UNIVERSITY OF CAPE TOWN

MASTERS THESIS

Using machine learning to guide automated intrusion response

Author:
Andre LOPES

Supervisor:
Dr. Andrew HUTCHISON

A thesis submitted in partial fulfillment of the requirements for the degree of Masters in Computer Science

in the

Department of Computer Science

October 24, 2019

UNIVERSITY OF CAPE TOWN

Abstract

Faculty of Science
Department of Computer Science

Masters in Computer Science

Using machine learning to guide automated intrusion response

by Andre LOPES

Traditionally Intrusion Response Systems (IRSs) have had a strong reliance on network administrators to perform various responses for a network. Though this is expected, particularly with networks containing sensitive data, it is not completely practical, considering the ever growing demand for speed, scalability and automation in computer networks. This work presents a proof of concept automated IRS that provides both for networks containing sensitive data and high speed networks, by using basic responses for complex attacks, and by using reinforcement learning for direct attacks. Responses for the latter are done by creating a response system that is able to learn from the effectiveness of its own responses. This work is evaluated in its effectiveness against the deactivation issue, which is concerned with the problem of automatically deactivating network responses after they've been activated by an IRS. All tests are conducted using an emulated network, that was designed to replicate real network behavior. Simulated attacks were used to train the IRS. Results of training were evaluated at intervals of 100, 500, 1000 and 2000 attacks. The findings of this work indicate that while applying reinforcement learning to IRSs is feasible, adjustments may still be required to improve its performance.

Acknowledgements

I'd like to thank my family for their support, and the encouragement and guidance given to me by my supervisor.

Contents

List of Figures

List of Tables

List of Abbreviations

DMZ	**D**e**m**ilitarized **Z**one
DoS	**D**enial **o**f **S**ervice
GNS3	**G**raphical **N**etwork **S**imulator **3**
GPU	**G**raphical **P**rocessing **U**nit
GUI	**G**raphical **U**ser **I**nterface
IDS	**I**ntrusion **D**etection **S**ystem
IP	**I**nternet **P**rotocol
IRS	**I**ntrusion **R**esponse **S**ystem
ISP	**I**nternet **S**ervice **P**rovider
MTTD	**M**ean **T**ime **t**o **D**etect
MTTR	**M**ean **T**ime **t**o **R**espond
NAT	**N**etwork **A**ddress **T**ranslation
NIC	**N**etwork **I**nterface **C**ard
OSI	**O**pen **S**ystems **I**nterconnection
PPO	**P**roximal **P**olicy **O**ptimization
RAID	**R**edundant **A**rray of **I**nexpensive **D**isks
RQ	**R**esearch **Q**uestion
SAO	**S**ecurity **A**utomation and **O**rchestration
SCP	**Sc**ure co**p**y
SOAR	**S**ecurity **O**rchestration, **A**utomation and **R**esponse
SSH	**S**ecure **Sh**ell
TCP	**T**ransmission **C**ontrol **P**rotocol
TRPO	**T**rust **R**egion **P**olicy **O**ptimization
URL	**U**niform **R**esource **L**ocator
vNIC	**v**irtual **N**etwork **I**nterface **C**ard

Chapter 1

Introduction

Network security is a prerequisite for any computer network [3], especially those containing sensitive data. IRSs are a key component in defending networks as they either prevent an attack from happening, or stop an attack from succeeding. IRSs are often accompanied by Intrusion Detection Systems (IDSs), which provide alerts about suspicious network behavior or data. IRSs make use of alerts, along with response policies, to best decide how to respond to an attack. Response policies make changes to a network to counter attacks, and are either enacted manually by a network administrator, or automatically by an IRS. These policies are, however, often not deactivated automatically, and, depending on the type of response, can cause damage to the network if not appropriately deactivated by a network administrator.

1.1 Research description

The problem of determining the best time to deactivate a response policy, known as the *deactivation issue*, is challenging and has scarcely been researched. A related approach to this issue is to determine the likelihood of attacks being successful, and based off of the likelihood value, use an upper-bound and lower-bound threshold to determine when response policies need to be activated and deactivated [20]. However, the issue of what value to set the upper-bound and lower-bound values to, still remains.

A potential solution to this problem is to use machine learning to determine when to enable and disable response policies. The challenge with machine learning is that there is no certainty to the values they output, and as a result, could cause damage to the network they protect with the decisions they make. For example, the upper-bound and lower-bound values may be set too closely to one another, causing response policies to be rapidly activated and deactivated, causing a degradation in network performance. This is the reason why machine learning, and more specifically, even reinforcement learning, is applied to IDSs [7], where mistakes are often tolerated in the form of false positives, but rarely applied to IRSs, where mistakes directly affect the network.

The issue of deactivating response policies with thresholds is an example of where metrics have to be set prior to running an automated IRS. This problem extends to many IRSs that use metrics to establish how aggressively they'll respond to a potential threat. The main issue with using such metrics, is that there is not always a definitive method used to set them [34]. In this work, we try to counteract this by using reinforcement learning to learn these metrics automatically. To avoid previous issues related to machine learning IRSs, the problem will be approached with

prediction in mind, whereby thresholds will not be estimated, but instead, deactivation/isolation times will. Additionally, responses will only be applied to Denial of Service (DoS) attacks in order to minimize the potential damage to the protected network in the event that the prediction was incorrect. Examples of mitigation strategies for other attack types will be provided, but only as a proof of concept to showcase a more complete IRS.

This work, in addition to introducing an automated IRS, that learns from the effectiveness of its own responses, and that uses reinforcement learning to respond to potential threats, will also present results on its effectiveness in solving the deactivation issue. If successful, the IRS will be able to reduce the reliance on network administrators in managing network attacks and will also minimize service downtime.

1.2 Research aims

The following work aims to create an IRS that uses reinforcement learning, a form of machine learning, to determine when to enable and disable response policies. The reinforcement learning algorithm will not be able to be trained from an existing dataset, and will have to learn from the effectiveness of its own responses. To avoid having the IRS damage the network it protects, especially during its training phases where mistakes are prone, reinforcement learning models will only be applied to DoS attacks. Additional responses, which do not make use of reinforcement learning, will be used to defend against more sensitive types of attacks.

Additionally, a test-bed will be created to benchmark the performance of the reinforcement learning algorithms, as well as to present the applicability of the sensitive response types.

1.3 Research questions

RQ1 Can an Intrusion Response System (IRS) be created, such that, it will *respond appropriately according to the type of attack?*

RQ2 Can an Intrusion Response System (IRS) be created, such that, it will *learn from the effectiveness of its own responses?*

RQ3 Can an Intrusion Response System (IRS) be created, such that, it will *deactivate network policies at optimal times?*

1.4 Thesis structure

As stated in the above sections, this thesis will present information relating to the creation and evaluation of an Intrusion Response System (IRS). The purpose of its creation, surrounding background knowledge, how it was created and how it performed will be discussed in 6 chapters. This chapter, the introduction (1), explains the context for why the research is being conducted. This context includes how the IRS fits into existing works, and how it will contribute to them. Consequently, research aims and Research Questions (RQs) that are relevant to this thesis are identified. The following chapter, background (2), discusses related works and topics. It

is particularly useful for understanding the technical aspects of the IRS and its construction, however, it may also provide further context for how the IRS contributes to existing works. The following two chapters, implementation (3) and experiment (4), describe the approach used to answer the various RQs. These two chapters constitute the core of the research being performed. The remaining two chapters, results (5) and, conclusion and future work (6), discuss what was found by conducting the research for all tests, and finally gives concluding remarks about the research and its limitations.

There is no defined reading order and introductions will be included in each chapter to summarize what has been discussed, and what is going to be discussed. The reader is, however, encouraged to refer to the background (2) and implementation (3) chapters, before reading the experiment chapter (4), as these preceding chapters may provide useful context.

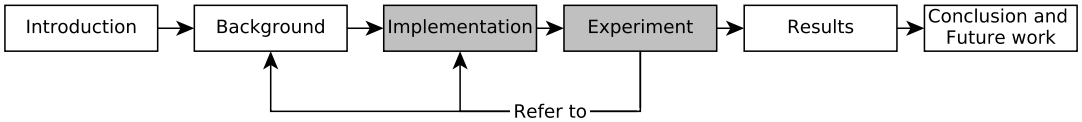

FIGURE 1.1: Thesis overview

1.5 Summary

This chapter has discussed the aims of this work and has mentioned how it relates to existing works. IRSs that respond automatically to network attacks with the use of reinforcement learning are not frequently seen in existing works. Furthermore, many current implementations of IRSs make use of frameworks, which, while being capable of optimization through machine learning, contain various problems that could negatively affect network performance. The IRS created in this work will learn from previous attacks and the knowledge that it learns will be tested by attempting to automate the activation and deactivation process that normally proves problematic in what is known as the deactivation issue.

Chapter 2

Background

The previous chapter (1) introduced a problem with IRSs, whereby, responses that are activated automatically are not deactivated automatically. The problem is known as the deactivation issue, and it can negatively affect a network if not handled correctly by a network administrator. The previous chapter also mentions how this work will reduce reliance on network administrators while attempting to provide a solution to the deactivation issue. These two are accomplished through the use of machine learning in IRSs, which is not often used due to the uncertainty of the values that machine learning models output. It is for this reason that machine learning will only be applied to DoS attacks, to minimize the damages inflicted on the network when the IRS responds incorrectly. To complete the IRS, other types of attacks will be mitigated, but without the use of machine learning.

This chapter provides technical information that is most useful in the experiment chapter (4), but may also aid in situating this work with regards to existing works. The information discussed describes the basic structure of intrusion response (2.1), and how learning can be performed in the form of supervised learning (2.2) and reinforcement learning (2.3). These three sections discuss the majority of the knowledge needed to understand the IRS in the experiment chapter, however, the following two sections, network emulation (2.4) and digital forensics (2.5), are still useful for understanding the test-bed and the different responses that were created, respectively. The remaining section on SOAR (2.6) provides information on a network security approach, that while not implemented in this work, does share the similar aim of reducing administrator involvement, and becomes particularly relevant when discussing future directions of this work.

2.1 Intrusion response

Intrusion response is the field concerned with responding to network attacks when given an alert. These systems work alongside IDSs, which are responsible for identifying network attacks and subsequently generating alerts. Like IDSs, IRSs are implemented using various methods. Taxonomies exist to classify these methods and the features they provide. These taxonomies are often similar, and contain the same concepts, however, the descriptions of these similar concepts differ. The following section attempts to simplify the descriptions of the various concepts in IRSs while providing a taxonomy (figure 2.1) for reference.

2.1.1 Levels of automation

Response systems can be placed into three categories. They can exist as notification systems, manual response systems or automated response systems. These categories

create what is known as the levels of automation [19, 31, 32, 33].

- *Notification systems* act to alert a network administrator about an attack. Administrators are required to respond to the attack themselves.
- *Manual response systems*, which in addition to alerting the network administrator about an attack, also provides a set of pre-configured responses, requiring less human intervention.
- *Automated response systems* do not require any human interaction as they respond to attacks independently.

The first two categories (notification and manual response systems) can be considered passive response systems, and can therefore be handled by an IDS. The last category (automated response system) is an active response system, and can only be handled by an IRS [1, 2, 3].

The remaining sections will list and describe further distinctions between automated response systems. For clarity, figure 2.1 has been created to summarize the information presented.

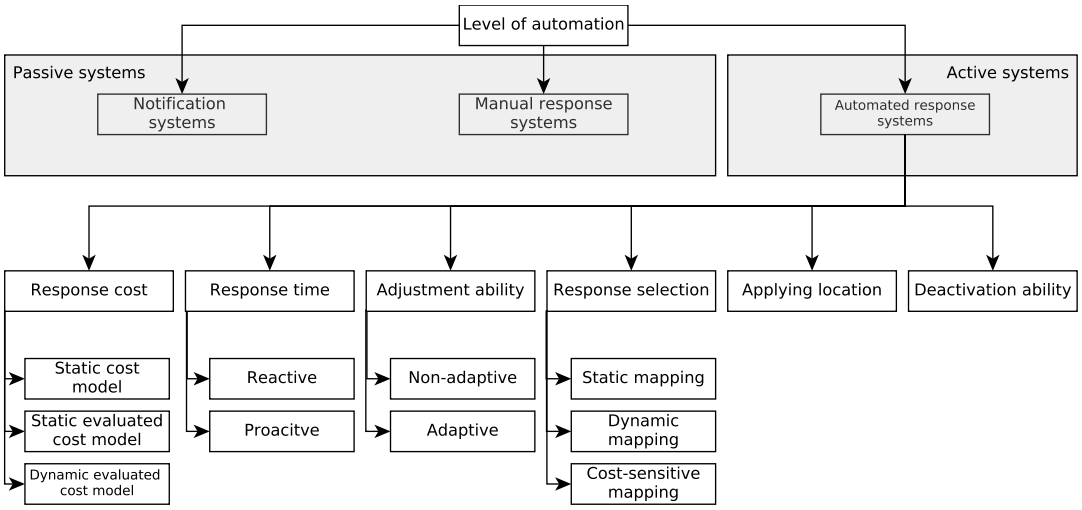

FIGURE 2.1: Our assessment of an IRS taxonomy

2.1.2 Adjustment ability

The ability to learn from previous responses and how effective they are, determines whether a response system is able to adjust or not. A response system can either be non-adaptive or adaptive [19, 31, 32, 33].

- *Non-adaptive* systems will always respond in the same way for the remainder of their execution. This does not change unless an administrator manually intervenes.
- *Adaptive* systems will respond differently throughout the duration of the system's execution. The reasoning behind why they adapt can vary. The system could learn from previous attacks, or implement a cost-sensitive response selection model (see section 2.1.3), or both. It could also start receiving different inputs from another IDS on the network, which would cause the responses to change.

2.1.3 Response selection

An automated IRS needs a way of responding to an attack. It can do this either statically, dynamically or through a cost-sensitive measure [19, 31, 32, 33].

- *Static mapping* determines a response based on the type of attack. While simple to build, this can be exploited, as attackers can learn how the system responds to certain actions. If the system responds by shutting down services, or dropping packets, attackers can use this to perform a DoS attack, whereby legitimate users will not be able to make use of the service.
- *Dynamic mapping* responds to an attack by taking into account the overall state of the network. For instance, if an attack is severe, and is affecting multiple services, more aggressive defense measures may be implemented. The approaches used to implement dynamic mapping are varied, however, none of them gain knowledge from an attack [28].
- *Cost-sensitive mapping* is a unique approach, as it tries to balance the damage caused by attackers, with the damage caused by the response system, since both can lead to a disrupted network. This method is often reliant on a cost model (see section 2.1.5) that determines the cost, or damage, that a specific response will inflict. The best response will be the one that has the lowest cost associated with it. Much like dynamic mapping, the approaches used in this model vary, though unlike dynamic mapping, this model has the potential to learn from previous attacks.

2.1.4 Response time

A response to an attack can occur either before or after an attack has happened. Intrusion response systems can be classified into reactive or proactive, depending on their approach [19, 31, 32].

- *Reactive* systems respond to an attack after it has occurred. They are often used to train a response algorithm and are rarely implemented in practice, as they fail to provide confidentiality for the data they're protecting.
- *Proactive* systems make predictions about attacks. While they do provide full security for a network, they are not always perfect, and can miss attacks, or even needlessly interrupt the network for attacks that won't happen.

2.1.5 Response cost

All responses to attacks have the ability to disrupt a network. It is possible to associate a response cost to these responses, which can then be used by other systems (such as in section 2.1.3). Response cost can be calculated using a static cost model, static evaluated cost model or a dynamic evaluated cost model [31].

- *Static cost model* systems use expert knowledge to assign costs to each response. The cost does not change throughout the execution of the system.
- *Static evaluated cost model* systems use an evaluation mechanism to assign cost to responses. This evaluation mechanism measures the positive and negative effects based on confidentiality, integrity, availability and performance, and uses these as inputs. Like the static cost model, this approach also does not change throughout the execution of the system.

- *Dynamic evaluated cost model* systems evaluate cost based on the state of the network. The cost of a response is increased if it interrupts a process with many dependencies. Unlike the static and static evaluated cost models, this approach will change the cost of responses throughout the execution of the system.

2.1.6 Applying location

Intruders in a network will need to have an attack path associated with their intrusion. The attack path determines how the intruder managed to reach the target machine on the network. Attack paths consist of four points:

- The start point (or intruders machine)
- The firewall point
- The midpoint (associated with intermediary machines that are exploited)
- The end point (the target)

An effective IRS will apply responses to the points that result in the lowest cost for the network (see section 2.1.5), though very few IRSs actually implement this feature [31].

2.1.7 Deactivation ability

It is often the case that a response will only need to be applied for a temporary period of time, after which, the response should be deactivated, and the network restored to its normal operational order. Deactivation ability is not a common feature included in most IRSs, since it is a complex problem to decide when and how to deactivate a response [20, 31].

2.2 Supervised learning

Supervised learning is a machine learning approach which analyzes data in order to create generalizations, which can be used to solve various prediction or classification problems. These generalizations are created through the means of various supervised learning algorithms. Most algorithms are only suited to specific problems, which is why careful selection of a supervised learning algorithm must be made. Data is also important for the effectiveness of supervised learning tasks, which is why data selection and preprocessing is also necessary. The following section will cover these aspects by providing a basic summary of the problems and processes surrounding supervised learning.

2.2.1 Relevant problems for supervised learning

Machine learning is applicable to problems that contain large amounts of data. The field of data mining, where new data is created from the analysis of existing data, is an example of one of the applications of machine learning [22]. Many tools exist, such as sci-kit learn and Weka, which further allow machine learning to be used not only by researchers in academia, but end users in the business world as well [15, 17, 27]. Supervised learning, in specific, is a machine learning approach that analyzes labeled data, which is data that contains information on what the output, to certain

inputs, should be (see section 2.2.2). Examples of its use are in medicine to classify an illness, or astronomy to classify a galaxy, or even finance to predict market prices. As can be seen, supervised learning is applicable to many fields, and consequently, many types of problems.

2.2.2 Dataset structure

Datasets, in the context of supervised learning, contain features (columns) and instances (rows), where each feature has a data type that is either continuous, categorical or binary. In addition, these datasets also contain a special feature known as the label or output, which is the feature that is to be predicted when the supervised learning algorithm (better known as a model, once it has learned) is presented with new data. A simple example of a dataset that collects information on various insects would be fit for supervised learning, whereby the features indicate aspects of an insect, such as the number of legs it has, or whether it is poisonous or not, and instances would represent the data collected for a single insect. The label or output field in each instance, would contain the name of the insect, thereby classifying it.

2.2.3 Dataset construction and pre-processing

Collecting the hypothetical insect dataset, mentioned in the previous section, might prove problematic, as not all insects share the same features, and some features may not sufficiently distinguish certain insects from one another. For example, a feature which records whether the sting of an insect is painful or not, would not be ideal, as not all insects sting, meaning that this feature would be irrelevant to them. Furthermore, the number of legs an insect has would also be a poor feature, as all insects have six legs, therefore, this feature would not distinguish insects from one another. Often, experts in the field are used to identify features for dataset construction, as this avoids the issues mentioned.

Once features and data have been built and obtained, it is preferable to make sure that the overall dataset won't cause learning problems. Two particular examples of problems that could arise are class imbalances and data dimensionality issues. The *class imbalance* issue occurs if instances of one specific label, or type of insect, are far more prominent than others types. Datasets with this issue will likely cause the supervised learning algorithm to develop a bias for the prominent types, or insect types [4]. The *data dimensionality* issue occurs if there are too many features in the dataset. The repercussions of the data dimensionality issues could result in slow learning times, high memory usage and noise while learning [21].

Though there are many issues with dataset construction, tools exist to aid in mitigating them. For example, feature construction is a tool that analyzes existing features to create new and more predictive features. Another example is feature selection, which is a tool that selects and keeps only the most predictive features among those that exist. These two tools, in addition to creating a better dataset, will also reduce the dimensionality of the data. More simple data cleaning tools also exist, which can aid in other issues. An example is a tool that duplicates instances of the labeled data that are not predominant in the dataset, which mitigates the class imbalance issue.

2.2.4 Training and testing

After a dataset has been properly constructed, training can begin. In supervised learning, training is performed with a training set and a testing set, whereby, one set is used to create generalizations of the output, and the other is used to test its performance. The training set and test set are often generated by splitting the original dataset into two sets. Normally, two thirds are assigned to training, while a third is assigned to testing. This form of training and test set construction is acceptable when the dataset available is large, however, in circumstances with small datasets, it may be preferable to perform cross-validation, whereby, the dataset is split into separate equal folds, and the algorithm is tested multiple times on all folds but one, with the last one being used for testing. This procedure is done until all folds have been the test fold once, at which point, the performances of all tests are summarized into a final performance, or error, score.

Algorithms can also be managed in addition to the dataset. The general procedure for training is to select an algorithm, and training parameters associated with the algorithm. Once selected, the combination is tested and the procedure is repeated for different training parameters until the best set of training parameters for that algorithm can be found. The entire procedure is then repeated again for another algorithm. This process of testing can be time consuming, particularly if many algorithms are being tested. Fortunately, procedures such as hyper-parameter optimization exist, which can automatically test each algorithm with different training parameters, and report back on the best combination. Auto-weka is an example of a tool that has this ability [36].

2.3 Reinforcement learning

Reinforcement learning, often regarded as a machine learning method alongside supervised and unsupervised learning, is a method that learns from experience, or trial-and-error. It is often suited, and structured, for problems that involve the following:

- A goal
- An action to be taken
- Some form of sensing to be done

Not all reinforcement learning algorithms are the same though, as each have their own unique parameters and are often suited for specific problems [16, 35].

2.3.1 Differences of reinforcement learning

Supervised learning is a form of learning where the outputs are trained to be generalized. An example of the generalization of supervised learning is perhaps best seen in a decision tree, where data is continuously split into separate, non-overlapping sections, until a complete tree is built. Pruning, which is the process of limiting the tree to a specific depth, is then performed to generalize the outputs so that the decision tree does not over-fit the training data, which is necessary for the decision tree to be effective on new data. Reinforcement learning by contrast does not generalize outputs, but instead maximizes rewards.

Unsupervised learning differs from reinforcement learning as well, as its intention is to uncover hidden structures within the data. K-means is an example an unsupervised learning algorithm, where data is clustered into K clusters. The information revealed from K-means indicates which data-points belong to which clusters. The difference is similar to supervised learning, in that nothing is maximized, though it is possible to use the results obtained from an unsupervised learning algorithm to aid a reinforcement learning algorithm.

In addition to the differences mentioned, reinforcement learning is also well equipped to handle interactive and non-stationary problems. An example of this is a chess game, where the learning algorithm needs to move a chess piece, even if it has no knowledge of the board configuration before it, and also has to adapt to the potentially changing play-style of its opponent.

To summarize, all methods can potentially handle the same input, but as output, the supervised learning algorithm will generalize its results, the unsupervised learning algorithm will reveal information about the data, and the reinforcement learning algorithm will maximize rewards.

2.3.2 Relevant problems for reinforcement learning

Reinforcement learning can be adapted to a variety of problems, though not all make full use of its capabilities. For example, reinforcement learning can be used to select an action when given nothing but a reward as guidance, and it will still learn, however, learning would be much more efficient if given an input, or state information (see section 2.3.3). With some form of state information, correlations between the optimal actions and the current state, can be made. This is better known as *associative search*. Figure 2.2 shows the difference diagrammatically.

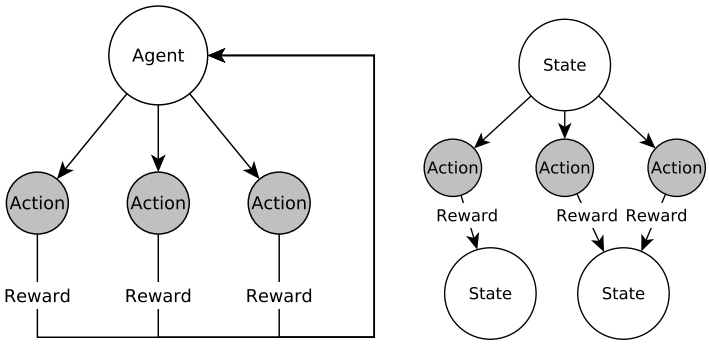

FIGURE 2.2: Diagram of learning from rewards (left) compared to learning from rewards and states (right)

Another capability of reinforcement learning is that of *delayed rewards*. Some problems may require a sense of foresight, whereby a number of specific actions will need to be performed in order to get the maximum reward. It is possible for reinforcement learning algorithms to select the best immediate reward, which would result in a greedy approach, and a less than optimal total reward. In order to obtain the best reward, some reinforcement learning algorithms will purposely select poor immediate rewards in order to get better rewards later on. Figure 2.3 shows an example of this.

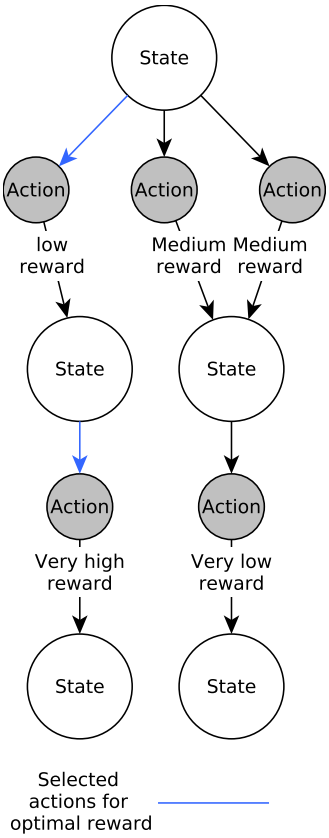

FIGURE 2.3: Example of learning with delayed rewards

Problems that incorporate both associative search and delayed rewards, are known to fully encompass the reinforcement learning problem.

The last capability mentioned is that of episodic or continuous tasks, though neither is necessarily better than the other with respect to rewards. Some problems will require episodes to represent one run, such as one game of a chess match. In episodic problems, there are always terminal states that represent the end of the episode, such as checkmate in chess. Other problems do not have terminal states, and instead run indefinitely, such as some control system, one that for instance would manage temperature in a room based on the temperature outside.

2.3.3 Reinforcement learning structure

The basic structure of a reinforcement learning scenario can be divided into three components: an agent, an environment and an action.

The *agent* is the primary component as it can be considered the reinforcement learning process itself. The agent remembers the states that the environment can be in and learns which actions to perform by analyzing the reward it receives from the environment. The *environment* is the elements surrounding the agent. The agent may understand the environment and how it works (e.g. rules of a game), but does not have absolute control over it, and can only influence it through actions. The *action* is the agent's choices and affects the interactions that the agent makes with the environment. The action may also change the states of the environment if the environment contains multiple states. In a game of chess, the environment would consist

of the board, each chess piece, and the rules governing their movements. The agent would learn the board configurations (as states) and invoke certain actions. The actions would be simple decisions on which pieces to move.

The basic components can be further de-constructed into four sub-elements: a policy, a reward signal, a value function and a model.

The *policy* defines how the agent will act in a given state. It specifies the probabilities of selecting specific actions in a state. The optimal policy for a problem will indicate which actions to choose to obtain the maximum reward for every state. The *reward signal*, or simply reward, comes from the environment and indicates whether the actions selected were good or bad. The signal is normally a real number. The *value function* can be used by various agents, or reinforcement learning algorithms, to estimate the reward it will get from performing an action, or a set of actions. There are variations to value functions, such as *policy gradient methods*, which learn probabilities instead of action-value mappings, and *actor-critic* methods, which use neural networks. All methods perform the same task, which is to estimate which action should be selected. The *model* is an instance of the agent that has been trained to some extent, and can be used to determine the next state and reward of the environment, given a certain action.

2.3.4 Parameters and trade-offs

Reinforcement learning algorithms contain a number of parameters, many of which specify trade-offs that need to be considered for different problems. The parameters discussed in this section are related to the reinforcement learning library, tensorforce [23], and are not necessarily the formal definitions used throughout the literature.

Learning rate is used to train the learning agent. It specifies the importance of more recent rewards, and if set to a high value, will more adversely affect how quick the agent adapts to changes in an environment. A higher learning rate will however make it more difficult for an agent to precisely converge to the correct value of a given state. Generally, learning rates are decreased over time, so as to quickly adapt initially, and then converge later on.

Discount is used in value functions for predicting rewards. It is primarily used for continuous tasks (see section 2.3.2), to make sure that the rewards are limited to some upper-bound. Without discount, the total reward would increase with every reward signal, up to infinity. Values for discount can range from 0 to 1, with values closer to 0 making the learning agent more concerned with immediate rewards, and values closer to 1 making the learning agent more concerned with future rewards. Values of 1 will ignore the discount completely, and make the learning agent perform as it does in episodic tasks.

Actions exploration is used in training and rewards. An agent can be exploitative, meaning that it will find an action that works well, and continuously select that action in the future. Alternatively, an agent can be explorative, and forgo the actions it knows about for actions that is doesn't know about. Exploration allows the learning agent to try different actions that may result in even better rewards than previous actions. Exploitation and exploration cannot be performed at the same time, and being explorative will negatively affect rewards initially. For this reason, learning

algorithms are generally set to explore initially, and then exploit later on. One parameter that affects this change is the chance of an action being explorative, which will decrease as the agent learns. Another parameter sets the number of time steps, or actions the agent will wait, before applying the chance parameter again. For example, in every 1000 actions, there will be a 50% chance that the agent selects an explorative action.

2.3.5 Reinforcement learning algorithms

It is not always possible to create an algorithm that performs well in any given problem [40], which is why many reinforcement learning algorithms adapt to the problem by learning a policy. Examples of such algorithms include: Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO), which have both proven to perform well on a variety of problems, irrespective of the problem's complexity. Unfortunately, not all reinforcement learning algorithms have managed to obtain the same adaptability that is achieved with TRPO and PPO, and will instead perform, for example, very well on complex problems, but poorly on simple problems, or, they will perform well on problems with discrete action spaces, but not on problems with continuous action spaces (where actions chosen are not limited) [29, 30, 37].

TRPO is an algorithm designed to be scalable, allowing it to perform well with a large amount of parameters, and reliable, allowing it to be useful on a number of problems without hyper-parameter optimization [29]. *PPO* expands the capabilities of TRPO by retaining its scalability and reliability, but also improves by reducing algorithm complexity and increasing compatibility with other reinforcement learning architectures [30].

2.4 Network emulation

Network emulation, sometimes referred to as network virtualization, though the term is overloaded [38], is the field of study that combines the technologies of virtualization and simulation to create a cost effective and realistic computer network. Though not a perfect solution when compared to other alternatives, network emulation still has its benefits and is particularly useful for testing purposes.

2.4.1 Network emulation structure

Network emulation is a testing approach between simulation and live testing. Some aspects of the testing environment or test-bed, are simulated technology, while other aspects consist of real technology. In general, the physical and data-link layers of the Open Systems Interconnection (OSI) model are simulated, while the layers in and above the network layer are real [24]. To give an example, virtualized Network Interface Cards (vNICs) emulate the hardware of real Network Interface Cards (NICs), and must replicate the encoding schemes used to convert digital data to a format that can be used on a communication medium (e.g. copper cable). Furthering the example, these vNICs must also replicate the multiplexing schemes used in the data-link layer (e.g. time division multiplexing) [38]. The remaining layers of the OSI model, in an emulated network, run real protocols and applications (e.g. Internet protocol for the network layer). Figure 2.4 represents the above example visually through the use of the Transmission Control Protocol (TCP)/Internet Protocol (IP) model.

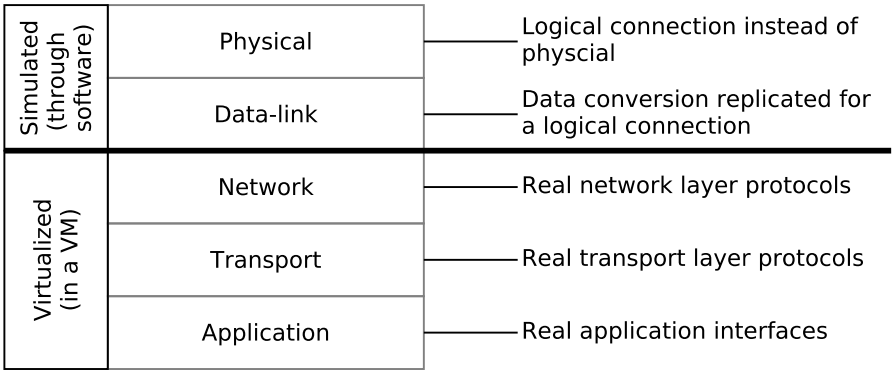

FIGURE 2.4: Example components of an emulated environment

What has been discussed in this section so far is a general description of emulation, however, emulation systems do vary in their execution. In particular, their behavior can be defined at three levels: the hardware level, the impairments level and the network level [24].

The *hardware* level considers whether the emulation is executed in a centralized or distributed system. Centralized systems are more common and simple to operate as synchronization issues between systems do not exist. Unfortunately centralized systems are limited in their computational power, which is a notable issue, considering that emulated networks do not scale well. The *impairments* level considers network impairments such as jitter, packet loss or bandwidth limits. Emulated systems are often perfect with regards to impairments, as the communication links between hosts in a network are software emulated. These links are generally only limited to the performance limitations of the host they're running on [38]. To create a realistic testing environment, these network impairments can be artificially added to the emulated network. Unfortunately, adding these impairments can introduce further unintentional impairments, such as the processing overhead that is required to add them. Impairments can be added either at the kernel or user level. In order to create reproducible results, these unintentional impairments must be accounted for and negated. The *network* level considers the scenario in which testing is conducted. Testing can be performed in a static emulation environment, where all parameters about the network (including impairments) remains constant throughout testing. Another approach is to use events to trigger changes to parameters in the network, such as specific conditions being met. The last approach of the emulated variant is the trace-based approach, where parameters of specific network configurations are obtained, and used for testing. These network configurations often represent some behavior in the network. Virtualization is an entirely different approach, where the parameters of the network are constantly changing due to real network traffic. For instance, a virtual router in other methods would retain a controlled packet loss throughout the execution of the test-bed, however, with virtualization, packet loss would be affected by factors such as the emulated size of the virtual router's buffer, which may overflow during high traffic scenarios. Virtualization is often favored for the realistic behavior it provides, but it can also be used to setup specific scenarios.

2.4.2 Technologies

As mentioned in the previous section, numerous emulation systems exist. *Graphical Network Simulator 3* (GNS3) is a particular example of an emulation tool/technology that allows for the creation and execution of emulated network environments. These environments require extensive setup, to the extent that is seen in real network equipment [10]. Following the taxonomy mentioned in the previous section, GNS3 is a centralized system that implements its impairments at the kernel level and uses virtualization to replicate network devices. Since GNS3 uses virtualization, it requires the support of various virtualization technologies.

Virtualization technologies allow a single physical computer system to host multiple isolated systems, each of which run in their own virtual environments. Methods of virtualization include hypervisor-based and container-based.

Container-based methods run at the operating system level, that is, they require a host operating system to run. Each virtualized system created with this method is referred to as a container, and runs as a normal process on the host machine, thus not requiring their own kernels. Container-based methods are usually easier to setup and are often lightweight [6], which is particularly useful for emulated systems that do not scale well (see section 2.4.1). *Hypervisor-based* methods run at hardware level (not to be confused with the hardware level in the previous section), meaning that they do not necessarily need a host operating system. Each virtualized system created with this method contains a guest operating system and an associated kernel, which significantly increases the hard drive and memory space required to store and run these systems. Furthermore, hypervisor-based methods have been shown to have poorer performance in comparison to container-based methods [11, 41]. Despite their disadvantages, hypervisor-based methods have proved to be more versatile, as they can support multiple operating system environments (e.g. Windows and Linux) on the same system [6].

Many implementations of virtualization exist, each with their own unique purpose. Docker and dynamips are two examples of virtualization tools that illustrate, to some extent, the differences seen in various virtualzation implementations, and why such implementations need to use specific virtualization methods.

Docker is a virtualization technology used for creating virtual environments. It uses a container-based approach that is particularly useful for solving the issue of computational reproducibility in research, whereby the results of research cannot be recreated due to problems with code dependencies or documentation [5]. A hypervisor approach would not be ideal for docker, as it would be difficult to distribute the potentially large images that contain the guest operating system and kernel. *Dynamips* is a virtualization technology used for Cisco routers. It more closely represents a hypervisor-based approach to virtualzation, however, with minor deviations, as it does not allow the guest operating system to communicate with the host operating system, and it has to emulate much of the hardware within Cisco routers [25]. Container-based approaches would not be suitable for dynamips, as the kernel in Cisco routers is tightly linked to the rest of the router's functionality, and thus would not operate without it. Dynamips would be far less compatible with a container-based approach.

2.4.3 Alternatives

All networks that use virtual servers, virtual NICs and virtual switches or virtual routers are considered to be emulated networks [38]. Alternatives to this include simulation and live experimentation [24].

Simulation is method that typically uses a specially crafted model to test very specific and simplified versions of the greater problem. It is economical and fast, but does not run in real-time (often runs faster than real-time) and frequently requires that simulated problems be evaluated in live or emulated test-beds as well. Many tools, such as OPNET or the ns simulators, aid in simulation creation by providing a core simulation engine and a set of pre-designed model protocols [8, 39].

Live experimentation uses real hardware and software components for the underlying network environment. It is ideal for gathering results, however, not always practical, as it is inflexible and expensive. For example, live testing would not work in a scenario that needs to test a technology that is not operational yet, whereas other methods would be able to simulate or emulate the technology. Furthermore, not only is there a large cost associated with the equipment required, but there is also a large investment in terms of time to setup the equipment.

Network emulation isn't perfect either. As mentioned before, it suffers from scalability issues, since each network device requires a non-insignificant amount of memory and processing power. This issue is particularly relevant with centralized emulation systems, where computational resources are finite. Another issue with centralized emulation systems, is that the available bandwidth to the real world is limited by the NIC and connection available to the host system that runs the emulation software. The last notable issue is that in some emulation systems, network traffic at, or below layer 2 of the OSI model, cannot be perceived from the outside, which is especially problematic for debugging purposes. Despite these imperfections, emulation is still a viable option for creating realistic and controlled network environments, since it is cheap and easy to setup in the context of research labs.

2.5 Digital forensics

The field of digital forensics is one that is concerned with the development and use of tools or methods that aid in the investigation of computer related crimes. Network forensics is a similar concept, but focuses on the exchange of network communication [9]. There are various difficulties not only in the creation of these tools/methods, but also in general in the field of digital forensics. One of the more notable difficulties is that of standardization. The field contains few standards, and many are not complete or in use. The lack of standardization means that there is a lack of commonly followed procedures, which can be problematic, as improper procedures can lead to data being lost at a crime scene. Despite the lack of standardization, some procedures are followed. The following section will outline these procedures along with the few standards that exist. The various difficulties in the field will also be explained in detail.

2.5.1 Difficulties

The primary problem faced in digital forensics is the lack of standardization. Most of the field is practitioner driven, which leads to many tools only being designed to

solve a specific problem [26]. Communities supporting the development of digital forensics standards are few in number, and the majority of them have ceased to be, due to various issues, such as a lack of funding or interest. In addition, many of the existing tools have become obsolete, due to evolving operating systems and file formats. Encryption schemes have also become more sophisticated, making it easier to obfuscate data. These problems have been projected to only get worse if the deteriorating nature of the field continues [13].

In the circumstances where data had been obtained, and tools were available, two more problems would persist. The first is of processing power, in which certain analytical tasks may need to be forfeited due to the processing time required to analyze the potentially large volumes of data. There have been arguments to create tools with performance in mind, but these have largely gone unimplemented [13]. The second issue after data has been collected, and assumedly processed, is that of the legality regarding the information discovered. Legal issues are particularly sensitive when a company that operates in multiple countries is involved, or when the servers hosting the company's data is outsourced [18, 26].

The last and consequential issue of the problems faced in digital forensics is the lack of trained staff whom qualify to be digital forensics investigators [13]. Digital forensics requires that investigators have knowledge not only on multiple operating systems, file systems, and encryption schemes, but also on the legal issues surrounding them. Finding well trained and equipped staff is a problem, especially when few standards exist, and when the tools available vary in how they operate.

2.5.2 Procedures

Though not standardized, there still exists some known procedures from the field of forensics in general. The most notable, is that of disturbing or tampering with the crime scene evidence, as this could make analysis more difficult and may invalidate the evidence from a legal perspective. The ideal scenario would allow for an exact replica of a suspect computer system to be created. This is unfortunately a complex task, as the process of copying data from the suspect computer system may cause data to be accidentally written to it. For hard drive backups, external block writers that act as an intermediary between the suspect hard drive and the backup hard drive are often used to prevent these accidental writes to the suspect hard drive (see figure 2.5). Once a backup is created, it is often preferable to create a hash, so as to ensure the integrity of the data. It may also be preferable to create a file-by-file copy in addition to a sector-by-sector copy of a suspect hard drive, as it would reduce the complexity required for specific digital forensics tools, for instance, when reconstructing files that were distributed across many suspect hard drives in a Redundant Array of Inexpensive Disks (RAID) configuration [14].

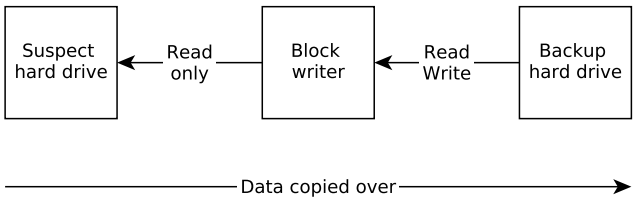

FIGURE 2.5: Block writer procedure

2.5.3 Standards and tools

The few communities that do exist in the field of digital forensics have attempted to create standards, often through frameworks and tools. Many are, however, underappreciated or are never used [13].

The most notable example of this is *PyFlag* which is a framework that attempts to merge disk forensics, memory forensics and network forensics in one package. It was designed to provide high level information, such as intent, identity and timelines, so as to simplify the forensics process for the investigator. PyFlag is particularly well suited to digital forensics problems that require scalability, where large files require processing [9]. Another example, created through the means of the *digital forensics working group*, is a taxonomy designed to define the research areas for digital forensic. The taxonomy contained: evidence modeling, network forensics, data volume, live acquisition, media types and control systems. It also approached various other issues in the field of digital forensics, such as education and legal issues [26]. Other tools that were not widely adopted include the *bulk_extractor*, which can be used to extract all email addresses and Uniform Resource Locators (URLs) from a computer's hard drive. This tool was created for information archiving purposes, with the intent of finding what services were used on a computer, but is usable for digital forensics purposes as well [14]. Other tools have focused on the digital forensics of social networks. One tool in particular creates what is referred to as 'social snapshots' of social network accounts. These snapshots contain information about the account's friends, exchanged messages and posted pictures [18].

2.6 SOAR

Enterprise networks expand both in size and complexity as new technological innovations develop. This expansion, while potentially beneficial for enterprises, creates a substantial strain on existing network defenses, which are already plagued with various issues. There exists, however, frameworks and standards, such as Security Orchestration, Automation and Response (SOAR), that aim to alleviate this strain. Though still a work in progress, frameworks like SOAR give an insight into the future of network defense, and steps are being taken to further develop them [12].

2.6.1 Issues approached

Network attacks are ever evolving, and many of the defenses created to thwart them never achieve complete success. This is due to the fact that network security in many enterprises require increasing amounts of data processing as the enterprise grows. This large amount of data is often processed by network administrators, or human experts, who then need to devise potential responses to attacks as well. Unfortunately, it is not feasible for human experts in large network environments to both analyze and respond to attacks in a short time frame. The Mean Time to Detect (MTTD) is often too large and so too is the Mean Time to Respond (MTTR), which reflects poorly in comparison to the time taken for an attacker to compromise a system. The issues of late detection and response are only furthered by the diminishing numbers of qualified staff in the field. The result of these issues is that enterprises face increasing risk with their growth.

2.6.2 Solution direction

The solution to the problems mentioned in the previous section calls for automation, as with automated or even semi-automated systems, reliance on the already limited number of staff will decrease, and will allow them, when necessary, to focus on responding to attacks.

SOAR, or also know as Security Automation and Orchestration (SAO), is a solution stack that maintains certain standards that are used in the processes of detection and response. The standards are designed to enable, to some degree, automation of the steps in each of the processes, making the network more independent of the administrators that maintain it. To achieve this, SOAR products are often integrated with systems that monitor network traffic, intelligence platforms that contain information on attacks, and control systems that mange responses to attacks.

The overall aim of SOAR is to reduce the MTTD and MTTR, which if successful, will enable faster responses to attacks, and will consequently minimize the damages inflicted on an enterprise network.

2.7 Summary

This chapter has discussed related works and topics, and has presented the technical details required for the experiment chapter (4). A taxonomy of intrusion response (2.1) as well as the different approaches to learning, supervised (2.2) and reinforcement (2.3), were discussed. These sections show that the structure of IRSs and of learning methods vary greatly in implementation. The learning methods in particular follow completely different concepts and goals while learning. Furthermore, the information provided on network emulation (2.4) explains the differences between simulated, emulated and live networks, and also provides a basic structure of emulated networks and mentions their associated technologies, such as GNS3. The final two sections on digital forensics (2.5) and SOAR (2.6) mention the associated difficulties that each face, with digital forensics expressing the importance of procedure, whereby data should not be tampered with when collected, and SOAR expressing the need for automation in the processes surrounding detection and response.

Chapter 3

Design and Implementation

The previous chapter (2) discussed a variety of related works. It can be seen that the deactivation issue, which this work aims to contribute to, belongs to a much larger taxonomy in IRSs, and is often not implemented in practice. Differences between supervised and reinforcement learning were also provided. Though both forms of learning are useful, it can be seen that reinforcement learning is preferable due to its ability to be applied to interactive problems, where the ideal solution can change. Other relevant information provided was about SOAR, which further justified the need for automated IRSs, due to the issues of growth in large enterprises. Digital forensics was also mentioned, and provided information on the procedures of responding to an attack. It can be seen that the evidence of an attack is susceptible to corruption if proper procedure isn't taken. Responses from an IRS therefore need to be careful not to tamper with the evidence, though this can be difficult to ensure, especially with an IRS that makes use of machine learning where the outputs are uncertain. This particular issue will be addressed in the next chapter (4), however, more relevant to this chapter was the section on network emulation (2.4), which discussed differences between emulation, simulation and live experimentation.

This chapter will present an overview of the test-bed, providing context for the technologies and network infrastructure that it uses. Both topics will be summarized in the overview, and discussed in detail in their own sections. Furthermore, these topics will be contrasted to real networks for comparison purposes, as the emulated network, which is used for the test-bed, cannot replicate all the behavior of real networks, and therefore, requires some compromises.

It is worthwhile noting that each section of this chapter is designed to increase in detail from the preceding section, with the overview being the least detailed, and the technologies section being the most detailed.

3.1 Overview

In this section, an overview of the test-bed is provided by summarizing its structure. The overview, which can be seen in figure 3.1, lists the three layers of the test-bed's structure: hardware, software and the emulated network. This section will discuss how these layers relate to each other, while the subsequent sections will discuss each of these layers in more detail.

The *hardware* layer represents the physical computer that the test-bed ran on. This layer could be populated with multiple physical computers, which run together in a distributed system (see section 2.4.1), however, the emulation software used, GNS3, did not allow for a distributed setup, which is why only a single computer is used.

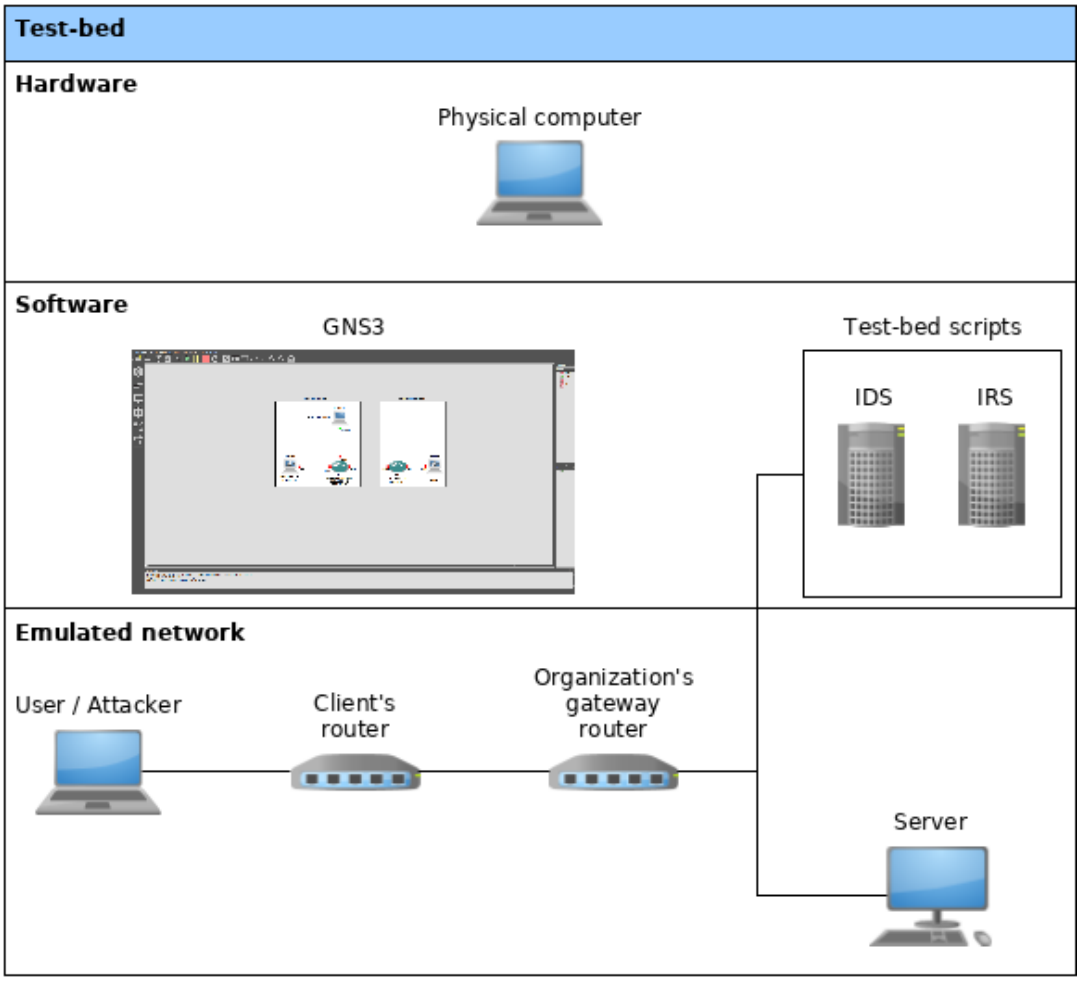

FIGURE 3.1: Overview of test-bed structure

The limitations of having a single computer mean that less resources are available for the emulated network, however, it does mean that the setup of the testing environment is easier. This layer allows for the execution of the software layer, using hardware resources which will be discussed in the hardware and software section (3.3.2).

The *software* layer allows for the execution of GNS3 and the test-bed scripts. GNS3 is the software used to create and run the emulated network, seen in the next layer. Other software such as docker (see section 2.4.2) is also used at the software layer, but is not depicted as it runs as part of GNS3. The details of all software used in the test-bed are further discussed in the test-bed technologies section (3.3). The second component of this layer, the test-bed scripts, were created for this work, and contain the functionality required by the IDS and IRS, which is to create alerts and learn from attacks. These scripts have been implemented at the software layer, as it allows changes to be applied immediately without interrupting the network. The alternative would be to create another virtual computer in the emulated network, that would contain the test-bed scripts. The IDS and IRS code would then have to be transfered to the virtual computer with every change, which happened frequently when learning parameters were adjusted. In this instance, the versatility of emulated networks was taken advantage of, as there is no difference between running the scripts at the software layer or at the emulated network layer.

The last layer represents the components of the *emulated network*. This network was designed to replicate the behavior of a real network, which requires compromises to be made. The emulated network is compared to a real network in more detail in the network comparison section (3.2), however, it is worth mentioning how the components communicate in this section. The most noticeable component is the user / attacker node, which, from the perspective of the network, is supposed to communicate with the server as a normal user. It is, however, labeled as an attacker as well, as it is assumed for testing purposes, that attacks will only occur from users who are outside the network. Furthermore, no actual attacks are performed, and instead, are simulated by having the IDS generate alerts for the IRS. In practice, the user / attacker node only pings the server to see if responses are in effect. If they are, the ping requests will fail to be received, and will either be dropped or rejected, depending on the response enacted. The server node itself does not host any services, and is instead used to respond to the client's ping requests. It is also setup to securely communicate with the IRS, and contains the required software dependencies to execute software that may be part of a response (e.g. virus scanning software). The remaining nodes are used purely for network infrastructure purposes, with the exception of the IDS and IRS nodes, which will be discussed in the experiment chapter (4).

3.2 Network comparison

The previous section covered the emulated network, which can be seen in figure 3.1. This section expands on the emulated network, by explaining each component in more detail, and by comparing it to the real network that is was designed to emulate. A hierarchy has been created for the real network by using the layers: clients, Internet, gateway and organization. For comparison, the emulated network mentioned in the previous section has been added to the right side of figure 3.2.

The *clients* layer considers normal users of the organization's network who connect to it from outside the network itself. These clients will be legitimate users of the network, containing their own login credentials, and using these credentials to access services from the server in the organization layer. It is however also assumed that attackers of the network will reside here, and will attempt to take advantage of vulnerabilities of the network. Multiple users or attackers can exist in this layer in a real network. Unfortunately, in an emulated network, it is computationally intensive to create multiple users and attackers, therefore, only one user is used to represent the client and attacker. However, as mentioned previously, for testing purposes, and due to scope limitations, no attackers actually exist in the clients layer, and instead, attacks are simulated by the IDS. In addition, the users will only issue ping commands to the server, to ensure that network responses are working, which can be verified if the ping requests are dropped or rejected.

The next layer, *Internet*, allows legitimate users and attackers alike to connect with the organization. Since the organization's router is directly connected to the Internet itself, and does not contain any firewall restrictions by default, there is no discrimination as to who can communicate to the organization's network. In real networks, it is assumed that multiple Internet Service Providers (ISPs) will give users or attackers the means to access the Internet, and subsequently the organization's network. It is consequently assumed that multiple IP addresses will communicate with the organization's network, which requires that the IDS and IRS filter those IP addresses that are attackers from those that are users. The emulated network unfortunately does

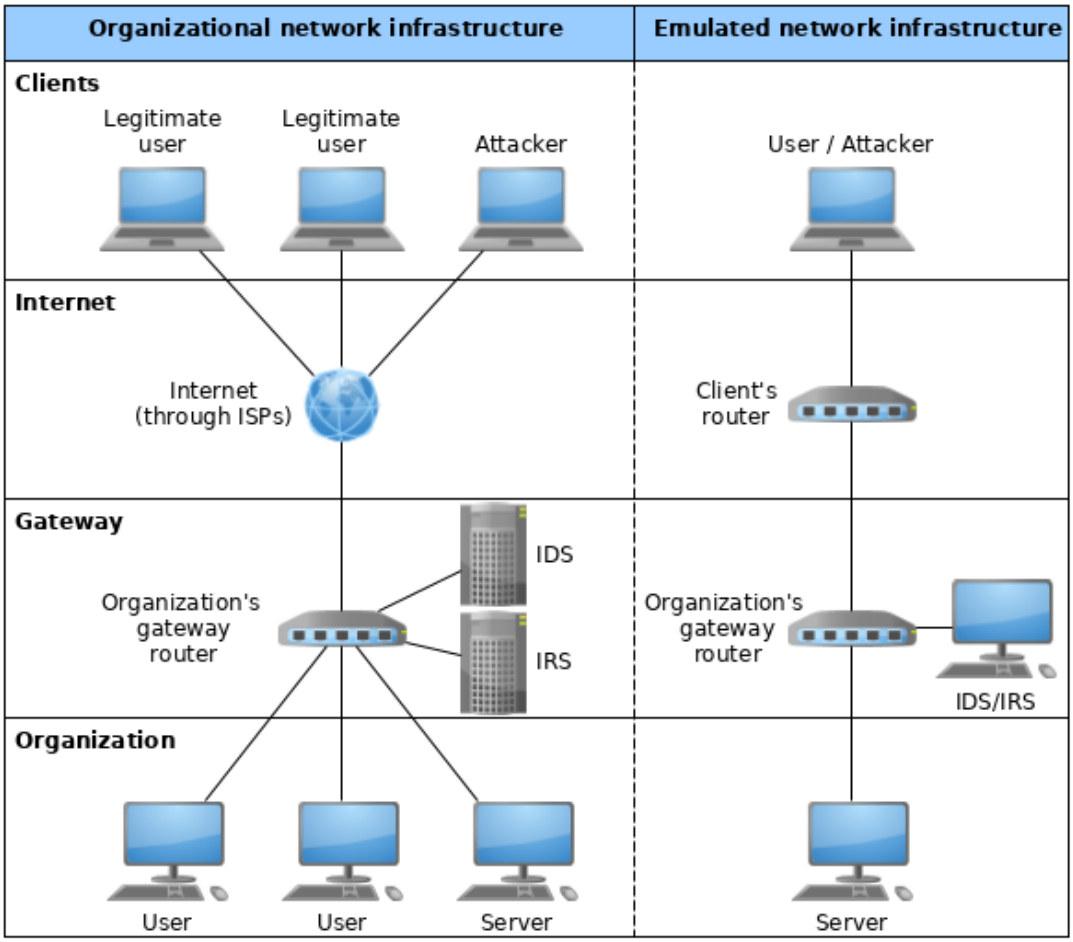

FIGURE 3.2: Comparison of real and emulated network structure

not simulate this filtering process, as there is only one user in the clients layer and all attacks are simulated. The Internet layer of the emulated network does however use a random IP address to represent the user, even though no ISPs are involved. This is done by manually assigning the random IP address to the client's router, which also contains a manually applied route mapping to the organization's router. Though not a perfect emulation of the Internet, this does allow the client to use a realistic IP address, and if used with multiple users, would allow for a more realistic networking, or Internet environment.

The *gateway* layer is concerned with access to the organization's network. This layer is directly connected to the Internet, the organization's users and servers, and the IDS and IRS systems that mitigate attacks. It is assumed that all network traffic will have to pass through the organization's gateway router to access the organization's network. This means that all network traffic passing through this point can be analyzed by the IDS to identify network attacks. Network traffic indicative of network attacks can then be mitigated by sending alerts to the IRS, which will enact responses on either the router or the server, or both. Since the gateway is only implemented by a single router, it makes it possible for an attacker to view the internals of the network. Additionally, attacks can be performed before being analyzed, as the network traffic does not have to pass through the IDS. These issues can be fixed through the use of a Demilitarized Zone (DMZ), whereby, two routers are used with the IDS, and

optionally the IRS, in between. Unfortunately, a DMZ was not used in the emulated network, as its use would increase modeling complexity and computational resource usage. For these reasons, only a single router was used. Additionally, for modeling simplicity, this router made use of two IP addresses, with one for the connection between itself and the client's router, and the other for the connection between itself and the IDS and IRS, which, in the emulated network, is the host computer as is seen in the software layer in the previous section.

The *organization's* network consists of users, who use network resources and services locally, or connect to services from inside the network itself. Like the users in the clients layer, these are also legitimate users of the network with their own login credentials, however, unlike the clients layer, it is assumed that there are no attackers among the users. This assumption is made, as defending against network attacks from within a network itself is a different problem to defending against external network attacks. As such, it is considered out of scope for the purposes of this work. The other component of the organization layer is the server, which is intended to provide services to both internal and external users of the network. The emulated network contains some differences to real networks though. In particular, it considers the server as the only device in the organization layer. This is done, as the addition of users at this layer would serve no purpose if none among them are attackers. Additionally, this aids in reducing the computational requirements of the emulated network. Another notable difference regards the server's behavior, whereby, it will not host any services and will only respond to ping requests, as mentioned in the previous section. This is done, as attacks are simulated, and therefore, no benefit is gained by hosting network services, since they cannot be attacked. Removing the requirement of hosting network services further reduces modeling complexity and the computational requirements of the emulated network.

3.3 Technologies

The aim of this section is to provide context for the software and hardware dependencies of the test-bed. It is divided into two sub-sections, which when combined, summarizes the hardware and software used in the test-bed, and covers the first two layers of the overview in figure 3.1. The first sub-section (3.3.1) categorizes the various software components used in the test-bed and explains how they contribute to it. Figure 3.3 is used to summarize the information discussed in this sub-section. The following sub-section (3.3.2) briefly discusses the hardware components and provides a full list of all software used, along with their version numbers.

3.3.1 Software applicability

Networks require a communications *medium* by which network data can be transfered. In the instance of real networks, this can be implemented through various physical or wireless mediums (e.g. copper cable). However, in the instance of an emulated network with virtual machines, this medium must be created through other means (see section 2.4.1). GNS3 is the software used to simulate the communications medium in the test-bed. This software does contain the ability to add network impairments such as packet loss, jitter and delay, however, these were ignored as the aims of this work were to test if the IRS could learn from attacks, therefore, the test-bed assumes perfect networking conditions.

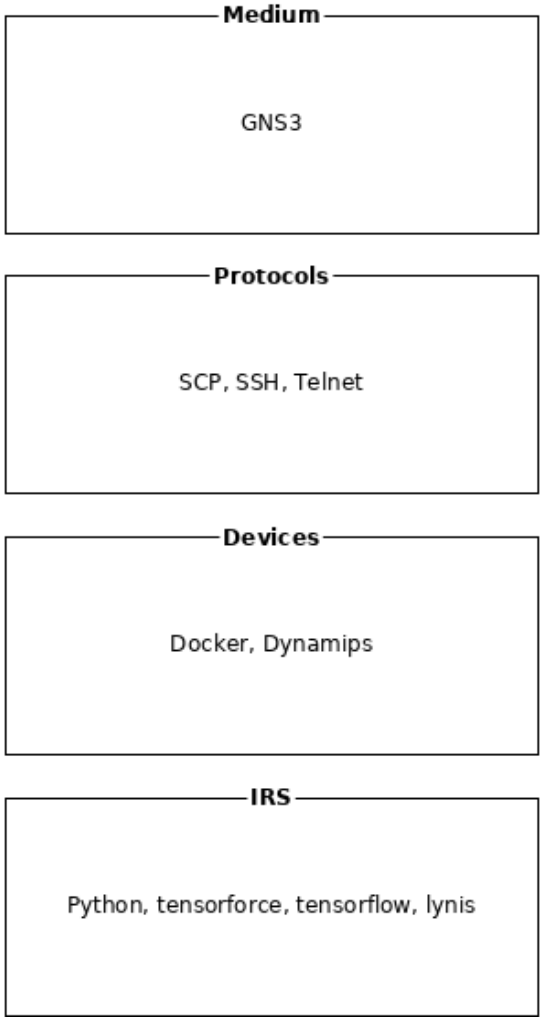

FIGURE 3.3: Technology infrastructure

Another prerequisite of networks involves the *protocols* they use. Those that were manually implemented in the test-bed consisted of: Secure Shell (SSH), Secure Copy (SCP) and telnet. These protocols were used to enact responses from the IRS, to the router and server. In specific, SCP was used to transfer virus scanning software to the server, and SSH was used to execute it. Though out of scope, the virus scanning software recommended SSH and SCP be used, which required that security keys be created and that the public key of the IRS be shared with the server. Once manually exchanged, the IRS could securely transfer files and execute commands. The router did not require strict security requirements, which is why telnet was used to access it, without login requirements enabled, though in practice in a real network, they would be enabled.

The *Devices* layer mentioned in figure 3.3 is concerned with the execution of all hosts and routers on a network. Since the test-bed is emulated, so too are the network devices. Virtualization methods are required to run these devices, which is where docker and dynamips are used (see section 2.4.2). Docker, due to its low resource usage, is used to virtualize debian computers for both the server and the user on the emulated network. These computers contain no Graphical User Interface (GUI) and only essential software for execution. To enable communication between the server and the IRS, ufw and openssh-client had to be installed to change firewall settings

and accept SSH and SCP requests, respectively. Like the server host, the user host, though not performing much, still required specific software to perform its task. The software the user required was iputils-ping, which comes pre-installed with the debian container, therefore, no additional software had to be installed. The last set of devices to consider in the emulated network are the routers, which were not able to be virtualized through normal virtualization software. Instead, these devices required dyamips, a virtualization software that was able, with the addition of some software simulation, to emulate the specific hardware seen in Cisco routers. Router images are needed for the execution of dynamips, which were fortunately supplied by Cisco for this work.

The *IRS* can be considered a device of the network, however, it has been separated to distinguish the software it uses. This work intends to create an IRS that can learn, which is why tensorflow and tensorforce [23] are used. These two python modules allow for the easy creation of a reinforcement learning agent. Other software used by the IRS is python itself, for programming the behavior of the IRS, and lynis, which is the software used for virus scanning mentioned earlier. All this software combined forms the IRS, which will be discussed further in the next chapter (4).

3.3.2 Hardware and software

The physical system, as depicted in figure 3.1, executed the emulated environment using an Intel i7-8700 and 16GB of ram. No Graphical Processing Unit (GPU) acceleration was used for learning, since the learning problem was simple, as will be discovered in the next chapter (4).

The following list provides a summary of all software used in the test-bed and the debian docker containers, along with associated version numbers:

```
OS : arch linux (host computer)
Kernel : 5.0.7
GNS3 : 2.1.15
Docker : 18.09.5
Dynamips: 0.2.20
Openssh : 7.9p1 (used for SSH and SCP)
Inetutils : 1.9.4 (used for telnet)
Python : 3.7.3
Lynis : 2.7.3
Tensorforce: 0.4.3
Tensorflow: 1.12.0
```

```
OS : debian (server)
Openssh : 7.4p1 (used for SSH and SCP)
Ufw : 0.35 (for local firewall settings)
```

```
OS : debian (user)
Iputils-ping : 3:20161105 (used for Ping)
```

3.4 Summary

This chapter has presented an overview of the test-bed, with explanations as to how its various components work together. Detail has been provided for the networking infrastructure and software used. In addition, the hardware was also briefly discussed. It can be seen that various compromises have been made for the purposes of reducing modeling complexity and computational overhead. Though not a perfect representation of a real network, it can be seen that the emulated network still remains comparable.

Chapter 4

Experiment

The previous chapter (3) introduced the emulated network that is used to test the IRS. The structure of the emulated network and the purpose of each node was discussed. In addition, the various forms of software used throughout the test-bed were also discussed in terms of structure and purpose. GNS3 is the main software component used to construct and manage the emulated network, while docker and dynamips are used to virtualize the various hosts in that network. These hosts connected to test-bed scripts, which performed the various IDS/IRS related tasks of the network, though the methods by which they function were not discussed.

The following chapter will expand on the previous chapter by explaining the test-bed scripts and how they interact with the surrounding environment, such as with the emulated network. In addition, this chapter will also provide the means to answering the RQs presented in the introduction chapter (1). In particular, a solution will be provided for RQ1, which allows the IRS to respond according to the type of attack. The components leading to the solution of RQ2 and RQ3 will also be discussed in this chapter, however, these RQs will only be solved in the results chapter (5), after testing has been performed, and only discussed in the final chapter (6), when conclusions are made.

4.1 Test-bed components

The operations of the experiment are divided into five components: alert, select response, response, emulated network and timeout models. The alert component performs the IDS related tasks of the experiment, while the select response and response components perform the IRS related tasks. The remaining components are supplementary, and serve to aid the IRS in its tasks and also help demonstrate the experiment's applicability in real world applications. The five components can be seen in figure 4.1 with the specific processes that each perform, the output of which can be seen in Appendix A and Appendix B. The following sections will discuss each component in more detail.

4.1.1 Alert

The alert component serves as the IDS for the experiment, though, the IDS is only simulated, and does not scan and analyze network traffic to determine network intrusions. Instead, IDS alerts are either explicitly called or randomly generated, depending on what attack is being tested. For the purposes of this experiment, three attack types have been used: DoS, virus and firmware altered. Each attack intends to test a specific response (see section 4.1.3), as not all responses can be automated due to issues regarding digital forensics (see section 2.5.2).

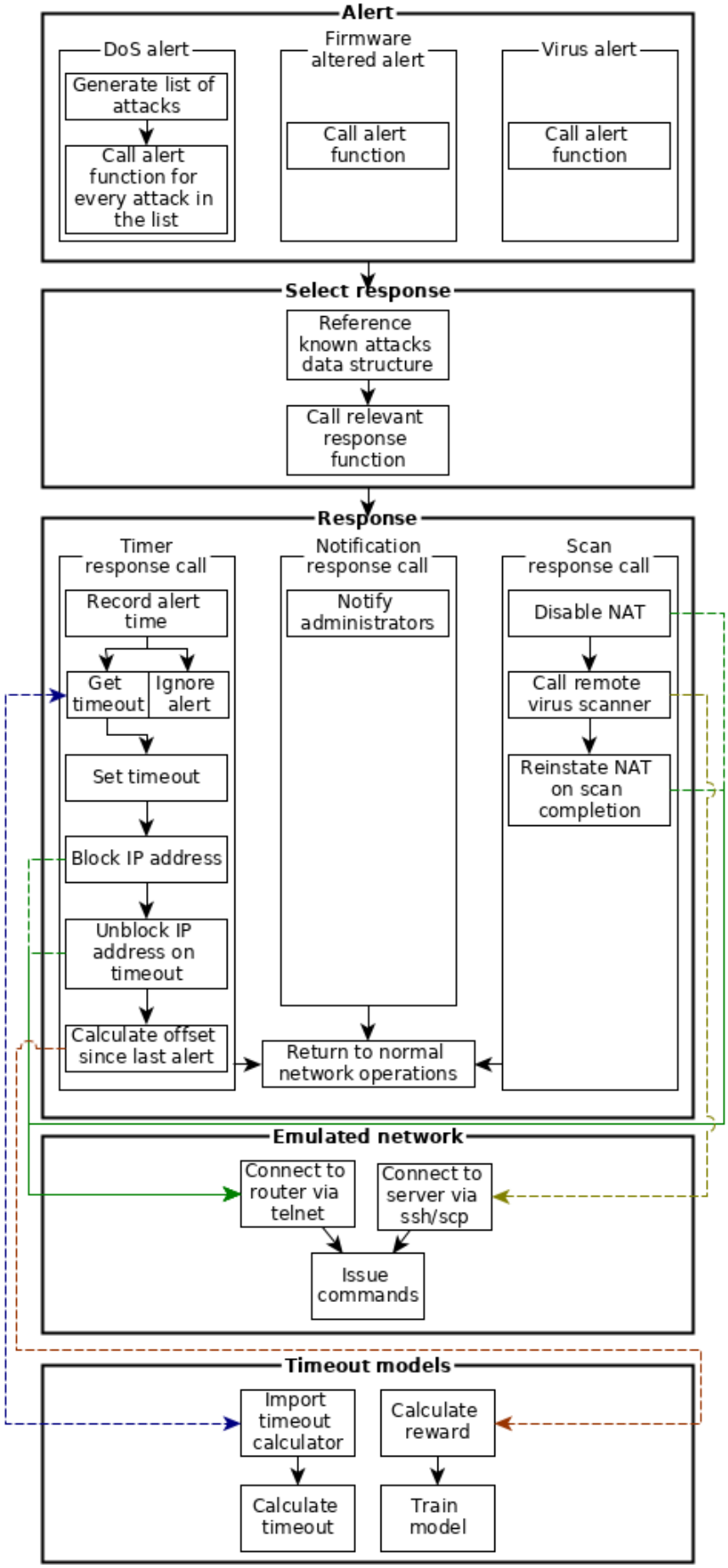

FIGURE 4.1: System procedure

The *DoS* alert aims to test automated responses that can be enhanced through reinforcement learning, since it is assumed that damage to the network with this attack will not be irreversible and will only affect its availability. The alerts themselves are structured to occur multiple times throughout an attack and each attack is structured to last a specific amount of time. There are multiple attacks, and the amount of time they last follows a sorted poisson distribution. The intention is to have the reinforcement learning algorithms learn the randomized poisson distribution, and thus be able to better predict how long attacks will last. Having the learning algorithms learn a specific distribution will also make it easier to discern whether they are learning or not in the results chapter (5). An example of both the randomly generated raw data forming a poisson distribution and the poisson distribution itself can be seen in figures 4.2 and 4.3 respectively. Following these figures, the first attack would last 38 seconds, with multiple alerts occurring during that time. Attacks would progressively become longer, and so too would their frequency, until attack times of 60 were reached, after which, the attack frequencies would decrease.

The *virus* alert aims to test limited automation, since it is assumed that a quick response to a virus will result in less damage to network integrity and confidentiality. It is also assumed, however, that learning responses, such as which program to run during a virus attack, are not preferable, as these responses could allow the network to undergo irreversible damage. Unlike DoS alerts, which are randomly generated, the virus alerts will only occur when explicitly called. The intention is to allow for alternate responses when automated ones that adapt through reinforcement learning are not applicable.

The *firmware altered* alert aims to show the applicability of no automation, since it is assumed that a periodic scan that discovers altered firmware on a device, will require further investigation. For scenarios requiring digital forensics, it is preferable that no automation takes place, so as to preserve the crime scene (see section 2.5.2). The intention is to allow for alternate responses when any form of automation is not applicable.

```
[38 39 39 41 42 42 42 42 43 43 44 44 44 44 44 45 45 45 45 46 46 46 46 46
 47 47 47 48 48 48 48 48 48 49 49 49 49 49 49 49 49 49 49 49 49 50 50 50
 50 50 50 50 50 50 50 50 50 50 50 50 50 51 51 51 51 51 51 51 51 51 51 51
 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 52 53 53 53 53 53 53 53
 53 53 53 53 53 53 53 53 53 53 53 53 53 54 54 54 54 54 54 54 54 54 54 54
 54 54 54 54 54 54 54 54 54 54 55 55 55 55 55 55 55 55 55 55 55 55 55 55
 55 55 55 55 55 55 55 55 55 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56
 56 56 56 56 56 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57
 57 57 57 57 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 59 59 59 59 59
 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 59 60 60 60 60 60 60
 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 61
 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62
 62 62 62 62 62 62 62 62 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
 63 63 63 63 63 63 63 63 63 63 63 63 63 64 64 64 64 64 64 64 64 64 64
 64 64 64 64 64 64 64 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 66 66
 66 66 66 66 66 66 66 66 66 66 66 67 67 67 67 67 67 67 67 67 67 67 67
 67 67 67 67 67 67 67 67 67 67 67 67 67 68 68 68 68 68 68 68 68 68 68
 68 68 68 69 69 69 69 69 69 69 69 69 69 69 69 70 70 70 70 70 71 71 71 71
 71 71 71 71 72 72 72 72 72 72 72 72 73 73 73 73 74 74 74 74 74 74 75 75
 75 75 75 75 75 76 76 76 76 76 76 77 77 77 77 78 78 79 79 80]
```

FIGURE 4.2: Example of the 500 attack instance data used in testing

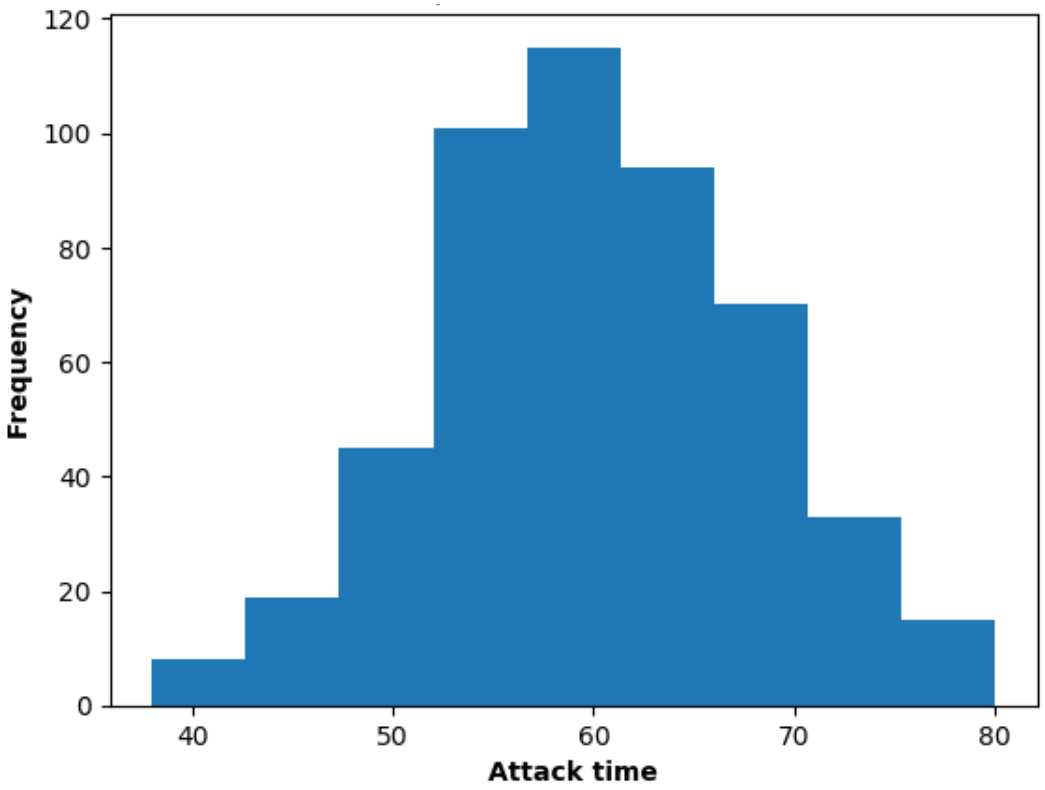

FIGURE 4.3: Example of the frequency of the 500 attack instance data

4.1.2 Select response

The select response component serves as the first component of the IRS in the experiment, and is responsible for receiving alerts and choosing the appropriate responses. Each alert received contains a parameter indicating the type of attack that was identified. Response selection is performed statically (see static mapping in section 2.1.3), according to the parameter in the alert. The DoS, firmware altered and virus alerts (see section 4.1.1) map to the timer, notification and scan responses (see section 4.1.3), respectively. The following data structure represents this information:

```
knownAttacks={
  'DoS' : 'timer_response'
  'firmware altered' : 'notification_response'
  'virus' : 'scan_response'
}
```

It is possible to map a response to multiple attack types, as shown below, however, for the purposes of this experiment, each response will remain mapped to one attack type, as shown above.

```
knownAttacks={
  'virus' : 'scan_response'
  'worm' : 'scan_response'
}
```

Once a response is selected, the appropriate response function is called.

4.1.3 Response

The response component serves as the second and main component of the IRS in the experiment, and is responsible for enacting the response chosen in the select response component (see section 4.1.2). Responses can be enacted in one of three approaches: timer, notification or scan. Note that some of these approaches interact with the emulated network that contains routers, a client and a server (see section 4.1.4).

The *timer* response prevents communication between a client and a server for a specific amount of time. To do so, the response system first notes the alert time, which is used for training purposes for the learning algorithm at a later stage. The response system then either ignores the alert, or generates what is called the timeout time, depending on whether or not the timer response is already active. The timeout time is generated using one of the learning algorithms (see section 4.1.5). Once done, the gateway router which connects the client to the server (router R1) will block the IP address of the client (note that this is done statically for testing purposes), thus preventing further communication. A process, called the timeout process, is then immediately started, and will activate when the timeout occurs (or when the system clock reaches the current time + the timeout time). The timeout process is multi-threaded, and therefore allows newer alerts to be received, however, only the arrival times of these alerts will be recorded, and the alerts themselves will be ignored, as a response is already active. Once the timeout occurs, the client's IP address is unblocked, and the accuracy of the timeout time is measured by subtracting the time of the last alert, from the time that the IP address was unblocked (or when the timeout occurred). This value, which is denoted as the offset in figure 4.1, is then sent to the reinforcement learning algorithm for training.

The *notification* response simply outputs a message to the console, or network administrator, about the alert. It is possible to automate some aspects of this response, such as permanently blocking communications between the client and the server, until a network administrator can unblock it, however, it is assumed that such automation could be harmful when digital forensics is required (see section 2.5.2). This response serves to complete the IRS by providing it with the option to not interact with the network, which can be beneficial in preventing undue harm to the network through its own responses.

The *scan* response automatically isolates the server in order to run a remote anti-virus tool. The response system does this by disabling the Network Address Translation (NAT) settings on the gateway router (router R1), that allow the server to communicate with external IP addresses. The connection between the IRS and the server, however, remains unchanged, thus allowing the IRS to communicate with the server. Once the server is isolated from external communication, the IRS copies an anti-virus tool to the server, and remotely commands it to perform a scan. For the purposes of this experiment, the anti-virus tool used is lynis. The IRS is able to wait until the tool is finished, after which, it will reinstate the NAT settings on the router. The commands to run the scan for the experiment, with 192.168.122.248 being the server's local IP address, are seen below. Additionally seen below are the commands used to shutdown the NAT, and to block the client's IP address (mentioned in the timer response):

```
scp -q ./files/lynis-remote.tar.gz debian
  @192.168.122.248:~/tmp-lynis-remote.tgz

ssh debian@192.168.122.248 'mkdir -p
  ~/tmp-lynis && cd ~/tmp-lynis &&
  tar xzf ../tmp-lynis-remote.tgz &&
  rm ../tmp-lynis-remote.tgz && cd lynis
  && ./lynis audit system'

ssh debian@192.168.122.248 'rm -rf
  ~/tmp-lynis'

scp -q debian@192.168.122.248:
  /tmp/lynis.log ./files/debian@
  192.168.122.248-lynis.log

scp -q debian@192.168.122.248:
  /tmp/lynis-report.dat ./files/debian@
  192.168.122.248-lynis-report.dat

ssh debian@192.168.122.248 'rm
  /tmp/lynis.log /tmp/lynis-report.dat'
```

```
conf t
no ip nat inside source static 192.168.1.2 150.1.1.2
yes #used to confirm removal
end
```

```
conf t
access-list 1 deny host 130.20.1.1
end
```

4.1.4 Emulated network

The experiment makes use of an emulated network to show its functionality. The emulated network contains five nodes, two of which are emulated Cisco c3745 layer 3 switches, and two of which are debian docker images (see section 2.4.2), that represent the client and server. The last node, called the IDS/IRS, is the host itself, and allows communication between the emulated network and the physical host. It is through this node that response commands (see section 4.1.3) are sent to, and executed on, the emulated network devices. A logical diagram showing the direction of communications, and the entire network within GNS3 showing all associated IP addresses, can be seen in figures 4.4 and 4.5, respectively. The relevant router configurations for the network can be seen as follows:

```
hostname R1

interface FastEthernet0/0
 ip address dhcp
 ip nat outside
 ip virtual-reassembly
 duplex auto
 speed auto

interface FastEthernet1/0
 no switchport
 ip address 150.1.1.1 255.255.255.0
 ip access-group 1 in
 ip nat outside
 ip virtual-reassembly

interface FastEthernet1/1
 no switchport
 ip address 192.168.1.1 255.255.255.0
 ip nat inside
 ip virtual-reassembly

ip route 130.20.1.0 255.255.255.0
  FastEthernet1/0

ip nat inside source static tcp
  192.168.1.2 22 interface FastEthernet0/0 22

ip nat inside source static
  192.168.1.2 150.1.1.2
```

```
hostname R2

ip dhcp pool POOL1
 network 192.168.1.0 255.255.255.0
 dns-server 192.168.1.1
 default-router 192.168.1.1

interface FastEthernet1/0
 no switchport
 ip address 130.20.1.1 255.255.255.0
 ip nat outside
 ip virtual-reassembly

interface FastEthernet1/1
 no switchport
 ip address 192.168.1.1 255.255.255.0
 ip nat inside
 ip virtual-reassembly
```

```
ip route 150.1.1.0 255.255.255.0
  FastEthernet1/0

ip nat inside source list 10 interface
  FastEthernet1/0 overload

access-list 10 permit 192.168.1.0 0.0.0.255
```

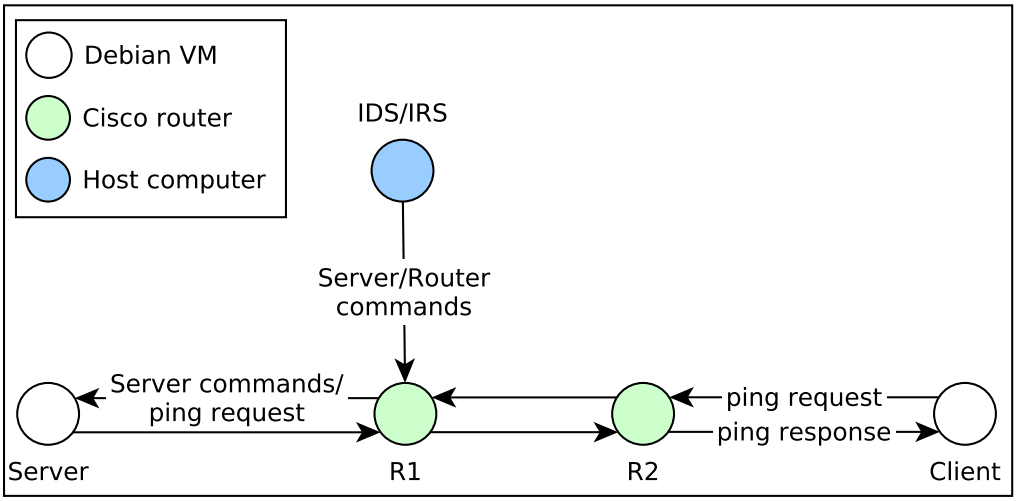

FIGURE 4.4: Logical network diagram with direction of communications

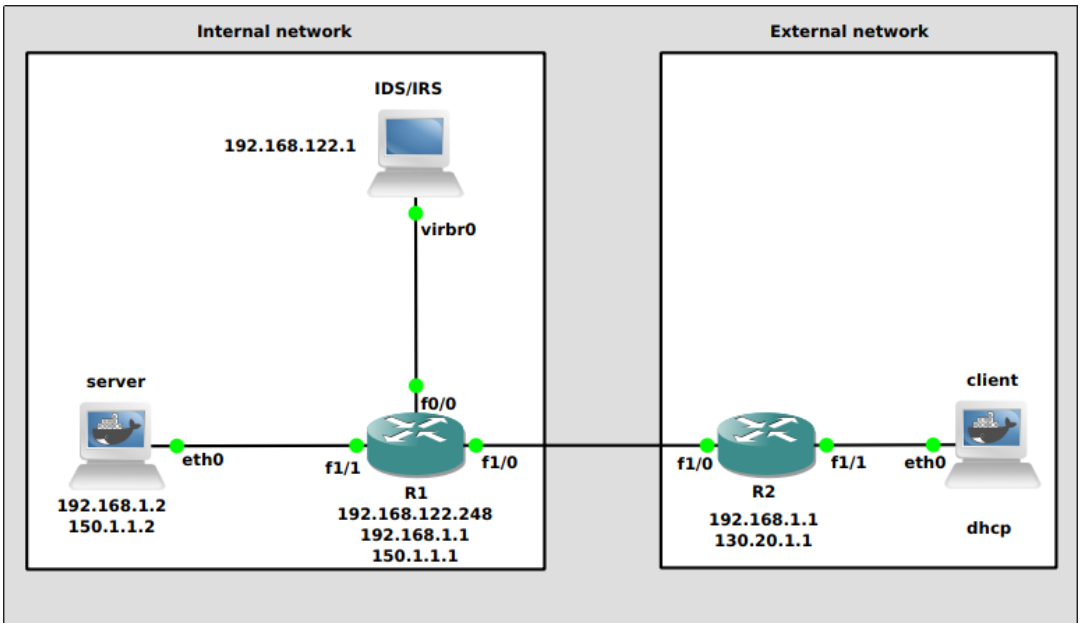

FIGURE 4.5: Emulated network structure

4.1.5 Timeout models

This component makes use of reinforcement learning algorithms to generate an estimate between 0 and 120 on how long an attack will last in seconds. Once generated, the estimate is then used in the timer response (see section 4.1.3). Reinforcement learning, as opposed to supervised learning, is specifically used since it is able to make estimates prior to having learned anything. The learning algorithms were implemented using a python library called tensorforce [23]. Three timeout models have been incorporated in the experiment, though only two use reinforcement learning, as one merely performs random guessing for comparison.

The environment the reinforcement learning agent interacts with, is similar to the K-armed bandit problem, whereby, only one state exists, and the learning agent can choose from multiple actions, with each giving different rewards. Actions can be chosen a number of times, and the goal of the reinforcement learning agent is to figure out which actions obtain the greatest reward (see section 2.3.2). An additional problem, however, is the non-stationary nature of the environment, whereby, performing the same actions repeatedly will not guarantee the same reward each time.

The reinforcement learning algorithms used were PPO and TRPO (see section 2.3.5). Default parameters were used for both algorithm in all cases, with the exception of the learning rate for TRPO, and the actions exploration parameters for both algorithms. Though they are the same, the parameters for both algorithms can be seen below, with initial_epsilon indicating the chance of exploration, and timesteps indicate the number of actions that are taken until an attempt at exploration is made (see section 2.3.4):

```
"type": "ppo_agent"
"learning_rate": 1e-3
"discount": 0.99
"actions_exploration": {
  "initial_epsilon": 0.5
  "timesteps": 100
}
```

```
"type": "trpo_agent"
"learning_rate": 1e-3
"discount": 0.99
"actions_exploration": {
  "inital_epsilon": 0.5
  "timesteps": 100
}
```

Once a guess is made, the learning algorithm has to learn how accurate it was. Higher rewards are associated with better performance. When the offset (mentioned in section 4.1.3) is sent to this component for training, it is converted into a reward, with higher values being fed into the learning algorithm, the closer the offset is to 0. The calculation for rewards is seen as follows with timeoutInaccuracy being the offset time:

```
reward = 5 / timeoutInaccuraccy
```

It should be noted that a value of 0.1 is added to timeoutInaccuracy if it has a value of 0 (if there was no offset). This is done to avoid a division by 0 and to limit the maximum reward. In addition to this, only the absolute value of timeoutInaccuracy is used, and therefore, no negative rewards are given to the learning algorithms.

4.2 Summary

This chapter has listed the various components of the IRS and has shown how they interact with external components, such as the IDS, the emulated network and the learning algorithms. The IRS is capable of solving RQ1, by using a static data structure to select which response to enact given a specific alert. The procedure of all implemented responses have been listed, and excerpts of commands that the IRS uses has been provided. In addition to these excerpts are configuration settings for the routers in the emulated network, the parameter settings for the learning algorithms and a line of code indicating how the rewards for the learning algorithms are calculated. The knowledge provided by this chapter and the previous chapter (3) cover the entirety of the test-bed and the IRS.

Chapter 5

Results and Analysis

The previous chapter (4) designed an IRS through the use of various components. While some components were part of the IRS itself, others were external and aided the IRS in its functioning. One of the internal components used was the select response component, which helped solve RQ1, by assigning static responses when given specific alerts by an external component, namely, the IDS. RQ2 and RQ3, which asked if the IRS could solve the deactivation issue through learning from its own responses, still remains unsolved, as testing is required to determine their success.

The following chapter will provide the results from tests conducted on the timer response (see section 4.1.3), seen in the IRS that was constructed in the previous chapter. These results will present the information necessary to determine whether RQ2 and RQ3 were successful or not, however, this will only be concluded in the following chapter (6).

Tests were conducted and summarized according to the number of attacks that occurred in the emulated network. The intervals for the number of attacks were 100, 500, 1000 and 2000. The length of the attacks, according to a poisson distribution, remained at 60 seconds for each interval, due to various limitations (see chapter 6). Two learning algorithms (see section 4.1.5) were used to test the performance of the IRS against the deactivation issue. In specific, the IRS algorithms would make estimates as to how long an attack would last, and consequently, how long a response policy would be enacted for. These estimates were recorded, grouped together, and graphed according to the frequency of their occurrences (see figures 5.1, 5.2, 5.3 and 5.4). In addition to the estimated times for each algorithm, these graphs also contain the frequency of the actual attack times that each attack lasted for, as well as the attack times for a random algorithm, which guessed the length of each attack randomly. The data on the actual and random attack times will be used for comparison purposes in the remaining sections, along with the various tables (see tables 5.1, 5.3, 5.5, 5.7 and 5.9), which summarize the average offset, or the average inaccuracy of the guesses made by the learning algorithms and the random algorithm.

5.1 100 Instances

The results for running the test-bed with 100 attack instances can be seen in figure 5.1 and in table 5.1. Both learning algorithms at 100 attack instances perform poorly, with an average offset, or inaccuracy estimate of 30 seconds for PPO and 29 seconds for TRPO. The algorithms perform similarly to random guessing, which can

been seen in the graph, as their estimates form a relatively uniform distribution in comparison to the poisson distribution of the actual attack times.

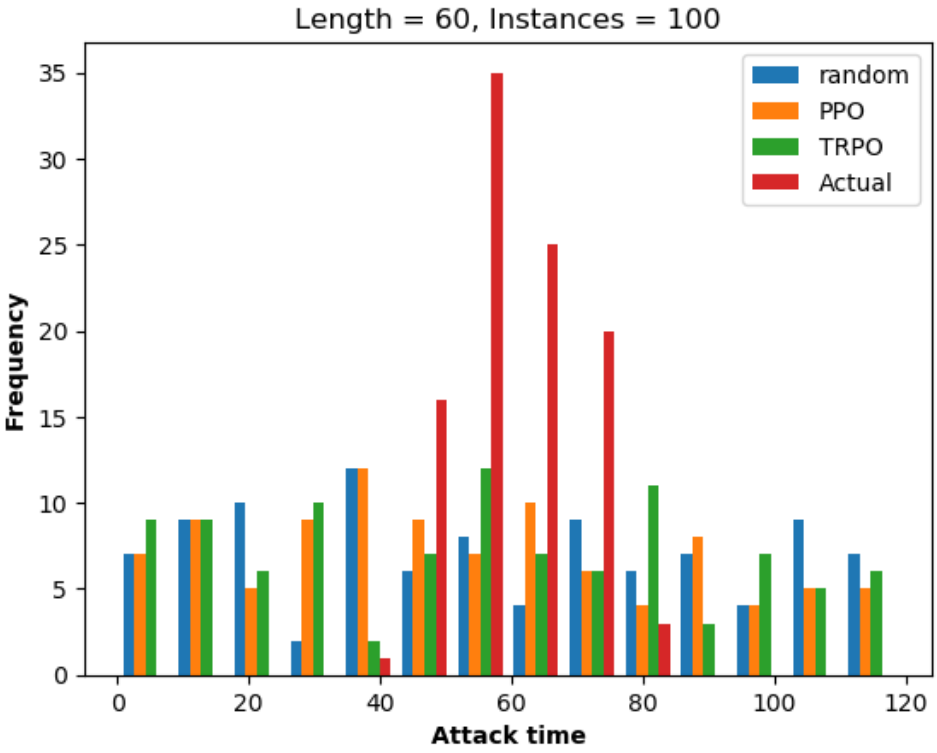

FIGURE 5.1: Actual vs estimated attack times for 100 instances

Algorithm/Instances	100
PPO	30
TRPO	29
Random	30

TABLE 5.1: Average offsets at 100 attack instances

Algorithm	Improvement	notes
PPO	0	Performance similar to random guessing
TRPO	0	Performance similar to random guessing

TABLE 5.2: Summary of 100 instances observations

5.2 500 Instances

At 500 instances, improvements start to become noticeable, as can be seen in figure 5.2 and in table 5.3. PPO improves by 6 seconds, with a new average offset, or inaccuracy estimate, of 24 seconds. TRPO, however, has only improved by 1 second, and remains similar to random guessing in terms of performance. The improvements of PPO can be seen in the graph, as a poisson distribution like shape starts to

form between roughly 50 and 80 seconds. Outside of this range, PPO, again, guesses relatively uniformly. TRPO, as indicated by the table, has not improved, and still maintains a relatively uniform distribution.

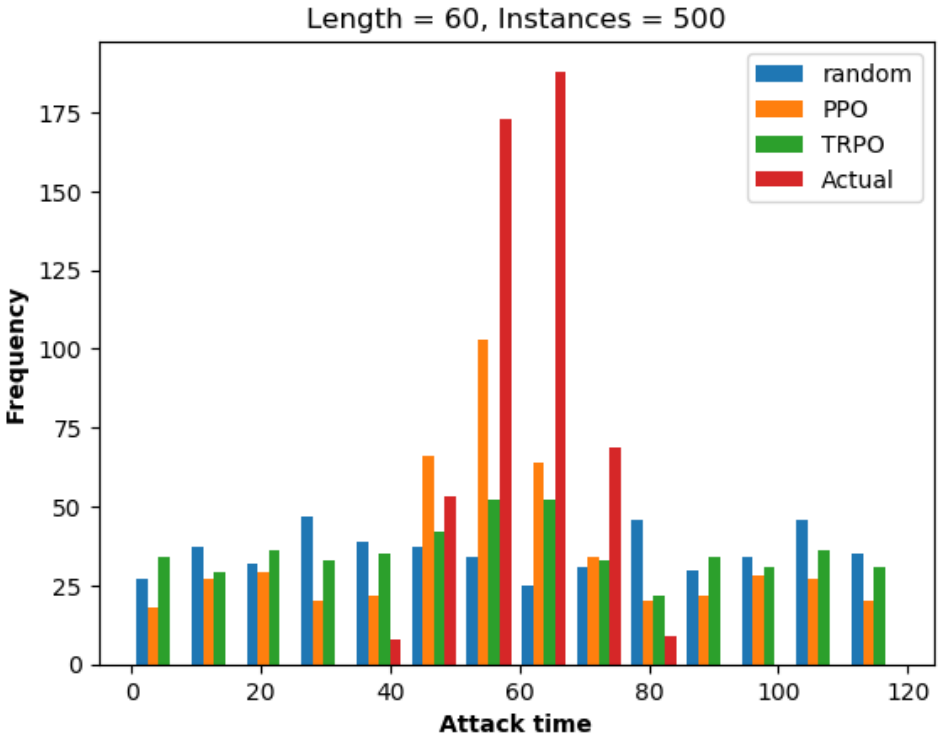

FIGURE 5.2: Actual vs estimated attack times for 500 instances

Algorithm/Instances	500
PPO	24
TRPO	28
Random	30

TABLE 5.3: Average offsets at 500 attack instances

Algorithm	Improvement	notes
PPO	6	Poisson distribution like shape forming
TRPO	1	Performance similar to random guessing

TABLE 5.4: Summary of 500 instances observations

5.3 1000 instances

At 1000 instances, improvements are made, as can be seen in figure 5.3 and in table 5.5. PPO improves drastically by 15 seconds from its previous best, with a new average offset of 9 seconds. TRPO, however, only improves by 1 second again, and though still similar to random guessing in terms of performance, it can be seen that

the algorithm is learning. This is particularly evident in the graph around attack times of 60 seconds, where TRPO makes most of its guesses. PPO's performance is also at its best at attack times of 60 seconds, and proves to obtain near identical results to the actual. In addition to this, PPO has also made far fewer guesses at attack times where the actual attack frequency is 0. Unfortunately, performance has become poor at attack times around 40 and 70 seconds. An explanation for the discrepancy in performance could result from the difference in training examples seen at the different attack times. For instance, both algorithms have had roughly 350 training instances for attack times around 60 seconds, whereas, they've had less than 10 training instances for attack times around 40 seconds. Another explanation could be algorithm behavior, as PPO displays qualities of being too greedy, since it over estimates the number of attacks at 40 and 60 seconds. This greedy behavior also prevents the algorithms from adjusting to different attack times, which can be seen in the low number of guesses in the times that follow (50 and 70 seconds).

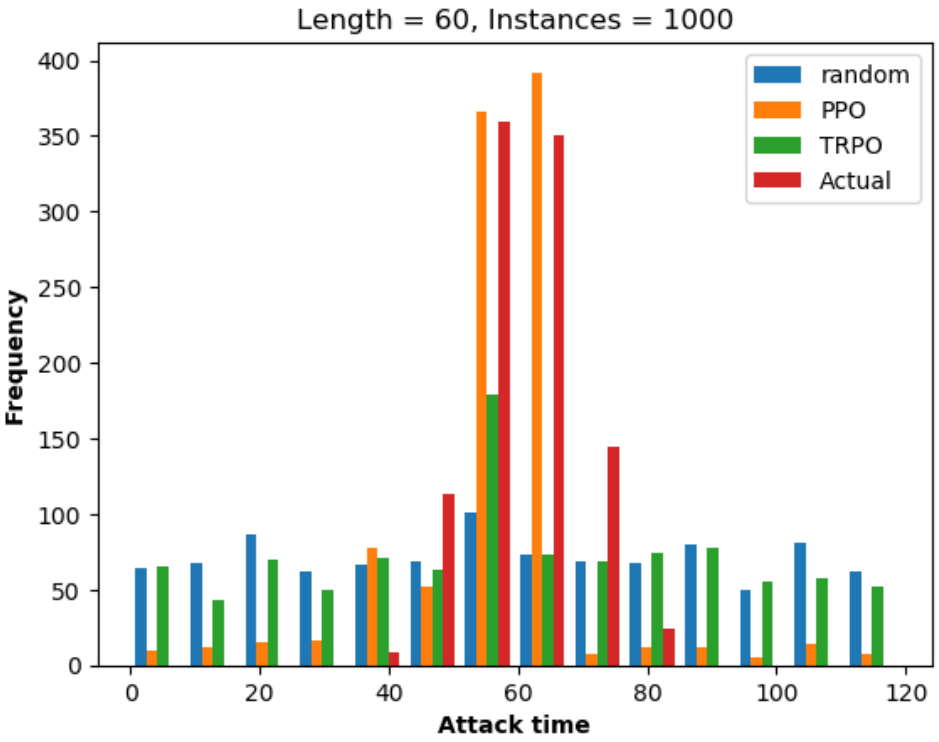

FIGURE 5.3: Actual vs estimated attack times for 1000 instances

Algorithm/Instances	1000
PPO	9
TRPO	27
Random	29

TABLE 5.5: Average offsets at 1000 attack instances

Algorithm	Improvement	notes
PPO	15 (21 total)	Accurate around 60 seconds, Greedy behavior
TRPO	1 (2 total)	Performance similar to random guessing

TABLE 5.6: Summary of 1000 instances observations

5.4 2000 Instances

At 2000 instances, both algorithms have learned to some extent, as can be seen in figure 5.4 and in table 5.7. PPO did not improve despite it training on an extra 1000 instances. It however did not perform any worse than before and still obtains an average offset of 9 seconds. TRPO, though still not performing as well as PPO, did improve by 6 seconds from its previous best, and is now distinctly different from random guessing. The graph shows this, as a trend has formed between the attack times of 50 and 60 seconds. PPO has also improved outside the attack range of 60 seconds and is still making very few guesses at times where the attack frequency of the actual is 0, however, it is also making fewer guesses at the attack range of 60 seconds, which could indicate that the algorithm has had troubles adjusting from what it has learned. This greedy behavior is similar to that seen with 1000 attack instances, and proof of it can be seen in the high number of guesses at the 50 seconds range.

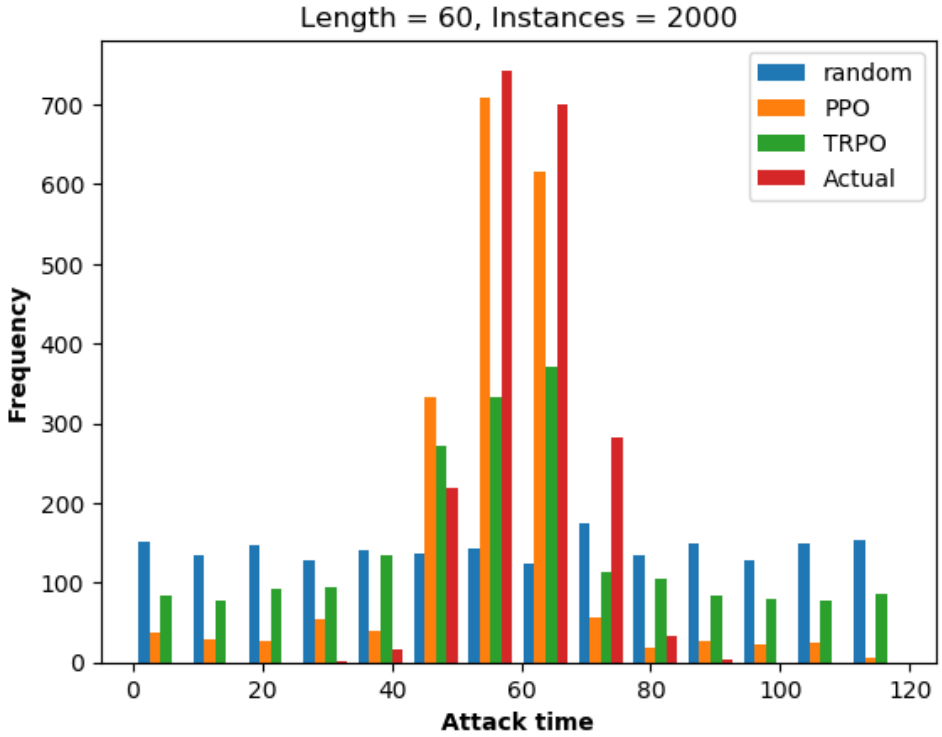

FIGURE 5.4: Actual vs estimated attack times for 2000 instances

Algorithm/Instances	2000
PPO	9
TRPO	21
Random	31

TABLE 5.7: Average offsets at 2000 attack instances

Algorithm	Improvement	notes
PPO	0 (21 total)	No improvement, Greedy behavior
TRPO	6 (8 total)	Poisson distribution like shape forming

TABLE 5.8: Summary of 2000 instances observations

5.5 Analysis and testing

Throughout testing, it can be seen that the performance between both algorithms improves most around 1000 instances. PPO achieves optimal performance before or upon reaching 1000 attack instances while TRPO obtains its largest improvement after 1000 instances. PPO is clearly the better suited algorithm to this problem, which is a simple problem that does not require complex states or delayed rewards (see section 2.3.2). TRPO, though performing worse, still showed signs of improvement, and if given enough training instances, could surpass PPO. Unfortunately, due to the problem, training on many attack instances isn't feasible, since it is time consuming and impractical, as many networks would suffer greatly without defense while waiting for an automated IRS to train. The problem is therefore best suited to algorithms that adapt quickly, but that also remain amenable to change. Though both algorithms were able to adapt, they showed signs of not being able to change, as they often got stuck at certain attack times. This frequently happened when many training instances were available. It may be possible to mitigate this issue through the use of better parameters for each algorithm, however, finding these parameters is a time consuming task (see section 2.4.3), especially for a problem that runs in real-time, and consequently, needs to be tested in real-time.

Algorithm/Instances	100	500	1000	2000
PPO	30	24	9	9
TRPO	29	28	27	21
Random	30	30	29	31

TABLE 5.9: Summary of average offsets

Algorithm	Improvement per instance	Notes per instance
PPO	0, 6, 15, 0 (21 total)	random, poisson, greedy, greedy
TRPO	0, 1, 1, 6 (8 total)	random, random, random, poisson

TABLE 5.10: Summary of instance observations

5.6 Summary

This chapter has presented the results obtained from testing. It can be seen that the algorithms do learn, as they progressively reduce their average guessing inaccuracy with increased training instances. Random guessing was used for comparison purposes and always remained at an average inaccuracy of around 30 seconds, while PPO and TRPO, at their best, obtained an inaccuracy of 9 seconds and 21 seconds respectively. PPO can be identified as the better algorithm, however, its performance stops improving after 1000 instances. In addition to not improving, the algorithm frequently uses a greedy approach which limits its ability to adapt, further limiting its performance. TRPO slowly improves, but does not reach PPO's performance. Both algorithms show behavior of performing well where the frequency, or number of training instances, is large, however, this is likely due to an exploitative approach being used by both of them.

Chapter 6

Conclusion

This work has presented various aims (see below) for an IRS, that was to be created to automate the process of responding to network attacks. This chapter will summarize how these aims were achieved, how well the test-bed performed and will provide concluding remarks as well as possible paths for future work.

The 3 research questions stated at the beginning of this thesis (see section 1.3) intended to design an IRS that is capable of:

RQ1 Responding appropriately according to the type of attack.

RQ2 Learning from the effectiveness of its own responses.

RQ3 Deactivating network policies at optimal times.

6.1 Assessment of research questions

The IRS created in the experiment chapter (4), was designed to be a combination of an automated response system and a notification system. This combination was necessary, as it was discovered that not all responses should be automated, as they may cause damage to the network and interfere with digital forensics related tasks (see section 2.5.2). Therefore, three categories of responses were created: those that respond automatically and learned (timer response), those that only respond automatically (scan response), and those that only notified the network administrator (notification response). These responses types are summarized in table 6.1 according to the taxonomy in section 2.1.

Category/ Feature	Level of automation	Response cost	Response time	Adjustment ability	Response selection	Applying location	Deactivation ability
Timer	Automated	None	Proactive	Adaptive	Static	Firewall	Supported
Scan	Automated	None	Reactive	Non-adaptive	Static	Firewall, Target	Supported
Notification	Manual	N/A	N/A	N/A	N/A	N/A	N/A

TABLE 6.1: Summary of response categories

6.1.1 RQ1 assessment

Switching between the above response types, depending on the type of attack, was the purpose of RQ1, and while not expressed in any results, it was possible to switch between response types due to the data structure seen in section 4.1.2. This form of response selection was the most basic from those available (see section 2.1.3), however, there was no need for a complex response selection process, as the test-bed only

used simulated attacks and a simple network. Any additional complexity to the response selection process could've interfered with testing, in the circumstance that inappropriate responses were selected. Though while basic, RQ1 was accomplished through static mapping, and helped to ensure that results from testing remained consistent.

6.1.2 RQ2 assessment

Like RQ1, much of RQ2 was expressed in the experiment chapter. This RQ aimed to build an IRS that could learn from its own responses. To achieve this, the timer response was created, and used two reinforcement learning algorithms, PPO and TRPO, to learn when to deactivate responses/policies. The algorithms were setup to be rewarded according to the accuracy of their previous guess. These rewards would influence the guesses that the algorithm would make in the future. As can be seen in the results chapter (5), both algorithms were able to learn, as their average inaccuracies were progressively reduced from 30 seconds as additional training instances were added. This can also be verified by the frequency of attack times seen in the results chapter, with the trained algorithms more closely representing the distribution of the actual attack time frequencies. These facts prove that the IRS was able to learn from its own responses, thus fulfilling the purpose of RQ2.

6.1.3 RQ3 assessment

Unlike the previous two RQs, RQ3 was fully expressed in the results chapter. This RQ investigated whether the IRS would be able to deactivate response policies at optimal times. It can be clearly seen from the success of RQ2, that both algorithms, PPO and TRPO, are capable of learning from their own responses, as both of them perform better than random guessing, with an average inaccuracy of 9 seconds for PPO, and 21 seconds for TRPO. Unfortunately, while RQ2 can be regarded as a success, and while the algorithms do learn, RQ3 remains questionable, as a 9 second inaccuracy is not necessarily optimal. In addition to this, both algorithms only perform well at attack times close to 60 seconds, and do not achieve optimal response times outside of this range. Furthermore, the results from PPO indicate that the algorithm's performance will not improve, meaning that a 9 second inaccuracy is the best it will achieve. TRPO may perform better if given more training instances, as it was still improving, even after 2000 instances, but unfortunately, training on this many attack instances may prove to be too slow for any practical IRS. It may be possible to improve TRPO's learning by adjusting its parameters (see section 4.1.5), however, there are disadvantages to this. For instance, it is possible to make TRPO more exploitative in order to reduce inaccuracy, however, the issue of exploration vs exploitation (see section 2.3.4) may appear, whereby, the algorithm will not explore enough, meaning that TRPO, like PPO, will get stuck in a sub-optimal solution. Increasing the learning rate may also contain negative effects, as the algorithm will place a high priority on correct guesses, however, when the optimal time changes, or when the guess is no longer correct, the algorithm will have difficulty changing or adapting to the new optimal time, due to the high priority it has placed on what it already knows. This issue is a result from the fact that the problem is non-stationary, meaning that the optimal solutions change as time progresses. Changing discount will also not improve performance. The reason for this is that discount only affects whether the learning algorithm is concerned with immediate rewards, or rewards

in the future. Unfortunately, the problem is very simple, since there is only one action and reward associated with a learning, or attack, instance. This means that it is impossible for the algorithm to consider future rewards, since there is only one reward to be considered. Changing discount from its default value of 0.99 to any other value, has been tested, and results in no change in TRPO's performance, which is why it was left unchanged.

6.2 Assessment of the test-bed

This work approached building a test-bed by using an emulated network. This approach was chosen to obtain accurate results, while reducing costs and setup time. The test-bed created was reliable enough to communicate with physical networks, and, despite some compromises (see section 3.2), was able to replicate the network necessary for testing. However, the test-bed still contained some limitations which affected testing. The following sections will discuss these limitations further.

6.2.1 Real-time results gathering

To allow for the emulated network to be stable and extensible enough so as to communicate with physical networks, it had to operate in real time. While this does allow for more realistic results to be obtained, it also limits the number of instances that can be tested. For example, the time it takes to run the test with 2000 attack instances is roughly 33 hours. Simulation methods may be better suited, especially for tests that aim for a number of attack instances larger than 2000. Using simulation may also aid in investigating whether TRPO does infact improve beyond an average inaccuracy of 21 seconds. While considering adjustments to the number of attack instances, it may also be worthwhile to consider adjusting the attack times, and have them differ from 60 seconds. This could not be done in this work, as attack times that were any longer, resulted in testing times greater than 33 hours, and attack times shorter than this, resulted in a higher chance of the poisson distribution containing attacks that lasted 0 seconds, which were not considered in this work.

6.2.2 IDS simulation

Though not mentioned in the experiment chapter (4), an IDS was successfully implemented in previous iterations of the test-bed, however, it was removed due to the large increase in scope and testing time that it required. An IDS would've beneficial though, as the learning performed in the IRS would then be affected by the inaccuracies of the IDS. Specifically, any false-positives, or times at which the IDS inaccurately guesses an attack when no attack was performed, would accidentally train the IRS on inaccurate information, leading it to become more inaccurate when a real attack occurs. The simulated IDS in this work was assumed to have perfect performance, and does not incorrectly train the IRS.

Another disadvantage of not having a functioning IDS is the inability to test network impairments, as corrupted network data would likely have a negative effect on the accuracy of the IDS. The increased inaccuracies of the IDS would only serve to further train the IRS on inaccurate information, making it less effective during actual attacks. The extent to which network impairments affect the learning of the IRS would be useful in understanding the feasibility of this work in real networks.

6.3 Future work

As seen in the assessment of research questions section (6.1), RQ1 and RQ2 were deemed successful, however, improvements could still be made to RQ3. In particular, improvements could be made to accuracy and training time. Unfortunately, due to the limitations of the complexity of the problem tested and specific limitations of the test-bed (see section 6.2.1), certain approaches that would've improved performance, couldn't be implemented. The following will discuss these approaches as areas of improvement (see sections 6.3.1 and 6.3.2) and will also highlight alternative paths of research for future works, that builds on this work (see sections 6.3.3 and 6.3.4).

6.3.1 Changing algorithms and problem complexity

TRPO and PPO are relatively new algorithms that perform well on simple problems, but it may be beneficial to consider making the problem more complex, by adding state information. In doing so, additional algorithms that work well on complex problems may be tested. The problem can also be made more complex by adding multiple actions and rewards per attack instance. This would allow the IRS to not only defend against attacks with multiple steps to them, but would also allow for the use of the discount parameter, which unfortunately did not have an effect on the performance in this work. The use of state information and multiple actions/rewards per instance may reduce the inaccuracy of the algorithms to what can be considered optimal times, thus improving on RQ3.

6.3.2 Hyper-parameter optimization

Hyper-parameter optimization has been mentioned before, but was not implemented, due to the excessive testing times that would result from its implementation (see section 2.2.4 and 5.5). However, if a simulated test-bed were to be created, such that the results reflect those in this work, it may be feasible to implement a form of hyper-parameter optimization, so that tests could be conducted with the best possible parameters for each of the algorithms. This would be an improvement, as the parameters in this work were adjusted manually.

6.3.3 Bootstrapping performance

An alternative area of improvement, related to this work, could be to consider a form of bootstrapping, whereby another IRS is used to respond to attacks until the learning algorithm is able to sufficiently respond to attacks itself. This would help in increasing the maximum amount of training instances available to the learning algorithm, without causing damage to the network being protected. The difficulty, however, is that the learning algorithm must undergo training despite the fact that its actions will not have an effect on the network/environment.

6.3.4 SOAR

The advantage of an automated IRSs is the ability to reduce reliance on network administrators or human experts. SOAR is a solution stack that similarly aims to reduce reliance on network administrators by simplifying the process of identifying attacks and responding to them (see section 2.6.2). This is a process that is not only time consuming for network administrators, but is also necessary for IRSs. While

in this work, where the automated IRS responded in an environment in which the problem was simple, and where responses could be manually constructed, in future works, it may be necessary to respond to more complex problems, which require increasingly complex responses (see section 6.3.1). The SOAR solution stack may provide the standardization necessary to construct these responses, enabling them to be utilized more effectively by network administrators. The addition of the IRS in this work would enable these responses to adapt and become more effective over time. If combined effectively, responses generated by SOAR could be generalized, so that they become applicable to many similar attacks, which the learning IRS would then optimize. This combination would further reduce the reliance on network administrators, more than if each system acted independently.

6.4 Summary

This work approached building an IRS that was capable of learning from its own responses. If successful, it would reduce the reliance of network administrators by more effectively responding to attacks and aiding in the solution of the deactivation issue. The background and construction can be seen in chapter 2, and chapters 3 and 4, respectively, while chapters 5 and 6 discuss the findings of this work. It can be seen that the IRS created does learn, and is capable of adjusting to attack type, allowing it to safely respond to DoS attacks without risk to the network being protected. However, it does not perform optimally, as can be seen by the outcome of RQ2 (see section 6.1.2). The performance of the IRS is however, not unexpected, as the problem and the complexity of the IRS were simplified to test the feasibility of a self learning IRS. Several future enhancements have been identified which could potentially improve performance, and perhaps provide for a more complete and functioning system, once developed.

Appendix A

Timer response logs

IDS/IRS:

```
ppo, LENGTH: 60, INSTANCES: 100
==== NEW ROUND ====
Alert: Actual attack time: 41 seconds
Info: Isolating server for: 63 seconds
Alerted at: 1568845805.9950788
Alerted at: 1568845809.999205
Alerted at: 1568845817.0064077
Alerted at: 1568845820.007548
Alerted at: 1568845826.0089672
Alerted at: 1568845829.0121663
Alerted at: 1568845834.0173614
Alerted at: 1568845842.0255587
Alerted at: 1568845843.026756
Alerted at: 1568845845.0753565
Info: timeout was offset by: 23.91987180709839 seconds
-timeDeactivated: 1568845868.9952228
-lastAlert: 1568845845.075351
-offset: 24
==== NEW ROUND ====
Alert: Actual attack time: 45 seconds
Info: Isolating server for: 115 seconds
Alerted at: 1568845869.646484
Alerted at: 1568845874.651672
Alerted at: 1568845876.6538699
...
```

Client:

```
PING 150.1.1.2 (150.1.1.2) 56(84) bytes of data.
64 bytes from 150.1.1.2: icmp_seq=3 ttl=62 time=22.6 ms
64 bytes from 150.1.1.2: icmp_seq=4 ttl=62 time=29.2 ms
64 bytes from 150.1.1.2: icmp_seq=5 ttl=62 time=25.9 ms
64 bytes from 150.1.1.2: icmp_seq=6 ttl=62 time=21.8 ms
64 bytes from 150.1.1.2: icmp_seq=7 ttl=62 time=28.1 ms
64 bytes from 150.1.1.2: icmp_seq=8 ttl=62 time=25.2 ms
64 bytes from 150.1.1.2: icmp_seq=9 ttl=62 time=33.4 ms
64 bytes from 150.1.1.2: icmp_seq=10 ttl=62 time=24.3 ms
64 bytes from 150.1.1.2: icmp_seq=11 ttl=62 time=24.1 ms
64 bytes from 150.1.1.2: icmp_seq=12 ttl=62 time=25.0 ms
```

```
64 bytes from 150.1.1.2: icmp_seq=13 ttl=62 time=25.7 ms
64 bytes from 150.1.1.2: icmp_seq=14 ttl=62 time=25.9 ms
From 150.1.1.1 icmp_seq=15 Packet filtered
From 150.1.1.1 icmp_seq=16 Packet filtered
From 150.1.1.1 icmp_seq=17 Packet filtered
From 150.1.1.1 icmp_seq=18 Packet filtered
From 150.1.1.1 icmp_seq=19 Packet filtered
From 150.1.1.1 icmp_seq=20 Packet filtered
From 150.1.1.1 icmp_seq=21 Packet filtered
From 150.1.1.1 icmp_seq=22 Packet filtered
From 150.1.1.1 icmp_seq=23 Packet filtered
...
From 150.1.1.1 icmp_seq=69 Packet filtered
From 150.1.1.1 icmp_seq=70 Packet filtered
From 150.1.1.1 icmp_seq=71 Packet filtered
From 150.1.1.1 icmp_seq=72 Packet filtered
From 150.1.1.1 icmp_seq=73 Packet filtered
From 150.1.1.1 icmp_seq=74 Packet filtered
From 150.1.1.1 icmp_seq=75 Packet filtered
From 150.1.1.1 icmp_seq=76 Packet filtered
From 150.1.1.1 icmp_seq=77 Packet filtered
64 bytes from 150.1.1.2: icmp_seq=78 ttl=62 time=28.2 ms
From 150.1.1.1 icmp_seq=79 Packet filtered
From 150.1.1.1 icmp_seq=80 Packet filtered
From 150.1.1.1 icmp_seq=81 Packet filtered
From 150.1.1.1 icmp_seq=82 Packet filtered
From 150.1.1.1 icmp_seq=83 Packet filtered
From 150.1.1.1 icmp_seq=84 Packet filtered
From 150.1.1.1 icmp_seq=85 Packet filtered
From 150.1.1.1 icmp_seq=86 Packet filtered
```

R1:

```
*Mar  1 00:00:43.803: %SYS-5-CONFIG_I:
  Configured from console by vty0 (192.168.122.1)
*Mar  1 00:01:47.319: %SYS-5-CONFIG_I:
  Configured from console by vty0 (192.168.122.1)
*Mar  1 00:01:48.003: %SYS-5-CONFIG_I:
  Configured from console by vty1 (192.168.122.1)
```

Appendix B

Scan response logs

IDS/IRS:

```
[+] Initializing program
------------------------------------

  ############################################################################
  #                                                                          #
  #   NON-PRIVILEGED SCAN MODE                                               #
  #                                                                          #
  ############################################################################

  NOTES:
  --------------
  * Some tests will be skipped (as they require root permissions)
  * Some tests might fail silently or give different results

  - Detecting OS...                                            [ DONE ]
  - Checking profiles...                                       [ DONE ]

  ---------------------------------------------------
  Program version:           2.7.0
  Operating system:          Linux
  Operating system name:     Debian
  Operating system version:  9.6
  Kernel version:            5.2.14
  Hardware platform:         x86_64
  Hostname:                  server
  ---------------------------------------------------
  Profiles:                  /home/debian/tmp-lynis/lynis/default.prf
  Log file:                  /tmp/lynis.log
  Report file:               /tmp/lynis-report.dat
  Report version:            1.0
  Plugin directory:          ./plugins
  ---------------------------------------------------
  Auditor:                   [Not Specified]
  Language:                  en
  Test category:             all
  Test group:                all
  ---------------------------------------------------
...
================================================================================
```

```
  Lynis security scan details:

  Hardening index : 55 [###########        ]
  Tests performed : 183
  Plugins enabled : 0

  Components:
  - Firewall               [V]
  - Malware scanner        [X]

  Lynis Modules:
  - Compliance Status      [?]
  - Security Audit         [V]
  - Vulnerability Scan     [V]

  Files:
  - Test and debug information      : /tmp/lynis.log
  - Report data                     : /tmp/lynis-report.dat

================================================================================
...
Info: Scan done, server reconnected
python alert.py  6.65s user 0.31s system 20% cpu 33.525 total
```

Client:

```
PING 150.1.1.2 (150.1.1.2) 56(84) bytes of data.
64 bytes from 150.1.1.2: icmp_seq=1 ttl=62 time=30.2 ms
64 bytes from 150.1.1.2: icmp_seq=2 ttl=62 time=29.1 ms
64 bytes from 150.1.1.2: icmp_seq=3 ttl=62 time=23.6 ms
64 bytes from 150.1.1.2: icmp_seq=4 ttl=62 time=27.4 ms
64 bytes from 150.1.1.2: icmp_seq=5 ttl=62 time=21.4 ms
64 bytes from 150.1.1.2: icmp_seq=6 ttl=62 time=25.8 ms
64 bytes from 150.1.1.2: icmp_seq=7 ttl=62 time=29.6 ms
64 bytes from 150.1.1.2: icmp_seq=8 ttl=62 time=23.9 ms
64 bytes from 150.1.1.2: icmp_seq=9 ttl=62 time=568 ms
64 bytes from 150.1.1.2: icmp_seq=10 ttl=62 time=630 ms
64 bytes from 150.1.1.2: icmp_seq=11 ttl=62 time=45.4 ms
64 bytes from 150.1.1.2: icmp_seq=12 ttl=62 time=25.2 ms
64 bytes from 150.1.1.2: icmp_seq=13 ttl=62 time=25.2 ms
64 bytes from 150.1.1.2: icmp_seq=14 ttl=62 time=35.6 ms
64 bytes from 150.1.1.2: icmp_seq=15 ttl=62 time=35.6 ms
64 bytes from 150.1.1.2: icmp_seq=16 ttl=62 time=35.1 ms
64 bytes from 150.1.1.2: icmp_seq=17 ttl=62 time=35.2 ms
64 bytes from 150.1.1.2: icmp_seq=18 ttl=62 time=36.2 ms
64 bytes from 150.1.1.2: icmp_seq=19 ttl=62 time=35.9 ms
64 bytes from 150.1.1.2: icmp_seq=40 ttl=62 time=30.7 ms
64 bytes from 150.1.1.2: icmp_seq=41 ttl=62 time=29.8 ms
64 bytes from 150.1.1.2: icmp_seq=42 ttl=62 time=30.4 ms
64 bytes from 150.1.1.2: icmp_seq=43 ttl=62 time=31.0 ms
64 bytes from 150.1.1.2: icmp_seq=44 ttl=62 time=21.7 ms
```

```
64 bytes from 150.1.1.2: icmp_seq=45 ttl=62 time=21.7 ms
64 bytes from 150.1.1.2: icmp_seq=46 ttl=62 time=21.8 ms
64 bytes from 150.1.1.2: icmp_seq=47 ttl=62 time=21.9 ms
64 bytes from 150.1.1.2: icmp_seq=48 ttl=62 time=22.7 ms
64 bytes from 150.1.1.2: icmp_seq=49 ttl=62 time=23.1 ms
```

R1:

```
*Mar  1 00:04:18.807: %SYS-5-CONFIG_I:
  Configured from console by vty0 (192.168.122.1)
*Mar  1 00:04:45.487: %SYS-5-CONFIG_I:
  Configured from console by vty0 (192.168.122.1)
```

Bibliography

[1] N. B. Anuar, M. Papadaki, S. Furnell, and N. Clarke. An investigation and survey of response options for intrusion response systems (irss). In *Information Security for South Africa (ISSA), 2010*, pages 1–8. IEEE, 2010.

[2] S. Anwar, J. Mohamad Zain, M. F. Zolkipli, Z. Inayat, S. Khan, B. Anthony, and V. Chang. From intrusion detection to an intrusion response system: fundamentals, requirements, and future directions. *Algorithms*, 10(2):39, 2017.

[3] R. Bace and P. Mell. NIST special publication on intrusion detection systems. Technical report, BOOZ-ALLEN and HAMILTON INC MCLEAN VA, 2001.

[4] G. E. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter*, 6(1):20–29, 2004.

[5] C. Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[6] T. Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[7] J. Cannady. Next generation intrusion detection: autonomous reinforcement learning of network attacks. In *Proceedings of the 23rd national information systems security conference*, pages 1–12, 2000.

[8] X. Chang. Network simulations with opnet. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*, pages 307–314. ACM, 1999.

[9] M. Cohen. Pyflag–an advanced network forensic framework. *Digital investigation*, 5:S112–S120, 2008.

[10] R. Emiliano and M. Antunes. Automatic network configuration in virtualized environment using gns3. In *Computer Science & Education (ICCSE), 2015 10th International Conference on*, pages 25–30. IEEE, 2015.

[11] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[12] F. Fransen, R. Wolthuis, and N. el Ouajdi. Security at machine speed. Technical report, TNO, 2019.

[13] S. L. Garfinkel. Digital forensics research: the next 10 years. *digital investigation*, 7:S64–S73, 2010.

[14] S. Garfinkel and D. Cox. Finding and archiving the internet footprint. *The British Library*, 2008.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[16] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[17] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench, 1994.

[18] M. Huber, M. Mulazzani, M. Leithner, S. Schrittwieser, G. Wondracek, and E. Weippl. Social snapshots: digital forensics for online social networks. In *Proceedings of the 27th annual computer security applications conference*, pages 113–122. ACM, 2011.

[19] Z. Inayat, A. Gani, N. B. Anuar, M. K. Khan, and S. Anwar. Intrusion response systems: foundations, design, and challenges. *Journal of Network and Computer Applications*, 62:53–74, 2016.

[20] W. Kanoun, N. Cuppens-Boulahia, F. Cuppens, and S. Dubus. Risk-aware framework for activating and deactivating policy-based response. In *Network and System Security (NSS), 2010 4th International Conference on*, pages 207–215. IEEE, 2010.

[21] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[22] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas. Supervised machine learning: a review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[23] A. Kuhnle, M. Schaarschmidt, and K. Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. Web page, 2017. URL: https://github.com/tensorforce/tensorforce.

[24] E. Lochin, T. Perennou, and L. Dairaine. When should i use network emulation? *annals of telecommunications-annales des télécommunications*, 67(5-6):247–255, 2012.

[25] S. Muniz and A. Ortega. Fuzzing and debugging cisco ios. *BlackHat Europe*, 2011.

[26] K. Nance, B. Hay, and M. Bishop. Digital forensics: defining a research agenda. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–6. IEEE, 2009.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[28] P. A. Porras and P. G. Neumann. Emerald: event monitoring enabling response to anomalous live disturbances. In *Proceedings of the 20th national information systems security conference*, pages 353–365, 1997.

[29] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In *Icml*, volume 37, pages 1889–1897, 2015.

[30] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[31] A. Shameli-Sendi, M. Cheriet, and A. Hamou-Lhadj. Taxonomy of intrusion risk assessment and response system. *Computers & Security*, 45:1–16, 2014.

[32] A. Shameli-Sendi, N. Ezzati-Jivan, M. Jabbarifar, and M. Dagenais. Intrusion response systems: survey and taxonomy. *Int. J. Comput. Sci. Netw. Secur*, 12(1):1–14, 2012.

[33] N. Stakhanova, S. Basu, and J. Wong. A taxonomy of intrusion response systems. *International Journal of Information and Computer Security*, 1(1-2):169–184, 2007.

[34] C. Strasburg, N. Stakhanova, S. Basu, and J. S. Wong. A framework for cost sensitive assessment of intrusion response selection. In *2009 33rd Annual IEEE international computer software and applications conference*, volume 1, pages 355–360. IEEE, 2009.

[35] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[36] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

[37] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[38] A. Wang, M. Iyer, R. Dutta, G. N. Rouskas, and I. Baldine. Network virtualization: technologies, perspectives, and frontiers. *Journal of Lightwave Technology*, 31(4):523–537, 2013.

[39] E. Weingartner, H. Vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *2009 IEEE International Conference on Communications*, pages 1–5. IEEE, 2009.

[40] D. H. Wolpert, W. G. Macready, et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[41] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.