Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Graph Generative Models for Decoy Targets in Active Directory

Master's thesis

Bc. Ondřej Lukáš

Master programme: Open Informatics
Field of study: Data Science
Supervisor: Ing. Sebastián García, Ph.D.

Prague, August 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Lukáš Ondřej** Personal ID number: **434714**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Branch of study: **Data Science**

II. Master’s thesis details

Master’s thesis title in English:

Graph Generative models for Decoy Targets in Active Directory

Master’s thesis title in Czech:

Generativní grafové modely pro klamné cíle v Active Directory

Guidelines:

1. Review the state-of-the-art methods for creating and deploying interactive honeypots with attention to systems focusing on Active Directory. Also, analyze state of the art of generative models with focus on Graph Models
2. Analyze common attacker behavior patterns and goals when targeting Active Directory.
3. Design and implement a model for adversarial generation of decoy targets in Active Directory Structures.
4. Experimentally evaluate proposed solution in real-world environment and compare it with currently used solutions.
5. Critically analyze the results and propose further extensions of the solutions with respect to possible integration with existing interactive honeypots.

Bibliography / sources:

[1] Goodfellow, Ian J., Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Warde-Farley, David, Ozair, Sherjil, Courville, Aaron C., and Bengio, Yoshua. Generative adversarial nets. NIPS, 2014
[2] Roger A. Grimes: Honeypots for Windows, Apress, 22. 11. 2006
[3] https://github.com/gentilkiwi/mimikatz
[4] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In AAAI, pages 2852–2858, 2017.
[5] Wu, Zonghan et al. “A Comprehensive Survey on Graph Neural Networks.” IEEE Transactions on Neural Networks and Learning Systems (2020): 1–21. Crossref. Web.

Name and workplace of master’s thesis supervisor:

Ing. Sebastián García, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **02.07.2019** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **19.02.2021**

Ing. Sebastián García, Ph.D.
Supervisor’s signature

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Dean’s signature

© VUT v Praze, Design: VUT v Praze, VIC

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Declaration

I hereby declare I have written this thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

In Prague, August 2020

....
Bc. Ondřej Lukáš

Abstract

Abstract

Active Directory (AD) is one of the cornerstones of internal network administration in many organizations. It holds information about users, resources, access rights and other relations within the organization's network that helps administer it.

Because of its importance, attackers have been targeting AD in order to obtain additional information for attack planning, to access sensitive data, or to get persistence and ultimately complete control of the domain. By design, any user with basic access rights can query the AD database, which means that a password leak of even the most unprivileged user is sufficient to gain access to the AD and eventually compromise other accounts with higher privileges.

A common technique while attacking the AD is called lateral movement. Attackers try to explore the network of the organization without being detected. During this time, they are performing reconnaissance in the AD in order to find high-value targets and ways of getting persistence in the domain. In these attacking scenarios the use of honeypots may greatly improve the detection capabilities of the organization by providing an early warning system. Honeypots are a well-known form of passive security measures. In the most basic form, they are decoys disguised as real devices or information about a user, in this last form they are known as honeytokens.

Despite being useful and promising a good detection, the basic constraint of a honeypot is that it should be found before the intruders attack a real target.Therefore, it is crucial to have the honeyuser placed correctly into the AD structure. However, with the complexity and diversity of AD structures, this task is very hard.

In this thesis we propose a machine learning framework for analysing an AD structure and enriching it with honeyuser accounts. We use graph neural networks and auto encoder models together with the original structure of the AD to select the best placement of the honeyusers. The models are trained and evaluated using a number of artificial datasets created from the analysis of real structures. We propose three variants of the model architecture and evaluate the performance of each them. Results show that the proposed models achieve F1 score over 0.6 in structure reconstruction tasks. Moreover, the validity ratio of the predicted placement is over 60% for the graphs of sizes similar to the real-world AD environments.

We conclude that recurrent neural networks modified for DAG processing are capable of modelling the structure of the AD and extending it with honeytokens. The generated honeytokens have similar properties to entities in the original graph which reduces the chance of their discovery.

Keywords: Honeypots, Active Directory, Machine Learning, Generative models, Autoencoders

Abstrakt

Služba Active Directory (AD) je základním stavebním kamenem interních sítí ve většině organizací. Jedná se o službu, která obsahuje informace o uživatelích, prostředcích v síti, kontaktech, přístupových právech k datům a dalších závislostech v rámci vnitřní sítě organizace. Z těchto důvodů je Active Directory cílem útočníků, kteří se snaží výše popsané informace získat a využít k dalšímu plánování útoku, přístupu k citlivým datům nebo získaní trvalého přístupu. AD je koncipováno tak, že každý uživatel s přístupem k vnitřní síti se může dotazovat řídícího serveru na další objekty v doméně, takže získáním přístupových údajů k libovolnému běžnému účtu bez zvláštních práv může útočník přímo komunikovat s řídícím serverem AD a zíkat informace a přístup k dalším účtům s většími pravomocemi.

V těchto případech je možné použitím honeypotů zvýšit šanci na včasnou detekci. Honeypot je běžně používaný nástroj pasivní ochrany. V nejjednodušší formě se jedná o past, která připomíná reálné zařízení službu či data. Poslední zmiňované se nazývá honeytoken.

Největším omezením při použití honeypotu je fakt, že k tomu aby byl účinný, jej útočníci musí najít před interakcí s reálných systémem. Proto je zásadní, aby i ve struktuře Active Directory byl honeypot vhodně umístěn. Vzhledem ke složitosti, kterou struktura AD může mít se jedná o netriviální úkol.

V této práci představujeme framework založený na strojovém učení, který analyzuje strukturu AD a rozšiřuje ji o honeytokeny. S využitím grafových neuronových sítí a autoenkodérů vybíráme vhodné umístění honeytokenu v exitujím AD. Modely jsou trénovany a testovány za použití uměle vytvořených datasetů, které jsou vytvořeny podle existujících AD. Představené modely dosahují 0.6 pro F1 metriku při rekonstrukci grafů a přes 60 % úspěšnnost při predikci hran pro honeytokeny a to i v grafech, které jsou velikostí srovnatelné s produkčními AD. Tato práce ukazuje, že rekurentní neuronové sítě upravené pro zpracování orientovaných acyklických grafů jsou schopné modelovat strukturu Active Directory a rozšířit ji o honeytokeny. Generované uživatelské účty jsou svými vlastnostmi podobné uživatelským účtům v původní struktuře, čímž se snižuje pravděpodobnost jejich odhalení.

Klíčová slova: Honeypot, Active Directory, Strojové učení, Generativní modely, Autoenkodéry

Acknowledgements

I would first like to thank my thesis advisor Ing. Sebastián García, Ph.D. from the FEE, Czech Technical University. I am truly grateful for his never-ending support, encouragement and guidance throughout the process of writing of this thesis. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to thank Ing. Maria Rigaki for her comments and insights as she was kind enough to help with this thesis as a specialist consultant. Without her passionate participation and input, the thesis could not have been successfully conducted.

Furthermore, I would like to acknowledge Ing. Veronica Valeros and all other members of Stratosphere Laboratory as additional readers of this thesis, and I am gratefully indebted to their valuable comments. I also gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan V GPU used for this research.

Finally, I must express my very profound gratitude to my parents and to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you all.

Ondřej Lukáš

List of Tables

2.1 Scopes of security groups in Active Directory [15] 9

5.1 Comparison of artificial datasets in terms of number of samples, number of nodes and edges. 42

6.1 Metrics used for the model evaluation in graph reconstruction experiment . 45
6.2 Performance of Model 1 for generating an AD structure using datasets of various graph sizes. 45
6.3 Performance of Model 2 using datasets of various graph sizes 47
6.4 Evaluation of DAG-RNN VAE using datasets with various graphs sizes . . 49
6.5 Color codes for the comparison of Models 2 and 3 52

7.1 Results of generative experiments for Model 1 55
7.2 Results of the generative experiments for Model 2 56
7.3 Results of the generative experiments for Model 3 56
7.4 Examples of generated honeyusers and predicted direct predecessors in the real AD 60

A.1 Evaluation of the influence of the latent space dimensionality on DAG-RNN Autoencoder performance (Dataset 50) 64
A.2 Evaluation of the influence of the latent space dimensionality on DAG-RNN Autoencoder performance (Dataset 150) 64

List of Figures

2.1 Active Directory attack kill chain 10
2.2 Example of a simple neural network 11
2.3 Basic recurrent cell and its unfolding 14
2.4 Block diagram of LSTM cell 15
2.5 Comparison of GRU and LSTM cells 15
2.6 Unfolding of a bidirectional RNN 16
2.7 Deep Sets Architecture - Invariant 16

4.1 General concept of the framework 25
4.2 Design of the generic model 26
4.3 DAG Ordering Example 28
4.4 Details of the structure of our special DAG-RNN layer, created for this thesis. 29
4.5 Decoder in Model 1 33
4.6 Decoder in Model 2 34
4.7 DAG-RNN VAE Model 37

5.1 Simple example of user-related graph 41
5.2 Matrix representation of the example User Related Graph 41
5.3 Example of AD data 43

6.1 ROC Curves of Model 1 per dataset 46
6.2 Precision/Recall Curve of Model 1 per dataset 46
6.3 ROC Curve of Model 2 per dataset 47
6.4 Precision/Recall Curve of Model 2 per dataset 47
6.5 Sample from Dataset 15 reconstructed with Model 2 48
6.6 Sample from Dataset 50 reconstructed with Model 2 48
6.7 ROC Curve of Model 3 per dataset 49
6.8 Precision/Recall Curve of Model 3 per dataset 49
6.9 Examples of VAE output - Dataset15 50
6.10 Examples of VAE output - Dataset50 50
6.11 PR curve comparison (Dataset15) 51
6.12 PR curve comparison (Dataset50) 51
6.13 Model 2 & 3 comparison (Dataset50) 52
6.14 Model 2 & 3 comparison (Dataset150) 52

7.1 Comparison of structures generated by Models 2 and 3 (Dataset 15) 58
7.2 Comparison of structures generated by Models 2 and 3 (Dataset 50) 59

A.1 δ hyper-paremeter tuning for Model 2 65
A.2 PR curve comparison (Dataset150) 65

A.3 PR curve comparison (Dataset500) 65

Contents

Abstract vii

Acknowledgements xi

List of Tables xiii

List of Figures xiv

1 Introduction 1

2 Background 4

2.1 Directory Service 4
2.2 Lightweight Directory Access Protocol (LDAP) 5
2.2.1 Distinguished Name (DNs) 5
2.2.2 Attributes 5
2.3 Active Directory 6
2.3.1 Active Directory Objects 7
2.3.2 Groups 8
2.3.3 Relations Between Objects in AD 8
2.3.4 Active Directory Attacks 9
2.4 Neural Networks 10
2.4.1 Backpropagation Algorithm 12
2.4.2 Stochastic Gradient Descent 12
2.4.3 Recurrent Neural Networks 13
2.4.4 Graph Neural Networks 15
2.4.5 Deep Sets 16
2.4.6 Generative Models 17
2.5 Honeypots 17
2.5.1 Types of Honeypots 18

3 Previous Work 20

4 AD Structure Modelling 23

4.1 Notation and Definitions 24
4.2 General Description of the Framework 24
4.3 Components Design in the Generic Model 26
4.3.1 DAG-RNN Encoder 27
4.3.2 DAG Recurrent Layer 27
4.3.3 Bi-directional DAG Recurrent Layer 29

4.3.4 Loss Function . . . 30
4.3.5 Implementation Details . . . 31
4.3.6 Model Training . . . 32
4.4 Model 1: Direct Edge Prediction . . . 32
4.4.1 Model Components . . . 32
4.5 Model 2: DAG-RNN AutoEncoder . . . 33
4.5.1 Model Components . . . 34
4.5.2 Loss Function . . . 35
4.6 Model 3: Variational AutoEncoder . . . 36
4.6.1 DAG-RNN VAE Model Architecture . . . 37
4.7 Transition from a User Related Graph to AD Entries . . . 39

5 Dataset **40**
5.1 Artificial Dataset Creation . . . 42
5.2 Extracting Data From The Active Directory . . . 43

6 Graph Reconstruction Experiments **44**
6.1 Evaluation Metrics . . . 44
6.2 Structural Experiment Results and Analysis . . . 45
6.2.1 Model 1: Direct Edge Prediction . . . 45
6.2.2 Model 2: DAG-RNN Autoencoder . . . 46
6.2.3 Model 3: Variational Autoencoder Model . . . 48
6.2.4 AD Structure Modelling: Model Comparison . . . 51

7 Generative Experiments **53**
7.1 Evaluation Metrics . . . 53
7.2 Generative Experiment Results and Analysis . . . 54
7.2.1 Generative Experiment: Model 1 . . . 55
7.2.2 Generative Experiment: Model 2 . . . 55
7.2.3 Generative Experiment: Model 3 . . . 55
7.2.4 Model Comparison . . . 57
7.2.5 Examples of Generated Structures . . . 57
7.3 Generation of Honeyusers . . . 57
7.3.1 Examples of Predicted Parent Nodes Testing Domain . . . 58

8 Conclusion **61**
8.1 Future Work . . . 62

A Detailed experiment results **64**
A.1 AD Structure Modelling . . . 64
A.1.1 Auto encoder with variable z-dimension . . . 64
A.1.2 δ hyper-parameter of Huber loss . . . 65
A.1.3 Model comparison with Datasets 150 & 500 . . . 65

References **69**

Chapter 1

Introduction

It is only a matter of time until an organization receives an attack. It is no longer a matter of *if*, but *when* [1]. The security community has known for a long time that some attackers will succeed, and the only solution for these cases is a security protection in every level of the organization that is dynamic and constantly evolving [2]. No unique solution is enough to deal with all attacks. Among the attacks that an organization can receive, the most critical are those which give the attacker access to the internal network. In such situations,the attackers are considered as part of the organization and security measures are more relaxed. In the last decade large companies like Sony, Austria Telekom, NTT and Citrix have been compromised and attackers gained access to their networks [3]–[6]. If those companies were breached, any company may be as well. Attackers that can access internal networks are not only amazingly hard to detect and stop, but also security protections in that level are scarce and difficult to implement.

Some of the reports of security attacks inside organizations suggest that attackers first gain access to the Active Directory (AD) system of an organization in order to learn about the internal structure and the assets to attack [7]. Therefore, many security solutions attempt to deal with how to secure AD systems and how to better gain visibility on the attackers before they get what they want.

Solving the problem of external attackers with access to the internal network and attacking the Active Directory is not an easy task, and it is usually addressed in different ways. First, there are solutions endeavoring to stop attackers from *accessing and exploring* the AD, for example by using network segmentation and limiting access to critical servers [8]. The key to performing an AD reconnaissance attack is to get access to any user in the domain. Due to the default nature of an AD, any user has the right to read the information stored in the AD. This allows the attackers to perform the initial reconnaissance before moving to privilege escalation attempts [9]. However, for the same reason, detecting scanning attacks to the AD is a very difficult task. Common defense practices

in this area use techniques that rely on hardening AD configurations and monitoring of system events [8], [10].

Another way of finding attackers in a network before they attack is to place honeytokens in the production environment. A honeytoken is a trap that is disguised as a real object and is designed to attract the attention of the attacker [11]. By definition, normal users should never interact with honeytokens and therefore, any interaction assures the detection of an attacker. Honeytokens have been used for other security detections in the past, for example as fake accounts, fake database entries, etc. [12], but nobody created, as far as we know, honeytokens for Active Directory services in order to detect the attackers as soon as they choose to access information about the fake users (also referred to as honeyusers).

The problem of creating a fake user in the AD system is larger than just creating the information about a user. An attacker can easily identify if a user is fake, thus it is important to create a user with realistic information and, more importantly, a user that is placed correctly inside the organization. Therefore, where to place a fake user inside the AD system is paramount for the success of the detection mechanism. Since there is no research so far solving these problems, AD systems in production right now do not have a good way of creating honeyusers inside their systems in a way that actually looks like a normal user.

In an attempt to solve these issues, this thesis proposes to reconstruct the structure of an existing AD and to generate a new structure that adds new fake users in it. This is done by training a generative machine learning model that generates AD structures with fake users inside. In our approach, we analyze the structure of the whole AD domain with deep learning methods and use a model to determine which is a suitable location for placing the fake users. Since Active directory is designed as a tree structure considering a group membership as a type of edge in the graph, the whole domain can be transformed into a directed acyclic graph. In recent years, deep learning methods focusing on graphs and graph structured data have been shown to be powerful enough to outperform traditional ML methods. This thesis researches the following problems: reconstruct a current AD structure effectively, and find the best location in that current AD structure to place honeyusers.

In order to train a good generative model of an AD structure we need good labeled data. However, since AD holds sensitive information about the structure of an organization, its components, users and resources, its is extremely difficult to obtain the real AD data from an organization. Sharing or even extracting information from a production AD is often forbidden for third parties by company policy. Since training an ML model required a substantial amount of data samples, we had to reuse limited real data samples,

by combining them with expert knowledge and known best practices for AD set up, to create artificial datasets. These datasets are good enough to perform the required task and they only differ in the number of nodes of their graph structures.

There are two main experiments performed: Evaluation of on task, where models attempt to reconstruct the original graph, and generative task, where models enrich the existing structure. The structures created in second type of experiments are evaluated with existing tools to verify compatibility.

We showed that model architecture based on the auto encoder is capable of capturing the relation withing the graphs and create node-level encoding of a fixed size. Comparison of the models resulted in finding that models base on direct edge prediction are scalable to a graph sizes common in the AD domains.

Experiments with sequential generation of honeyusers for the existing domains showed that proposed framework can be utilized for such task. The proposed model produces AD objects with properties similar to the objects of same type in the original. The experiment results suggest that objects generated using proposed framework are viable for using as a honeypots. However, further evaluation with human interaction is necessary for conclusive proof of this hypothesis. One of the outcomes of this work is a functional tool for automated honeypot deployment in the Active Directory. For the model, we created a Tensorflow implementation of DAG-RNN scalable for use in structures containing hundreds of nodes, which is two orders of magnitude higher than the original paper. The custom layer is based on the Tensorflow/Keras API and is compatible with other modules in the library.

The thesis is structured as follows: Chapter 2 describes the directory services with special impact on Active Directory. Additionally, it briefly mentions the basic building blocks of modern neural networks. Chapter 3 describes the state of the art of graph neural networks with special attention to generative models and autoencoders. It also shows examples of work using ML methods in honeypot creation and deployment. Moreover, it also mentions commonly used tools for scanning the Active Directory. Chapter 4 contains proposed model architectures for processing the AD structure, mainly the description of our design and implementation of the DAG-RNN layer and its use in graph encoding. Chapter 5 explains how the datasets used in this thesis are created and their properties. Chapter 6 provides information about the experiment setup structural modelling evaluation while Chapter 7 shows methodology and results for generative experiments.

Chapter 2

Background

2.1 Directory Service

Directory service is a shared infrastructure used to manage, organize and locate resources in a network. Such structures can include data, users, devices and groups that are being used on a daily basis. Directory service is a cornerstone of shared resources, accounts, and credentials within a computer network inside an organization. The directory server, also known as name server, provides the service for the particular network. Each object in the network has a collection of attributes associated to it and also a name that is unique in the namespace defined by the directory service. This illustrates how a directory service and a relational database can be similar. However, with a directory service, data can be redundant in the interest of performance. There are two basic types of attributes which a class of objects can have. These are defined in a Directory Schema:

- Must - attributes which each instance of a particular class must have
- May - attributes which may be defined for a instance but can be omitted. (Similarly to NULL in a relational database)

In 1980s, the International Telecommunication Union (ITU) and International Organization for Standardization (ISO) published a collection of standards for directory services known as X.500 which are also incorporated in ISO/IEC 9594[13]. Based on this standard the Lightweight Directory Access Protocol (LDAP) was founded as an open, vendor-neutral, string encoded protocol for accessing and maintaining directory services over the Internet.

2.2 Lightweight Directory Access Protocol (LDAP)

Lightweight Directory Access Protocol (LDAP) is a protocol based on TCP/IP which is designed to perform a variety of operations in a directory server. The standard TCP ports for LDAP are 389 for unencrypted communication, and 636 for LDAP over a TLS-encrypted channel. However, for a variety of reasons it is not uncommon for LDAP servers to listen on alternate ports.

An LDAP entry is a collection of information about an entity. There are three components in each entry: the distinguished name, a collection of attributes, and a collection of object classes.

2.2.1 Distinguished Name (DNs)

Distinguished name of an entry, often referred to as DN, is a unique identifier of an entry and its position within the directory information tree. It is much like a path to a file in file system. A DN is composed of zero or more elements called Relative Distinguished Names (RDNs). If an entry has multiple RDNs, their order specifies the exact location of the entry in the structure. RDNs are separated by commas, and each RDN in a DN represents a level in the hierarchy in descending order (moving closer to the root of the tree, which is called the naming context). That is, if you remove an RDN from a DN, you get the DN of the entry, considered the parent of the former DN. For example, the DN *"uid=john.doe,ou=People,dc=example,dc=com"* has four RDNs, with the parent DN being *"ou=People,dc=example,dc=com"*.

Each RDN consists of a name-value pair. Note that despite each component of a DN is a RDN, it is common practice to refer to the leftmost component of an entry's DN as the RDN for that entry. In the example ***"uid=john.doe,ou=People,dc=example,dc=com"*** the component ***"uid=john.doe"*** would be called the RDN of the entry. The attribute name-value pairs in this leftmost component must be present in the entry (so the entry ***"uid=john.doe,ou=People,dc=example,dc=com"*** must contain a uid attribute with a value of ***"john.doe"***).

2.2.2 Attributes

Attributes are the elements used for storing data in a directory. The LDAP Schema defines the rules for which AttributeTypes may be used in an LDAP Entry, which values those AttributeTypes may take, and how users may interact with those Attribute Values. Microsoft's AD Schema further holds:

1. the syntax of each Attribute in the schema

2. which Attributes are replicated to the Global Catalog
3. whether they are SINGLE-VALUE or MULTI-VALUE
4. which class of objects can use each attribute.

A complete list of AD object attributes can be found in [14]. Object Classes are elements that specify a collection of attributes types that can be related to a particular object. Each entry has its structural object class which defines the core type of the entry. Structural classes are the only classes that can have instances.

2.3 Active Directory

Active Directory (AD) is an implementation and extension of Directory Services created by Microsoft for Windows domain networks. AD was first released with Windows Server 2000 and its functionality was extended over the years. Nowadays, AD is composed of the following services:

- Domain Services
- Certificate Services
- Lightweight Directory Services
- Rights Management Services
- Federation Services

In this work, we are mainly focused on the Domain Services part of Active Directory (AD DS). It is the core part which manages users and computers and allows sysadmins to organize the data into logical hierarchies. AD DS also provides services for security certificates, Single Sign-On (SSO), LDAP, and rights management. There are two important classes of objects we will focus on: container classes and account classes. Containers are objects designed to hold other objects such as OrganizationalUnit or GroupPolicyContainers. Account classes are objects which represent a specific entity in the structure. An obvious example is the User object, although Computer objects are also part of this class. In the following subsections, we will examine objects that are used later in the thesis in detail.

2.3.1 Active Directory Objects

Domains

The core entity in the logical structure of AD is the domain. It is a special object which allows grouping of other objects to reflect the company organization. The domain also serves as a security boundary. Access Control Lists (ACLs) are used to define which users or groups of users can gain access to an AD object and what kind of access. Security policies do not cross from one domain to another.

Trees

In case the organization has more than one domain which all share a namespace, those are organized in a tree. All domains in a tree share a common schema, which is a definition of all object types and additional attributes. Moreover, all domains within a tree share a common Global Catalog, which is a centralized repository of information about all objects in a tree. Parent and child domains in a tree are linked by a special type of connection called trust. Trust allows users from one domain to access resources in another assuming they have access.

Forests

A forest is a collection of one or more trees. All trees in a forest share the same schema. Similarly to trust links in one tree, trust between trees can be formed in a forest to connect one or more trees.

Organizational Units

An Organizational Unit (OU) is special type of container in AD which can hold different objects from the same domain such as other containers, groups, users, and computer accounts. The structure of OUs usually follows the structure of the organization either functionally (Sales, R&D, IT, etc.) or geographically. Note that since an OU is a type of container, it can be nested. There are two main reasons for using OUs:

1. Delegation of management and administrative tasks to other administrators and users without the necessity of granting them domain administrator permissions

2. Linking Group Policies to all objects within the OU

2.3.2 Groups

In order to simplify the communication and administration of large organizations, there are two group types in AD: Distribution groups and Security groups. Distribution groups are used to distribute messages to group members with email applications such as Microsoft Exchange. Distribution groups are not related to security and therefore cannot be used to assigning permissions to resources. For that reason we are not focusing on them in this thesis and use the term group as an equivalent to Security Group.

Security Groups

The purpose of security groups is to allow system administrators to assign permissions and user rights to members of the group. Granting permissions for the whole group rather than for each user independently is much more efficient. Additionally, it allows for changing the rights of single users just by adding or removing them from the group based on the current requirements while leaving the groups' permissions or rights unchanged.

Groups can have different scopes, meaning the permissions and user rights of that group are only valid in certain parts of the AD structure. Table 2.1 describes the differences in scopes of each security group type [15].

2.3.3 Relations Between Objects in AD

So far, we have talked about different object types within the Active Directory. Let us briefly look at the relations that can exist between the objects. Earlier we described the simplest relation determined by RDN. It describes the position of the object within the AD as a path from the root node to the particular object. Other relations are described by the permissions, and most importantly by attributes *memberOf* and *members* which describe membership of objects in groups. By combining the attributes of a pair of objects one can infer more relations such as *AdminTo*, *hasSession* and a number of specific Access Control Entries. Using these relations as oriented edges and objects as nodes, we can view the whole structure of an Active directory as a Directed Oriented Graph. A notable tool that uses graph theory to plot and analyze the structure of an AD is called Bloodhound [16]. Bloodhound is an open source tool that relies on Powershell[17] and LDAP to query the AD, and uses a neo4j [18] database to store and analyse the structure. It is widely used by both system administrators and red teams to find the weak points in AD setups and to plan attack vectors for gaining persistence in the domain and further escalate privileges.

Scope	Possible Members	Can Grant Permissions	Possible Member of
Universal	Accounts from any domain in the same forest Global groups from any domain in the same forest Other Universal groups from any domain in the same forest	On any domain in the same forest or trusting forests	Other Universal groups in the same forest Domain Local groups in the same forest or trusting forests Local groups on computers in the same forest or trusting forests
Global	Accounts from the same domain Other Global groups from the same domain	On any domain in the same forest, or trusting domains or forests	Universal groups from any domain in the same forest Other Global groups from the same domain Domain Local groups from any domain in the same forest, or from any trusting domain
Domain Local	Accounts from any domain or any trusted domain Global groups from any domain or any trusted domain Universal groups from any domain in the same forest Other Domain Local groups from the same domain Accounts, Global groups, and Universal groups from other forests and from external domains	Within the same domain	Other Domain Local groups from the same domain Local groups on computers in the same domain, excluding built-in groups that have well-known SIDs

Table 2.1 Scopes of security groups in Active Directory [15]

2.3.4 Active Directory Attacks

Holding information about users, devices and resources within the organization, Active Directory is a natural target. The scope and goals of attacks spread from stealing and leaking user data to complete destruction of the domain. Example of the latter can be the Maersk incident [19]. In 2017, one of the offices of this international shipping company was attacked with a malware called NotPetya. According to the reports, within minutes from the infection of the first user, the worldwide network of the company was rendered useless. This included more than 40,000 devices, over 1,000 applications, file-sharing and printing capabilities, cloud, and Active Directory servers all being put offline.

Despite having a different outcome, the attacks typically start with breaching one account using various phishing techniques. The goal of this part of the attack is to get access to any account in the domain. Due to the protocol design, an initial foothold on the AD Domain allows the attacker to query the AD server and get additional information about the objects in the domain. In this thesis we are not focusing on the detection of the first step of the attack which is the credential breaching. We work with the assumption

that some non-privileged account was already compromised.

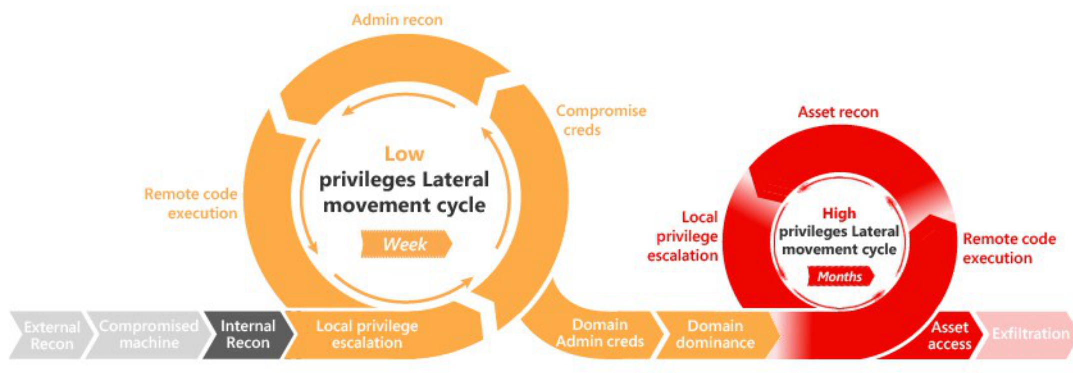

Figure 2.1 | **Active Directory attack kill chain.** Sequence of steps that the attackers perform to dominate a domain [9].

The common sequence of steps that the attackers perform when attacking AD is shown in Figure 2.1. After breaching the domain, the first phase of the AD recon starts - Low privileges lateral movement. In this phase, attackers try to find and compromise accounts with higher privileges, map the domain, and prepare the ground for more targeted attacks. It is common practice to achieve persistence even in case of a password change in the compromised accounts. The final step of this phase is called Domain dominance and at this point, the attacker controls the domain, has access to admin accounts, can execute code and move freely in the domain. Note that the lateral movement phase can take anything between a couple of hours to weeks depending on the size of the domain. In this period, it is crucial for the attacker to remain undetected until the domain is fully controlled. In this thesis we aim to focus on using honeyusers to detect attackers during lateral movement phase.

2.4 Neural Networks

The human brain is capable of learning tasks without prior knowledge, using examples and experience. This has been a great inspiration and a founding idea for a field of artificial intelligence called machine learning. In certain domains it is impractical or even impossible for a human to create a program to perform a particular task. However, we can collect a set of samples described by features and encode the desired output for each of them. For utilizing the training process, a mathematical model which can adapt itself is required. Such models are designed to use statistics to find and encode underlying patterns in the training samples.

Inspired by the way information is processed in a biological system, mathematical model neural systems were created [20], which we call them Neural Networks. The first idea of artificial neuron was proposed in 1943 in [21] and later extended and further developed into more complex systems such as Multi-Layer Perceptron [22].

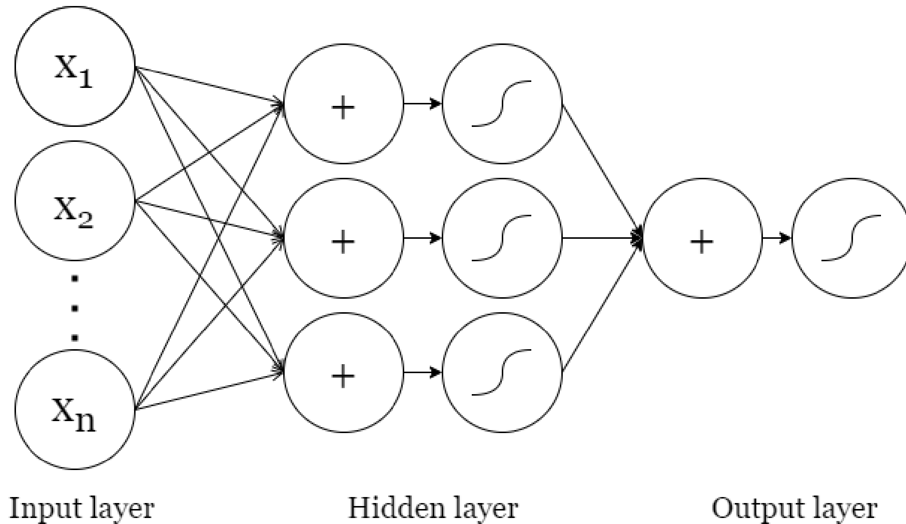

Figure 2.2 | **Example of a simple neural network.**

Figure 2.2 shows an example of a simple neural network. On the left, there is an input layer which takes a feature vector $X = \{x_1, x_2, ..., x_n\}, x_i \in \mathbb{R}$ describing an arbitrary data sample. Next, there is a hidden layer, consisting of three neurons. There are two operations performed in each neuron during the forward propagation of information: weighted sum of the inputs and activation functions

Next there is a hidden layer, which consists of three neurons $\{h_1, h_2, h_3\}$. Each neuron in the hidden layer first computes a linear combination of all of its inputs. In case of h_1 it is computed as:

$$h_1^{in} = x_1 w_{1,1} + x_2 w_{2,1} + x_3 w_{3,1} + ... + x_n w_{n,1} \tag{2.1}$$

where $w_{i,j}$ represents weight on the link from x_i to h_j. The collection of all such weights in the network is called trainable parameters and finding the optimal values for them is the core task during the training of the model. After computing the weighted sum of inputs, the activation function is used to produce h_j^{out}, the output of neuron h_i. In the example such activation function is the sigmoid function so the computation is as follows:

$$h_i^{out} = \frac{1}{1 + e^{-h_i^{in}}} \tag{2.2}$$

In Figure 2.2, there is only one neuron in the output layer which follows the exact steps. In some cases, the activation in the output layer is omitted, which is equal to using a linear

activation function. To finish the computation, the output o of the model is computed as

$$o = \frac{1}{1 + e^{-(\sum_{h_i \in \{h_1, h_2, h_3\}} h_i w_i)}} \tag{2.3}$$

That is the last step of forward propagation in the network. We can use the output of the model and the expected value (training samples consist of inputs and expected outputs) and compute the error. For that we use a loss function L. The loss function is a key component in the training because it servers as a feedback for the model and for the estimation of how well it performs for each training sample. In order to decrease the loss of the model we need to tune the parameters of each of the operations in the forward propagation. In the example in Figure 2.2 the only trainable parameters are the weights of the linear combinations w_i. The optimization of the parameters of the model are done with the Backpropagation algorithm.

2.4.1 Backpropagation Algorithm

For an estimate of the error for a given training sample we need to know how to adjust individual parameters in order to lower the loss. Backpropagation takes this error value and computes its partial derivatives with respect to every parameter in the network. The value of the derivative for each weight shows how much does a weight contribute to the output of the network. Computation of backpropagation runs in the exact opposite direction as computation of the outputs of the network. Using the chain rule, we can reuse the computed derivatives that make the algorithm efficient as all the partial derivatives are computed in one pass. The backpropagation algorithm was one of the first ways of showing that neural networks are actually capable of learning non-trivial features. Until then, hand-crafting features was often the taken approach, which was limiting because of the time and computational power required to include more fields of problems. Now, it is commonly used together with a gradient descent type of algorithm to complete the whole training process of a neural network.

2.4.2 Stochastic Gradient Descent

The main goal is still optimization, meaning that we want to converge to the minimum possible error regarding the results of our network. One possible way to perform the right adjustments to the weights of our network, is to use an algorithm such as gradient descent. As described in [23], we pick a point in the weight space by initializing all the weights in our network. By computing the gradient with the backpropagation algorithm, we will move to a neighboring point, which is downhill and repeat until we converge to a minimum. A very important part of this process is the learning rate which determines

how large is the step taken. It can either be a constant or it can change overtime.

However, using the gradient descent method over the whole data set may be very costly, because data sets for neural networks tend to be large. One solution is to use stochastic gradient descent, which is probably one of the most used optimization algorithms when it comes to deep learning. The SGD works over mini-batches and not the whole data.

Algorithm 1: Stochastic Gradient Descent

---

1 Stochastic gradient descent update in time step k requires learning rate ϵ_k and initial parameters Θ on the input. The function f denotes computation done by neural network.

2 Until a stopping criterion is met, repeat:

1. Sample a mini-batch of data samples $\{x^1, x^2, ..., x^n\}$ and their corresponding labels $\{y^1, y^2, ..., y^n\}$

2. Compute the gradient

$$\hat{g} \leftarrow +\frac{1}{n}\nabla_\Theta \sum_i loss(f(x^i; \Theta), y^i)$$

3. Update the parameters

$$\Theta \leftarrow \Theta - \epsilon\hat{g}$$

Further development of optimizers focus on making the training faster, more reliable, and avoid situations when the algorithm finds local minimum. Detailed explanations of the methods and their empirical comparison can be found in [24]. For training our models we use the Adam optimizer [25].

Capabilities of such models have been thoroughly examined. The Universal Approximation Theorem states that feed forward neural network with a single hidden layer of finite number of neurons is capable of approximating continuous functions under mild assumptions on the activation function. Leshno et al. [26] showed that a multilayer feedforward network with a locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy if and only if the activation function is not polynomial.

2.4.3 Recurrent Neural Networks

In some domains, such as time series prediction, sequence classification or text processing, there is an implicit sequence of data samples. Recurrent Neural Networks (RNNs) are architectures which are designed to process sequential data and learn the underlying dependencies. The building block of RNN is called a recurrent cell. In theory, recurrent neural networks can process sequences or arbitrary length.

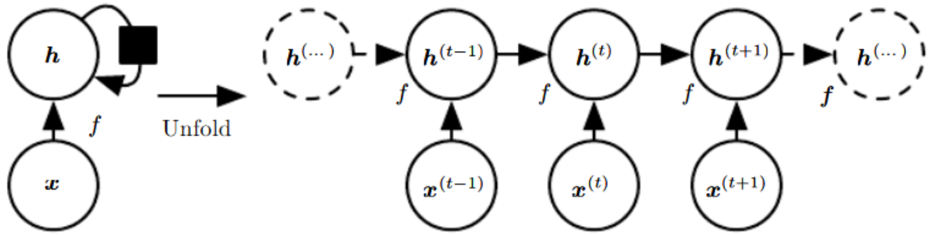

Figure 2.3 | **Basic recurrent cell and its unfolding.** [22]

In Figure 2.3 we can see an example of the basic type of recurrent cell. In each step of the sequence, the cell has two inputs:

1. x^i: features of the current step of the sequence
2. $h^{(i-1)}$:output of the cell in a previous step of the sequence

When the cell is unfolded (also known as unroll), it can be see the how the information flow as the sequence is being processed. Based on the task, either the concatenation of hidden states from each step is used as an output or just the last output of the sequence. Since the cell uses the same weights and biases for each step of the sequence, we can see the process as learning what information to store in memory. In practice, however, simple RNN cells tend to struggle with learning long-term dependencies as they keep "overwriting" the memory with incoming data. There are two extensions of the RNN cell architecture designed to combat this problem.

Long-Term Short Memory cells

Long Short-Term Memory (LSTM) [27] cells, are an extension of simple RNN cells designed to capture long-term dependencies in the data. There are three non-linear functions called *gates* added to the cell as shown in Figure 2.4.

Each of the gates has the same inputs: previous hidden state and current feature vector. Such architecture enables controlled forgetting in the training process. The downside of this approach is that the amount of parameters for training is much higher.

Gated Recurrent Units

Gated recurrent units (GRU) were first proposed in 2014 in [28] as a tool for neural translation in a sequence-to-sequence manner. In the diagram shown in 2.5it can be seen that unlike LSTMs, GRU cells don't have an output gate which means they have less parameters to train. However, it also means that there is less control over the forgetting. In [30] it has been shown that LSTMs consistently outperform GRU cells in Natural Language Processing tasks.

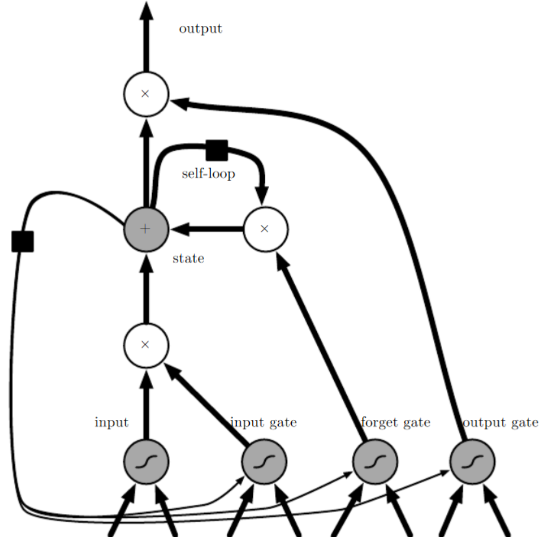

Figure 2.4 | **Block diagram of LSTM cell.** [22]

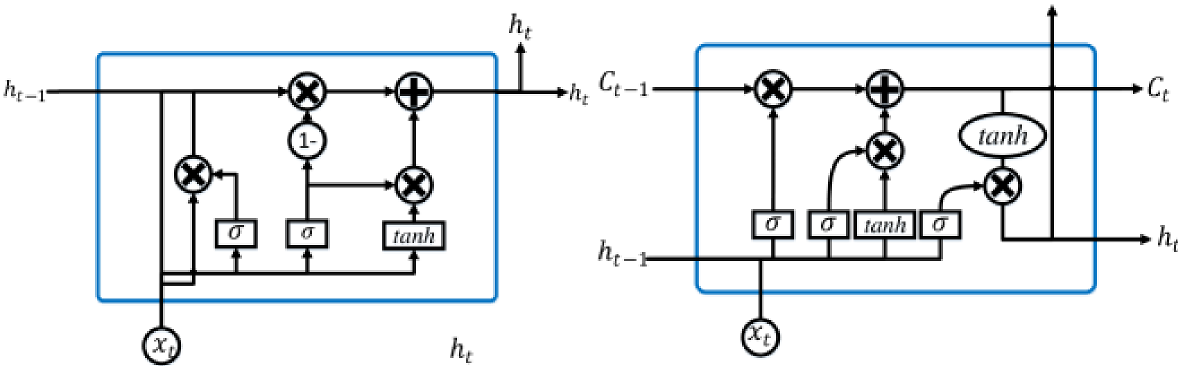

Figure 2.5 | **Comparison of GRU and LSTM cells.** Diagrams showing different gates in Gated Recurrent Unit(left) and Long-Term Short Memory (right) cells [29]

Bidirectional RNN Cells

In some domains, such as text processing, elements of the sequence are dependant not only in the previous elements but also on those coming after. Bidirectional RNNs take this into account and process the sequence in both directions combining or concatenating the outputs. Results from processing the sequence in either direction are either stacked or aggregated. Commonly, simple aggregation functions such as sum or max are used, but in theory, any differentiable aggregation can be applied for this task.

2.4.4 Graph Neural Networks

Graph structures are commonly used for representation of data in various domains. Inspired by the the success of convolutional neural nets in image processing, there have been attempts in recent years to use the same concept on graphs. Unlike images, where the

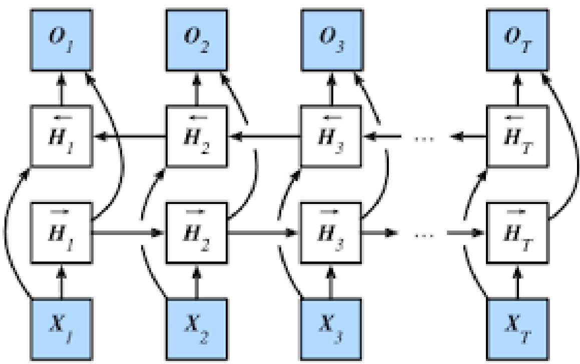

Figure 2.6 | **Unfolding of a bidirectional RNN.** [31]

neighborhood of a pixel is defined by a grid of fixed size, in graphs we use the edges to gather the information from the node's neighbours in a similar manner. Such methods were very successful in a variety of tasks from node classification to graph modeling.

2.4.5 Deep Sets

Traditional approaches for training neural networks account for fixed dimensions of feature vectors. Image processing can be listed as an example of a domain that is suitable for such an approach. In recent years, there have been several attempts to extend machine learning tasks to sets of samples. Pevný and Somol [32] showed that the underlining tree structure of the data can be used for aggregation of partial estimators to perform classification tasks on the complete structure. Similarly, Zaheeer et al. [33] proposed a framework for inference over a set of objects which outperform the approach using recurrent networks. Both teams use similar techniques to process a set (bag) of samples: Firstly, each item in a set is embedded in a fix size vector space using a estimator ϕ. Next, all embedded items are aggregated using a sum function and fed into a second estimator ρ. Such approach is applicable to sets with arbitrary sizes and element ordering.

Battagalia et al. [34] showed that learning from a set of samples can be viewed as a special case of graph learning (considering a set to be a graph without edges) and that Graph Neural Networks are applicable for such a task.

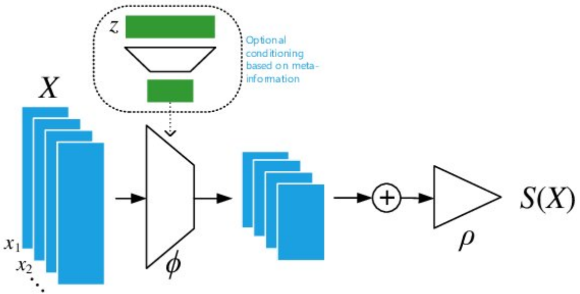

Figure 2.7 | **Deep Sets Architecture - Invariant.** [33]

2.4.6 Generative Models

Neural networks are commonly used for two types of tasks: classification of samples and regression. With advances in image and text processing, another use-case for such models became popular: generation of new data with similar properties as those of the training samples [35]. There are two popular directions in the development of generative models: Generative adversarial networks (GANs) and autoencoders.

Generative Adversarial Networks

The first approach uses Game Theory to find the equilibrium state of a system of two models which compete against each other. The first model, called the generator, attempts to generate a sample. The second model, called the discriminator, takes as input a mixture of real data points and generated samples and attempts to classify them as either real or fake. Using each others outputs, both model update their parameters until convergence.

Autoencoders

The second approach to generative models also uses two separate models, but in a very different fashion. Training samples from $\mathbb{R}^n$ are embedded (encoded) in a fixed size latent space $\mathbb{R}^m$ $m < n$ by a first model called an Encoder. Afterwards, the second model, called a Decoder, attempts to reconstruct the samples in the original space $\mathbb{R}^n$. Since the dimensionality of the latent space is lower than in the case of the real data, there is implicit information loss. Encoder-Decoder architectures are optimized to minimize the information loss of the process.

2.5 Honeypots

A honeypot is a system which is set up as a decoy to lure attackers and to detect and study attempts to gain unauthorized access to information systems. By definition, no legitimate user should ever interact with a honeypot, therefore anyone who attempts to connect or interact with it is considered an attacker.

As with any other defense system there are pros and cons when deploying honeypots. The main benefit of a honeypot is that it allows researchers and security professionals to collect real data from actual attacks and unauthorized activities in the network. Such information provides insight about the course of the attack, tools used, and all together allows for designing better defense mechanisms. Another aspect is that since there is no interaction from legitimate users, the false positive rate is significantly reduced. Deploying honeypots is also cost-effective: in contrast with the majority of intrusion detection

systems, with honeypots there is no need to analyze large volumes of network traffic which reduces hardware demands. It is common to use virtual machines as a honeypots.

There are a couple of disadvantages of honeypot usage which often discourage system administrators from using the technology. The most pressing question is about the security of a honeypot: If it allows a lot of interaction with a potential attacker, there is a risk of misconfiguration, or even errors in the honeypot system which can lead to security breaches.. There is a trade-off between the distinguishability of a honeypot and the amount of interaction it provides. Especially in setups where the attacker suspects a presence of honeypots in the network, one has to pay a lot of attention to make them look real and important enough to attract the attacker. Last, but not least, is the issue of analysis of the collected data. Honeypots, which are freely accessible in the Internet can generate big volumes of data mainly because of the background "noise" which is present in the network. In general, honeypots are not a substitution for intrusion detection systems, but can be a valuable addition to a setup that provides more information about the attacks. In the following section we present a brief classification of honeypots.

2.5.1 Types of Honeypots

Honeypots can be split into several groups. The most common classification is based on the level of interaction the system allows: pure, high-interaction and low-interaction.

Pure Honeypots

A pure honeypot is a full-fledged production system which has been assigned as a bait. To monitor the attacker's actions additional software needs to be installed. While a pure honeypot may be useful when the defence mechanisms are required to be exorbitantly stealthy, a more controlled environment is usually desirable [36].

High-interaction Honeypots

High Interaction honeypots make use of the actual vulnerable service or software. High-interaction honeypots are usually complex solutions as they involve real operating systems and applications. In High Interaction honeypots nothing is emulated - everything is real. High Interaction honeypots provide a far more detailed information of how an attack or intrusion progresses, or how a particular malware executes in real-time. Since there is no emulated service, High Interaction honeypots help in identifying unknown vulnerabilities. However, they are more prone to infections and increased risk because attackers can use these real honeypot operating systems to attack and compromise production systems.

Low-interaction Honeypots

When adversaries exploit a high-interaction honeypot, they gain capabilities to install new software and modify the operating system. This is not the case with a low-interaction honeypot. A low-interaction honeypot provides only limited access to the operating system. By design, it is not meant to represent a fully featured operating system and usually cannot be completely exploited. As a result, a low-interaction honeypot is not well suited for capturing zero-day exploits. Instead, it can be used to detect known exploits and measure how often a network gets attacked. The term low-interaction implies that an adversary interacts with a simulated environment that tries to deceive him to some degree but does not constitute a fully fledged system. A low-interaction honeypot often simulates a limited number of network services and implements just enough of the Internet protocols, usually TCP and IP, to allow interaction with the adversary and make them believe they are connecting to a real system.

Chapter 3

Previous Work

In this chapter we examine state of the art techniques for graph processing with ML methods, as well as tools for honeypot generation and deployment. We also look at using ML methods in honeypot deployment with special focus on generative models. Additionally, we present some of the existing honeypot generation and management tools for active directory.

The automatic generation of data for Active Directory is not a very common topic in the industry. This is because most companies already have a structure with users and groups, and it does not make sense to generate data automatically. The only situations where this may be needed are during testing of AD deployments or for fake AD setups for simulations. Another situation where this may be needed is for automatic honeypot generation, which is the topic of this thesis. As far as we know, there is no academic research on the automatic generation of AD structures. However, there are some tools for this task which require human interaction and can be later used for managing existing honeypots.

The main tool for detecting malicious activities in an Active Directory is the Advanced Threat Analytics by Microsoft [37]. It is a complex tool which monitors all the traffic of a domain controller and uses the data for detection of known attacks such as password bruteforcing, Pass-the-Hash, Malicious replications, etc. Additionally, it detects abnormal activity in the domain and reports results to the system administrator. It also contains modules for detecting weak points in security such as shared passwords, broken trust and known protocol vulnerabilities. BlueHive [38] is a Honeypot user management tool. It is used for manual creation, management and monitoring of fake users in an Active Directory. It can track the history of actions for registered user accounts and provides automatic updates for lastLogOn attribute and other attributes which change in time. Static values in such attributes show that the account In areas outside the AD domain, there are some attempts to use automation and machine learning methods to design

honeypots or emulate responses of certain devices or services. Leita et al. [39] proposed to use state machines to generate scripts for creating honeypots for the popular open-source honeypot manager honeyd [40]. Downling et al. proposed to use Reinforcement Learning for generation of honeypot responses in order to extend the duration of the attacker's session. In particular, the authors worked with a Cowrie honeypot which is based on emulation of SSH, Telnet and several other protocols [41] .

The problem of placing a honeypot or honeytokens in such a way as to attract the most attackers has been a point of interest for Game Theory (GT) researchers. In the GT approach, the interaction between the attacker and the system administrator is modeled as a two player game and the goal is to find an optimal strategy for either player [42].

Advances in machine learning methods in various domains of the last decade influenced progress in the area of graph processing. Graphs in general are universally used across various domains of computer science and therefore applying the methods of ML on them has become a hot research topic in recent years. Wu et al. [43] provided a detailed study of various ML methods for graph processing, along with an evaluation of performance for different models and graph classes. One of the first papers on generative graphs models was GraphRNN [44]. The graph in GraphRNN is generated iteratively using two recurrent modules, one on the node level and another on the graph level. The authors show that GraphRNN outperforms methods based on Graph Convolutional Networks in the task of generating realistic looking undirected graphs. In addition, the GraphRNN method can scale up to structures of hundreds of nodes.

In the area of generative models for graphs, Simonovsky and Komodakis successfully used Variational Autoencoders to generate small undirected graphs representing molecules. The method, however, lacks scaling capabilities and was not able to capture complex interactions in larger molecules [45]. Usage of Graph Recurrent Neural Networks proposed in [46] showed great success in modeling protein data with results exceeding both GrapVAE and GraphRNN by combining the GNN with attention layers. All the above mentioned methods work with undirected graphs.

Taking directed edges in the graph into account, [47] proposed to use custom RNN cells to analyze a DAG structure of Logical formulas and train the model to directly solve the SA Logical Formula Satisfiability (SAT) problem. Results show that the proposed model is able to learn the structural information of the graph using a sequential propagation of DAG structured data. Further advancing the proposed idea, Kaluza et al. [48] proposed a framework for DAG to DAG Translation inspired by Sequence-to-Sequence processing in the Natural Language Processing domain. The framework is based on the Encoder-Decoder architecture using similar modules as in [47] in the encoder part of the model. The proposed framework is evaluated on the task of simplification of logical formulas

where it shows promising results on a dataset with limited graph size. The scaling ability of the proposed framework is yet to be investigated.

There are a number of libraries implementing models of graph neural networks. DeepGraphLibrary [49] is an open-source library for efficient graph storing and pre-processing, mostly built for the PyTorch framework. It is well integrated with NetworkX. Spektral [50] is a Tensorflow related library implementing mostly Graph Convolutional and Polling layers. An even more complex library introduced by DeepMind in [34] is based on Tensorflow and Sonnet frameworks.

To our best knowledge, the amount of publications using generative models for honeypot design is limited. There is a very recent work on using generative models in honeypots, mainly the NeuralPot [51], where authors propose to use Generative Adversarial Networks or AutoEncoders to generate the network traffic of the industrial Modbus honeypot. Despite the low complexity of the model it shows that it is able to generate traffic that closely resembles the original protocol.

There are several tools which are being used by red team members and attackers. Bloodhound [16] is an example of a software designed to simplify the task of AD Reconnaissance and taking over of a domain. It is based on Powershell [17] and neo4j database [18] and is used to visualize the AD structure and to find potential attack vectors for dominating the domain. Similarly, the ADRecon tool by Sense of Security [52] uses the LDAP protocol to gather information about the domain and its components. Both tools require valid credentials of a domain user with no additional privileges.

Finally, it is worth mentioning some tools which are specifically designed to detect honeypots. An example of such tool is Honeypot Buster developed by Javelin [53]. It analyses the attributes of the objects in a AD looking for missing, repeated or inconsistent entries and reporting it to the user.

Chapter 4

AD Structure Modelling

In this chapter we describe the proposed framework for placing honeyusers in the existing Active Directory structure. In its simplest form, AD is a Tree structure for a single domain or a Forest for the cases of multiple domains. When we take into account the security group membership and relations provided by the Global Policy Objects, the resulting structure forms a Directed Acyclic Graph (DAG).We consider all edges to be the same and therefore we don't use any edge features in the framework. Such a graph has exactly one starting node representing the Domain. The goal of the proposed framework is to process the whole graph structure and predict a location (in this context all edges) for additional nodes which are used as honeytokens. Finding a meaningful location for the honeypot reduces the chance of its detection.

In contrast with most of the existing graph processing ML frameworks which were developed for general graphs, the proposed framework utilizes the properties of the DAG.One of the properties of a DAG is that there exists a topological ordering of the vertices. In general, such ordering is not unique unless there is a directed path in the graph which contains all vertices. With such ordering, we can make a sequence of nodes from graph G which guarantees that for any node v_i in the sequence **all** of its predecessors have been processed in timestamps $t < i$.

Subsequently, every node v_i can be defined by a sequence of its predecessors starting with the root node. Each node in the graph is defined by a combination of the features of the node and the sequence of predecessors.

The task of adding a new node to the existing structure consists of two actions: Finding the features describing the node itself and finding a sequence of its predecessors. Given the fact that the existing structure must not be changed in the process, it means evaluating all nodes in the existing graphs and deciding which of them should be direct predecessors of the node appended to the structure. In other words, the framework is finding a missing links between the nodes in that are already in the structure and the new node.

To the best of our knowledge there is no Tensorflow 2 compatible library with implementation of DAG Recurrent Neural Network and its extensions. In this thesis we created set of modules capable of processing arbitrary data in structured as DAG which are compatible with Tensorflow 2 framework and its GPU acceleration. All models designed in this thesis is free software.

4.1 Notation and Definitions

Let $G = \langle \mathcal{V}_G, \mathcal{E}_G \rangle$ denote a Directed Acyclic Graph. We assume that $\mathcal{V}_G$, the set of nodes of G is ordered according to the topological sort of the DAG. Let $\pi_G(v), v \in \mathcal{V}_G$ be a set of direct predecessors of v in G. $\delta^-(v_i)$ represents the in-degree of the node v_i which is the number of incoming edges.

Additionally, for a given DAG G, let G^r be a *reversed* DAG with the same set of nodes and reversed edges. Using the same topological sort, nodes of G^r are in reversed order of the sorted nodes of G. Lastly, for a given DAG G, we define $\mu_G : \mathcal{V}_G \to \mathbb{R}$ to be a d-dimensional vector function defined on nodes of G. We call μ_G encoding function and use it in all proposed models.

As for the inputs and outputs of the models, $X \in \{0,1\}^{n \times f}$ is used to represent the original feature matrix of the model using one-hot encoding. n represents the number of nodes and f the size of the feature vector.$A \in \{0,1\}^{n \times n}$ is the adjacency matrix of the original graph. m represents the number of new nodes and $X' \in \{0,1\}^{m \times f}$ is the feature matrix of the new nodes.

The$\hat{}$ symbol is used for estimated values, in particular $\hat{A}$ for the generated adjacency matrix and $\hat{X}$ generated feature matrix, in cases where the model is outputting one. $\mathbf{h} = [h_0, ..., h_i]$ represents hidden states of nodes. When working with a hidden state of a single node we use $[h_i$, while $\mathbf{h}$ is used for operations with the list of the **all** hidden states. $\mathbf{H} = [H_0, ..., H_i]$ represents the hidden states of the whole graph after processing node i.

4.2 General Description of the Framework

In contrast to other graph generation tasks, in our domain we are not generating the whole graph as the original AD structure has to remain unchanged. Therefore, the main task is to generate a extension of the existing graph with properties as close to the original as possible. The key of successful deployment of the honeypot is in disguising it properly so it is not recognisable for the attacker. It is even more important in scenarios when the adversary expects the presence of the honeypot. We assume that proposed models are capable of generating node placements to meet such criteria while remaining worthy for

the adversary to interact with.

In the generation process, machine learning is utilized in the structural part of the task. For the node feature generation which consists of generating attributes independent on the position of the node in the graph we use external tools. The process is described in 4.7.

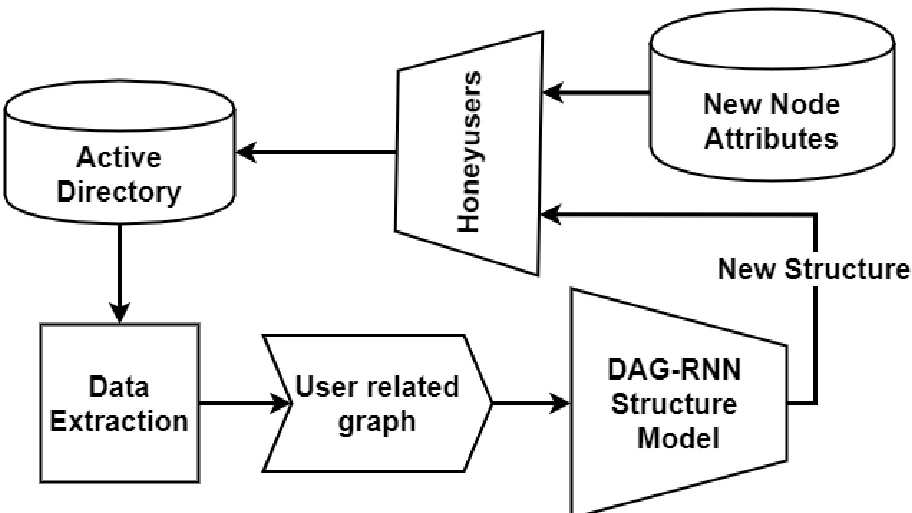

Figure 4.1 | **General concept of the framework.** Sequence of steps for extending Active Directory with GNN generated Honeyusers

The framework pipeline shown in Figure 4.1 starts with extracting the existing structure from the AD. For that, either the Sharphound tool[54] can be used or a collection of Powershell utilities. Since the Active Directory is much richer than the graph structure, the next step is processing of the raw data in order to get the graph of user-related nodes and dependencies. We refer to the graph as User Related Graph. The data in the graph is later represented in two input matrices for the ML model: A which is is the binary matrix representing input edges to each node and X which is the one-hot encoded matrix of node types. The machine learning model predicts the incoming edges for the new nodes which, together with A, form $\hat{A}$. The part of $\hat{A}$ which contains the original graph must remain intact because any changes in this sub-graph would influence the functionality of the AD. Using the extended graph, node features for the honeyusers can be generated. Since some of the features, such as membership in organizational units depend on the position in the graph, they cannot be generated beforehand. The last part of the pipeline is the actual addition of the newly generated objects in the structure of AD. Powershell cmdlets or LDAP addition queries are used at this point. If any anomaly detection tool or honeypot manager is used in the system we can register the newly added users for proper reporting and alerting.

4.3 Components Design in the Generic Model

This section is focused on the core of the framework; the machine learning model that processes the original structure and generates the extended structure.

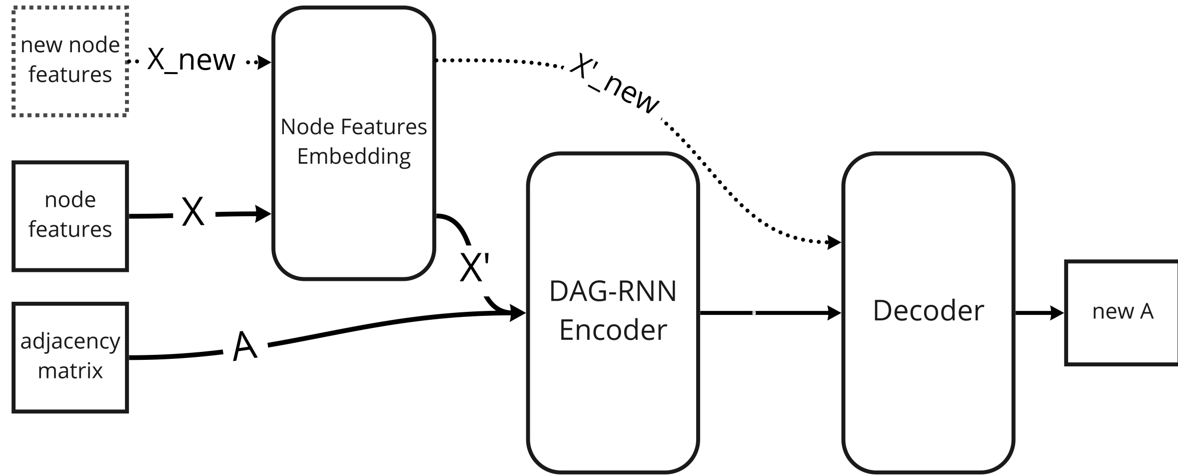

Figure 4.2 | **Design of the generic model.** The different components of the generic model used for generating Honeyusers.

In Figure 4.2 we can see the main components of the model. The leftmost nodes are the inputs: the matrix $A \in \{0,1\}^{n\times n}$, matrix $X \in \{0,1\}^{n\times f}$ and matrix $X_{new} \in \{0,1\}^{m\times f}$ where n is the number of nodes in the original structure, f is the number of object types, and m the number of new nodes to be added to the structure. The matrix A represents the structure of the graph because it has the relationships between nodes (edges). It is a lower triangular matrix due to the directed edges in the graph and the topological ordering $\mathcal{O}$. Matrices X and X_{new} are one-hot encoded feature vectors of the nodes of the original structure and the new nodes which should be inserted in the graph. The matrix X_{new} is denoted as a dotted line because in later variants (Model3) it is not used.

The first component in all versions of the model is the node embedding layer. When we consider the space of one-hot encoded features the distance between any two node types is exactly 1. In reality however, some node types are more similar than others. For example, distances between node types Computer and User should be smaller than the distance between User and Domain because of the properties of either class. The node embedding layer is capable of learning the parameter mapping from the original space to continuous vectors. In all of our models, the embedding layer is an off-the-shelf model from the Tensorflow/Keras library. Using feature embedding has proven to be a powerful tool for enhancing the models capabilities in various domains (such as in word2vec [55], and node2vec [56]). The embedded node features together with the adjacency

matrix are processed by the encoder, described in detail in Subsection 4.3.1. Using either the node-level encoding or the graph-level encoding output of the encoder we use another module to reconstruct the original structure with added nodes. We call the last module in the model the Decoder. The architecture of the decoder module, and subsequently the way using the output of the encoder is where the models proposed in this thesis differ from the work in [47]. We tested three variants of the model:

Model 1 (Section 4.4) attempts to directly predict the edges to the new nodes using the exiting node encoding and the candidate node type.

Model 2 (Section 4.5), uses the hidden state of the complete graph as additional source of information for the edge prediction in the decoder.

Model 3 (Section 4.6) uses a Variational Auto Encoder to regularize the latent space of the node representation as a probability distribution and sample the new node representation from the distribution.

4.3.1 DAG-RNN Encoder

The DAG-RNN Encoder is a model for encoding either a graph or its individual nodes of any Directed Acyclic graph. It is the core part of all of the models used in this thesis.

The primary motivation for the Encoder is to capture the information from the nodes and all relations among them in a vector of fixed size. The encoding can be used for a wide variety of tasks such as node clustering, node classification, graph classification, edge prediction or even graph generation as we demonstrate later in the thesis. The motivation for finding such embedding space is twofold: Firstly, it allows to have a fixed the size representation of combined node features and its relations within the graph, and using the representation in models which are usually designed for an input of fixed size. Secondly, studying properties of the latent space allow better control of the properties of generated artificial samples.

4.3.2 DAG Recurrent Layer

For the DAG-RNN we created our own DAG recurrent layer. This new type of layer was needed to deal with the details and complexities of the DAG. This subsection explain its components, structure, computation and training process, as well as possible extensions.

The concept of a Recurrent Layer designed to process Directed Acyclic Graphs shares the concept of standard RNNs. We have shown that in every DAG there exits a topological ordering $\mathcal{O}$ which allows processing the nodes sequentially. In a regular RNN cell there are two inputs: the feature vector of the current element in the sequence and the previous state of the cell which is the output of the cell after processing the previous element in the

sequence. The DAG-RNN is an extension of this idea. For the input feature vector at each timestamp, we use the features of the nodes under the ordering $\mathcal{O}$. In the previous chapter we showed that unlike normal time series or text processing, in the case of DAG, there can be a situation when two nodes share only one node in their sequence of predecessors (the first node of the sequence), yet they can be following each other in the ordering $\mathcal{O}$. Figure 4.3 shows an example of such a setup.

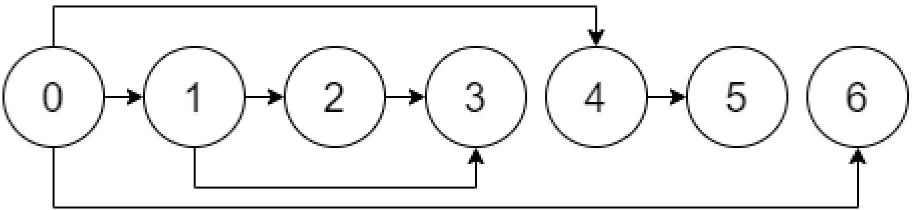

Figure 4.3 | **DAG Ordering Example.** A Simple Directed Acyclic Graph example where node 4 is following node 3 in the topological ordering $\mathcal{O}$ despite sharing only one predecessor.

We can see that node 4 should be processed right after node 3, but using only the output of the previous timestamp would mean to completely ignore the topological structure of the graph. The desired behavior is to use the outputs of the the direct predecessors of node 4 which is output of node 0.

When processing node 3 there are multiple direct predecessors which means that all their outputs have to be aggregated. There a is variety of functions that can be used for the aggregation of a set predecessors of node v, which we previously defined as $\pi_G(v)$. In theory any differentiable function can be used. In this work we are using two methods: sum function and DeepSets which are described in Subsection 2.4.5. Predecessors $\pi_G(v)$ are gathered using the adjacency matrix A which is one of the inputs for the model. We assume that all processed graphs are valid structures which means they have exactly one starting node. It also implies that every node in the graph has except fro the starting node has at least one predecessor.

With the aggregation of previous states, we can use regular RNN cells to process the sequence. Following the ideas from [47] we use Gated Recurrent Units for the task. The complete DAG-RNN layer structure is shown in the Figure 4.4: Input a_t is the part of the matrix A which contains the incoming edges to the node v_t. Input x_t is an output of the node feature embedding layer for the node v_t. Instead of using the previous output of the RNN cell, the aggregation of the set of hidden states of the direct predecessors $\{h_i, i \in \pi_G(v_t)\}$ is used as previous state for the RNN cell. In the

The RNN cell in the layer has no adjustments in the architecture. We use the GRU cell which has two gates: the update gate z_t which decides how much of the new input to store and the reset gate r_t which decides which part of the memory to erase. The

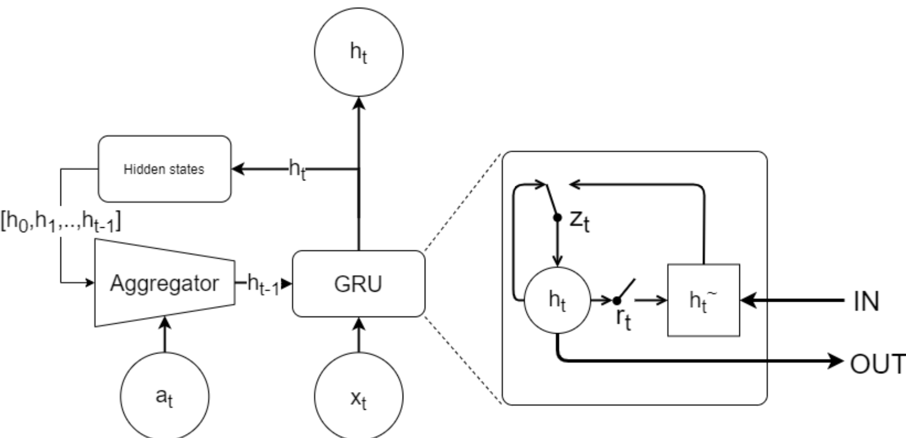

Figure 4.4 | **Details of the structure of our special DAG-RNN layer, created for this thesis..**

GRU cell is used instead of an LSTM to reduce the amount of parameters in the training process.

There are two possible outputs of the DAG-RNN layer depending on the configuration. The first option is to just output a sequence of hidden states $\mathbf{h} = [h_0, h_1, ..., h_n]$. The second option is to aggregate all of the hidden states $h_{i \leq t}$ at the timestamp t creating a representation of the complete sub-graph containing nodes $v_0, v_1, ..., v_t$. We previously defined such graph representation as H_i. Arbitrary aggregation function such as sum or mean can be used for combining $[h_0, h_1, ..., h_i]$ into H_i.

The DAG-RNN can be used as a building block for more complex models. In order to capture better all the relations in the structure.

4.3.3 Bi-directional DAG Recurrent Layer

One of the most common techniques when using RNNs is to process the sequence in both directions. It has proven to be successful in text processing tasks to capture the context of a sentence. Following this idea we also implemented our own bi-directional DAG-RNN.

Our DAG-RNN can work in a bi-directional setup, using one instance of DAG-RNN for each direction of the edges. Transposing the adjacency matrix reverses the edge directions in the graph and with the reversed ordering we can apply the same operations to reversed graph G^r.

Bidirectional RNNs produce one output for each of the directions the sequence is processed in. We can either use the outputs independently or combine them to obtain a more detail representation of each element in the sequence. Commonly, the two outputs are concatenated producing a representation of 2∗rnn cell dimension or combined using a simple aggregation function such as sum, mean or max to keep the dimensionality of the layer output same as in the unidirectional RNN. In our implementation, sum of the

outputs is the default option. Bi-directional DAG-RNN is used as an encoder in Model 3(4.6).

4.3.4 Loss Function

To estimate how correct the model is and subsequently to train the model we need to define a loss function. This subsection defines a loss function that is used in all our model types. Moreover, Models 2 and 3 use additional components of the loss functions which are described in their respective sections of the thesis.

Since the goal for the model to reconstruct the original matrix A, we call this function the *reconstruction loss.*

Binary Cross Entropy

We consider the presenceor absence of an edge to be a binary classification problem. A commonly used loss function for binary classification tasks is called Binary Cross-Entropy and it is defined as follows:

$$BCE(p, y) = \begin{cases} -log(p) & \text{if } y = 1 \\ -log(1-p) & \text{otherwise} \end{cases} \tag{4.1}$$

where p are the predicted values and y the ground truth values. We can rewrite the formula using parameter p_t

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{otherwise} \end{cases} \tag{4.2}$$

as

$$BCE(p, y) = BCE(p_t) = -log(p_t) \tag{4.3}$$

We can see that misclassification of either class is treated the same. This is well suited for domains where both classes are balanced. In the case of the adjacency matrix this assumption is not met (the reason why is described in detail in Chapter 5). The majority of edges are not present and therefore the prior probability for the classes is skewed towards the no-edge option.

Focal Sigmoid Loss

To mitigate the issue of imbalance in the target classes, we use a modification of binary cross-entropy called Sigmoid Focal loss which was proposed by the Facebook AI Research

(FAIR) team in [57]. The Focal loss (FL) advances the idea of weighted cross-entropy defined as:

$$BCE(p_t) = -\alpha_t log(p_t) \tag{4.4}$$

where $\alpha \in [0, 1]$ for class 1 and $1 - \alpha$ for class -1.

"While the parameter α balances the importance of positive/negative examples, it does not differentiate between easy/hard examples" Lin, Goyal, Girshick, *et al.* [57]. The proposed method adds a modulating factor $(1-pt)^\gamma$ to the cross entropy loss where $\gamma \geq 0$. Then Focal loss is defined as:

$$FL(p_t) = -(1 - p_t)^\gamma log(p_t) \tag{4.5}$$

It accounts for high imbalance in the classes and subsequently makes Type II errors more severe. It other words, we consider that not adding an edge in the place where it is expected is more severe than predicting an extra edge. Focal loss was originally used in computer vision problems where an object's presence in the background or foreground of an image is predicted. In practice, the FAIR team uses the α balanced version of the focal loss as it showed slightly better results in their experiments:

$$FL(p_t) = -\alpha_t(1 - p_t)^\gamma log(p_t) \tag{4.6}$$

4.3.5 Implementation Details

To the best of our knowledge there is no Python based Tensorflow 2 compatible DAG-RNN framework. The Graph Nets library [34] contains tools which simplify the building of such models but requires using the in-house library Sonnet. Our implementation is based purely on Tensorflow 2 and Keras frameworks and it is fully compatible with standard Tensorflow pipelines. Since it uses the Tensorflow backend for all the computation it is usable with CUDA GPUs which significantly speed up the computations.

In the aspect of scalability of the implementation we are aware of sub-optimal memory usage. The implementation works with the adjacency matrix A represented as a dense matrix. But due to the ordering $\mathcal{O}_\mathcal{G}$, the matrix is lower triangular. One possible future improvement is to utilize the concept of Sparse Tensors introduced in Tensorflow 2 which allows a more efficient representation and computation.

The time complexity of the encoder is $\mathcal{O}(N)$. Since the nodes are processed sequentially according to the ordering, before processing node v the model needs to wait until all nodes $u \in \pi_G(v)$ are processed. The decoding part of the framework works in $\mathcal{O}(N^2)$ time as each pair of nodes (u, v), where u is a node from the original graph and v new node, is evaluated.

For the graph pre-processing we use numpy [58] and networkx[59] libraries for numerical and graph operations respectively. All tools used in this thesis are free software and accessible at `https://github.com/stratosphereips/AD-Honeypot`.

4.3.6 Model Training

Models are trained using a single computer with 32 GB of RAM and Nvidia Titan V GPU[1] card with 12 GB of memory. The artificial datasets described in Chapter 5 were used for training of all model instances. In all cases the Adam [25] optimizer was used for the training with exponentially decayed learning rate for fine tuning of the parameters towards the end of the training period. All the models use the Encoder with 64 units in the RNN cells and same MLP module for the edge prediction. Weights in the models are initialized using the Glorot uniform initializer [60] with the exception of the $z_{\text{log variance}}$ Dense layer in Model 3 where weights are initialized to 0.

4.4 Model 1: Direct Edge Prediction

Model 1 is designed to predict the edges between x_i, the existing nodes in the structure and the new nodes $x`_j$. During training we attempt to reconstruct the existing adjacency matrix using the node embedding generated by the DAG encoder. In other words, we aim to learn the probability of adding an edge from a exiting node encoded in the latent space h_i to a candidate node with known type x'_j.

4.4.1 Model Components

Inputs for the models are matrix X which contains one-hot encoded node features of the existing graph, the adjacency matrix A of the existing graph and matrix X' which consists of one-hot encoded node types of target nodes. Model 1 has a single output: the estimated adjacency matrix $\hat{A}$.

The first part of the model is the node embedding layer which transforms the one hot encoded node types into a continuous vector of a set dimension. The embedding dimension is one of the hyper-parameters of the model.

Figure 4.5 shows the architecture of Model 1. Using each pair of $(h_i, x'_j) \in \mathbf{h} \times \mathbf{x'}$ the decoder performs a binary classification task with a MLP producing a probability of

[1]Provided by NVIDIA as part of Higher Education and Research Grant

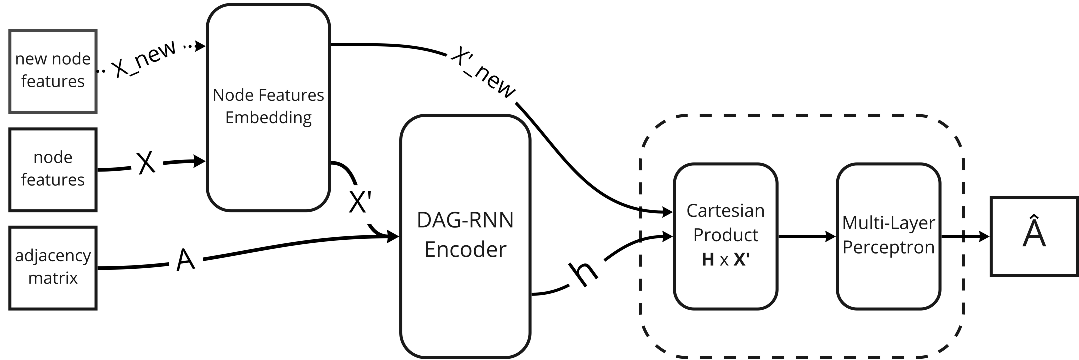

Figure 4.5 | **Decoder in Model 1.** Detailed architecture of the decoder part of Model 1. The decoder performs a binary classification task with an MLP that outputs the probability of adding an edge.

adding an edge. The output matrix $\hat{A}$ elements are the individual outputs of the decoder:

$$\hat{A} = \begin{bmatrix} D(h_0, x'_0) & \cdots & D(h_0, x'_m) \\ \vdots & \ddots & \vdots \\ D(h_n, x'_0) & \cdots & D(h_n, x'_m) \end{bmatrix}$$

Where $D(h_i, x'_j) \in [0, 1]$ is the output of the Decoder and a probability estimate of edge presence from node i to node j.

Training and Generation Process

During the training process we encode the existing structure of the graph to get the node hidden states $\mathbf{h} = [h_0, ..., h_i]$. Using the embedding layer and the original features of the nodes X the we obtain the embedding of the input node features $x' = [x'_0, ..., x'_i]$. Afterwards, pairs of (h_i, x'_j) are classified with the MLP forming predicted adjacency matrix $\hat{A}$.

During the generation, we use the node types of the candidate nodes as input X' instead of the original X. Focal Loss function showed in Subsection 4.3.4 is used for the model training.

4.5 Model 2: DAG-RNN AutoEncoder

The motivation for this model in comparison to the previous architecture is that it takes full advantage of the latent space encoding. Experiments using the Model 1 as an Autoencoder where pairs of hidden states (h_i, h_j) are used for predicting edges from i to j showed promising results even with small dimensionality of the latent space (See A.1.1). In the Model 1, the candidate node is represented only by its type. That leads to more precise

prediction in the nodes in the beginning of the node sequence. However, experiments with larger structures show that the simplicity of the architecture of Model 1 results in very poor performance in the graph structure reconstruction tasks.

In order to improve the quality of the reconstructed matrix $\hat{A}$, additional information for predicting the edge to the candidate node from nodes in the original structure is required. In Model 2 we propose to add context information with the graph level hidden state H_t. Which is computed as aggregation of node hidden states $h_0, h_1, ..., h_t$ where t is index of currently processed node. Edge prediction is performed on using the $(h_i, \hat{h_j})$ where $\hat{h_j} = [x'_j, H_i]$. Inputs and outputs for the model remain same as in the Model 1: matrices X, A, X' are inputs of the model and matrix $\hat{A}$ is the only output.

4.5.1 Model Components

Encoding of the original graph remains similar to the case of Model 1 with one extension: Apart from the hidden state of the nodes$\mathbf{h} = [h_0, h_1, ..., h_n]$, the encoder also produces hidden states of the graph after processing the node i $\mathbf{H} = [H_0, H_1, ..., H_n]$

In the decoder part of the model, there are two steps (and subsequently two MLP modules to perform the operations): Firstly, estimating $\hat{\mathbf{h}} = [\hat{h}_0, \hat{h}_1, ..., \hat{h}_n]$ using the concatenation of $[H_i, x'_j]$ and secondly, predicting the edge for each pair $(h_i, \hat{h}_j)$ forming the estimated adjacency matrix $\hat{A}$. The whole architecture of Model 2 is shown in Figure 4.6.

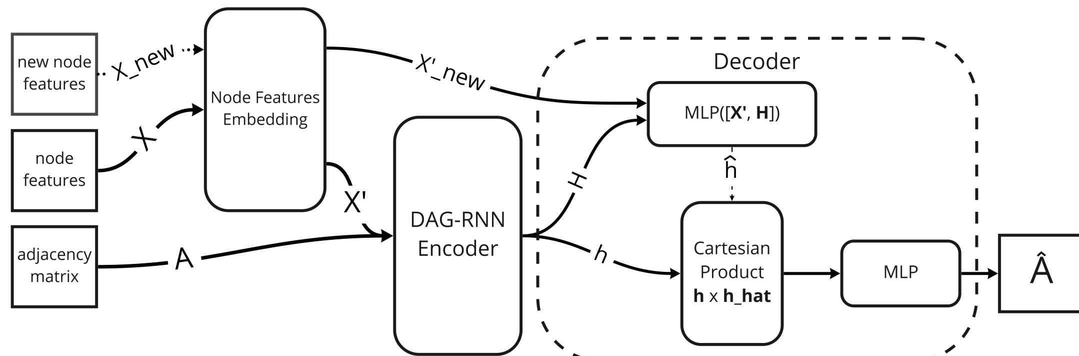

Figure 4.6 | **Decoder in Model 2.** Detailed architecture of Model 2. There are two parts in the decoder, each of them with their own MLP. First to estimate the hidden states h_hat; and second predicting the edge of each pair of nodes to finally output a complete adjacency matrix.

Training and Generation Process

When training the model, we use the input matrices as targets and essentially use the model as an autoencoder where we try to minimize the reconstruction loss of the adjacency

matrix. Such setup forces the model to encode the nodes in a way that they can be transformed again from the latent space. In addition we attempt to train another MLP to use H_{G_i} the state of the graph after processing node i and the candidate node type embedding to estimate the hidden state of the candidate node $\hat{h}_{i+1}$.

When extending the graph during the generation process, the model adds one node at a time using the given feature vector of a candidate node x_{n+1} and the hidden state of the graph H_n to estimate the hidden state of the candidate node h_{n+1}. After that, probabilities of adding an edge from nodes $v_0, v_1, ...v_n$ to candidate node v_{n+1} are estimated. Edges are added based on the predicted probability and a threshold given as a hyper-parameter. Afterwards, H_{n+1} and h_{n+1} are computed with the encoder and used for predicting edges for next candidate node v_{n+2}.

4.5.2 Loss Function

In the case of Model 2, the loss function is composed of two parts: first is the reconstruction loss shown in Subsection 4.3.4 which estimates the correctness of the generated adjacency matrix. Second component of the loss measures how well the model estimates the hidden states of the candidate nodes $\hat{h}$. For each node, the difference between the predicted values $\hat{h}_i$ and the output of the encoder h. Both h_i and $\hat{h}_i$ are vectors of real numbers. Two most commonly error measures used for regression tasks are Mean Squared Error (MSE):

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{4.7}$$

and Mean Absolute Error (MAE):

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n} \tag{4.8}$$

While MAE can handle outliers in the data much easier that MSE (where the square in the error makes the overall mean very high), it suffers from a drawback: Unlike MSE where gradients are high for large loss and decrease as the loss approaches zero, for MAE the gradients can be large even for small loss which leads to difficulties in the gradient descent algorithms with fixes learning rate. In Model 2 we use the Huber loss shown in equation4.9 which is behaves like MAE when the error is large and turns into MSE in cases where the error gets smaller than hyper-parameter δ. Additionally, unlike MAE, it is differentiable in 0 (when $y = \hat{y}$). Empirically, we have found $\delta = 0.01$ to be a well performing value for the parameter. In the Tensorflow library, $\delta = 1$ by default. In our proposed model, however, the cell unit we use in the encoder has a hyperbolic tangent

activation which means that $h \in [-1, 1]$. Similarly, the last layer of the MLP estimating the $\hat{h}$ has the same output range which means that the latent loss values are very small. Figure A.1.2compares various values of δ parameter.

$$L_\delta d(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}|^2 \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{4.9}$$

The loss for Model 2 is a weighted sum of the Focal loss(Equation 4.5) and the Huber loss(Equation 4.9) and is computed as follows:

$$\mathcal{L} = \frac{n^2 FL(A, \hat{A})}{2} + |h| L_\delta(h, \hat{h}) \tag{4.10}$$

Where n is a number of nodes an $|h|$ the dimensionality of the latent space (which is equal to the number of units in the RNN cell)

4.6 Model 3: Variational AutoEncoder

Having shown that the DAG-RNN is capable of encoding the DAG nodes in the latent space, we want to utilize it now for generating nodes which are not present in the input (new nodes that we want to add to the graph). For doing so, knowing the properties of the latent space is essential so the output of the model is easily controlled. Variational Autoencoder is a special type of Auto Encoder model which parametrizes the latent space as a probability distribution. The parameters of the distribution are estimated during the training process. During the generation phase new samples are generated from the probability distribution with the estimated parameters and decoded from the latent space. In addition to simple way to sample the latent space representation of the new samples, modelling the latent space as probability distribution helps to regularize it.

In contrast with Model 2,learning the parameters of the probability distribution corresponding to the latent space is to remove the direct $\hat{h}$ estimation as it is reducing the generalizing capabilities of the model. Since the we use hyperbolic tangent as the activation in the recurrent cell which creates the hidden states of the nodes, the range of the hidden states is in interval $[-1, 1]$. Lastly, with estimated parameters of the probability distribution we can sample hidden states of nodes with "mixed" types which is impossible with the previous models. Model 3 has fewer inputs than the previous model architectures as the matrix X' is omitted.

4.6.1 DAG-RNN VAE Model Architecture

For encoding the individual nodes of the graph structure we use the Bidirectional DAG-RNN described in Subsection 4.3.3. The parameters of Normal distribution $_\mu$ and z_σ are estimated using the separate Dense layers with no activation, and $\mathbf{z} = [z_0, z_1, ..., z_n]$ which are the estimates of the hidden states $\mathbf{h} = [h_0, h_1, ..., h_n]$ which are sampled from the probability distribution $\mathcal{N}(z_\mu, z_\sigma)$. The next step in the computation is to make a Cartesian product of h and z: $\mathbf{h} \times \mathbf{z} = \{\, (h_i, z_j) \mid h_i \in \mathbf{h} \;\text{ and }\; z_j \in \mathbf{z}\,\}$. The final step is to use the MLP with sigmoid activation function to classify each pair of (h_i, z_j) to determine the presence of the edge from node i to node j. All the pair form the estimate of the adjacency matrix $\hat{A}$. The complete structure of the DAG-RNN VAE is shown in the Figured 4.4

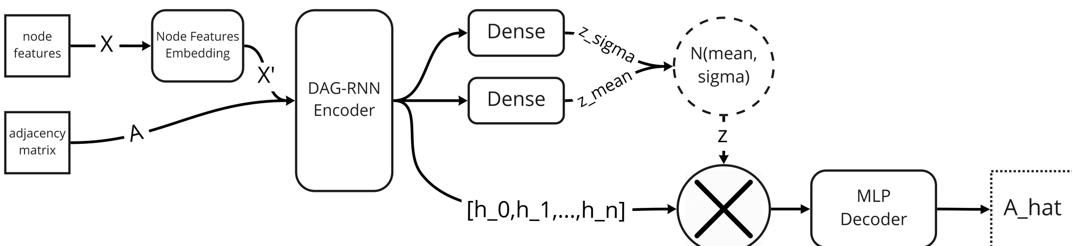

Figure 4.7 | **DAG-RNN VAE Model.** Architecture of DAG-RNN Variational AutoEncoder for adjacency matrix reconstruction. The parameters of the normal distribution are estimated using two separate dense layers. Then the layer does a Cartesian product which is the input into a MLP decoder with a sigmoid activation that determines the presence or not of an edge between the elements. Th output is the estimate matrix of adjacency of nodes.

Training Process

The hidden states $\mathbf{h} = [h_0, h_1, ..., h_n]$ are used to estimate the parameters μ, σ of the latent space and producing $\mathbf{z} = [z_0, z_1, ..., z_n]$ from the corresponding normal distribution. Decoding the Cartesian product of $\mathbf{h} \times \mathbf{z}$ with the MLP decoder yields the estimate of adjacency matrix $\hat{A}$. There are two loss functions used for the training of the model. For estimating the error of the parameters z_μ and z_σ we use the Kullback-Leibler Divergence which estimates the difference of two probability distributions. We refer to this loss as *latent loss*. For two normal distribution $p = N(\mu_1, \sigma_1^2)$ and $q = \mathcal{N}(\mu_2, \sigma_2^2)$ the Kullback-

Leibler divergence is computed as follows:

$$\begin{aligned} D_{KL}(p \parallel q) &= -\int p(x)\log q(x)dx + \int p(x)\log p(x)dx \\ &= \frac{1}{2}\log(2\pi\sigma_2^2) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}(1+\log 2\pi\sigma_1^2) \\ &= \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \end{aligned} \tag{4.11}$$

The latent loss is used for regularizing the latent space and allowing sampling representations of new nodes from it.

The second part of the loss function for Model is the *reconstruction* loss which estimates the quality of the result matrix $\hat{A}$. For this purpose, we use the Focal loss described in detail in Subsection 4.3.4. The two parts of the loss function have contradictory effects: While the reconstruction loss pushes the model to predict the matrix $\hat{A}$ more accurately regardless of the latent space properties, the latent loss forces the model to encode the nodes as samples from the Normal distribution regardless of the quality of the output. The final combined loss function for Model 3 is defined as weighted sum of the Focal loss and the latent loss.

$$\mathcal{L} = \frac{n^2 FL(A, \hat{A})}{2} + |z| D_{KL}(\mathcal{N}(z_\mu, z_\sigma^2) \parallel \mathcal{N}(0,1)) \tag{4.12}$$

where n is the number of nodes, is A the original adjacency matrix,z_μ and z_σ are the estimated parameters of the normal distribution and $\hat{A}$ is the estimated adjacency matrix. We divide the Focal loss term by 2 because we only estimate the half of the matrix A since it is a lower triangular matrix.

Generation process

During the training phase, two fully connected layers are used for learning the parameters z_μ and z_σ. Afterwards, z_j are sampled from $\mathcal{N}(\mu_2, \sigma_2^2)$ and used for predicting the edge from node i to node j. When generating an extended graph, the latent space representation of desired nodes $\mathbf{z} = [z_0, z_1, ..., z_m]$, z_i are sampled $\sim \mathcal{N}(0,1)$ where m is number of nodes to be added in the structure.The decoding step is the same as is the training phase. Examples of generated structures are shown in Chapter 7.

4.7 Transition from a User Related Graph to AD Entries

In the described models we focused on analyzing the input structure and finding a placement for extra nodes of a given type. Despite working with artificially created datasets, the framework has to be applicable in the real world environments, which means extraction of the data from the real AD (described in Chapter 5) and after the processing inserting the new nodes and their relations back in the AD

For that we need to populate the attributes of the created nodes. There are two types of attributes we focus on: attributes independent on the context and attributes implied by the placement in the structure. An example of the second type is Distinguished name or group membership. DN is build from all RDN on the way from the node to the root node, which means it is depend on the placement of the node as the parent node (OU or container) RDN is necessary for building the DN. Same applies for group memberships and other ACLs. The edge in the user related graph from the group node to the user node marks the group membership which means that after creating the new node entry, the next step is to add the membership to all the selected groups.

The other type of attributes is not dependent on the graph context and therefore can be randomly generated using external tool. Examples of such attribute is name, address, email , etc. For this task we propose to use either LDIF Generator[61] or services such as FakeNamesGenerator[62] which use either large database of human identity information or are based on ML models. The main pitfall when generating the fake identity for a honeyuser is ensure that the generated attributes do no reveal the presence of the honeyuser. Some of the attributes such as last logon, last password change etc, are filled in automatically by the AD server. Build-in commands of th AD can be used for inserting the data in the AD database.

Chapter 5

Dataset

This chapter contains description of the all data used for training and evaluation of all models presented in this thesis. As described in the previous chapters, AD holds sensitive data of possibly high value. Some of the information is considered private under the recent regulations of privacy of user data. Since deep learning models require thousands of data samples to train, using real data is not an option. Sharing or even extracting information from the production AD is often forbidden for third parties by company policy. Moreover, some of the data stored in the AD are considered personal under the law of the EU and as such, consent of the owner is required for processing it.

Therefore, we used the expert knowledge gained from the analysis of a few real AD samples to create an artificial dataset for training the models. The real data samples were used for evaluation and testing. The generated data samples are valid directed acyclic graphs with correct node type sequences. In Chapter 2 we showed that there is a wide variety of built-in object types in AD further extened with the AD schema. In order to model such structures we limited the number of classes in the data to ones closely related to the user type, which is our primary target. The structure is referred to as User Related Graph and it can contain following node types:

- User
- Computer
- Domain
- OrganizationalUnit (OU)
- Group (in this context equivalent to SecurityGroup)

Assumption is made in this thesis, that relations among listed object are notable when inferring the value of the targets int the AD.

The models, trained using the dataset, are designed to predict the presence or absence of an edge between a pair of nodes. It is important to note that due to the structure of the graph, the cases are highly imbalanced. Let us see the example of an AD graph in 5.1.

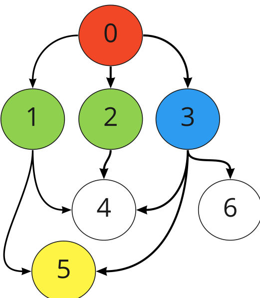

Figure 5.1 | **Simple example of user-related graph.** Example showing structure with 1 Domain node (in red), 2 Groups (green), 1 Organizational Unit (blue), 2 User nodes (white) and one Computer (yellow). The numbers in the nodes represent the ordering of this DAG

$$X = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 2 \\ 3 \\ 4 \\ 3 \end{bmatrix}, A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Figure 5.2 | **Matrix representation of the example User Related Graph.** Matrix representation of the structure in Figure 5.1. X represents the node types and A is the adjacency matrix of incoming edges. Nodes are ordered according to their IDs showed in the figure.

Even for the small graph, such as the one in Figure 5.1, we can see that its corresponding matrix shown in Figure 5.2 is rather sparse. When adding the new nodes to the structure, we have to account for possible edges with any of the existing nodes. For a structure of containing n nodes and m new nodes that means $n * m$ possible edges.

Yet in most of the cases, the edges do not exist which means that the prior probability of adding a new edge is shifted heavily towards 0 (meaning no connection). A common practice in similar situations is resampling of the dataset. In particular there are two options: under sampling and oversampling. Undersampling is a process during which, samples from the majority class are randomly deleted to match the class size of the minority class. The opposite approach, called oversampling, tackles the imbalance by artificially creating samples from the minority class with similar properties.Neither of the two methods is applicable in the case of the User Related Graph because such modifications would change the very structure of the graph we want the model to learn.

A second possible approach to imbalanced dataset handling is to make the model aware of the situation. The easiest way to do so is to modify the loss function that is being used for training in such way, that it is penalizing errors of minority class more. This places more impact on that particular class which is shown in Subsection 4.3.4.

In simple terms, in our domain that means considering one type of error less important than the other, more specifically, we consider adding an extra edge less severe than missing

an edge which should be added.

5.1 Artificial Dataset Creation

For the training and evaluation of the model, we created several artificial datasets which differ in the size of the graphs and the average degree of nodes in the graph. The process of generating samples in the datasets consists of three step. First one is a generation of a random directed graph using the Networkx [59] library. In the generation the the amount of edges in the graph is samples from a Normal distribution with mean determined from the statistic of the real examples.

Second step is assigning node types to all vertices in the generated graph. Nodes are labeled according to the constrains described in Subsection 2.3.3. Cases where there are multiple valid node types are assigned randomly with prior probabilities extracted from the real samples. The last step of the generation process is the validity check. In particular, any inconstant edges are removed from the graph. An example of such inconstancy is the User node being a successor of two Organizational unit nodes which is not allowed in the AD. In case of an opposite situation where user node is not a member of any OU, the relation is created with a randomly selected OU in the graph. Subsequently, each of the graphs contain at least one Organizational Unit node which is constant with the AD architecture where Organizational Unit "Users" is present by default.

Four artificial datasets were created for model training and evaluation. Table 4.3.4 shows the details of each of the datasets.

Dataset	#samples	Mean $\|V\|$	Mean $\|E\|$
Dataset15	2,000	12.5155	19.0255
Dataset50	2,000	39.8835	65.4965
Dataset150	2,500	115.11	192.49
Dataset500	1,000	353.369	600.173

Table 5.1 Comparison of artificial datasets in terms of number of samples, number of nodes and edges.

The number used in the name of the dataset references the number of nodes in the graphs of that dataset. The smaller datasets, Dataset15 and Dataset50, are motivated by the work in [47] which used a DAG of up to 30 vertices. We include the smaller datasets to evaluate our approach on graphs of similar size. For active directory domain however, such graphs are unrealistic and therefore we also work with 150 nodes in the Dataset150 and 500 nodes in the Dataset500, which have a size more comparable with a small to medium size organization.

5.2 Extracting Data From The Active Directory

Structure of the existing AD has to be extracted in order to use the proposed framework for extending it with honeytokens.

The simplest way of extracting data from an existing AD is a tool called Sharphound [54] which is a utility designed to gather the data required for the Bloodhound software. It performs several queries in the AD and gathers the nodes of each type in separate JSON files. By loading the JSON files in the Bloodhound, one can see the whole AD structure as shown in Figure 5.3.

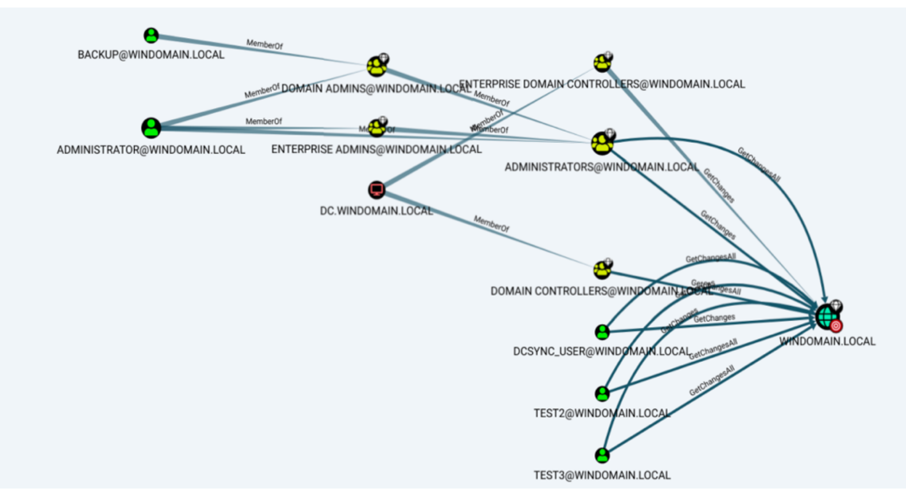

Figure 5.3 | **Example of AD data.** Data extracted with the Sharphound tool and visualized using the Bloodhound tool.

Even though Sharphound only reads the AD and makes no modification, Windows Defender, the built-in antivirus software by Microsoft has the signature of Sharphound in the database and therefore it blocks its execution. In order to bypass the alert, one needs Administrator rights in the domain.

The second option is to query the AD directly using either Powershell cmdlets or the LDAP protocol. For this purpose we have developed Powershell tool which gathers the information in a JSON format and a Python utility to process the JSON files either from Sharphound or the AD query results. To build the graph we are using the networkx library. Both of the tools are available in the Git repository. Once the data is extracted from the AD it is used for building of the User Related Graph which can be processed by the machine learning models.

Chapter 6

Graph Reconstruction Experiments

This chapter describes the AD structure modelling experiments and analysis of the results for each proposed model. The goal of the experiments is to evaluate how well each of the model is capable of capturing the structure of the processed graph and reconstruct it. That is important for the honeypot placement task since the main goal of the framework is to find a honeypot placement *with respect* to the original structure. Additionally, the models that are based on autoencoders are evaluated to estimate the amount of information lost in the autoencoding process. The models are evaluated using the artificial datasets that we created and was shown in Chapter 5.

6.1 Evaluation Metrics

For estimating the quality of the output of the models, some metric functions have been defined. The models perform a repeated binary classification task for each pair of nodes, and therefore the common metrics used for the evaluation of classifiers are applied. The most common metric for classification tasks is accuracy. However, in the case that the classes are unbalanced, such metric can be misleading. Chapter 5 describes why one of our datasets contains a class with the majority of the samples. Since accuracy is computed as $accuracy = \frac{TP+TN}{TP+TN+FP+FP}$ we can see that a high accuracy can be achieved even if the model predicts only the majority class. In our domain that means predicting no edge at all times. For that reason we need to evaluate the quality of the model using other metrics. In particular Precision, Recall and F1 score which is a harmonic mean of the previous two. To emphasize the importance of precision or recall, a F_β score is used where β is a parameter which determines how many times one of the metric is more important than the other. In this thesis, we worked with $\beta = 2$ which shifts importance in F score towards recall.

The complete list of metrics used in this experiment together with the formulas is

shown in Table 6.1

Metric	Formula
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
Precision	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F1 Score	$2 * \frac{Precision*Recall}{Precision+Recall}$
F_β Score	$(1+\beta^2) * \frac{Precision*Recall}{\beta^2*Precision+Recall}$

Table 6.1 Metrics used for the model evaluation in graph reconstruction experiment

Additionally, the Area Under the Curve (AUC) of Precision/Recall and the curve itself is used for analysis of the model quality and comparison of the model types. Precision/Recall curve, similar to the well known ROC curve, shows the precision and recall of the model with various thresholds used for the binary classification. Area under the curve can be used for evaluating of the model quality. Apart from the evaluation, this curve can be used for choosing the optimal threshold if there are additional constrains (such as that either metric has a minimal lower bound). AUC for both PR curve and ROC curve, has range $[0, 1]$) where higher value means better model ($AUC = 1$ would be a perfect classifier which makes no mistakes with arbitrary threshold).

6.2 Structural Experiment Results and Analysis

This section shows the results for the experiments modelling the AD Structure for each model type. The results are evaluated per dataset.

6.2.1 Model 1: Direct Edge Prediction

Metric	Dataset15	Dataset50	Dataset150
Precision	0.1693	0.0793	0.0392
Recall	0.9525	0.9530	0.6069
AUC (PR Curve)	0.2442	0.1205	0.0524
F1 Score	0.2875	0.1464	0.0783
F_β Score($\beta = 2$)	0.4947	0.2974	0.1557

Table 6.2 Performance of Model 1 for generating an AD structure using datasets of various graph sizes.

From the data in Table 6.2 we can see that the performance of the Model 1 is unsatisfactory. As the size of the graphs grows, it converges towards the performance of the

random model as shown in the Figures 6.1 and 6.2. Even with a small dataset with a maximum graph size of 50, the area under the Precision/Recall curve is extremely small which means that there is a high trade-off between the two metrics. In other words, even a slight increase in recall means a huge drop in precision.

The main explanation for the results is that the simplicity of the model and the amount of additional information about the node pair where an edge is predicted, causes the poor performance. We can demonstrate such a case on the following example: Let there be a hidden state of a node h_i and two candidate ancestors of type group x'_1, x'_2 out of which only the first one is a true ancestor. The model attempts to classify pairs (h_i, x'_1) and (h_i, x'_2) and the desired output is 1 in the first case and 0 in the second case. However, since both candidate nodes x'_1, x'_2 are of the same type, the embeddings are identical which means that the model is forced to output different predictions with identical input.

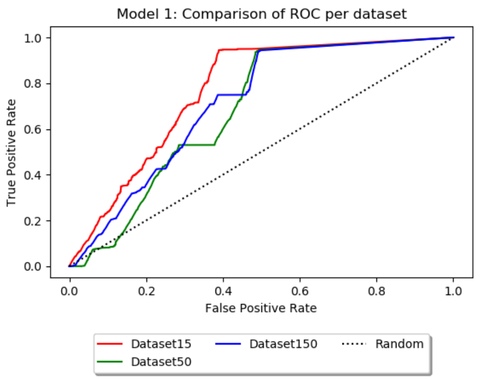

Figure 6.1 | **ROC Curves of Model 1 per dataset.**

Figure 6.2 | **Precision/Recall Curve of Model 1 per dataset.**

For achieving better results we either need to provide more information for the model to distinguish the two nodes shown in the example, or change the architecture of the model in such a way that the predictions are not independent. This way at the time of predicting the edge for (h_i, x'_2) the model can take the result of (h_i, x'_1) into account.

6.2.2 Model 2: DAG-RNN Autoencoder

In the results in Table 6.3 we can see that with the extension of the model capacity, we achieve significant improvements in smaller graph datasets. Data from experiments in larger graph datasets show, that the generalizing capabilities of the model are still limited. Figure 6.4 shows that the difference of AUC for PR curve in the smallest and the largest dataset is more than 60%.

The comparison between reconstructed graphs and the original structure shown in Figure 6.5, shows that in the small graphs, Model 2 is capable of capturing and correctly

Metric	Dataset15	Dataset50	Dataset150	Dataset500
Precision	0.6921	0.6854	0.2351	0.0421
Recall	0.6253	0.1981	0.2109	0.5733
AUC (PR Curve)	0.7501	0.4041	0.2025	0.0720
F1 Score	0.6569	0.3074	0.2223	0.0784
F_β Score($\beta = 2$)	0.6376	0.2309	0.2153	0.05167

Table 6.3 Performance of Model 2 using datasets of various graph sizes

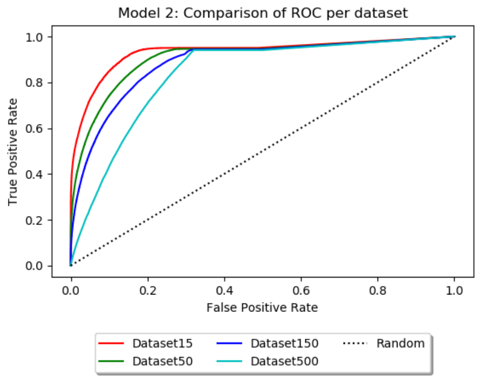

Figure 6.3 | **ROC Curve of Model 2 per dataset.**

Figure 6.4 | **Precision/Recall Curve of Model 2 per dataset.**

predicting the majority of the relations among the nodes. One of the notable errors that Model2 makes, is to predict more edges from Organizational units to User nodes. In other words, the model correctly decides that there should be an edge, but is not able to limit the edges from multiple OUs to a single User node. Similarly, to Model 1, the decision about a pair of nodes is isolated and other predictions are not taken into account which results in this type of error. Model 2 also tends to predict more edges than are present in the original which can be reduced by increasing the threshold. Higher thresholds, however, often yield graphs which are not connected.

With an increased graph size the reconstructive powers of the model drop, something that was already illustrated in the PR Curve. In Figure 6.6 we can see an example of such a case. The reconstructed graph is connected, but we can see that while nodes 21-24 are assigned to two OUs, other User nodes in the left part of the graph are not members of any OU. From other examples it is clear that even the capacity of Model 2 is not enough to correctly capture the relationships in the graph. Nevertheless, Model 2 is able to learn some of the underlying pattern in the graph. Despite the limitations described in this subsection, Model 2 can still be used for generating new nodes for a graph. In the generation process, the model only predicts edges to a single node at a time which is a simpler task in comparison to predicting the whole adjacency matrix.

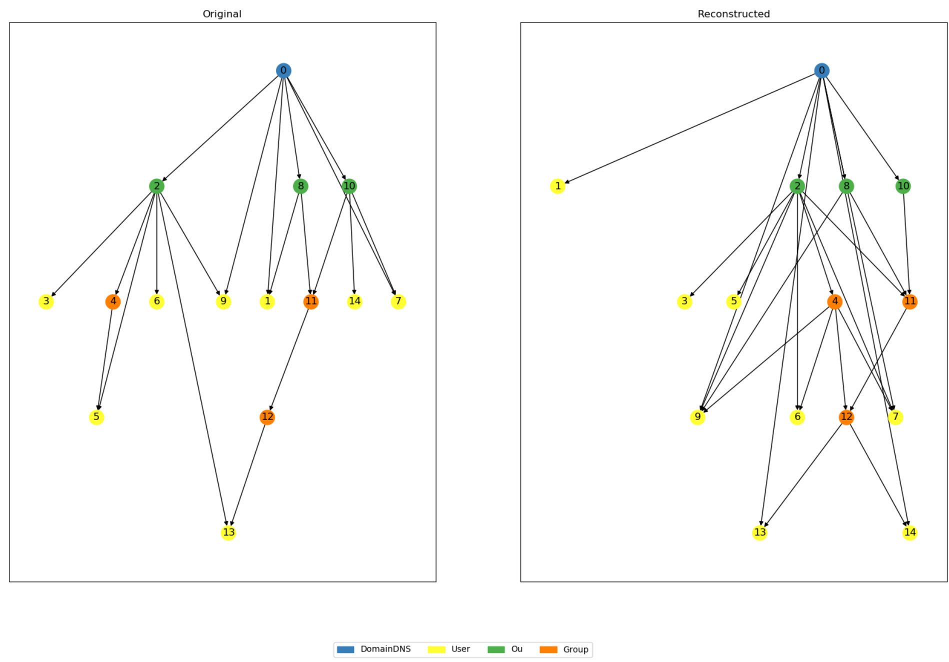

Figure 6.5 | **Sample from Dataset 15 reconstructed with Model 2.**

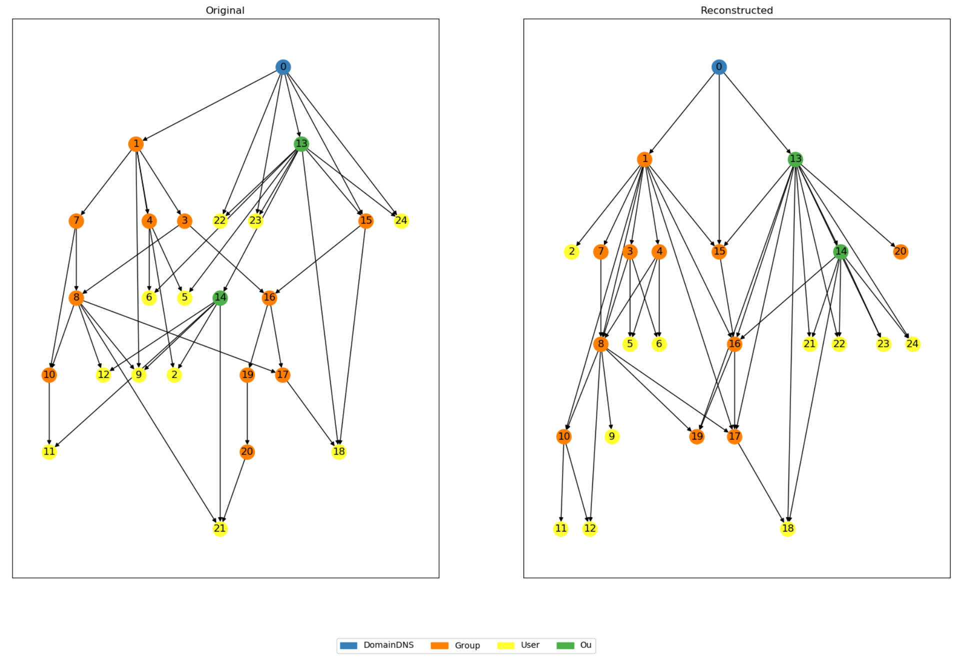

Figure 6.6 | **Sample from Dataset 50 reconstructed with Model 2.**

6.2.3 Model 3: Variational Autoencoder Model

This subsection shows the results of AD Graph structure modelling experiments for Model 3. Apart from the metrics, we show examples of reconstructed graphs together

with the original structures which were used as input for the model.

The results per dataset listed in Table 6.4 show that Model 3 outperforms previous models in all datasets. Unlike Models 1 and 2, the performance drop with the increased graph size is much milder. Despite promising results in Dataset 50 and Dataset 150, in the largest dataset, the AUC of the PR curve is notably lower. In the case of the curves for the smaller datasets, we can clearly identify the point where the area under the curve is largest and it is therefore the optimal threshold for out model. In the case of Dataset 500, depicted in cyan in the plot, finding such point is not straightforward.

Metric	Dataset15	Dataset50	Dataset150	Dataset500
Precision	**0.8093**	0.7994	0.8038	0.5185
Recall	**0.9456**	0.8948	0.4553	0.7267
AUC (PR Curve)	**0.9210**	0.8877	0.7281	0.6923
F1 Score	**0.8722**	0.8444	0.5813	0.6052
F_β Score($\beta = 2$)	**0.9147**	0.8739	0.4985	0.6726

Table 6.4 Evaluation of DAG-RNN VAE using datasets with various graphs sizes

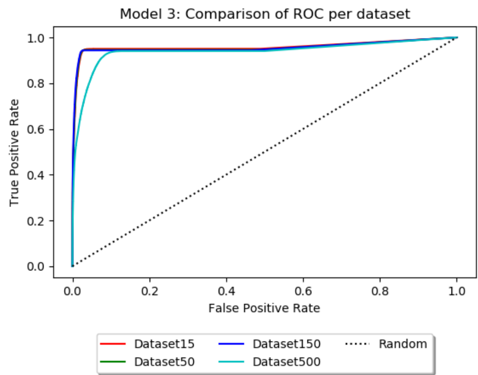

Figure 6.7 | **ROC Curve of Model 3 per dataset.**

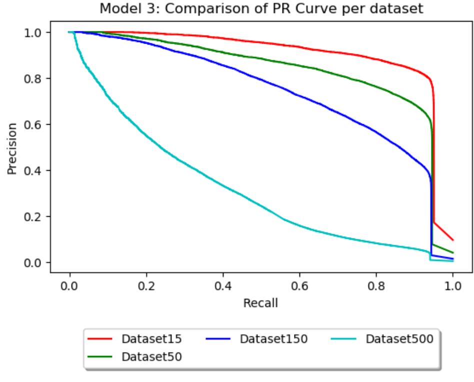

Figure 6.8 | **Precision/Recall Curve of Model 3 per dataset.**

In Figures 6.9 and 6.10 we see the original structure and the output reconstructed using Model 3. Despite the significant similarity of the graphs, there are inconsistencies introduced during the auto encoding process. Namely, in Figure 6.9, node 14 is shown to be a member of 2 Organizational Units something which is not possible in the AD. Additionally, nodes 12 and 14 are incorrectly placed in the structure. In the larger structure shown in 6.10, differences between the original and the prediction are even more visible. Similarly to Model 2, we can see that some of the User node are located in two Organizational Units simultaneously while other are not located in an which is equally wrong. Majority of the group membership is estimated correctly which is great improvement from the Model2 where group membership was often interchanged.

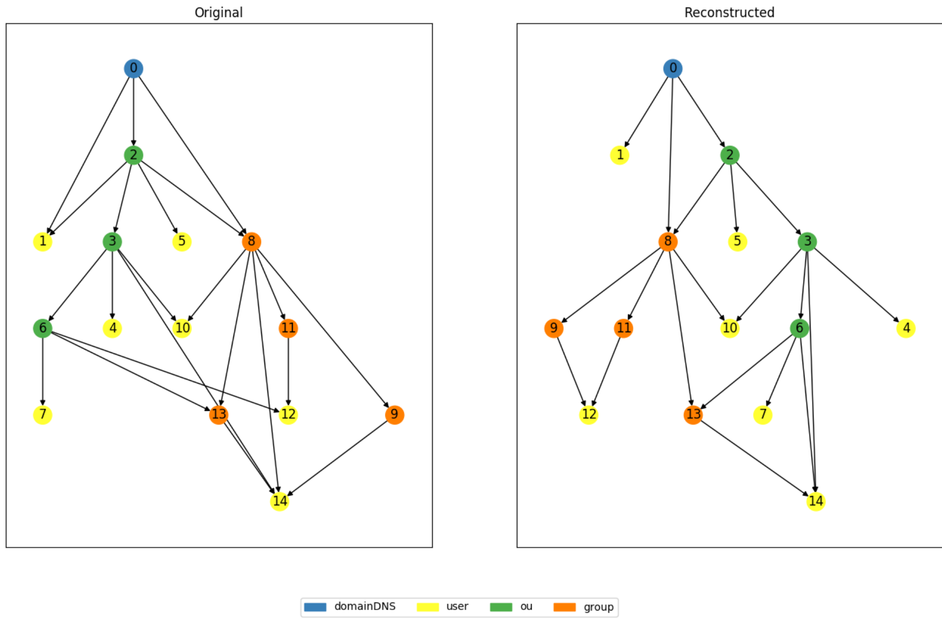

Figure 6.9 | **Examples of VAE output - Dataset15.** Examples of structures generated with the VAE of Model 3. Sample from Dataset 15 is used in the comparison

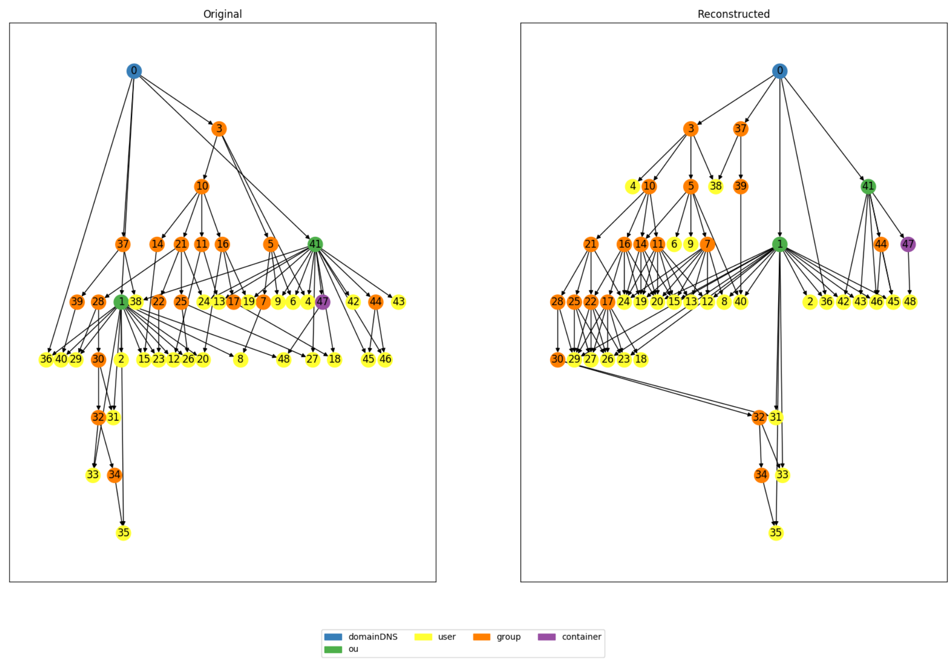

Figure 6.10 | **Examples of VAE output - Dataset50.** Examples of structures generated with the VAE of Model 3. This is a specific sample from Dataset50 that was used for a comparison.

6.2.4 AD Structure Modelling: Model Comparison

So far we evaluate each model type individually. In this subsection we compare the results examine the differences and show outputs of each model for the same inputs.

We compare the model quality using mainly the Precision/Recall curve as it is not dependable on single classification threshold.PR curve shows the performance of the model when certain levels or Precision and Recall metrics are set. For the comparison, we use the area under the PR curve. Higher value means better model because of there is smaller trade off when increasing the threshold for either metric. Perfect model has AUC=1. As a baseline, in the figures depicted with dotted line, we use the Random classifier. The baseline is model using the prior probability of the edge presence for each pair of node types.

Figures 6.11 and 6.12 show model comparison in Dataset15 and Dataset50. We can see that while in the smallest dataset Model 1 is slightly better than random classifier, as the size of the graphs increases, Model 1 performance drops to the same level as in the random classifier. In case of Model2, we ca see almost 50% drop in AUC when graph size increase from 50 to 150.

Note that such result does not necessarily imply flaws in the generated structures. We discuss generative capabilities of the model in Chapter 7 which also shows examples of generated structures.

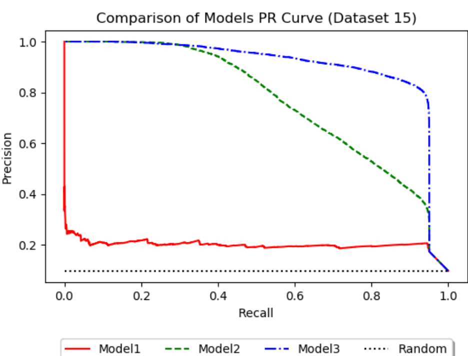

Figure 6.11 | **PR curve comparison (Dataset15).** Graph showing Precision-Recall curve for each of the models for Dataset15. Random model is shown as a baseline.

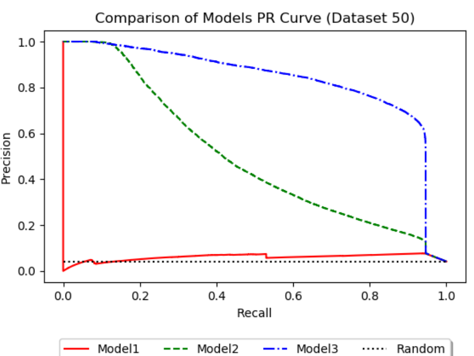

Figure 6.12 | **PR curve comparison (Dataset50).** Graph showing Precision-Recall curve for each of the model for Dataset50. Random model is shown as a baseline.

For comparison of Models 2 and 3 we show reconstruction of the same input graphs from Dataset 50 and Dataset 150. Use visualize the original A, $\hat{A}_{Model2}$ and $\hat{A}_{Model3}$ as red, green and blue channels of RGB image. In the visualization we can see for every edge, how well can each model predict it. Cases where both models predict the edge correctly,

are shown in black for true positives and in white in true negatives. Any depicted pixel in color shows an error in one of the models. Detail color codes of all possible cases are shown in 6.5.

	Model 3 TP	Model 3 FP	Model 3 TN	Model 3 FN
Model 2 TP		None	None	
Model 2 FP	None			None
Model 2 TN	None			None
Model 2 FN		None	None	

Table 6.5 Color codes for the comparison of Models 2 and 3

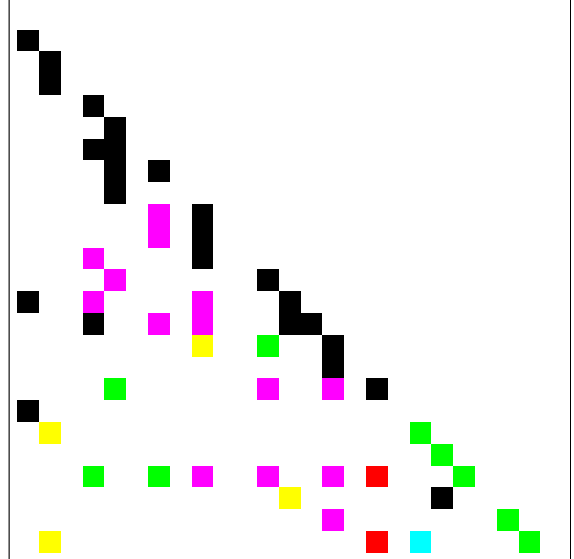

Figure 6.13 | **Model 2 & 3 comparison (Dataset50).** Comparison of graph adjacency matrix reconstruction for Models 2 and 3 using the Dataset 50. Color codes listed in Table 6.5.

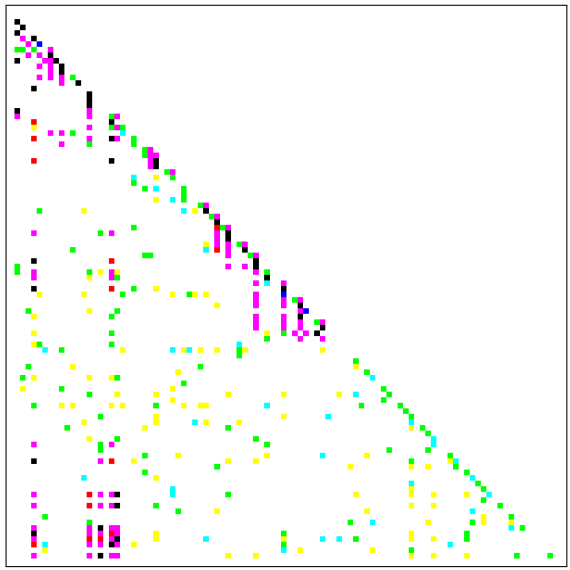

Figure 6.14 | **Model 2 & 3 comparison (Dataset150).** Comparison of graph adjacency matrix reconstruction for Models 2 and 3 using the Dataset 150. Color codes listed in Table 6.5

As we can see in Figures 6.13 and 6.14 in task of adjacency matrix reconstruction, Model 2 performs well in the nodes close to the root of the structure. Moving further in the node sequence, the performance drops significantly. The reconstructive capabilities of the Model 3 are far more consistent with the increasing size of the graph. In general, both models tend predict more edges than is present in the original A. As we described earlier such type of error is more acceptable in the the process of honeyuser creating since we put more impact on not missing an edge.We can see that especially in the larger domains Model 3 clearly outperforms Model 2 and therefore is more suitable for modelling the structure of the AD it can scale to thousands of nodes.

Chapter 7

Generative Experiments

The main goal of the framework is to extend exiting structure with generated users nodes. This chapter describes experiment which evaluates the generative capacity of the models. There are several conditions for what we consider a well-placed honeypot: Firstly, it has to be accessible by the attacker. In the context of the AD, it means not only connected to the structure, but also detectable by AD reconnaissance or domain listing tools. Secondly, its placement in the structure must not raise suspicion of a honeypot presence. In other words, we aim for such a placement, which is not abnormal, or alarming in any way for the attacker. Additionally, since certain objects are expected to be connected together if the generated placement doesn't meet these expectations, it increases the chances of being spotted by the attacker.

7.1 Evaluation Metrics

In general, evaluation of artificially generated samples struggles with absence of good criterion which means it is difficult to define a measure of quality. For the purposes of the AD graph extension task, we define three metrics. Disconnected node ratio, Edge Validity Ratio and Mean User Node Edge Count Ratio which are defined as follows:

Disconnected Node Ratio

$$DisconnectedNodeRatio = \frac{1}{m}\sum_{i=1}^{V_0} 1, \text{where } V_0 = \{v \mid \delta^-(v) = 0\} \tag{7.1}$$

where m is a amount of newly added nodes, $\delta^-(v)$ number of incoming edges to a node v (in-degree) and V_0 set of all nodes with $\delta^-(v) = 0$. For this metric, a lower value means a better result because in Active Directory the whole structure has to be connected.

Validity Ratio

$$ValidityRatio = \frac{\sum_{i=1}^{m} |E_{valid}^{i}|}{\sum_{i=1}^{m} \delta^{-}(v_i)} \tag{7.2}$$

The validity ratio measures how many of the valid edges to a node the model got correctly. where m is a number of new nodes, $\delta^{-}(v_i)$ is a in-degree of a node and $|E_{valid}^{i}|$ is th set of valid edges to a node i. Edge (x, y) is considered valid if such a relation can exist in the AD. For instance, if the group membership is consistent with the AD structure and there is an edge from a node type Group to the new User node, we consider it a valid edge.

A special case is when there are multiple edges from Organizational Unit nodes to a single User node. Such setup is inconsistent with the AD and if there are multiple edges from an OU node to a single User node, only one of them is considered valid.

Mean User Node Edge Count Ratio

$$MeanUserNodeEdgeCount = \frac{\sum_{i=1}^{n_{User}} \delta^{-}(v_i)}{n_{User}} \tag{7.3}$$

The Mean User Node Edge Count measures mean amount of incoming edges for a user node type. In the previous equation n_{User} is the number of user nodes in the graph and $\delta^{-}(v)$ is the in-degree of a node v.

By computing the *Mean User Node Edge Count* for the extended graph and dividing by the *Mean User Node Edge Count* of the original structure we get the *Mean User Node Edge Count Ratio*. The best value for this metric is 1, where the user nodes in the extended graph have in average the same amount of incoming edges as in the original. Values higher than 1 suggest that the User nodes in the extended graph have a higher in-degrees while values lower than zero show less incoming edges. Only connected nodes (nodes with at least one incoming edge) are included in this metric.

7.2 Generative Experiment Results and Analysis

In order to better evaluate our proposal we focused on two main criteria: If the resulting structure is connected and the ratio of invalid edges to the new nodes. An additional evaluation metric is the overall number of added nodes with respect to the node type average in the original graph. In contrast with the usual setup of models which generate artificial samples, in the framework the model is not designed to generate the graph from scratch but extend the existing structure in a meaningful way. We evaluate the quality

with two metrics: Connectivity of the graph and validity of the predicted edges. Moreover, we examine if the average number of connection to the new nodes is similar to the same node type in the original graph. We directly compare the models and their results.

7.2.1 Generative Experiment: Model 1

From the data in Table 7.1 we can see that the threshold parameter does not influence the output in any way. Even in the smallest dataset only one third of the predicted edges is valid despite predicting 9 times more edges per User node on average. It is clear that Model 1 does not have enough capacity for performing the generative task in neither small nor large graphs. For this reason, Model 1 is not included in the model comparisons.

	$t = 0$	$t = 1e^{-5}$	$t = 1e^{-3}$	$t = 0.1$	$t = 0.25$	$t = 0.5$	$t = 0.75$
Dataset15	0.3627	0.3627	0.3627	0.3627	0.**3627**	NaN	NaN
Dataset50	0.3051	0.3051	0.3051	0.3051	0.**3051**	NaN	NaN
Dataset150	0.3437	0.3437	0.3437	0.3437	NaN	NaN	NaN

(a) Edge validity ratio with various thresholds

	$t = 0$	$t = 1e^{-5}$	$t = 1e^{-3}$	$t = 0.1$	$t = 0.25$	$t = 0.5$	$t = 0.75$
Dataset15	9.0902	9.0902	9.0902	9.0902	9.0902	0.0	0.0
Dataset50	27.0835	27.0835	27.0835	27.0835	27.0835	0.0	0.0
Dataset150	73.040	73.040	73.040	73.040	0.0	0.0	0.0

(b) Mean User Node Edge Count Ratio with various thresholds

Table 7.1 Results of generative experiments for Model 1

7.2.2 Generative Experiment: Model 2

With Model 2, we can see substantial improvement in both metrics shown in Table 7.2 across all datasets. With edge threshold close to 0.25 most of the edges in the generated nodes are valid while keeping the edge count average very close to the original nodes. In contrast withe the experiment described in 6, the performance of the Model 2 does not drop as the size of the processed graph increases. In the examples shown, it can be seen that despite producing valid edges, Model 2 tends to predict the majority of the new nodes in the same location within the structure. That is undesirable in the honeypot placement scenarios.

7.2.3 Generative Experiment: Model 3

Based on the results of graph reconstruction experiment, where Model 3 dominated the other models, it is expected to perform similarly well in the generative tasks. In 7.3 we can

	$t = 0$	$t = 1e^{-5}$	$t = 1e^{-3}$	$t = 0.1$	$t = 0.25$	$t = 0.5$	$t = 0.75$
Dataset15	0.3627	0.6487	0.6586	**0.6838**	0.64510	0.3084	0.1444
Dataset50	0.3051	0.6887	0.7187	**0.7221**	0.6742	0.5126	0.1921
Dataset150	0.3437	0.4574	0.4577	0.4587	**0.6918**	0.4335	0.4456
Dataset500	0.30698	0.3734	0.39182	**0.5886**	0.4789	0.4642	0.0

(a) Edge validity ratio of with various thresholds

	$t = 0$	$t = 1e^{-5}$	$t = 1e^{-3}$	$t = 0.1$	$t = 0.25$	$t = 0.5$	$t = 0.75$
Dataset15	9.0902	5.0372	4.7765	3.6223	2.7446	**0.7753**	0.6249
Dataset50	27.0835	11.9958	11.4949	11.3986	7.4761	**1.0479**	0.6266
Dataset150	73.0409	19.1629	9.9250	2.6583	**1.0819**	0.0671	0.8781
Dataset500	220.2978	172.6761	172.6001	139.595	1.8808	**0.9523**	0.0

(b) Mean User Node Edge Count Ratio with various thresholds

Table 7.2 Results of the generative experiments for Model 2

see that with increasing size of the graphs, the threshold with best performance lowers.

In most cases, at least one node generated by the Model 3 remains unconnected which results in higher Disconnected Node ratio for Model 3. That is placement which is not compatible withe AD, but there is a simple way to overcome such problem: After predicting the node edges, additional validation is performed and if no edges are predicted the node is not added to the structure. Repeated generation of nodes with no edges can be also considered a stopping signal.

	$t = 0$	$t = 1e^{-5}$	$t = 1e^{-3}$	$t = 0.1$	$t = 0.25$	$t = 0.5$	$t = 0.75$
Dataset15	0.3627	0.5773	0.6868	**0.69277**	0.6693	0.61876	0.2756
Dataset50	0.3051	0.4296	0.3901	0.3922	0.4116	**0.5959**	0.2916
Dataset150	**0.3437**	0.3311	0.2521	0.1761	0.1466	0.0471	0.0
Dataset500	0.3069	0.39165	**0.6751**	0.2675	0.05726	0.0	0.0

(a) Edge validity ratio with various thresholds

	$t = 0$	$t = 1e^{-5}$	$t = 1e^{-3}$	$t = 0.1$	$t = 0.25$	$t = 0.5$	$t = 0.75$
Dataset15	9.0902	4.4599	2.5240	**1.0157**	0.7231	0.3556	0.1582
Dataset50	27.0835	8.9144	4.9581	**1.3672**	0.6860	0.06724	0.0020
Dataset150	73.0409	19.1629	9.9251	2.6583	**1.0819**	0.0671	0.0086
Dataset500	84.9209	13.9511	2.5811	**0.10462**	0.04576	0.0	0.0

(b) Mean User Node Edge Count Ratio with various thresholds

Table 7.3 Results of the generative experiments for Model 3

7.2.4 Model Comparison

From results in Tables 7.2 and 7.3 it is clear the edge threshold hyper-parameter plays important role in the quality of the outcome. The results show that the best performing value of threshold remains consistent as the graph size grows, with lower values for Model 3.

Unlike Model 2, which produces connected graphs in vast majority of times, Model 3 has tendency of not connecting all nodes. However, nodes with predicted edges are well distributed in the structure which is desirable with respect to the honeypots. Model 2 produces node placements which are similar to each other resulting in structures with lot of new nodes in the same place. Figure 7.1 shows the case where Model 2 predicts edges from multiple Organizational Units to a single node while Model 3 distributes the new nodes in multiple OUs in the graph.

Figure 7.2 shows the result of the generation process of Model 2 and 3 in larger dataset. It shows the same specifics of generation as we described for graphs in Figure 7.1. Model 2 places all nodes in one organizational unit while Model 3 leaves more than half of the nodes disconnected.

In comparison with the graph reconstruction experiment, where Model 3 outperformed the other models, in this experiment the results do not show such dominance in the performance. Results of Model 1 are unsatisfactory and in current setup it is not possible to utilize for generating additional nodes for graph extension.

7.2.5 Examples of Generated Structures

7.3 Generation of Honeyusers

Apart from analysis of the generated structures in the artificial datasets, testing in real-world environment is in order. The honeyusers are meant to attract the attention of the attacker so any inconsistencies can reveal that the user is not real. For this evaluation we are using data from an existing Active Directory. Data extracted with Sharphound tool. For the purpose of the evaluation the data has been anonymized to avoid leaking sensitive data. For the purpose of the experiment, the domain name has been changed to *TEST.cz* and name of the container objects translated to English with kept semantic. The domain originally consists of 120 User objects, 106 Computer objects, 35 Organizational Units and 170 Groups. In the experiments, 20 new nodes are predicted as a honeyusers.

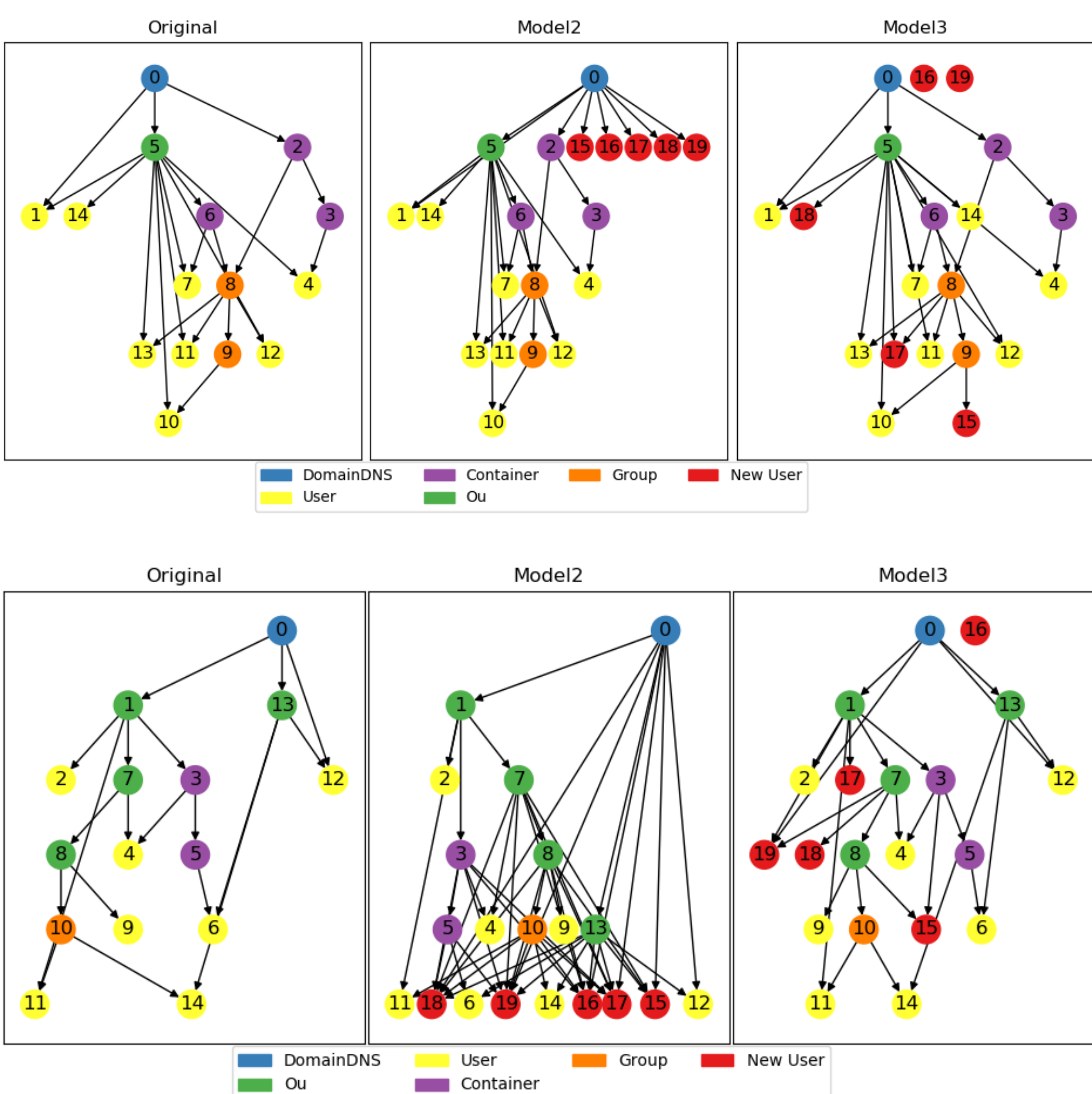

Figure 7.1 | **Comparison of structures generated by Models 2 and 3 (Dataset 15).** Model 3 generates more realistic looking placements for the new User nodes but it also generates nodes that are disconnected from the rest of the graph.

7.3.1 Examples of Predicted Parent Nodes Testing Domain

In this subsection we show the output of the model when applied to real production AD. Table 7.4 shows examples form the generation process. Only for 25 % of the generated users the model correctly predicted parent Organizational Unit. On average, for 21,6 % of generated users at least one of the predicted predecessors was inconsistent. In the first example in Table 7.4 both errors were made as the user is was predicted to be a successor

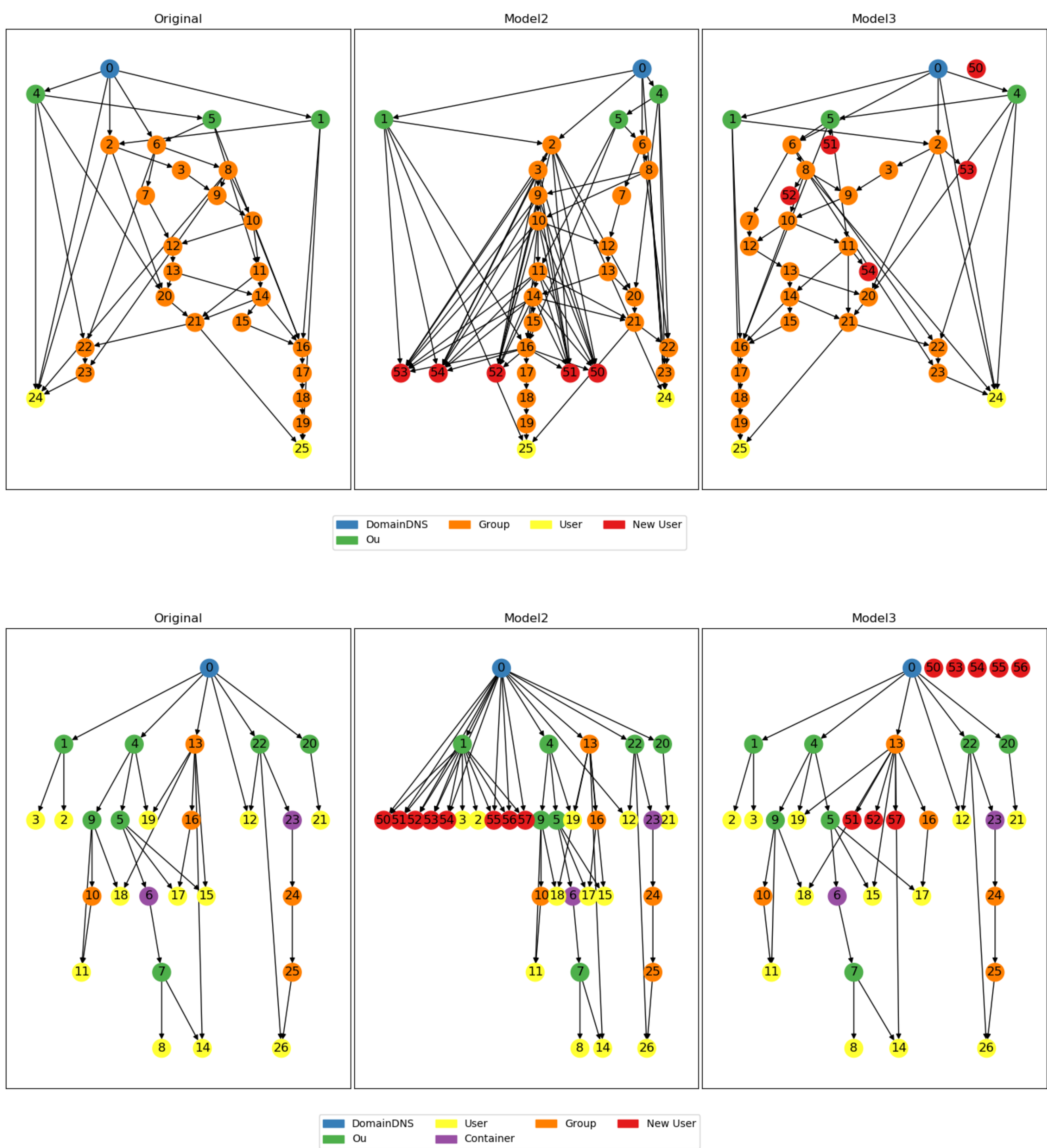

Figure 7.2 | **Comparison of structures generated by Models 2 and 3 (Dataset 50).** Model 2 generates new User nodes with a lot of connections or under the same OU, while Model 3 generates better placements but more disconnected nodes.

of a Computer object "CN=CZPRG-P-EXC01".

The example of successful placement of the honeyuser is shown in the last row of the table. For the user, being placed in the OU **Engineering**, the model also predicted

membership in groups representing access list of the organizational unit.

attribute	value
RDN	CN=CN=Sizemore C. Colin
Ou	*None*
memberOf	CN=Domain Admins,CN=Users,DC=TEST,DC=cz
	CN=ACL TEST domain users,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=Administrators,CN=Builtin,DC=TEST,DC=cz
	CN=CZPRG-P-EXC01,OU=Servers,OU=Computers,OU=_TEST,DC=TEST,DC=cz
RDN	CN=Zalewski Adam
Ou	OU=Operations,OU=_Users,OU=_TEST,DC=TEST,DC=cz
memberOf	CN=Exchange Windows Permissions,OU=Microsoft Exchange Security Groups,DC=TEST,DC=cz
	CN=Role Group SER2,OU=User roles,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=ACL Disk S&PS share Full Access,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=Domain Users,CN=Users,DC=TEST,DC=cz
	CN=Account Operators,CN=Builtin,DC=TEST,DC=cz
	CN=ACL TEST domain users,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
RDN	CN=Růžičková Vlasta
Ou	OU=Engineering,OU=_Users,OU=_TEST,DC=TEST,DC=cz
memberOf	CN=Exchange Windows Permissions,OU=Microsoft Exchange Security Groups,DC=TEST,DC=cz
	CN=Role Group ENG2,OU=User roles,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=ACL Disk Stag Full Access,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=ACL Disk Public Projects Full Access,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=ACL Disk Public Skill Matrix - Full Access,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=ACL TEST domain users,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=Domain Users,CN=Users,DC=TEST,DC=cz
	CN=ACL TEST domain users,OU=Access rights,OU=Groups,OU=_TEST,DC=TEST,DC=cz
	CN=Engineering - TEST,OU=Distribution,OU=Groups,OU=_TEST,DC=TEST,DC=cz

Table 7.4 Examples of generated honeyusers and predicted direct predecessors in the real AD

Chapter 8

Conclusion

In this thesis we showed how modern machine learning techniques for graph processing can be utilized in honeypot deployment, mainly in the Active Directory domain which is a common target of Advanced Persistent Threat (APT) attacks.

We introduced a framework which is capable of processing the existing structure of an Active Directory and propose where to place honeypot users. Such task is commonly done manually by system administrators or security professionals.

The core of the framework is a machine learning model based on the DAG-RNN module which utilizes the structure of the graph to create a node-level encoding. The DAG-RNN Encoder implemented in this thesis processes the complete graph in a single pass and is capable of working with graphs of arbitrary size. Additionally, we proposed, implemented and evaluated three model types using the DAG-RNN Encoder which differ in the Decoder module. All proposed models are implemented for the Tensorflow 2 library and they are compatible with the standard API of the library.

Given the sensitivity of the data stored in the AD and subsequent complications with the collection of a dataset large enough for the model training, we created four artificial AD User Related Graph Datasets. Expert knowledge was used in the generation process to ensure that the datasets contain valid structures for model training and evaluation. The generated datasets differ in maximal size of the graphs which was used to test the models' scalability.

We evaluated proposed models in two experimental directions: modelling of the AD structures and generation of additional nodes for existing structures. In the first experiment we showed that while all models perform well in the small-sized graphs, with an increasing amount of nodes in the structure, Model 3, which is based on Variational Autoencoder architecture, has superior results with F1 score over 0.6 and recall over 70%. In the same experiment we demonstrated that while the DAG-RNN is suitable for DAG node encoding, a model using only the target node type is not able to scale up to the

sizes of graphs common in the real-world AD structures.

In the generative experiments, we showed that the clear dominance of the VAE model in the structural modelling task, does not imply the same results in the existing graph extension task. We showed that the performance of the models depends heavily on the threshold parameter. Models 2 and 3 showed promising results both in small and large graphs, producing node placements with similar properties to the nodes in the original structures. In particular, Model 2 generated over 50% of valid edges and the Model 3 achieved 67.5% of edge validity in the largest dataset while keeping the mean amount of edges per node close to the value in the original graph.

The main outcome of this thesis is a novel framework for honeypot placement in an AD domain. Moreover, we created a free software implementation of a DAG-RNN model which can process Directed Acyclic Graphs of arbitrary size. Our implementation is compatible with Tensorflow 2, one of the leading ML frameworks in the industry, and its GPU acceleration.

8.1 Future Work

There are several directions for future development of the proposed framework. One of the main drawbacks is the memory inefficiency of the data representation in the current implementation. With Tensorflow 2 support of Sparse tensor operations and the fact that the majority of the adjacency matrix elements are zeros, a transition to the sparse representation would enable processing even bigger structures using the same hardware.Another implementation aspect to be addressed in the future is that of full compatibility with the Tensorflow API, especially the sequential API and the default pipeline. In the current state, the Sequential API is not applicable due to the output shapes of the custom DAG-RNN encoder layer which is designed to process graphs of arbitrary size.

In the model design area, future plans include involving the previous predictions into the decision making process making the model aware of the other predicted edges for a candidate node. Such direction could further improve both the autoencoding and generative capabilities of the model. The same applies for including additional node features as well as extra node types in the model design.

One of the most important future steps is an evaluation with real users and attackers or penetration testers. Since honeypots are mainly focused on rational attackers, testing within a real-world environment is crucial for the proper evaluation of the honeypot quality.

Despite showing that the framework produces object with similar properties as the same type objects in the original graph, further evaluation with security professionals,

penetration testers and red-team members is necessary to for conclusive analysis of the quality of the generated honeyusers.

Appendix A

Detailed experiment results

A.1 AD Structure Modelling

A.1.1 Auto encoder with variable z-dimension

Data from the experiment that is focused on the variable latent space size and its influence on DAG-RNN based autoencoder reconstructive capabilities.

z dimension	Precision	Recall	F1 Score	AUC (P-R curve)
$\|z\| = 2$	0.4118	0.4489	0.4295	0.4532
$\|z\| = 8$	0.3968	0.7795	0.5259	0.6167
$\|z\| = 32$	0.5732	0.9302	0.7093	0.8325
$\|z\| = 128$	0.6686	0.9377	0.7806	0.8885

Table A.1 Evaluation of the influence of the latent space dimensionality on DAG-RNN Autoencoder performance (Dataset 50)

z dimension	Precision	Recall	F1 Score	AUC (P-R curve)
$\|z\| = 2$	0.2087	0.1418	0.1689	0.1498
$\|z\| = 8$	0.2556	0.2948	0.2738	0.2494
$\|z\| = 32$	0.3228	0.4886	0.3887	0.4007
$\|z\| = 128$	0.47099	0.8382	0.6031	0.7264

Table A.2 Evaluation of the influence of the latent space dimensionality on DAG-RNN Autoencoder performance (Dataset 150)

A.1.2 δ hyper-parameter of Huber loss

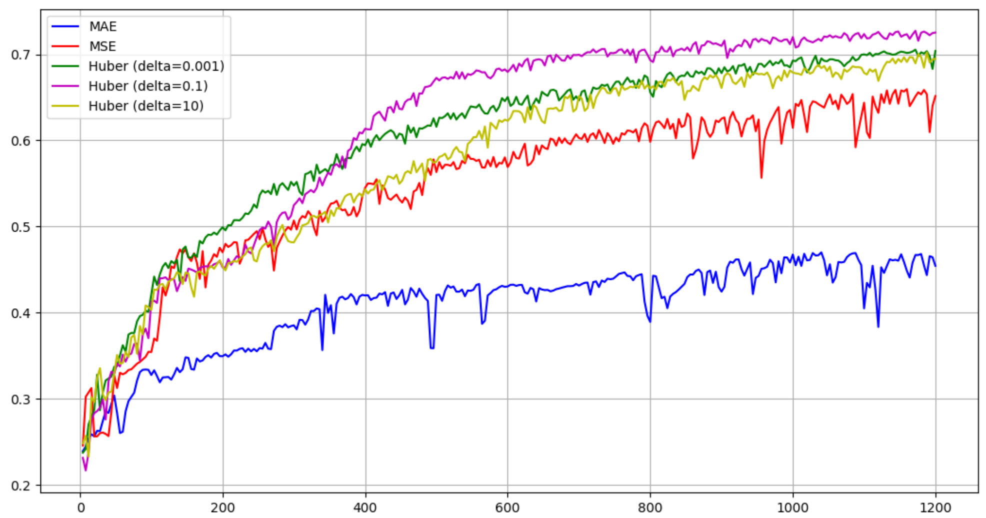

Figure A.1 | **δ hyper-paremeter tuning for Model 2.** AUC of the PR curve using various δ values. MSE and MAE shown for comparison.

A.1.3 Model comparison with Datasets 150 & 500

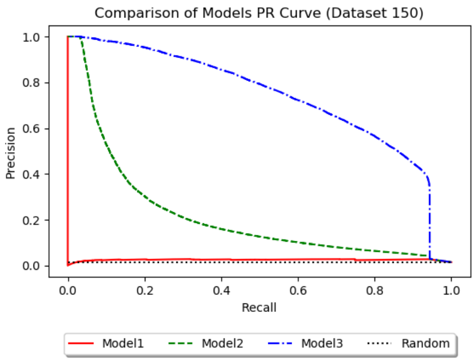

Figure A.2 | **PR curve comparison (Dataset150).** Graph showing the Precision-Recall curve for each of the model for Dataset150. andom model is shown as a baseline.

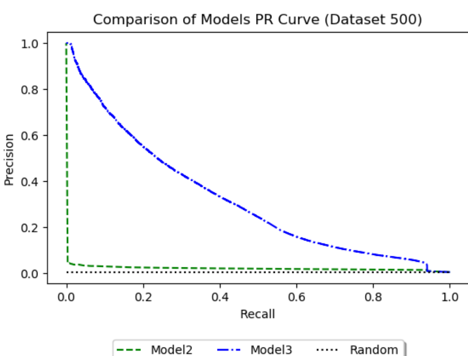

Figure A.3 | **PR curve comparison (Dataset500).** Graph showing the Precision-Recall curve for each of the model for Dataset500. Random model is shown as a baseline.

References

[1] T. Norris, *Perspectives: It's Not If, It's When, A Cyber Attack Will Hit.* [Online]. Available: https://www.rsa.com/en-us/blog/2019-06/perspectives-its-not-if-its-when-a-cyber-attack-will-hit.

[2] National Security Agency of the United States of America (NSA), *Defense in Depth*, 2010. [Online]. Available: https://apps.nsa.gov/iaarchive/library/ia-guidance/archive/defense-in-depth.cfm (visited on 11/08/2020).

[3] C. Cimpanu, *Hackers breached A1 Telekom, Austria's largest ISP.* [Online]. Available: https://www.zdnet.com/article/hackers-breached-a1-telekom-austrias-largest-isp (visited on 11/08/2020).

[4] Z. Whittacker, *Hackers went undetected in Citrix's internal network for six months.* [Online]. Available: https://techcrunch.com/2019/04/30/citrix-internal-network-breach (visited on 11/08/2020).

[5] C. Cimpanu, *Fortune 500 company NTT discloses security breach.* [Online]. Available: https://www.zdnet.com/article/fortune-500-company-ntt-discloses-security-breach (visited on 11/08/2020).

[6] K. Zetter, *Sony Got Hacked Hard: What We Know and Don't Know So Far.* [Online]. Available: https://www.wired.com/2014/12/sony-hack-what-we-know (visited on 11/08/2020).

[7] J. Crabtree, *Active Directory Attacks Hit the Mainstream.* [Online]. Available: https://www.darkreading.com/endpoint/authentication/active-directory-attacks-hit-the-mainstream/a/d-id/1337405 (visited on 11/08/2020).

[8] S. Metcalf, *Red vs. Blue: Modern Active Directory Attacks, Detection, & Protection*, 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Metcalf-Red-Vs-Blue-Modern-Active-Directory-Attacks-Detection-And-Protection-wp.pdf (visited on 11/08/2020).

[9] Microsoft, *What threats does ATA look for?* [Online]. Available: https://docs.microsoft.com/en-us/advanced-threat-analytics/ata-threats (visited on 11/08/2020).

[10] R. Nurfauzi, *Active Directory Kill Chain Attack & Defense*, 2020. [Online]. Available: https://github.com/infosecn1nja/AD-Attack-Defense (visited on 11/08/2020).

[11] A. P. de Barros, *RES: Protocol Anomaly Detection IDS - Honeypots.* [Online]. Available: https://seclists.org/focus-ids/2003/Feb/95 (visited on 11/08/2020).

[12] X. Han, N. Kheir, and D. Balzarotti, "Deception techniques in computer security: A research perspective", *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–36, 2018.

[13] "ISO/IEC 9594-1:2017 Information technology — Open Systems Interconnection — The Directory — Part 1: Overview of concepts, models and services", 2017.

[14] https://ldapwiki.com/, *ldapwiki.com*. [Online]. Available: https://ldapwiki.com/wiki/Microsoft%20Active%20Directory%20Attributes (visited on 07/02/2020).

[15] Microsoft, *Active Directory Security Groups*. [Online]. Available: https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/active-directory-security-groups (visited on 11/08/2020).

[16] A. Robbins and R. Vazarkar, *Bloodhound*. [Online]. Available: https://github.com/BloodHoundAD/BloodHound (visited on 09/07/2020).

[17] Microsoft, *Powershell*. [Online]. Available: https://docs.microsoft.com/en-us/powershell (visited on 12/06/2020).

[18] https://neo4j.com/, *Neo4j*. [Online]. Available: https://neo4j.com/ (visited on 12/06/2020).

[19] A. Greenberg, *The Untold Story of NotPetya, the Most Devastating Cyberattack in History*. [Online]. Available: https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world (visited on 11/08/2020).

[20] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[21] G. Palm, "Warren McCulloch and Walter Pitts: A Logical Calculus of the Ideas Immanent in Nervous Activity", in *Brain Theory*, G. Palm and A. Aertsen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 229–230, ISBN: 978-3-642-70911-1.

[22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[23] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[24] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, *On Empirical Comparisons of Optimizers for Deep Learning*, 2019. arXiv: 1910.05446 [cs.LG].

[25] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, 2014. arXiv: 1412.6980 [cs.LG].

[26] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function", *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.

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

[28] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014. arXiv: 1406.1078 [cs.CL].

[29] Kowsari, J. Meimandi, Heidarysafa, Mendu, Barnes, and Brown, "Text Classification Algorithms: A Survey", *Information*, vol. 10, no. 4, p. 150, Apr. 2019, ISSN: 2078-2489. DOI: 10.3390/info10040150. [Online]. Available: http://dx.doi.org/10.3390/info10040150.

[30] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, *Massive Exploration of Neural Machine Translation Architectures*, 2017. arXiv: 1703.03906 [cs.CL].

[31] classic.d2l.ai, *classic.d2l.ai*. [Online]. Available: https://classic.d2l.ai/chapter_recurrent-neural-networks/bi-rnn.html (visited on 09/07/2020).

[32] T. Pevny and P. Somol, *Discriminative models for multi-instance problems with tree-structure*, 2017. arXiv: 1703.02868 [cs.CR].

[33] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola, *Deep Sets*, 2017. arXiv: 1703.06114 [cs.LG].

[34] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, *Relational inductive biases, deep learning, and graph networks*, 2018. arXiv: 1806.01261 [cs.LG].

[35] Avinash Hindupur, *The GAN zoo*. [Online]. Available: https://github.com/hindupuravinash/the-gan-zoo (visited on 11/08/2020).

[36] S. J. Russell and P. Norvig, *ECIW2006-Proceedings of the 5th European Conference on i-Warfare and Security: ECIW 2006*. Academic Conferences Limited, Jan. 2019, p. 286, ISBN: 9781905305209.

[37] Microsoft, *https://docs.microsoft.com/en-us/advanced-threat-analytics*. [Online]. Available: https://docs.microsoft.com/en-us/advanced-threat-analytics/ (visited on 09/07/2020).

[38] https://github.com/leeberg, *https://github.com/leeberg/BlueHive*. [Online]. Available: https://github.com/leeberg/BlueHive (visited on 09/07/2020).

[39] C. Leita, K. Mermoud, and M. Dacier, "ScriptGen: an automated script generation tool for Honeyd", in *21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005, 12 pp.–214.

[40] http://www.honeyd.org/, *www.honeyd.org*. [Online]. Available: http://www.honeyd.org/ (visited on 07/05/2020).

[41] S. Dowling, M. Schukat, and E. Barrett, "Using Reinforcement Learning to Conceal Honeypot Functionality", in *ECML/PKDD*, 2018.

[42] W. Tian, X.-P. Ji, W. Liu, J. Zhai, G. Liu, Y. Dai, and S. Huang, "Honeypot game-theoretical model for defending against APT attacks with limited resources in cyber-physical systems", *ETRI Journal*, vol. 41, no. 5, pp. 585–598, 2019.

[43] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks", *IEEE Transactions on Neural Networks and Learning Systems*, 1–21, 2020, ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.2978386. [Online]. Available: http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[44] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, *GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models*, 2018. arXiv: 1802.08773 [cs.LG].

[45] M. Simonovsky and N. Komodakis, *GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders*, 2018. arXiv: 1802.03480 [cs.LG].

[46] R. Liao, Y. Li, Y. Song, S. Wang, C. Nash, W. L. Hamilton, D. Duvenaud, R. Urtasun, and R. Zemel, "Efficient Graph Generation with Graph Recurrent Attention Networks", in *NeurIPS*, 2019.

[47] S. Amizadeh, S. Matusevych, and M. Weimer, "Learning To Solve Circuit-SAT: An Unsupervised Differentiable Approach", in *International Conference on Learning Representations*, 2019. [Online]. Available: `https://openreview.net/forum?id=BJxgz2R9t7`.

[48] M Kaluza, C. De Paolis, S. Amizadeh, and R. Yu, "A neural framework for learning DAG to DAG translation", in *NeurIPS'2018 Workshop*, 2018.

[49] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. Smola, and Z. Zhang, *Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs*, 2019. arXiv: `1909.01315 [cs.LG]`.

[50] D. Grattarola and C. Alippi, *Graph Neural Networks in TensorFlow and Keras with Spektral*, 2020. arXiv: `2006.12138 [cs.LG]`.

[51] I. Siniosoglou, G. Efstathopoulos, D. Pliatsios, I. Moscholios, A. Sarigiannidis, G. Sakellari, G. Loukas, and P. Sarigiannidis, *NeuralPot: an industrial honeypot implementation based on convolutional neural networks*, Apr. 2020. [Online]. Available: `http://gala.gre.ac.uk/id/eprint/27976/`.

[52] Sense of Security, *ADRecon*. [Online]. Available: `https://github.com/sense-of-security/ADRecon` (visited on 11/08/2020).

[53] Javelin Networks, *Honeypot Buster*. [Online]. Available: `https://github.com/JavelinNetworks/HoneypotBuster` (visited on 11/08/2020).

[54] BloodHoundAD, *https://github.com/BloodHoundAD/SharpHound*. [Online]. Available: `https://github.com/BloodHoundAD/SharpHound` (visited on 07/05/2020).

[55] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient Estimation of Word Representations in Vector Space*, 2013. arXiv: `1301.3781 [cs.CL]`.

[56] A. Grover and J. Leskovec, *node2vec: Scalable Feature Learning for Networks*, 2016. arXiv: `1607.00653 [cs.SI]`.

[57] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, *Focal Loss for Dense Object Detection*, 2017. arXiv: `1708.02002 [cs.CV]`.

[58] T. Oliphant, *NumPy: A guide to NumPy*, USA: Trelgol Publishing, 2006–. [Online]. Available: `http://www.numpy.org` (visited on 11/08/2020).

[59] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using", in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 –15.

[60] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[61] Willeke, J and Wolkhart,M, *LDIFGenerator*. [Online]. Available: `https://github.com/jwilleke/LDIFGenerator` (visited on 11/08/2020).

[62] C. Works, *Fake Name Generator*. [Online]. Available: `https://www.fakenamegenerator.com` (visited on 11/08/2020).