set of testing payloads. To parse and extract the payloads of the messages included in the input test cases, we employ input protocol language parsers that recognize the formats of the messages. The obtained payloads are forwarded to the testing payload extractor to be organized and selected for the next phase. An output protocol specification provides the definition of the protocol that the target server implements. This specification gives the ability to construct messages according to the recognized formats and state machine of the protocol. To make the target server go from the initial protocol state to another state requires that a sequence of messages of the correct type and with the correct payload is transmitted – the prelude of the test case. Hence, a target server transition context provides the necessary information about the server's configuration and execution to create protocol messages with the correct parameter values (e.g., valid credentials and/or file system structure) so that it can reach any protocol state.

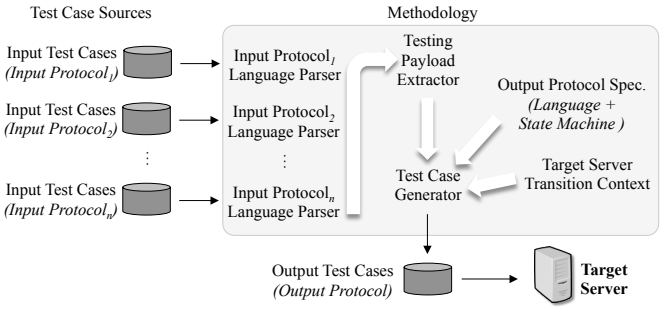


Figure 1. Overview of our methodology.

The test case generator produces the output test cases to be sent to the target server. It generates test cases that cover the whole protocol space, containing the extracted testing payloads. To exhaustively test each protocol feature (or type of request), several approaches can be employed, such as using all combinations of the testing payloads in each message field or pairwise testing. Therefore, various strategies can be implemented in the generator, which tradeoff the time and resources to complete with the coverage of the testing.

III. Implemented tool

This section provides details about an implementation of our methodology. The general architecture of the tool is depicted in Figure 2. The implementation differs from the methodology in a few key points. Most notably our tool does not require individual protocol language parsers and a protocol specification to be provided, since we resort to additional samples of network traces and protocol reverse engineering techniques to infer the language of the protocols.

A. Generic protocol language parser

The tool implements a *generic protocol language parser* component to infer input protocol language parsers from additional samples of network traces. The additional network traces can be collected from custom network servers that implement the protocol used in the input test cases, or downloaded from the Internet as packet capture files. They only need to include the messages of regular client-server interactions. Then, a reverse engineering method is employed to derive the message formats of the protocols.

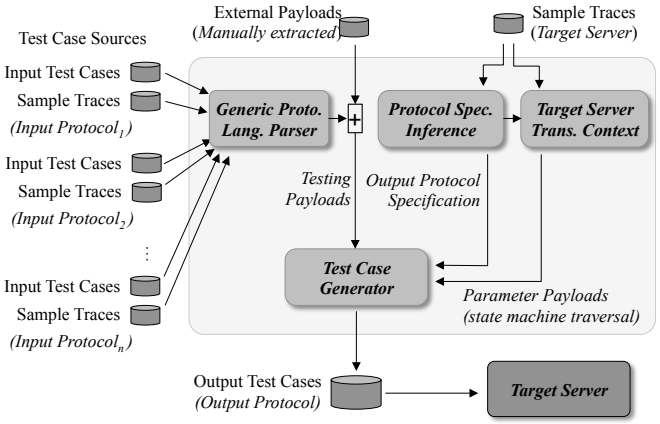


Figure 2. Architecture of the implemented tool.

The current implementation is based on the reverse engineering techniques developed in ReverX for text-based protocols [10]. Messages in text-based protocols are usually composed of a sequence of text fields. Some fields hold a limited range of predefined command names, which discriminate the type of the protocol request. Following the field with the command name, the message usually has one or more parameter fields that further refine the function of the protocol request. Based on this idea, reverse engineering divides the messages of the trace in their respective fields, and then builds a prefix tree acceptor (PTA) that recognizes all messages. The PTA is minimized to obtain a more condensed representation, which corresponds a finite-state machine automaton. Next, a statistical analysis is performed on the automaton to identify the command and parameter fields. The parameters are generalized as a regular expression, to produce a new automaton that is capable of accepting other messages of the protocol that are not present in the traces. This automaton is however non-deterministic, and therefore a determinization and a minimization procedure is performed to create the final automaton, which understands the formats of the messages.

Naturally, the sample traces that are used to infer the language automata must be sufficient and representative enough. Otherwise, the automata may be incomplete and fail to recognize some of the protocol messages of the test cases. To guarantee that the test cases are fully supported, our tool simply adds the messages of the test cases to the sample traces during construction of the input parser.

To extract the testing payloads, the tool feeds the test cases to the language parser and extracts the parameter data, i.e., the content of the fields that are identified as parameters. Normally, the most interesting payloads are those found in the last message of the test cases (the preceding messages act as prelude to take the server to the designated state). However, it happens sometimes that the fault is triggered by a message in the middle or even the beginning of the test case. Therefore, it is preferable to collect the payloads from all messages, than risking neglecting payloads that could detect vulnerabilities.

B. Protocol specification inference

The tool resorts to the same techniques of protocol reverse engineering used in the generic parsers to infer the output protocol specification from a sample of network traces from the target server. As before, the traces only need to have normal protocol sessions.

The reverse engineering process of this component derives the language of the protocol, and in addition infers the state machine of the protocol. The state machine is obtained by identifying the implicit causal relations among the various types of messages as observed in the traces. Naturally, parts of the protocol that are missing from the traces cannot be inferred, and therefore will not be contemplated in the test case generation. The client sessions must therefore exercise all the functionality that one wants to test. This approach supports the creation of test cases for new protocol extensions that may be missing from the standard specifications, simply by including in the traces the client-server messages related to those features. The server implementing the custom modifications (or the features still under development) only needs to be experimented with a client and the messages corresponding to the new extensions have to be captured.

Figure 3 shows the inferred protocol specification from a small trace containing 5 FTP sessions. The reverse engineering process starts by deriving the language automaton, where each path corresponds to a unique message format. Seven distinct FTP types of requests were identified – CDUP and QUIT with no parameters, and USER, PASS, CWD, RNFR and RNTO each with one parameter. Then, the protocol state machine is obtained by converting each client session into a sequence of message formats. Next, an automaton is built to accept the various sequences of messages of each session and to represent any implicit causal relation between message formats. For example, CWD and CDUP messages are used interchangeably after the PASS messages, so they are merged into the same protocol state (*Q2*). On the other hand, since PASS messages always follow USER messages (and not the inverse), and the causal relation is captured with three states ($Q0 \rightarrow Q1 \rightarrow Q2$).

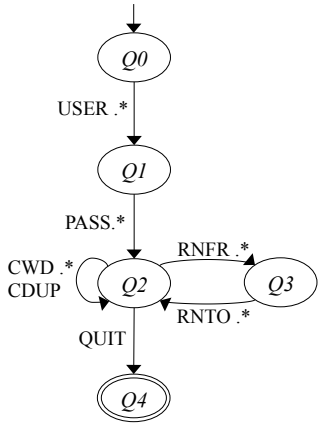
C. Target server transition context

The sample traces from the target server contain messages from the clients that make the server go from the initial state to other protocol states. The information included in these messages consist of valid usernames and passwords, and other parameter data that are specific to the target server. Examples of these parameter values are “clark” and “bruce” in USER messages and “kent” and “wayne” in PASS messages (see Figure 3a). The target server transition context component can take advantage of this data to obtain the necessary information about the server’s configuration.

The component implements a table that correlates the parameter values in the sample traces with the previously

Sample traces (FTP target server)	
Session *S1*	USER clark
	PASS kent
	QUIT
Session *S2*	USER bruce
	PASS wayne
	CWD /home/bruce
	RNFR cave
	RNTO batcave
	QUIT
Session *S3*	USER peter
	PASS parker
	CWD /home/peter
	CWD daily
	CDUP
	RNFR news
	RNTO journal
	QUIT
Session *S4*	USER clark
	PASS kent
	CWD /home/clark
	CDUP
	QUIT
Session *S5*	USER bruce
	PASS wayne
	QUIT

(a) Sample traces of a target server.



(b) Inferred protocol specification.

Figure 3. Reverse engineering the protocol specification.

inferred output protocol specification. The table is built by parsing each message that causes the protocol to make a state transition and extracting the values of each field. Table Ia shows an example for the traces of Figure 3a and using derived protocol specification of Figure 3b. The first two columns have the source and destination states; the third column depicts the message format from the language automaton (here abbreviated as the command name); and the forth column contains the list of observed parameter values and their respective sessions.

The table is then used to build the preludes of the test cases, i.e., the sequences of protocol messages that take the target server from the initial state to any other protocol state. The first step to create a message sequence is to select the shortest path in the protocol state machine from the initial state to the desired state. For instance, the smallest sequence of messages to reach state *Q3* would be: USER .*, PASS .*, and RNFR .*. The second step is to select the correct parameter values, which cannot come from different sessions because this could take the server to an unexpected state. Therefore, the table is searched to find which sessions reach the desired state. When more than one session is found, the one that reaches the desired state with a smaller number of messages is chosen. For instance, session *S2* is selected to go to state *Q3* (*S3* also reaches *Q3*, but only after the six messages). Then, the parameter values of the respective messages are used. In the example, the prelude would be: "USER bruce", "PASS wayne" and "RNFR cave". Table Ib shows the generated preludes, allowing each one of the protocol states to be reached.

Table I
TARGET SERVER TRANSITION TABLE AND PRELUDE GENERATION.

(a) Target server transition table.

From	To	Msg type	Parameter values (from session)
Q0	*Q1*	USER	clark (*S1,S4*); bruce (*S2,S5*); peter (*S3*)
Q1	*Q2*	PASS	kent (*S1,S4*); wayne (*S2,S5*); parker (*S3*)
Q2	*Q3*	RNFR	cave (*S2*); news (*S3*)
Q3	*Q2*	RNTO	batcave (*S2*); journal (*S3*)
Q2	*Q4*	QUIT	none

(b) Generated prelude to test cases.

Dest. state	Preludes (from session)
Q0	none (*initial state*)
Q1	USER clark (*S1*)
Q2	USER clark (*S1*), PASS kent (*S1*)
Q3	USER bruce (*S2*), PASS wayne (*S2*), RNFR cave (*S2*)
Q4	USER clark (*S1*), PASS kent (*S1*), QUIT (*S1*)

D. Test case generator

The test case generator component aims at creating a set of test cases that covers the entire protocol space. So, it exhaustively creates tests for all states and transitions of the output protocol specification. Each test case is composed by a prelude of messages plus a testing message.

The generator iteratively selects a protocol state and one of its transitions. To reach the selected protocol state, the component uses the previously derived prelude of messages. The transition is associated to a message type, so the generator uses the inferred message format for the creation of a testing message. The testing message consists of a variation of the message format with a special payload—one of the extracted testing payloads. Therefore, several test cases can be produced for a single protocol state and transition. The existing implementation resorts to single parameter testing, i.e., only one of the parameters has a testing payload, which is the normal scheme used by most tools (e.g., fuzzers). Naturally, other techniques, such as pairwise or exhaustive testing, could also be programmed.

Table II presents a subset of the generated test cases for the example of Figure 3. The test cases were created for the entire inferred protocol using extracted testing payloads (e.g., "AAAAAA1" or "CCCCCC2"). The tool starts by generating test cases for the initial state, which only accepts

Table II
TEST CASE GENERATION EXAMPLE.

Dest. state	Prelude	Transition		Generated test case
Q0	none (*initial state*)	*USER .**	#1	USER AAAAAA1
			#2	USER AAAAAA2
				...
Q1	USER clark	*PASS .**	#8	USER clark
				PASS AAAAAA1
			#9	USER clark
				PASS AAAAAA2
				...
Q2	USER clark, PASS kent	*CDUP*		none (no parameters)
		*CWD .**	#15	USER clark
				PASS kent
				CWD AAAAAA1
				...
		*RNFR .**	#22	USER clark
				PASS kent
				RNFR AAAAAA1
				...
		QUIT		none (no parameters)
Q3	USER bruce, PASS wayne, RNFR cave	*RNTO .**	#29	USER bruce
				PASS wayne
				RNFR cave
				RNTO AAAAAA1
				...
			#35	USER bruce
				PASS wayne
				RNFR cave
				RNTO CCCCCC2
Q4	USER clark, PASS kent, QUIT	none (final state)		

one transition (or message type) "USER .*". Since this is an initial state, the prelude for these test cases is empty, and therefore they consist of only the testing message (e.g., "USER AAAAAA1"). As this message only has one parameter, the number of test cases that is produced is equal to the number of different testing payloads.

Test cases for transitions of non-initial states require a prelude of messages. For instance, to test the "RNTO .*" transition, the generator would first get the corresponding prelude (i.e., "USER bruce", "PASS wayne" and "RNFR cave"), and then generate several variations of the "RNTO .*" message, each one with a different testing payload (e.g., test cases #29 and #35).

IV. EVALUATION

This section presents the results of the evaluation of our approach by analyzing and comparing it with several commercial and open source fuzzers and vulnerability scanners. Due to time and space restrictions we focused on a well-known protocol, FTP [9], as the target protocol of the generated test cases. FTP has a reasonable level of complexity and there is a rich set of testing tools that support it, so it can clearly illustrate our approach and support our evaluation. However, this approach can be used on other target protocols, as long as their specification is provided or reverse engineered from network traces.

Table III
TEST CASE GENERATION TOOLS FOR FTP PROTOCOL.

Type of tool	Testing tool
Specialized FTP fuzzers	Codenomicon Defensics FTP test suite 10.0
	Infigo FTPStress Fuzzer 1.0
Framework fuzzers (FTP setup)	Metasploit Framework (FTP Simple Fuzzer) 4.0
	FuzzTalk Fuzzing Framework 1.0
	Bruteforce Exploit Detector (BED) 5.0
	DotDotPwn 2.1
	fuzzer.py 1.1
Vulnerability scanners (FTP test cases)	Nessus 4.4.1
	OpenVAS 4
	NeXpose (Community Edition) 2011

Table IV
COVERAGE OF THE FTP PROTOCOL SPACE.

	Basic FTP (RFC959)	Full FTP (10 RFCs)	State machine	Total test cases
Codenomicon	**97%**	61%	**67%**	71658
Infigo	86%	66%	17%	292644
Metasploit	94%	72%	17%	3125
FuzzTalk	91%	**80%**	17%	257892
BED	69%	44%	17%	24918
DotDotPwn	6%	3%	17%	28244
fuzzer.py	63%	44%	17%	197616
Nessus	20%	13%	17%	148
OpenVAS	14%	8%	17%	53
NeXpose	23%	14%	17%	54
Our approach w/ traces from ee.lbl.gov	**97%**	**92%**	**67%**	1463151

A. Protocol Space Coverage

Table III shows the test case generation tools used in the experiments. We chose different types of tools that have varying levels of protocol coverage and payloads: two commercial fuzzers specialized in FTP, five fuzzer frameworks that support FTP out-of-the-box, and three vulnerability scanners with the most up-to-date versions of their vulnerability databases (which included some test cases for previously reported FTP vulnerabilities). All tools were setup to be as complete and thorough as possible, without modifying the default configuration too much (only some minor options that were specific to the server configuration were changed, such as the network address and login credentials).

We analyzed the protocol space coverage achieved by the tools by observing which FTP commands were tested. Columns *Basic FTP* and *Full FTP* of Table IV indicate the percentage of the commands that were tried by the tools when compared to the basic standard RFC 959 and the complete specification (with all known non-obsolete extensions defined in nine extra RFCs), respectively. With regard to Basic FTP, the specialized FTP fuzzers have a high command coverage because they are focused on generating test cases for a single protocol. The vulnerability scanners, on the other hand, have a very low coverage because they create test cases designed to confirm the existence of vulnerabilities that are known to

Table V
TAXONOMY OF PAYLOADS FROM EXPLOITS OF 131 FTP VULNERABILITIES.

Type of payload	Vulns	Test case example	Extrated payload
Arbitrary strings	BO, ID, DoS	`MKD Ax255`	`Ax255`
Strings w/ special prefix	BO, FS, DoS	`LIST ~{`	`~{`
Large numbers	BO, DoS	`REST 1073931080`	`1073931080`
Format strings	FS, DoS	`USER %x%x%x`	`%x%x%x`
Directory structure	BO, FS, ID, DoS	`LIST ../../../`	`../../../`
Known keywords	ID, DoS, DC	`USER test`	`test`
Special constructs	BO, EE	`USER ')UNION SELECT (...)'`	`')UNION SELECT (...)'`

BO: Buffer Overflow; FS: Format String; EE: Ext. App Exec. and SQL injection;
ID:Dir. Traversal and Inf. Disclosure; DoS: Denial of Service and Resource Exhaustion; DC: Default Configuration

exist in specific versions of FTP servers. These test cases thus contain very precise testing payloads, addressing only a small part of protocol space.

Although a tool may have a good basic coverage of the standard, it is important to test the full command space, including protocol extensions and even non-standard commands. This less used part of the protocol is also often less tested, and therefore its implementation might be more prone to vulnerabilities. In effect, Codenomicon's fuzzer, which has the highest coverage of the FTP basic standard (97%), lacks support for many of the extensions (it has 61% of overall protocol coverage). The coverage of the fuzzer frameworks depends on how comprehensive is the setup or template for testing a specific protocol. In our study, FuzzTalk and Metasploit have the highest coverage with 80% and 72%, which implies that 20% or more of the commands are left without being checked. DotDotPwn has a lower protocol coverage because it is designed to search for directory traversal vulnerabilities, and therefore, only path traversal commands are tried (e.g., CWD and RETR). Nevertheless, as we will see in Section IV-D, it produces payloads that are useful to detect other types of vulnerabilities. We also found that none of the tools tested other parameters of the commands, besides the first one (e.g., TYPE A, TYPE E, HELP USER), thus ignoring these parts of the implementation.

We analyzed the coverage of the protocol state machine achieved by the tools, i.e., if they test the commands in their correct state (e.g., trying a RNTO command after sending a RNFR command). Column *State machine* of the table depicts, for the basic RFC 959 specification, the percentage of transitions that change the state of the protocol that are checked by the tools. A value of 17%, for instance, reveals that most tools only support the USER $\rightarrow$ PASS sequence in their test cases. Codenomicon's fuzzer is the only tool that experiments some additional protocol transitions, such as RNFR $\rightarrow$ RNTO, REST $\rightarrow$ STOR, and REST $\rightarrow$ RETR. Even though testing the commands in the wrong state is useful to discover certain kinds of flaws, they should also be tried in their correct state. Otherwise, simple validation mechanisms implemented at the server (e.g., that discard arriving commands unexpected for the current state) could prevent the processing of the malicious payloads, rendering the test cases ineffective in practice.

The last row of Table IV outlines the results of our approach. In these experiments we used a subset of a network trace from 320 public FTP servers located at the Lawrence Berkeley National Laboratory (LBL) over a period of ten days[1]. The full trace contains more than 3.2 million packets exchanged between the servers and thousands of clients. Our tool does not require such a large amount of messages to infer a protocol specification, so we randomly selected a 5 hour period from the LBL trace. Next, we added a trace with the messages collected while the ten tools of Table III tested the FTP server. This second trace provides the messages with the execution context of the server and also complements the LBL trace with unusual message types, increasing the diversity of the exchanges. The complete trace was then used to infer a specification of the FTP protocol and to obtain the transition context of the target server.

The set of test cases generated by our tool covers 97% and 92% of the overall command space for the basic and full FTP specification, which is equal or better than the results achieved by any of the other tools. The second best coverage for the full standard, for instance, was obtained by FuzzTalk with 80% of the overall protocol space. With respect to the state machine coverage, our solution also matches the best coverage of 67%. These results are particularly interesting because our specification is automatically generated from network traces, while the others are manually programmed by experts in the protocol being tested. The downside of our better protocol coverage is a larger number of test cases (see column *Total test cases*). For this experiment, the tool generated over one million test cases as a consequence of the combination of several testing payloads with a more complete protocol coverage.

The overall trace used by our tool contained the most used types of FTP requests as well as messages defined in protocol extensions. An analysis on the command coverage of the trace showed that the LBL trace only covered 45% of the full command space, which was complemented with the messages from the tools' test cases in the second trace, thus resulting in the final coverage of 92%. In any case, the LBL trace contained some unconventional message types that were

[1]http://ee.lbl.gov/anonymized-traces.html

Table VI
POTENTIAL VULNERABILITY COVERAGE OF FTP TEST CASES.

# Vulns and type of payload	Codenomicon	Infigo	Metasploit	FuzzTalk	BED	DotDotPwn	fuzzer.py	Nessus	OpenVAS	NeXpose	**Our approach**
50 Arbitrary strings	94%	94%	94%	94%	94%	94%	94%	94%	10%	10%	94%
10 Strings w/ special prefix	25%	50%	0%	25%	25%	0%	50%	0%	0%	25%	50%
3 Large numbers	100%	100%	0%	100%	33%	0%	100%	0%	0%	0%	100%
11 Format strings	100%	91%	0%	73%	82%	64%	82%	0%	0%	0%	100%
41 Directory structure	76%	71%	0%	61%	27%	63%	54%	2%	0%	5%	90%
14 Known keywords	21%	0%	0%	14%	7%	7%	0%	50%	29%	36%	64%
2 Special constructs	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
131 Potential coverage	**78%**	74%	36%	66%	58%	62%	63%	42%	7%	10%	**88% (actual)**
Total test cases	71658	292644	3125	257892	24918	28244	197616	148	53	54	1463151
Distinct payloads	5554	14116	12	3450	139	10286	1023	45	12	23	23569

not published, such as MACB and CLNT requests. MACB commands are sometimes used by FTP clients running in the Macintosh Operating Systems (e.g., CuteFTP or WebTen) to transfer files in MacBinary mode, while CLNT refers to an obscure feature of a particular FTP client (NcFTP) apparently used to identify it and to access shell utilities. Little more information is available for these two non-standard commands, as they are not specified by any RFC or other official document. This further reveals the importance of using real traces, since it allows our approach to generate test cases that cover otherwise unknown parts of a server implementation.

B. FTP test cases

Another aspect analyzed in our evaluation is the quality of the payloads generated by the tools. In order to measure such attribute, we studied the potential coverage of these payloads in detecting vulnerabilities. Note that this metric is optimistic in the sense that it does not reflect the ability of the tool to detect those vulnerabilities, but rather how many vulnerabilities could be detected by these payloads *if applied to the correct commands in the right state*. For instance, a tool may be using a rich set of payloads, but if it does not generate test cases for the vulnerable commands, it will not be able to detect them.

To perform this experiment, we exhaustively searched the web for all known FTP vulnerabilities and obtained the respective exploits (i.e., the FTP command and the payload). We retrieved a total of 131 known vulnerabilities and classified them according to the type of payloads that were used in the exploits. In total, seven classes of payloads were defined, covering six types of vulnerabilities, from buffer overflows to SQL injection (see Table V for examples).

Table VI depicts the coverage of the payloads of the test cases generated by the tools with respect to the 131 vulnerabilities. Only the specialized fuzzers have a potential vulnerability coverage of over 70%, whereas the fuzzer frameworks achieved values between 36% and 66%. The vulnerability scanners posses the lowest coverage of 7% and 10%, with Nessus as exception with 42%, due to its high coverage in buffer overflow payloads and known keywords. In many cases, vulnerability scanners use passive test cases that only obtain the welcome banner of the server to check if the reported version is known to be vulnerable. This type of test case does not aim at triggering vulnerabilities and therefore its payload has a null potential vulnerability coverage.

Naturally, the highest potential vulnerability coverage was obtained by our tool since it extracts the testing payloads of all the input test cases. This result shows that our approach effectively combines the test cases from different sources to produce a new set of test cases with a higher vulnerability coverage than even the best specialized testing tools. Furthermore, given that our test cases had a very high protocol space coverage (92%), the *actual* vulnerability coverage should be very close to the observed potential vulnerability coverage (88%). See Section IV-D for an experiment that gives evidence to confirm this observation.

An increased vulnerability coverage comes at the expense of executing a larger number of test cases, which may be too costly in some scenarios. We believe, however, that in security testing maximizing the vulnerability coverage is crucial. In any case, a good coverage can still be achieved with our approach by resorting only to the test cases of two tools, as Section IV-D will show. Moreover, some kind of prioritization could be used to selected the actually tried test cases, therefore reducing the number of tests to some desired level, but this is left as future work.

C. Non-FTP test cases

This experiment studies the effectiveness of our approach when it resorts to test cases designed for implementations of other protocols. Table VII presents the ten sources of test cases for non-FTP protocols that were used. It includes fuzzer frameworks and vulnerabilities scanners that create test cases for the SMTP [11] and POP3 [12] protocols, as well as two plain payload generators, including the popular fuzz program from 1990 [3].

Table VIII depicts the potential vulnerability coverage of the extracted payloads. As with the previous experiment, our generated test cases got the best vulnerability coverage (83%). In fact, these test cases even obtained a better potential vulnerability coverage then the best specialized FTP

Table VIII
POTENTIAL VULNERABILITY COVERAGE OF NON-FTP TEST CASES.

	SMTP					POP			Plain payloads		FTP
# Vulns and type of payload	BED	Metasploit	FuzzTalk	sfuzz	Nessus	BED	sfuzz	Nessus	DotDotPwn	fuzz	**Our approach**
50 Arbitrary strings	94%	76%	94%	96%	94%	94%	94%	94%	94%	76%	96%
10 Strings w/ special prefix	25%	0%	25%	0%	0%	25%	0%	0%	0%	75%	75%
3 Large numbers	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	100%
11 Format strings	82%	0%	73%	100%	0%	82%	100%	82%	64%	100%	100%
41 Directory structure	27%	0%	0%	0%	0%	27%	0%	0%	63%	80%	88%
14 Known keywords	7%	0%	0%	0%	0%	7%	0%	0%	0%	14%	14%
2 Special constructs	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
131 Potential coverage	57%	29%	45%	45%	36%	57%	44%	43%	61%	**71%**	**83% (actual)**
Total test cases	34288	1300	106136	3660	13	16195	676	367	6984	150370	9939567
Distinct payloads	1087	304	303	156	29	193	676	7	6984	150370	160109

Table VII
TEST CASE GENERATION TOOLS FOR NON-FTP PROTOCOLS.

Target protocol	Testing tool
SMTP	Bruteforce Exploit Detector (BED) 5.0
	Metasploit Framework (SMTP Simple Fuzzer) 4.0
	FuzzTalk Fuzzing Framework 1.0
	Simple Fuzzer 0.6.2 (sfuzz)
	Nessus 4.4.1
POP3	Bruteforce Exploit Detector (BED) 5.0
	Simple Fuzzer 0.6.2 (sfuzz)
	Nessus 4.4.1
Plain payloads	DotDotPwn 2.1
	Fuzz Generator (fuzz)

testing tools, such as Codenomicon's or Infigo's fuzzers, with 78% and 74% respectively.

This result indicates that recycling the test cases is helpful because the payloads from one protocol can also be successfully used in other protocols, since the classes of vulnerabilities they might experience are relatively similar (and therefore, they can be exploited in equivalent ways). The table also suggests that using other sources of test cases can contribute to a richer set of payloads. For instance, fuzz, which is a simple random payload generator, has a high potential coverage for FTP vulnerabilities.

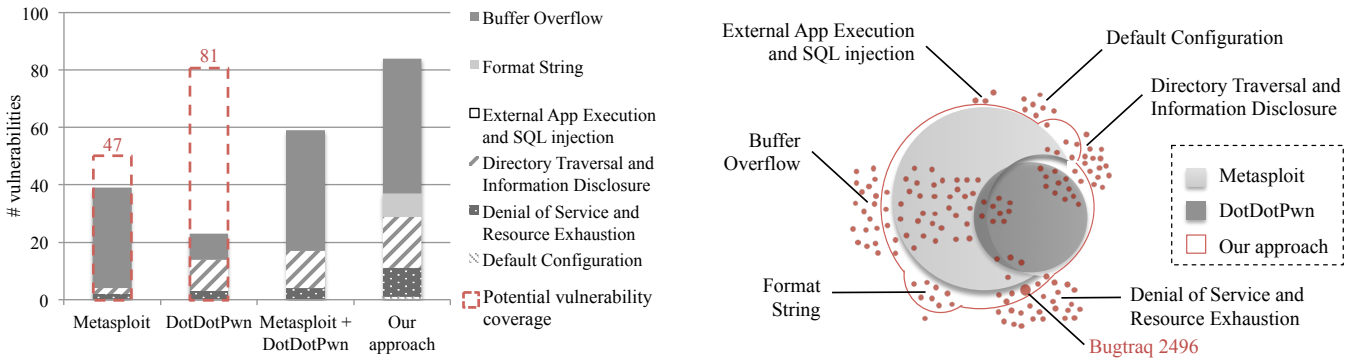
D. Case study with two FTP fuzzers

This experiment provides a closer look at how our approach can create test cases with greater coverage than the original input test cases. To better illustrate this, we restricted the sources of input test cases for our tool to two testing tools with very different values of protocol space and potential vulnerability coverage. We chose the FTP fuzzer from the Metasploit Project, which has a protocol coverage of 72% and a potential vulnerability coverage of 36%, and the DotDotPwn, a fuzzer specialized at directory traversal vulnerabilities, which has the lowest protocol coverage of 3%, but a high potential vulnerability coverage of 62%.

We analyzed the test cases produced by the tools and evaluated their *true vulnerability coverage*. One way to calculate this coverage is by trying the tools with the various FTP servers to determine if they discovered the corresponding vulnerabilities. Unfortunately this is not feasible to do in practice because several of the vulnerable versions of the servers are no longer available, either because they are deleted from the web to avoid the dissemination of the flaws or because they are no longer distributed by the commercial software vendors. Additionally, in other cases, some of the support software (operating system and/or specific libraries versions) is also missing, since some of the vulnerabilities have a few years.

In alternative, we calculated the true vulnerability coverage by inspecting manually the generated test cases to find out if one of them was equivalent (from a testing perspective) to the exploit of the vulnerability. Figure 4a shows the actual coverage of the two fuzzers and our tool. We can see that neither fuzzer achieved their full potential vulnerability coverage. Metasploit's fuzzer, for instance, is able to detect 39 vulnerabilities, although it has a potential vulnerability coverage of 47 vulnerabilities (36%). DotDotPwn is a good example of a tool that even though it produces good payloads (with a potential vulnerability coverage of 62%), it is only able to cover 23 vulnerabilities (only 18% of the total number of vulnerabilities).

The graph further reveals the gain of combining both tools to detect vulnerabilities, where together they can discover 59 FTP vulnerabilities (about 45% of the total number). Our tool, however, generated a set of test cases with a coverage of 64% of the known FTP vulnerabilities. This shows that our approach can cover an additional 25 vulnerabilities that none of the tools created test cases for. Figure 4b displays a representation of the coverage of the FTP vulnerabilities by Metaploit's FTP fuzzer, DotDotPwn and our own approach. Each dot corresponds to one of the 131 vulnerabilities, and vulnerabilities of the same type are graphically depicted near to each other. The area of the circles depicts the number and type of vulnerabilities found by each tool. For example, there are 65 dots for the buffer overflow type of vulnerability, where 35 were found by Metasploit, 9 by DotDotPwn, and 47 were disclosed by our tool (an additional 5 when compared with using both tools), while 18 still remained undetected



(a) True vulnerability coverage.

(b) Representation of the vulnerability coverage.

Figure 4. Analysis of the true vulnerability coverage of experiment 3.

from all tools.

The figure shows that the Metasploit's FTP fuzzer has a much larger coverage than DotDotPwn, circumscribing many vulnerabilities. Nevertheless, DotDotPwn can cover many vulnerabilities that Metasploit cannot, such as directory traversal and information disclosure and buffer overflow. Our approach is depicted by the line surrounding both circles. The area delineated by this line is larger than the union of both circles because it combines a much better protocol space coverage with the payloads of both tools. As a last experiment to help to confirm this hypothesis, we actually installed the open source server wu-ftp 2.6.0 that is known to contain a denial of service vulnerability (bugtraq 2496). Then, we used the three tools to test it, and found out that both Metasploit and DotDotPwn fuzzers were incapable of finding the flaw, but our tool was successful.

V. Related Work

The paper describes a methodology and a tool that automates the process of generating test cases. In different contexts, other approaches also perform systematic and automatic generation of test cases, however, to the best of our knowledge there is no approach that reuses and combines different test cases.

Fault Injection is an experimental approach for the verification of fault handling mechanisms and for the estimation of various parameters that characterize an operational system [13], [14]. Traditionally, fault injection has been utilized to emulate hardware and software faults, ranging from transient memory corruptions to permanent stuck-at faults (see for instance [15], [16]). The emulation of other types of faults has also been accomplished with fault injection techniques, such as software and operator faults [17], [18].

Robustness Testing Mechanisms study the behavior of a system in the presence of erroneous input conditions. Their origin comes both from the software testing and fault-injection communities, and they have been applied to various areas, such as POSIX APIs and device driver interfaces [19], [20]. However, due to the relative simplicity of the mimicked faults, it is difficult to apply these tools to discover security vulnerabilities.

Fuzzers deal with this complexity by injecting random samples as input to the software components. For example, Fuzz [3] generates large sequences of random characters to be used as testing parameters for command-line programs. Many programs failed to process the illegal arguments and crashed, revealing dangerous flaws like buffer overflows. By automating testing with fuzzing, a significant number of interesting input permutations can be tried, which would be difficult to write as individual test cases [21], but with the cost of only exercising a random sample of the system behavior. Throughout the years fuzzers have evolved into more intelligent vulnerability detectors by exploring knowledge about the system under test [22], [4], [5], [23], [24].

Fuzzing Frameworks have been developed to ease the process of tailoring fuzzing tools to specific systems [2], [6], [7]. Metasploit is one of these frameworks that provides support for the creation of new testing modules [2]. It allows the tester to define the complete testing process, from the payload generation to the actual test case execution and interaction with the target system.

Vulnerability Scanners are a practical type of testing tools whose purpose is the discovery of *known* vulnerabilities [1], [25], [26], [27], [28]. These tools resort to a database of previously reported vulnerabilities and attacks that allow their detection. One entry in this database typically contains the server type and version where the vulnerability is known to exist, as well as the test case to detect it. Usually, this test case can only be used to confirm the existence of that vulnerability on that particular version of the server, and it cannot be used on other kinds of servers, such as those that implement other protocols. The analysis of a system is usually performed in three steps: first, the scanner interacts with the target to obtain information about its execution environment (e.g., operating system and available services); then, this information is correlated with the data stored in the database, to determine which vulnerabilities have been

observed in this type of system; then, the scanner performs the corresponding attacks and presents statistics about which ones were successful. Even though these tools are extremely useful to improve the security of systems in production, they have the limitation of being unable to uncover new vulnerabilities.

VI. Conclusion

This paper proposes to recycle existing test cases from several sources, including those designed to test implementations of other protocols. The approach automatically extracts the malicious payloads of the original test cases, and then reuses them in a new set of test cases. These new test cases are generated based on a protocol specification inferred from network traces of the target server, allowing the inclusion of not only the normal protocol exchanges but also extensions and non-standard features. The experiments with ten FTP testing tools and ten other sources of test cases for non-FTP protocols, show that in both cases our tool is able to get better or equal protocol and vulnerability coverage than with the original test cases. In a experiment with the Metaploit and DotDotPwn FTP fuzzer frameworks, our tool showed an improvement of 19% on the vulnerability coverage when compared with the two combined fuzzers, being able to discover 25 additional vulnerabilities.

Acknowledgements*:* This work was partially supported by EC through project FP7-257475 (MASSIF) and by FCT through the Multi-annual Program and project PTDC/EIA-EIA/100894/2008 (DIVERSE).

References

[1] Tenable Network Security, "Nessus vulnerability scanner," 2008, http://www.nessus.org.

[2] Rapid7, "Metasploit project," 2011, http://www.metasploit.com.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[4] Codenomicon, "Defensics X," 2011, http://www.codenomicon.com.

[5] Infigo Information Security, "Infigo FTPStress Fuzzer," 2011, http://www.infigo.hr/.

[6] AutoSec Tools, "Fuzztalk fuzzing framework," 2011, http://www.autosectools.com/Page/FuzzTalk-Guide.

[7] C. Navarrete and A. Hernandez, "DotDotPwn," 2011, http://dotdotpwn.blogspot.com/.

[8] Internet Engineering Task Force, "RFC Editor," 2011.

[9] J. Postel and J. Reynolds, "File Transfer Protocol," RFC 959, 1985.

[10] J. Antunes, N. Neves, and P. Verissimo, "Reverse engineering of protocols from network traces," in *Proc. of the Working Conf. on Reverse Engineering*, 2011.

[11] J. Klensin, "Simple Mail Transfer Protocol," RFC 5321, 2008.

[12] J. Myers and M. Rose, "Post Office Protocol - Version 3," RFC 1939, 1996.

[13] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Trans. on Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[14] M. Hsueh and T. Tsai, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[15] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software fault injection and monitoring in processor functional units," in *Proc. of the Int. Working Conf. on Dependable Computing for Critical Applications*, 1995, pp. 135–149.

[16] T. Tsai and R. Iyer, "Measuring fault tolerance with the FTAPE fault injection tool," *Quantitative Evaluation of Computing and Communication Systems*, pp. 26–40, 1995.

[17] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults," in *Proc. of the Int. Symp. on Fault-Tolerant Computing*, 1996, pp. 304–313.

[18] J. Durães and H. Madeira, "Definition of software fault emulation operators: A field data study," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2003, pp. 105–114.

[19] P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *Proc. of the Int. Symp. on Fault-Tolerant Computing*, 1999, pp. 30–37.

[20] M. Mendonça and N. Neves, "Robustness testing of the windows DDK," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2007, pp. 554–564.

[21] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security and Privacy*, vol. 03, no. 2, pp. 58–62, 2005.

[22] University of Oulu, "PROTOS – security testing of protocol implementations," 1999–2003, http://www.ee.oulu.fi/research/ouspg/protos/.

[23] M. Sutton, "FileFuzz," 2005, http://labs.idefense.com/labs-software.php?show=3.

[24] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[25] Greenbone Networks GMBH, "Openvas," 2011, http://www.openvas.org/.

[26] Rapid7, "NeXpose," 2011, http://www.rapid7.com.

[27] Saint Corp., "SAINT network vulnerability scanner," 2008, http://www.saintcorporation.com.

[28] Qualys Inc., "QualysGuard enterprise," 2008, http://www.qualys.com.