ProtoGANist: Protocol Reverse Engineering using Generative Adversarial Networks

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Security

Carolina M. Zarate

B.S., Computer Science, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May, 2019

Acknowledgements

First and foremost a huge thank you to my thesis advisors, Dr. Vyas Sekar and Dr. Giulia Fanti, who have shared their extensive knowledge and experience in the subject matter. This research had me working with new and challenging ideas and I feel like I have gained a lot from this experience. I am grateful for the opportunity you have provided me in this work and your guidance and experience has been truly been instrumental in my research.

I would also like to thank the many folks at Carnegie Mellon University that have made my time at CMU a great experience: Professor Mark Stehlik, Dr. David Brumley, Professor Dave Eckhardt, Dr. Dena Haritos Tsamitis, and so many others. I am thankful for the opportunities and help you have provided for me to succeed during my time at CMU. Your faith in my abilities has given me the determination to succeed.

Thank you to my peers in CyLab and the research community that have supported me during the course of this research — specifically Sekar Kulandaivel and Zinan Lin, who have provided much help and feedback during the course of my research. And a thank you to the many current and previous research students that have helped — Soo-jin Moon, Jay Bosamiya, and Dr. Tiffany Bao. Your experiences and advice have helped me better prepare myself to complete this work.

This research was self-funded by the researcher.

To my friends near and far — Matthew Savage, Zachary Wade, Corwin de Boor, Noelle Toong, John Davis, Sarah Bien, Azer Wang, Ananya Rajgarhia, Ana Vlajnic, Eunice Chung, Manuel Guillen, Alex Jia, Chris Thompson, and so many others. Your moral support has helped me achieve many things.

Thank you to Joseph Kim, for all his love and support in my many crazy endeavors.

Lastly, a thank you to my family: my mother, Silvina Zarate, my father, Carlos Zarate, and my siblings, Diego and Sofia Zarate. Your unconditional love, support, and goofing off has brought me so much joy and strength in all my undertakings.

Again, a huge thank you to the immense number of people that have supported me. Your encouragement has played a huge part in the completion of this research and other aspects of my life.

Abstract

Many reported vulnerabilities are related to the way that a system accepts, processes, and interprets protocol packets and the information contained therein. Adversaries can trigger these vulnerabilities by sending specially crafted packets to the system. Typical solutions to this problem include generating packets in accordance with the protocol format, sending them to the system, and observing the resulting behavior on the system. However, these solutions fall apart when dealing with a black box system and black box protocols, because it is unclear how to generate realistic protocol packets. We present ProtoGANist, a system to model unknown protocol message formats and produce messages similar to the underlying format using generative machine learning models. Given sample messages from a black-box protocol and a black-box system that uses the protocol, our goal is to learn to produce randomized protocol-compliant messages. The difficulty of this task lies in the complexity of the protocol message format. Message fields' values, lengths, and overall structure may be defined by complex functions that depend on other fields. These dependencies are difficult for existing tools to capture, primarily because they may be a result of several operations performed on the value or length of many fields, such as in checksums. Generative Adversarial Networks (GANs) have been shown to have the ability to learn to generate samples that are similar to the data given to them. GANs traditionally have been used in image processing to create generative models of images. We leverage this capability in a novel way for the purposes of learning the message format of an unknown protocol. Ground-truth sample messages of the unknown protocol are provided to the GAN system. We show that ProtoGANist is able to identify and learn about complex message format features. We demonstrate

that this feature of ProtoGANist is able to outperform other state-of the-art tools in this manner with a separate testing system. This testing system is able to produce protocols with different characteristics to test the complexities that may exist in protocol message formats.

Table of Contents

Acknowledgements **ii**

Abstract **iv**

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

1.1 Security Testing of Software 2

1.2 Security Testing in Black Box Systems 3

1.3 Contributions 6

1.4 Thesis Outline 7

2 Protocol Reverse Engineering **8**

2.1 Protocol Reverse Engineering Overview 8

2.1.1 Goals of Protocol Reverse Engineering 8

2.1.2 Inference Process 9

2.2 State of the Art Tools 10

2.2.1 Terminology 11

2.2.2 Techniques 11

2.2.3 Limitations in State of the Art 16

3 Generative Adversarial Networks **17**

3.1 Architecture Overview 17

3.1.1 Adversarial Approach 19

3.2 Wasserstein Generative Adversarial Networks 20

3.3 Security-Related Applications of GANs . . . 21
3.3.1 Avoiding Detection of Malware . . . 21

4 Problem Overview **23**
4.1 Overview . . . 23
4.2 Protocol Message Format Definition . . . 24
4.3 Goals . . . 26

5 System Overview **28**
5.1 ProtoGANist Protocol Format-Learning System . . . 28
5.1.1 Challenges . . . 28
5.1.2 System Setup . . . 30

6 Evaluation **32**
6.1 Experiment Setup . . . 32
6.2 Methodology . . . 32
6.3 Experimentation Process . . . 35
6.4 Synthetic Protocol Modeler . . . 36
6.5 Protocol Message Format Properties . . . 37
6.5.1 Experiment 1: Ranges (ASCII Printable Characters) . . . 37
6.5.2 Experiment 2: Basic Operations (XOR Bit Operation) . . . 38
6.5.3 Experiment 3: Checks (TCP Checksum) . . . 40
6.5.4 Experiment 4: Stress Test (Controller Area Network) . . . 41
6.6 Investigating Difficulties with CRC . . . 45
6.6.1 Training Set Size vs CRC-4 . . . 47
6.7 Memorization vs Learning . . . 48

7 Discussion and Future Work **51**
7.1 Findings . . . 51
7.1.1 Limitations and Improvements . . . 52
7.2 Future Work . . . 53

8 Conclusion **55**

Bibliography **57**

List of Tables

Table 2.1 Protocol RE tool inference process 15

List of Figures

Figure 1.1 Approach to test software interacting w/ protocols. 2
Figure 1.2 Approach to test software in black box scenario 3
Figure 1.3 TCP checksum toy protocol 6

Figure 2.1 Inputs and outputs for protocol RE tools 9
Figure 2.2 Definitions of common protocol RE terms 11
Figure 2.3 Needleman-Wunsch Algorithm 12

Figure 3.1 Overview of a GAN 18

Figure 4.1 Definition of protocol message format 25

Figure 5.1 Handling of variable-length messages 29
Figure 5.2 ProtoGANist System Overview 30

Figure 6.1 Types of content to test 33
Figure 6.2 List of experiments 35
Figure 6.3 Synthetic protocol modeler 36
Figure 6.4 Experiment 1 protocol 37
Figure 6.5 Experiment 1 evaluation 38
Figure 6.6 Experiment 2 protocol 39
Figure 6.7 Experiment 2 evaluation 39
Figure 6.8 TCP checksum function 40
Figure 6.9 Experiment 3 protocol 40
Figure 6.10 Experiment 3 evaluation 41
Figure 6.11 Experiment 4 protocol 43

Figure 6.12 Experiment 4 protocol - length 43
Figure 6.13 Experiment 4 evaluation - length 43
Figure 6.14 Experiment 4 protocol - static 44
Figure 6.15 Experiment 4 evaluation - static value 44
Figure 6.16 Experiment 4 protocol - check 45
Figure 6.17 Experiment 4 evaluation - check 46
Figure 6.18 Experiment 4 protocol - check 46
Figure 6.19 Experiment - CRC-4 47
Figure 6.20 CRC-4 accuracy based on training set size 48
Figure 6.21 Amount of generated messages that were new content 49
Figure 6.22 Accuracy of generated messages with new content 50

Figure 7.1 Summary of findings 52

1

Introduction

Ensuring the security of software for bugs is a critical step in the software engineering process. Many software artifacts interact with network protocols, allowing users and other systems to remotely interact with it. It is thus important to test the code interacting with these protocols in the software testing process. In some cases, this objective is difficult to achieve simply because the protocol is proprietary or poorly documented. An analyst would not have little intuition on how to test software on a black box system with black box protocols.

Current solutions to this issue revolve around reverse engineering the protocols [9][27][32]. However, many of these techniques do not fully characterize the complexities of protocols and furthermore may have requirements outside the scope of common protocol reverse engineering scenarios.

Our goal is to identify an alternative technique to perform this protocol reverse engineering in black box scenarios. We look to achieve a sufficiently high accuracy such that our solution is better than naive generation.

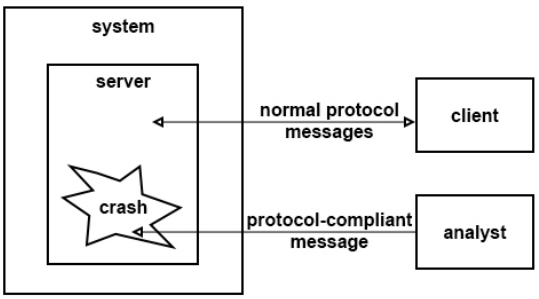

Figure 1.1: Software testing techniques such as fuzzing involve an analyst sending protocol-compliant messages to a system & seeing if any trigger unexpected behavior in the system.

1.1 Security Testing of Software

There are no guarantees that the content received by a system over a network is of the expected format or that it will not trigger unintentional behavior. Many reported Common Vulnerabilities and Exposures (CVEs) stem from a specially-crafted protocol packet triggering unintended behaviors in the receiving system. Some relevant CVEs include the following:

- **2010:** A case study of Tire Pressure Monitoring System (TPMS) had an occurrence where the Engine Control Unit (ECU) was bricked by fuzzing TPMS packets [15].
- **Late 2018:** An ICMP packet was demonstrated to be able to trigger an out-of-bounds write in the Apple XNU kernel [3].
- **Feb 2019:** Several CVEs in Remote Desktop Protocol (RDP) clients were found [16].

The typical setup for software testing is depicted in Figure 1.1. A **client** sends protocol messages following the **protocol specification** to a **system**. The system

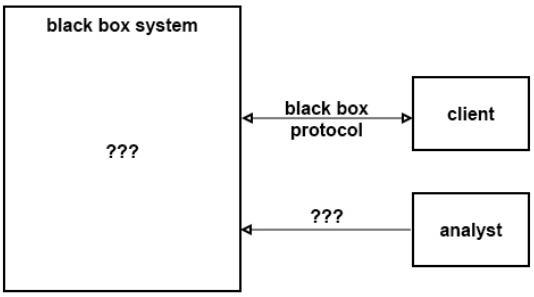

Figure 1.2: A black box system is such that there is no prior knowledge of or access to the software running on the system. A black box protocol is a proprietary or poorly-documented protocol that we have no knowledge about the specification.

has some **software** running on it which would send and receive messages.

An analyst tests a system by sending protocol messages to it either with a tool or by hand. The analyst's goal is to induce some unintended behavior on the system with these messages. Unintended behavior can vary between different situations. Crashes in software are one such common unintended behavior. Another example may be the engine unexpectedly shutting off in a car in the middle of the road. We will assume for the purposes of this work that the analyst is interested in testing the functionality of the software for unintended behavior beyond the checks that software may make on the input.

1.2 Security Testing in Black Box Systems

A **black box system** is one which we have no knowledge of or access to the software running on the system. This means we cannot perform any analysis directly on the software itself. The black box system interacts with client via some **black box protocol**. We have no prior knowledge of the specification of the black box protocol. The described black box scenario is depicted in Figure 1.2.

Black box systems and protocols require security testing as well. Oftentimes,

security and privacy were not prioritized in many of these systems during their development. Increases in interconnectivity of devices have both significantly increased the attack surface and increased the potential for vulnerabilities.

One such case of this has been in the automobile industry. Automobile systems were created with a focus on speed and reliability, not security and privacy. The development of automobile entertainment systems, Vehicle-to-Vehicle (V2V) communication, and automobile automation have led to serious concerns about security in cars. Miller and Valasek demonstrated the ability to remotely hack and control an automobile in their 2015 work [26].

Many of the protocols used in automotive systems are proprietary. For instance, the Tire Pressure Monitoring System (TPMS) protocol is one such proprietary protocol not equipped with security or privacy features. A 2010 TPMS case study demonstrated the many security and privacy issues in the car [15]. Most notably, they demonstrated that sending many random TPMS packets to the system could brick the Engine Control Unit (ECU) [15].

Another realm in which black box protocols have been deployed is the Internet of Things (IoT) field. The IoT space has rapidly expanded in the past several years. This has led to many issues in security and privacy, namely in the Mirai botnet incident [1]. Furthermore, proprietary and poorly-documented protocols are the norm in the IoT space [17], adding an additional layer of difficulty to security testing of these systems.

TPMS and many of these other proprietary protocols are important to investigate because they are a largely unexplored attack vector [15] [26]. In order to be able to efficiently test the software using these protocols, we would have to reverse engineer the protocol format to understand the expected format of the messages. This process can take anywhere from weeks to months to years depending on the protocol, amount of information available, and skill level of the analyst performing

the reverse engineering. In the case of the Samba project, it took approximately 12 years to reverse engineer the Microsoft SMB protocol [35].

Security testing of these systems is difficult without any prior knowledge of the protocol. Software is typically structured such that after receiving user input, it checks that parts of the message follow an expected format. For instance, software may expect a certain field to contain an integer number and check for such. If the sample messages used to test software do not follow the format, they are immediately thrown out. Therefore, for a message to be effective in testing a system for security vulnerabilities, it must align closely with the protocol specification.

In the case of a black box scenario, we do not have access to the software interacting with the protocol, so we do not know the types of protocol specification checks that are performed. Furthermore, the protocol the software interacts with is a black box protocol so we have no information about how the protocol messages should look.

Current techniques involved manual testing, fuzzing, static analysis, or dynamic analysis [22]. Effective fuzzing is only possible if we have information about the software binary either through prior knowledge or performing static or dynamic analysis on the software binary. However, in the black box scenario we presented, the best we can do is blind fuzzing since we have no prior knowledge about the system or the protocol. This is because other fuzzing and software testing techniques typically require a binary to statically or dynamically analyze. In naive testing, we conceive every possible combination of input and send it to the system to look for unintended behaviors. This naive testing is inefficient and highly unlikely to find unintended behavior.

Let us present a toy protocol to highlight the issue of naive testing. The toy protocol consists of three fields as represented in Figure 1.3. The first two fields accept any value. The third field's value is the TCP checksum function on the values

Figure 1.3: A toy protocol consisting of two 8-bit fields accepting any value and a third 16-bit field whose value is the TCP checksum of the first two fields.

of the first two fields. To pass checks on a message following this specification, the third field must be the TCP checksum of the first two fields. For a combination of fields, if we had to guess the TCP checksum, we would get it correct $\frac{1}{2^{16}} = \frac{1}{65536}$ of the time. If this was extended to more complex protocols with many fields each with complex values and lengths, guessing a message that passes all the checks would be highly unlikely.

1.3 Contributions

The protocol reverse engineering process is as follows. First, a model is created that describes the protocol in some way. Messages are then synthesized from this model to test software. In this thesis, we apply a deep learning technique, Generative Adversarial Networks [13], to the black box protocol reverse engineering problem. We demonstrate that a system using this technique is able to implicitly learn the protocol message format of a black box protocol by skipping over the modelling stage straight to the synthesis of messages.

The specific contributions of this thesis are as follows:

- We create a formal definition of protocol message formats and a synthetic protocol modeler to generate sample messages of that format.
- We built ProtoGANist, a protocol message format-learning system that applies Generative Adversarial Networks.
- We set up an experimental framework that highlights different aspects of the

protocol message format. From these experiments, we gained insight on the limitations of ProtoGANist, specifically examples of what it can or cannot learn.

1.4 Thesis Outline

This thesis is outlined as follows. Chapter 1 introduces the general setup of software verification and the overall motivation behind this work. Chapter 2 goes into the goals and state of the art of current protocol reverse engineering. We specially focus on black box protocol reverse engineering state of the art that follows our particular scope. Chapter 3 describes a new application of Generative Adversarial Networks to solve the black box protocol reverse engineering problem. Chapter 4 gives an overview of the problem. Chapter 5 gives an overview of our system using the proposed Generative Adversarial Networks technique. In Chapter 6, we perform the evaluation of our system. The results are discussed in Chapter 7.

2

Protocol Reverse Engineering

There has been extensive work in protocol reverse engineering [9][27][32]. The techniques proposed in these works address the large variety of content, formats and properties, use cases, restrictions, and more in protocols. This chapter first presents an overview of protocol reverse engineering and the state of the art with respect to the black box scenario we presented in Section 1.2.

2.1 Protocol Reverse Engineering Overview

2.1.1 Goals of Protocol Reverse Engineering

The overall goal of protocol reverse engineering is to infer some information about a protocol. The exact goal for a protocol reverse engineering tool varies depending on the use case it is geared towards. We focus on tools that automate the protocol reverse engineering process.

Tools may look to reverse engineer a protocol to model a network for the purposes of simulating a given protocol [9]. Models can be used to simulate the network to allow for convincing interactions with attackers such as in honey pots [20][21] or

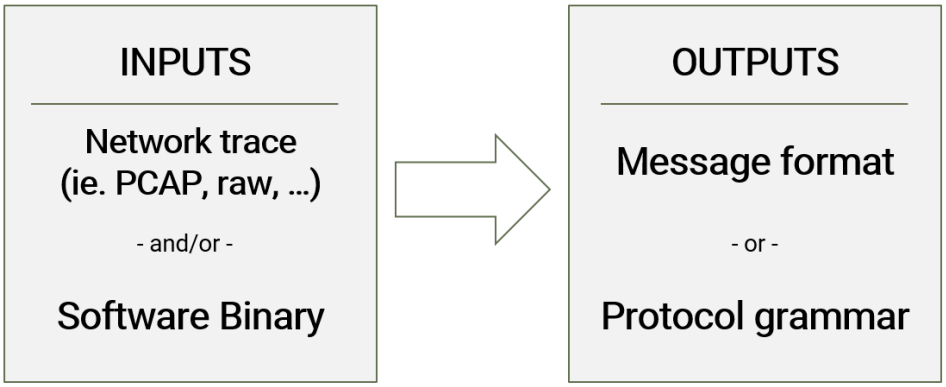

Figure 2.1: A summary of different types of inputs and outputs protocol reverse engineering tools may have.

to gain understanding about malware and botnets [7][8]. Information about the format of messages can also be used to interface with software [35] or to identify vulnerabilities [6][10].

2.1.2 Inference Process

Protocol reverse engineering tools can be summarily categorized by their inference process. The inference process consists of the inputs, outputs, and analysis techniques. Figure 2.1 depicts the input and outputs in these tools.

Input The input to a tool may be a **network trace** and/or **software binary**. A network trace is a series of protocol messages in some format such as a packet capture. For tools that analyze a network trace, they will identify patterns that indicate a format within them. The software binary is the software running on the system that interacts via the protocol messages. Tools that take a software binary can either perform static analysis on the content of the binary or dynamic analysis by examining application execution traces.

Output Tools will either try to determine the **protocol message format** or the **protocol grammar**. The protocol message format is the format of an individual message of the protocol. The protocol grammar adds another layer of complexity, referring to the protocol message format over time between different messages. For instance, in a protocol grammar, an id field may be some field incrementing by 1 with each message. In the protocol message format, that field would just be considered as an integer field since there is no concept of statefulness between messages. While we leave protocol grammar as future work for ProtoGANist, we believe that it is important to evaluate the feasibility of adding in such a feature.

Analysis Analysis may be done passively or actively. **Passive analysis** is analysis that is done without any feedback or additional information given during the learning process. This may be the case when an analyst is given a packet capture but cannot interact with the system. **Active analysis** is when a system actively seeks out feedback to incorporate into the learning process. An example of active analysis would be creating a network protocol model and then updating it based on live network traffic or performing dynamic analysis of a software binary.

Examples of this would be tweaking a network protocol model based on responses received from a server or dynamic analysis of a software binary.

2.2 State of the Art Tools

We will describe the set of tools designed to solve the black box protocol reverse engineering problem. Furthermore, we will consider how this process can be done using only network message traces as input and the protocol message format as the output.

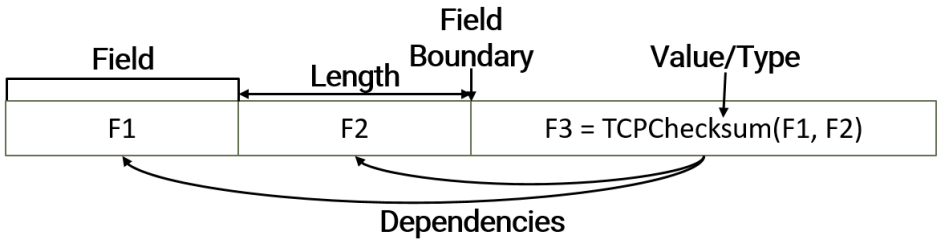

Figure 2.2: Protocol message components commonly referenced in protocol reverse engineering tools.

2.2.1 Terminology

We will define terminology in Figure 2.2 to help characterize the abilities of these protocol reverse engineering tools.

Field A primitive segment of the protocol in which data is stored.

Field Boundary The point of separation between distinct fields.

Length The size of the field. This may vary in different messages of the protocol.

Value/Type A functional description of the inhabitants of a field. In Figure 2.2 this happens to be the TCP checksum function.

Dependencies The fields from which the value/type function of a field takes input.

2.2.2 Techniques

There are a wide variety of black box protocol reverse engineering tools. These tools draw inspiration from many other fields such as bioinformatics, natural language processing, image processing, and more to approach this problem.

We will focus on three of these tools that are available and open-source. A summary of the techniques used by other tools is in Table 2.1.

Protocol Informatics Project (PIP)

PIP[4] applies bioinformatics algorithms such as **Smith Waterman** [33], **Needleman-Wunsch** [28], and **UPGMA** [29] to identify fields within variable-length messages. First, the Smith Waterman algorithm is used to perform local alignment of message bits to identify messages with similar subsequences. This can be used to cluster similar messages into families. Next, the Needleman-Wunsch algorithm is used to perform global alignment for messages within a cluster. Needleman-Wunsch is used in bioinformatics to align protein sequences in the most likely configuration. In PIP, messages are aligned from beginning to end such that similar components are grouped together with padding in between. This process can be seen in the example from the PIP paperFigure 2.3. Lastly, UPGMA can be used to create a phylogenetic tree of messages. UPGMA uses multiple sequence alignment to create a phylogenetic tree showing the similarity in structure between messages.

```
1 GET /index.html HTTP/1.0
2 GET /__________ HTTP/1.0
```

Figure 2.3: An example of running the Needleman-Wunsch algorithm on "GET / HTTP/1.0" and "GET /index.html HTTP/1.0".

This paper was an early protocol reverse engineering paper and the first to thoroughly examine the application of bioinformatics techniques to the protocol reverse engineering problem. Later tools have been developed atop the success of PIP, oftentimes applying the proposed algorithms as an initial step in identifying the approximate locations of fields.

However, PIP struggles with large sets of binary data. If the data is not easily characterizable by a simple function, it is difficult for it to be properly interpreted as a field. Furthermore, aligning data of a single field does not take into account dependencies on other fields.

Netzob

Netzob [5] is a tool that applied the findings of PIP. Netzob uses the bioinformatics techniques outlined in the PIP paper to perform initial field alignment. Netzob applies contextual information about the fields in order to improve clustering of fields and learn additional information such as basic dependencies and types. **Maximal Information Coefficient (MIC)** [30] is used to identify dependencies within data even if they are not linear. MIC allows Netzob to identify fields and field boundaries and characterize the values as functions dependent on other fields.

However, Netzob relies on these functions to be predefined. The code contains predefined attributes, relationships, and types which are then efficiently searched for in the messages using the Maximal Information Coefficient technique. If the function is not defined, Netzob does not know to search for it.

This issue stems from a naive solution in protocol reverse engineering where one would conceive every possible function in existence and test every subarray of the message to see if the values in the field match the function. While Maximal Information Coefficient prevents us from having to naively compare the function with every subarray of the message, we still have no method of determining what functions characterize the field values in the protocol.

NEMESYS

Lastly, NEMESYS [11] is able to efficiently determine field boundaries by examining the distribution of changes in the bits throughout the protocol message. By inferring

format information from values within a single message, this technique allows for analysis to be done per message rather than between every message. The approach worked well in inferring field boundaries in binary protocols, which many other tools had difficulties in handling.

While NEMESYS is able to identify field boundaries, it cannot give information about the protocol message format as a whole. In addition, cases with consecutive fields of the same data may be difficult to identify as separate fields.

	Techniques	**Uses**
PIP	Smith-Waterman	Local alignment
	Needleman-Wunsch	Global sequence alignment
	UPGMA	Phylogenetic tree (historical similarity)
ScriptGen	Needleman-Wunsch	Global sequence alignment
	Byte type, frequency, variability	Grouping similar bytes into fields
RolePlayer	Needleman-Wunsch	Global sequence alignment
	Pairwise constraint matrix	Alignment via field semantics
Discoverer	Needleman-Wunsch	Global sequence alignment
	Parsing left to right	Cluster messages of similar format in parsing order
LZfuzz	Lempel-Ziv compression	Packet segmentation into tokens
ASAP	Statistical t-test	Identify non-volatile or constant strings
	Matrix factorization	Get building blocks (fields) of message
Biprominer	Transition Probability Model	Determine probabilities of field values
ProDecoder	Needleman-Wunsch	Global sequence alignment
	N-gram sequences	Find keyword fields
Netzob	Needleman-Wunsch	Global sequence alignment
	Maximal Information Coefficient	Retrieves non-linear dependencies of functions
AutoReEngine	Apriori	Identify protocol keywords
PULSAR	Non-negative matrix factorization	Part-based clustering into similar structures
	DFA minimization	Minimize DFA representing message language
WASp	Byte analysis (feature, range, etc.)	Identify contents of fields
	Shannon entropy	Determine fixed byte values
	N-gram test	Consistency of packet contents
NEMESYS	Delta of bit value congruence	Identify field boundaries
	Gaussian filter	Determine likelihood of field boundaries

Table 2.1: The inference approaches protocol reverse engineering tools take.

2.2.3 Limitations in State of the Art

We can summarize the approach of the majority of these tools as follows:

1. Use bioinformatics algorithms to perform initial field identification.
2. Apply some additional technique such as MIC and delta bit congruence to further refine this information.

From our review of current literature, none of the tools performing black box protocol reverse engineering for protocol message formats were able to fully characterize complex functions or dependencies within protocol messages. Referring back to our toy TCP checksum protocol in Figure 1.3, PIP and NEMESYS would likely struggle with identifying field boundaries from randomized binary. Furthermore, both would not be able to model the functional dependency between fields. Netzob's application of Maximal Information Coefficient was a step in the right direction to identify field dependencies, but it relied too heavily on the user knowing the types of functions present in the protocol message format.

Additionally, current tools for software vulnerability testing would not be able to function well in our proposed black box scenario. **Fuzzers** work by creating content to use as input into software binaries and then looking for unintended behavior such as crashes. One state of the art fuzzer, American Fuzzy Lop (AFL), generates seeds for fuzzing by mutating inputs and focusing on testing new code paths [36]. However, AFL requires access to the software binary for information that it uses to create inputs. Other fuzzing and security testing techniques also face the issues of ineffectiveness in a black box system [22].

3

Generative Adversarial Networks

Generative Adversarial Networks are a type of machine learning system that perform unsupervised learning. GANs employ an adversarial process to produce a generative model for a data distribution.

3.1 Architecture Overview

We will describe the architecture of Generative Adversarial Networks as depicted in Figure 3.1.

The **generative neural network**, or **Generator**, aims to construct a generative model characterizing data distributions [12][13]. A generative model is a model which learns about the distribution of a data set such that it can generate new samples from the distribution [13]. Specifically, the Generator attempts to create a mapping between a latent space and a data distribution. The Generator can produce new samples from this generative model that fit the data distribution.

The **discriminator neural network**, or **Discriminator**, looks to identify similarities between the data distribution and a new sample of data by discriminating between the two [12][13]. The discriminative model aims to identify the difference

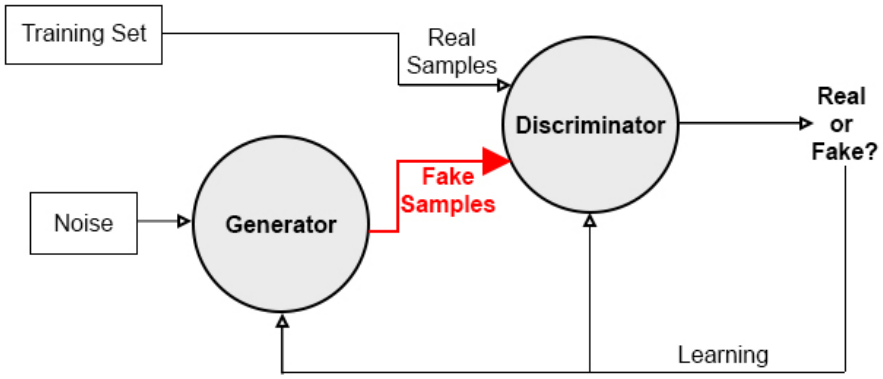

Figure 3.1: GANs consist of a Generator and Discriminator neural network. The Generator learns to generate fake samples similar to the real samples and the Discriminator learns to discriminate between real and fake samples.

between samples of two distributions — in our case, between the training set data distribution and fake samples.

Prior work had focuses specifically on either discriminative or generative models [13]. Whereas generative models provided the useful feature of generating new data from a sample data distribution, discriminative models found more success at the time.

Generative Adversarial Networks (GANs) were proposed to leverage the features of both generative and discriminative models [13]. GANs employ an adversarial approach between generative and discriminative networks. A high-level overview of this adversarial approach is as follows:

1. The Generator attempts create a generative model characterizing a data distribution.

2. Samples produced from the Generator's generative model as well as samples from the training set are presented to the Discriminator.

3. The Discriminator attempts to match the sample to its understanding of the training set distribution or the generated data distribution.

4. Based on whether the Discriminator's answer is correct, the Generator and Discriminator learn to improve their approach and better achieve their goals.

This new technique has been largely successful, particularly in the area of image generation. Prior work has demonstrated the ability to transform images such as in CycleGAN [37], which could transform images of horses to look like zebras or vice versa. There has also been work in creating non-existent variations images such as in StyleGAN [18] which could produce images of non-existent faces. Overall, Generative Adversarial Networks have proven to be successful in characterizing data distributions and generate realistic mimics of data.

GANs are currently only able to **implicitly learn** about the data they are modelling. This means that while Generative Adversarial Networks are able to learn to mimic data, they are not able to describe the relationship between the input and output. For instance, in the context of StyleGAN, the GAN system is not able to describe the choices it made in creating images of faces (ie. nose, eyes, etc.) even though it can generate non-existent faces. If GANs were able to **explicitly learn**, they would be able to describe their understanding of eyes, noses, etc. to the user.

3.1.1 Adversarial Approach

The adversarial component of the learning process follows a minimax game characterized by Equation 3.1.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{3.1}$$

The Discriminator wants to maximize the probability that it correctly characterizes the data and identifies a sample. This is represented as $\mathbb{E}_{x\sim p_{data}(x)}[\log D(x)]$,

where $p_{data}(x)$ is the distribution over the real data samples and $D(x)$ is the probability that x came from the real data sample distribution rather than the generated data distribution p_g.

On the other hand, the Generator wants to minimize the likelihood of the Discriminator giving the correct answer. This is represented as $\mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$ where $G(z)$ is the fake sample generated from noise and p_z is the data distribution over the noise z.

In short, the Discriminator and Generator have the following goals [13]:

- The Discriminator wants $D(x)$, the probability of correctly classifying the real samples to be 1.
- The Discriminator wants $D(G(z))$, the probability of incorrectly classifying fake generated samples as real samples to be 0.
- The Generator wants $D(G(z))$ to be 1.

The loss function in GANs is a way of quantifying the similarity between the distribution of real samples and the distribution of generated fake samples. This provides a way to quantify $D(x)$ in the minimax game equation. Standard GANs use Jensen-Shannon Divergence as the loss function to measure the distance between the two distributions [13].

3.2 Wasserstein Generative Adversarial Networks

While Generative Adversarial Networks are a promising technique to characterize a training set data distribution, there are several problems with their architecture noth in the context of our domain and in general.

Both the Discriminator and Generator attempt to find the Nash equilibrium of the two-player game as described in Section 3.1.1 [31]. However, they do so independently

which means convergence is not guaranteed [31]. GANs could also experience "mode collapse" where the variety of generated samples is limited to a few different samples [2]. The loss functions typically used with GANs, Kullback-Leibler and Jensen-Shannon divergence, are not able to properly model discrete distributions [2].

Wasserstein Generative Adversarial Networks (WGANs) proposed the usage of **Wasserstein Distance** or **Earth Mover's distance** to provide a continuous measure for discrete distributions [2]. While WGANs may become unstable and thus are imperfect [2], they have solved many of the issues faced in the original proposed Generative Adversarial Network both in effective sample generation and dealing with discrete distributions.

3.3 Security-Related Applications of GANs

3.3.1 Avoiding Detection of Malware

Malware Detection Avoidance MalGAN [14] aims to solve the issue of attackers sending malware to a system. The attackers oftentimes do not know what the internals of malware detection on a system are and thus must treat it like a black box. The findings suggested that MalGAN was successful at circumventing machine learning-based malware detectors.

MalGAN provided some unique insight on how Generative Adversarial Networks could be applied as a technique for attackers. This raises the possibility that ProtoGANist could be used by both analysts and malicious attackers.

Seed Generation SmartSeed [23] provides a unique approach to generating high-value seeds by applying Wasserstein Generative Adversarial Networks to the problem. High-value seeds are those that have a higher probability of triggering crashes or reaching new code paths. SmartSeed was able to generate high-value seeds at a higher rate than other seed generation techniques. These seeds were able to produce

twice the number of crashes and 5,040 more code paths than the best competing strategies.

This work provides a similar application of Wasserstein Generative Adversarial Networks for performing security verification. This suggests that ProtoGANist would be a promising approach to fuzzing software working with protocols.

4

Problem Overview

4.1 Overview

We present Generative Adversarial Networks as a technique in solving the black box protocol reverse engineering problem. In particular, we believe that GANs can bypass the modeling stage and skip straight to synthesizing messages of the protocol format. This characteristic is useful in fuzzing where one would not necessarily need the model to fuzz the software. The intuition underlying this decision stems from requirements needed to fully solve this problem and the properties of GANs that meet these requirements.

Requirement 1 *The technique must be able to characterize complexities with the protocol message format including properties defined as functions and determine dependencies between fields.*

Generative Adversarial Networks are able to learn how to model complex data distributions that may not have an obvious function generation [13]. For instance, modeling the exact features that make up a realistic face is not trivial, but StyleGAN was able to learn to develop such a generative model [18]. Specifically, they

are able to characterize distributions of data that may not have a straightforward method of characterization.

Requirement 2 *The technique must be able to characterize the protocol message format and encode this knowledge in some format.*
Generative Adversarial Networks can implicitly learn about the data distribution by presenting generated samples following the distribution.

Requirement 3 *The technique must take only protocol messages for its input. There is no prior knowledge of either the protocol or the software that interacts with the protocol.*
Generative Adversarial Networks learn only by the data in the training set given to them.

Requirement 4 *The technique must not involve human participation in the learning phase.*
Generative Adversarial Networks learn via an unsupervised learning process.

Thus we believe that Generative Adversarial Networks would be capable of addressing the black box protocol reverse engineering problem.

4.2 Protocol Message Format Definition

First let us present a formal definition for the protocol message format to identify the features we would like to see if GANs can model. This description is depicted in Figure 4.1.

Protocol Message Format The protocol message format is the format of an individual message for a particular protocol. This differs from protocol grammar,

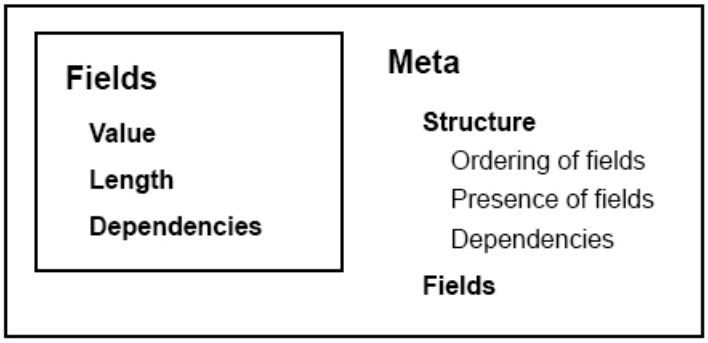

Figure 4.1: A protocol message format is composed of a meta structure and several fields along with the characteristics that make up each.

which extends protocol message format to the format of messages over time. We focus solely on the protocol message format as Generative Adversarial Networks have not been used to solve the black box protocol reverse engineering problem and thus we would like to first examine the feasibility of such an approach.

Fields A protocol message format is made up of a series of fields. Each field conveys a different information in the protocol message. Fields have several properties that define the content they carry.

- **Field Dependencies:** References other fields in the message needed to compute the value of this field. The field dependencies of a TCP checksum would be the fields the checksum is used to detect errors.
- **Value:** The function that characterizes the value of this field. The function may take other fields' values or lengths as input.
- **Length:** The function that characterizes the length of this field. The function may take other fields' values or lengths as input. For instance, a data field in CAN varies is a byte-aligned length from 0 to 8 bytes.

Meta An individual protocol format will contain data beyond what is needed for its constituent fields. The meta segment of the message describes the manner in which fields are organized within a message.

- **Fields:** The fields which may be present within the function and their properties. Note that not all fields may be present in a particular message of the protocol.
- **Structure:** The functional definition of the presence and ordering of fields based on the values of field dependencies. For any particular message, not all fields may be present. The concept of structure encompasses this idea. HTTP may have different structures depending on the type of HTTP method such as GET or POST.
- **Structural Dependencies:** As with field dependencies, the structure of a message itself may be itself dependent on fields' values or lengths. The structure of HTTP may be dependent on the context of the method field such as GET or POST.

The formal definition of the protocol message format allows us to understand the types of properties our system may encounter with real protocols. This allows us to define experiments that test these properties to understand the limitations in the system.

4.3 Goals

For this work, we look to investigate whether providing protocol messages to a Generative Adversarial Network system would result in it generating messages that fit the protocol message format specification. The characteristics of Generative Adversarial

Networks support the requirements needed to solve the black box protocol reverse engineering problem, provided that it is able to accomplish this goal.

We also looked to understand the limitations of such an approach. State of the art black box protocol reverse engineering tools struggle in characterizing complex functions and dependencies within protocol messages. We were interested in seeing if our approach would have similar issues.

5

System Overview

This chapter covers an overview of the ProtoGANist system. This includes our approach the ProtoGANist system itself and specific challenges encountered in the development of the system.

5.1 ProtoGANist Protocol Format-Learning System

We will now describe the setup of our GAN-based protocol format-learning system. The system can be generalized to three components: the training set or input data, the learning system, and the evaluation of the output.

5.1.1 Challenges

First, we will describe the challenges encountered in these components during the development of our system.

Data Representation

We decided to represent our data as bits rather than bytes. Protocols oftentimes define 1 bit-long flag fields or other fields of some non-byte aligned length. We believed

that representing the data as bits would allow the GANs to better understand the content and dependencies within these types of fields. Each protocol message is then represented as the 0's and 1's (bits) that make it up.

Variable Length Data

The GAN setup that we used required that training set be provided as a 2-D array with a fixed length for each row (message). However, protocol messages may vary in length as fields may contain different-sized content and fields may or may not be included in a message. Furthermore, patterns of 0's and 1's cannot be used to indicate the end of a message. Protocol definitions could be such that the particular pattern of 0's and 1's could potentially show up as part of the protocol specification.

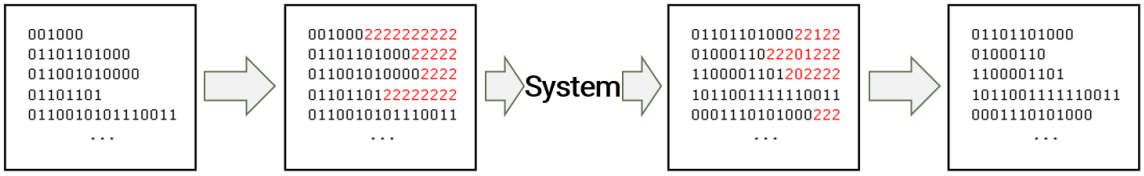

Figure 5.1: Variable-length messages are supported in the GAN system by padding shorter messages with 2's.

To circumvent these issues, we introduced "2 padding". Since messages are defined with only 0's and 1's, a 2 would clearly not be part of the protocol message.

The training sets are not created or collected containing the 2's. If a training set contains variable-length messages, it is padded to the longest message using 2's as a pre-learning step. Following this process, the data was able to be fed into the system. This process is shown in Figure 5.1.

The system would have to learn about the 2's padding as part of the learning process. The resulting generated samples it produces should contain the 2's as padding. Everything from the first 2 onwards is stripped from the end of the message to inter-

pret variable-length messages. Note that there may be 0's and 1's included in this stripped-out data.

Discrete Distributions

The metric used as a loss function in most Generative Adversarial Networks does not perform well when characterizing discrete distributions. This is problematic in our case as the representation of protocol messages, bits 0 and 1, are discrete distributions.

Our solution to this issue is to employ a different metric, Wasserstein distance or Earth Mover's distance, as a loss function in our GAN system [2]. This metric is able to characterize discrete distributions by approximating in between them, as described in Section 3.2.

5.1.2 System Setup

We will now describe the ProtoGANist system setup. The final ProtoGANist system setup is illustrated in Figure 5.2.

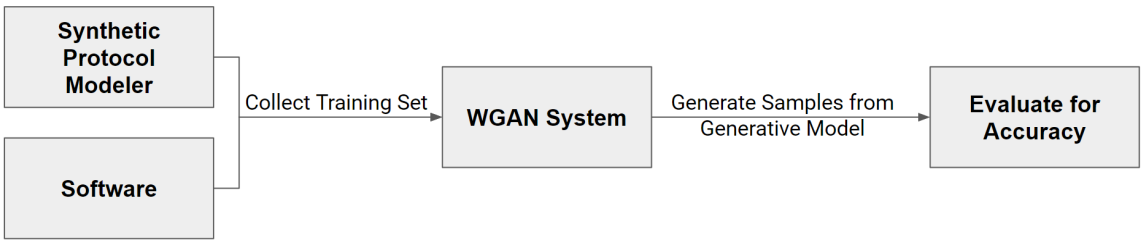

Figure 5.2: ProtoGANist consists of three main components. (1) The training set is created by modeling synthetic protocols or collecting from real protocols. (2) The GAN system learns to characterize the protocol messages in the form of a generative model. (3) Samples from the generative model are evaluated for accuracy of the feature of interest.

Training Set The training set consists of messages from a synthetic or real protocol. In the case of a synthetic protocol, we write a definition for the specification of the protocol message format and use our synthetic protocol modeler to create m sample messages for our training set. For a real protocol, we collect m samples of the protocol for our training set.

Learning System The learning system is a Wasserstein Generative Adversarial Network (WGAN) employing Wasserstein distance as the loss metric. Both the GAN Discriminator and Generator were 6-layer neural networks with a layer dimension of 600. The system is fed the training set of protocol messages as the input. Every i iterations, the Generator in the WGAN system is asked to generate n sample messages based on its current characterization of the training set distribution. These generated samples are considered the output and are evaluated for accuracy.

Output Evaluation Each set of n generated sample messages is evaluated. For a given experiment, we consider one or more several fields of interest where a field of interest has some unique feature we would like to experiment. We evaluate the generated messages by checking if the field or characteristic of interest fits the defined specification. The evaluation is solely dependent on the field of interest rather than the message as a whole as it allows us to evaluate components individually even when there were many complexities in a particular protocol message format. The accuracy for the set of n messages is the total number of messages with a correct field out of the total number of messages. This allows us to examine the accuracy of the WGAN Generator over time in the context of the experiment.

6

Evaluation

This chapter evaluates the proposed ProtoGANist system for accuracy in the context of different properties.

Simpler experiments (up to the stress test experiment) were performed on a Windows 10 laptop with an Intel Core i7-7700HQ CPU @ 2.80 GHz and NVIDIA GeForce GTX 1060 GPU. The latter experiments were performed on a server with an Intel Xeon CPU E5-2698 v3 and a NVIDIA Tesla V GPU, as they required more time to run.

6.1 Experiment Setup

This section describes the approach to experimentation, both in terms of creating and carrying out the experiments.

6.2 Methodology

The properties of the protocol message format defined in Section 4.2 characterize the different content that may be seen in real protocols. We would need to test these properties in order to identify limitations with the system that would become

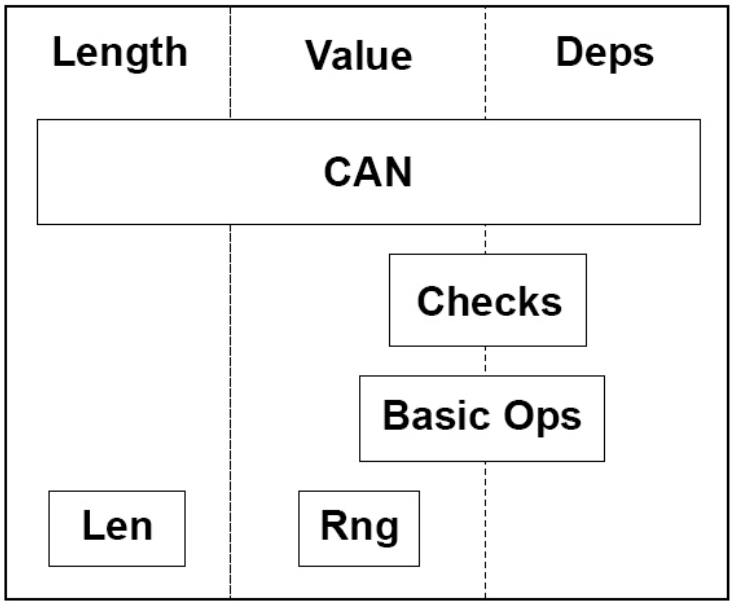

Figure 6.1: The different experiments needed to test length, value, and dependencies. Each of the experiments builds up in complexity by using some aspect of a previous experiment. For example, error detection functions using basic functions.

apparent when working with real protocols.

We had three criteria in identifying the types of features to test in our experiments:

1. The content must be commonly found in real protocols.

2. The content must highlight specific properties within the protocol message format definition. All the content together must test all the properties of interest.

3. The tested features should build up in complexity. For example, we would like to test basic operations before functions that use those operations.

These criteria was characterized into the following types of content to test.

Ranges of Values This content defines particular values that are considered valid. They may be continuous or disjoint ranges or values or static content. Many protocol fields contain this type of content. We decided to test ASCII bytes ranges with a field that accepts any printable character.

Basic Operations This includes bit operation functions such as XOR, OR, AND, etc. and mathematical functions such as addition, subtraction, etc. Basic operations are used in more-complex functions and general use. Note that these functions take in values as input, and thus basic operations spans both value and dependencies.

Checks These include error detection checks such as TCP checksum and Cyclic Redundancy Check (CRC), which are often used in protocols for error detection. These types of functions involve some combination of basic operations, allowing us to build on those tests. Furthermore, checksums have been considered a difficult edge chase in fuzzing and software verification when generating content following an input specification. We picked TCP checksum, a basic 1's complement check using addition for this experiment.

Real Protocol (Stress Test) As a stress test, we take a real protocol that combines several of the properties in the protocol message format definition. The goal of this test is to highlight potential limitations that are not present in the prior simple synthetic experiments. The Controller Area Network (CAN) protocol used in vehicular networks has a combination of features. Not only could the messages vary in length, but one of the fields' values depended on the length. A check in the form of CRC-15 was present in the protocol as well. CRC-15 consists of a combination of bit shifts and other basic operations. Lastly, there was a static set of acknowledgements at the end of every message.

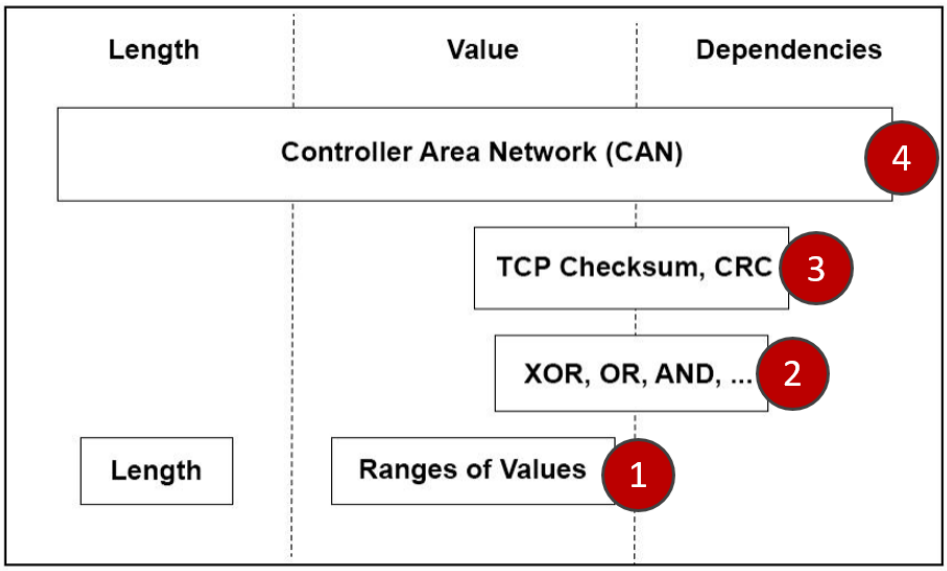

Figure 6.2: Experiments were picked that tested different characteristics of protocol format message (range of values, dependencies, and length) and built up on each other.

6.3 Experimentation Process

We will now describe the parameters for our experiments. Experiments were run in order of increasing complexity.

The first three experiments required synthetic protocols to be defined. The synthetic protocols kept as many components static as possible to focus on the content of interest. The synthetic protocol modeler was used to generate samples following the given specification. The fourth experiment tested a real protocol. We collected randomized messages of the protocol where non-specified fields were randomized. 10,000 samples were generated as the input training set for the WGAN system.

During the learning phase, the WGAN system Generator would output 10,000 samples at a fixed interval of iterations. This allowed us to examine the accuracy of the system over time.

For each experiment, the specific field(s) of interest are examined for accuracy. A message is considered correct if the field follows the given specification. As an example, in our TCP checksum toy example in Figure 1.3, if we were evaluating the TCP checksum feature, a message would be considered correct if its value was the TCP checksum of the other two fields' values. For each 10,000 generated samples, we calculate the accuracy as the number of correct messages out of 10,000.

Experiments were run for approximately 10 hours until the model converged. The exception to this was for cases where we ran for just over 24 hours to debug if a limitation was a result of training time.

6.4 Synthetic Protocol Modeler

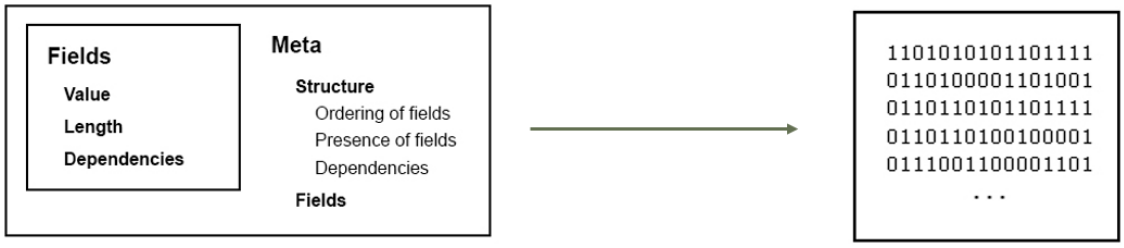

Figure 6.3: The synthetic protocol modeler accepts a synthetic protocol specification following the protocol message format definition and can generate messages following the specification.

In order to fully test our system, we need the ability to work with simple protocols. This allows us to focus the testing on a particular property.

To achieve this, we created a synthetic protocol modeler. This applies our definition of a protocol message format from Section 4.2 as a template. The synthetic protocol modeler would then take this synthetic protocol definition and produces random sample messages following the specification, as depicted in Figure 6.3.

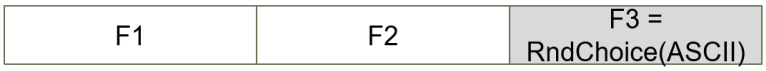

Figure 6.4: Protocol message format for experiment 1.

The synthetic protocol modeler allows us to finely tune the experiments to run on the GAN system. It also provides a way of generating training sets for these protocol messages.

6.5 Protocol Message Format Properties

We will now present the results of the experiments.

6.5.1 Experiment 1: Ranges (ASCII Printable Characters)

Our first experiment consisted of the simplest non-static test; whether the ProtoGANist system can handle accepted ranges (continuous or discrete) of bytes. The synthetic protocol defined for this experiment in Figure 6.4 consisted of two fields with any value and a third accepting printable ASCII characters. There were no field or structural dependencies or dynamic lengths in this setup.

The ASCII printable characters consist of a few discrete ranges of values: a large range for letters, numbers, punctuation, and some whitespace characters such as space and other separate ranges for whitespace characters such as newline (0x0a). This type of distribution would likely have to be characterized as a piecewise-type function since there are non-printable characters like 0x0e. ProtoGANist handled it remarkably well, achieving approximately 90-100% accuracy early on in the learning process as seen in Figure 6.5.

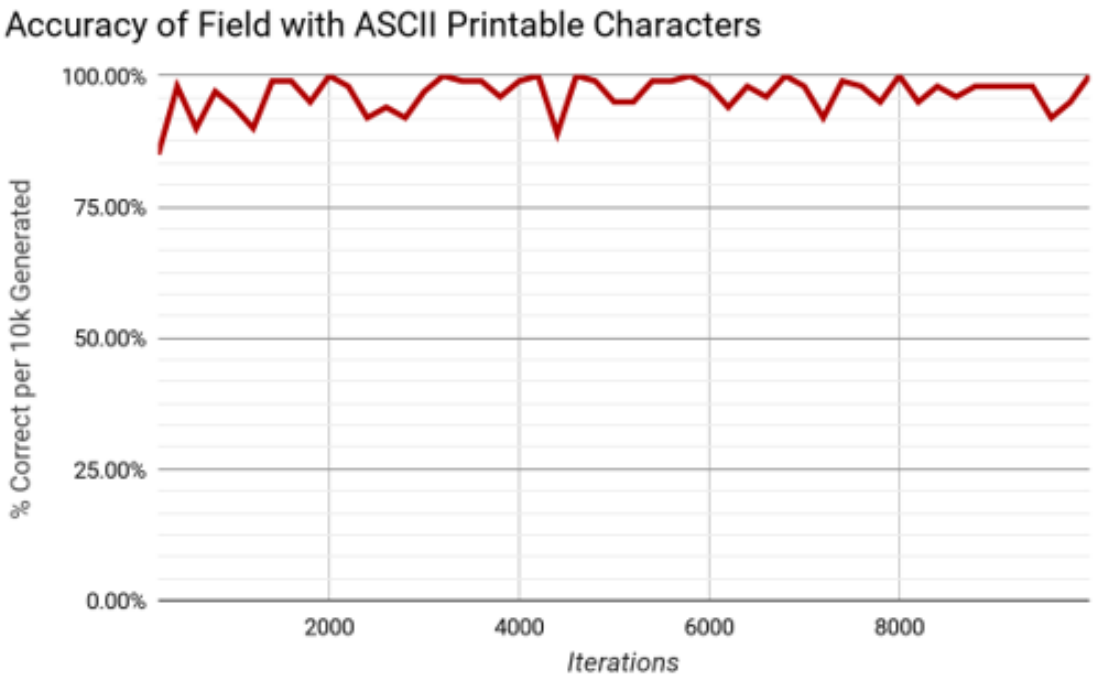

Figure 6.5: Experiment 1: An evaluation of how ProtoGANist handled ranges of printable characters. It handled it remarkably well almost instantaneously, between 90-100% accuracy.

6.5.2 Experiment 2: Basic Operations (XOR Bit Operation)

The second experiment looked to test a basic operation. XOR was picked as it is frequently used in protocols in various forms and within more complex functions such as in encryption and checksums [25]. Furthermore, XOR is a more difficult function to derive than the previous range-inclusion function. The synthetic protocol for this experiment also consisted of two fields of any value and a third field whose value was the XOR of the first two fields as seen in Figure 6.6. This experiment added on the complexity of dependencies within the protocol message format but retained the static length.

ProtoGANist took significantly longer to accurately characterize this synthetic

F1	F2	F3 = F1 ^ F2

Figure 6.6: Protocol message format for experiment 2.

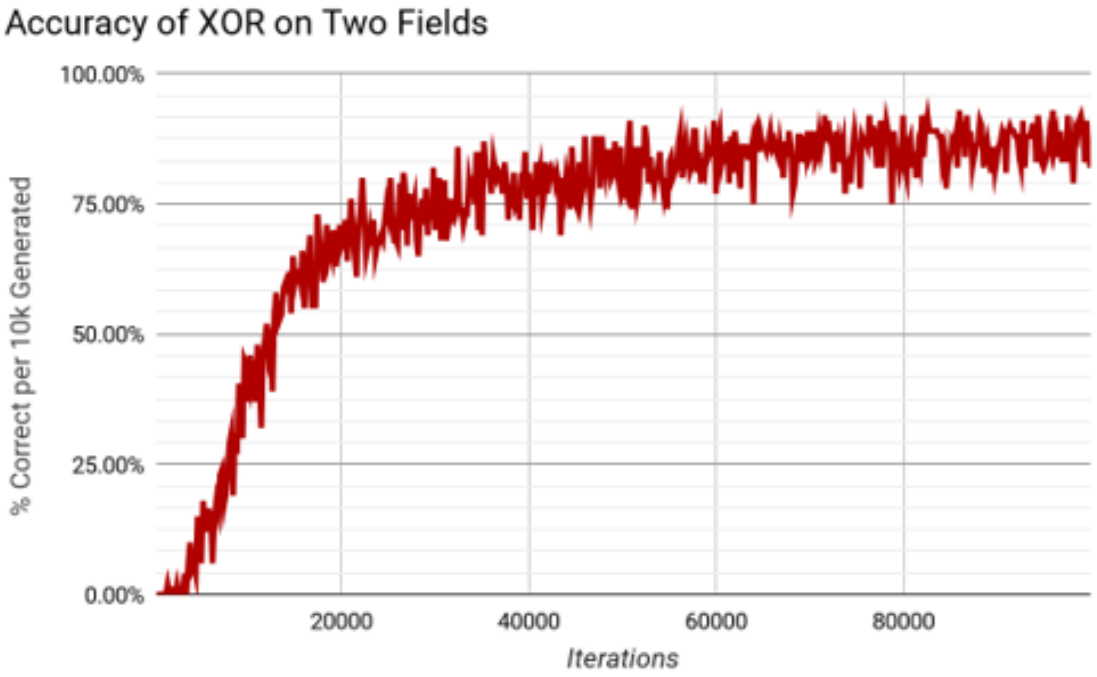

Figure 6.7: Experiment 2: An evaluation of how ProtoGANist handled XOR as a value function. It struggled a bit with learning but was still able to characterize the XOR.

protocol and even then, was able to achieve 80-90% accuracy as seen in Figure 6.7. Data distributions consisting of XOR may be difficult to model as a function. The model for this experiment took much longer to converge than the ASCII printable character experiment which converged almost immediately.

6.5.3 Experiment 3: Checks (TCP Checksum)

In our experiment for checks, we picked TCP checksum. TCP checksum is a 1's complement addition-based checksum function. The code for this is in Figure 6.8. This type of checksum is used as an error detection technique in protocols such as IP, TCP, and UDP. This protocol message format also retained a static length. The protocol message format followed that of the prior experiments; two fields accepting anything and a third field computing the value as the TCP checksum function on the first two fields in Figure 6.9. Note that while the first two experiments had fields of equal length (8 bits), the TCP checksum is 16 bits long and thus longer than each of the first two fields.

```
def tcp_chksum(msg):
    s = 0
    for i in range(0, len(msg), 2):
        if (i+1) < len(msg):
            a = ord(msg[i])
            b = ord(msg[i+1])
            s = s + (a+(b << 8))
        elif (i+1)==len(msg):
            s += ord(msg[i])
        else:
            raise "Error"
    s = s + (s >> 16)
    s = ~s & 0xffff
    return s
```

Figure 6.8: The TCP checksum function as Python code.

Despite being a combination of basic operations, ProtoGANist's accuracy with

Figure 6.9: Protocol message format for experiment 3.

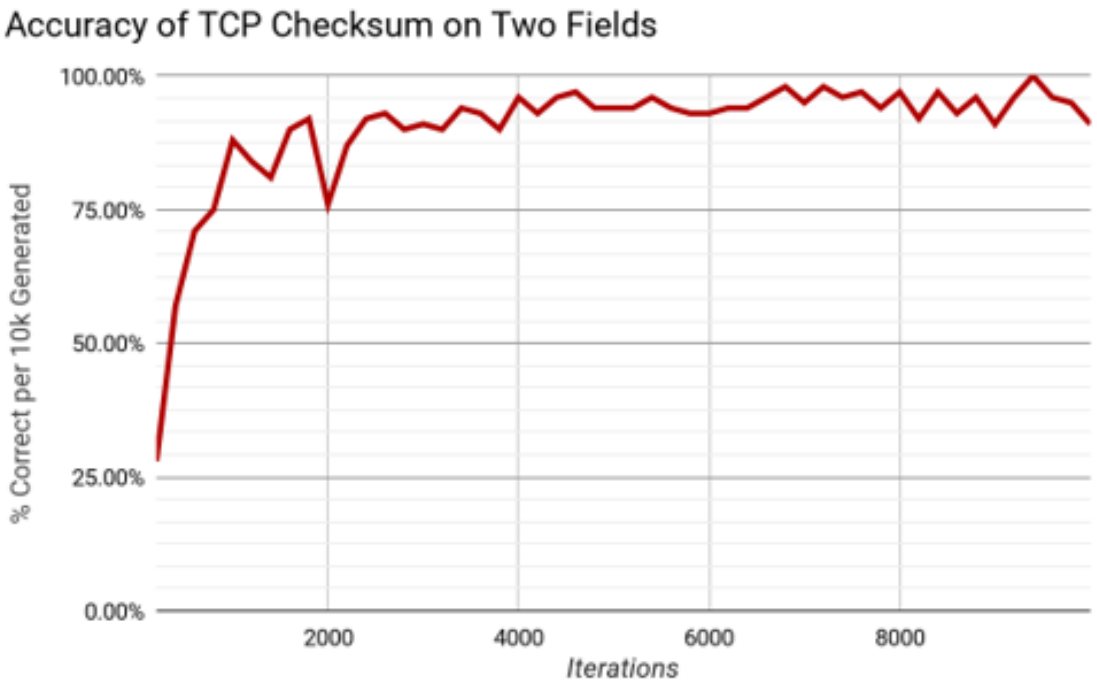

Figure 6.10: Experiment 3: An evaluation of how ProtoGANist handled TCP checksum as a value function. It was able to learn this one far quicker than XOR.

TCP checksum took far less time to converge than in the XOR experiment. This is somewhat surprising given that the TCP checksum is 16 bits versus the result of the XOR which was 8 bits long. It is possible that addition or some other characteristic of the TCP checksum may be easier to characterize as a function than XOR.

6.5.4 Experiment 4: Stress Test (Controller Area Network)

For our final experiment, we wanted to perform a stress test with a real protocol that encompassed all the different field properties — length, value, and dependency. The hope was that by testing a real protocol, we could uncover limitations that were not present in the synthetic protocol experiments we picked. In addition, it would also demonstrate how well ProtoGANist worked with real protocols.

We picked the Controller Area Network (CAN) protocol for the stress test. CAN

is a protocol used in automobiles and other vehicles. CAN has several features of interest including a variable-length field, a field whose value is dependent on that length, a CRC-15 check, and static bits at the end. These features are described in Figure 6.11.

We will individually consider each feature of interest and evaluate as before.

Length

Field 3 of the CAN protocol holds a variable length message passed by the CAN protocol and varies between 0 and 64 bits. Field 2 as seen in Figure 6.12 contains information about the length of the data in terms of the number of bytes of data. This allowed us to test a case where ProtoGANist deals with variable-length messages and a field that was a function of that length.

Since every other field is of a static length, we can determine the length of the data field by shaving off bits from the beginning and end.

We believed that length would be a more difficult concept to learn than some of the other features. The length did take some effort for ProtoGANist to learn about, but it did achieve an accuracy of approximately 80-85%. We believe part of this issue stems from the fact that not only does ProtoGANist have to learn about the contents of the fields, but it has to learn about padding in the form of 2's. The extra dimension of learning may impede the process either in terms of time to learn or accuracy.

Static Bits

Field 5 in Figure 6.14 contains the static bits '010' at the end of every message. These bits are used as an acknowledgement to previous information sent.

Our initial expectation is that ProtoGANist would not have any issues with learning static bits. Experiment 1 demonstrated that ProtoGANist did not struggle

F1	F2 = Len(F3)/8	F3	F4 = CRC15(F1,F2,F3)	F5 = 010

Figure 6.11: Protocol message format for experiment 4.

F1	F2 = Len(F3)/8	F3	F4 = CRC15(F1,F2,F3)	F5 = 010

Figure 6.12: Protocol message format for experiment 4. One of the features we evaluated was a field value dependent on variable length.

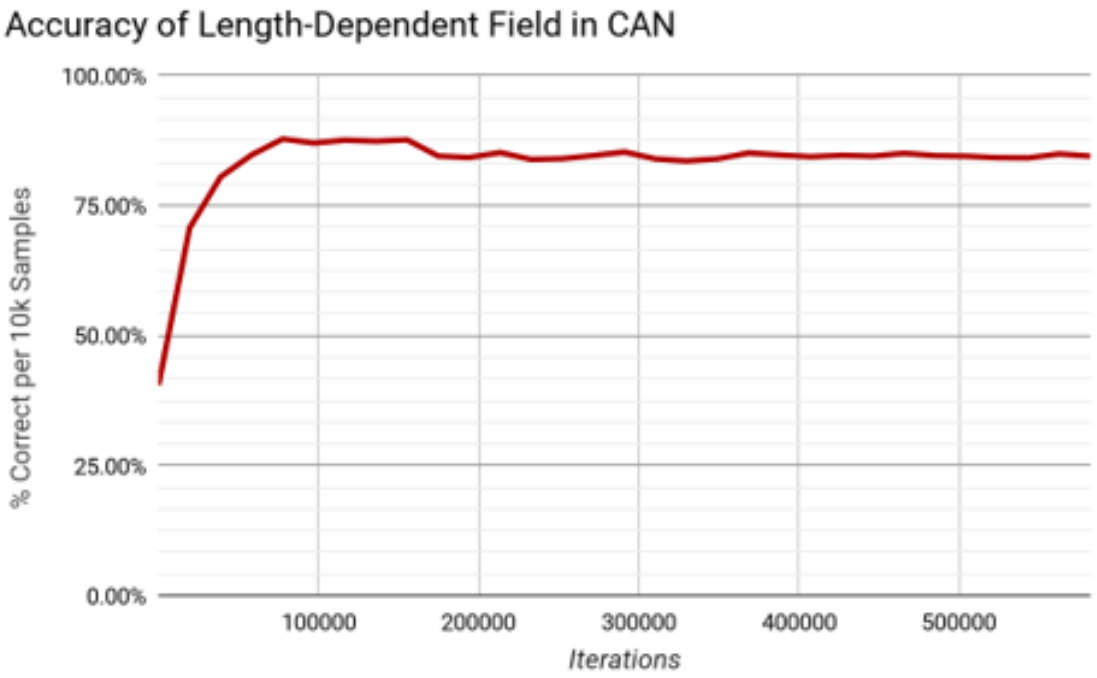

Figure 6.13: Experiment 4: An evaluation of how ProtoGANist handled a field value dependent on the length of another.

F1	F2 = Len(F3)/8	F3	F4 = CRC15(F1,F2,F3)	F5 = 010

Figure 6.14: Protocol message format for experiment 4. One of the features we evaluated was static bits at the end of the message.

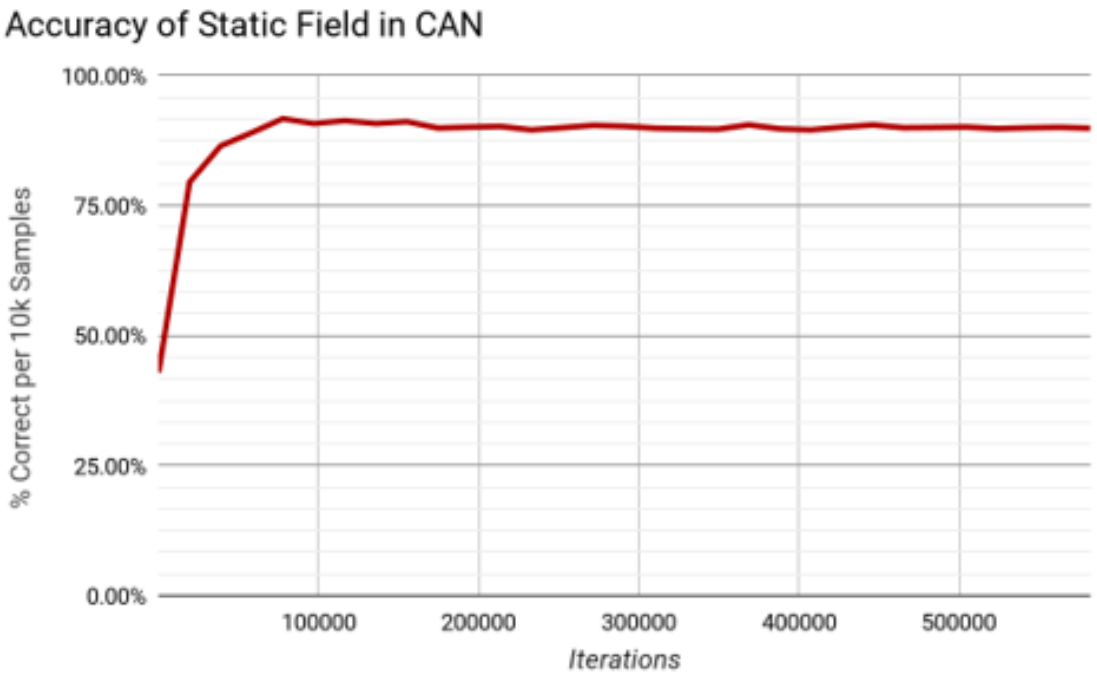

Figure 6.15: Experiment 4: An evaluation of how ProtoGANist handled static value at the end of the message.

even when presented with discrete ranges of values.

However, ProtoGANist was not able to immediately learn about the static bits at the end. Rather, it took some time to learn and even then, only achieved about 85% accuracy. The reason for this was not immediately clear but comparing Figure 6.14 and Figure 6.12 suggested that it was related to ProtoGANist learning about the variable length. The accuracy curve for the static bits in field 5 followed a similar

F1	F2 = Len(F3)/8	F3	F4 = CRC15(F1,F2,F3)	F5 = 010

Figure 6.16: Protocol message format for experiment 4. One of the features we evaluated was a CRC-15 check on the first part of the message.

pattern to that of the length accuracy curve, but with a slightly higher accuracy. While previous experiments showed that ProtoGANist can normally handle ranges of values, ProtoGANist does take a bit to learn to interpret the 2's as padding for variable lengths. Whenever the system messes up the interpretation of 2's for padding and variable lengths, it likely affects the static acknowledgement bits since they are directly before any padding.

CRC-15 Check

The CRC-15 field in Figure 6.16 was an error detection check on the first three fields. We decided that CRC-15 was an important function to evaluate as it is present in many protocols and has been considered a difficult edge case in software verification and fuzzing.

While we expected ProtoGANist to struggle with CRC-15, it performed miserably achieving a maximum of 2 out of 10,000 correct samples.

6.6 Investigating Difficulties with CRC

To better understand the issues with CRC-15, we scoped the problem down to a simpler experiment. This helps identify if CRC itself was an issue or if perhaps the many moving parts of the CAN protocol as we presented contributed to the problem.

The experiment was simplified back to a synthetic protocol as seen in Figure 6.18. Again, there are two fields that accept any value and a third which is the CRC-4 of the first two fields. Note that in this case, the third field is 4 bits long vs the first

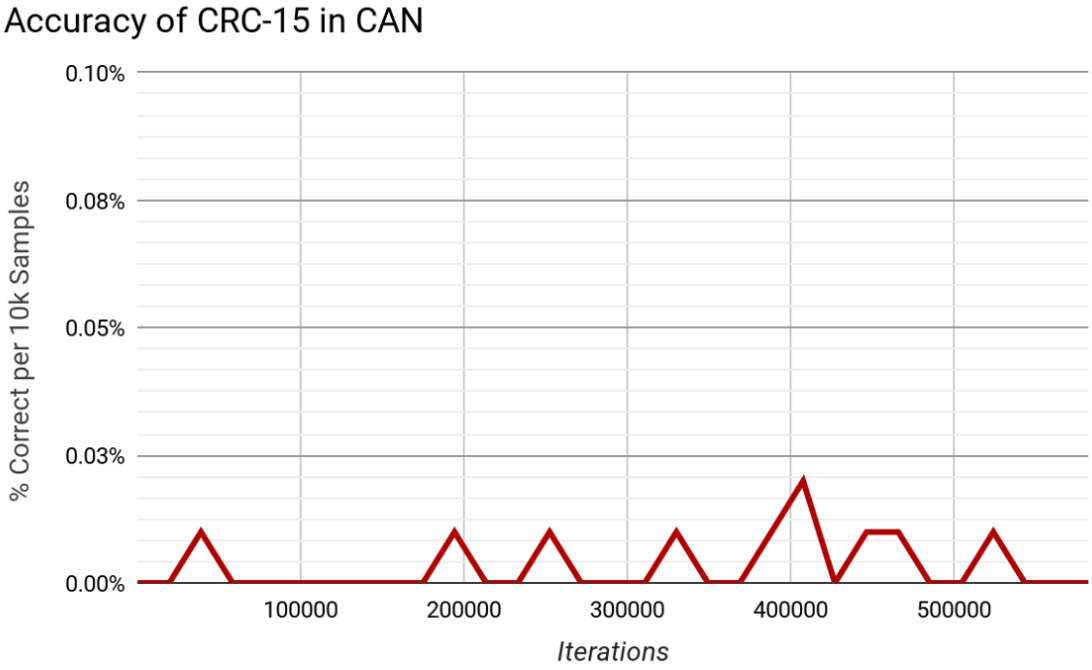

Figure 6.17: Experiment 4: An evaluation of how ProtoGANist handled CRC-15.

two fields which are each 8 bits long.

We found that ProtoGANist was, after a significant number of iterations, able to learn the CRC-4 function. While the accuracy had not yet fully converged in Figure 6.19, the system was able to get up to about 65% accuracy. This was significantly better than guessing and suggested that there were actions we could take to characterize more-complex functions.

F1	F2	F3 = CRC4(F1,F2)

Figure 6.18: Protocol message format for experiment 4. One of the features we evaluated was a CRC-15 check on the first part of the message. Note that our y-axis scale goes up to 0.10%.

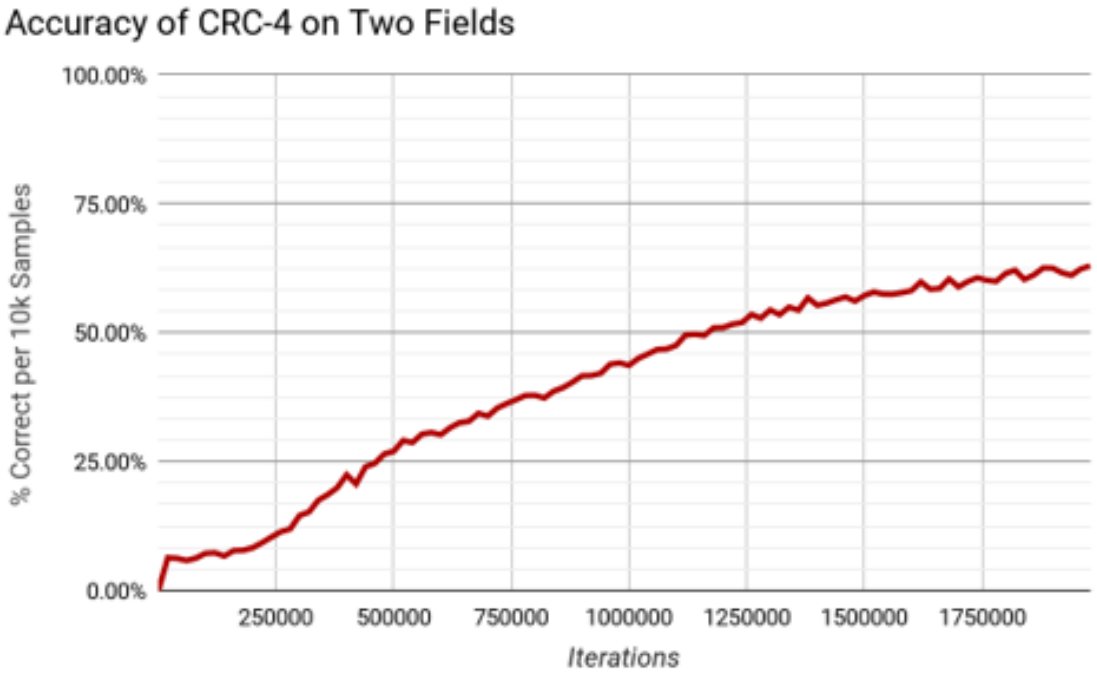

Figure 6.19: Re-examine CRC in a simpler content to understand what the limitation is and how we might be able to circumvent it.

6.6.1 Training Set Size vs CRC-4

Among the variables that influence the accuracy of the CRC-4 experiment, the size of the training set was believed to be an influential one. We wanted to investigate how reducing the training set size would impact the learning process of the experiment.

For a training set size of 7,500 samples, the accuracy in Figure 6.20 followed approximately the same curve as with 10,000 samples but ended with a slightly lower accuracy by 10-15 percentage points. As the accuracy did not converge before the completion of data collection, it is unclear if the accuracy in the end would be impacted by the smaller training set.

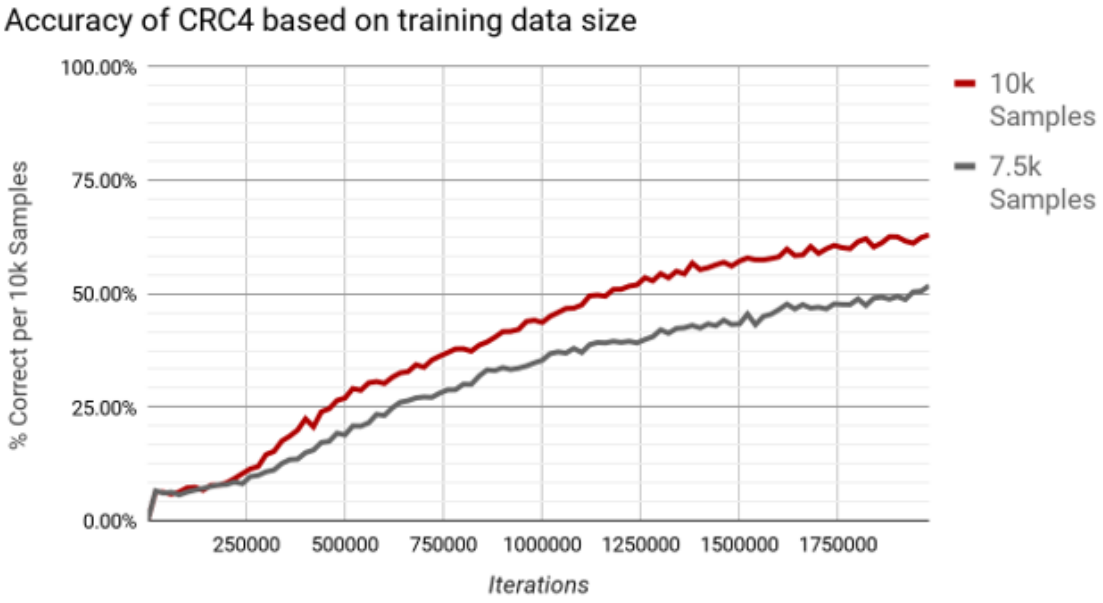

Figure 6.20: If the training set size for the CRC-4 experiment is reduced, we get approximately the same curve but with slightly lower accuracy.

6.7 Memorization vs Learning

Lastly, we wanted to understand if ProtoGANist was learning or if it was just memorizing the training set contents.

The impetus for this experiment was that not every combination of content in dependency fields was present in the training set. For example, in the TCP checksum example, the third field value is dependent on the values of the first two fields. Since the first two fields were each 8 bits long, there are $2^{16} = 65536$ different combinations of field 1 and field 2 values. However, there are only 10,000 samples in the training set. Thus, there are at least 15,536 different mappings from unseen combinations of the first two fields that ProtoGANist has not yet seen.

If ProtoGANist was only memorizing, it would struggle when dealing with some combination of the first two fields not present in the training set. It would not know

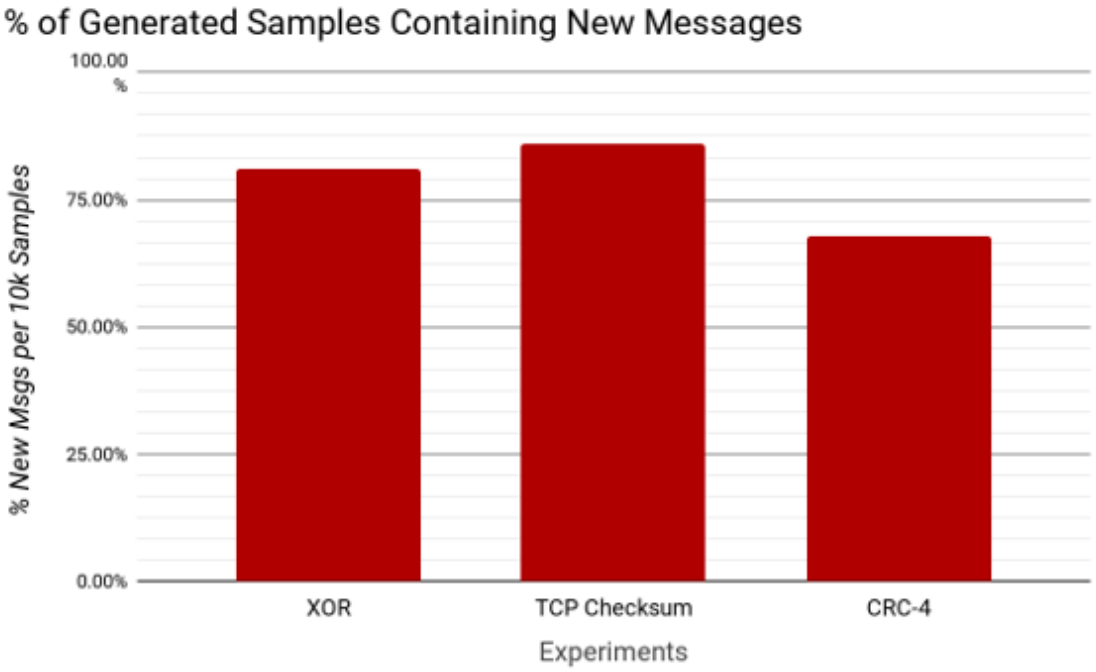

Figure 6.21: We looked at the percentage of new content generated by ProtoGANist. The system was confirmed to generate new content rather than only repeating content from the training set.

what field 3 value is mapped to it.

First, we examined the number of generated samples that had such combinations of dependency fields not present in the training set. We tested this in three experiments with the number of dependency combinations larger than the training set. The three functions we tested were XOR, TCP checksum, and CRC-4. We found, as shown in Figure 6.21, that there was a sizeable number of generated samples with dependency content not present in the training set.

Then we looked into the accuracy with respect to only those generated messages. The accuracy of these messages was found to be still relatively high. We also compared the accuracy to the expected results from guessing the field with dependencies and found that ProtoGANist did significantly better.

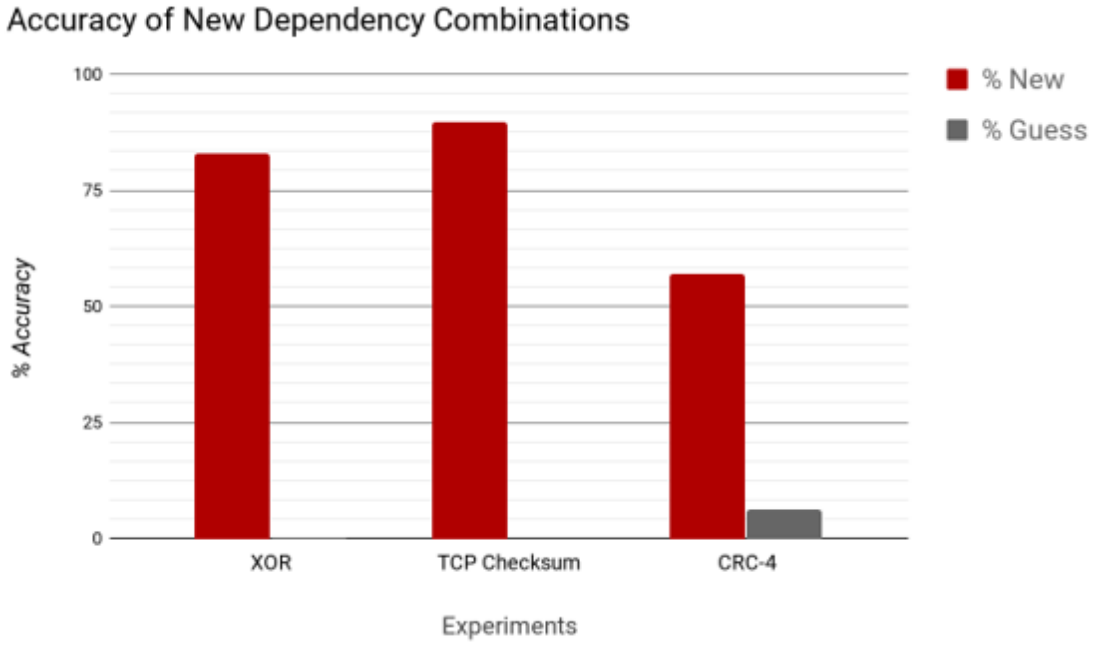

Figure 6.22: We examined the accuracy of only generated messages containing new dependency content. ProtoGANist's accuracy remained high and was significantly better than guessing.

7

Discussion and Future Work

We now present a discussion of our proposed ProtoGANist system in this chapter.

7.1 Findings

Overall, we have demonstrated that ProtoGANist is a promising approach to the black box protocol reverse engineering problem. We summarize these findings in Figure 7.1.

We demonstrated that ProtoGANist can handle most of the different properties of the protocol message format. It was able to achieve an accuracy of greater than 80% on length, ranges of values, and basic operations. ProtoGANist attained such an accuracy for TCP checksum, but struggled with CRC-4 and outright failed in learning CRC-15.

Despite this failure, ProtoGANist succeeded in all other components in the CAN experiment. This suggests that even in cases where there is a single field that ProtoGANist struggles with, it is not prevented from learning about other fields.

We also developed and ran an experiment that proved that ProtoGANist was actually learning about the functions making up the properties of the field rather

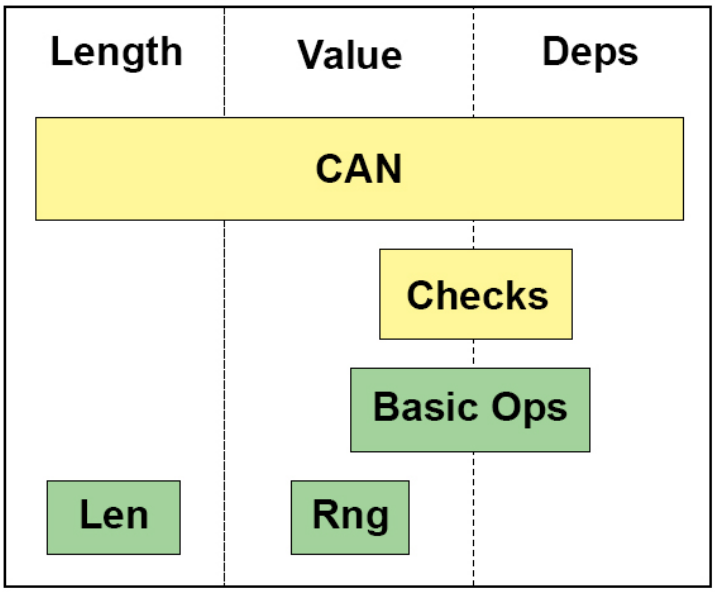

Figure 7.1: Most experiments worked well, but CRC-15 was a challenge.

than just memorizing the training set. We also found that ProtoGANist did generate many samples that were not present in the training set. This is important in the context of security verification in which we want to test new parts of the software further than the input checks the software would make. ProtoGANist is able to generate protocol-compliant messages not seen in the training set.

7.1.1 Limitations and Improvements

Our work has revealed several limitations in the ProtoGANist approach.

We demonstrated that while ProtoGANist can still learn about complex functions and dependencies with a smaller training set, the accuracy is somewhat reduced, and the model takes more time to train. This results in a trade-off in terms of the amount of data collected for the training set and the time for the system to learn about the protocol message format. Depending on the scenario in which ProtoGANist is being used, it may make more sense to trade one for the other.

While ProtoGANist was able to characterize functions that were not straightforward, it did struggle with learning complicated functions. Its accuracy with CRC-15 was the same as if it were just guessing. However, we demonstrated that in a scoped-down version of the experiment with CRC-4, it was able to attain an accuracy significantly higher than guessing. This suggests that there may be actions we can take to improve ProtoGANist's accuracy in these difficult cases. Two solutions may be to increase the training set size or run the learning portion for a longer period of time. However, this may not always be possible depending on the scenario. Users may have restrictions in the amount of data they can collect or the limit of time they can run our proposed technique. Thus, improvements may need to be restricted to tweaks within WGAN system. Parameters such as the number of layers in the neural network or the dimension of each layer could be tweaked to improve the system. These would help characterize more complexities in a data distribution.

7.2 Future Work

While we demonstrated that the ProtoGANist approach is promising, more work is needed to make it a viable alternative among other black box protocol reverse engineering tools.

First, while we developed experiments to test different properties of the protocol message format, we did not test every possible feature found in real protocols. In particular, we found that testing a real protocol was able to uncover limitations of our system. We would like to create a system to create random protocols based on our protocol message format definition from which we can automate a testing process in our system. This will allow us to better scope the limitation within our system and identify potential points of improvement.

Our focus in black box protocol reverse engineering was specifically scoped down to examine the feasibility of such an approach. We focused on the simpler problem of

learning about the protocol message formats instead of protocol grammar. However, many protocols do have some dependencies in between messages and thus it will be important to investigate the feasibility of supporting protocol grammar as well.

The current iteration of ProtoGANist is also only able to perform implicit learning of a protocol message format rather than explicit learning. This is because Generative Adversarial Networks are not able to encode the generative model of the data distribution in terms that people would easily understand. While this has been a known issue in Generative Adversarial Networks, it would be useful if the technique could be modified to additionally provide information its learned protocol message format in human-readable format. Disentangled Generative Adversarial Networks are a new approach that may provide a solution in this space. Disentangled GANs aim to capture the features responsible for the learned generative model [19][24][34] but are still being improved. If this information can be interpreted as a message format, it would provide a method for ProtoGANist to perform explicit learning.

8

Conclusion

In this work, we have presented a new approach to solving the problem of black box protocol reverse engineering. Prior state of the art struggled in presenting a way to characterize complex functions such as nonlinear functions or those with dependencies. Our proposed approach, Generative Adversarial Networks, were found to provide a promising approach to characterizing these types of functions even when several instances of such are present in a single protocol message format.

There are limitations to our system, as we found that it was difficult to learn CRC-15 error detection within the Controller Area Network protocol. Despite this, we demonstrated that ProtoGANist is able to learn other content within the same protocol message format. Furthermore, it was able to obtain an accuracy significantly better than guessing when we scoped the problem down to CRC-4. We believe there are steps that we can take to improve the accuracy in these case both outside the system and within the system itself.

Overall, we believe that ProtoGANist is a promising approach to solving the black box protocol reverse engineering. It can characterize complex functions and dependencies that make up protocols with an accuracy significantly higher than

guessing. We believe that this technique will be useful in software security verification techniques such as fuzzing that benefit from sending protocol-compliant messages to a system.

Bibliography

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[2] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," *arXiv preprint arXiv:1701.07875*, 2017.

[3] K. Backhouse, "Kernel crash caused by out-of-bounds write in apple's icmp packet-handling code (cve-2018-4407)," 2018. [Online]. Available: https://lgtm.com/blog/apple_xnu_icmp_error_CVE-2018-4407. [Accessed 2019-05-06].

[4] M. A. Beddoe, "Network protocol analysis using bioinformatics algorithms," *Toorcon*, 2004.

[5] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proceedings of the 9th ACM symposium on Information, computer and communications security.* ACM, 2014, pp. 51–62.

[6] S. Bratus, A. Hansen, and A. Shubina, "Lzfuzz: a fast compression-based fuzzer for poorly documented protocols," *Darmouth College, Hanover, NH, Tech. Rep. TR-2008*, vol. 634, 2008.

[7] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security.* ACM, 2009, pp. 621–634.

[8] C. Y. Cho, E. C. R. Shin, D. Song *et al.*, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM conference on Computer and communications security.* ACM, 2010, pp. 426–439.

[9] J. Duchêne, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche, "State of the art of network protocol reverse engineering tools," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 53–68, 2018.

[10] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[11] E. Gelenbe, G. Görbil, D. Tzovaras, S. Liebergeld, D. Garcia, M. Baltatu, and G. Lyberopoulos, "Nemesys: Enhanced network security for seamless service provisioning in the smart mobile ecosystem," in *Information Sciences and Systems 2013*. Springer, 2013, pp. 369–378.

[12] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," *arXiv preprint arXiv:1701.00160*, 2016.

[13] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[14] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on gan," *arXiv preprint arXiv:1702.05983*, 2017.

[15] R. M. Ishtiaq Roufa, H. Mustafaa, S. O. Travis Taylora, W. Xua, M. Gruteserb, W. Trappeb, and I. Seskarb, "Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study," in *19th USENIX Security Symposium, Washington DC*, 2010, pp. 11–13.

[16] E. Itkin, "Reverse rdp attack: Code execution on rdp clients," 2019. [Online]. Available: https://research.checkpoint.com/reverse-rdp-attack-code-execution-on-rdp-clients/. [Accessed 2019-05-06].

[17] A. J. Jara, A. C. Olivieri, Y. Bocchi, M. Jung, W. Kastner, and A. F. Skarmeta, "Semantic web of things: an analysis of the application semantics for the iot moving towards the iot convergence," *International Journal of Web and Grid Services*, vol. 10, no. 2-3, pp. 244–272, 2014.

[18] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *arXiv preprint arXiv:1812.04948*, 2018.

[19] H. Kazemi, S. M. Iranmanesh, and N. Nasrabadi, "Style and content disentanglement in generative adversarial networks," in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2019, pp. 848–856.

[20] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *Proceedings of the 5th ACM workshop on Security and artificial intelligence*. ACM, 2012, pp. 37–48.

[21] C. Leita, K. Mermoud, and M. Dacier, "Scriptgen: an automated script generation tool for honeyd," in *21st Annual Computer Security applications Conference (ACSAC'05)*. IEEE, 2005, pp. 12–pp.

[22] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 Fourth International Conference on Multimedia Information Networking and Security*. IEEE, 2012, pp. 152–156.

[23] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

[24] L. Ma, Q. Sun, S. Georgoulis, L. Van Gool, B. Schiele, and M. Fritz, "Disentangled person image generation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 99–108.

[25] T. C. Maxino and P. J. Koopman, "The effectiveness of checksums for embedded control networks," *IEEE Transactions on dependable and secure computing*, vol. 6, no. 1, pp. 59–72, 2009.

[26] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, p. 91, 2015.

[27] J. Narayan, S. K. Shukla, and T. C. Clancy, "A survey of automatic protocol reverse engineering tools," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 40, 2016.

[28] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[29] M. Nei, F. Tajima, and Y. Tateno, "Accuracy of estimated phylogenetic trees from molecular data," *Journal of molecular evolution*, vol. 19, no. 2, pp. 153–170, 1983.

[30] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti, "Detecting novel associations in large data sets," *science*, vol. 334, no. 6062, pp. 1518–1524, 2011.

[31] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in neural information processing systems*, 2016, pp. 2234–2242.

[32] B. D. Sija, Y.-H. Goo, K.-S. Shim, H. Hasanova, and M.-S. Kim, "A survey of automatic protocol reverse engineering approaches, methods, and tools on the inputs and outputs view," *Security and Communication Networks*, vol. 2018, 2018.

[33] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[34] L. Tran, X. Yin, and X. Liu, "Disentangled representation learning gan for pose-invariant face recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 1415–1424.

[35] A. Tridgell, "How samba was written," *Retrieved February*, vol. 26, p. 2014, 2003.

[36] M. Zalewski, "American fuzzy lop," 2014. [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed 2019-05-06].

[37] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.