PREUNN: Protocol Reverse Engineering Using Neural Networks

Valentin Kiechle[1], Matthias Börsig[2][a], Sven Nitzsche[2][b], Ingmar Baumgart[2] and Jürgen Becker[3]
[1]*AMAI GmbH - AI Experts, Karlsruhe, Germany*
[2]*FZI Research Center for Information Technology, Karlsruhe, Germany*
[3]*Institute for Information Processing Technology (ITIV), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany*
valentin.kiechle@am.ai, {boersig, nitzsche, baumgart}@fzi.de, becker@kit.edu

Keywords: protocol reverse engineering, artificial intelligence, machine learning, neural networks, fuzzing

Abstract: The ability of neural networks to universally approximate any function enables them to learn relationships between arbitrary kinds of data. This offers great potential in information security topics such as protocol reverse engineering (PRE), which has seen little usage of neural networks (NNs) so far. In this paper, we provide a novel approach for implementing PRE with solely NNs, demonstrating a simple yet effective reverse engineering of text-based protocols. This approach is modular by design and allows for the exchange of neural network models at any step with better performing models. The architectures used include a convolutional neural network (CNN), an autoencoder (AE), a generative adversarial net (GAN), a long short-term memory (LSTM), and a self-organizing map (SOM). All of these models combine for a new protocol reverse engineering approach. The results show that the widespread application layer protocols HTTP and FTP can successfully be mimicked by artificial intelligence, thereby paving the way for use cases such as fuzzing. A direct comparison to other PRE approaches is not possible due to the black-box nature of neural networks and represents the main limitation of our work. Our experiments showed that this multi-model approach yield up to 19% better message clustering, improved context distribution, and proving LSTM to be the best candidate for generating new messages with up to 67.6% valid HTTP packages and 100% valid FTP packages.

1 INTRODUCTION

In 2012 deep neural networks (DNN) (Krizhevsky et al., 2012) were introduced, showing great ability in automated feature extraction for classification, which led to a breakthrough in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)[1]. An increasing number of challenges have been tackled by deep learning since then. One exception to that trend has been protocol reverse engineering (PRE), as most of its progress was published from 2004 to 2013, missing the modern artificial intelligence (AI) boom. PRE is a specific information security task that attempts to recreate specifications about an unknown application layer protocol from artifacts of its communications. These inferred specifications can be used in a variety of other security-related applications such as fuzzing. The coverage of bugs and edge cases is generally improved with guiding knowledge about the basic message structure and internal state of the protocol.

The capacity of deep learning to mimic network protocols using reverse engineering and how to model such an approach are the core scientific questions that drove this project. This ability could enable fuzzing to better harness future advances of the AI research community. In our initial literature we selected the two renowned approaches Prospex (Comparetti et al., 2009) and Discoverer (Cui et al., 2007) to represent a large body of ideas and strategies commonly used in PRE. A novel approach is created based on these two PRE designs to serve as an orientation for solvable neural network tasks and a framework for experimentation. We then define several metrics and implement the experimentation to come to a successful conclusion. Our key contributions include the analysis of how to apply deep learning in a modular and extendable way to the problems of protocol reverse engineering, implementing a neural network-based PRE approach on text-based protocols, and showing a promising direction of future research. There are also a few minor contributions of smaller workarounds to solve problems during the experimentation, like the convolutional embedding for the LSTMs.

[a] https://orcid.org/0000-0002-6060-6026
[b] https://orcid.org/0000-0002-3327-6957
[1] http://www.image-net.org/challenges/LSVRC/

DOI: 10.5220/0010813500003120
ISBN: 978-989-758-553-1; ISSN: 2184-4356
In Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022), pages 345-356
Event website: https://icissp.scitevents.org/

2 BACKGROUND

As we attempt to combine two areas of modern research in this paper, a brief overview of both areas' theoretical backgrounds is given in this section.

2.1 Protocol Reverse Engineering (PRE)

Communication over the internet and other digital networks requires standardized protocols to ensure uniform behavior. These protocols are generally used in a stack with multiple layers, where each layer has its specific tasks and serves as the operational basis for the layers above. The widely known ISO/OSI-7-layer model works as the standard guideline for such a protocol stack (TC97, 1984).

Protocols can be divided into four types: text-based or binary and stateful or stateless. PRE aims to learn as much as possible about the specifications of an unknown protocol by analyzing the artifacts that the communication creates. This includes, but is not limited to, message formats, syntax, grammar, constants and keywords, message types, implicit or explicit state machines, and more. The taxonomy shown in Table 1 lists the two possible kinds of artifacts: network traffic messages or system calls inside the binary of an application that communicates using the target protocol. The two most commonly inferred properties are the general message format/syntax and the underlying state automaton of a stateful protocol. A successfully reverse-engineered protocol allows for further analysis of the communication like deep package inspection (Brook, 2018) and fuzzing (Besic, 2019). Both can work more effectively if they have the detailed specification of the protocol.

Two prominent PRE examples are Prospex (Comparetti et al., 2009) and Discoverer (Cui et al., 2007). They were published in the middle of the major period for PRE research between 2004 and 2013 (Duchene et al., 2018; Narayan et al., 2015). From our initial literature research, we judge both to be good representations of their respective PRE classes, as mentioned in Table 1.

Protocol Specification Extraction (Prospex) Prospex is a two-part PRE approach basing its analysis on both network messages and execution traces of a binary at runtime to infer the message format (Wondracek et al., 2008; Caballero et al., 2007) and state automaton in a second part (Comparetti et al., 2009). The highly distinctive features selected for the clustering step are among the core reasons for the success. They include file system reactions and memory analysis by tainting bytes. Therefore, the similarity between messages is not only computed by their format, but also by the reaction and response they create. Next, an acceptor automaton was designed to identify valid sequences of messages. This automaton was later reduced using the exbar algorithm (Lang, 1999). The final tool was extended to be compatible with the fuzzing tool Peach Fuzzer[2]. As a direct result, fuzzing can be improved on stateful protocols. We learn from Prospex how a separation of tasks such as the extraction of features, clustering using an off the shelf method and finally reverse engineering are key in PRE.

Discoverer Discoverer identifies small clusters of tokens within a message, which are then merged and refined recursively. For this, it relies on the assumption that protocols use common delimiter symbols to separate parts of their message format, such as commas, whitespaces, or line breaks. The interdependence of various fields (e.g., for addresses/variables) in the message format and their data type (text or binary) is then learned by heuristics. We can see the importance of identifying key features and their influence on clustering in the Discoverer approach.

2.2 Neural Network Architectures

All the models we use in this paper are types of neural networks. They consist of layers of interconnected neurons, where each connection has a value to adjust the importance of the connection for the next layer. These weights are adjusted to minimize an error function during training, but remain fixed for testing and thereafter. This concept allows for complicated mapping of input data distributions to output results according to the universal approximation theorem (Hornik et al., 1989). As such, neural network architectures are suitable tools for different data operations such as classification, regression, clustering, and more. In the following paragraphs, we describe the different architectures used in the experimentation of Section 4, what problems they are good at solving and why we considered them for PRE.

Convolutional Neural Network (CNN) CNNs became popular for their performance in image recognition tasks (Krizhevsky et al., 2012). The layers in this architecture have a particular property: they are used in a sliding window method over the input data. Their weights for each of the sliding window steps remain the same and can therefore be used to detect small, fixed patterns. Many such "kernels" exist in

[2]http://peachfuzzer.com

Table 1: Taxonomy for classifying PRE approaches by requirements (columns) and by results (rows)

PRE taxonomy requirements and results	Only network messages	Messages and executable binary at runtime
Inferring message format/grammar	Discoverer (Cui et al., 2007)	Wondracek (Wondracek et al., 2008)
Inferring state automaton	**PREUNN**	Prospex (Comparetti et al., 2009)

parallel to each other in separate "channels" of one layer. This structure allows individual kernels to approximate highly descriptive feature filters, thereby removing the traditional step of manual feature extraction from the classification task. This architecture is beneficial in any task involving images/pixel maps with individual features occupying many adjacent pixels. However, in theory, this concept can be applied to any form of input data with patterns.

Autoencoder (AE) Autoencoders are neural networks designed to achieve dimensionality reduction for a given input while retaining as much information as possible. In training, the loss function demands that the output equals the input as closely as possible. The architecture itself has a bottleneck in a middle layer to force the network to compress data but keep reconstructable information (Hinton and Salakhutdinov, 2006). The middle layer's size has to balance the compressing size and keep relevant information in some unknown encoding. Naturally, this divides the neural network into two components, namely the encoder and the decoder part. After the AE has been trained, the decoder element is removed so that any input data will be returned in its encoded form only.

Generative Adversarial Network (GAN) GANs were developed to create a generative model for learned data distribution. The peculiarity of this architecture comes from the use of two competing networks (Goodfellow et al., 2014). The first, known as generator, uses random noise as input and tries to create an image similar to those found in the training dataset. The second network, known as the discriminator, is given either a true or a fake image randomly and has to learn the distinction between them by merely classifying true from fake. The error correction for the generator is based on the classification result by the discriminator. This combination of two networks causes a setup of competitive learning where each NN tries to outdo the opposing one.

Long Short-Term Memory (LSTM) Recurrent neural networks (RNNs) describe a type of architecture, where some part of the internal hidden or input state is recursively put into the network again for the next time step, thereby allowing the network to find time dependencies inside the data. LSTMs are a kind of RNNs with a strong ability to find sequential dependencies over time while avoiding some common problems with recurrent error correction (Hochreiter and Schmidhuber, 1997). For working with text data (sequential letters), it is common to use a dictionary or alphabet along with a suitable embedding to condense the information in a medium-sized vector for each word or letter. The text is transformed into a matrix of $\text{length}_{\text{text}} \times \text{length}_{\text{embedding}}$.

Self-Organizing Map (SOM) SOMs are architectures with one-dimensional or two-dimensional output maps of neurons that contain the topology of the data (Kohonen, 1982). This means that data with similar properties and distribution of features will be found in the same general area of the output map. Thereby, clustering is implied. SOMs give each output neuron a score related to the data. The winning node can be returned for indexing.

3 MAIN APPROACH

This section presents a novel way of looking at the underlying task structure of reverse engineering a protocol. It is divided into multiple steps that are handled mostly sequentially. The approach is designed to fit the capabilities of various neural network architectures and provides modularity. This allows for the replacement of any model in the system by a better performing one and thereby enabling novel AI designs to be directly inserted into the process.

3.1 Data Gathering

To train a neural network, a representative dataset is required. We chose a set of text-based application layer protocol artifacts as a basis to allow for an easier result evaluation as we do not have a direct comparison with other PRE approaches. The chosen protocols are HTTP v1.1 and FTP because they are commonly used, abundantly available and lack encryption. We use several sources of datasets to cover a broader mix of implementations and message type distribution.

3.2 Feature Extraction

In this first part, we want to extract highly distinguishable features. Both in Prospex and Discoverer, the feature selection was an essential part of the work, but the features were chosen by the researchers. We intend to automate this process with neural networks. Of particular interest are keywords, punctuation, syntactic characters, and other patterns that distinguish between different messages. Pseudo-random strings such as tags and cookies are avoided, as they are generally unrelated to the protocol specifications.

3.3 Feature Reverse Engineering

In traditional protocol reverse engineering, the analysis infers rules, lists of variables or constants, and grammar from the communication artifacts. With a neural network, the learned knowledge is intrinsically non-representative, meaning it is challenging to interpret by humans. We use a generative evaluation approach to judge the quality of the features learned (and recreated) by the respective architectures. Such a method will create new samples from the training distribution and provide insights into what the neural networks learned.

3.4 Clustering

Protocol messages can usually be grouped into types whether or not the protocol explicitly specified these groups. We can cluster these messages using information like sequential order, functionality and general format. The clustering would imply various types of messages, and we consider this task to be well-suited for neural networks (Bação et al., 2005).

3.5 State Recognition

A typical communication session usually involves multiple messages being sent or received. In stateful protocols, particular sequences of messages achieve a more complex state between the communication parties. These sequential patterns imply a representative state automaton for the inner state of the protocol. We think that a neural network with the capability of understanding sequential dependencies should be able to learn the order of different message types and their likelihood. This interpretation allows the usage of time series prediction to imply the next state of the protocol.

3.6 Sequence Generation

As a last step, we combine all trained models into one generative PREUNN AI. Context information such as cluster index and sequential dependencies can be included in the feature reverse engineering to achieve more accurate results. Ideally, this AI is capable of producing valid messages, which are not part of the training set but have comparable statistical properties.

4 EXPERIMENTS

This section lists all implementations for the main approach and the experiments, that were used to test various neural network architectures. The hyperparameters of all neural networks are set to well-performing numbers after some semi-extensive manual testing. The scope of the project did not include major optimizations as the hardware was unable to handle large search spaces for automated hyperparameter tuning. Our code is available on GitHub[3]. All experiments were written in Python 3 using an object-oriented programming style to ensure easy modification and extension of the experiments to new protocols. We selected PyTorch[4] as the deep learning framework, and all experiments were conducted on an NVIDIA GTX 970.

4.1 Data Preprocessing

We used two datasets in our tests. The first one consists of the combination of multiple HTTP sources (Garfinkel, 2008; Shiravi et al., 2012; Goo et al., 2019; Sharafaldin et al., 2019) that were selected to cover different implementations and scenarios. The second dataset consists of FTP messages (Pang and Paxson, 2003). Before any experiments can be conducted, we examined the data for outliers and irregularities. Data engineering usually takes up a significant amount of time in any machine learning project; however, with network traces in pcap files, we were able to shorten this process. The network analysis tool Wireshark[5] provides a widely used parser for protocol package analysis. We filtered for valid packages of HTTP and FTP respectively and discarded the rest.

The length of application layer datagrams is limited by the underlying TCP protocol. An analysis of the new pcap file showed that lengthy messages only occur at image transfers (HTTP) or custom messages

[3]https://github.com/PREUNN/preunn

[4]https://pytorch.org/

[5]https://www.wireshark.org/

(FTP) and can subsequently be cut without significant loss of information. We chose the length limit of 1024 bytes for all packages to uniform the neural network inputs. Only 0.33% of HTTP and 0.18% of FTP data that we use exceed this length limit.

As a further step for the HTTP experiments, all content after the message header was removed, to avoid XML and other non-HTTP data. We achieved this by splitting each statement at every "\r\n\r\n" occurrence and only using the first element of that split. This double line break is the HTTP sign for only payload following, and thus represents a convenient way to clean the data. No filter was applied for the FTP data.

Dataset bias is a common pitfall when developing machine learning solutions. It describes an uneven distribution of classes within the data, which causes suboptimal feature learning in the neural networks. The protocols themselves do not specify classes of messages directly; however, the manually created classification for both protocols are our baselines to orient the class balance on. It is desirable that the neural networks still learn which types are more common than others, but the imbalance in our dataset is overwhelmingly in favor of two or three common message types. To balance the number of messages per class, we came up with this improvised formula:

$$\text{No}_{\text{samples per class}} = \sqrt{\text{No}_{\text{occurrences}}} * 100 \quad (1)$$

and visualized the distribution to see the effect in figure 1 and 2.

The packages for each class were selected randomly until the limit was reached. This includes multiplications of rare messages to get a significant sample size in every class. Tests without this dataset balancing have shown strong signs of overfitting in most experiments. We did not simply set all classes to the same limit, as the notion of common and uncommon should not be lost.

4.2 Feature Extraction

Neural networks are known to be highly flexible self-learning feature extractors across various tasks. When we try to interpret each message character as a pixel integer value and thus the entire message as a one-dimensional image, we can apply solutions for image-based tasks. We selected two types of architectures for feature extraction: an autoencoder and a convolutional neural network. The feature mappings learned by the models in these experiments later serve as input to the clustering. By the black-box nature of neural networks, we cannot directly measure the quality of the feature mappings and will instead use the later results from clustering to evaluate both architecture. We only defined auxiliary metrics for each experiment.

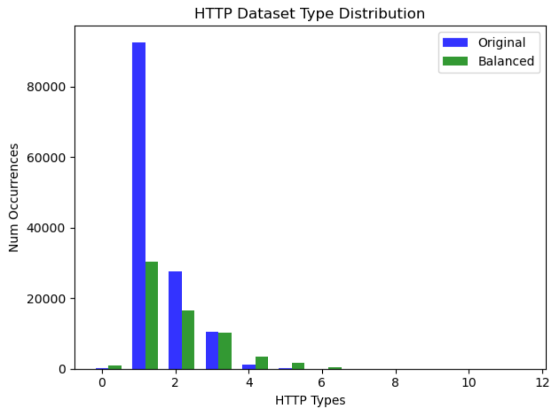

Figure 1: HTTP dataset distributions in original (blue) and balanced (green). The overwhelming bias in favor of GET messages has been mitigated while preserving the notion of this type being the most common.

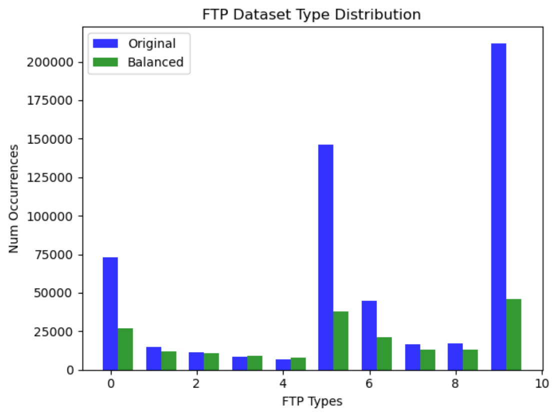

Figure 2: FTP dataset distributions in original (blue) and balanced (green). With three dominant types and several underrepresented types, the rebalancing smoothed the distribution significantly while preserving tendencies.

Autoencoder Autoencoders only work on data of fixed length and learn a compact representation of the data. The network messages we use as data vary in length, but generally do not exceed 1024 bytes. To unify them, we pad shorter messages with zeros to fill up the length of 1024. Small-scale tests indicate, that padding/capping to this length does not impact the performance of any model significantly. As the expected output for the AE architecture is equal to the input, we used the Hamming distance as a guiding measure for success during the experimentation. We train an autoencoder the following layer sizes: $1024 \rightarrow 256 \rightarrow 128 \rightarrow 256 \rightarrow 1024$. We use softplus activations, the Adam optimizer is set to a learning rate = 0.0005, the loss function is Mean Square Error (MSE), and the batch size = 128 for 10 epochs. We use the resulting 128 neurons wide output of the encoder part as the feature encoding for a message. The data for this experiment was interpreted as pixels and subsequently has a continuous nature in the interval $[0, 255]$. The distance achieved for data samples of length 1024 bytes in training is 254.44 on average for HTTP and 41.28 for FTP. The very high Hamming distance results from the continuous nature of the byte interpretations, even the padding symbols (0) couldn't be reconstructed entirely. We observe that padding symbols often miss the same ASCII symbol by 1 or 2 values if we interpret the ASCII table as a scale from 0 to 255. In an image, such small coloration mistakes would hardly be noticeable. For the alphabet as a continuous scale, the results often look wrong. However, an autoencoder's primary purpose is to reduce the feature dimensionality for minimal loss of information and the learning of patterns. In this case, we managed to reduce the dimension from 1024 down to 128 while retaining satisfying reconstruction properties during the experimentation. The lower average Hamming distance for FTP can be explained by their much shorter average message length.

Convolutional Neural Network CNNs are commonly used for supervised image classification. Our training data does not contain any labels that can be used for supervised learning. We came up with our own unsupervised learning technique using data augmentation. Messages are replicated and modified into several known classes of augmentation types to extract information about the syntactic context. Our idea is, that the ability of a CNN to differentiate between these augmentation classes will teach it to become fine-tuned to common patterns in the syntax. HTTP or FTP statements are again padded/capped to a fixed length of 1024, then divided into segments of various lengths (1024, 32, 16, 8, 4), which are then put into random order within the same statement. This approach results in 5 classes (unchanged and scrambled into blocks of length 32, 16, 8, and 4, respectively). A simplified example is illustrated in figure 3.

We chose an architecture that uses blocks of 1D convolution, 1D batch normalization, softplus activation, and 20%-dropout in a total of 5 layers following the channel sizes as follows: $1 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8$. The classification task was performed by two fully connected layers, which were removed after the training to get a feature map of $8 * 30 = 240$ neurons, which is larger than that of the autoencoder. We used a visualization of the learned features as an additional analytical method to evaluate the performance. A va-

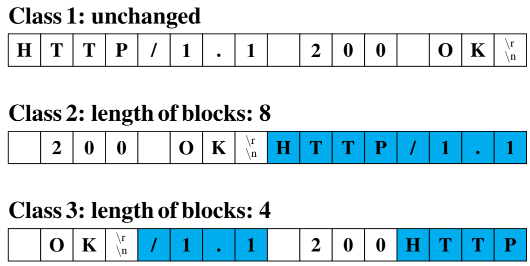

Figure 3: An HTTP statement scrambled into 3 classes (unchanged and blocks of length 8 and 4). The color indicates the block-size and the order of the blocks is randomized.

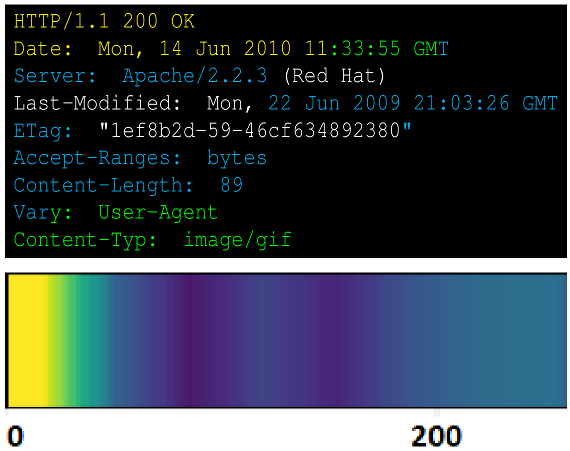

Figure 4: Example of HTTP statement classification score visualized by Grad-CAM. The padding has been cut off. The semantically significant and consistent parts are highlighted, while pseudo randomness in the ETag is ignored. The text highlighting in this figure is an approximation.

riety of methods for model explanation have been proposed for image classification tasks. We use the visualization of pixel importance known as Grad-CAM (Selvaraju et al., 2017) as our quality measurement for syntactical features found by the model. Our experiments showed mixed results. Regarding HTTP, we see that various parts of a protocol message are highlighted differently, see figure 4. The highlighted parts often match syntactically relevant pieces, while pseudo-random strings are ignored. This is precisely the kind of syntactical feature extraction that is desired for this experiment.

For FTP, the results are less visible in Grad-CAM, as the average message length is much smaller. The overall convergence of the experiment was also much flatter than that of the HTTP version.

4.3 Feature Reverse Engineering

A common result of protocol reverse engineering is the representation of the target protocol in the form of rules or clusters. Neural networks do not allow us to directly visualize their internal representation of the features they extracted, however. To have a tangible result outside clustering and sequence recognition, we want to be able to generate new messages as proof of the correctness of the feature learning ability. We achieve this by using generative neural network models and examine their outputs. There are two possible choices of how a text message can be interpreted: an image-like byte interpretation (as in the feature extraction before) or a sequential interpretation as a sequence of ASCII alphabet symbols. We investigated both alternatives, using a default GAN architecture for image-like byte interpretation, roughly following the suggestions given by the original authors (Goodfellow et al., 2014; Radford et al., 2015), and an LSTM model for interpretation as a sequence of ASCII symbols, with a modified embedding for accounting the randomness in some parts of protocol messages (cookies, addresses, etc.).

Generative Adversarial Net The two networks a GAN consists of, the generator and the discriminator, are trained in parallel. The generator uses four 1D transposed convolution layers with (kernel size, stride)-parameter tuples of (2, 2), (4, 4), (8, 8), and (16, 16) in ascending order. The first three layers are followed by a 1D batch normalization and ReLU activation each, while the last one ends with a sigmoid function. The number of channels is as follows: $1024 \rightarrow 1024 \rightarrow 128 \rightarrow 32 \rightarrow 1$. The discriminator uses four 1D convolution layers with 1D batch normalization (except for the first layer) and LeakyReLU with a 0.2 slope and 20%-dropout. The network is capped with a 360-neuron fully connected layer. Channels are as follows: $1 \rightarrow 10 \rightarrow 20 \rightarrow 60 \rightarrow 90 \rightarrow 1$. We use a formula to keep either network from overtaking the other in training: If one network error went over a threshold *OR* the other network under a particular border, then the overperforming network is removed from training until the other catches up. Both use an Adam optimizer with a learning rate = 0.0005 and betas = (0.5, 0.99). As a loss function, we used Binary Cross-Entropy (BCE).

The GAN model's training did not indicate a significant convergence towards a stable state. As the pixel interpretation's inaccuracy is limiting the GAN from the start, combined with this architecture's generally unstable nature, we are not surprised by this disappointing result. Creating text from a continuous data interpretation only sounded promising to us when we considered the static structure used by a protocol. However, not even the padding symbols have successfully been replicated with any significant accuracy (about ± 3 ASCII values) by the GAN model. Further, experiments using this architecture are omitted.

Long Short-Term Memory LSTMs are advanced recurrent networks, trained on sequences of fixed lengths. To create such conditions, padding is out of the question, as it would insert undesirable sequential dependencies. We instead use a script, which concatenates statements to achieve more than four times the required length, then choosing a substring of correct length starting from a random position. This script is only used for the training to teach the sequential dependencies from letter to letter. Before and after each statement, a unique symbol for start-of-package (SOP) and end-of-package (EOP) is inserted so that the network can learn to distinguish between different messages. The data is represented as a one-hot-encoding of the ASCII alphabet. The architecture uses an embedding layer followed by a convolution layer (kernel size=4, stride=4) for embedding adjustment. This trick is introduced here as convolutional embedding, designed to balance between character-based and word-based embedding in data formats with a lot of random noise on character level. The hidden size of the embedding is flipped with the feature-length dimension of the tensor; the convolution layer interprets the hidden width as channels and the feature-length dimensions as image dimensions. Only the batch size remains the same. This dimension transposition is reversed after the convolution, resulting in an embedded tensor with a quarter of the length. This is given as input to a single-layered LSTM. The output goes through the reversed procedure of the convolution embedding in a 1D transposed convolution and a fully connected layer with the same hyperparameters and dimension transpositions. Figure 5 shows an overview of how this embedding works. The way to interpret this kind of embedding is a learnable, weighted 4-gram of characters, where the whole architecture only has to learn the next character (from 0–1023 to 1–1024 by index). This local context inside the 4-gram and the long-term dependencies, which have been shortened by a factor of 4, are easier to learn for the model and more stable to sample. For training, an Adam optimizer with a learning rate = 0.005 and standard beta is used. Negative Log-Likelihood (NLL) is used as the loss function since the error is measured on character level. This, of course, requires the input data mentioned at the top to consist of message strings with a length that are multiples of 4.

This sequence-based attempt at recreating HTTP and FTP statements shows good results. The LSTM architecture converges towards a minimal loss after

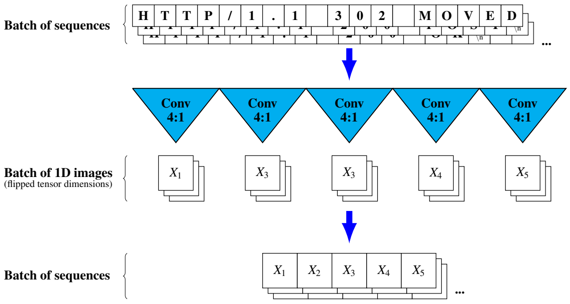

Figure 5: Simplified illustration of the convolutional embedding. It allows for a weighted length reduction of text and increases the size of the alphabet.

less than one epoch, indicating excellent structural learning and repeating patterns. We can explain this by the nature of a text-based protocol such as HTTP and FTP, which use keywords, a fixed grammar, and a consistent alphabet. When sampling the LSTM (letter by letter), a string with a valid message is produced and can be parsed using the SOP and EOP symbols. We wrap the resulting statements with valid but random headers of TCP and IP to become complete network packages in a pcap file. The network analysis tool Wireshark[6] showed 67.6% of the HTTP messages as valid, with the remaining ones classified as TCP with a random payload. For FTP, a 100% quota was reached, however FTP messages can be rather simple to be valid. A few messages are repeated often in the training data, which also appear in the output of this experiment. Some generated examples can be seen in figure 6. We see these results as a sound basis for further experimentation.

4.4 Clustering

For clustering, the feature extraction results are relevant to encode the data samples to a smaller format. We initialize three SOMs and compare their results: a baseline model with capped and padded messages to a fixed length of 1024, a second SOM model using the AE encoding, and a third model using the CNN feature map. These three models differ in their input size but have an identical output map dimension of 16×1 for HTTP and 64×1 for FTP. The different numbers for the output dimensions for both protocols are based on experimentation and roughly match the variety of different message types for each protocol. This can be adjusted by parameter for any new protocol or optimization purposes. Training is performed with a learning rate = 0.005 and sigma = 1.5 for HTTP and sigma = 3 for FTP. For details on the parameters, please see the "MiniSom" library documentation[7].

[6]https://www.wireshark.org/

[7]https://github.com/JustGlowing/minisom

```
HTTP/1.1 200 OK
Date:  Mon, 14 Jun 2010 13:20:25 GMT
Server:  Apache
Last-Modified:  Mon, 21 Jun 2010 14:18:09 GMT
ETag:  "2de9573-2b-486717fb77ac0"
Accepted-Ranges:  bytes
Content-Length:  43
Connection:  close
Content-Type:  text/html; charset=iso-8859-1

HTTP/1.1 200 OK
Date:  Mon, 14 Jun 2010 13:20:25 GMT
Server:  Apache
Content-Length:  43
Connection:  close
Content-Type:  text/html; charset=iso-8859-1
```

Figure 6: This figure shows two examples of HTTP statements generated by an LSTM. For comparison, figure 4 shows an actual HTTP message from the dataset. One can see that the general structure is similar, but the variable contents have been changed, and some optional information was added or altered. For fuzzing purposes, this is the desired behavior.

Table 2: The FTP types have been manually grouped into clusters of keywords/codes with similar meaning or purpose

0	misc
1	ACCT, ADAT, AUTH, CONF, ENC, MIC, PASS, PBSZ, PROT, QUIT, USER
2	230, 331, 332, 530, 532
3	PASV, EPSV, LPSV
4	227, 228, 229
5	ABOR, EPRT, LPRT, MODE, PORT, REST, RETR, TYPE, XSEM, XSEN
6	125, 150, 221, 225, 226, 421, 425, 426
7	ALLO, APPE, CDUP, CWD, DELE, LIST, MKD, MDTM, PWD, RMD, RNFR, RNTO, STOR, STRU, SYST, XCUP, XMKD, XPWD, XRMD
8	212, 213, 215, 250, 257, 350, 532
9	120, 200, 202, 211, 214, 220, 450, 451, 452, 500, 501, 502, 503, 504, 550, 551, 552, 553, 554, 555

For this evaluation to work, we have to know the true classes (synonym for types) for both protocols in advance. This is not straightforward, as neither protocol specifies explicit types/groups of messages apart from requests and responses. For HTTP, the responses especially have a wide range of meanings, which we grouped by their respective code's first digit for this analysis, as they represent a basic meaning of the code without going into too much detail. This means that all messages from 200–299, all 300–399, and all 400–499 messages are considered to be of the same type, respectively. Along with all the valid keywords that an HTTP statement can start with (GET, POST, HEAD, DELETE, OPTIONS, PUT, TRACE, CONNECT), this gives us a total of 11 clusters with an additional miscellaneous one (MISC). For FTP, we manually defined groupings of keywords and keycodes with similar meaning or purpose, as shown in Table 2. This was a manual process, and we do not claim to have achieved perfection with this grouping.

For the experimentation on clustering, we use the first multi-model approach. We compare three different configurations of self-organizing maps in terms

Table 3: Results of the clustering experiments for comparison. One can see an improvement over the baseline model when using Autoencoder.

(a) HTTP clustering

Architecture	Accuracy (dominant)	Avg. Confidence (dominant)
Baseline SOM	75% (75%)	58.34% (58.34%)
CNN + SOM	68.75% (68.75%)	53.61% (53.61%)
AE + SOM	**87.5% (87.5%)**	**69.24% (69.24%)**

(b) FTP clustering

Architecture	Accuracy (dominant)	Avg. Confidence (dominant)
Baseline SOM	60.94% (72.22%)	51.8% (61.4%)
CNN + SOM	29.69% (29.69%)	18.19% (18.19%)
AE + SOM	**67.19% (86%)**	**56.11% (71.82%)**

of their performance on our clustering metrics against each other. Firstly, the 128 neuron-wide SOM will use the encoding from the autoencoder model. Secondly, the 240 neuron-wide SOM will use the feature maps of the CNN architecture. Lastly, as a baseline, we use a raw SOM with 1024 neuron inputs for whole statements.

We use two metrics to judge the effectiveness of the clustering for each setup. The first is accuracy: how many detected clusters match a message type of the protocol with more than 50% confidence, which will be referred to as a "dominant" cluster. Here, confidence is defined as the relative share of a type among all messages assigned to one cluster. If there are 120 messages of type A and 80 messages of type B, all put into one cluster, then the confidence of this cluster to represent type A will be $\frac{120}{120+80} = 60\%$. The second metric is the average confidence among all clusters. Both metrics are also reported for dominant types only ($> 50\%$ for one type), to remove empty and small clusters from the average.

Table 3 shows the results of the experiments. Some setups using dedicated feature extractors can outperform the baseline significantly. The autoencoder appears to be better suited for this task, perhaps because the CNN architecture required a workaround with data augmentation to even allow for training. For further experiments that require cluster indexes, we combine the AE and SOM architectures as a pipeline to replace messages with their cluster index.

4.5 State Recognition

To recognize deeper states in a protocol, we use the correlation of message sequences as they appear chronologically in the dataset. For this, we replaced all messages with their cluster assigned by the clustering model. A simple LSTM with fitting dimensions

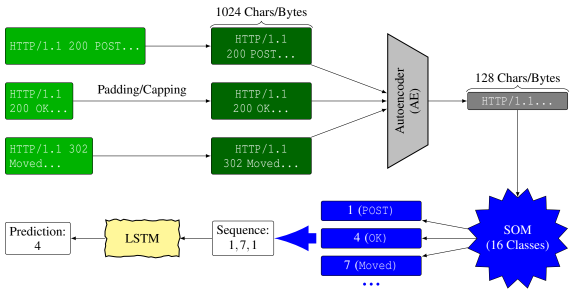

Figure 7: This figure shows examples of HTTP statements as they are processed for state recognition.

to the SOM output is sufficient to indicate highly correlated sequences by presenting the network's confidence for what the following message type could be. The central idea of SOM learning is arranging similar types of messages in proximity to one another, thereby allowing the use of the MSE loss function to approximate the correct message type in the LSTM output. The use of Cross-Entropy (CRE) loss has shown no convergence. For training, we use an Adam optimizer with a learning rate = 0.005 and betas = (0.3, 0.9).

The setup is shown in figure 7. A simple time-series prediction using a recurrent architecture shows promising results in FTP, where actual states are implied in the protocol. With the 64 indicated possible clusters, the LSTM matched 42% of the predicted message types to the following actual message. This number may seem low at first glance; however, for an in-depth state fuzzing approach using many attempts, we view this as a significant improvement over random choice.

This experiment was also conducted for the HTTP protocol, and the results are less impressive. Out of the 16 implied clusters, 72% were correctly predicted. This number may seem high at first glance but relies on predicting the average cluster number aligned with the dataset bias. In other words, the prediction simply states two or three alternating types for the GET message, which drives the prediction accuracy higher than any different prediction pattern. The dataset could not be balanced for classes in this experiment, as we wanted to avoid changing any sequential dependencies by randomly picking messages. As a result, the input data for HTTP has a significant bias towards GET message types, as the original data have, which ultimately explains this behavior of the LSTM.

The results in this section showcase the potential hazards of working with machine learning techniques, mixing their embeddings and approaches, as well as the interpretation of a predefined metric. Even though the results were a success in regard to the stateful protocol, which was our original aim for this experiment, applying the same efforts to a stateless protocol shows the weakness and dangerous pitfalls when interpreting the metric. For both protocols, the metric is misleading at face value. Only an analytic look at actual prediction results did correct the error.

4.6 Sequence Generation

To fully mimic all aspects of the behavior of a protocol, we must be able to create syntactically correct messages with real-world distribution. We use an LSTM model similar to the one for feature reverse engineering with added context. For any message, instead of using a generic SOP and EOP symbol as markers for beginning and end, we introduce new special symbols, which are individualized by cluster type. This results in 2×16 extra symbols for HTTP and 2×64 additional symbols for FTP to be added to the ASCII alphabet in their respective setups and subsequently to the one-hot-encoding. The sequences put into the LSTM, which uses a hidden size of 100 neurons in two layers to account for the extra complexity, have the beginnings and endings of each message indicated by a cluster special SOP or EOP symbol. The rest of the model architecture is identical to the one used for feature reverse engineering, including convolutional embedding. Again, for training, we use the Adam optimizer with a learning rate = 0.005 and default betas with an NLL loss function.

We use the same metric as for the feature reverse engineering LSTM. We parsed the generated sequence into HTTP/FTP statements wrapped up in some valid low-level protocol headers and collected them in a pcap file. 63% of all sequences were identified as valid for HTTP and 100% for FTP by Wireshark. The excellent result on FTP is explainable for the same reasons as in the feature reverse engineering, suffering from the simplicity of the messages and training data. Figure 9 shows the type distribution of both the feature reverse engineering and the sequence generation for FTP to visualize the impact of adding the other steps via cluster indexing. Only one class stands out, which is either due to its simplicity or a miscategorization of FTP types on our part. The curve indicates, that a wider, more balanced distribution of types was inferred. For the comparatively lower HTTP result, as shown in figure 8, the messages displayed a greater correlation between requests and responses in the sequence generation. This implies that the added context of types allows the neural network to learn these connections more effectively. We expect that more capable and larger architectures will show an increase in the effectiveness of the observed behavior.

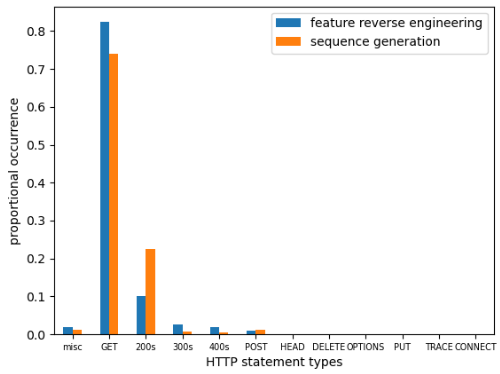

Figure 8: HTTP feature reverse engineering (blue) vs. sequence generation (orange) results in terms of type distribution. Sequence generation shows better request/response correlation.

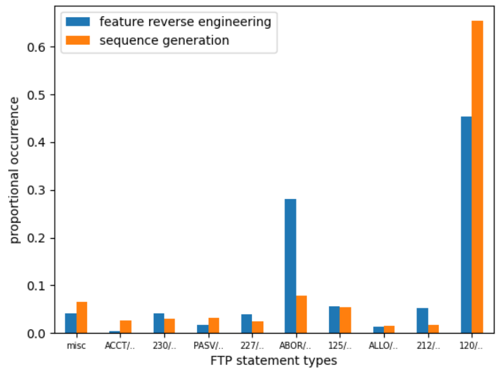

Figure 9: FTP feature reverse engineering (blue) vs. sequence generation (orange) results in terms of type distribution. Sequence generation shows a better distribution except for 1 class.

This result lays the final foundation we needed to enable deep learning-based fuzzing of any protocol, as a PRE approach is designed to be generally applicable.

5 RELATED WORK

Learn&Fuzz: Machine Learning for Input Fuzzing Based on the idea that fuzzing only requires an approximately correct input, the authors of (Godefroid et al., 2017) used a recurrent neural network to help in fuzzing a PDF parser in a browser. The syntax for objects, a central building block of PDFs, was inferred by an AI in this work. The authors used three different sampling strategies from their network to address various challenges in fuzzing: NoSample, Sample, and SampleSpace. The first one sees the RNN pick the letter with the highest score among the output to ensure syntactically correct objects. The second strategy takes the network's softmax output as a probability distribution for picking the next letter. This increases the fuzzing coverage but may result in many ill-formed objects. Lastly, the SampleSpace strategy combines the two previous. No-Sample is used until a whitespace is generated. The next following character is sampled by probability, only to switch back to the No-Sample strategy again. This creates more syntactically correct objects while also creating a more comprehensive mix of different objects. These strategies were used in the LSTM sampling. Our sampling, however, has to cover message context as well.

Machine Learning for Black-Box Fuzzing of Network Protocols The authors (Fan and Chang, 2017) see their work as the first attempt to combine an in-depth learning approach with black-box fuzzing. They deploy a sequence-to-sequence model, a two-part LSTM model invented for machine translation (Sutskever et al., 2014), to learn the semantics of network protocols and create new output for fuzzing purposes. Their work differs from this paper in the number of steps taken and models employed to learn details about the protocol. Here, a whole model is presented, which was derived from previous work on PRE. Additional information such as clustering or context-based generation is missing as well.

GANFuzz: A GAN-based Industrial Network Protocol Fuzzing Framework The GANFuzz Framework (Hu et al., 2018) represents an alternative approach to protocol fuzzing using deep learning but falls into the same category as the previous paper. Instead of a sequence-to-sequence model, GANFuzz uses a model known as a SequenceGAN (SeqGAN) (Yu et al., 2017). This is a version of the common GAN model but adjusted for text-based data using reinforcement learning. The authors of the paper train one SeqGAN per message type, which they deduce from a variety of clustering heuristics. This makes their approach more similar to PREUNN but is still missing several work steps, a concise work step mode, and the generation of data done by context symbols instead of entirely separate models.

Deep neural network-based automatic unknown protocol classification system using histogram feature Jung and Jeong demonstrated a machine-learning-based PRE approach to classify unknown protocols (Jung and Jeong, 2020). They use statistical methods to extract features from ten different protocols and feed them into a deep belief network, classifying the message based on these features. Based on the approach, they report a classification accuracy of about 99% on unknown datasets. Their method differs from PREUNN in several ways. They semi-automatically extract features using statistical methods, while we fully automated this process using an autoencoder. Furthermore, their work's goal is to classify unknown protocol messages, whereas we aim to generate new valid messages based on a novel protocol. Classification in PREUNN happens implicitly.

Network Traffic Classification (NTC) The topic of NTC is related to PRE regarding feature extraction. Several papers are combining NTC with ML and NNs (Lopez-Martin et al., 2017; Michael et al., 2017; Li et al., 2018). However, the more complex steps of PRE, like generating new packages for an unknown protocol, are not addressed in those papers.

6 CONCLUSION

PREUNN represents a novel approach for the separation of traditional protocol reverse engineering tasks that can be implemented using only neural networks. In this paper, the widespread application layer protocol HTTP v1.1 and FTP were successfully reverse-engineered using our approach. This highlights the potential of several deep learning architectures such as AEs for feature extraction, LSTMs for feature reverse engineering, SOMs for clustering, LSTMs for state recognition, and finally a combination of the above for sequence generation. The results achieved include a decent, intuitively agreeable clustering and a context-capable message generation model. We omitted optimizations and testing the use of PREUNN for fuzzing entirely due to time restraints and leave them for future work. Our modular approach allows the use of newer architectures from the fields of deep learning, such as natural language processing (e.g. BERT) for improvements. Further future work could also include the use of reinforcement learning utilizing these pre-trained models to create an automated fuzzer that is capable of reverse engineering any similarly structured message-based language.

REFERENCES

Bação, F., Lobo, V., and Painho, M. (2005). Self-organizing maps as substitutes for k-means clustering. In *International Conference on Computational Science*, pages 476–483. Springer.

Besic, N. (2019). What is a fuzzer and what does fuzzing mean. https://www.neuralegion.com/fuzzing-what-is-fuzzer/.

Brook, C. (2018). What is deep packet inspection? how it works, use cases for dpi, and more. https://digitalguardian.com/blog/what-deep-packet-inspection-how-it-works-use-cases-dpi-and-more.

Caballero, J., Yin, H., Liang, Z., and Song, D. (2007). Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329.

Comparetti, P. M., Wondracek, G., Kruegel, C., and Kirda, E. (2009). Prospex: Protocol specification extraction. In *2009 30th IEEE Symposium on Security and Privacy*, pages 110–125. IEEE.

Cui, W., Kannan, J., and Wang, H. J. (2007). Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14.

Duchene, J., Le Guernic, C., Alata, E., Nicomette, V., and Kaâniche, M. (2018). State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, 14(1):53–68.

Fan, R. and Chang, Y. (2017). Machine learning for black-box fuzzing of network protocols. In *International Conference on Information and Communications Security*, pages 621–632. Springer.

Garfinkel, S. (2008). Nitroba university harassment scenario. Dataset: https://digitalcorpora.org/corpora/scenarios/nitroba-university-harassment-scenario.

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE.

Goo, Y.-H., Shim, K.-S., Lee, M.-S., and Kim, M.-S. (2019). Http and dns traffic traces for experimenting of protocol reverse engineering methods. http://dx.doi.org/10.21227/tpqf-fe98.

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

Hinton, G. E. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313:504 – 507.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hornik, K., Stinchcombe, M., White, H., et al. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Hu, Z., Shi, J., Huang, Y., Xiong, J., and Bu, X. (2018). Ganfuzz: a gan-based industrial network protocol fuzzing framework. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 138–145.

Jung, Y. and Jeong, C.-M. (2020). Deep neural network-based automatic unknown protocol classification system using histogram feature. *The Journal of Supercomputing*, 76(7):5425–5441.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological cybernetics*, 43(1):59–69.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Lang, K. J. (1999). Faster algorithms for finding minimal consistent dfas. *NEC Research Institute, Tech. Rep.*

Li, R., Xiao, X., Ni, S., Zheng, H., and Xia, S. (2018). Byte segment neural network for network traffic classification. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–10.

Lopez-Martin, M., Carro, B., Sanchez-Esguevillas, A., and Lloret, J. (2017). Network traffic classifier with convolutional and recurrent neural networks for internet of things. *IEEE Access*, 5:18042–18050.

Michael, A., Valla, E., Neggatu, N. S., and Moore, A. (2017). Network traffic classification via neural networks. Technical Report UCAM-CL-TR-912, University of Cambridge, Computer Laboratory.

Narayan, J., Shukla, S. K., and Clancy, T. C. (2015). A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys (CSUR)*, 48(3):1–26.

Pang, R. and Paxson, V. (2003). Lawrence Berkeley National Laboratory - FTP - Packet Trace. Dataset: https://ee.lbl.gov/anonymized-traces.html.

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626.

Sharafaldin, I., Lashkari, A. H., Hakak, S., and Ghorbani, A. A. (2019). Developing realistic distributed denial of service (ddos) attack dataset and taxonomy. In *2019 International Carnahan Conference on Security Technology (ICCST)*, pages 1–8. IEEE.

Shiravi, A., Shiravi, H., Tavallaee, M., and Ghorbani, A. A. (2012). Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *computers & security*, 31(3):357–374.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

TC97, I. (1984). Basic reference model. *International Standard, ISO/IS*, 7498.

Wondracek, G., Comparetti, P. M., Kruegel, C., and Kirda, E. (2008). Automatic network protocol analysis. In *NDSS*, volume 8, pages 1–14.

Yu, L., Zhang, W., Wang, J., and Yu, Y. (2017). Seqgan: Sequence generative adversarial nets with policy gradient. In *Thirty-first AAAI conference on artificial intelligence*.