Dartmouth Computer Science Technical Report TR2008-634

LZfuzz: a fast compression-based fuzzer for poorly documented protocols

Sergey Bratus, Axel Hansen, Anna Shubina[1]

Department of Computer Science

Dartmouth College, Hanover, NH

September 16, 2008

[1]This work was supported in part by the National Science Foundation, under grant CNS-0524695. The views and conclusions do not necessarily represent those of the sponsors.

Abstract

Computers make very fast, very accurate mistakes.

From a refrigerator magnet.

Real-world infrastructure offers many scenarios where protocols (and other details) are not released due to being considered too sensitive or for other reasons. This situation makes it hard to apply fuzzing techniques to test their security and reliability, since their full documentation is only available to their developers, and domain developer expertise does not necessarily intersect with fuzz-testing expertise (nor deployment responsibility). State-of-the-art fuzzing techniques, however, work best when protocol specifications are available. Still, operators whose networks include equipment communicating via proprietary protocols should be able to reap the benefits of fuzz-testing them.

In particular, administrators should be able to test proprietary protocols in the absence of end-to-end application-level encryption to understand whether they can withstand injection of bad traffic, and thus be able to plan adequate network protection measures. Such protocols can be observed in action prior to fuzzing, and packet captures can be used to learn enough about the structure of the protocol to make fuzzing more efficient.

Various machine learning approaches, e.g. bioinformatics methods, have been proposed for learning models of the targeted protocols. The problem with most of these approaches to date is that, although sometimes quite successful, they are very computationally heavy and thus are hardly practical for application by network administrators and equipment owners who cannot easily dedicate a compute cluster to such tasks.

We propose a simple method that, despite its roughness, allowed us to learn facts useful for fuzzing from protocol traces at much smaller CPU and time costs. Our fuzzing approach proved itself empirically in testing actual proprietary SCADA protocols in an isolated control network test environment, and was also successful in triggering flaws in implementations of several popular commodity Internet protocols. Our fuzzer, *LZfuzz* (pronounced “lazy-fuzz”) relies on a variant of Lempel–Ziv compression algorithm to guess boundaries between the structural units of the protocol, and builds on the well-known free software *GPF* fuzzer.

Section 1

Introduction

Fuzzing has become a popular method for software testing and vulnerability discovery. It is very well-suited for "black box" and "grey box" settings, in which the source code of an application or an OS component is not available, and potentially much more powerful source code-based analysis tools cannot be brought to bear. At the core of the fuzzing approach is generation of crafted input intended to trigger faults in the receiving software.

A typical fuzzer generates a broad range of malformed inputs as fast as they can be accepted by the targeted software, creating them by superimposing some pre-programmed "basic" flaws. These combinations of injected flaws are "random" (more precisely, stochastic, according to a particular generative model), but if the software does any kind of input sanity checking, should mimic valid inputs closely enough to pass muster.

The underlying intuition that "random" inputs can create execution conditions unforeseen by software developers and testers has been amply validated empirically, but construction of efficient fuzzers still remains an art rather than science.

The challenge of fuzzing. The challenge of generating fuzzed input is fundamentally that of producing input faulty enough to trigger flaws, yet well-formed and "normal" enough to first pass the target code's initial sanity checks and then to cause faults during the actual processing. Also, once flaws in early stages of processing are uncovered, it may be preferrable for the crafted inputs to be "normal" enough to get to later stages in the application to probe their logic in turn.

Thus either pre-programmed knowledge about the target protocol or some form of machine learning of its features is necessary.

Indeed, as we describe below, state-of-the-art fuzzing frameworks (e.g., *Sulley*[1]) use block-level descriptions of targeted protocols as generative models to produce their crafted inputs. Absence of protocol specifications detailed enough to derive an

[1]At the time of writing available from `http://www.fuzzing.org/fuzzing-software`

idea about the basic blocks or units of the protocol creates a problem that must be solved before efficient fuzzing becomes possible.

Why LZfuzz is useful. The problem of fuzzing plain-text proprietary protocols is not as artificial as one might think. Real world infrastructure provides many examples of protocols that were not designed to provide end-to-end confidentiality or integrity protections, such security goals being outside of the developer's scope, or assumed to be taken care of by the products' environment.

Also, retrofitting encryption and cryptographic-based integrity protections to software that must be operated under changing environment security assumptions is not easy, for various reasons. It may require rewriting substantial portions of code; it also opens the "can of worms" that is cryptographic key management.

As a result, fully or partially plain-text protocols remain in operation, and must be protected by other means such as end-to-end encryption in lower network layers and other network security measures. In order to better understand the actual risks, network administrator should be able to find out just how brittle their protocols are. In particular, a means of fuzz-testing proprietary protocols without specifications is desirable.

Why fuzzers must adapt to their targets. It is notable that, despite a substantial number of free and commercial fuzzers available, and the relatively high profile of fuzzing as a software testing technique, it continues to prove an amazingly fruitful technique for finding new exploitable software faults. Although this continued success of adversarial fuzzing might be explained away by the failure of software vendors to apply similar techniques at the QA phase of their product development cycle, reports appear to contradict this explanation.

We must assume, therefore, that most fuzzing tools in actual use by software authors in fact cover rather narrow classes of all possible faulty inputs, so that introduction of new tricks into the generation process tends to pay off immediately and dramatically in new flaws that previous generators had failed to locate.

Of course, diagnosing a triggered flaw as an exploitable vulnerability and developing it into an actual exploit requires substantial investment in the corresponding vulnerability development framework that involves debugger integration, binary code instrumentation, analysis tools, etc., and indeed a lot of effort has been invested into building such frameworks (see, e.g., [sul, aut, Rit07]). We note that there has been great progress and a number of novel technical solutions in this area; however, in this paper we concentrate on the issue of fuzzed input generation alone. We note that in control networks a simple denial of service condition caused by crashes and restarts of vulnerable processes is likely to be a bigger concern than in other kinds of networks.

On evaluating fuzzer efficiency. Emprically, a fuzzer's worth is "proved" by its ability to successfully induce faults in the target processes, or, simply put, to crash its targets. Beyond such empirical evidence, the quality of the fuzzer's generation component is very hard to measure. Towards the end of this paper we discuss how our fuzzing approach can be compared with more precise and computationally demanding ones.

A reasonable theoretical measure of fuzzer quality would be its ability to trigger all existing vulnerabilities;[2] as such, it is impossible to even approximate it. Other measures, such as code coverage, i.e., the portion of the target code actually executed when processing the fuzzed input, have been proposed and implemented, but they characterize only isolated aspects of the fuzzer's generative behavior that affect its success only under rather strong assumptions. For example, "covered" code may have been executed with benign data, so the fact that it has been reached does not necessarily mean that it was reached with the right data to trigger its flaws; still, the assumption that less tested code is more likely to contain trivially triggered errors is quite realistic, and thus the metric is useful.

In the absence of a clear metric, development of fuzzer's generation components is driven by intuition and the apparent empirical yield of found flaws. Such, too, is the nature of our argument in this paper: we build a fuzzer that attempts to extract information about the "tokens" of a protocol from a packet capture and argue that the results of the dissection it performs are useful for subsequent fuzzing. We compare our dissection results with those obtained through a much more computationally demanding bioinformatics method described in [Bed] and introduce a metric that captures our intuition on why our dissector, while somewhat less accurate, should perform comparably, while being much faster.

[2]Arguably, an even more useful measure would also take into account likelihood and impact of vulnerabilities.

Section 2

Block-based fuzzing for proprietary protocols

Block-based fuzzing of well-specified protocols. When protocol specifications are available, the *block-based* fuzzer architecture has proved to be the most effective and popular one. In this architecture, crafted input is modeled and generated as a sequence of byte-array blocks corresponding to the structural units of the actual protocol. These blocks are filled randomly by the fuzzer code with values from non-uniform distributions of protocol field values that are deemed more likely to trigger a fault (such as big integers close to `MAXINT` for a fixed length interger field).

The knowledge about the protocol is expressed in the specification of block types and relations between blocks (e.g., one block can be specified to be filled with the size or a control sum of another).

Dave Aitel's *SPIKE*[1] has been a very successful example of this architecture; the authors of the state-of-the-art *Sulley*[2] recognize it and build on this design, as do some older fuzzers such as *Peach*[3].

From the object-oriented point of view, a block-based fuzzer represents the hierarchical units of the target protocol as a composition of objects corresponding to the atomic units such as integers of various fixed widths or byte arrays of varying width. Each such object provides a method for traversing either all possible values of the respective protocol field, or only values that are likely to cause faults either by themselves or in combination with others. In the parlance of "design patterns", these block objects are *iterators* over some probability distributions used to fill the corresponding fields.

The generation part of the fuzzer itself is built as a composition of these objects,

[1]At the time of writing available from `http://www.immunitysec.com/resources-freesoftware.shtml`

[2]At the time of writing available from `http://www.fuzzing.org/fuzzing-software`

[3]`http://peachfuzzer.com/`

and operates as an aggregate "iterator" over a set of malformed inputs indended to induce faults. Clearly, this design is only possible when the object compositions in question express the relevant parts of the protocol specification.

Fuzzing without protocol specification. Although quite powerful, this architecture assumes considerable knowledge of protocol internals, mostly unavailable in case of proprietary protocols. Assuming that the protocol's connections can be captured and either replayed or modified (e.g., that there are no strong cryptographic integrity and authentication protections on it), one can start with heuristics that attempt to guess the boundaries and types of those blocks that would be most amenable to fuzzing, and fuzz them.

This latter approach is demonstrated by the *General Purpose Fuzzer* (GPF).[4] GPF heuristically partitions a TCP session reconstructed from a packet capture into "tokens", such as apparent ASCII strings, and produces the fuzzed input by applying various token-specific transformations, such as inserting large runs of ASCII characters inside these suspected strings and adding random combinations of delimiters where a delimiter is detected, whereas apparent binary fields are subjected to mutations such as bit flips. The transformations for a particular token are chosen randomly, from a series of hard-coded distributions that are different for each guessed token type. To get the fuzzed input past "sanity checks" such as checks of known checksum fields, the user is given the capability of adding custom fix-up functions, applied successively after all mutations take place.

GPF's approach, although definitely useful in the initial phases of protocol testing, is clearly limited, since it discards most of the information available from packet captures of plain text-based protocols (the *evolutionary fuzzing* extention of GPF compensates for it in a different way).

Bioinformatics connections. Several research projects, notably *PI* [Bed] and *PROTOS* [HVLR06] (see also the overview in [SGA07]) attempt to extract information about the protocol structure from captured traffic. Their application of bioinformatics methods such as dynamic programming sequence alignment algorithms and philogenetic trees to protocol dissection is fascinating and, intuitively and anecdotally, appears very promising.

We note, however, that these algorithms require many tens of hours of CPU time even on relatively small traffic samples.[5] The capability to process large packet samples is desirable in order to build better models of a protocol, if only because they provide more accuracy for estimating distributions of its field values (or, simply, may indicate variability of a field that would otherwise appear constant in a small

[4]Available from `http://www.vdalabs.com/tools/efs_gpf.html`, together with research presentations outlining its further development.

[5]For example, analysis of a sample of 550 ICMP packets using *PI* took over 6 hours on a 600Mhz Intel Coppermine processor.

sample). Unfortunately, the runtime requirements of bioinformatics methods are, generally speaking, exponential in CPU and RAM, although they can be reduced with various heuristics.

Moreover, it is not clear how much of the information derived by bioinformatics methods can be effectively used for fuzzing proper. Whereas intuitively finer protocol dissection is better, fuzzed fault injection into captured or proxied data may be effective with much rougher generative models than those used by geneticists (after all, the injected faults are usually quite simplistic).

Our hypothesis and approach. We hypothesize that a simpler class of fast captured data-driven algorithms can produce protocol models effective for use in fuzzers. Our key intuition that we quantify in Section 4 is that not all mistakes made by a protocol dissector in the process of constructing a generative model of the protocol are equally harmful to fuzzing efficiency. In other words, a fast rougher dissector can be just as good as a slow, more accurate one. Although it is hard to quantify the overall "quality" of a fuzzer, and thus to validate this claim exhaustively, we feel that fast, simple dissectors are worth studying.

In this paper, we present a simple fuzzer that uses the string table generated by running the Lempel–Ziv compression algorithm over the protocol payload to guess the field boundaries and structural units of an unknown protocol. Although obviously prone to errors, it performed well in our tests, causing faults in targeted software.[6]

[6]We used our fuzzer in the course of testing proprietary production SCADA protocols. Unfortunately, non-disclosure agreements prevent us from sharing the specific results of this testing.

Section 3

LZfuzz in operation

Our fuzzer operates as a fuzzer-in-the-middle (FITM) proxy fuzzer, taking advantage of the plaintext nature of the target protocols. In our experiments, we used an ARP poisoning technique[1] to intercept the IP packets exchanged between the communicating parties. This setup was necessitated by our testing circumstances: we could not instrument the systems at one or both endpoints of fuzzed communications, since we were not allowed to modify them.

The packets were then segmented into "tokens" by the Lempel–Ziv compression process, running with a pre-collected string table. The string table was derived from a compression pass on the previously captured data, with the shortest and the rarest strings optionally removed. The tokens were then either randomly replaced according to a probability table, or fuzzed with the standard GPF fuzzing operations. The GPF was accordingly modified to accept and handle the tokenized input rather than the assembled TCP stream (its normal mode of operations). The packets were then re-assembled and sent to the original destination. Figure 3.1 illustrates these operations and Figure 3.2 details the architecture of the tool.

At its simplest, our fuzzer operated on individual packets; however, the underlying "tokenization" and GPF-based token mutation can use a buffer that collects larger parts of the stream than packets prior to mutation and replay. We note that for many protocols an Ethernet packet's length is enough to transmit whole commands and data units.

The string table produced during the training run phase (in which the fuzzer acted as a simple man-in-the-middle forwarder and did not introduce any modifications into the forwarded data) accumulates the frequently repeated byte substrings, which – we hope – correspond to protocol tokens. The results of compression other than the resulting partition of the input stream are discarded.

Of course, the process of filling the Lempel–Ziv string table is rather random: for example, in its beginning when the table is empty, a number of shorter strings will be inserted into it, in the order they are encountered and in no clear relation to

[1]Specifically, the *arp-sk* tool, explained and available at `http://sid.rstack.org/arp-sk/`.

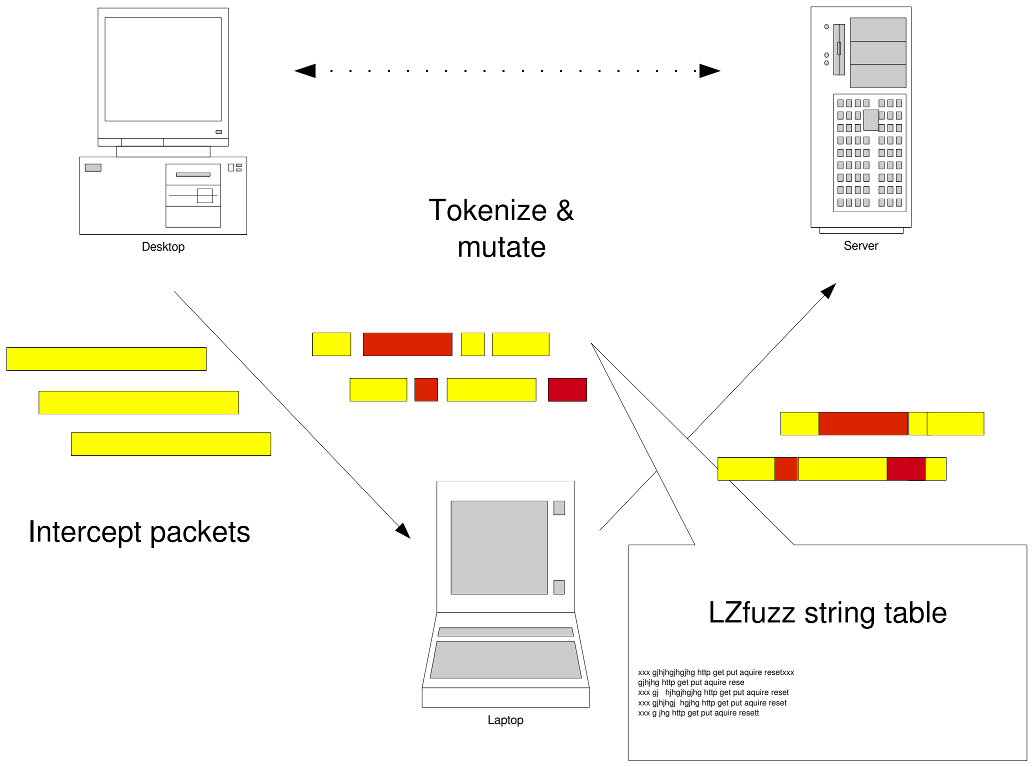
Desktop
Server
Tokenize & mutate
Intercept packets
Laptop
LZfuzz string table
xxx gjhjhgjhgjhg http get put aquire resetxxx
gjhjhg http get put aquire rese
xxx gj   hjhgjhgjhg http get put aquire reset
xxx gjhjhgj hgjhg http get put aquire reset
xxx g jhg http get put aquire resett

Figure 3.1: LZfuzz in operation

their prevalence in the subsequent stream. We use various heuristics to somewhat decrease the effect of this "warm-up" phase, such as supressing shorter or infrequent tokens from the string table used in the fuzzing phase.

This approach can be compared to that *PI* of Beddoe [Bed], where bioinformatics methods are applied to dissecting the protocol into constant and "mutating" tokens, the former assumed to be elements of the protocol's syntax. The resulting dissection can then be used to generate fuzzed input that generally conforms to the protocol's format. A similar idea of *protocol genes* is proposed in [HVLR06].

Our dissection is especially prone to off-by-one kind of alignment errors, in which the token boundaries produced by the compression algorithm are shifted with respect to those of the actual protocol fields, because of the frequently observed combinations of bytes (e.g., in the case of a frequent cross-boundary byte bigram, due to frequent co-occurrence of the ending byte and the leading byte of two adjacent multi-byte fields). We note, however, not all of these errors are equally damaging to the fuzzer's generative model. We compare our dissections with those produced by *PI* in Section 5.

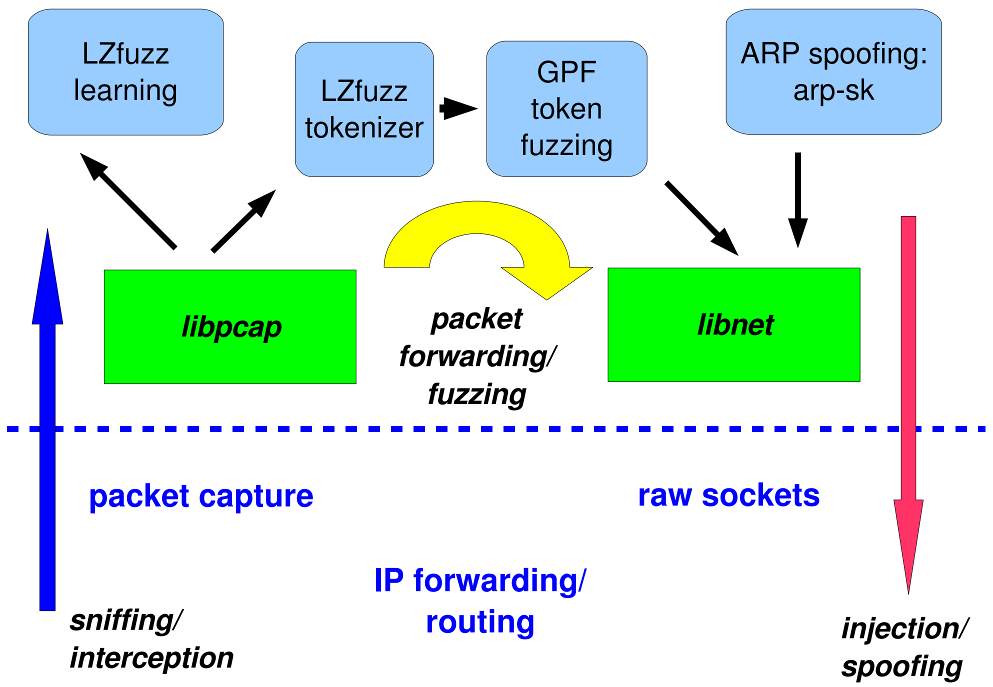

Figure 3.2: LZfuzz architecture

Section 4

Fuzzing dissection quality metric

In this section we define a metric that captures our intuition regarding the dissection part of protocol modeling for producing fuzzed inputs. This metric depends on several simplifying assumptions, listed below, which arguably ignore much the complexity of real protocols. However, we note that existing fuzzing practices achieved their impressive results while relying on much rougher assumptions, and also that, even in the more general machine learning domains such as Bayesian models, strong independence assumptions were not found to perform necessarily inferior to more complex ones (see, e.g., [FGG97]).

A natural choice for a protocol dissection metric is its precision/recall score: the ratio of correct field boundary guesses to the total number of guesses vs. the ratio of correct guesses to the total number of actual boundaries. It is given in Tables 5.2–5.5 for two samples of well-known protocols. We also give the same scores with allowances of 1 and 2 bytes.

However, not all alignment errors counted by these metrics are equally detrimental for the end-goal, generation of effective fuzzed input. We propose to weigh them differently, *depending on the observed variation of the fields' contents*, based on the following assumption:

Assumption. Protocol fields that show high variability in packet captures are likely to be associated with the code paths executed more frequently and in more diverse environments. As a result, these code paths are likely to have been more throughly tested and debugged. Conversely, non-constant fields that show less variability are more likely to be processed by less frequently exercised codepaths, and therefore a more likely place to find undetected flaws.

Accordingly, for each actual protocol field F_i we compute $H(F_i)$, the entropy of the frequency distribution $D(F_i)$ of the distinct values of F_i observed over the training packet capture T (reference protocol parses were produced by the dissector plugins of the *Wireshark* free software network analyzer[1]). We take this entropy as a measure of variability of the field and normalize it by either the overall bit length of the field for fixed-length fields or the longest observed field length for variable-length fields. We attach the coefficient

$$q_i = 1 - \frac{H(F_i)}{\text{Maxlen}_{D(F_i)}(F_i)}$$

to the alignment errors for the field F_i. Note that q_i is close to 0 when the distribution $D(F_i)$ is close to uniform and tends to 1 when the entropy tends to 0.

Figure 4.1 visualizes the application of this metric applied to the results of dissecting a set of ICMP packets. It compares Wireshark's packet dissections (top bar of each pair) with our LZ-based dissection (bottom bar of each pair). The true dissections' fields are colored in grayscale according to the entropy of their observed frequency distribution normalized by length (darker for higher entropy), the guessed fields merely alternate in color. Note that the darkest actual field is the ICMP checksum, to be fixed by the user-defined fix-up before replay. Also, note the constant (white) second byte, which did not change across the training capture set; as a result, it is not detected by our dissector and gets included into larger guessed tokens.

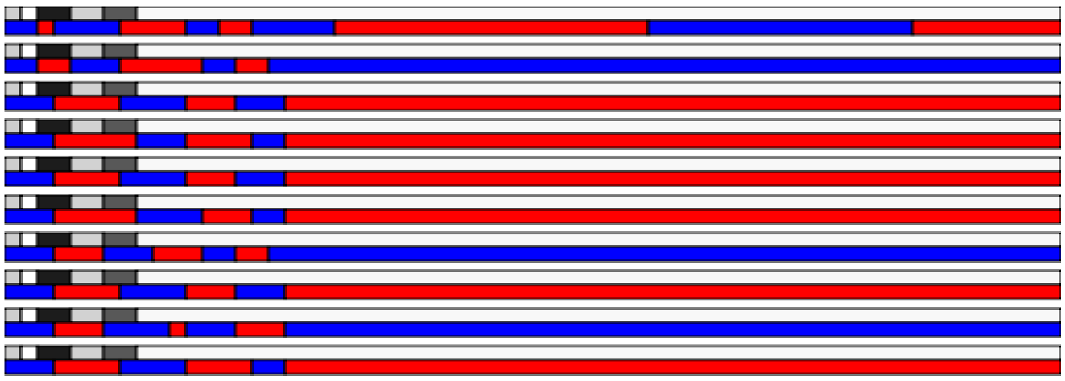

Figure 4.1: ICMP packet set: actual fields vs LZfuzz dissection. Actual ICMP fields shown in gray scale, LZfuzz partitions in alternating red and blue.

As a dissection quality metric, this formula, of course, relies on the availability of the correct dissection (e.g., produced by a protocol analyzer such as Wireshark). However, it has another important application for fuzzing proper: applied to *hypothesized* tokens, it can select the priority order in which these tokens are to be fuzzed. By the fundamental assumption above, tokens that show less entropy across the training set are better targets for fuzzing.

[1]Available from `http://www.wireshark.org/`, formerly known as *Ethereal*.

Section 5

Measurements

In this section we compare the dissection errors made by our LZfuzz token dissector and the *PI* dissector on captures of two different protocols, ICMP and HTTP, ICMP, the Internet Control Message Message Protocol, is a network-layer binary protocol used by Internet routers and hosts to test connectivity and report error conditions, whereas HTTP is an application-level text-based protocol, representative of complex plain text application protocols.

We chose them as representatives of the "binary" and "text-based" classes of protocols. The ICMP packet capture consisted of 551 packets, and the HTTP capture of 471 packets.

Table 5.2 shows how well LZfuzz dissects an ICMP capture with 551 packets. LZfuzz misses boundaries sometimes when there is little variation near the boundary; to take this error into account, any boundary within a threshold of the true boundary is counted as correct. When a threshold is used, the number of counted correct boundaries sometimes exceeds the number of real boundaries. This could occur if LZfuzz interprets a real boundary as two boundaries that are both within a threshold of the true boundary, and both of the dissected boundaries are counted. We see that both our algorithm and PI framework end up with more false positives than correct finds.

Table 5.3 shows how similar the results are between PI and LZfuzz. In this table, PI is interpreted as the correct dissection. More than half of the boundaries that LZfuzz finds per packet match those found in PI when using a threshold of 2 bytes. Taking into account LZfuzz's significantly lower running time (see Table 1), the difference between the dissections is quite small.

Table 5.4 shows how the accuracy of LZfuzz can be improved by removing short tokens from the string table between the first and second iterations of the compression algorithm. Whenever we removed tokens, we used a threshold of less than 4 bytes. The number of correct finds nearly doubles, and the accuracy of LZfuzz's results becomes closer to that of PI when small tokens are removed.

Table 5.5 shows the results of the two dissectors weighted by entropy and length

	PI			**LZfuzz**		
	real	user	sys	real	user	sys
HTTP	264m24.695s	263m13.107s	1m11.588s	0m2.560s	0m2.536s	0m0.024s
ICMP	45m34.292s	45m33.579s	0m0.440s	0m0.470s	0m0.464s	0m0.008s

Table 5.1: Dissector running times for HTTP and ICMP packet captures. LZfuzz is much faster.

Match	precise	±1 byte	±2 bytes
Correct finds	520	1220	1247
Correct per packet	0.94	2.21	2.26
Incorrect (false positive)	2378	1651	1651
FP per packet	4.32	3.00	3.00
Not found (false negative)	2235	224	224
FN per packet	4.05	0.41	0.41

Table 5.2: LZfuzz dissector scores for ICMP, unweighted. LZfuzz is prone to false positives.

(see section 4). Although PI finds more correct boundaries, it also finds more severe incorrect boundaries. The boundaries that LZfuzz does not find are not very severe: 0.2 incorrect boundaries are found per packet when using a threshold.

Match	precise	±1 byte	±2 bytes
Correct finds	1792	2673	2771
Correct per packet	3.25	4.85	5.03
Incorrect (false positive)	1106	225	127
FP per packet	2.01	0.41	0.23
Not found (false negative)	6473	3414	1822
FN per packet	11.75	6.20	3.31

Table 5.3: LZfuzz dissector compared to PI, on ICMP. PI-produced boundaries are taken as ground truth, as if the ICMP specification were not available. LZfuzz finds most of PI-found boundaries.

Match	LZfuzz, ± 2 bytes, toks < 4 removed	LZfuzz, ± 2 bytes	PI, ± 2 bytes
Correct finds	2333	1247	3857
Correct per packet	4.23	2.26	7.00
Incorrect (false positive)	2841	1651	4408
FP per packet	5.16	3.00	8
Not found (false negative)	0	224	0
FN per packet	0	0.41	0

Table 5.4: Dissector comparison on ICMP capture, unweighted, with string table filtering heuristics applied.

Match	LZfuzz weighted, precise	PI, weighted, precise	LZfuzz, ± 2 bytes	PI, weighted, ± 2 bytes
Correct finds	520	2755	1247	3857
Correct per packet	.94	5	2.26	7
Incorrect (false positive)	1345.29	4157.03	1553.16	4144.97
FP per packet	2.44	7.54	2.82	7.52
Not found (false negative)	701.02	0	114.14	0
FN per packet	1.27	0	0.21	0

Table 5.5: Dissector comparison on ICMP capture, weighted. When weighted by field entropy, LZfuzz's boundary errors appear less severe.

Section 6

Related work

In this section we review the history of fuzzing. We refer the reader to Section 2 for a discussion of the state-of-the-art fuzzing methods and the ongoing challenges that the fuzz-testing methodology faces.

Fuzz testing was formalized in 1989 at the University of Wisconsin-Madison by Professor Barton Miller and the students [MFS90]. The tools developed by Miller et al. threw randomly generated, unstructured input at UNIX programs.

Following the research of Miller et al, numerous fuzzers were developed in attempts to improve the efficiency of the fuzzing approach by generating more structured input. In 1999, a group of researchers at the University of Oulu introduced the PROTOS test suite [RLTK02], which conducted fuzz testing by first analyzing a protocol and then producing inputs based on the protocol specification. PROTOS was followed by SPIKE [Ait02], which introduced block-based protocol analysis. SPIKE uses protocol descriptions in the form of lists of block structures and generates fuzzing data by filling those blocks with randomly generated data, which may contain strings from SPIKE's library of fuzz strings.

Whereas most fuzzers are based on detailed knowledge of the layout of the input to be fuzzed, a few attempt to analyze inputs with unknown structure automatically. A bioinformatics approach to the problem of automatic protocol analysis was implemented by Marshall Beddoe in PI framework [Bed]. PI framework detects fields of protocol packets by aligning packets to find similar sequences, similarly to aligning sequences of genetic information in biology. Automatic detection of building blocks of a protocol, so-called *protocol genes*, was also the motivation of the PROTOS Protocol Genome Project [HVLR06], which uses formal grammars for representing protocol genes.

A genetic algorithm approach was implemented in a fuzzer called Sidewinder, which was presented at the BlackHat security conference in 2006 [ESC06]. A genetic algorithm starts with a population of solutions, selects the most fit solutions, mates them, mutates them, and uses the resulting solutions as the new population to repeat the entire process. Sidewinder's algorithm uses the control-flow graph of the binary under examination; as acknowledged by the authors in the BlackHat presentation, this technique needs testing on more complex problems.

EFS (Evolutionary Fuzzing System), another fuzzer implementing a genetic algorithm, was presented by a group from Michigan State University at BlackHat and Defcon in 2007 [DEP07]. Similarly to Sidewinder, EFS evaluates the fitness of inputs based on the path through the code. EFS uses PaiMei debugging framework to set breakpoints in the code and keep track of hits of these breakpoints.

Another interesting approach to automated protocol dissection was taken by Dan Kaminski in his CFG9000 fuzzer [Kam]. CFG9000 uses the Sequitur algorithm [NMW97] to generate a grammar of the data to be fuzzed. This approach appears to be more suitable to file fuzzing, because of substantial amounts of data required to learn the grammar.

Section 7

Empirical results

We developed our fuzzer, LZfuzz (pronounced "lazy-fuzz") for testing of plain text SCADA protocols and tested it on actual equipment's communications in an isolated control network test environment, successfully validating our methodology. We view the results as a justification of further research into using simpler, faster tricks to model protocols for fuzz-testing.

We also tested LZFuzz on several popular protocols using the FITM setup described earlier. In these tests, since we were fuzzing in real time, the Lempel-Ziv algorithm was only run once over each packet. Fuzzing the protocol used by AOL Instant Messager crashed the Gaim client in Ubuntu. We were also able to hang the iTunes client (version 2.6) consistently by fuzzing the music sharing protocol. We chose not to investigate these conditions further.

We note that LZfuzz is essentially a proof-of-concept. Its accuracy can likely be improved by retaining in the string table and using the information about the origin of the token strings; also, our heuristics to mitigate the accuracy-reducing effects of the initial population of the string table are primitive and can no doubt be improved. We leave these for future work.

Section 8

Conclusions and future work

In this paper, we used a variant of Lempel–Ziv compression algorithm as a very rough protocol dissector for the purposes of fuzz-testing protocol implementations. Although predictably not as accurate as bioinformatics approaches, it nevertheless appears to be able to match the protocol structures well enough to contribute to efficient fuzzing. It is also much faster and requires much less CPU power than bioinformatics methods, which makes it possible to apply it for "online" proxy-based learning and fuzzing.

Addressing the needs of asset owners, we plan to develop a "fuzzer-in-a-box" package that could be deployed and used by network administrators interested in testing proprietary equipment communicating via plain-text type protocols for possible weaknesses, such as DoS conditions (assuming that the attacker gains control of a machine on the same network). In this scenario, the administrators' knowledge about the protocol is minimal (although it may have to include the location and type of control sums, so that the fuzzed packets could be appropriately fixed for testing).

Bibliography

[Ait02] Dave Aitel. The Advantages of Block-based Protocol Analysis for Security Testing. Technical report, Immunity, Inc., February 2002.

[aut] Autodafé. http://autodafe.sourceforge.net/.

[Bed] Marshall A. Beddoe. Network Protocol Analysis Using Bioinformatics Algorithms. http://www.4tphi.net/ awalters/PI/PI.pdf.

[DEP07] Jared D. DeMott, Richard J. Enbody, and William F. Punch. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. Black Hat USA 2007 & DefCon 15, 2007.

[ESC06] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. "Sidewinder": An Evolutionary Guidance System for Malicious Input Crafting. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Embleton.pdf, 2006.

[FGG97] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian Network Classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[HVLR06] Aki Helin, Joachim Viide, Marko Laakso, and Juha Röning. Model Inference Guided Random Testing of Programs with Complex Input Domains. www.ee.oulu.fi/research/ouspg/protos/genome/papers/paper/paper.pdf, 2006.

[Kam] Dan Kaminski. Black Ops 2006. `http://www.doxpara.com/slides/dmk_blackops2006_ccc.ppt`.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[NMW97] Craig G. Nevill-Manning and Ian H. Witten. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[Rit07] Nathan Rittenhouse. Byakugan – Automating Exploitation. ToorCon 9, 2007.

[RLTK02] J. Röning, M. Laakso, A. Takanen, and R. Kaksonen. PROTOS - Systematic Approach to Eliminate Software Vulnerabilities. `http://www.ee.oulu.fi/research/ouspg/protos/`, 2002.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, 2007.

[sul] Sulley Fuzzing Framework. http://www.fuzzing.org/2007/08/02/sulley-fuzzing-framework-release/.