Proceedings of the 54th Hawaii International Conference on System Sciences | 2021

HoneyCode: Automating Deceptive Software Repositories with Deep Generative Models

David D. Nguyen
UNSW Sydney, Data61
Cyber Security CRC
d.d.nguyen@unsw.edu.au

David Liebowitz
Penten
UNSW Sydney
david.liebowitz@penten.com

Surya Nepal
Data61
Cyber Security CRC
surya.nepal@data61.csiro.au

Salil S. Kanhere
UNSW Sydney
Cyber Security CRC
salil.kanhere@unsw.edu.au

Abstract

We propose HoneyCode, an architecture for the generation of synthetic software repositories for cyber deception. The synthetic repositories have the characteristics of real software, including language features, file names and extensions, but contain no real intellectual property. Fake repositories can be used as a honeypot or form part of a deceptive environment. Existing approaches to software repository generation lack scalability due to reliance on hand-crafted structures for specific languages. Our approach is language agnostic and learns the underlying representations of repository structures, filenames and file content through a novel Tree Recurrent Network (TRN) and two recurrent networks respectively. Each stage of the sequential generation process utilizes features from prior steps, which increases the honey repository's authenticity and consistency. Experiments show TRN generates tree samples that reduce degree mean maximal distance (MMD) by 90-92% and depth MMD by 75-86% to a held out test data set in comparison to recent deep graph generators and a baseline random tree generator. In addition, our RNN models generate convincing filenames with authentic syntax and realistic file content.

1. Introduction

Cyber attacks are growing by 10% per annum and costing on average $13 million per attack [1], posing a significant threat to corporate and government organisations. Deception is an increasingly important aspect of defending against these attacks, providing a number of benefits [2]. The most apparent of these is discovery of a breach through the alert generated when an attacker interacts with a honeypot [3]. However, deception can also provide insight into the intent and tactics, techniques and procedures (TTPs) of an attacker. These benefits come from high interaction deceptions, where the attacker is allowed to engage with the deception. The key to successful high interaction deceptions is realism, which provides greater scope for the adversary to express their intentions through their choice of actions and exercise their TTPs. The interactions must take place in an environment that, while realistic, exposes little or nothing of the defending organisation's real data or intellectual property.

This paper proposes HoneyCode, a machine learning system that creates a deceptive software repository that has the characteristics of a real repository. It has similar folder structure, file names and files with recognisable language features but, critically, does not contain any actual compilable source code or intellectual property. Such a repository, trained on an organisation's own or open source code, can serve as a honeypot, protecting software which may be fundamental to the company's core intellectual property [4, 5]. In addition, a repository on a network or a workstation is immensely valuable as an element of a larger deception system - the "pocket litter" essential to sustaining the illusion around a honeypot or sandbox[6].

Content based deception was famously successful when Clifford Stoll uncovered the location of an intruder on the Lawrence Berkeley National Laboratory network in the 1980s[7] [1] by creating an entire fictitious department, complete with emails, reports and forms to fill in. Stoll engaged the attention of the intruder and kept him online for long enough to allow international law enforcement efforts to pinpoint his location and arrest him. An intrusion detection system based on honeyfiles - fake document honeypots - was described by Yuill[9]. Subsequent efforts have developed methods to automate the creation of honeyfiles[10, 11] to allow for large scale deployment in document repositories.

Creating fake software repositories extends this work in two senses: the range of document types is increased to include plausible source code, and the entire structure of a repository file system and file names must be synthesized. The structure of a repository depends on its purpose and the conventions of the dominant programming language. A repository is also likely to contain a diverse range of files that use different syntactic patterns based on their

[1] described at length in the very readable *The Cuckoo's Egg* [8]

URI: https://hdl.handle.net/10125/71456
978-0-9981331-4-0

languages.

To address these challenges, we introduce a triple neural network architecture, HoneyCode, based on learning and generating representations of these components. We introduce a novel Tree Recurrent Network (TRN) to synthesize directed rooted trees to represent the folder and file structures of a new repository, a name generator network that generates files and folder names for each node, and a file content generator network that synthesises source code. Steps in the process are conditioned on features from previous steps to increase the realism and consistency of the overall repository.

We demonstrate that our architecture, in particular TRN, is superior in generating trees with samples that reduce degree mean maximal distance (MMD) by 90-92% and depth MMD by 75-86% to a held out test data set in comparison to a recent deep graph generative model and a random tree generator. These tests mainly follow the approach in [12]. Furthermore, we visually compare the filenames and file content of our RNN models to their authentic counter-parts and show that these models generate convincing filenames with authentic syntax and realistic file content.

In summary, this paper makes the following three novel contributions

1. TRN, a novel directed rooted tree generator that synthesizes the structure of a software repository.
2. HoneyCode, a scalable triple network architecture that improves authenticity and consistency through conditional features. [2]
3. An evaluation of the capabilities of conditional character-level RNN models as a tool for software repository name and content generation.

In Section 2, we review existing approaches which serve as the foundation for our work and briefly cover theoretical concepts related to graphs and metrics. We cover our architecture in detail in Section 3, which describes each of the three models that generate the key components of a honey repository. In Section 4, TRN is evaluated visually and quantitatively against other deep generative models. Samples of filenames and contents are visually assessed based on their syntactic patterns in comparison to their authentic repository counterparts.

2. Related work

Our architecture is based on the creation of three key components: a repository structure, file names and file content. In this section we discuss the foundations for these components: graph generation, language models and source code generation.

[2]The HoneyCode implementation can be found at `https://github.com/dngu7/honeycode`.

2.1. Graph Generation

Graph generation has been studied for many potential applications in network science [13], protein/molecular construction [14] and drug discovery [15]. Modelling and generating graphs is challenging due to the high-dimensional distributions underlying graph structure and non-local dependencies between edges. The majority of early methods focused on hand-crafting models for graphs with well defined aggregate properties [16, 17]. In recent years, however, the focus has shifted to modelling graphs by learning from a set of observed graphs, in particular using deep learning [18, 19]. One of the early non-sequential deep learning models proposed a variational autoencoder (VAE) based approach, however this idea is limited to single, small graphs with 40 or fewer nodes [20]. Grover [18] proposed using a node-embedding based approach and a graph neural network to learn a graph structure, but the approach is limited to a fixed set of nodes.

GraphRNN [21] applies a recurrent neural network to adjacent matrix representations of graphs to approximate the complex dependencies and distributions over real-world graph structures. GRAN [22] builds on this approach by utilising graph attention mechanisms to improve performance, scalability and reduce computation bottlenecks caused by sequential hidden states in GraphRNN.

This paper builds on these approaches for generating directed graphs, or more specifically *direct rooted trees*, to represent the folder and file structures of software repositories. We find that GRAN performs well, but is unable to consistently generate trees, even when trained exclusively on trees due to its dependence on a mixture Bernoulli distribution in edge prediction. Our novel tree generator is a modification of GRAN and is presented in Section 3.3.3.

Existing tree based learners, such as Recursive Neural Network[23] and RNN Grammars[24] focus on exclusively parse trees. Parse trees are binary trees; where a parent node has two or less child nodes. In a software repository, folders can have one or more child nodes therefore parse tree models were not considered suitable for our problem.

2.2. Language Models

The state of the art in language generation are dominated by attention based models like GPT [25]. Such models are typically trained on a very large corpus and then fine-tuned on a modest amount of data to specific tasks. Software, however, is unlike natural language, so this approach would require very large language specific source code data set to train base models.

Character based language models have a minimal vocabulary and can generally be trained on relatively small datasets. They are thus a good choice for training a model on a smaller code repository. A number of well implemented RNN based character models appeared in response to Karpathy's influential blog post [26], and so provide a basis for further development below [27]. The RNN based models, usually built on LSTM [28] or GRU [29] units, can increase their capacity and effective memory by increasing the number and size of the layers, and can be somewhat tuned to the availability of language specific training data.

2.3. Source Code Generation

The task of generating realistic source code has elements of the complexity of natural language and strict syntactic formality of programming languages. The use of neural network based probabilistic code models trained on large code bases have yielded natural samples [30] with focus on formal semantics and modality [31]. Code can be modelled as sequences of characters or abstract syntax trees (AST). Sequential models produce more visually realistic code while abstract syntax trees approaches focus on syntax accuracy and compilable qualities [31].

Sequential models predict the next character or code token based on the existing passage by learning the distribution of code patterns and relationships found in large code bases. There have been several papers that successfully generate realistic-looking scripts for a single programming language through large quantities of data [32, 33].

Software repositories, however, are highly likely to contain a diverse group of languages including Turing-complete languages, markup languages and serialization formats. Some programming languages also vary in their patterns depending on their directory locations, such as serialization data formats. Since only a subset of files in software repositories use abstract syntax trees, our paper mainly focuses on investigation the application of sequential models.

2.4. Terminology

In our approach, the repository structure is treated as a directed, rooted tree, sometimes formally known as an arborescence. Throughout this paper, the term **tree** will refer to this structure, described briefly below. We rely on the unique properties of the tree structure to create an effective generation model.

2.4.1. Arborescence

An **arborescence** is an acyclic directed tree, $G = (V, E)$, with directed edges oriented away from a single designated node called the root r [34]. This implies that an arborescence with root node r, has a unique path from node r to any other node $v \in V \setminus r$. A consequence of this construction is that all non-root nodes only have one inward edge, formally:

$$\forall u \,.\, u \in V \setminus r \Rightarrow |E_{\text{in}}(u)| = 1 \qquad (1)$$

2.4.2. Graph metrics

To compare samples of arborescences, we use degree and depth to compare populations of graphs quantitatively. **Depth** of node $u \in V$, formally $\text{dep}(u)$, is the number of nodes -1 in the shortest path between the root node r and node u.

Populations of graphs can be compared by using the univariate distribution of depth and degrees which will be investigated in Section 4.

3. Approach

We start with an brief discussion of the requirements for deceptive code repositories, followed by an overview of the architecture, HoneyCode, in Section 3.2, and descriptions of data pre-processing and each of the core neural network models used to generate the components of a repository.

3.1. Requirements

Any deception has a limited lifespan and ability to withstand scrutiny, and should be evaluated with respect to the expected observer and the tools available to them. As an example from deception in the physical domain, a well camouflaged soldier is hard to see with the naked eye, but vulnerable to detection with infra-red technology or a sniffer dog. To evaluate HoneyCode we make the following assumptions in our model of the adversary: The attacker has gained a foothold on the network with the aim of exfiltrating information related to a specific topic or topics (which we do not know but would like to discover). The attacker is a member of an intrusion team, but not a software engineer or other topic specialist, does not know the exact locations or extent of the information they seek, and is time constrained. The interfaces available to the attacker are a command line that allows search of file systems with Unix tools, standard filesystem browsers (like File Explorer on Windows) and document and code repository search engines via a browser interface (similar to the Github search interface, for example).

Under these assumptions, HoneyCode should generate content that passes the following tests. (1) Structure: a directory listing or file browser navigation of the repository should reveal a plausibly realistic looking folder tree, with typical locations for resource types and appropriate filenames. (2) Searchability: an intruder searching using keywords should discover content that mimics real code, contains similar variable names and other indications of content like data types. Training the HoneyCode model on the real source code being protecting makes this element of realism more convincing. (3) Local context: generated code has to pass cursory inspection by a time constrained malicious actor seeing the response to searched keywords. This is because a search engine or grep process will typically display a few lines around the search result. This code block should look correct in terms of syntax, layout and indentation, at least to the casual inspection of someone who is not a language expert.

It is, of course, also possible to hand-craft deceptive content. Such efforts can be time consuming and expensive, so the generation process must be automated and scalable. With these requirements in mind, then, we examine the components of HoneyCode.

3.2. Triple Network Overview

We approach the construction of a system capable of generating synthetic software repositories by factoring the problem into three separate components: repository structures, names and file content. This decision to separate the tasks was motivated by their different underlying structures and representations. The proposed system architecture thus uses a separate neural network to learn a model of each component, and uses them sequentially to generate synthetic repositories. At each step, subsequent networks can utilize a conditional property from prior network samples to improves the overall consistency of the result.

We introduce a novel deep tree generative model called **Tree Recurrent Network (TRN)**. TRN effectively generates the structure of a repository representing the folder and files. A conditional RNN called, **Name Generator**, creates explicit names for files and folders while another RNN, the **Content Generator**, creates file content for a variety of file extensions.

3.3. Tree Recurrent Attention Network

Our TRN architecture is a variant of the Graph Recurrent Attention Network (GRAN) [21]. We introduce several modifications to GRAN to better address tree generation, rather than general graphs. The most significant change is the replacement of the Bernoulli mixture by a multinomial to represent the probability distribution of all possible inward edges between a new node and set of existing nodes.

To illustrate, given an existing tree with k nodes, TRN generates a multinomial distribution with k possible events. At every step, the new node's inward edge is sampled from this distribution. This simple idea was motivated by the observation that all nodes of trees only have *one inward edge* (see Equation 1). Furthermore, this approach matches the generator output with the exact structure of the required graph. This removes the need for post-sample processing and filtering as required in obtaining trees from general graph based algorithms such as GRAN.

Another significant property is the introduction of a new depth feature into the node state and messages, represented by a one-hot vector. This helps differentiate edge prediction distributions based on the distance of the node from the root. It can be interpreted as introducing node type or heterogeneity to the tree's overall representation.

In the remainder of this section, we detail our approach to tree representation, hidden node/message states and edge prediction.

3.3.1. Representing trees as topologically sorted directed matrices

A directed rooted tree can be represented by an adjacency matrix S^π with node ordering π. If we

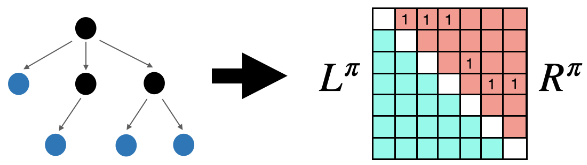

Figure 1. A tree directed adjacency matrix with an empty lower left sub-diagonal L^π and upper right sub-diagonal R^π

enforce a topological ordering of nodes the first node of π must always be the root. Trees sorted topologically can be completely represented using only the upper right triangle R^π of an entire adjacency matrix S^π, as shown in figure 1.

This allows us to define the reconstruction of R^π as the goal of our generation problem. The representation of R^π importantly provides the distribution of all possible **outgoing edges** $E^*_{\text{out}}(\pi(v))$ for all nodes:

$$p(R^\pi) \equiv p(E^*_{\text{out}}(\pi(v))) \quad \forall v \in V. \tag{2}$$

Computing the distribution of all outgoing edges of a new node may work, but is challenging because a node can have zero or more outgoing edges.

Our approach simplifies this problem by taking advantage of the property in Equation 1 and, instead, learning the transposition of R^π which is represented as L^π. This is equivalent to the distribution of **inward edges** for each node, $p(E^*_{\text{in}}(\pi(v)))$, formalized as:

$$p(R^\pi) \equiv p((L^\pi)^T) \equiv p(E^*_{\text{in}}(\pi(v))) \quad \forall v \in V. \tag{3}$$

As Equation 3 shows, we can recreate the intended R^{π} by re-transposing final output L^{π}. By using this simple trick, each node's row is a one-hot vector where the index of the one value represents the parent's node. This new representation is leveraged to generate input-label pairings as demonstrated in Figure 2. From a single tree, L^{π} is sub-graphed at various indexes to learn the distribution underlying the entire matrix.

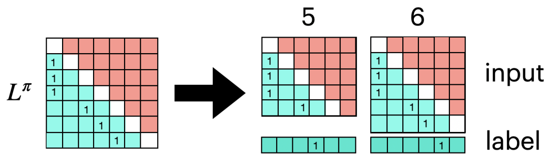

Figure 2. Using L^{π} as the base matrix allows creation of input-label pairings where, importantly, each label is a one-hot vector.

This change enables us sample a new inward edge using a multinomial which, as we will demonstrate in section 4, boosts sample quality significantly and guarantees a valid tree that satisfies Equation 1.

3.3.2. Message feature set

Constructing a rich message feature set is key to building an effective graph neural network. The message encodes the node's state transition and is passed throughout the node's neighborhood. Using only inward and onward edges as node state, demonstrated in several deep generative models [21, 12], can be supplemented by introducing additional features such as node type. In TRN, the node state is supplemented by a depth-based feature which helps the model differentiate edge prediction patterns in the underlying representational space.

Edge-based features During experimentation, we found that combining L^{π} and its transposition L^{T} generates better performance in comparison to only using L^{π}. We will denote the addition of these two matrices as A^{π}. This new matrix could be as seen as sending a message that encodes both the node's inward and outward edges which would be important in edge prediction.

Depth-based feature We utilize depth as a feature set which improves the sample quality, particularly in depth MMD. The depth feature set acts as a representation of the node's location on the overall tree, thus introducing new distributions at different depths.

This feature is represented as a one-hot vector where the depth of the root node is one and newly generated nodes is zero. Newly generated nodes are nodes where, for the current training or generation step, we are predicting the inward edge from existing nodes.

3.3.3. GRAN Variant

Depth-based Message Passing Framework We use a message framework which converts a feature set F into a message and uses self-attention to vary the strength of each feature during propagation. This mechanism, modified from [21], has shown to effectively share information across a node's neighborhood, which allows us to learn the underlying representation of a tree. At each propagation step, each node's hidden representation is updated using the self-attention messages through an LSTM module. The initial hidden representation h_v^0 for node v is a linear mapping of A^{π} discussed in section 3.3.2. For a given node u, the hidden state can be represented as:

$$h_u^0 = W_h A_u^{\pi} + b_h \tag{4}$$

During the k-th round of message propagation, the message from node v to node u is calculated as:

$$m_{vu}^k = [f(h_v^k - h_u^k), OH_{dep}(u)] \tag{5}$$

where h_v^k is the hidden representation for node v at propagation step k and $OH_{dep}(u)$ is the one hot vector of node depth, $dep(u)$. The message function f is a 2-layer neural network with ReLU activation function.

LSTM module TRN uses an LSTM module to approximate the state of each node, which we found to perform marginally better in comparison to a GRU module. The main difference between the two RNN variants is that an LSTM has an initial cell state c_0 and hidden state h_0 where as GRU only has a hidden state h_0. We allow the cell state to become a map of A^{π} by applying a linear layer demonstrated as:

$$c^0 = W_c A^{\pi} + b_c \tag{6a}$$

During the k-th round of message propagation, these states are updated by self-attentive messages via the LSTM module. These self-attentive messages combine the attention weights a_v^r of node v over its neighborhood of nodes $N(v)$:

$$\hat{h}_v^{k+1}, \hat{c}_v^{k+1} = LSTM(h_v^k, c_v^k, \sum_{u \in N(v)} a_{vu}^r m_{vu}^r) \tag{7}$$

3.3.4. Edge prediction with multinomial distribution

Once all node states h^K are updated with K propagation steps, we use these state representations to calculate the probability distribution of all potential edges p($E_{\text{in}}^*(v)$) for new node v. To calculate the probability of an edge between new node v and the existing nodes $u \in V$ we take the difference between their states and apply function fs with an output of size V which represents the number of nodes in the existing graph. Softmax is applied to the output and creates a probability distribution over all possible edges which is shown in the equation below:

$$p(E^*_{\text{in}}(v)) = Softmax(fs(h^K_v - h^K_u)) \quad \forall\, u \in V. \quad (8)$$

where fs is a feed forward network with output size of V and $p(E^*_{\text{in}}(v)) \in R$ is a vector of size V.

Due to the observation of Equation 1, we introduce a hard constraint by sampling with a multinomial using $p(E^*_{\text{in}}(v))$ to select a single $E_{\text{in}}(v)$ for node v.

This new edge is added to the existing graph and adjacency matrix R^π and used for the next training step allowing us to create the intended learning framework:

$$p(R^\pi) = \prod_{k=1}^{N} p(R^\pi_k \mid R^\pi_{0:k-1}) \qquad (9)$$

where k is the generation step and N the maximum number of nodes. This generation process is summarized in Figure 3 where, at each step, a new node is generated and the model predicts the inward edge from the set of existing nodes.

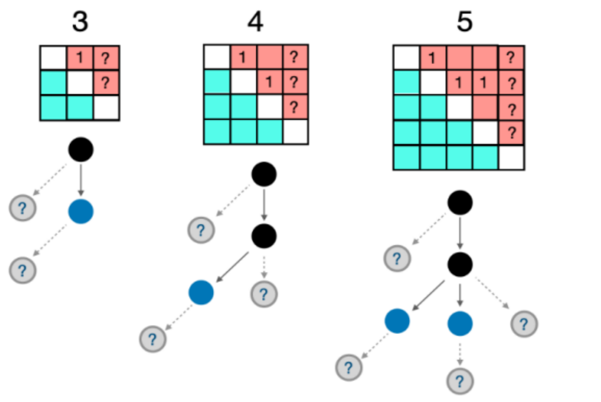

Figure 3. The growth of a tree and its corresponding adjacency matrix where ? represents the location of potential new edges.

3.4. Folder and File Name Generator

The goal of vanilla character level language model is to learn the probability of the next character c_{n+1} conditioned on the previous characters $(c_1, ..., c_n)$. Our approach utilizes additional conditional features which yield more realistic filenames and repository names. We propose the use of a conditional character level GRU model, since there is a negligible performance difference in comparison to an LSTM, and it requires only one hidden state. The hidden states of the name generator are initialized to zero.

HoneyCode uses the repository structure G as input to the Name Generator Network. For the remainder of the section, a list of characters of length N representing a partially generated name will be denoted with a capital $C_N = (c_1, c_2, ..., c_n)$.

3.4.1. Features and conditionals

The naming convention of repository nodes is heavily affected by whether the node is a file or a folder and where the node is located in the overall repository. For these reasons, given C_N for node v, the list of features used to predict c_i are: (1) Prior character, c_{i-1}, (2) Depth of current node, $dep(v)$ and (3) Node type (based on node outdegrees, $deg_{out}(v)$). The node type of v is a file if $deg_{out}(v) = 0$ and is a folder if $deg_{out}(v) > 0$.

The set of possible characters is mapped to an embedding layer Em_c. The depth of the current node $dep(v)$ is mapped to embedding layer Em_{dep}. The node type feature, formally $nt(v)$ for node v, is mapped to a binary set where files are zeros and folders are ones. This feature was also mapped to an embedding layer Em_{nt}.

3.4.2. Network architecture

The input to the GRU neural network is a matrix, inp_{all}, computed as the concatenation of all 3 embedding layers outputs discussed in the previous section: inp_c, inp_{dep} and inp_{nt}. The hidden layer h_t of the GRU network is initialized as a matrix of zeroes with a size of y. The output of the GRU network is passed through a final linear layer creating a new vector of size P.

3.5. Content Generator

The approach for the Content Generator is very similar to the Name Generator. The main difference is that the Content Generator uses file extensions as its only conditional feature. The reason for this approach is that the syntax structure of file content differs significantly depending on file extensions.

The Content Generator use embedding layers, character-level modelling and conditional stacked GRU architecture, which is exactly the same as the name generation model approach. The Content Generator, however, produces significantly more characters than the name generation network. The model also only applies to nodes which are files, which is determined by the set of nodes which follow:

$$E_{\text{out}}(v) = 0, \ \forall v \in V.$$

Given a file for node v containing existing characters C_T , the list of features used to predict c_{t+1} are: (1) The prior character in C, c_{i-1} and (2) extension type, $ext(u)$.

The number of available file extension types EXT per GRU network is limited. Above 4 extensions per network, the quality of samples begins to deteriorate based on our observations. Given sufficient compute power, one could simply add more networks horizontally to add more file extensions.

4. Evaluation

In this section we evaluate the performance of HoneyCode by sampling components of each model and performing visual and statistical tests in comparison to their authentic counterparts. We train and test our architecture on 3254 publicly available software repositories from Github. The primary language of these repositories is Julia, however they contain of a wide range of formats and languages including markdown, XML and TOML.

4.1. Tree Recurrent Network

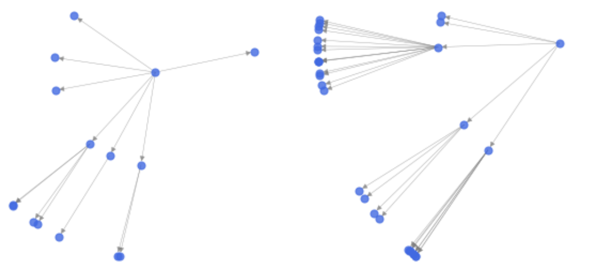

Figure 4. Graph visualization of folder structures from the test data set

We evaluate the performance of TRN compared to GRAN and a baseline random generator model using Mean Maximal Distance (MMD) [35] as a framework to assess the quality of each model's samples. We consider a random generator as a benchmark because it demonstrates that TRN can learn the underlying representational distribution of the data, as opposed to randomly choosing any parent node at every decision step. GRAN is used as a benchmark to demonstrate that the modifications applied to TRN result in direct improvement in generative performance for direct rooted trees.

We use the Total Variance (TV) as the MMD kernel which is significantly faster than the alternative Earth Mover Distance (EMD) kernel and provides similar results. To compare populations of graphs, we use the univariate distribution of degree and depth. Several other metrics are proposed by graph generation authors [21, 12] including orbital count, clustering coefficient and spectral properties however these are not applicable to tree structures. We also visually inspect and compare several

Metrics	GRAN	Random	TRN
Tree (%)	13.6	100	**100**
Degree$_{mmd}$	0.1232	0.1309	**0.0110**
Depth Density$_{mmd}$	0.1007	0.1913	**0.0259**

Table 1. A comparison of the 160 samples from multiple models to the test data set after 100 epochs of training using the MMD framework. Smaller is better.

samples from each model to the test data set shown in Figure 4.

The best performing TRN, within hardware constraints, is a graph neural network with an LSTM module containing 7 layers and performing 1 propagation step of messages. The dimension size for each feature are $f_{dim} = 512$ and $m_{dim} = 512$.

For comparative purposes, we use similar hyper parameters where possible for the baseline GRAN. GRAN works primarily on undirected graphs so we adapt their model for directed graphs by allowing edge predictions for both inward and outward edges for every new generated node.

The random generator model enforces the key property described in Equation 1, one inward edge per node. At each generation step, the new inward edge is selected using a pseudo-random choice between existing nodes. This model is referred to as **Random** in Table 1.

One of the major difference between the three models is the percentage of samples that satisfy the properties of a tree. This is calculated under the metric *Tree (%)*, in Table 1. GRAN achieves less than 100% because the model computes the edge prediction distribution as a Bernoulli mixture, and does not enforce the constraint that the graph must be a tree. It thus generates general graphs despite having been trained exclusively on trees. This property is apparent in Figure 5, which shows GRAN generates graphs with isolated nodes and nodes with more than one inward edge. Upon further inspection, the largest graph in the population of valid trees was three nodes connected by two edges to form a single line. These visualizations show that GRAN's base model is not suitable for generating direct rooted trees. Other general graph generators, such as DeepGMG [19] and GraphRNN [12], are likely to suffer from the same problem because their decoders do not follow the property described in Equation 1.

In our model, edge prediction is modelled as a multinomial distribution which ensures that only 1 inward edge is generated per new node. This is demonstrated in Table 1 where all samples satisfy the properties of a tree. Furthermore, this model produces samples that have a smaller degree distribution mmd and depth density distribution mmd in comparison to both

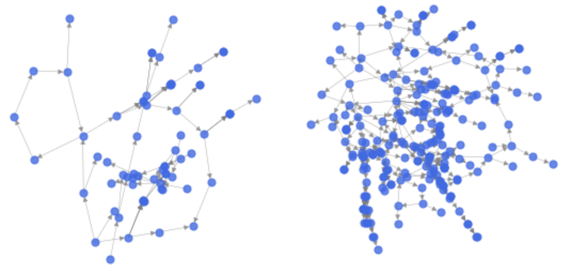
Figure 5. Visualization of samples from GRAN generative model trained over 100 epochs

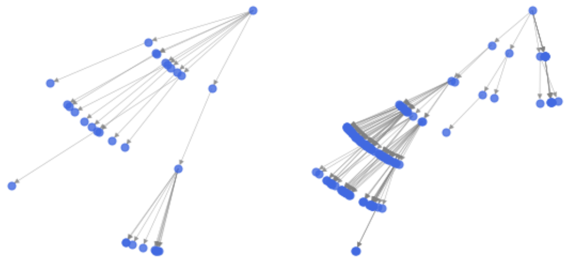
Figure 6. Visualization of samples from TRN generator trained over 100 epochs

GRAN and the random generator. This indicates that the sample's depth and degree distributions fit closer to the test data set in comparison to other models.

4.2. Name Generator

In this section, we evaluate samples from the Name Generator by comparing their outputs to the test data set. We also explore the importance of our conditional approach by comparing samples across various conditional features. It will become apparent that without a conditional approach, the filenames and folder names would not be consistent with an authentic software repository and would likely be detected as fake by an adversary simply viewing directory listings.

The data-based hyper-parameters used for the evaluation model are $C_{dim} = 512$, $dep_{dim} = 128$ and $nt_{dim} = 128$. The model uses 1500 hidden units and three stacked layers of GRUs to reduce training times within our restricted time constraints. These hyper-parameters produce sufficiently good samples for production purposes, as we will see below.

The model is sampled using various conditional combinations of node types (folder, file) and node depths, up to 5. The samples for folders and files can be found in Tables 2 and 3.

In Table 2, we find that the majority of folder name samples with depth=0 have an end pattern '.jl' which is consistent with the names of Julia repositories found on Github. Furthermore, the samples have human readable words with the exception of number four.

With folder name samples of depth=1, the quality continues to be very high as the folder names, 'src', 'contrib', 'examples' and 'test', commonly appear as the first folders of Julia repositories. Most importantly, we can tell the depth conditional features provide useful information because the '.jl' pattern no longer exists for folders greater than 0. At depths between 2 and 4, the names are human-readable and realistic although we note that there is a repeated folder name 'examples'.

Samples of filenames in Table 3 display the same pattern of quality seen in samples of folder names. The quality of filenames at depths between 0 and 3 are high. For example, the model produces names such as 'LICENSE.md', 'README.md' at depth 1 which are very commonly found in Github repositories. At depth 2, the samples all contain the extension '.jl' and have descriptive readable names, such as 'runtests.jl' which are consistent with julia scripts. However, at depth 4 and 5, we begin to see the beginnings of some deterioration with invalid file extensions such as 'berlin-mitter-med.000150' and '60000_by_1000000_run3'. This is likely due to that fact that the majority of files appear at depths lower than 5.

Despite these limitations, its still important to note that the model learns to always generate file extensions, a critical characteristic of realistic file names. Furthermore, the Repository Structure Generator produces trees where $\approx 99\%$ of files and folders exist at depths less than 5. It is unlikely that these poorer quality samples would be generated by the name model in large quantities.

4.3. Content Generator

In this section, we evaluate the performance of the Content Generator by visually inspecting the quality of generated samples. Due to space constraints, we limit our analysis to one file extension, despite actually training the generator with four different file extensions (Julia, Markdown, TOML, YAML).

The embedding layer dimension sizes used for the evaluation model are $C_{dim} = 512$ and $Ext_{dim} = 128$. The model uses 2000 hidden units and three stacked layers of GRU to reduce training times within our restricted time constraints.

Julia Language As shown in Figure 7, the Content Generator model learned key syntactic patterns found in the Julia language such as function and type definitions, variable assignment and for loops. The code sample is also visually similar to an authentic Julia script due to the indentation structure, one of the most important visual characteristics. Upon closer inspection, a human with basic programming capabilities should be able to quickly identify that the function will not compile.

Depth	1	2	3	4	5
0	common.jl	NetworkParallelLemes.jl	JSONLDD.jl	qquisites	network.jl
1	deps	src	contrib	examples	test
2	examples	HLF2	yem	utils	figs
3	ALA_flux	pales	figs	base	Optimizer
4	58	vim	opt	pathing	pages
5	page	vid	prob	plans	gitchety

Table 2. Samples of folder names from the name generator. The left-column represents folder depth and the top-row represents the depth's respective sample id. Samples of depth=0 represents the names of parent repository folder.

Depth	1	2	3	4	5
1	Project.toml	LICENSE.md	appveyor.yml	README.md	observables.jl
2	ehsely.jl	runtests.jl	dataframes.jl	tools.jl	vtkFieldData.jl
3	negmres_test.jl	test_observables_consts.jl	foda.html	sort.jl	commands.txt
4	yorian_terminal.jl	BasicRJCHP.jl	berlin-mitter-med.0015	t722.xml	sets.jl
5	5.png	60000_by_1000000_run3	helper.h	page.html	amount.txt

Table 3. Samples of filenames from name generator. The left-column represents file depth and the corresponding box contains 5 samples in random order. Depth of size 1 represents files in the parent repository folder.

```
function  sparse !(d :: AbstractMixtureModel, x)
    K = ncomponentwise_logpdf!(Matrix{eltype(x)}(undef, nd, n), one(x))
    logc0 = /x * detach(1)
    logc0 = max(length(I0)*length(J)+1

    for i = 1:( length(Acolptr) <= 1)
        VR = similar(A[1], T, nrows, ncols)
    elseif  size(X, 2) == 1
        logX = log(u)
        cm2 /= sqrt(1 - abs2(z)/2) * sqrt(z2)
        alpha3 = max(unsqueezeb, Tuple{Float64})
        $fname(pp, pos, durbin, sort)
            return ( false , Int [])
        end
    else
        if ! in_single_quotes
            const_prop_profitable (buf) == 0 && return false
            write( buffer , '  ')
        end
    end
    return nothing
    return  write_project (env)
end
```

Figure 7. A Julia sample from the content generation network.

Malicious adversaries with access to internal systems would likely using a text-based search engine that returns short snippets of code. If we benchmark this model against the task of deceiving this adversary, then this model performs sufficiently well in this regard as it creates visually authentic code. While there remain several obvious variable declarations omissions in the code, an adversary could consider that the variables were declared elsewhere in the environment or as a programming mistake by a novice programmer. Such investigations by the malicious adversary would require copying or opening several files or compiling the repository which should ideally trigger the appropriate detection systems.

Limitations The existing content generator network displays fragility when trained with over 4 file extensions. Additional complexity, such as filename conditionals, would have deteriorated performance further, thus for future work we plan to explore more stable and potentially larger neural network architecture such as GPT-2.

5. Conclusion

We propose a new architecture, HoneyCode, to automate the generation of synthetic software repositories for cyber deception. This language agnostic approach generates a rich set of authentic repository structures, convincing human-readable names and file content that can be used as a honeypot or part of a larger deception environment. This automated and scalable technique can become invaluable in efforts to detect and gain insight into attacker's intent, tactics, techniques and procedures. Experiments demonstrate that our novel Tree Recurrent Network (TRN) generates superior repository structures that reduces degree MMD by 90-92% and depth MMD by 75-86% on a test data set against recent deep generative graph models and baseline random models.

As future work, we plan to investigate further techniques to improve the authenticity of honey repositories such as inter-file dependency relationships and filename to content consistency. Files in authentic repositories can be linked by imports and relative references which can be represented as a large directed multi-graph intertwined with the existing folder structure. This structure could be exploited to generate file content with a higher degree of authenticity across the honey repository. Furthermore, it is self-evident that the contents of a file have a relationship to its filename. Consistency can be improved by investigating whether the filename or local group of filenames can be introduced into the file content generator as a conditional representation.

References

[1] Accenture, "2019 cost of cybercrime study: 9th annual," 03 2019.

[2] M. H. Almeshekah and E. H. Spafford, "Cyber security deception," in *Cyber deception*, pp. 23–50, Springer, 2016.

[3] L. Spitzner, *Honeypots: tracking hackers*, vol. 1. Addison-Wesley Reading, 2003.

[4] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[5] W. S. Humphrey, *Winning with software: An executive strategy*. Pearson Education, 2001.

[6] F. J. Stech, K. E. Heckman, and B. E. Strom, "Integrating cyber-d&d into adversary modeling for active cyber defense," in *Cyber deception*, pp. 1–22, Springer, 2016.

[7] C. Stoll, "Stalking the wily hacker," *Communications of the ACM*, vol. 31, no. 5, pp. 484–497, 1988.

[8] C. Stoll, *The cuckoo's egg: tracking a spy through the maze of computer espionage*. Simon and Schuster, 2005.

[9] J. Yuill, M. Zappe, D. Denning, and F. Feer, "Honeyfiles: deceptive files for intrusion detection," in *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pp. 116–122, IEEE, 2004.

[10] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo, "Baiting inside attackers using decoy documents.," in *SecureComm*, vol. 19, pp. 51–70, Springer, 2009.

[11] B. Whitham, "Automating the generation of enticing text content for high-interaction honeyfiles," in *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[12] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," *arXiv preprint arXiv:1802.08773*, 2018.

[13] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[14] A. A. Canutescu, A. A. Shelenkov, and R. L. Dunbrack Jr, "A graph-theory algorithm for rapid protein side-chain prediction," *Protein science*, vol. 12, no. 9, pp. 2001–2014, 2003.

[15] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik, "Automatic chemical design using a data-driven continuous representation of molecules," *ACS central science*, vol. 4, no. 2, pp. 268–276, 2018.

[16] S. Wasserman and P. Pattison, "Logit models and logistic regressions for social networks: I. an introduction to markov graphs and p^*," *Psychometrika*, vol. 61, no. 3, pp. 401–425, 1996.

[17] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 985–1042, 2010.

[18] A. Grover, A. Zweig, and S. Ermon, "Graphite: Iterative generative modeling of graphs," *arXiv preprint arXiv:1803.10459*, 2018.

[19] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, "Learning deep generative models of graphs," *arXiv preprint arXiv:1803.03324*, 2018.

[20] M. Simonovsky and N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders," in *International Conference on Artificial Neural Networks*, pp. 412–422, Springer, 2018.

[21] R. Liao, Y. Li, Y. Song, S. Wang, W. Hamilton, D. K. Duvenaud, R. Urtasun, and R. Zemel, "Efficient graph generation with graph recurrent attention networks," in *Advances in Neural Information Processing Systems*, pp. 4257–4267, 2019.

[22] W. Kawai, Y. Mukuta, and T. Harada, "Gram: Scalable generative models for graphs with graph attention mechanism," *arXiv preprint arXiv:1906.01861*, 2019.

[23] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning, "Parsing natural scenes and natural language with recursive neural networks," in *ICML*, 2011.

[24] C. Dyer, A. Kuncoro, M. Ballesteros, and N. A. Smith, "Recurrent neural network grammars," *arXiv preprint arXiv:1602.07776*, 2016.

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[26] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," *Andrej Karpathy blog*, vol. 21, p. 23, 2015.

[27] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[29] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[30] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from" big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.

[31] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[32] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," *arXiv preprint arXiv:1704.07535*, 2017.

[33] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[34] J.-C. Fournier, *Graphs theory and applications: with exercises and problems*. John Wiley & Sons, 2013.

[35] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A kernel two-sample test," *Journal of Machine Learning Research*, vol. 13, no. Mar, pp. 723–773, 2012.