ARL-TR-7543 ● DEC 2015

# Automatic Traffic-Based Internet Control Message Protocol (ICMP) Model Generation for ns-3

by Jaime C Acosta, Felipe Jovel, Felipe Sotelo, and Caesar Zapata

Approved for public release; distribution is unlimited.

## NOTICES

### Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Approved for public release; distribution is unlimited.

ARL-TR-7543 ● DEC 2015

# Automatic Traffic-Based Internet Control Message Protocol (ICMP) Model Generation for ns-3

**by Jaime C Acosta and Felipe Jovel**
***Survivability/Lethality Analysis Directorate, ARL***

**Felipe Sotelo and Caesar Zapata**
***University of Texas at El Paso***

**Approved for public release; distribution is unlimited.**

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 0704-0188* |
|---|---|

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED (From - To) |
|---|---|---|
| December 2015 | Final | June–August 2015 |

**4. TITLE AND SUBTITLE**
Automatic Traffic-Based Internet Control Message Protocol (ICMP) Model Generation for ns-3

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Jaime C Acosta, Felipe Jovel, Felipe Sotelo, and Caesar Zapata

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
US Army Research Laboratory
Cybersecurity and Electromagnetic Protection Division
Survivability/Lethality Analysis Directorate ATTN: RDRL-SLE-I
White Sands Missile Range, NM 88005-5513

**8. PERFORMING ORGANIZATION REPORT NUMBER**
ARL-TR-7543

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSOR/MONITOR'S ACRONYM(S)**

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
The urgency of measuring the security posture of network systems continues to increase with the development of new technologies and the number of vulnerabilities that these introduce. The most effective way of testing these networks is through field tests on real systems in operational environments. Although this provides the highest level of accuracy, the disadvantages that arise with this approach can limit its efficiency and success. These challenges include high costs, time constraints, and the coordination efforts involved in the execution of these tests. One potential solution is the generation of system models that facilitate the continuous experimentation and testing on simulations of these networks in a laboratory environment. Models aid in the testing and analyzing of network systems, but as things stand today, there are limitations to this approach: models can lack synchronization with actual systems and must be built mostly from scratch. In this report, we introduce the ns-3 Model Generator; a tool aimed at automating the generation of protocol models and scenario files that can be run on the ns-3. Our focus for this work was to recreate the Internet Control Message Protocol (ICMP) Ping protocol and an ns-3 scenario using only a 10-second Wireshark network capture. Our results show that in many aspects, the autogenerated protocol is closer to ground truth.

**15. SUBJECT TERMS**
ns-3, simulation, emulation, ARL, US Army Research Laboratory

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| | | | | | Jaime C Acosta |
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | 19b. TELEPHONE NUMBER (Include area code) |
| Unclassified | Unclassified | Unclassified | UU | 40 | (575) 678-8115 |

**Standard Form 298 (Rev. 8/98)**
**Prescribed by ANSI Std. Z39.18**

## Contents


**List of Figures** **iv**

**1. Introduction** **1**

1.1 General Problem 1

1.2 Summary of Methodology 1

**2. Methods and Procedures** **1**

2.1 Gap Analysis 2

2.1.1 A Survey of Commonly Used Protocol Reverse Engineering Tools 2

2.1.2 Applicability and Limitations of Existing Tools for the Model Generator 5

2.1.3 Model Generation 9

2.2 Path Forward 10

2.2.1 "pdmlExtractor.py" 11

2.2.2 "xmlToNs-3Scenario.py" 14

2.2.3 "packetTypeExtractor.py" 15

2.2.4 "fieldVocab.py" 15

2.2.5 "fieldVocabToNs-3Model.py" 17

2.2.6 Prospex 18

2.2.7 "dotToCppGrammar.py" 18

2.3 Accuracy Analysis 19

**3. Results** **20**

**4. Conclusions** **21**

**5. References** **22**

**Appendix. Detailed Analysis of Protocol Reversing Tools** **25**

**List of Symbols, Abbreviations, and Acronyms** **32**

**Distribution List** **33**

## List of Figures

Fig. 1 Netzob main modules ..... 5
Fig. 2 Raw network packet ..... 6
Fig. 3 Prospex system overview[7] ..... 7
Fig. 4 Prospex state machine inferencing tool input file ..... 8
Fig. 5 State tree file output (left); final Ping protocol state machine (right) ..... 8
Fig. 6 ICMP header format ..... 9
Fig. 7 ns-3 automated model generator system overview ..... 10
Fig. 8 PCAP to PDML sample ..... 12
Fig. 9 ICMP protocol found example ..... 12
Fig. 10 "fields.txt" file example ..... 13
Fig. 11 "flows.txt" file example ..... 13
Fig. 12 ICMP "flows.xml" file ..... 14
Fig. 13 "packetTypeSequence.txt" generation ..... 15
Fig. 14 Vocabulary generation ..... 16
Fig. 15 Model standardized XML file ..... 17
Fig. 16 Ping "labeledStateMachine.dot" file ..... 18
Fig. 17 State objects with stored transitions ..... 19
Fig. 18 ns-3 Model Generator script execution parameters ..... 20
Fig. 19 PCAP accuracy comparison ..... 20

# 1. Introduction

## 1.1 General Problem

The network simulation engine, ns-3, is a discrete-event network simulator. Simulations are composed of models that are coded using the C++ programming language. While some graphical tools exist for creating network topologies (e.g., Boston University Representative Internet Topology Generator [BRITE])[1] writing models and scenarios remains a manual, complex, and time-consuming task. Models are used to represent nodes, computers, and other networking components such as physical network interface cards (NICs), network addresses, protocols, etcetera. When put together, these models compose a scenario. Many of the protocol models that are provided by ns-3 are generic and when analyzed with Wireshark,[2] produce network traffic that differs greatly from real protocol traffic. The accuracy of simulation results relies on these models, yet there is a lack of model validation. To address these issues, we introduce a tool that can take a traffic capture of a network and extract the necessary protocol fields, their vocabulary and its state machine, automating the generation of a precise protocol model based on the behavior of real execution.

## 1.2 Summary of Methodology

Our steps in developing the model generator included researching existing technologies that could extract fields from binaries or protocols. We identified limitations and implemented a system that could utilize some of these tools to extract the vocabulary and grammar. We collected 3 network traffic captures. The first was a capture from a real network Internet Control Message Protocol (ICMP) Ping. For the second, we created an ns-3 scenario with 2 nodes; 1 node sent ICMP Ping packets to the other using the icmpv4 model (i.e., the Ping simulator that comes with ns-3). Third, we used our model generator to create a Ping model and a scenario automatically (using the live capture as input). In many aspects, the capture from our model generator was the most similar to the live capture.

# 2. Methods and Procedures

We started by investigating the current state of the art in reverse engineering and binary analysis of network protocols. We identified tools, algorithms, and approaches that we could leverage for our work. Afterward, we designed our system with a modular architecture to facilitate debugging and to support future component integration. Lastly, we built our system and tested its effectiveness by

Approved for public release; distribution is unlimited.

comparing its outputs against 2 controls. We describe this in the following subsections.

## 2.1 Gap Analysis

The actual behavior of protocols can vary depending on their implementation. These variances within the same type of protocols can yield unexpected results when performing field tests. The success of these tests depends greatly on the accuracy of the protocol models that make up the scenario. The data that results from nodes communicating a protocol, including the field vocabulary (i.e., the possible values that packet fields can hold), can affect the outcome of these tests. Determining the information required for extraction from the network packets was crucial for the generation of precise models.

### 2.1.1 A Survey of Commonly Used Protocol Reverse Engineering Tools

Security analysts are often presented with network traffic captures that contain undocumented protocols or binary files. The content and purpose of these protocols is usually a manual process that consists of an analyst conducting reverse engineering. This is the process of extracting the structure, attributes, and data from a network protocol implementation without access to its specification.[3] Several tools have been implemented in attempts to provide a solution to this problem. Some of these tools are described in the following subsections.

#### 2.1.1.1 Automatic Semantics-Aware Analysis of Network Payloads (ASAP) and Protocol Inspection and State Machine Analysis (PRISMA)

The ASAP[4] is an open-source tool, written for the "R"[5] statistical computing language that is designed to automatically extract semantics-aware components from recorded traffic. The author, Tammo Krueger, utilized this tool to characterize normal network protocol behavior by analyzing information in specified fields, and then subsequently to identify and sanitize anomalous, malicious hypertext transfer protocol (HTTP) requests. The ASAP leverages 2 other tools as part of its analysis process. The PRISMA[4] contributes to ASAP by learning the stateful models from the network traffic of a service that can be used for simulating valid communication. Sally[6] is a small tool for mapping a set of strings to a set of vectors. Together these tools contribute to ASAP's functionality:

1) An alphabet of relevant strings is extracted from raw data and used to map payloads into a vector space for analysis.

2) Matrix factorization is applied to identify base directions in the vector space, characterizing usage patterns of mapped payloads.

3) Base directions are traced back to a conjunction of strings from the alphabet resulting in a template of typical communication content.

ASAP initially uses Sally to read a raw file, “asap.raw”, which contains raw network packet payloads. When Sally maps out the strings to a vector space, they are characterized by specific features that include bytes, tokens, n-grams of bytes, or n-grams of tokens. It also generates a sparse vector of count values. ASAP then takes in this output and extracts an alphabet constructed from a set of basic strings and relevant strings. This alphabet is then refined using filtering and correlation techniques.

### *Alphabet Extraction*

Part of the alphabet extraction process involves applying correlation techniques in order to identify nonconstant and nonvolatile strings. Strings within network payloads naturally appear with different frequency, ranging from volatile to constant occurrences. For instance, HTTP requests contain the string “HTTP” in the header, whereas other parts—such as timestamps or session numbers—are highly variable. Statistical t-tests are applied to determine a p-value, which indicates the likelihood of encountering certain values. A tutorial for using this application, provided by Tammo Krueger, included an example of GET requests, HTTP, Simple Mail Transfer Protocol (SMTP), and File Transfer Protocol (FTP), where the alphabet is based on tokens. For binary network protocols such as Domain Name System (DNS), Server Message Block (SMB), and Network File System (NFS), the tool utilizes n-grams for the alphabet. Additional correlation techniques are applied to combine co-occurring strings.

### *Matrix Factorization*

Mapping the network payloads to a vector space reflects characteristics captured by the alphabet. Payloads that share several substrings will appear closer to each other, while those with different content are farther apart. This process allows the discovery of semantics-aware components and base directions in the vector space.

### *Generating Communication Templates*

Strings of the alphabet that exceed a specific threshold are then selected inside the base directions to construct a template. Alphabets of n-grams are concatenated to regain parts of the original ordering. For example, if we have a basis containing the 3-grams “Hos”^”ost”^”st”, then it can be inferred that these tokens overlap and can be concatenated to make up the string “Host”. The procedure is then repeated until no more overlaps exist. This token is then added to the representation list and the procedure is repeated for the next token with the highest value left.

Approved for public release; distribution is unlimited.

#### 2.1.1.2 Prospex

Prospex[7] is a system that can automatically infer specifications for stateful network protocols, including state machine information. This tool can extract the format specifications from a message by monitoring the application as it processes its inputs. The Prospex system's main contributions follow:

1) The system can determine when 2 messages in the session are similarly based, not only on their formats, but on the effects that they have on the receiving server's execution.

2) A state machine can be inferred, which specifies the order in which messages can be exchanged—given no prior knowledge of the protocol under analysis.

This system consists of several modules. The state machine inference module, which is available as open source, is used for our model generator. This module accepts a text file, "sessions.txt", which is composed of the sequences of messages that are observed within a PCAP file. A state machine is then produced from these sequences representing the behavior of the network protocol. Another algorithm, Exbar[8] is then applied to reduce (compress) the state machine by merging similar states. This minimal state machine represents the behavior of the network protocol.

#### 2.1.1.3 Netzob

Netzob[9] is an open-source reverse engineering toolset sponsored by IMOSSYS and Supe`lec. Netzob has a rich feature set that includes inter- and intrapacket-dependency inference, packet simulation, and exports for Wireshark and Peach[10] pit files. It enables protocol message format and state machine inference through passive (i.e., no human intervention) and active processes. Afterward, the model can be used to simulate realistic and controllable traffic. This toolset has been successful at reversing unknown protocols, such as the Zeroaccess botnet command and control (C2) communication.[11] Figure 1 shows the main modules of Netzob.

Approved for public release; distribution is unlimited.

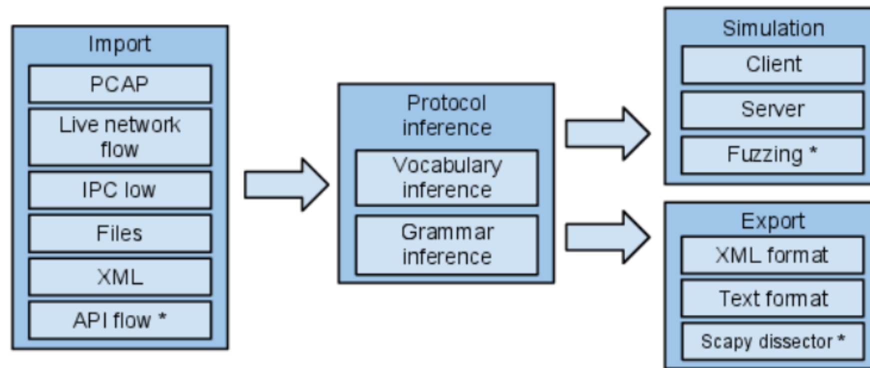


**Fig. 1 Netzob main modules**

- **Import module:** Data can be imported utilizing the built in packet sniffer or by specifying an existing capture, network flow, or other accepted formats.
- **Protocol inference modules:** The vocabulary and grammar inference methods constitute the core of Netzob. This tool has automated and manual mechanisms built in, which allows the reverse engineering of communication flows.
- **Simulation module:** Netzob utilizes the vocabulary and grammar models previously inferred to understand and generate communication traffic between multiple actors.
- **Export module:** Netzob can export an inferred model of a protocol in multiple formats making it extendable to third-party software.

### 2.1.2 Applicability and Limitations of Existing Tools for the Model Generator

We wanted to leverage existing tools, but we had to determine their suitability for our problem. Most tools were either closed source, unavailable, or were not mature enough for our needs. However, we were able to reuse some capabilities of available tools (e.g., Exbar) and develop our software in a way that supports parallel development (i.e., our tool will plug-and-play with other tools: Prospex, Netzob, and ASAP). In the following subsections, we describe our analysis of these tools and some limitations we encountered when considering them for our work.

### 2.1.2.1 ASAP

The author of ASAP, Tammo Krueger, utilized ASAP/PRISMA to extract network protocols by analyzing information in specified fields and then identified malicious HTTP requests and sanitized them. He provides a step-by-step tutorial that shows the process of passing a raw data file into ASAP and then viewing the results. This tutorial includes the sample input files containing raw network payloads with HTTP GET requests (see Fig. 2).

**Fig. 2 Raw network packet**

For the purposes of the model generator, we attempted to run this tool on payloads containing non-HTTP data. We did this by executing every step in the tutorial, with the exception of using a different file as input. When read into Sally, a new file named, “asap2.sally”, was generated and then used as input for ASAP (which used R). “R” successfully read the new file using the “loadPrismaData(asap2)” command, but failed when attempting to create the dataset using “data(asap2)”. This is the function that allows one to see the matrix and other pertinent data. After further analysis and testing, we determined that the reason the “data(asap)” instruction works on the original file is because the generated dataset is already prepackaged inside of PRISMA. To arrive at this conclusion, we removed all of the asap files from the folder that the “R” program accesses when testing the “data(asap)” command. This command still succeeds, meaning that the file is not being read from the local file system, but instead, is being accessed from the installed packages.

Another limitation with ASAP is that it utilizes a delimiter that needs to be manually specified for parsing of the raw data file. This means that prior knowledge of the protocol is required for this program to succeed—rendering the process only semiautomatic.

### 2.1.2.2 Netzob

Netzob has been used to successfully reverse engineer unknown protocols, such as the Zeroaccess botnet C2 malware communication.[11] Being open source, we were hoping to take advantage of this tool's vocabulary and grammar inference modules. However, it is currently not well suited for documented protocols; there is no way to import structures, field sizes, or dependencies from specifications including requests for comment documents or network sniffer data, such as Wireshark pdml data. Grammar inference is not supported for Open Standards Interconnect (OSI) Layer 3 and below protocols such as Optimized Link State Routing (OLSR), Open Shortest Path First (OSPF), Ping, and etcetera. Lastly, there has been little maintenance since 2013. The detailed evaluation steps of Netzob are in the Appendix.

### 2.1.2.3 Prospex

Prospex[7] was very promising due to its success at generating protocol state machines. After researching several tools, one common property found in many systems is that they focus only on extracting the format of individual protocol messages. This makes generating the protocol state machine and producing the specifications for stateful network protocols much more difficult. The initial limitation found with the Prospex tool is that only the state machine inferencing tool was available for download, along with the Exbar algorithm needed to minimize the state machine. The last phase of Prospex is shown in Fig. 3.

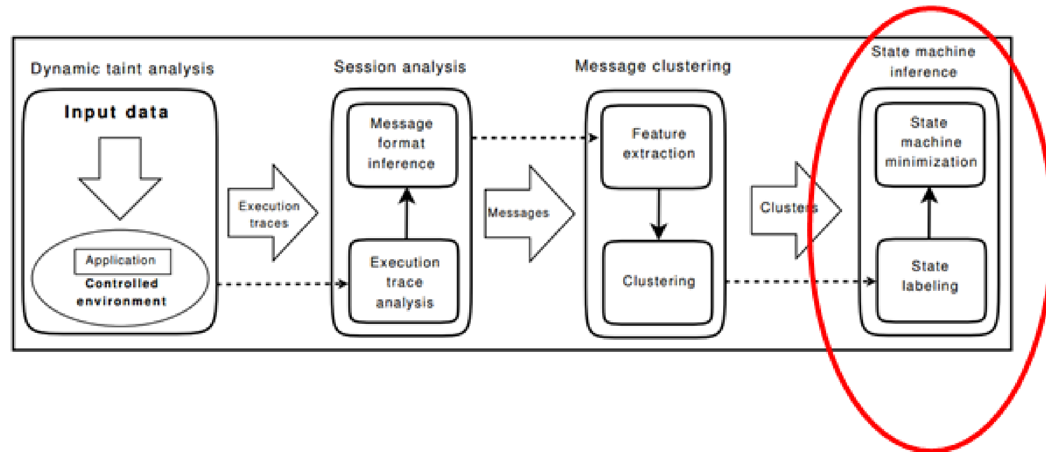


**Fig. 3 Prospex system overview[7]**

The state machine inferencing tool reads a text file composed of sequences of message types observed from a PCAP file. Figure 4 shows a sample input file.

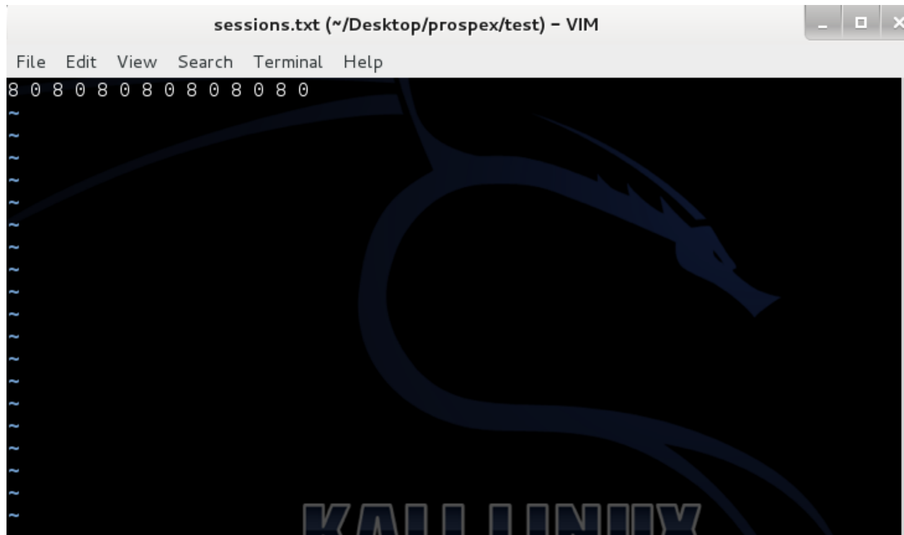


**Fig. 4 Prospex state machine inferencing tool input file**

Figure 4 shows one sequence of ICMP message type. Type 8 is a request, and type 0 is a reply. The input file describes the order in which the different message types are found in the PCAP file. Utilizing this input, this tool generates a state tree dot file. The dot utility is part of the Graphviz[12] drawing package. This is used to generate a visualization of the state tree as shown in Fig. 5a. Exbar compresses this tree and generates the final state machine file. Figure 5b shows the final state machine for the Ping protocol.

**Fig. 5 State tree file output (left); final Ping protocol state machine (right)**

Approved for public release; distribution is unlimited.

Based on our analysis, we determined that the State Machine Inferencing feature of the Prospex tool would be beneficial to our work. In order to incorporate Exbar, we had to generate compatible data (i.e., the input file, "sessions.txt") that must only contain integer values. This process is described in Section 2.2.3.

### 2.1.3 Model Generation

To generate an accurate protocol model, there is specific information needed from the packet captures (i.e., PCAP files). This data includes:

- Protocol fields
  - Size
  - Position
  - Value (hex)
- Field vocabulary
  - Entropy
- Grammar (state machine)

#### 2.1.3.1 Protocol Fields

Figure 6, which appears in Data Communications and Networking,[13] shows the fields that exist in the ICMP Ping protocol. Protocols are described by their fields, including sizes, possible values, and states.

**ICMP Header Format**

| Offsets | Octet | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Octet | Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 0 | Type | | | | | | | | Code | | | | | | | | Checksum | | | | | | | | | | | | | | | |
| 4 | 32 | Rest of Header | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

**Fig. 6 ICMP header format**

After analyzing different tools, we found that Tshark[14] was the best choice for extracting the fields required for our model generation. This tool is a command line version of Wireshark. It has a vast amount of existing protocol dissectors, which allows it to parse the structure of protocols including field names, data types, sizes, etcetera.

#### 2.1.3.2 Field Vocabulary

Figure 6 shows that the ICMP header is composed of 4 fields. Type, for instance, will contain the type of message that is being sent. In this case, this message would be a Type 8, "request", or Type 0, "response". Extracting this information is crucial for generating automated model generation. The vocabularies for protocol fields

(i.e., the values that can exist in each field) can be identified from a PCAP file. To measure the amount of variance in a field's vocabulary, we calculate its entropy. If the entropy is closer to "1", then there is more randomness to this particular field. If it is closer to "0", then there is a smaller range of possible values that exist for that field within the PCAP. Entropy of "0" indicates that the vocabulary is static, which means that the vocabulary consists of a single, unchanging value.

To facilitate the process of extracting the vocabulary, we used Tshark to convert the PCAP files (which are in binary format) to a Packet Details Markup Language (PDML[15]—a markup language that is very similar to HTML or XML).

#### 2.1.3.3 Grammar (State Machine)

Once the fields and vocabulary are extracted from the PCAP files, knowing how the protocols need to behave is the next step. To closely mimic protocol behavior, we needed to infer the state machine specification. We leveraged the State Machine Inferencing tool from Prospex. To do this, we filtered out the protocol of interest from the PCAP file and then extracted the message types from the fields to produce an input file for Prospex to generate the state machine. This state machine is then used to produce the ns-3 protocol models.

## 2.2 Path Forward

The ns-3 model generator reads a PCAP file and—by leveraging Tshark and Prospex—generates the protocol model C++ files needed to run the simulation in ns-3. Figure 7 shows the system overview.

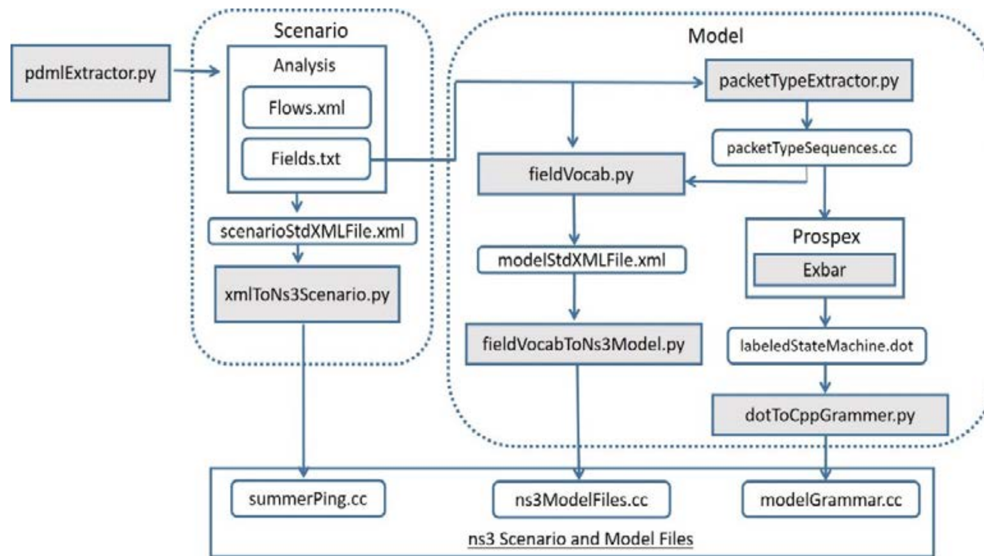


**Fig. 7 ns-3 automated model generator system overview**

This tool is composed of 7 different modules. It takes a PCAP file as input and produces 3 C++ files that are the ns-3 scenario and protocol model. The modules that make up this tool are listed below.

- pdmlExtractor.py
- xmlToNs-3Scenario.py
- packetTypeExtractor.py
- fieldVocab.py
- fieldVocabToNs-3Model.py
- Prospex (third party)
- dotToCppGrammer.py

The "pdmlExtractor.py" program generates 2 files: "fields.txt" and "flows.xml". The "flows.xml" file is used to generate the scenario standardized XML file that is eventually passed to the "xmlToNs-3Scenario.py" program. This will produce the "summerPing.cc" ns-3 simulation file. The "fields.txt" file is fed into the "packetTypeExtractor.py" program to produce the "packetTypeSequence.txt" file that Prospex will eventually use to produce the state machine. Both the "fields.txt" file and "packetTypeSequences.txt" files are needed by the "fieldVocab.py" program to produce the model standardized XML file that is input into the "fieldVocabToNs-3Model.py" for the model generation. Prospex will then read in the "packetTypeSequences.txt" file and produce the state machine in dot format. The "dotToCppGrammer.py" then uses this file to generate a C++ class interpretation of the state machine. All of the C++ files generated are used by ns-3 to generate the protocol simulation.

### 2.2.1 "pdmlExtractor.py"

The "pdmlExtractor.py" Python script reads a PCAP file as input. The first thing this program does is filter all of the protocols in the PCAP and keep only the protocol specified when the program is run. The program then converts the PCAP file from binary to PDML using Tshark. Figure 8 shows how the PDML file is formatted.

Approved for public release; distribution is unlimited.

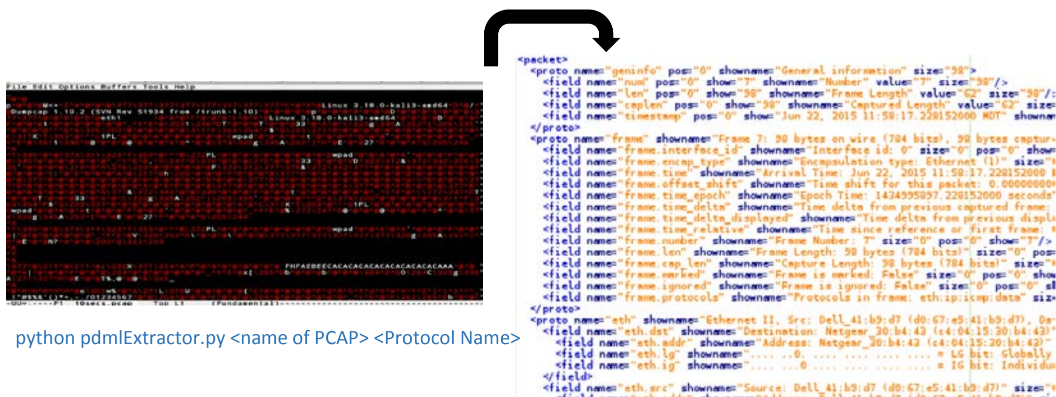


**Fig. 8 PCAP to PDML sample**

Once the file is converted, the program will traverse each packet within the PCAP file and find the protocol that is specified when the program is called. Once it finds the position in the packet, it remembers that position for each packet. Figure 9 shows an example of the line in the PDML file that shows the name of the protocol. In this example we use the ICMP protocol.

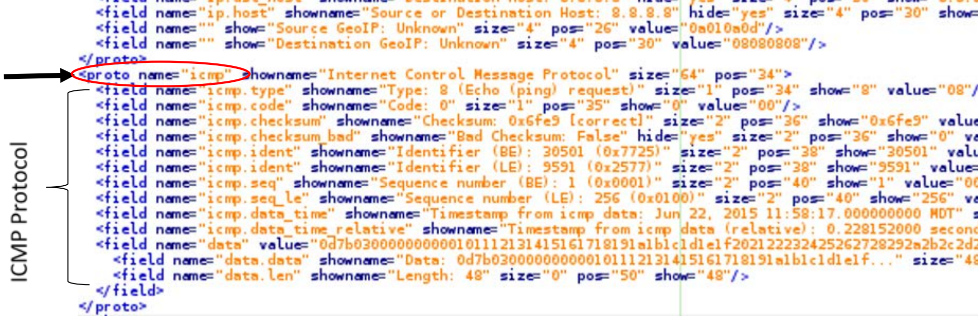


**Fig. 9 ICMP protocol found example**

Most of the lines contain the “name”, “showname”, “show”, “size”, “pos”, and “value” attributes. The “pdmlExtractor.py” will extract all of these attributes and write them out to a file, “fields.txt” with “#” as the delimiter. These will be part of the vocabulary and will be used to calculate the entropy for each field of the protocol. The program then continues the same process for each packet within the PCAP file. Figure 10 shows an example “fields.txt” file.

Approved for public release; distribution is unlimited.

```
File  Edit  View  Search  Terminal  Help
Type: 8 (Echo (ping) request): #size: 1 #show: 8 #pos: 34 #value: 08 #name: icmp.type
Code: 0: #size: 1 #show: 0 #pos: 35 #value: 00 #name: icmp.code
Checksum: 0x6fe9 [correct]: #size: 2 #show: 0x6fe9 #pos: 36 #value: 6fe9 #name: icmp.checksum
Bad Checksum: False: #size: 2 #show: 0 #pos: 36 #value: 6fe9 #name: icmp.checksum_bad
Identifier (BE): 30501 (0x7725): #size: 2 #show: 30501 #pos: 38 #value: 7725 #name: icmp.ident
Identifier (LE): 9591 (0x2577): #size: 2 #show: 9591 #pos: 38 #value: 7725 #name: icmp.ident
Sequence number (BE): 1 (0x0001): #size: 2 #show: 1 #pos: 40 #value: 0001 #name: icmp.seq
Sequence number (LE): 256 (0x0100): #size: 2 #show: 256 #pos: 40 #value: 0001 #name: icmp.seq_le
Timestamp from icmp data: Jun 22, 2015 11:58:17.000000000 MDT: #size: 8 #show: Jun 22, 2015 11:58:17.000000000 #p
s: 42 #value: b94c885500000000 #name: icmp.data_time
Timestamp from icmp data (relative): 0.228152000 seconds: #size: 8 #show: 0.228152000 #pos: 42 #value: b94c885500
00000 #name: icmp.data_time_relative
unspecified: #size: unspecified #show: unspecified #pos: unspecified #value: 0d7b030000000000101112131415161718191
a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637 #name: data
Data: 0d7b0300000000001011121314151617181191a1b1c1d1e1f...: #size: 48 #show: 0d:7b:03:00:00:00:00:00:10:11:12:13:1
:15:16:17:18:19:1a:1b:1c:1d:1e:1f:20:21:22:23:24:25:26:27:28:29:2a:2b:2c:2d:2e:2f:30:31:32:33:34:35:36:37 #pos: 5
 #value: 0d7b0300000000001011121314151617181191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637 #name:
ata.data
Length: 48: #size: 0 #show: 48 #pos: 50 #value: unspecified #name: data.len
                                                                                    1,1           Top
```

**Fig. 10 “fields.txt” file example**

The “pdmlExtractor.py” program also generates a “flows.xml” file. This file will utilize other fields within each packet to extract the following information:

1) Source Internet Protocol (IP) address
2) Destination IP address
3) Source Mac address
4) Destination Mac address
5) Epoch time

This information is extracted from each packet within the capture and is also written to the flows file. Figure 11 shows an example of the “flows.xml” file for a Ping protocol.

```
xmlFlow.xml (~/Desktop/xmlScript/pcapParser/09:20:18.4984
File  Edit  View  Search  Terminal  Help
<root>
   <flow>
      <protocol>icmp</protocol>
      <macsrc>d0:67:e5:41:b9:d7</macsrc>
      <macdst>c4:04:15:30:b4:43</macdst>
      <ipsrc>10.1.10.13</ipsrc>
      <ipdst>8.8.8.8</ipdst>
      <time>2015-06-22 11:58:17:228152</time>
   </flow>
   <flow>
      <protocol>icmp</protocol>
      <macsrc>c4:04:15:30:b4:43</macsrc>
      <macdst>d0:67:e5:41:b9:d7</macdst>
      <ipsrc>8.8.8.8</ipsrc>
      <ipdst>10.1.10.13</ipdst>
      <time>2015-06-22 11:58:17:250320</time>
   </flow>
   <flow>
      <protocol>icmp</protocol>
      <macsrc>d0:67:e5:41:b9:d7</macsrc>
      <macdst>c4:04:15:30:b4:43</macdst>
      <ipsrc>10.1.10.13</ipsrc>
      <ipdst>8.8.8.8</ipdst>
      <time>2015-06-22 11:58:18:229549</time>
   </flow>
```

**Fig. 11 “flows.txt” file example**

This “flows.xml” file is also used to generate a scenario standardized XML file. After researching other network simulation programs, we found a structure that would be easy to integrate with other simulation engines. This structure describes all of the protocols that come out of each node or computer—assuming that the node can be distinguished based on a single-source Mac address. This part of the program will tie each protocol with a specific node and will list the source addresses and time, etcetera. Figure 12 shows a “scenarioStdConfigurationFile.xml” file for a Ping protocol that was captured with Wireshark.

```
<network>
  <node>
     <number>0</number>
     <macsrc>d0:67:e5:41:b9:d7</macsrc>
     <flow>
       <protocol>icmp</protocol>
       <macdst>c4:04:15:30:b4:43<macdst>
       <ipsrc>10.1.10.13</ipsrc>
       <ipdst>8.8.8.8</ipdst>
       <time>
          <first>2015-06-22 11:58:17:228152</first>
          <last>2015-06-22 11:58:26:241474</last>
          <count>10</count>
       </time>
     </flow>
  </node>
  <node>
     <number>1</number>
     <macsrc>c4:04:15:30:b4:43</macsrc>
     <flow>
       <protocol>icmp</protocol>
       <macdst>d0:67:e5:41:b9:d7<macdst>
       <ipsrc>8.8.8.8</ipsrc>
       <ipdst>10.1.10.13</ipdst>
       <time>
          <first>2015-06-22 11:58:17:250320</first>
          <last>2015-06-22 11:58:25:283245</last>
          <count>9</count>
       </time>
     </flow>
  </node>
</network>
```

**Fig. 12 ICMP “flows.xml” file**

This example describes 2 nodes that were communicating with each other using the Ping protocol. Each node is numbered and is distinguished with a unique Mac source address. It also includes the protocol that is generated by the node, encapsulated within the flow tag,“<flow>”. This flow includes the protocol name, the Mac destination address, and IP addresses. The epoch time and the number of times this flow was seen in the PCAP file are also described.

### 2.2.2 “xmlToNs-3Scenario.py”

The “xmlToNs-3Scenario.py” Python script generates the actual scenario C++ file that is used by ns-3 to run the simulation. This program takes the “scenarioXMLConfigurationFile.xml” as input.

### 2.2.3 “packetTypeExtractor.py”

One common feature utilized in other protocol extraction tools is the use of a vocabulary dictionary. This feature was useful when trying to extract the message type for each protocol. Each protocol has a message type that is described by one of its fields within the protocol. The challenge is that every protocol uses a different naming convention to describe their type. For example, the ICMP message type can be found in a field named “Type”, but other protocols might use the string, “Command” as the field name. For this reason, each protocol would need to be treated differently when extracting its type. The “packetTypeExtration.py” tool takes in the “fields.txt” file produced by the “pdmlExtractor.py” program and reads line by line, comparing every string against a dictionary of common words. This dictionary is specified inside this method and is easily expandable. If a type is found within the packet, the contents of that field will be stored in an array data structure in Python named, “allTypes”. If this is the first occurrence of the type it will also be stored in a separate array of unique message types called “uniqueTypes”. Once all of the types are found, the array containing the unique message types found in the PCAP file is used to generate the “packetTypeSequences.txt” file. This text file is used to generate the state machine using Prospex. Because Prospex can only accept integer data types in the sequences file, we use the index of their position in the unique array as the message type in the order that they are found in the “alltypes” array. Figure 13 illustrates this process.

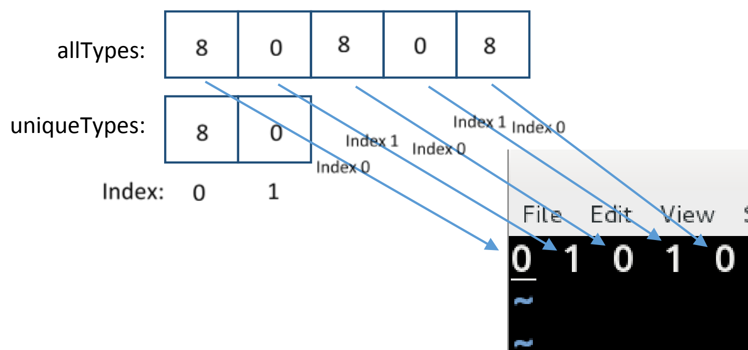


**Fig. 13 “packetTypeSequence.txt” generation**

### 2.2.4 “fieldVocab.py”

The “fieldVocab.py” program will utilize the “fields.txt” output file from the “pdmlExtractor.py” tool to generate the vocabulary for the protocol. This tool will look at every field for each packet and will store the values that are different. Some of the field values never change and therefore will only have one vocabulary stored, while other fields may have a different value for every packet. This process is done by creating a field class for every field in the packet. Figure 14 illustrates this process.

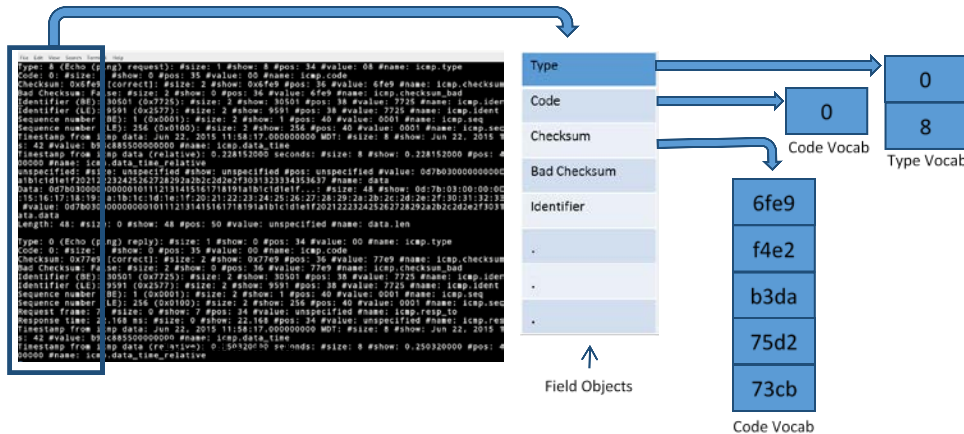


**Fig. 14 Vocabulary generation**

As shown in Fig. 14, the initial step is to create an array of field objects. Each field object contains a list that will hold the vocabulary for that field. The vocabulary in every field of each packet in the “fields.txt” file will be compared to the values that are already stored in the vocabulary list for that object. If that value does not exist in the list, it will be added. If it exists, then the value will be ignored. This process will occur for the entire “fields.txt” file. Once this process is complete, each list that is generated for the fields will have their entropy calculated. The calculated entropy describes the randomness of the vocabulary within a list. This entropy is calculated using the number of values that are stored in the vocabulary list; the length of the list. For example, Fig. 14 shows that the “Type” object contains a list of 2 values: 0 and 8. This means that there are only 2 possible values for the “Type” field in the entire PCAP file. The entropy for this vocabulary is 0.5. The algorithm used to calculate entropy is:

$$Entropy = (x-1)/x, \text{ where } x \text{ is the size of the vocabulary} \quad (1)$$

This algorithm ensures that entropy is never >1. The larger the vocabulary, the closer the entropy is to 1. If there is only one item in the vocabulary, the entropy is 0. This means that the value is the same in every packet in the PCAP file. Once the

entropy is calculated, the values are used to generate the standardized XML file for the model generation. Figure 15 shows an example of this file for the Ping protocol.

File Edit View Search Terminal Help

```
<summerping>
   <mtype id = '0'>
      <mfield>
         <mname>icmp.type</mname>
         <mshowname>Type: 8 (Echo (ping) request)</mshowname>
         <msize>1</msize>
         <mpos>34</mpos>
         <mshow>8</mshow>
         <mvocab>08;</mvocab>
         <mentropy>0.0</mentropy>
      </mfield>
      <mfield>
         <mname>icmp.code</mname>
         <mshowname>Code: 0</mshowname>
         <msize>1</msize>
         <mpos>35</mpos>
         <mshow>0</mshow>
         <mvocab>00;</mvocab>
         <mentropy>0.0</mentropy>
      </mfield>
      <mfield>
         <mname>icmp.checksum</mname>
         <mshowname>Checksum: 0x6fe9 [correct]</mshowname>
         <msize>2</msize>
         <mpos>36</mpos>
         <mshow>0x6fe9</mshow>
         <mvocab>6fe9;f4e2;b3da;75d2;73cb;c8c5;9dc0;a7b8;95b1;55ac;</mvocab>
         <mentropy>0.9</mentropy>
      </mfield>
      <mfield>
         <mname>icmp.checksum_bad</mname>
         <mshowname>Bad Checksum: False</mshowname>
         <msize>2</msize>
         <mpos>36</mpos>
         <mshow>0</mshow>
```

**Fig. 15 Model standardized XML file**

The standardized file is structured based on the message types. In this Ping example, there are only 2 types of messages: 0 and 8. These are separated and encapsulate all of the fields that pertain to that type. Each field also contains the name of the field along with their size, position, vocabulary, and entropy. This is used later to generate the protocol model.

### 2.2.5 "fieldVocabToNs-3Model.py"

This Python script takes in the model standardized XML file and will generate the C++ model files needed by ns-3. The script executes 4 tasks to create the ns-3 model:

1) Extract all field attributes (i.e., names, sizes, vocabulary).
2) Using the data from 1, C++ variables are generated (using names) along with data types (using sizes).
3) Append initialization values for variables (the first value in the vocabulary is used).

Approved for public release; distribution is unlimited.

4) Append comments in the C++ code that indicate the entropy associated with a variable (i.e., based on vocabulary).

### 2.2.6 Prospex

Prospex was used due to its State Machine Inferencing tool. This tool takes in the “packetTypeSequences.txt” file, which is described in Fig. 3. Refer to the Prospex Section 2.1.2.3 for further details. There are several files that are produced by the program. The DOT file named, “labeledStateMachine.dot” is the one that we are interested in. This file will be used to create the grammar and state machine, for the protocol.

### 2.2.7 “dotToCppGrammar.py”

This module will read in the “labeledStateMachine.dot” file that is generated by the State Machine Inferencing tool provided by Prospex. The format in which this file is generated makes it easy to translate into a C++ file that is needed for the model generation. Figure 16 shows an example DOT file for the Ping protocol.



```
digraph message{
node0 [label="8" ,shape=doublecircle ];
node29 [label="0 8" ,shape=circle ];
node0 -> node29 [label="8"];
node29 -> node0 [label="0"];
}
~
~
~
~
```

**Fig. 16 Ping “labeledStateMachine.dot” file**

The Fig. 16 example shows how we distinguished states and transitions. The “dotToCppGrammar.py” program takes advantage of this by creating State objects in Python that will each contain their own transitions. This process is illustrated in Fig. 17, which shows each state and the list of transitions stored inside the state object. The outer arrows describe the transitions that are included in the objects.

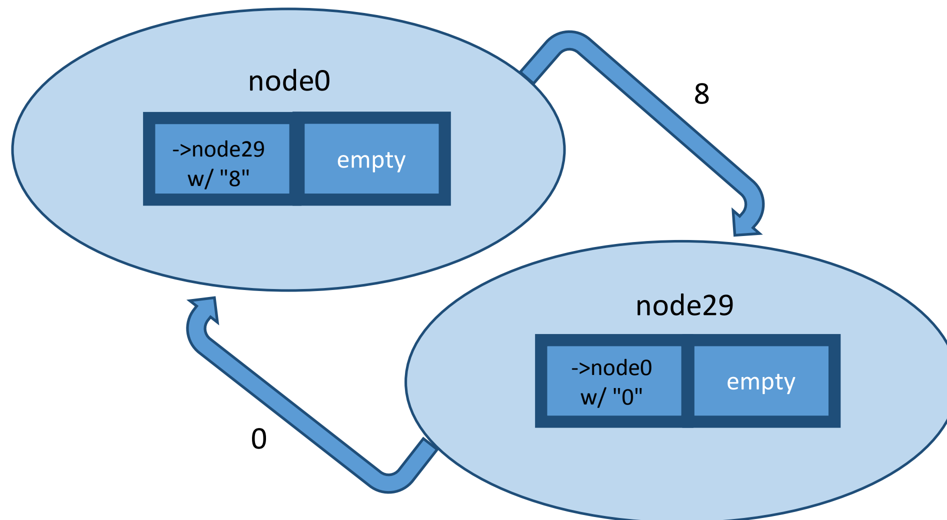


**Fig. 17 State objects with stored transitions**

The characteristics of this state machine are all incorporated into the C++ code that is generated by this script. Included in this code is a function called ' "getNextState()", which is used to know the message type that needs to be described in the model at a specific point during the simulation. The state object is responsible for keeping track of the protocol current state and, when requested, will return the next possible states. The "getNextState()" method will accept an integer value that represents the message type the model is changing to, or receiving, and the method will return the possible types that the model can then change to, or reply as. This is all written in one C++ file generated by the "dotToCppGrammar.py" program (i.e., <protocolName>.cc.)

## 2.3 Accuracy Analysis

For testing purposes, as part of ns-3, a PCAP file is generated when the simulation of the Ping protocol is run. This capture file is used to compare the packets that are produced by the simulation with the actual packets in the original PCAP. In addition, the packets are also compared with those from existing ns-3 models available through ns-3 and other resources. This process shows firsthand, the accuracy of the ns-3 models being produced by the model generator.

The experiments that we ran used a 10-second (s) PCAP capture using the Wireshark network protocol analyzer. This program takes in the PCAP file, the name of the protocol that we want to simulate with ns-3, and the naming convention we want to use for the model (see Fig. 18).

Approved for public release; distribution is unlimited.

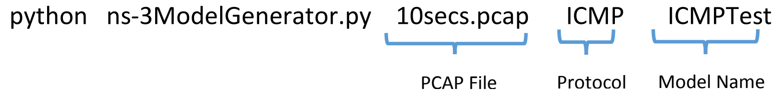


**Fig. 18 ns-3 Model Generator script execution parameters**

The entropy—which is included in the model standardized XML file—is crucial for the accuracy of the protocol models. This is because some protocols might include a large list of possible vocabulary for some fields. If this is the case, the user must manually choose one of the vocabularies for that field, or choose to ignore it completely. This can affect the fidelity of the models.

## 3. Results

This tool was tested using the ICMP Ping protocol. It created an accurate Ping ns-3 model and generated a simulation file that runs successfully on ns-3. The ns-3 simulation also creates a PCAP file, as described earlier, which we use to determine the accuracy of the model. By comparing the packets that are produced by our simulation with the actual packets in the original PCAP file, we find that the autogenerated model produces data very similar to the original.

Figure 19 shows images of the PCAP files (i.e., in Wireshark). The original 10-s capture is the ground truth. The ns-3 sample PCAP contains only null values in the data field. The PCAP file generated by our tool contains realistic data and values that match the data field from the 10-s capture. In the generated model, several fields had low entropy (e.g., field type, code, and others) and some had high entropy. High-entropy fields include the checksums, timing, and sequence numbers. While human intervention will be required to more closely mimic the original protocol, an analyst is provided with an annotated template as a starting point.

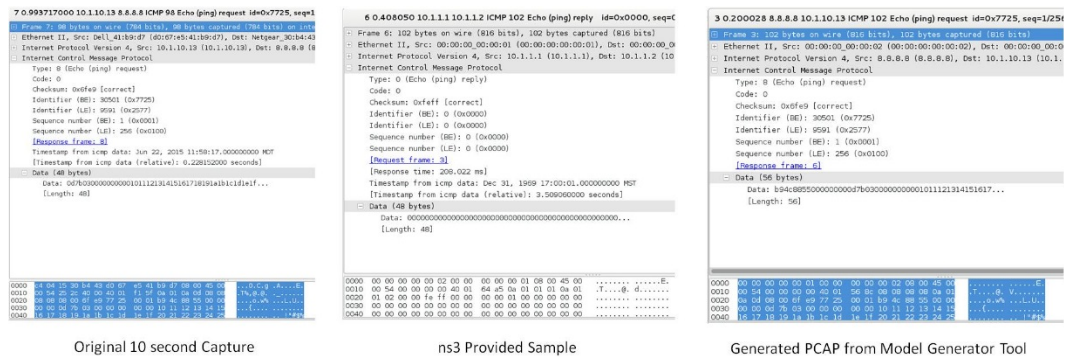


**Fig. 19 PCAP accuracy comparison**

## 4. Conclusions

We have created a model generator and demonstrated how it can be used to extract protocol fields, vocabulary, and state machine specifications for a Ping protocol. These features are crucial in the generation of accurate protocol models. By leveraging Tshark and Prospex, this tool can generate the ns-3 C++ files needed to run a simulation of a protocol from a PCAP file. Due to the vast implementations of different protocols, the entropy of the vocabulary needs to be calculated to determine the values included in the model. The model standardized XML file lists the possible vocabulary that can be chosen for all fields.

Some future near-term improvements that need to be made include testing with more protocols (especially at different layers of the OSI model), implementing an inference engine to extract inter- and intrapacket dependencies, and also adding the ability to generate models that can be used in other simulator/emulators as well as network scripting engines (e.g., scapy).

Approved for public release; distribution is unlimited.

## 5. References


1. Swenson BP, Riley GF. Simulating large topologies in ns-3 using BRITE and cuda driven global routing. Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques; 2013; Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST).

2. Chappell LA, Combs G. Wireshark 101: Essential skills for network analysis. Protocol Analysis Institute, Chappell University, 2013.

3. Lee D, Sabnani K. Reverse-engineering of communication protocols. Network Protocols, 1993. Proceedings 1993 International Conference on IEEE; 1993.

4. Bossert G, Guihery F, Hiet G. Towards automated protocol reverse engineering using semantic information. Proceedings of the 9th ACM Symposium on Information, Computer and Communications security. ACM; 2014.

5. Ihaka R, Gentleman R. (1996); R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, *5*(3), 299–314.

6. Rieck K, Wressnegger C, Bikadorov A. Sally: A tool for embedding strings in vector spaces. The Journal of Machine Learning Research 13.1. 2012;3247–3251.

7. Comparetti PM, Wondracek G, Kruegel C, Kirda E. Prospex: Protocol specification extraction. Security and Privacy, 2009 30th IEEE Symposium. IEEE, 2009.

8. Lang KJ. Faster algorithms for finding minimal consistent DFAs. 4 Independence Way, Princeton (NJ): NEC Research Institute (US); 1999.

9. Bossert G, Guihéry F. Security evaluation of communication protocols in common criteria using Netzob. ICCC, 2013.

10. Peach Fuzzing Platform; 2008 [accessed 21 Sep 2015]. http://peachfuzzer.com.

11. Shearer J. Trojan: Zeroaccess threat report. Symantec, 2011.

12. Ellson J, Gansner E, Koutsofios L, North SC, Woodhull G. Graphviz—open source graph drawing tools. Graph Drawing. Springer Berlin Heidelberg, 2002.

13. Forouzan BA. Data Communications and Networking (4th ed.). Boston (MA): McGraw-Hill; 2007. p. 621–630.


Approved for public release; distribution is unlimited.

14. Combs G. TShark: The Wireshark Network Analyser. [accessed 21 Sep 2015]. http://www. wireshark.org.

15. PDML Specification, 2015. [accessed 21 Sep 2015]. ftp://ftp.tuwien.ac.at/.vhost/analyzer.polito.it/docs/dissectors/PDMLSpec.htm

16. Postel J. RFC 792: Internet control message protocol. InterNet Network Working Group (1981).

Approved for public release; distribution is unlimited.

INTENTIONALLY LEFT BLANK.

Approved for public release; distribution is unlimited.

# Appendix. Detailed Analysis of Protocol Reversing Tools

Approved for public release; distribution is unlimited.

## A.1 Setting Up the Automatic Semantics-Aware Analysis of Network Payloads (ASAP)/Protocol Inspection and State Machine Analysis (PRISMA)

1) Download PRISMA from https://github.com/tammok/PRISMA.

2) Download Sally from *http://www.mlsec.org/sally/*.

3) Install the latest version of R.

   It is necessary for the version of R to be 3.2 or higher.

**"R" Install Instructions**

a) Add the following lines to source.list in *etc/apt/sources.list*.

   ## R BACKPORTS FOR WHEEZY

   deb http://cran.revolutionanalytics.com/bin/linux/debianwheezy-cran3/

   #deb-src
   http://cran.revolutionanalytics.com/bin/linux/debianwheezycran3/

b) Then do the following commands:

   apt-get update

   apt-get upgrade

   apt-get install r-base r-base-dev

4) Run "R" and enter the following commands. This step will install the necessary libraries into R for running ASAP/PRISMA.

   install.packages ("PRISMA")

   library ("PRISMA")

There are some dependencies that might need to be installed manually due to incompatible versions. This can be done using the same command: *install.packages("<NAMEOFDEPENDENCY>")*.

5) Install Sally.

   a) Install the necessary packages as listed in the README.md provided in the downloaded Sally directory. To do so enter the following commands:

      1. apt-get install libz-dev

2. apt-get install libconfig8-dev
3. apt-get install libarchive-dev
4. (if gcc isn't installed) apt-get install gcc

b) Run the following commands:

1. ./configure
2. make
3. make check
4. make install

Sally should be ready to run.

**Running ASAP on a sample Payload.**

6) From here the steps from the tutorial should be followed. The steps are listed here as well.

a) Extract asap.tar.gz which is located under /PRISMA-master/inst/extdata/.

b) Create the Sally output file from the raw data provided by using the following steps.

It is necessary to change some of the features in the sally.cfg file.

To do this, follow these steps:

i. Open the sally.cfg file.

ii. Under the feature configuration section add the following line: granularity = "tokens".

iii. Under the same section, change the variable name "ngram_delim" to "token_delim" without changing its value delimiter.

iv. Save changes.

Create the Sally file using the following command.

sally-c asap.cfg asap.raw asap.sally

c) To speed up the loading of the data in "R", apply the "sallyPreprocessing.py" Python script with the following command: *python sallyPreprocessing.py asap.sally asap.fsally*.

Approved for public release; distribution is unlimited.

d) Load the data into "R"

i. Start-up "R" (enter "R" in the terminal).

ii. Load the data via: loadPrismaData ("PRISMA").

Following the tutorial, additional information is provided about the payload that is being analyzed.

Keywords:

1) Payload: a string of bytes contained in network communication.
2) T-test: a statistical t-test result is one in which the difference between 2 groups is unlikely to have occurred due to chance.
3) Pearson correlation coefficient: a measure of the linear correlation between 2 variables X and Y, giving a value between +1 and –1 inclusive.

**Converting to XML:**

T-shark command to get PDML transformation:

tshark-r "<.pcap_file>" -T pdml -E separator=, > <.pdml_file>

-r : read from file

-T: transform to specified format

-E: Use delimeter

> output to specified file

Command to get all the protocols that exist within the PCAP file to extract all protocols:

*tshark -r test.cap -z io,phs -q | tr -s ' ' | cut -f 2 -d ' ' | tail -n +7 | head -n -1*

## A-2 Netzob Detailed Evaluation

We evaluated Netzob by running through the example provided on the website tutorial. We were able to complete the tutorial using the sample data provided on the website and we then proceeded to rerun the tutorial with our 10-second (s) ICMP PCAP file.

The following are the steps in the tutorial:

- Import of a PCAP file

- Format message inference
    - Partitionment of messages following a specific delimiter
    - Regroupment of messages following a specific key field
    - Partitionment of a subset of each message following a sequence alignment
    - Search for relationships in each group of messages
    - Modification of the format message to apply found relationships
- Grammar inference
    - Generation of an automaton with one main state according to a captured sequence of messages
    - Generation of an automaton with a sequence of states according to a captured sequence of messages
    - Generation of a Prefix Tree Acceptor (PTA) automaton according to a captured sequence of messages
- Traffic generation and fuzzing
    - Generation of messages following the inferred message format of each group and through visiting the inferred automata
    - Fuzzing of an implementation by generating altered message formats

We tested with 3 versions of Netzob. First, we used the official stable release of the software (that allows using the guided user interface [GUI]). Second we tried with the github branch labeled “next”, and third with the source code branch labeled “master”. It seems, by looking at the Git log that the developers quickly removed the GUI functionality after the release of Netzob 0.4.1, probably due to several modifications to the application programming interface (API); making the GUI interface no longer usable. We noticed that a developer by the name of Georges was actively updating both branches of the source code (different files in each branch).

While running Netzob, the first problem occurred when importing the PCAP file from the tutorial. We had to modify the call to the “PCAPImporter.readFile(“target_src_v1_session1.pcap”).values()” by adding a parameter that specified that data only up to the network Layer 3 (i.e., the Internet Protocol [IP] layer) should be parsed. After fixing the “PCAPImporter” call, the

Approved for public release; distribution is unlimited.

tutorial code ran all the way through to completion, but the results were incorrect. In the next step of the tutorial, the authors show that they partition the messages in the sample PCAP using a "#" delimeter. This character is used to separate the commands in the command and control (C2) traffic. In our PCAP file (i.e., the 10-s live Ping capture), this character was present, but it was not meant to be a delimeter. We looked into other ways to partition the data.

There are several ways that one can parse symbols (or packet types). The tutorial uses sequence alignment, but with Ping—in this particular case with the loopback traffic—the fields were parsed out most correctly when using the simple partitioning technique (this technique separates data based on the dynamicity of bytes, whether they change across messages.) Looking more closely at the source code, we identified the location of the partitioning functions: "src/netzob/Inference/Vocabular/Format.py".

In the code, the name of the module is "Format" and the function is called "splitStatic". It is possible to call this function with 4 parameters the "unitSize", "mergeAdjacentStaticFields", and "mergeAdjacentDynamicFields". We obtained the best results specifying only the unitSize. We also tried splitting the input into 8-bit segments and then using merge to create fields from the ICMP Ping specification.[16] While this did work for display purposes (i.e., splitting fields into 8-bit segments), when merged the field types become *aggregate* instead of *raw* causing incompatibility with the other algorithms in Netzob (e.g., the clustering algorithm no longer worked on the data). We proceeded by using the splitStatic function with only the symbol array as an input. We originally thought that we could create a converter module that could take as input a protocol specification and then use Netzob to parse symbols based on that specification. However, we were unsuccessful in finding a nontrivial way to do this.

The next step in the tutorial uses the "clusterByKeyField" function in the Format module (the actual file is located at src/netzob/inference/Vocabulary/FormatOperations/ClusterByKeyField.py). This will use values in the fields that are specified to cluster message types. When we ran this with Field 5 (i.e., the ICMP type field that contained 2 values: \x00 and \x08), Netzob (the *next* branch) would crash stating that the last field in the message was not aligned correctly (i.e., by the ParallelDataAlignment.py module). We modified the code to use the DataAlignment module instead. We isolated the problem in the "ClusterByKeyField.py"; it was using the "\x00" (i.e., valid ASCII) character to infer the type of the field as ASCII. When "\x08" was read, the character was invalid (i.e., not a valid ASCII character) and Netzob would crash. We modified the code to force the type as HexaString. This fixed this problem.

Approved for public release; distribution is unlimited.

The splitStatic algorithm worked well with this particular instance of the Ping traffic (which was loopback traffic, but did not work the same way when using a capture of pings to a google.com server; this was strange because the Netzob 0.4.1 stable release gave correct results).

We executed the next step in the tutorial (i.e., search for relationships in each group of messages), but this produced no relationships.

Afterwards, we used the Automata module to generate state machines using different functions: "generateChainedStateAutomata", "generateOneStateAutomata", and "generatePTAAutomata". The "generateChainedStateAutomata" generates all possible states for each unique transition. The "generateOneStateAutomata" generates a universal receiver (i.e., will accept all traffic as valid input). Regardless of what is received, it will respond. The "generatePTAAutomata" takes as input several communication sessions and then identifies common paths and merges these into a single automata. The resulting state machines only consisted of 3 states: start state, open channel, and close channel (i.e., end state). To generate an automata that can work with real traffic, the user must pass real traffic into Netzob. Only then will a usable grammar (i.e., state machine) be generated.

The final steps in the tutorial create a traffic generator and a fuzzer. Because the previous steps did not yield successful results, we were not able to complete these.

## List of Symbols, Abbreviations, and Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| ASAP | Automatic Semantics-Aware Analysis of Network Payloads |
| BRITE | Boston University Representative Internet Topology Generator |
| C2 | command and control |
| DNS | Domain Name System |
| FTP | File Transfer Protocol |
| GUI | Graphical User Interface |
| HTTP | Hypertext Transfer Protocol |
| ICMP | Internet Control Message Protocol |
| IP | Internet Protocol |
| NFS | Network File System |
| NIC | Network Interface Card |
| OLSR | Optimized Link State Routing |
| OSPF | Open Shortest Path First |
| OSI | Open Standards Interconnect |
| PDML | Packet Details Markup Language |
| PRISMA | Protocol Inspection and State Machine Analysis |
| PTA | Prefix Tree Acceptor |
| SMB | Server Message Block |
| SMTP | Simple Mail Transfer Protocol |

Approved for public release; distribution is unlimited.

1 (PDF) DEFENSE TECHNICAL
INFORMATION CTR
DTIC OCA

2 (PDF) DIRECTOR
US ARMY RESEARCH LAB
RDRL CIO LL
IMAL HRA MAIL & RECORDS
MGMT

1 (PDF) GOVT PRINTG OFC
A MALHOTRA

1 (PDF) DIRECTOR
US ARMY RESEARCH LAB
RDRL SLE I
J ACOSTA

INTENTIONALLY LEFT BLANK.

Approved for public release; distribution is unlimited.