Journal Pre-proof

RNNIDS: Enhancing Network Intrusion Detection Systems through Deep Learning

Soroush M. Sohi, Jean-Pierre Seifert, Fatemeh Ganji

PII: S0167-4048(20)30424-7
DOI: https://doi.org/10.1016/j.cose.2020.102151
Reference: COSE 102151

To appear in: *Computers & Security*

Received date: 20 December 2019
Revised date: 6 December 2020
Accepted date: 13 December 2020

Please cite this article as: Soroush M. Sohi, Jean-Pierre Seifert, Fatemeh Ganji, RNNIDS: Enhancing Network Intrusion Detection Systems through Deep Learning, *Computers & Security* (2020), doi: https://doi.org/10.1016/j.cose.2020.102151

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2020 Published by Elsevier Ltd.

Journal Pre-proof

RNNIDS: Enhancing Network Intrusion Detection Systems through Deep Learning

SOROUSH M. SOHI, Security in Telecommunications
Technische Universität Berlin
JEAN-PIERRE SEIFERT, Security in Telecommunications
Technische Universität Berlin
FATEMEH GANJI, Electrical and Computer Engineering Department
Worcester Polytechnic Institute

Security of information passing through the Internet is threatened by today's most advanced malware ranging from orchestrated botnets to simpler polymorphic worms. These threats, as examples of zero-day attacks, are able to change their behavior several times in the early phases of their existence to bypass the network intrusion detection systems (NIDS). In fact, even well-designed, and frequently-updated signature-based NIDS cannot detect the zero-day treats due to the lack of an adequate signature database, adaptive to intelligent attacks on the Internet. More importantly, having an NIDS, it should be tested on malicious traffic dataset that not only represents known attacks, but also can to some extent reflect the characteristics of unknown, zero-day attacks. Generating such traffic is identified in the literature as one of the main obstacles for evaluating the effectiveness of NIDS. To address these issues, we introduce RNNIDS that applies Recurrent Neural Networks (RNNs) to find complex patterns in attacks and *generate* similar ones. In this regard, for the first time, we demonstrate that RNNs are helpful to generate new, unseen mutants of attacks as well as synthetic signatures from the most advanced malware to improve the intrusion detection rate. Besides, to further enhance the design of an NIDS, RNNs can be employed to generate malicious datasets containing, e.g., unseen mutants of a malware. To evaluate the feasibility of our approaches, we conduct extensive experiments by incorporating publicly available datasets, where we show a considerable improvement in the detection rate of an off-the-shelf NIDS (up to 16.67%).

Additional Key Words and Phrases: **Network Security; Network Intrusion Detection Systems; Worm Mutants; Dataset Generation; Deep Learning; Recurrent Neural Networks.**

1 INTRODUCTION

Nowadays we are witnessing rapidly escalating Internet threats, which have become increasingly mature as the Internet and its applications evolve. Today's Internet provides ubiquitous connectivity to a wide range of devices, with different operating systems, which indeed expands the available attack surface including several different attack vectors. As a prime example, according to the recent Symantec report[1], a significant increase can be observed in different classes of attacks, e.g., internet of things (IoT) devices (more than 600%), new downloader variants (more than 92%), etc. [58]. This increase has been partially fueled by the increased availability of user-friendly hacking tools, demanding solely superficial knowledge from attackers, as illustrated first by Lipson [39] and further extended in [11]. Among malicious activities practiced by attackers, several classes of attacks can be recognized, for instance, Denial of Service (DoS) [40], disclosure, manipulation,

[1]Symantec^{TM} has established the largest civilian threat collection network in the world, and has one of the most comprehensive collections of cyber security threat intelligence through the Symantec Global Intelligence Network composed of about 126 million attack sensors [57].

Journal Pre-proof

impersonation, and repudiation. These classes can be lumped together by an umbrella term, namely intrusion. Along with the emergence of increasingly sophisticated intrusions, intrusion detection systems (IDS) have been developed to cope with these threats. Regarding where or at which point an IDS is placed, two types of such systems can be distinguished: network intrusion detection systems (NIDS), and host intrusion detection systems (HIDS) [14]. The latter is run on a device or an individual host in the network, whereas an NIDS is located within the network, at a strategic point, to monitor the traffic to and from all devices. Irrespective of this classification, the intrusion detection systems share some commonalities; first, they take advantage of the connectivity provided by networks, and secondly, they either use the known, specific patterns or apply anomaly detection techniques.

Anomaly detection-based systems aim to establish behavior patterns, being different from the normal behavior of the system [70]. Although these systems can be employed to detect previously unknown malicious activities, the main drawback of them is the high rate of false positives, i.e., a legitimate activity may be categorized as malicious [17]. On the other hand, signature-based detection systems attempt to match a known rule, a so-called *signature*, with the contents of packets [13]. To this end, after receiving an incoming traffic, it undergoes a careful and continual process of analyzing and possibly generating signatures. This approach is similar to virus scanning mechanisms, where the database of the signatures, should be kept updated that can take from a few minutes to a few days. Despite this fact, the alternative method, i.e., anomaly-based detection, may require more processing power than signature-based ones. Additionally, although a less number of rules are required in comparison to signature-based systems, in highly dynamic environments it can be challenging to train anomaly-based detectors. Nevertheless, a careful signature definition can enable us to benefit from some of the advantages of using anomaly detector [41]. To this end, two major obstacles can be identified, as expalined below.

Generating signatures for intrusion variants: The result of the signature generation process should be a set of *effective* signatures, being narrowed down enough to characterize a specific attack, but flexible enough to detect some variations or modifications in an attack [31]. In other words, the probability of classifying benign traffic as malicious (i.e., false positives) must be low as well. Moreover, with high probability (i.e., low rate of false negatives) an effective signature should detect an attack, and possibly, its variants. The latter is related to the fact that although every day hundreds to thousands of signatures are generated and/ or updated by institutes and companies responsible for intrusion detection [9], attackers create variants or *mutants* of an intrusion to evade detection. Several different ways to generate an attack variant can be considered, ranging from simple interleaving of malicious data to sophisticated obfuscation of that. The point is that even a simple, small modification results in a completely new attack, similar to a new unknown zero-day attack, which requires performing the whole process of signature generation. In an ideal world, the signature generation should be conducted automatically so that an NIDS analyzes the incoming traffic and can distinguish between malicious and benign traffic with regard to inherent and unique characteristics of an attack, which offer a basis for generating the respective signatures. In addition to this, the problem with the derivation of an adequately specific signature are two of the greatest challenges faced by NIDS designers today.

Testing an NIDS by acquiring data as the ground truth: Another serious obstacle to the implementation of an NIDS is how the effectiveness of the NIDS should be evaluated. For this purpose, traditionally a set of malicious data should be collected to serve as ground truth. Being close to the real traffic passing through real networks is one the specifications of such a set. DRAPA dataset [33], KDD Cup 99 [51], and CDX [15, 49] can be considered as attempts to address this issue. However, they suffer from a lack of nearly real traffic data, and they do not include traffic for all of

Journal Pre-proof

the network protocols and all variants of attacks in today's world [72]. As an example of remedies for this, in [52] another dataset called "Kyoto 2006+" for research purposes has been offered. In one of the most recent attempts, the authors of [7] have combined several types of malicious datasets available on the Internet to produce a mixed dataset, which satisfies the requirement of having a wide range of various attacks. They have applied a method called "overlay methodology" to produce synthetic dataset [5]. According to this method, data related to malware activities is merged with benign data by sending the data to the machines on an external network.

Despite the above, according to at least two main reasons, collecting a set of *malicious data* providing the ground truth is not straightforward [5]. First and foremost, typical network traces contain sensitive information, which is carefully controlled and cannot be shared with other parties. This is due to the fact that even after a cautious anonymization process, it can be still possible to extract some sensitive information from the data [43]. Second, given that an NIDS is tested by feeding a small set of data, due to the heterogeneous nature of the real-world network traces, the results of the test cannot be representative. Hence, it has been proposed to use synthetic data traces, see, e.g., [69]. Nevertheless, the impact of possible biases and limitation with respect to realism should be considered in this case [5]. More importantly, the synthetically generated traces mainly follow known trends such as distributions of users, applications as well as the network behavior, and do not reflect further detailed characteristics of the traffic, e.g., the payload that is crucial for efficient intrusion detection. Therefore, it is of great importance to establish a methodology which can be used to achieve realism to a higher degree.

Furthermore, and more crucially, when a detector is implemented and its effectiveness against zero-day attacks should be tested, it is necessary to generate a traffic pool reflecting the nature of the zero-day attacks. In the literature, this has not been completely addressed so far. As a prime example, [28] re-defines zero-day attacks as the attacks discovered after the NIDS under study is released. By taking this into account, it has been shown that a signature-based NIDS, namely SNORT [48], can detect a set of zero-day attacks. Unfortunately, the issue with ground truth remains a critical challenge.

Our contributions: This work aims at removing obstacles mentioned above by contributing to the following points.

(1) Generating malware mutants: By applying a novel and sound method, we demonstrate how mutants of a malware can be generated. This has been illustrated in Figure 1 and marked as Step 1. Our method relies on the fact that an unknown pattern, so-called grammar, can be extracted and learned by a recurrent neural network (RNN). This fact enables us to generate new, unseen sequences, i.e., attack mutants. These new variants of a malware can be indeed helpful, when there is a need to generate a dataset that serves as the ground truth.

(2) Generating attack signatures: As another example of how RNNs can be used in intrusion detection, we demonstrate that by extracting the signatures of an NIDS and feeding them into an RNN it is possible to generate synthetic signatures. These signatures can be further added to the database of an NIDS to improve its performance in terms of finding unseen mutants of an attack, as shown in Figure 1 (see Step 2, for the results, see Section 6.3 "Experiment 1").

(3) Synthetic data generation: The methodology that we apply in this paper can represent a change of direction for generating synthetic data. More specifically, we show that along with applying the overlay methodology by using synthetic signatures and mutants generated by an RNN, it is possible to generate synthetic data to test an NIDS. We stress that when evaluating the performance of the NIDS by feeding this synthetic data, the synthetic signatures used to generate that should have been removed from the database of the NIDS (see Section 6.3 for more details).

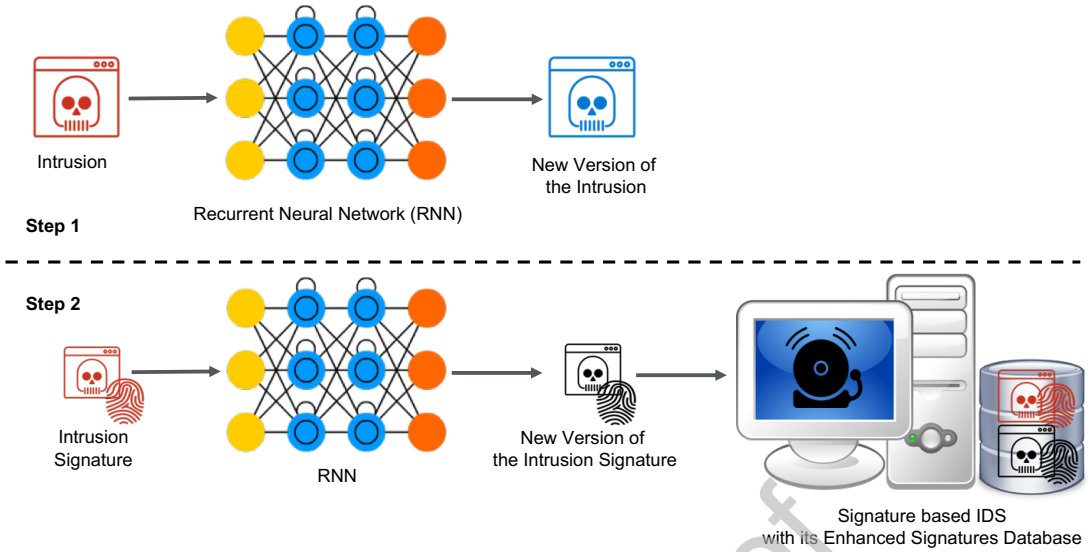


Fig. 1. Two steps that are taken in our framework to present how deep learning, namely, Recurrent Neural Networks (RNNs), can improve the effectiveness of an NIDS. The first step shown in this figure corresponds to our first contribution, namely, generating attack mutants. The second step is taken by us to generate not only attack signatures, but also synthetic data that can be used to evaluate the performance of an existing NIDS.

Last but not least, we verify the relevance of the above claims empirically by applying the proposed concept to one of the well-studied, off-the-shelf NIDS, namely Zeek (formerly Bro, henceforth called Bro) NIDS [71]. According to our experimental results, when incorporating a publicly accessible dataset, the performance of Bro can be improved by up to about 17%. Nevertheless, the scope of our work is not restricted to this NIDS. In other words, our methodology is applicable in other scenarios, where a signature-based NIDS is employed to find an intrusion.

Organization: This paper is structured as follows. Section 2 introduces the notations and concept used to describe our scheme. Section 3 provides a brief overview of the mechanism of Recurrent Neural Networks (RNNs) and how they can be used to generate texts. In Section 4, we elaborate on our methodology applied to generate new mutants of malware as shown in Step 1 in Figure 1 (see Section 4.1), while Step 2 in our framework is described in Section 4.2. Moreover, Section 5 is devoted to the definition of the metrics and the description of the experimental setup, and the design of the experiments. We present our experimental results in Section 6, and finally, conclude the paper in Section 7.

2 NOTATIONS AND DEFINITIONS

Although we assume that the reader can be familiar with the concept of regular languages, Deterministic Finite Automata (DFA), and deep learning, we define the functions and notations used throughout this paper. Note that standard notation is used here, as found in [23].

Preliminaries on Formal Languages and DFAs

Consider the alphabet $\Sigma = \{0,1\}$ and the set of all strings Σ^* over Σ. A set $L \subseteq \Sigma^*$ is called a language over the alphabet Σ.

A grammar is a 4-tuple (N, V, P, S), with the sets of non-terminal (N) and terminal (V) vocabularies, i.e., strings, a finite set of production rules (P), and the start symbol (S). For each grammar, there exists a language corresponding to that as well as an automaton recognizing the strings of

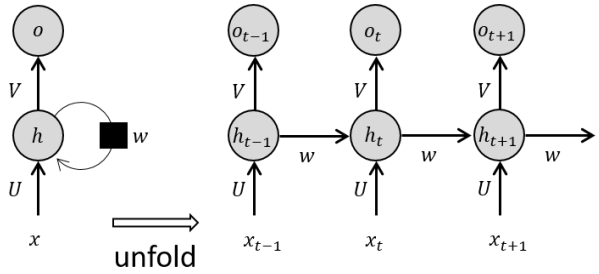


Fig. 2. In an RNN, an input from the previous state of the hidden layer is fed into the next hidden layer designated with a weight w. Weights corresponding to the input and the output layers are U and V [36].

that grammar. Our focus is on deterministic and regular grammars, which can be recognized by a DFA.

A DFA A is defined by $A = (Q, \delta, \Sigma, q_0, F)$ over the alphabet Σ, where Q is the set of states, the initial state is denoted by q_0, and the accepting states are $F \subseteq Q$. $\delta : Q \times \Sigma \to Q$ is the transition function defined as follows. For all $q \in Q$, $a \in \Sigma$ and $c \in \Sigma^*$, we have $\delta(q, \lambda) = q$ and its canonical continuation to Σ^*, i.e., $\delta(q, ac) = \delta(\delta(q, a), c)$. Giving strings to A, it accepts a set of them, called its accepted language $\mathbf{L}(A) := \{c \in \Sigma^* \mid \delta(q_0, c) \in F\}$, i.e., a regular language.

From another perspective, particularly by taking a bottom-up approach, regular languages can be described by their respective regular expressions (often called, regex). In an informal and intuitive way, regular expressions enable us to begin with building blocks and combine them to generate other regular expressions. These building blocks are regular expressions representing the empty language $\{\emptyset\}$, the language of the empty string $\{\lambda\}$ with $|\lambda| = 0$, and the languages of the sets containing only one alphabet of Σ, i.e., $\{a\}$ s.t. $a \in \Sigma$. In order to combine these building blocks and obtain new regular expression, we can apply the concatenation, the union, the Kleene closure, and the intersection operators (for more details see, e.g., [29]). It has been proved that L is a regular language iff there is a regular expression R such that $\mathbf{L}(R) = L$ [29]. Moreover, it is known that DFAs and regular expressions are equivalent, i.e., if a language L is built up by regular expressions s.t. $\mathbf{L}(R) = L$, there exists a DFA A that accepts this language, i.e., $L = \mathbf{L}(A)$ [29].

Notations Related to Deep Neural Networks:

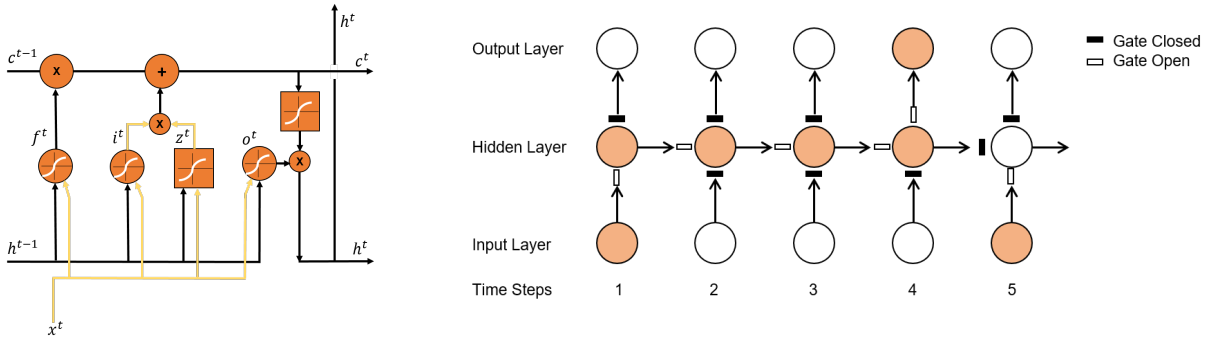
One of the functions widely applied when working with deep learning models is (logistic) sigmoid: $\sigma(x) = 1/1 + \exp(-x)$. In addition to the sigmoid function, the hyperbolic tangent function, $\tanh(\cdot)$, is used as an activation function in deep networks.

3 BRIEF OVERVIEW OF RECURRENT NEURAL NETWORKS (RNNS)

Although the concept and applications of recurrent neural networks (RNNs) are nowadays part of the common knowledge in our community, as a self-contained paper, our work briefly introduce the most important concepts required to understand our method.

The concept of deep neural networks, as an approach to artificial intelligence (AI), has been first motivated by problems that could not be addressed properly in traditional machine learning. As examples, speech recognition and generating text [56] can be mentioned [23]. One of the most prominent, and widely-applied neural networks is feed-forward networks, where via a series of weights the inputs is given to the outputs directly. Although being a very powerful tool for several applications, feed-forward networks cannot cope with issues, in which a sequence of inputs should be processed. To address this, RNNs have been proposed (see, e.g., [50]), which exhibit temporal characteristic fulfilling the conditions for processing sequences of data.

Journal Pre-proof



(a) (b)

Fig. 3. (a) Structure of an LSTM memory cell. The previous states of the cell and the hidden states could be erased, updated, and read by the gates. Squares are hyperbolic tangent activation functions and circles are logistic sigmoid activation functions, (b) Input, output, and forget gates in an RNN with LSTM cells. These gates control the flow of information in different time steps [18, 24].

In addition to being superior in learning sequences [24], RNNs can be employed to generate sequences that are similar to training sequence [36]. In Comparison to feed-forward neural networks, each stage of an RNN has an input coming from the previous state, in a similar way to memorizing the history of all the past states (see Figure 2). In the fields of speech recognition and Natural Language Processing (NLP), RNNs are usually used to predict the next word in a sequence.

A major problem with RNNs is the vanishing or exploding gradients in back-propagation process [23]. One of the first designed RNN (so-called vanilla RNN), which is the simplest one suffers from this problem. In other words, legacy, vanilla version of RNNs could not save the memory for a long time and it could vanish during the time. Consider an RNN that is trained over 10000 samples and builds up the memory upon those. As each new sample with new attributes arrives in a sequence, the memory is overwritten after a while and it forgets the previous memory. Long Short-Term Memory (LSTM) RNNs have attempted to solve this problem [24]. The concept of LSTM was first introduced in 1997 by Hochreiter and Schmidhuber [27]. An LSTM RNN can be turned to a gated RNNs, which has the capability to learn for a long time, and, when needed, it has the gates to forget the memory and learn again based on new inputs [18]. These gates also allow RNNs to pass the information unchanged to other layers in the network, i.e., featuring read, write and clear functions as illustrated in Figure 3b. When a gate is open, it allows the information propagation through the gate. More formally, in Figure 3a, i^t , f^t, and o^t are the output signals of the corresponding input, forget, and output layers at the time step t, whereas h^t is an updated hidden state value used for the next time step and the output to the upper layer at that time step. Moreover, c^t is the cell state (so-called, memory state) fed into the next time step, and similarly, c^{t-1} is the cell state coming from the previous time step. And let W_f, W_i, W_z, and W_o be the input weights related to each gates, and in a similar fashion, R_f, R_i, R_z, and R_o be corresponding to the recurrent inputs from the previous states. Now the output of the input layer at the time step t can be formulated as follows.

$$i^t = \sigma(W_i x^t + R_i h^{t-1} + b_i),$$

where b_i is the bias for the input gate. By substituting the weights and biases for the forget and output gates in the above equation, we obtain the outputs of the other layers, namely f^t, o^t, and z^t. Finally, the cell state at the time step t is $c^t = f^t \cdot c^{t-1} + i^t \cdot z^t$, and the updated hidden state $h^t = o^t \cdot \tanh(c^t)$.

Journal Pre-proof

The structure explained above enables LSTM RNNs to cope with long-term dependencies more effectively than the simple recurrent networks [23]. Therefore, LSTM RNNs have become powerful and widely accepted tools for speech [25] and handwriting recognition as well as text generation [24].

3.1 Application of RNNs in Text Generation

Sequence learning offers a wide range of applications, for instance, natural language processing, time series prediction, and DNA sequencing. Due to these applications, sequence learning can be considered as a discipline or of a particular line of research developed to address problems such as sequence prediction, sequence recognition, etc. [55]. Among those problems, sequence prediction (mostly referred to as sequence generation) attracts a great deal of attention because of its applications in several domains of study, e.g., human-machine interaction [34].

After the development of RNNs in 1980s, they are widely used in enormous studies related to sequence prediction. Giles et al. have pursued this line of research by demonstrating fundamental strength of RNNs and their close relationship to deterministic finite automata, see, e.g., [19–21, 45]. They began with a formal model of sequences, i.e., formal grammar and machines generating and recognizing them, namely their respective automata. More specifically, they have proved that RNNs can be trained to simulate deterministic finite automata (DFA) and recognize their corresponding grammar [21]. Moreover, it has been demonstrated that, even unknown, (small) grammars can be learned and further extracted from the RNN (i.e., generating the corresponding DFA) [20]. It is worth noting here that the problem of grammatical inference is proved to be NP-complete [22], and results presented by Giles at al. are in line with heuristic approaches attempting to address grammatical inference (see, e.g., [3]).

Comparison to Generative Adversarial Networks (GANs): One may argue why GANs have not been taken into account in our framework. The point is that GANs have achieved a promising result in the image generation domain; however, for text generation, several attempts have failed as discussed below. When combined with RNNs, GANs cannot be trained effectively due to the non-differentiable nature of generating discrete symbols. Hence, pre-training and joint training methods have been suggested to tackle this issue, see [46] for an exhaustive discussion. On the other hand, CNN-based GANs are also studied for text generation, see, e.g., [26], which lead to less desirable results, namely the generated text contains spelling errors and has had little coherence. Therefore, for our purpose, we stick to RNNs, which not only require relatively less effort to be trained, but also generate more coherent text.

4 METHODOLOGY: APPLICATIONS OF RNNS IN INTRUSION DETECTION

This section covers the methodology applied to generate mutants of worms as well as synthetic signatures used in an NIDS. The main idea behind these observations is to demonstrate not only the possibility of generating new variants of worms and signatures from a small number of examples (see Figure 1, Step 1), but also to show how these variants can extend the dataset of an NIDS to improve its detection rate (Step 2 in Figure 1). More specifically, this section is mainly devoted to theoretical and practical approaches that we apply to generate those variants. In fact, we show that an LSTM RNN can extract the "deep structural properties" of an encoded worm and a signature to generate variants of them.

4.1 Generating New Mutants of Polymorphic Worm

As discussed in Section 1, one of the major obstacles that signature-based NIDS encounter is the lack of knowledge about new variants of an attack. As an example of attacks with enormous variants,

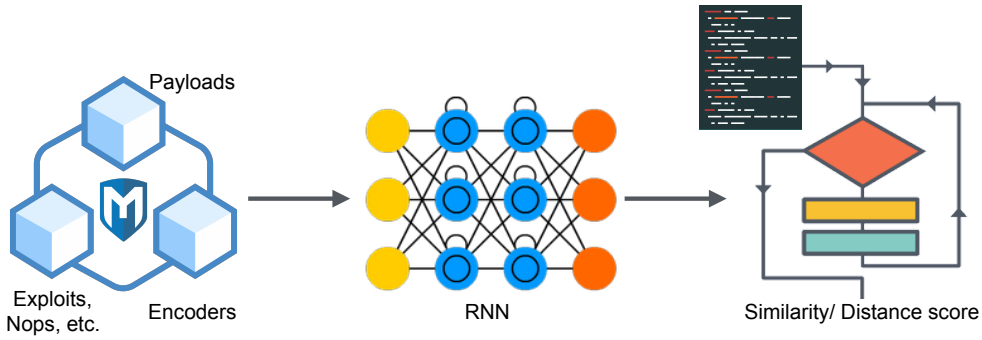


Fig. 4. Steps taken in our approach: (from left to right) generating the worms and their mutants using the Metasploit framework, then feeding them into an RNN, and finally compare the results in terms of the similarity.

we focus on polymorphic worms. It is known that these worms are able to change their behavior through generating mutants or variants of themselves to pass through NIDS without being detected. They attempt to hide their encrypted malicious code in all variants of themselves. The notion of "generating mutants" of an exploit has been introduced in the literature, which should not be confused with mutation testing. Following the reasoning provided in [68], in our scenario we generate mutants of an attack and the targeted NIDS remains untouched, in contrast to mutation testing approaches. An example of a mutant generator for NIDS, compatible with our scenario, has been introduced in [68]. Similar to our goal, [68] aims at evaluating the effectiveness of an NIDS by feeding mutants of a known attack. Nonetheless, the main drawback of their proposed method is that the mutation mechanism should be maintained continuously. In other words, the existing mechanisms have to be frequently updated, and more importantly, the parameters reflecting the nature of an attack should be extracted before generation process begins. This section of our paper attempts to address these issues.

More specifically, the primary goal of our approach described in this section is to observe if new, unseen variants or mutants of a known worm can be generated by an LSTM[2], given some previously collected variants of that. In particular, we take the steps illustrated in Figure 4. To provide proof of concept of how our approach can be applied, here we focus on different state-of-the-art encoder engines implemented in ADMMutate [1] as well as the Metasploit framework [62], namely the XOR encoder [65] and the Shikata Ga Nai (in Japanese means "nothing can be done about it") encoder [64]. Using such encoders are crucial since when an exploit in the form of, e.g., a shellcode is generated, it cannot be directly used. But it should be encoded to obtain a pure alphanumeric code, by removing bad characters (e.g., null bytes). Moreover, the encoded exploit may suit 64-bit target systems, which can be achieved by deploying an encoder. For instance, in the Metasploit framework, the XOR encoder [65] employs an 8-byte key and leverages relative addressing used by x64 operating systems, whereas the Shikata Ga Nai encoder is a polymorphic XOR additive feedback encoder for the x86 architecture. The decoder stub of this encoder is generated by substituting the instructions and ordering the blocks dynamically. In this regard, after each iteration different outputs are generated in the hope that signature recognition can be prevented. Furthermore, the key used by the encoder is modified through additive feedback. Additionally, the decoder stub is also obfuscated (for more details, see [16]). Beside the XOR and Shikata Ga Nai encoders, we also consider the ADMMutate engine exhibiting the following features. The payloads are encoded by

[2]Hereafter, we may use the terms "RNN" and "LSTM" interchangeably, since we solely focus on and implement LSTM RNNs.

Journal Pre-proof

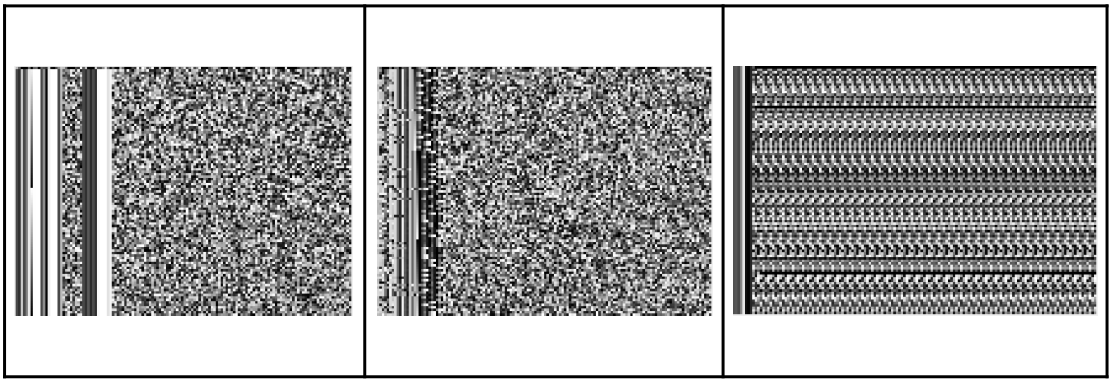


Fig. 5. So-called spectral images illustrating the patterns (the visible columns) formed by the bytes repeated in the same places within all encoded bash files. In all examples, each row of the pixels shows a decoder generated by an engine, whereas each pixel corresponds to a byte of that decoder. Although in comparison to the XOR encoder (left) such patterns are less visible for the Shikata Ga Nai (middle) encoder and ADMMutate (right), they can be still distinguished.

applying 16-bit sliding keys. Moreover, the ADMMutate supports randomized NOP generation, banned characters, insertion of non-destructive junk instructions and the reordering/substitution of code as well as polymorphic payload decoder generation with multiple code paths [68].

Comparing the Shikata Ga Nai encoder with the XOR encoder, as indicated by Metasploit [62], it is expected that Shikata Ga Nai encoder outperforms the XOR encoder. However, successful detection of exploits using these encoders have been reported in the literature, see, e.g., [53] . This is due to the fact that these encoders leave traces in the encoded payload that can be used to detect the exploit. A visual, effective procedure proposed in [53] to illustrate such traces for, e.g., Shikata Ga Nai encoder. We follow that procedure to see if for the ADMMutate, the XOR, and the Shikata Ga Nai encoders any pattern can be observed, which can be further used by an LSTM to generate new mutant of a worm.

To this end, we define the following setting. To generate *examples* of exploits in the form of bash files, we apply the Shikata Ga Nai, XOR, and ADMMutate encoders to the OS X x64 Shell Bind TCP payload [63], which binds an arbitrary command to a port chosen by the attacker. Moreover, the number of iterations for each encoder varies from zero (i.e., without encoding) to 100 so that 101 bash scripts are generated. These bash files are sorted in matrices in a row-wise manner. Note that we add padding (all zero values) at the end of the rows to obtain rows with the same length since for these encoders, the outputs of these encoders are non-fixed length bash files. Afterwards, the matrices are displayed as grayscale images, where a byte value of 0x00 (0xFF) corresponds to a black (white) pixel. Figure 5 shows portions of the worms encoded by the Shikata Ga Nai, XOR, and ADMMutate encoders.

The most important message to be conveyed here is that the bash files generated by applying the above-mentioned encoding engines do not comprise of completely, and truly random values. In other words, there are some patterns, i.e., the bytes repeated in the same places within all bash files, which form the visible columns in Figure 5. We stress that while the results discussed above should not be extrapolated to all payloads/encoders, they show patterns that can be useful for generating new variants of an attack. This is in line with what has been observed in [38], where the approach relying on finding unavoidable byte patterns is called “content-based”. According to this definition, our approach can be classified as content-based as well. The main advantage of such approaches is that there exists no dependency on protocol or server information [38]. As well described by

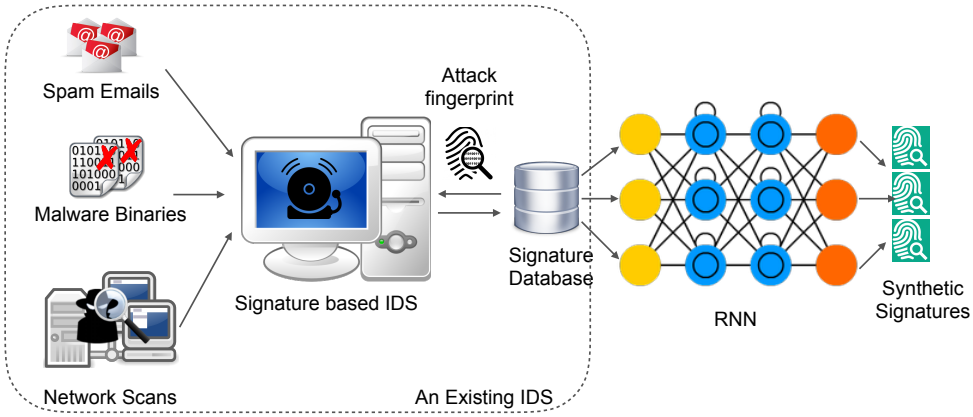


Fig. 6. Our framework to extend and enhance an existing NIDS: (from left to right) extraction of the signatures stored in the database of the NIDS, then feeding them into an RNN, and at the end the synthetic signatures generated by the RNN used to extend the database of the signatures.

Li et al., in the real-world, the protocol frame part and the control data of the worm cannot be manipulated by an attacker to obtain a new mutant of a worm [38]. These parts create patterns that can be indeed used by an LSTM network to generate similar and unseen bash files as mutants of the original payload.

More formally, by providing examples of worms we train our LSTMs to learn and extract rules corresponding to a DFA A, as proved by Giles et al. [20]. This can be explained by the fact that the grammar of bounded-length strings is regular. Recall that there is a one-to-one mapping between a DFA and its grammar. Let the grammar corresponding to the DFA A be denoted by (N, V, P, S), and the language generated by this grammar be L'. In fact, $L' = Enc(L)$, where $Enc(\cdot)$ represent the encoding operation applied to encode the original language generating a worm L. After training, the grammar (N, V, P, S) is used by the LSTM to generate unseen strings, which belong to L'.

Up to this point, we have discussed the first two steps of our approach shown in Figure 4, namely generating the original worms and their respective mutants by applying the concept of LSTM. In Section 6, we discuss the last step that is computing the similarity or the distance score. Additionally, we provide the results that we obtain through empirical evaluation of our approach in Section 6.

4.2 Generating Synthetic Signatures

As discussed in Section 1, we mainly focus on signature-based intrusion detection methods. Although both signature- and anomaly-based classes of NIDS may suffer from false positives, signature-based systems can exhibit a lower rate of that [2, 9, 41]. The popularity of signature-based NIDS could stem from this fact.. As a prime example of such popular NIDS, one can mention Zeek (previously called Bro) [71] and Snort [48]. Albeit significant developments and improvements in the design of signature-based NIDS, attackers are often steps ahead of these defense systems. This can be explained by the fact that nowadays, a wide range of methods and tools are accessible by an attacker, and the attack patterns and behaviors change rapidly.

Generation of signatures for known attacks has become a mature area of research. In this context, numerous approaches applying various methodologies, ranging from machine learning and data mining (see, for example, [37, 67]) to matrix factorization [32], have been proposed in the literature. Nevertheless, our work does not share a great deal of commonality with these approaches. It is due to our approach that can be seen as an add-on to each and every NIDS to improve the effectiveness of that. One can observe a closer relationship between our approach and studies discussing synthetic

Journal Pre-proof

data generation for evaluation of NIDS, e.g., [6]. A comparison between the method employed in [6] and our approach can explain why we shift our focus to RNNs from Markov chain models, as proposed by Bar et al. [6]. Although being a predominant framework for different applications, Markov chain models suffer, in particular, from the following disadvantages. They need a significant amount of knowledge of a task performed by them. Moreover, the assumptions underlying the design of these chains can be questionable, e.g., the dependency assumptions. Our RNN-based approach attempts to address these issue.

Although we do not limit our approach to Bro NIDS, we take that as a prime example of a popular signature-based NIDS. Bro is an open source application running on Unix based systems. It has its own scripting language, enabling users to write the user-specific events and logs scripts. Additionally, Bro has the capability of changing the structure at many levels according to the user demands. It supports many popular transmission protocols and is able to analyze them. Moreover, it can extract the data packets and match the data with the predefined rules. Last but not least, Bro supports regular expression for pattern matching [66]. In more details, in each signature embedded in Bro, the information related to the payload of the attack is represented by its corresponding regular expression.

The latter feature of Bro is interesting from the point of view of our methodology. As can be seen in Figure 6, which illustrates our roadmap for generating synthetic signatures, the signatures of Bro is fed into an LSTM. More precisely, the regular expression R related to the payload of an attack is given to the LSTM. When the LSTM learns the DFA associated with R, it extract the DFA A such that $\mathbf{L}(R) = L$, and equivalently, $L = \mathbf{L}(A)$. In the next step, the LSTM uses the information about L, and A to generate another valid regular expression R', where $\mathbf{L}(R') = L$. Afterwards, the languages L and $L' = \mathbf{L}(R')$ can be combined by using, e.g., the union operator to obtain an enhanced regular language $L \cup L'$, with the associated regular expression $R \cup R'$. This union can be used as a substitute for R to improve the detection rate of Bro. We evaluate this quantitatively and present the results of deploying this method in Section 6.

5 EXPERIMENT DESCRIPTION

Before providing the results achieved by experimentally evaluating the performance of our methods (see Section 4), here we first introduce the metrics, experimental setup, and explain the design of the experiment.

5.1 Metrics

We define two comprehensive sets of metrics used for (1) computing the distance, or equivalently, comparing the similarity between the original sample (i.e., a set of original worms or Bro signatures) and samples generated by an RNN, and (2) evaluating the performance of the enhanced Bro in practice.

Similarity Comparison (Distance Measuring)

In order to examine how good the outputs of the RNN are, i.e., how far they are from the original samples, two famous similarity metrics are taken into account.

Levenshtein distance: this distance metric defines how many substitutions, deletions or insertions are required to transform one string to another one. To assess the similarity of two strings, this metric has been used in the intrusion detection-related literature, e.g., [10, 60]. Beside that, it has its drawbacks: the main drawback of the Levenshtein algorithm is that it focuses on the global comparison between two strings, i.e., among all the variables in two strings. For example, the Levenshtein similarity percentage between “John Smith” and “Smith, John” equals zero. While

Journal Pre-proof

we are interested in performing local similarity comparisons to find all pairs of substrings in two strings, we consider the Smith-Waterman distance as well.

Smith-Waterman distance: the Smith-Waterman algorithm is an alignment algorithm [44], which gives a score to the strings. This score is based on a calculation of three main factor, namely match, mismatch, and the penalty score. A match is mostly a positive number, and it indicates that one character is aligned to a character in another string, whereas any mismatch reduces the score. The penalty score is reduced from the score and indicates how long one match score could continue. In [8], it is demonstrated that the maximum score is not a factor of similarity and should be further normalized. The authors of [4] have addressed this by suggesting a new normalized factor based on the local distance between the position of the maximum score and the starting point, where the score is zero. We follow their procedure to compute the (normalized) similarity percentage between two strings.

Acceptable ranges of the similarity percentage: note that in contrast to typical machine learning tasks, our approach does not aim to achieve a high similarity percentage. Indeed, achieving a high similarity percentage implies that the variants generated by our framework are similar to an example (e.g., a worm or a signature) previously seen by the algorithm, and consequently, can be detected by the existing NIDS. This is contrary to our objective that is generating unseen mutants, which are only *close enough* to known examples. In this regard, if the similarity percentage is low (i.e., considerably less than 50%), it means that the algorithm could not extract the pattern needed to generate a new mutant.

Assessing the Quality of an NIDS

To evaluate how the quality of an NIDS in terms of attack detection is improved by employing our approach, we use the following metrics.

False Positives (FP): the number of cases, where the NIDS improperly detects a flow from the benign dataset as malicious.

False Negatives (FN): the number of cases, where a flow from the malicious dataset is not detected by the NIDS.

Furthermore, True Positives (TP) and True Negatives (TN) are defined, where the former indicates the number of cases that the NIDS correctly detects malicious flows. On the other hand, TN denotes the number of cases, where the NIDS correctly classifies the benign flows as "benign." In addition to these, we use the following frequently-used metrics defined to evaluate binary classification in the context of intrusion detection[9, 59].

- *Sensitivity*: the ratio of flows correctly classified as malicious to all malicious flows $TP/(TP+FN)$. Sensitivity is also called TP rate.
- *Specificity*: the ratio of flows correctly classified as benign to all benign flows $TN/(TN+FP)$.
- *Positive Predictive Value (PPV)*: the ratio of flows correctly classified as malicious to all items classified as malicious $TP/(TP+FP)$.
- *Negative Predictive Value (NPV)*: the ratio of items correctly classified as benign to all items classified as benign $TN/(TN+FN)$.

In addition to these, we report the *FP rate*, i.e., the significance level defined as $1 - Specificity$.

5.2 Setup of the Experiments

The hardware that we have used in our experiments are commercially available laptops, and a Graphics Processing Unit (GPU) server. The laptops act as either a user device to send traffic to the NIDS or a platform, on which Bro 2.5.3 is run. Note that portions of Bro 2.5.3 documents, which have been used in this work, is now available in Bro 2.5.5 documents [66]. Each laptop is equipped

Batch size	1
Learning rate	0.001
Number of the epochs	100
Number of the hidden layer	2
Word vector size	64
Sequence Length	1

Table 1. The LSTM configuration for our experiments.

```
signature dpd_ssh_client }
ip-proto == tcp
payload /^[sS][sS][hH]-[12]./
requires-reverse-signature dpd_ssh_server
enable "ssh"
tcp-state originator}
```

Table 2. Bro Signature for SSH Client protocol [61].

1	100					
2	58.6	100				
3	65.5	58.4	100			
1*	50.4	48.1	49.1	100		
2*	49	52.2	48.8	46.2	100	
3*	47.8	47.9	48.6	49.6	47.8	100
	1	2	3	1*	2*	3*

1	100					
2	60.7	100				
3	66.4	55.8	100			
1*	55	49.5	52.6	100		
2*	51.5	58.9	51	48.3	100	
3*	52	47.2	54.4	53.1	48.5	100
	1	2	3	1*	2*	3*

Table 3. The Smith-Waterman similarity percentages between the worms generated by the XOR encoder. RNN generated mutants of three worms are labeled with a star. In this experiment, the worms are given to the LSTM individually and one-by-one. The match value equals is set to 1 (left) and 5 (right). The example number marked with a star shows the corresponding example generated by the LSTM.

1	100					
2	59.9	100				
3	59.1	58	100			
1*	49.6	49	51.2	100		
2*	49.7	49.7	50.9	50	100	
3*	50.6	50.7	54.1	51.1	53.4	100
	1	2	3	1*	2*	3*

1	100					
2	58.2	100				
3	59.9	58.8	100			
1*	54.3	52.6	52.3	100		
2*	51.6	53.5	50.4	51.3	100	
3*	54.6	51.6	58.2	54.2	55	100
	1	2	3	1*	2*	3*

Table 4. Results of the experiment for Shikata Ga Nai encoder (the same setting as for the Table 3).

with an Intel Core i7 - 2.6 GHz (4 Cores 8 Threads) CPU and a 16 GB DDR3 RAM, and its operating system is Linux (Ubuntu 64 Bit).

Moreover, in line with other AI-related research studies, we use GPU clusters rather than CPU (Central Processing Units) computing resources. This is due to the performance and computational speed of GPU clusters in comparison to CPU ones in deep learning tasks [12, 47]. In addition to an Intel Core i7 - 3.4 GHz (6 cores 12 Threads) CPU, our GPU server composed of two Nvidia 1080-Ti GPU cards, and a 128 GB DDR4 RAM. On our server, we run Linux (Ubuntu 64 Bit) and Torch accounting for deep learning tasks. Torch supporting Lua is selected due to the availability of many models for RNNs and LSTMs and the ability to run the algorithms on the GPUs. Furthermore, Torch is one of the best tools for small-scale projects and fast prototyping.

Journal Pre-proof

Journal Pre-proof

1	100					
2	44.8	100				
3	47.3	64.2	100			
1*	52.5	46.6	48.7	100		
2*	48.7	92.9	56.5	49.4	100	
3*	46.8	49.9	57	49.7	52.3	100
	1	2	3	1*	2*	3*

1	100					
2	47	100				
3	48	64.2	100			
1*	58.2	47.2	48	100		
2*	41.2	96.3	59.8	38	100	
3*	49.6	55.3	64.1	51.6	56.8	100
	1	2	3	1*	2*	3*

Table 5. Results of the experiment for the ADMMutate encoder (the same setting as for the Table 3).

5.3 Experiment Design

Generating New Mutants of Polymorphic Worms

As discussed in Section 4.1, we take into account the polymorphic worms generated by employing the XOR, the Shikata Ga Nai, and the ADMMutate encoders. Besides that, in our view, each worm is a string of bytes and depending on no protocol or server information, in line with the content-based approach taken in [38]. In this regard, we generate variants of the OS X x64 Shell Bind TCP payload [63], with the different number of encoding iterations: without encoding, one iteration, etc. Worms mutants are then saved as a text file. Before feeding the text files into the LSTM, some preprocessing is needed to change the text into an understandable format for the RNN. Preprocessing here refers to the process of converting a text to the vectors of numbers to be understandable for the neural network. There are several ways to convert a text to a vector, e.g., using word to vector algorithms or decoding the text into the Unicode system. Selecting the appropriate model depends on the context of the data. For instance, using a word to vector models is more beneficial, when there exists a text written in a natural language. Here in the preprocessing step, we use the UTF-8 encoder for the purpose of converting a text to a vector.

Table 1 presents more details about the configuration of the LSTMs used in our experiments. For each input given to the LSTM (e.g., a worm), the network is configured individually based on the number of characters composing the worm. For instance, when a worm file contains 100 characters, the number of nodes in each hidden layer is set to 100. Moreover, we feed each and every worm only once.

Generating Signatures of Bro

Bro analyzer supports several different Internet protocols. This means that some of the attacks launched via these protocols have been analyzed and their signatures are extracted and added into Bro signature set. Seventeen of these signatures related to the attacks via HTTP, SMTP, POP3, FTP, etc. are selected to feed into the LSTM. As an example of these signatures, Table 2 presents the signature for SSH protocol. A Bro signature has mainly two parts, namely conditions and actions. Conditions are defined for the header and the content of the packet. In the content conditions, matching occurs against the payload part of the packet. The second part, an action, is the response given by Bro system, when a signature matching happens, e.g., rising individual events or enabling a special analyzer for the matched protocols or data [61].

For the experiment conducted to generate synthetic signatures, the LSTM is configured using the parameters provided in Table 1. The number of nodes in each hidden layer is twice the length of the signature fed into the LSTM as an input.

Post-processing the outputs of the LSTM

The RNN generates an output according to its pre-trained model for each signature. These outputs have the same number of characters as the corresponding inputs. Each output should be recognized in Bro system as a regular expression term. Therefore, some editing is needed such as

1	100									
2	65.3	100								
3	65.3	62.9	100							
4	67.6	62.9	72.2	100						
5	62.9	67.6	65.3	65.3	100					
1*	48.7	49.8	49.8	49	51.7	100				
2*	49.4	49.4	50.3	51.3	50.5	50.2	100			
3*	49.9	50.3	49.7	49.4	49.8	50	49.4	100		
4*	47.1	48.1	47.7	47.7	48.4	46.2	49.6	49.7	100	
5*	49.6	50.3	50.7	50.7	49.3	49.7	48.9	50.1	51.2	100
	1	2	3	4	5	1*	2*	3*	4*	5*

1	100									
2	66.5	100								
3	66.5	64.6	100							
4	68.7	64	72.9	100						
5	64.8	68.4	66.3	66.3	100					
1*	54.1	55.5	55.3	54.2	53.4	100				
2*	54.8	54.5	55.6	55.6	55	53.9	100			
3*	55.5	54.5	56	56.8	55.5	54.2	55.2	100		
4*	52.7	52.7	53.4	53.9	52.9	53.3	54.3	53.3	100	
5*	54.2	54.8	55.1	55.3	53.9	54.8	54.2	54.6	56.1	100
	1	2	3	4	5	1*	2*	3*	4*	5*

Table 6. The Smith-Waterman similarity percentages between the worms generated by the XOR encoder, with the same number of iteration. In this experiment, we feed a set of five worms, all together, into the LSTM. The match value equals is set to 1 (left) and 5 (right).

1	100									
2	67.1	100								
3	61.7	58	100							
4	58.8	57.5	60.8	100						
5	62.6	57.7	59.8	59.4	100					
1*	47.9	48.1	47.7	47.9	48.3	100				
2*	48.1	47	47.6	46.6	47.5	50.2	100			
3*	53	52	52.7	52	52.3	50.1	46.7	100		
4*	48.7	48.2	48.5	47.3	47.8	48.4	48.9	48.5	100	
5*	47.4	47.1	47.1	46.9	46.7	46.5	47.1	45.9	49.1	100
	1	2	3	4	5	1*	2*	3*	4*	5*

1	100									
2	68.2	100								
3	62.5	59.8	100							
4	60.2	59.8	61.9	100						
5	64	59.7	61.5	61	100					
1*	53.5	53.9	53.7	53.7	53.5	100				
2*	52.9	53.7	53.8	53.3	56	55.3	100			
3*	56	55.9	57.3	55.4	54.6	53.9	53.6	100		
4*	55.5	54.9	55.1	54.6	54	54.1	54.5	53.9	100	
5*	53.8	54.1	52.1	53.3	53.4	53.4	52.3	53.5	53	100
	1	2	3	4	5	1*	2*	3*	4*	5*

Table 7. Results of the experiment for the Shikata Ga Nai encoder (the same setting as for the Table 6).

1	100									
2	86.1	100								
3	90.8	83.8	100							
4	86.1	83.8	83.8	100						
5	88.5	88.5	83.8	86.1	100					
1*	58	53	65.2	53.4	53.2	100				
2*	52.6	47.8	61.2	48	48	68.6	100			
3*	56.6	54.4	65.2	50.8	53.1	63.5	65	100		
4*	54.4	47.9	63.5	48.2	48.1	68.6	87.3	70.1	100	
5*	54	66.5	52.2	52.3	56.3	57.8	55	52.8	55	100
	1	2	3	4	5	1*	2*	3*	4*	5*

1	100									
2	86.1	100								
3	90.8	83.8	100							
4	87.6	86.6	85.2	100						
5	89.8	88.5	83.8	87.5	100					
1*	63.4	57.9	70.6	58.5	58.3	100				
2*	56.8	52	64.4	53.2	52.4	69.6	100			
3*	62.1	58.2	70.3	58.1	57	68.2	71.1	100		
4*	58.3	51.6	67.5	53.4	51.9	72.2	90.8	73.8	100	
5*	58.6	71.7	56.6	59.1	60.9	62.7	59.4	58.4	59.2	100
	1	2	3	4	5	1*	2*	3*	4*	5*

Table 8. Results of the experiment for the ADMMutate encoder (the same setting as for the Table 6).

closing the open brackets or removing the slashes appearing in the middle of the output string. Theses minor modification can be easily done by an operator, or even, a scrip can be developed to perform this task. In our experiments, we have followed both of these procedures: open brackets are closed by the operator, and our simple, in-house-developed script is used to remove the misplaced slashes.

Journal Pre-proof

Journal Pre-proof

6 RESULTS AND DISCUSSION

6.1 Similarity between original and RNN-generated Worms

As discussed in Section 5.1, due to the drawbacks of the Levenshtein approach for comparing the distance between strings, we mainly focus on the results achieved by using the Smith-Waterman metric. Here, for the sake of completeness, we briefly present the Levenshtein similarity percentages (see [8]) computed for our worms.

We choose three worms generated as described in 5.3 and compute the Levenshtein similarity percentages between the original worm and their corresponding RNN-generated ones. Note that for the XOR engine, the numbers of characters in the worms are 606, 777, and 952. This number is 606, 724, and 842 for Shikata Ga Nai engine and 606, 1946, and 1946 for ADMMutate engine. For worms encoded by the XOR engine, the average of the Levenshtein similarity percentage is 48.9%, whereas for the Shikata Ga Nai and ADMMutate engines, it is 49.43% and 31.46%, respectively.

Table 3 to Table 5 show the (normalized) Smith-Waterman similarity percentages between mutants of the worms themselves and the RNN generated mutants, marked with a star. Note that since the normalized values (over the length of the matching substrings) are reported, the length of the worms can be discarded. As expected, the Smith-Waterman Algorithm shows a sufficiently high similarity ratio for the worms here. Furthermore, when increasing the match value from one to five, the similarity percentages between worms remain in the same order of magnitude. Remarkable is that the similarity percentage, and accordingly the distances, between the mutants of the worms generated by the RNN (synthetic mutants) themselves, and between synthetic mutants and original ones are similar to the distances between an original mutant and other original mutants of a worm. This is an interesting and important result since we consider the normalized similarity percentage. Therefore, this result demonstrates that the LSTM not only can learn and extract the grammars underlying the given worms, but also can generate substrings with the same similarity percentage.

It can be thought that if instead of feeding the worms one-by-one, a set of worms is given to the LSTM, the similarity percentages can be different. We examine this by choosing a set of 5 worms, each with a different number of encoding iterations, namely, one to five iterations. These worms are concatenated together into a single text file that is given to the LSTM. The results achieved for the XOR, Shikata Ga Nai, and ADMMutate encoders are shown in Table 6 to Table 8. In this case, an improvement in the similarity percentages (on average) can be observed. This can be explained by the fact that the LSTM network is definitely larger than the network used for the other experiment (Table 3 to Table 5) due to the larger number of characters in the text file given to that [3]. Additionally, by giving more examples to the LSTM, it can improve its prediction by observing the examples with the same grammar. Nonetheless, when having only one example of a worm (i.e., one mutant of that), it is still possible to obtain a close-enough unseen mutant, as presented in our previous experiments (Table 3 to Table 5).

6.2 Similarity between original and RNN-generated Bro Signatures

Following the same procedure described in Section 6.1, we feed Bro signature individually into the LSTM. Afterwards, we compute the Levenshtein and Smith-Waterman similarity percentages between Bro signature and their associated signatures generated by the LSTM. In this scenario, alike the previous scenario explained in Section 6.1, the similarity percentages between the LSTM-generated signatures and the original ones demonstrate that the LSTM could find some patterns, and based on them generate a similar signature with low distance from Bro signature, see Table 9. It

[3]Recall that in our experiments the number of nodes in each hidden layer equals the number of characters in the given example.

Journal Pre-proof

Protocol	# Characters in signature	Levenshtein Similarity [%] [67]	Smith-Waterman Similarity [%]
SOCKS	214	25.23	81.3
DNP3	38	84.21	47.6
RFB	30	80	35.9
KRB	314	16.24	89.7
FTP	100	49	55.7
Tunnels	142	54.93	83.9
DCE/RPC	36	75	29.1
SMTP	108	54.63	65.5
SIP	173	46.82	77.5
RDP	98	58.04	87.8
SSH	62	58.06	48.1
SSL	357	40.06	68.8
IRC	359	42.62	57.4
XMPP	72	80.56	46.3
DHCP	30	80	26.7
HTTP	406	78.82	27.7
POP3	141	59.57	59.3

Table 9. The Levenshtein and Smith-Waterman similarity percentages between Bro signatures and the LSTM generated ones.

is worth noting here that the synthetic signatures should be solely *sufficiently* close to the original ones. In other words, in order to provide an NIDS (in our case Bro) with effective signatures, we should feed new signatures that are similar to the known signatures, but also represent variants of the respective attack.

Comparing the Smith-Waterman and the Levenshtein similarity percentages shown in Table 9, as expected, these percentages are not always consistent. This is due to the fact that the Levenshtein similarity percentage can reflect the results of the global comparison between two strings. On the contrary, the Smith-Waterman similarity percentage present the local, sub-string-based comparison between two strings. Although at first glance, it seems surprising that in some cases the Levenshtein similarity percentages are higher than the Smith-Waterman ones, the length of the examples (i.e., the number of the characters in the signature) can explain this. For shorter signatures, and smaller LSTM networks equivalently, the number of characters given to the network is not sufficient enough for the LSTM to extract and learn the grammar underlying the strings. And, therefore, the LSTM may repeat the same characters. In contrast to this for longer signatures, the LSTM extract the grammar and generate new substrings that locally match the substrings of the given signature. Hence, we suggest that for different signatures with various lengths, one should compute both the Smith-Waterman and the Levenshtein similarity percentages.

6.3 Enhancing the performance of Bro

Here we aim to evaluate the performance (regarding the metrics introduced in 5.1) of Bro, whose set of signatures is expanded by adding the synthetic signatures generated by the LSTMs. This section covers two series of experiment: the first set is conducted to examine how the performance of Bro can be improved in terms of detecting worms and their mutants. Towards the same goal, the second set of experiments is performed to observe the improvement in a general case, i.e., detecting a broad range of intrusions. For both of these sets, at the first stage, it is necessary to collect appropriate samples of network traffic flows. In our experiments, the benign data pool contains 408588 packets (330 MB in total), available in Wireshark repository [35]. As shown in [38, 59], the size of the data

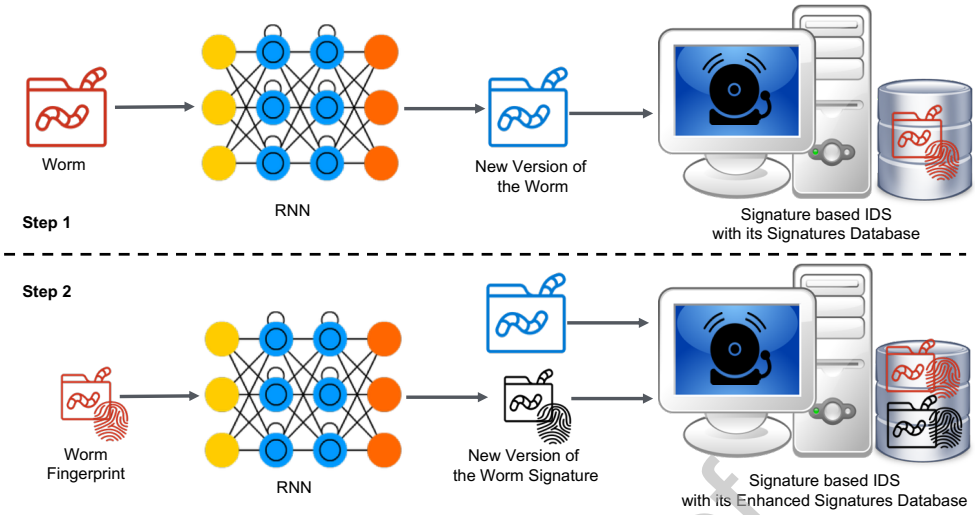


Fig. 7. In order to verify to what extent the performance of Bro can be improved through our approach, we first conduct Experiment 1. In this experiment, as the first step, we generate the mutants of a worm, which are used to evaluate the performance of Bro. The second step in this process is to generate a variant of the signature associated with the worm and add that to the database of Bro.

pool of the benign traffic may not significantly influence the performance of an NIDS under test. Hence, we stick to the number of packets mentioned above.

Experiment 1: As explained above, this experiment is designed to verify if integrating new, synthetic signatures into the database of Bro can improve its performance. For this purpose, we select the worm Code-Red that can be tracked by Bro. This worm exploits the buffer-overflow vulnerability to gain full system level access to its victim [42]. The signature associated with this worm is further available and can be used to generate a new synthetic signature for Code-Red. For this experiment, RNN is employed in two different steps and settings (see Figure 7):

- Step 1: it is used to generate mutants of the worm from given examples of that. This step attempts to reflect how a new mutant can be generated by an attacker. Note that the results of our experiments presented in Section 6.1 enable us to ensure that the mutants generated in Step 1 are sufficiently different from the original Code-Red worm.
- Step 2: given the *signatures* of the worm available in the Bro database, a new version of these signatures is generated by the RNN. Note that this step is independent of Step 1.

After taking these steps, it is examined if the performance of Bro can be improved in terms of detecting new worms generated in Step 1, when the database of that contains the signature generated in Step 2. In this regard, at the first stage, we apply Bro detector against solely the benign traffic to calculate the FP that equals 2.2%. Afterwards, the worms are given to Bro, whose database is extended by adding the synthetic signature. As a result, in this case, all the mutants of the Code-Red are detected ($FN = 0$), whereas, without the synthetic signature, $FN = 16.67\%$. Moreover, we do not observe any change in the FP.

Experiment 2: For this set of experiments, we generate a pool of the malicious data by using not only the signatures embedded in Bro, but also a *set* of synthetic signatures. It is crucial to note that the set of synthetic signatures used to generate the malicious pool and the set of signatures integrated in the database of Bro are disjoint. In other words, we use different variant of a signature to generate the pool. Following a procedure commonly employed in the literature (see, e.g., [59, 60]),

Journal Pre-proof

both of these sets of the signatures are given to a module responsible to create malicious data from the signatures, by using, for instance, an inverse regular expression generator. The result is then mapped to hosts that are not involved in the benign traffic flows. As can be understood, we follow an overlay-like procedure to generate our pools and combine them. Most importantly, as advised in [5], we do not map all the malicious traffic to one host, but randomly we choose a host from a set of hosts, absent in the benign traces. In this way, we leverage the advantages of the overlay method, namely providing a better understanding of a real-world scenario and ground truth, and accordingly a fair method to estimate the performance of an NIDS. Moreover, since it is possible to map more than one malicious traffic to a host, there are cases for "concurrent infections" caused by the same host [5, 54]. Employing our procedure, we generate 3061 malicious traffic packet (3.2 MB in total).

To evaluate the performance of Bro in terms of the FP and the FN, four different scenarios are taken into account. First, we apply Bro detector against solely the benign traffic to calculate the FP. First, similar to what we have already mentioned about Experiment 1, we test Bro in terms of FP and obtain $FP = 2.2\%$. The second scenario is similar to the first one, but the FN of Bro is tested by giving a mixed pool of the benign and malicious traffic flows. In this case $FN = 4.15\%$.

In the third and fourth scenarios, we extend the set of Bro signatures by adding our LSTM-generated signatures. In the third case, we calculate the FP for the extended, enhanced Bro. The result ($FP = 2.7\%$) demonstrates that adding new signatures does not considerably impair the performance of Bro in terms of the FP. In the fourth scenario, we calculate the FN of Bro, when our synthetic signatures are activated as well. In this case, our synthetic signatures are configured as Bro signatures in a signature script to match the new activities or malware. We obtain $FN = 3.12\%$, which shows that the performance of Bro is indeed improved. Note that this result is achieved by adding solely a small set of new signatures, namely seventeen signatures. In addition to the reduction in the FN the number of the alarms raised by Bro can further illustrate the potential of our approach. In this regard, note that in the second scenario (without the synthetic signatures) the number of alarms is 2662, whereas in the last scenario, where the synthetic signatures are also activated, the number of alarms is 2761.

Furthermore, the statistical measures introduced in Section 5.1 can be used to compare the performance of off-the-shelf (values inside the brackets) and enhanced versions of Bro as follows. Sensitivity is 96.65% (95.44%), and NPV is 96.86% (95.89%), where both of them show improvement as FN is reduced. PPV is 97.02% (97.46%), and Specificity is 97.21% (97.71%), where the degradation is related to a slight increase in FP. Similarly, the FP rate is increased from 2.28% to 2.78%. It is worth noting that the significantly high sensitivity and Specificity observed for the enhanced Bro make our approach a promising solution for NIDS.

Summary and Discussion: This section covers the details and results of three main experiments. First, we experimentally verify that the (LSTM) RNN can learn and extract the grammars underlying the given polymorphic worms, and therefore, it can be employed to generate new, unseen mutant of sophisticated polymorphic worms, with encoded exploits. Besides, our experimental results demonstrate that the RNN is capable of generating synthetic signatures that are sufficiently close to the original ones, and simultaneously different from the original signatures. These signatures represent variants of the attacks that may not be detected by an NIDS, i.e., Bro in our experiments. In order to show to what extent these synthetic signatures can help us to improve the performance of the NIDS, we conduct additional experiments, whose results are summarized in Table 10. For this purpose, we provide a framework for generating a set of malicious data that involves malicious traffic flows generated from a *set* of synthetic worms and signatures. We stress that the signatures added to the database of Bro are entirely different from the ones used to generate malicious data.

Journal Pre-proof

Experiment	NIDS	FP [%]	FN [%]
Experiment 1 (Detection of Worm Mutants)	Off-the-shelf Bro	2.2	16.67
	Enhanced Bro	2.2	0
Experiment 2 (General Case)	Off-the-shelf Bro	2.2	4.15
	Enhanced Bro	2.7	3.12

Table 10. Performance of the off-the-shelf Bro and the enhance one, evaluated in two main scenarios: detection of worms and the general case, i.e., several classes of malware.

Moreover, as the malicious traffic flows are crafted by using an inverse regular expression generator, they are further randomized.

Finally, we put emphasis on key characteristics of RNNs, making them suitable for our purpose: RNNs can learn and extract unknown grammars and generate unseen sequences that belong to the respective grammar. As an example of other neural networks that may not help us with this issue, Convolutional Neural Networks (CNNs) can be mentioned. Although CNNs offer great potential in various applications, RNNs outperforms them in applications similar to ours, while they exhibit temporal characteristic fulfilling the conditions for processing sequences of data.

7 CONCLUSION AND REMARKS

This paper provides a framework for applying deep learning techniques in the area of cyber security. While zero-day attacks are vastly propagated in the network, our approach attempts to improve the ability of NIDS systems to defend against them by (1) extending their signature databases, and (2) generating a more realistic and close to the real-world ground truth to test an NIDS.

More specifically, we take advantage of the immense power of the recurrent neural networks (RNNs) in distinguishing complex patterns in a text and generating similar ones. As an example of a possible application of these networks in intrusion detection, an LSTM is used to generate several mutants of polymorphic worms. These synthetic worms are evaluated by applying two powerful similarity metrics, namely the Levenshtein the Smith-Waterman similarity percentages. Furthermore, as another example of how RNNs can be beneficial to intrusion detection, we demonstrate that an LSTM can be used to generate synthetic signatures to enhance the detection rate of an NIDS. Nevertheless, our approach can be seen as the first step in this regard. More concretely, to further improve such systems, RNNIDS could benefit from methods devised to minimize false positives (see, e.g., [30]), which we leave as future work.

Last but not least, we stress that although the applications of RNNs have been widely studied and accepted in the machine learning-related literature, their capability to provide help with intrusion detection should be considered more closely. This paper paves the way for a more detailed exploration of this capacity in cyber security.

8 ACKNOWLEDGEMENTS

This work has been partially supported by the European Union's Horizon 2020 Research and Innovation Program under grant agreement No. 700176 (SISSDEN, Project ID: 700176, funded under: H2020-EU.3.7. - Secure societies - Protecting freedom and security of Europe and its citizens). Moreover, the authors would like to acknowledge the support of the Bundesministerium für Bildung und Forschung under grant 01S180251 and BIFOLD agility project.

REFERENCES

[1] ADMmutate 0.8.4. 2016. https://github.com/K2/ADMMutate/blob/master/README [Accessed: 15-May-2019]. (2016).

Journal Pre-proof

[2] Jason Andress. 2014. *The basics of information security: understanding the fundamentals of InfoSec in theory and practice.* Syngress.

[3] Dana Angluin and Carl H Smith. 1983. Inductive Inference: Theory and Methods. *ACM Computing Surveys (CSUR)* 15, 3 (1983), 237–269.

[4] Abdullah N Arslan, Ömer Eğecioğlu, and Pavel A Pevzner. 2001. A New Approach to Sequence Comparison: Normalized Sequence Alignment. *Bioinformatics* 17, 4 (2001), 327–337.

[5] Adam J Aviv and Andreas Haeberlen. 2011. Challenges in Experimenting with Botnet Detection Systems. In *Proc. of the 4th Conf. on Cyber Security Experimentation and Test.* USENIX Association, 6–6.

[6] Ariel Bar, Bracha Shapira, Lior Rokach, and Moshe Unger. 2016. Scalable Attack Propagation Model and Algorithms for Honeypot Systems. In *Big Data (Big Data), 2016 IEEE Intl. Conf. on.* IEEE, 1130–1135.

[7] Elaheh Biglar Beigi, Hossein Hadian Jazi, Natalia Stakhanova, and Ali A Ghorbani. 2014. Towards Effective Feature Selection in Machine Learning-based Botnet Detection Approaches. In *Comm. and Network Security (CNS), 2014 IEEE Conference on.* IEEE, 247–255.

[8] Borenstein, Elhanan. 2012. Sequence Comparison: Significance of Similarity Scores. http://elbo.gs.washington.edu/courses/GS_373_12_sp/slides/6A-significance_scores.pdf [Accessed: 25-Apr-2018]. (2012).

[9] Anna L Buczak and Erhan Guven. 2015. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications surveys & tutorials* 18, 2 (2015), 1153–1176.

[10] Silvio Cesare and Yang Xiang. 2011. Malware Variant Detection Using Similarity Search Over Sets of Control Flow Graphs. In *Trust, Security and Privacy in Computing and Comm. (TrustCom), 2011 IEEE 10th Intl. Conf. on.* IEEE, 181–189.

[11] Mikhail V Chester and Braden R Allenby. 2020. Perspective: The Cyber Frontier and Infrastructure. *IEEE Access* 8 (2020), 28301–28310.

[12] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep Learning with COTS HPC Systems. In *Intl. Conf. on Machine Learning.* 1337–1345.

[13] Dorothy Denning and Peter G Neumann. 1985. *Requirements and Model for IDES-a Real-time Intrusion-detection Expert System.* SRI Intrl.

[14] Dorothy E Denning. 1987. An Intrusion-detection Model. *IEEE Trans. on Software Engineering* 2 (1987), 222–232.

[15] Cyber Defense Exercise. 1999. CDX Dataset. https://www.usma.edu/crc/sitepages/datasets.aspx [Accessed: 15-May-2019]. (1999).

[16] Ryan Farley and Xinyuan Wang. 2014. Codext: Automatic Extraction of Obfuscated Attack Code from Memory Dump. In *Intl. Conf. on Information Security.* Springer, 502–514.

[17] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. 2009. Anomaly-based Network Intrusion Detection: Techniques, Systems and Challenges. *Computers & Security* 28, 1-2 (2009), 18–28.

[18] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to Forget: Continual Prediction with LSTM. (1999).

[19] C Lee Giles, Clifford B Miller, Dong Chen, Hsing-Hen Chen, Guo-Zheng Sun, and Yee-Chun Lee. 1992. Learning and Extracting Finite State Automata With Second-Order Recurrent Neural Networks. *Neural Computation* 4, 3 (1992), 393–405.

[20] C Lee Giles, Clifford B Miller, Dong Chen, Guo-Zheng Sun, Hsing-Hen Chen, and Yee-Chun Lee. 1992. Extracting and Learning an Unknown Grammar with Recurrent Neural Networks. In *Advances in Neural Information Processing Systems.* 317–324.

[21] C Lee Giles, Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and Dong Chen. 1990. Higher Order Recurrent Networks and Grammatical Inference. In *Advances in Neural Information Processing Systems.* 380–387.

[22] E. Mark Gold. 1978. Complexity of Automaton Identification from Given Data. *Information and control* 37, 3 (1978), 302–320.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

[24] Alex Graves. 2012. Supervised Sequence Labelling. In *Supervised Sequence Labelling with Recurrent Neural Networks.* Springer, 5–13.

[25] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. In *Acoustics, Speech and Signal Processing (icassp), 2013 ieee intl. conf. on.* IEEE, 6645–6649.

[26] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. 2017. Improved Training of Wasserstein GANs. In *Advances in neural information processing systems.* 5767–5777.

[27] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9, 8 (1997), 1735–1780.

[28] Hannes Holm. 2014. Signature Based Intrusion Detection for Zero-day Attacks: (Not) A Closed Chapter?. In *System Sciences (HICSS), 2014 47th Hawaii Intrl. Conf. on.* IEEE, 4895–4904.

[29] John E Hopcroft. 2008. *Introduction to Automata Theory, Languages, and Computation.* Pearson Education India.

Journal Pre-proof

Journal Pre-proof

[30] Neminath Hubballi and Vinoth Suryanarayanan. 2014. False alarm minimization techniques in signature-based intrusion detection systems: A survey. *Computer Communications* 49 (2014), 1–17.

[31] Christian Kreibich and Jon Crowcroft. 2004. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Comm. Rev.* 34, 1 (2004), 51–56.

[32] Tammo Krueger, Nicole Krämer, and Konrad Rieck. 2010. ASAP: Automatic Semantics-aware Analysis of Network Payloads. In *Intl. Workshop on Privacy and Security Issues in Data Mining and Machine Learning*. Springer, 50–63.

[33] MIT Lincoln Laboratory. 1999. DARPA Intrusion Detection Evaluation Data Sets,. https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-data-set [Accessed: 15-May-2019]. (1999).

[34] Philip Laird and Ronald Saul. 1994. Discrete Sequence Prediction and Its Applications. *Machine learning* 15, 1 (1994), 43–68.

[35] Laura Chappell. 2017. Wireshark 101: Essential Skills for Network Analysis (Book Supplements). https://www.chappell-university.com/ [Accessed: 15-May-2019]. (2017).

[36] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436.

[37] Wenke Lee, Salvatore J Stolfo, et al. 1998. Data Mining Approaches for Intrusion Detection. In *USENIX Security Symp.* San Antonio, TX, 79–93.

[38] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. 2006. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *Security and Privacy, 2006 IEEE Symp. on*. IEEE, 15–pp.

[39] Howard F Lipson. 2002. *Tracking and Tracing Cyber-Attacks: Technical Challenges and Global Policy Issues.* Technical Report. Carnegie Melon Univ., Eng. Inst.

[40] Georgios Loukas and Gülay Öke. 2010. Protection Against Denial of Service Attacks: A Survey. *The Computer Journal* 53, 7 (2010), 1020–1037.

[41] Jimmy McGibney. 2004. Intrusion Detection Systems & Honeypots. *Waterford Inst. of Technology, Ireland, Prezentacija sa konf. INET/IGC, Barcelona* (2004).

[42] David Moore, Colleen Shannon, et al. 2002. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet Measurement*. ACM, 273–284.

[43] Arvind Narayanan and Vitaly Shmatikov. 2008. Robust De-anonymization of Large Sparse Datasets. In *Security and Privacy, 2008. SP 2008. IEEE Symp. on*. IEEE, 111–125.

[44] James Newsome, Brad Karp, and Dawn Song. 2005. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Security and privacy, 2005 IEEE symp. on*. IEEE, 226–241.

[45] Christian W Omlin and C Lee Giles. 1996. Constructing Deterministic Finite-state Automata in Recurrent Neural Networks. *Journal of the ACM (JACM)* 43, 6 (1996), 937–972.

[46] Ofir Press, Amir Bar, Ben Bogin, Jonathan Berant, and Lior Wolf. 2017. Language Generation with Recurrent Generative Adversarial Networks without Pre-training. *arXiv preprint arXiv:1706.01399* (2017).

[47] Rajat Raina, Anand Madhavan, and Andrew Y Ng. 2009. Large-scale Deep Unsupervised Learning Using Graphics Processors. In *Proc. of the 26th Annual Intl. Conf. on Machine Learning*. ACM, 873–880.

[48] Martin Roesch. 1998. Snort - Network Intrusion Detection and Prevention System. https://www.snort.org/ [accessed 16 April 2018]. (1998).

[49] Benjamin Sangster, TJ O'Connor, Thomas Cook, Robert Fanelli, Erik Dean, Christopher Morrell, and Gregory J Conti. 2009. Toward Instrumenting Network Warfare Competitions to Generate Labeled Datasets. In *Cyber Security Experimentation and Test (CSET)*. USENIX.

[50] Jürgen Schmidhuber. 1993. *Netzwerkarchitekturen, Zielfunktionen und Kettenregel (Network Architectures, Objective Functions, and Chain Rule).* Ph.D. Dissertation. Technische Universität München.

[51] School of Information and Computer Science, Univ. of California, Irvine, KDD Cup 1999 Data. 1999. KDD99 Dataset. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html [Accessed: 15-May-2019]. (1999).

[52] Jungsuk Song, Hiroki Takakura, Yasuo Okabe, Masashi Eto, Daisuke Inoue, and Koji Nakao. 2011. Statistical Analysis of Honeypot Data and Building of Kyoto 2006+ Dataset for NIDS Evaluation. In *Proc. of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. ACM, 29–36.

[53] Yingbo Song, Michael E Locasto, Angelos Stavrou, Angelos D Keromytis, and Salvatore J Stolfo. 2007. On the Infeasibility of Modeling Polymorphic Shellcode. In *Proc. of the 14th ACM Conf. on Computer and Comm. Security*. ACM, 541–551.

[54] Brett Stone-Gross, Thorsten Holz, Gianluca Stringhini, and Giovanni Vigna. 2011. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns. *LEET* 11 (2011), 4–4.

[55] Ron Sun and C Lee Giles. 2001. Sequence Learning: From Recognition and Prediction to Sequential Decision Making. *IEEE Intelligent Systems* 16, 4 (2001), 67–70.

Journal Pre-proof

Journal Pre-proof

[56] Ilya Sutskever, James Martens, and Geoffrey E Hinton. 2011. Generating Text with Recurrent Neural Networks. In *Proc. of the 28th Intl. Conf. on Machine Learning (ICML-11)*. 1017–1024.
[57] Symantec. 2017. ISTR Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf [accessed 16 April 2018]. (2017).
[58] Symantec. 2018. ISTR Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf [accessed 16 April 2018]. (2018).
[59] Paweł Szynkiewicz and Adam Kozakiewicz. 2017. Design and Evaluation of a System for Network Threat Signatures Generation. *Journal of Computational Science* 22 (2017), 187–197.
[60] Yong Tang, Bin Xiao, and Xicheng Lu. 2009. Using a Bioinformatics Approach to Generate Accurate Exploit-based Signatures for Polymorphic Worms. *Computers & Security* 28, 8 (2009), 827–842.
[61] Bro Project Team. 2018. Signature Framework–Bro 2.5.3 Documentation. https://old.zeek.org/manual/2.5.5/index.html [Accessed: 21-Oct-2020]. (2018).
[62] Rapid7 Team. 2018. Metasploit–The World's Most Used Penetration Testing Framework. https://metasploit.com/ [Accessed: 23-Apr-2018]. (2018).
[63] Rapid7 Team. 2018. Metasploit, OS X x64 Shell Bind TCP payload. https://github.com/rapid7/metasploit-framework/blob/master/modules/payloads/singles/osx/x64/shell_bind_tcp.rb [Accessed: 23-Apr-2018]. (2018).
[64] Rapid7 Team. 2018. Metasploit, Shikata Encoder. https://github.com/rapid7/metasploit-framework/blob/master/modules/encoders/x86/shikata_ga_nai.rb [Accessed: 23-Apr-2018]. (2018).
[65] Rapid7 Team. 2018. Metasploit, XOR Encoder. https://github.com/rapid7/metasploit-framework/blob/master/modules/encoders/x64/xor.rb [Accessed: 23-Apr-2018]. (2018).
[66] The Bro Project. 2018. Bro Manual. https://old.zeek.org/manual/2.5.5/index.html [Accessed: 21-Oct-2020]. (2018).
[67] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. 2009. Intrusion Detection by Machine Learning: A Review. *Expert Systems with Applications* 36, 10 (2009), 11994–12000.
[68] Giovanni Vigna, William Robertson, and Davide Balzarotti. 2004. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proc. of the 11th ACM Conf. on Computer and Comm. Security*. ACM, 21–30.
[69] Kashi Venkatesh Vishwanath and Amin Vahdat. 2009. Swing: Realistic and Responsive Network Traffic Generation. *IEEE/ACM Trans. on Networking (TON)* 17, 3 (2009), 712–725.
[70] Ke Wang and Salvatore J Stolfo. 2004. Anomalous Payload-based Network Intrusion Detection. In *Intrl. Workshop on Recent Advances in Intrusion Detection*. Springer, 203–222.
[71] Zeek Project. 2018. Zeek: An Open Source Network Security Monitoring Tool. https://zeek.org [Accessed: 21-Oct-2020]. (2018).
[72] Richard Zuech, Taghi M Khoshgoftaar, Naeem Seliya, Maryam M Najafabadi, and Clifford Kemp. 2015. A New Intrusion Detection Benchmarking System.. In *FLAIRS Conference*. 252–256.

Journal Pre-proof

Journal Pre-proof

Declaration of interests

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

We declare that due to the involvement in the same project, we have conflict of interest with the following scholars.

1. Paweł Szynkiewicz
2. Adam Kozakiewicz
3. Konrad Rieck

Journal Pre-proof