Machine Learning and Probabilistic Methods for Network Security Assessment

Isaac Matthews

Supervisors: Prof. A. van Moorsel

Prof. S. Soudjani

School of Computing

Newcastle University

This dissertation is submitted for the degree of

Doctor of Philosophy

March 2022

I would like to dedicate this thesis to my family, especially my loving parents Charles and Esther Matthews, for their care and generous support, and without whom this would not have been possible. I would also like to thank Roisin Matthews who never failed to encourage and believe in me, and who became my wife during the PhD process.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 80,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Isaac Matthews

March 2022

Acknowledgements

First, I would like to thank my supervisors Professors Aad van Moorsel and Sadegh Soudjani for their guidance, advice and insights over the course of this PhD. Their feedback and direction were invaluable.

I would also like to thank the EPSRC Centre for Doctoral Training in Cloud Computing for Big Data for providing me with the opportunity to continue my studies, and I thank all involved with the centre for the time and effort they put in. I am grateful to all of the staff at Newcastle University that I met or worked with over the course of my studies.

The staff of XQ Digital Resilience were always generous with their time and attention, as well as their expertise, and made me feel welcome whenever I visited. I would like to thank them, and especially Dr. Tom Duffy who became a friend and always listened to my many bad ideas.

Finally, I would like to thank Dr. Nigel Thomas and Professor Emil Lupu of Imperial College London for a challenging and well researched viva examination.

Abstract

Computer networks comprised of many hosts are vulnerable to cyber attacks. One attack can take the form of the exploitation of multiple vulnerabilities in the network along with lateral movement between hosts. In order to analyse the security of a network, it is common practice to run a vulnerability scan to report the presence of vulnerabilities in the network and prioritise them with an importance score. The scoring mechanism used primarily in the literature and in industry ignores how multiple vulnerabilities could be used in conjunction with one another to achieve a goal that previously was not possible. Attack graphs are a common solution to this problem, where a scan along with the topology of the network is turned into a graph that models how hosts and vulnerabilities can be connected. For a large network these attack graphs can be thousands of nodes in size, so in order to gain insight from them in an automated way, they can be turned into Bayesian attack graphs (BAGs) to model the security of the network probabilistically. The aim of this thesis is to work towards the automation of gathering insight from vulnerability scans of a network, primarily through the generation of BAGs.

The main contributions of this thesis are as follows:

1. Creation of a unified formalism for the structure of BAGs and how other graphs can be translated into this formalism.

2. Classification of vulnerabilities using neural networks.

3. Design and evaluation of a novel technique for approximation in the computation of access probabilities in BAGs (referred to in the literature as the static analysis of BAGs) with no requirement for the base graph to be acyclic.

4. Implementation and comparison of three stochastic simulation techniques for inference on BAGs with evidence (referred to in the literature as the dynamic analysis of BAGs), enabling security measure evaluation and sensitivity analysis.

5. Demonstration of a sensitivity analysis for BAG priors and a novel method for quick computation of sensitivities that is more readily analysed than the traditional technique.

6. Development and demonstration of a fully containerised pipeline to automatically process vulnerability scans and generate the corresponding attack graph.

With a single formalism for attack graphs, alongside an open-source attack graph generation pipeline, our work serves to enable future progress and collaboration in the field of processing vulnerability scans using attack graphs by simplifying the process of generating the graphs and having a mathematical basis for their evaluation. We design, implement, and evaluate various techniques for calculations on BAGs. For the process of computation of access probabilities we provide an algorithm that requires no processing or trimming of the initial graph, and for inference on BAGs we recommend likelihood weighting as the best performing sampling technique of the three we implement. We also show how inference techniques can be applied to sensitivity analysis on BAGs, and provide a new method that allows for more efficient and interpretable sensitivity analysis, enabling more productive research into the area in future. This research was originally undertaken in collaboration with XQ Cyber.

Table of contents

List of figures

List of tables

Nomenclature

Mathematical Symbols

v'	Child of v
$p : \mathscr{V} \rightarrow [0,1]$	Local probability assignment
$\mathscr{G}$	Bayesian attack graph
$\mathfrak{B}$	Bayesian network tuple
$\mathscr{E}$	Set of edges
$\mathscr{V}$	Set of nodes
$\mathscr{T}$	Set of probability tables
N	Number of samples
$pa(v)$	Parents of v
$pa(v)''$	Instantiated parents of v
$pa(v)'$	Uninstantiated parents of v
$XP(pa(v)')$	Set of all conditioning cases possible for the uninstantiated parents of node v
π	Configuration of nodes specified by the evidence
V_a	Set of AND nodes

V_l	Set of LEAF nodes
V_o	Set of OR nodes
$sp(v)$	The state space of v
Z	Set of nodes present in the given evidence

Acronyms / Abbreviations

AG	Attack graph
BAG	Bayesian attack graph
BN	Bayesian network
BoW	Bag-of-words
BS	Backward simulation
CCE	Categorical cross-entropy
CVE	Common vulnerabilities and exposures
CVSS	Common vulnerability scoring system
DAG	Directed acyclic graph
FN	False negative
FP	False positive
HACL	Host access control list
LW	Likelihood weighting
MulVAL	Multi-host, multi-stage vulnerability analysis language [86]
NASL	Nessus attack scripting language

NN	(Artificial) Neural network
NVD	National Vulnerability Database [80]
OpenVAS	Open vulnerability assessment system [35]
OVAL	Open vulnerability and assessment language [14]
PLS	Probabilistic logic sampling
PPV	Positive predictive values
TF-IDF	Text frequency-inverse document frequency
TN	True negative
TPR	True Positive Rate
TP	True positive

Chapter 1

Introduction

As modern businesses, and society as a whole, become more dependent on technology for their operation, the damage that can be caused by a cyber attack increases. An attack can cause many unwanted outcomes, like the interruption of important services in a denial of service attack, or the exfiltration of private user information from a database breach. These occurrences can compromise the privacy and security of users, terminate small enterprises and start-ups, and cost large business millions of pounds (for example: the Epsilon data breach cost $4 billion, $252 million was stolen from credit cards in the Hannaford Bros attack, and WannaCry cost the NHS £100 million [109, 3]). Cybercrime incurs an estimated cost of $6 trillion USD globally in 2021, and is estimated to rise to an annual cost of $10.5 trillion by 2025 [76]. Discovering what problems are present in the security of the network is thus extremely valuable as it enables remediation of the network and prevention of an attack.

A *vulnerability scan* of a given network is a scan of the network that provides information on the topography of the network along with information as to the vulnerability of the hosts on that network with respect to known attacks and exploits. A scan can report information about a variety of circumstances that impact the security of a host or the network as a whole. For example a scan might detect: out of date software running on a machine that needs to be patched to have the latest security fixes applied, a host that allows the use of default login credentials for access, or enabled protocols on the network that are not secure.

Networks with many hosts currently have too many vulnerabilities to fix in a reasonable time scale. At present, after a scan of a network, the vulnerabilities with the highest 'score' are prioritised, a metric defined arbitrarily by one of many organisations. The most common example of this is using the common vulnerability scoring system, or CVSS, score to decide on the importance of vulnerabilities [108]. The present methodology ignores the structure of the network and the damage that can be done by moving through a network and engaging multiple hosts. Ideally, this prioritisation should depend on the context of the network, and should evaluate the scope that a vulnerability gives for continued intrusion on top of the severity of the individual vulnerability.

An attack graph is a representation of a system and its vulnerabilities in the form of a directed acyclic graph (DAG). It models how the system's vulnerabilities can be leveraged during a single attack to progress through a network. Many types of attack graphs and similar models exist, but in general there are two main categories: state attack graphs, and dependency attack graphs. State, or complete, attack graphs model entire network states including all vulnerability and network information and look at transitions between the states of the network through the use of model checking (sometimes called scenario graphs) [100, 103]. Compact, or dependency, attack graphs use the causal dependencies of every individual vulnerability to make a model [116, 86, 45]. We will use dependency attack graphs as opposed to the state enumeration style of attack graphs in order to better understand the importance of individual vulnerabilities [100].

One of the objectives of this thesis is to combine Bayesian statistics with attack graphs to better automatically prioritise network vulnerabilities from a probabilistic view point. This will allow attack graph analysis to be automated while also improving the actionable insights taken from them. The combining of attack graphs with Bayesian networks creates something called a Bayesian attack graph, or BAG. BAGs have been well formalised in literature; Frigault et al. [26] demonstrate how a BAG can be constructed mathematically using CVSS scores to create probabilities that are propagated through the attack graph, and Xie et al. discuss important considerations and think more practically about how one might overlay a Bayesian network onto an attack graph [120]. However, as Miehling, Rasouli and

Teneket observe [75], constructing a BAG is a non-trivial process and thus a platform that can automate their generation would be a significant contribution to the area.

The reasoning for using Bayesian networks as our method for introducing probabilities on these attack graphs is multi-faceted. Primarily, the reason is that there is inherent uncertainty in the analysis of the resilience of a network and as such a probabilistic viewpoint makes the most sense. Bayesian networks already have a large body of research into efficient computation for DAG structures as examined by Kordy et al. in their section on Bayesian networks for security [60]. On top of this, the network can be re-evaluated to calculate a posterior distribution given some new set of data, whether that be changes in the network or a current attack threat, allowing for dynamic modelling of the system.

Once a BAG has been generated, various processes can be used to extract value in the form of a quantitative evaluation of a network's security or a prioritisation of vulnerability patching. The BAG can be analysed from an exclusively static perspective in order to identify the weakest areas of a network, as well as identify quantitatively the risk that a certain asset will be compromised in the case of an attack. Further extensions to this kind of analysis include the introduction of attack profiles to modify the probabilities; for example this could be done using the attack complexity metric in the CVSS base vector to determine the ease of an attack. One approach to this is detailed by Cheng et al. [115] along with a method to include dependency relationships between vulnerabilities. Another application of the BAG is as a dynamic risk assessment tool. This is a method where an administrator can model new security controls and their effects on a network, as well as dynamically analyse a deployed network's most likely attack paths that can be updated dependent on information from something like an intrusion detection system [91].

Some work on using stochastic modelling techniques has been done for BAGs too. The automation of this process, however, is yet to be investigated, as is the performance of different simulation techniques. The application of these stochastic modelling techniques for the purpose of sensitivity analysis has also been performed to some extent in the literature [84], but the process is inefficient and requires analysis by hand.

The research in this thesis was originally undertaken in collaboration with XQ Cyber, a cyber security enterprise, who developed a vulnerability scanner known as CyberScore. As such, the examples used in most cases are generated using the CyberScore vulnerability scanner. However, the formal methods and techniques are applicable to any vulnerability scan, and indeed CyberScore is an improvement on the base open-source scanner OpenVAS [35] and as such the technology can be readily used on widely available OpenVAS scans. Scans from the popular vulnerability scanner Nessus [110] have also been processed and are compatible with the full software pipeline.

1.1 Research Motivation and Problems

This research is undertaken with the objective of maximising the value of *vulnerability scans* on networks. This can be approached from multiple angles, from increasing the ease of comprehension of a scan to novel methods to analyse a scan. The outcome of a scan should be an enterprise's network administrator fixing important vulnerabilities in the network; we research how to correctly prioritise vulnerabilities from a scan and present the information to the administrator in such a way that the scan's utility is maximised. This will involve investigating how expert knowledge can be disseminated such that someone with little expertise can understand a vulnerability scan and its discovery of problems on a network. Consideration must also be given to the scalability of any solution due to the large and ever-growing size of computer networks in industry. These concepts can be distilled into a single question:

Given a vulnerability scan, what should be done to most efficiently harden the network?

Efficient use of a vulnerability scan to harden a network involves both the speed and simplicity of the processing of the scan and the resulting output, as well as identifying and prioritising the biggest risks to the network. In order to achieve this goal we identify and investigate the following research problems in the area of vulnerability scanning:

Vulnerability dataset The largest freely available dataset for vulnerabilities and the most commonly used, the National Vulnerability Database [80], is suffering from missing entries, data duplication, erroneous chronology, as well as a lack of documentation. There is a need to develop methods that can work with this database and produce sufficiently accurate results while resolving the issues around these negative aspects of the database.

Complex scan outputs The output of a vulnerability scan is a large and complex list of vulnerabilities and hosts that can be difficult to translate into actionable steps to remove the vulnerabilities. There is a requirement for a process of simplification such that a scan for a large network can be efficiently used, and that the process for remediation for a scan of any size can be understood without a security professional.

Barrier for entry to process scans and generate attack graphs In order to create an attack graph, either proprietary software must be used, or complex setup and installation procedures are required. Even if the installation is successful, the scans themselves are likely to be required to have been generated by proprietary scanning software. For more people to be able to perform research in the area there should be a simple, widely compatible, and freely usable platform for the generation of attack graphs.

We also investigate further research problems that exists in relation to the creation, use and analysis of Bayesian attack graphs:

Bayesian attack graph formalisms Several formalisms exist to define attack graphs. However these formalisms are generally not rigorously defined, and their relation to each other is unknown. A single, well defined formalism that other formalisms can be translated into would allow for a common ground for research

discussions, without invalidating past research.

Cycles in attack graphs When an attack graph is generated, cycles naturally occur. In order to understand an attack graph probabilistically it is transformed into a Bayesian network, but Bayesian network techniques require that directed graphs are acyclic. Methods should be developed that do not have the acyclicity constraint.

Belief updating for Bayesian attack graphs In the event of an attack on the network, the probability that an attacker will reach a specific host will change depending on the progress the attacker has made so far. The probabilities throughout the Bayesian attack graph of the system should also change accordingly, given information about the attack. Techniques for performing this inference process should be implemented and compared.

Analysis of effect of prior assignment When creating and analysing attack graphs, the original assignment of the probabilities on the nodes is imprecisely decided using information about the vulnerability in question from a vulnerability database. The effect of this assignment needs to be investigated. Currently, the only process for investigating these assignments is computationally expensive and requires complex analysis. As such, a new technique that improves the computation efficiency and produces more readily usable results is necessary.

1.2 Goals and Original Contributions

The work presented in this thesis makes up a number of original contributions. The most significant are:

1. Creation of a unified formalism for the structure of BAGs and how other graphs can be translated into this formalism. This allows a formal and well defined basis from

which current and future discussion of BAGs can be performed, while still allowing past research to be used within the new framework.

2. Classification of vulnerabilities using neural networks. The vulnerabilities within the NVD database are sorted using a machine learning model into families that share a remediation process, simplifying the reporting of the discovery of vulnerabilities and eliminating the problems that exist with the NVD data.

3. Design and evaluation of a novel technique for approximation in the computation of access probabilities in BAGs (referred to in the literature as the static analysis of BAGs) with no requirement for the base graph to be acyclic. The accuracy of this technique has been evaluated against exact methods and on common examples, and the performance has been tested on large graph sizes.

4. Implementation and comparison of three stochastic simulation techniques for inference on BAGs with evidence (referred to in the literature as the dynamic analysis of BAGs), enabling security measure evaluation and sensitivity analysis. Using different stochastic simulation techniques has not yet been investigated in the literature, and these techniques allow for performing inference on BAGs so they can be used with an intrusion detection system.

5. Demonstration of a sensitivity analysis for BAG priors and a novel method for quick computation of sensitivities that is more readily analysed than the traditional technique. This new technique allows future research into the effect of prior assignment of vulnerabilities on the resulting attack graph. A similar process can also be used to measure the effect of implementing new security measures on the network.

6. Development and demonstration of a fully containerised pipeline to automatically process vulnerability scans and generate the corresponding attack graph. The pipeline is open-source, and allows anyone to convert vulnerability scans into attack graphs, as well as including a module for implementing custom visualisation techniques. The

input scans can be generated by free open-source scanning tools, meaning the entire process is available to all.

1.3 Publications

During the course of my PhD, I have authored the following publications:

[70] Matthews, I., Mace, J., Soudjani, S., van Moorsel, A. (2020). Cyclic Bayesian Attack Graphs: A Systematic Computational Approach. 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 129-136

This work introduces the formalisms for Bayesian attack graphs as well as discussing and demonstrating cycle evaluation. We calculate probabilities for attack graphs regardless of the presence of cycles. We achieve this by using an algorithm that we go on to evaluate. The formalisms discussed form part of Chapter 2. The discussion and evaluation of cyclic attack graphs and the algorithm we use form the basis for Chapter 4.

[71] Matthews, I., Soudjani, S., van Moorsel, A. (2021). Stochastic Simulation Techniques for Inference and Sensitivity Analysis of Bayesian Attack Graphs. 2021 International Conference on Science of Cyber Security (SciSec), pp.171-186 This work is also published in Springer's Lecture Notes on Computer Science book series. This work discusses the use of stochastic simulation techniques to perform inference on Bayesian attack graphs in the presence of evidence. Three techniques are described, implemented, and then compared. The use of these techniques to perform sensitivity analysis is then discussed and performed. This work forms the basis for Chapter 5.

Papers not forming part of this thesis

[97] Ryder, T., Prangle, D., Golightly, A., Matthews, I. (2021). The Neural Moving Average Model for Scalable Variational Inference of State Space Models . The 37th Conference on Uncertainty in Artificial Intelligence (UAI).

This work has been accepted for publication at the 2021 Conference on Uncertainty in Artificial Intelligence. In this work, we extend mini-batch variational methods to state space models of time series data. To do this we introduce a novel generative model as our variational approximation, a local inverse autoregressive flow. This allows a subsequence to be sampled without sampling the entire distribution allowing short portions of the time series to be used in training iterations at low computational cost. We illustrate this method on AR(1), Lotka-Volterra and FitzHugh-Nagumo models and achieve accurate parameter estimation in a short time.

1.4 Thesis Structure

Figure 1.1 shows the thesis structure, as well as the prerequisites for each chapter and any context. The thesis can be read from start to finish in order. Alternatively, if only one of the contribution chapters is to be read, Chapters 3 to 6, the Background chapter should also be read beforehand. Chapter 6 provides a technical context for Chapters 4 and 5, but is not required for understanding. It also showcases a method for implementing Chapter 3 in a visualisation technique. Chapter 7 discusses the links between each chapter and how they can be used as a whole to achieve the goal of network hardening using vulnerability scans and as such requires all other chapters to be read.

1.5 Chapter Summaries

2. Background and Related Work. This chapter introduces the concept of vulnerabilities and their importance to security and privacy for people and businesses alike. Next, we explore how vulnerabilities are analysed and how the results of such

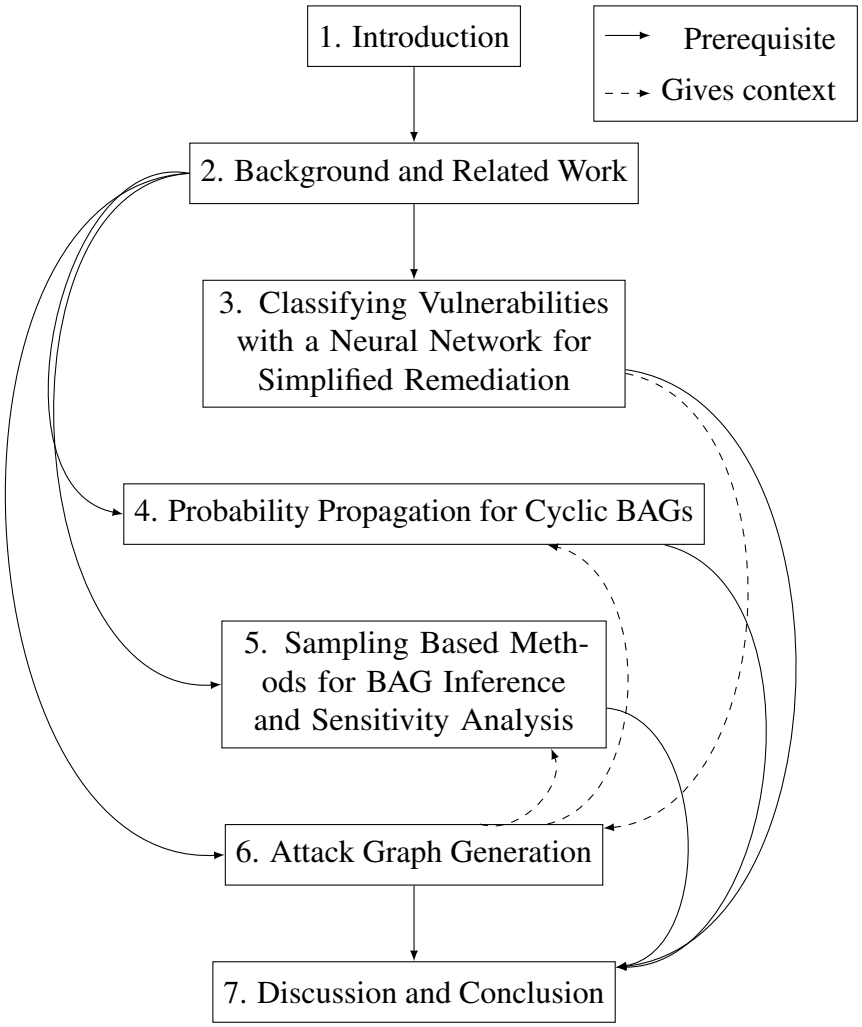


Fig. 1.1 Structure of thesis contents.

an analysis can be turned into a model. We then introduce two formalisms for such models, including a unified formalism that other formalisms can be translated into, fulfilling Contribution 1 in the Abstract. Finally we examine other related work relevant to the models.

3. Classifying Vulnerabilities with a Neural Network for Simplified Remediation In this chapter we introduce a new application for machine learning with the vulnerability dataset. We demonstrate that a neural network can be trained as a classifier to automatically sort vulnerabilities into families. The aim of this work is to simplify the understanding and remediation steps for users who perform a vulnerability scan. We demonstrate the accuracy and efficacy of our model for the task and so fulfil Contribution 2 in the Abstract. We then discuss and demonstrate a technique that can further improve the accuracy of the classifier, and finally we discuss any related and future work in the area.

4. Probability Propagation for Cyclic BAGs This chapter is adapted from our paper "Cyclic Bayesian Attack Graphs: A Systematic Computational Approach". We introduce the idea of cyclic attack graphs, and examine a process for understanding and evaluating the probabilities within them. We interpret the cycles using combinational logic circuits with probabilistic inputs, as well as performing exact calculations using variable elimination. We present an algorithm that approximates probabilities throughout the attack graph without modifying it in any way, fulfilling Contribution 3 in the Abstract, and develop an attack graph simulator to demonstrate the performance of the algorithm. We also compare the algorithm to variable elimination, and show its calculations for common attack graph examples in the literature (both cyclic and acyclic).

5. Stochastic Simulation Techniques for Inference and Sensitivity Analysis of Bayesian Attack Graphs This chapter is adapted from our paper "Stochastic Simu-

lation Techniques for Inference and Sensitivity Analysis of Bayesian Attack Graphs". We introduce the three objectives of the work: to compute access probabilities of attack graphs using stochastic simulations, then to perform inference on the attack graphs using a set of evidence values, and then to apply these techniques for the sensitivity analysis of the prior access probabilities. We introduce and discuss three techniques for performing stochastic simulation (probabilitistic logic sampling, likelihood weighting, and backward simulation), fulfilling Contribution 4 in the Abstract. We then implement and compare the three techniques and show that for most applications likelihood weighting is superior. Then, we show how these techniques for performing inference can be applied to sensitivity analysis, in fulfilment of Contribution 5 in the Abstract. Finally, we discuss and compare with related work.

6. Attack Graph Generation Platform In this chapter we discuss the technical basis for our work throughout the thesis, in the form of a software pipeline that enables the user to process and visualise a vulnerability scan using open-source software. This fulfils Contribution 6 in the Abstract. We describe the overall requirements and use cases for the platform. Next, we introduce all of the software used in the creation of the pipeline, and discuss how we modify or use each of them. Then, we describe the requirements for each step in the process of generating and visualising an attack graph, and we show the final layout of the platform. We also show three usage examples for the visualisation platform. This work allows any user to quickly begin generating, investigating and visualising attack graphs without the need for complex setup processes or long lists of prerequisite software.

7. Discussion and Conclusion Finally, we discuss how all of the work presented in this thesis can be used to achieve the overall goal of efficiently using vulnerability scans to improve network security. We examine the limitations of our work, and suggest and discuss several future possibilities for the continuation of progress in

our area of research.

Chapter 2

Background and Related Work

In this chapter we demonstrate that while dependence on computer networks is large and ever increasing, so is the vulnerability of these networks and the importance of keeping them secure. We discuss vulnerabilities and a vulnerability categorisation system, and methodology for discovering vulnerabilities in networks. We then look at using vulnerabilities to build graphical models and different tools for this process. We introduce two formalisms and demonstrate their equivalence, as mentioned in Contribution 1. Finally we discuss the body of work concerning attack graphs and their computation.

2.1 The Danger of Vulnerability

Businesses are becoming more and more reliant on technology. Large businesses have required technology for communication at the minimum, but a recent investigation by Deloitte has shown that 99% of small businesses use digital tools in their day-to-day workflow [21]. There is also a correlation between being more reliant on digitalisation and being more profitable and successful. In an ecosystem with such a dependence on technology there is necessarily more opportunity for attackers; in 2019 Ipsos Mori surveyed over 2000 organisations in the UK and discovered that approximately one-third of small businesses

knew they had been breached within the last twelve months, while two-thirds of medium and large business knew of a breach [46]. The median number of breaches for a medium business was 6, while large businesses had a median of 12 breaches with an average cost of over £20,000 for each breach. As such, the security of the networks used is highly important from both a pecuniary stand point, as well as for the protection of sensitive user data.

2.2 Vulnerabilities

A vulnerability in the context of the security of computers and computer networks is a complex thing to define. Generally speaking in the context of computing a vulnerability is a weakness or flaw in a system that can be exploited either intentionally or accidentally with an outcome outside of the normal use of the system [106, 104]. The system within which the vulnerability exists can be one of many things including operating systems, misconfigurations, installed software, hardware, security policies and running protocols. As new software is released, and other software becomes older and better understood, more vulnerabilities are discovered. Using data from the data feeds produced by the National Vulnerability Database (NVD) maintained by NIST [80], Figure 2.1 shows the increasing number of vulnerabilities reported each year. In 2020 there was an average of just over 50 new vulnerabilities reported each day, and there a vulnerabilities present in every major operating system available. Vulnerabilities are ubiquitous, and the list is constantly growing, meaning that the process of securing a network is never complete and contributions that increase this process' ease or efficiency are valuable.

2.2.1 Common Vulnerabilities and Exposures System

When a vulnerability is discovered, it is disclosed using the Common Vulnerabilities and Exposures (CVE) system, a system developed and maintained by the Mitre Corporation. It is assigned a CVE identifier that has the prefix of 'CVE' and the year of disclosure, and is followed by a series of digits to identify the vulnerability:'CVE-YYYY-NNNN'. The CVE is added to the NDV along with its Common Vulnerability Scoring System (CVSS) vector.

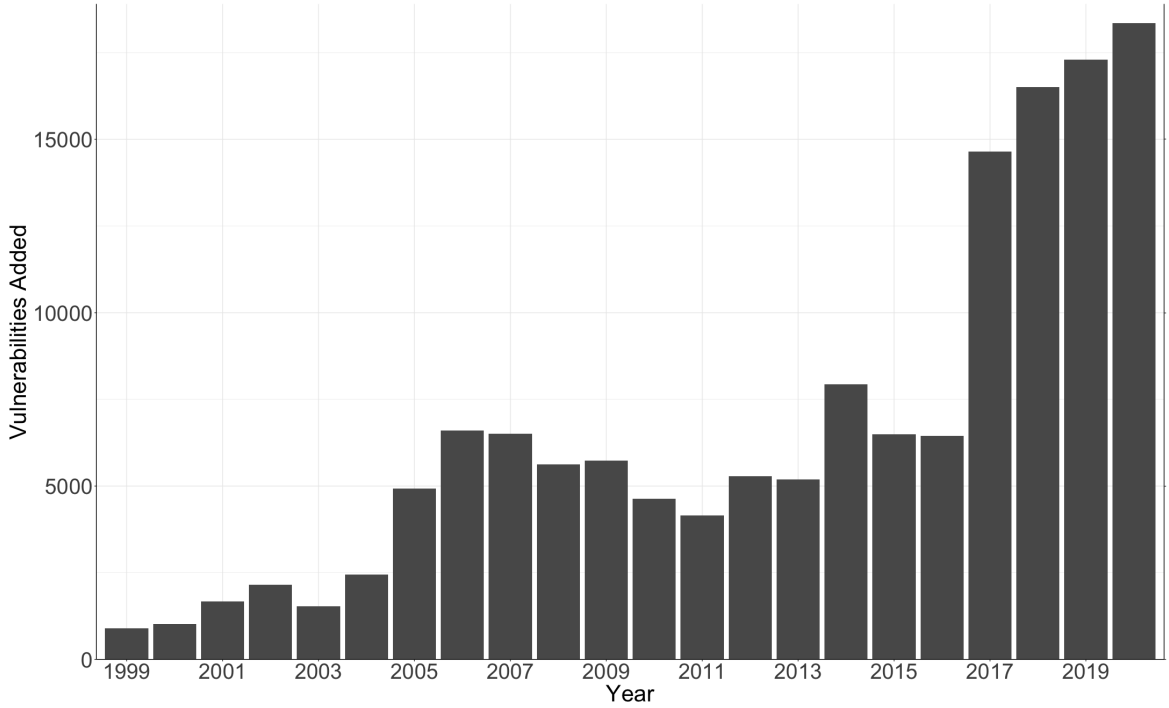
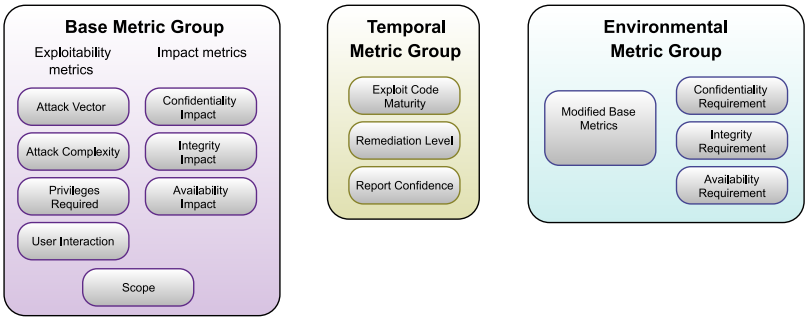


Fig. 2.1 Vulnerabilities reported and subsequently added to database listings per year.

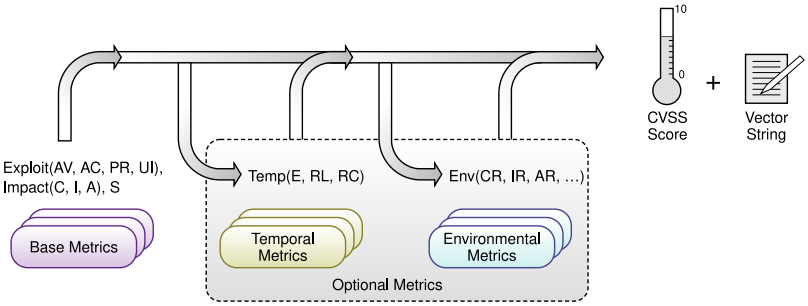
This is an open standard maintained by FIRST [25] that assigns a vector to a vulnerability that encodes all the information available about the vulnerability, along with an overall 0 to 10 score generated from this vector that is used to estimate the severity of the vulnerability, with a rating of 'None' at 0.0 or 'Low' at 0.1 to 3.9 up to 'Critical' severity at scores between 9.0 and 10.0. The score available from the NVD is generated exclusively from the Base metric group, but can be modified using the *Temporal* and *Environmental* metric groups in order to make the score specific to an implementation. This process is shown in Figure 2.2b. All of the metrics available in the vector and their groupings are shown in Figure 2.2a. We will discuss the metrics most relevant to attack graphs next.

2.2.2 CVSS Metrics

The vectors are comprised of three groups of metrics. The *Base* metric group is the representation of information specific to the vulnerability that does not change over time or with the location of the vulnerability. It is further divided into two groups: *Exploitability*



(a) The metrics used in the CVSS [25]



(b) Calculating a CVSS score [25]

Fig. 2.2 The Common Vulnerability Scoring System

which consists of how readily the vulnerability can be exploited in terms of complexity, pre-requisites and if it can be remotely performed, and *Impact* which measures the effect of exploitation in confidentiality, integrity and availability. Next is the *Temporal* group, which represents the components of a vulnerability that change over time, especially the availability and quality of code to perform the exploit and any remediation efforts. Finally, the *Environmental* group is used for specific instances of a vulnerability on a network to evaluate a score while factoring specifics of a network that may modify any of the previous information set in the *Base* metrics, and provides a way to set requirements for maximum impact of a vulnerability on the network.

Base Group

Attack Vector The *Attack Vector* metric measures the distance from which an attack can be performed, in terms of both the physical distance and the logical distance. It can take any of the values shown in Table 2.1. This is an important vector as there is a large difference

Table 2.1 *Attack Vector* metric values.

Value	Score	Description
Network	0.85	The vulnerability can be remotely exploited.
Adjacent	0.62	An attack must come from somewhere logically adjacent to the vulnerable component. For example, local network access may be required, or physical proximity may be required for a Bluetooth based attack.
Local	0.55	The attacker must have local access either physically using the keyboard or via protocols like SSH and RSH. Alternatively user interaction may be required like the clicking of a compromised URL.
Physical	0.2	Physical access to the vulnerable device is required, for example any USB based attack.

Table 2.2 *Attack Complexity* metric values.

Value	Score	Description
Low	0.77	An attack does not require a great deal of knowledge or special conditions to succeed, it should be repeatable with a majority of success.
High	0.44	There are requirements for the success of an attack that may be outside of the control of an attacker, or a reasonable amount of preparation is required. For example certain configuration settings must be known to the attacker, or the attack relies on an amount of social engineering.

in the number of potential attackers when comparing attacks that can be performed over the internet versus an attack that requires physical disk access, and as such there is a sizeable difference in scoring between the most and least problematic attack vectors.

Attack Complexity Table 2.2 shows the two values available for *Attack Complexity*. This metric encodes the difficulty of exploiting the vulnerability, especially with regard to conditions that the attacker can not control. Generally speaking if the attack is repeatable with a high level of success and does not require a certain environment or user interaction it is Low complexity, otherwise it is High.

Privileges Required The privileges that are required at the beginning of the exploit are encoded in this metric. There are three possible value roughly equating to privilege levels of:

Table 2.3 *Privileges Required* metric values.

Value	Score	Description
None	0.85	The attacker can begin the attack unauthorised.
Low	0.62	A privilege level similar to a basic user is required beforehand by the attacker, or the attacker must have access to non-sensitive network components.
High	0.27	The attacker must have acquired administrator-tier privileges before launching the exploit, with control over an entire network component or across many components.

Table 2.4 *Exploit Code Maturity* metric values.

Value	Score	Description
Not Defined	1	Not enough information.
High	1	No code required with readily available information, or fully autonomous code is accessible. The code works for any situation.
Functional	0.97	Working code is available for the exploitation of the vulnerability and it works in the majority of situations.
Proof-of-Concept	0.94	The code available is proof-of-concept and may not work in many situations or requires a large amount of input from the attacker.
Unproven	0.91	The exploit is still theoretical.

none, normal user, and administrator. A full description and the values are shown in Table 2.3.

Temporal Group

Exploit Code Maturity This is a metric from the *Temporal Group* and as such will change over time and must be re-evaluated whenever it is used. The metric quantifies the availability and ease of use of any code that can be used to exploit the vulnerability.

2.2.3 Scanning for Vulnerabilities

Vulnerability scanning is an automated process that uses software to test an attack surface and discover vulnerabilities and security flaws. A vulnerability scanner can be specific to a certain

technology, for example a web application vulnerability scanner crawls through web pages to discover incorrect configurations and authentication problems, and agent-based scanners are installed on specific devices to deliver local reports on that system's vulnerabilities. Alternatively, vulnerability scans can be 'agentless' and instead are run across the entirety of a computer network. This type of scan can incorporate the other types of scan, and in general works by communicating with each host in a network and running a battery of tests against the hosts, depending on the initial information discovered by the scan. For example, if a specific port is discovered to be open on a host, all relevant tests to that port would be run against that host to discover what vulnerabilities are present. There is a further distinction between authenticated and unauthenticated scans. An unauthenticated scan performs the scan with no credentials for the network and as such performs a more limited analysis of the available routes for an attack. An authenticated scan is provided with the credentials for the network and so can fully evaluate all running software, operating systems and available protocols in the network. From here on we use *vulnerability scan* to mean agentless vulnerability scan with full authentication.

Any vulnerability scan generated for this work was performed using the CyberScore scanner developed by XQ Digital Resilience. The CyberScore scanner is built from the OpenVAS [35] scanner tool with extra tests performed using the NASL scripting language to test for more vulnerabilities. OpenVAS (Open Vulnerability Assessment System) is an open source vulnerability platform that can be used to scan and manage vulnerabilities, and it can be used in place of the CyberScore scanner as the two have identical output types. Another popular scanner, the proprietary Nessus scanner, can also be used after it is translated using the system described in Chapter 6.

2.3 Graphical Models for Security Analysis

Graphical models are used in many areas of security analysis due to their visual interpretability and ability to represent many pieces of information in conjunction with each other, along with their algorithms and formalisms for quantitative analysis. Attack trees were one of the

first examples of the use of a graphical model for security in 1998 by Salter et al. [99] and Schneier in 1999 [101]. An attack tree is a tree structure that has a specific goal as it's root node. The rest of the nodes in the graph are different attacks that can be achieved to reach the goal node, with edges representing requirements between the nodes. A good description of the notation and formalism of attack trees is by Mauw and Oostdijk [72]. They are, in general, only used as a static analysis, although some work has been done to include defence actions in the trees [61, 59]. Various permutations of attack trees also exist, like attack fault trees [62] that include fault trees in the model to incorporate information on the safety of the network, and ordered weighted average trees that introduce probabilities into operations [122]. A description and comparison of attack tree variations is available in Hong et al. [40]. However, the constraint of only being able to represent one goal in a complex network is a significant one given the size of modern enterprise networks that have many assets that require protection. For this reason we look to attack graphs as the replacement for attack trees, enabling more complex modelling and analysis that can evaluate many different goals in a network while also preventing duplication in terms of similar routes between attack trees.

2.4 Attack Graphs

Attack graphs are graphical models of a network's insecurities. They model how connections between hosts and vulnerabilities present in the network can be leveraged to achieve states of privilege in a network. There are two main types of attack graph. State, or complete, attack graphs [107] model an entire network state, or macro-state, as a whole and use model checking to discover state transitions. This form tends to be less common as it suffers from the state explosion problem as more vulnerabilities are added meaning it can quickly become cumbersome [117]. Compact attack graphs are the alternative, and function by encoding individual vulnerabilities, actions and states and the causal dependency between them. In these attack graphs, the nodes approximately represent an attackers location in an attack situation, and the edges represent the dependencies between locations and states in the network (a more precise definition is presented in Definition 2.4.4).This form is more

popular due to its scalability and interpretability with regard to specific vulnerabilities, and in general where the literature refers to attack graphs these are compact attack graphs. Other less common attack graph models are attack scenario-based attack graphs which contain nodes representing actions that an attacker can take, with edges representing the sequence of attacks in the the attack scenario [41], and host-based attack graphs. In a host-based attack graph each host is represented as a node on the graph and any exploit that allows an attacker to move between hosts is a labelled edge on the graph [125].

These attack graphs can be further separated into two groups based on how the pathways in the graph are determined. One technique is using a logic-based method to deduce attack paths using knowledge of exploits and their relationship with network states under a set of rules. MulVAL is an example of a system that uses this method, using XSB (an extension to the logic engine and programming language Prolog) to query over the set of relations made up of facts, which are the properties of the network like the hosts and exploits that are present, and rules, which is a pre-defined ruleset that defines possible behaviour. The other technique is using a graph traversal method to perform a search and generate the nodes and edges in the graph. An example of this is in Ammann et al. [5] where the authors use a breadth first search, starting at the initial nodes in the graph and creating layers according to how many conditions must be satisfied to reach a node in that layer.

In order to create an attack graph, certain information needs to be collected regarding the network. The output of an authenticated vulnerability scan should be sufficient to create an attack graph, as it will contain: the vulnerabilities and their locations in the network, a listing of all the services and programs running on each host, and a *HACL* or host access control list. A HACL is a representation of the allowed connections within the network. An entry into the list will contain the IP addresses or names of the client and the server in a connection, the protocol or service being used, and the port used in the connection, `(client IP/name, server IP/name, protocol, port)`. For example, if a user is allowed access to a MySQL database using the default port 3306 then one of the entries on the HACL would be `(workstation,MySQLDatabase,TCP,3306)`.

This information is then supplied to an attack graph generator in order to construct the attack graph. There are many options for attack graph generators, and several survey papers have been compiled to compare these tools [123, 45, 7]. There are both open source (MulVAL [86], TVA [47], NetSPA [45]) and proprietary (Cauldron, FireMon [24], Skybox View) offerings. We have chosen to use MulVAL to generate our attack graphs for three reasons: (1) its is open source and can be readily and freely used, (2) it scales well compared to other techniques, using a polynomial algorithm $O(n^2)$ to $O(n^3)$ with regard to number of hosts, (3) it is the tool used in the majority of the implementations of the literature.

An attack graph, or *AG*, can become a Bayesian attack graph, or *BAG*, when the elements of the graph are associated with probabilities. These probabilities are often generated using the scores mentioned in Section 2.2.2 (for a more detailed discussion refer to Chapter 3). Depending on the attack graph generator used, the resulting BAG will be in a certain formalism. Next, we define the two formalisms and demonstrate that one is more general, as well as demonstrating how to convert between the two formalisms.

2.4.1 Formalisms

In this section, we introduce two different attack graph formalisms widely used in the literature for modeling a network from the security perspective. The first one is proposed in [107] and used in [48, 47**?**] and the second one in [85] and is used in much of the rest of the literature. We will then show in Section 2.4.2 that the second formalism is more general than the first one, and work with that representation after this section. Note that both formalisms require the attack graph to be acyclic. We use these formalisms as a basis for generalising them to cyclic graphs. We first define cycles and loops in directed graphs.

Definition 2.4.1 *Given a directed graph $G = (V,E)$ with the set of nodes V and the set of edges $E \subset V \times V$, a* cycle *is a sequence of nodes $(v_1, v_2, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$ for all i and $v_n = v_1$. The graph is called acyclic if it does not have any cycles. A* loop *is a sequence of nodes $(v_1, v_2, \ldots, v_n)$ such that $v_n = v_1$ and for any i, either $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$.*

Loops can be seen as cycles in the undirected version of the graph, i.e., when the pair (v, v') is treated the same as (v', v). According to Definition 2.4.1, any cycle is also a loop but in the sequel, we use the word 'loop' to refer to those that are not cycles. Moreover, an acyclic graph can still have loops. Most of the literature on BAGs is focused on acyclic graphs as defined next.

Plain BAGs

Definition 2.4.2 *An attack graph G is a directed graph $G = (E \cup C, R_r \cup R_i)$ where E is a set of exploits, C a set of conditions, and $R_r \subseteq C \times E$ and $R_i \subseteq E \times C$.*

Remark 1 *According to Definition 2.4.2, the attack graph is* bipartite*, i.e., the set of nodes of the graph is divided into two disjoint and independent sets E and C such that every edge can only connect a node from one to another. That is why the set of nodes is partitioned into a subset of $C \times E$ and a subset of $E \times C$.*

The edges connecting conditions to exploits have a particular meaning: all the conditions connected to an exploit must be satisfied in order to execute that exploit. This is the equivalent of taking the conjunction of the incoming conditions to the exploit. Similarly, any of the exploits connected to a condition can be used to satisfy that condition. This is equivalent to a disjunction between multiple exploits that satisfy the same condition. Such an interpretation together with individual scores assigned to the nodes fully characterises the attack model.

Definition 2.4.3 *Given an* acyclic *attack graph $G = (E \cup C, R_r \cup R_i)$, and an individual score assignment function $p : E \cup C \to [0,1]$, the cumulative score function $P : E \cup C \to [0,1]$ is defined as*

$$\begin{aligned} P(e) &= p(e) \cdot \Pi_{c \in R_r(e)} P(c) \\ P(c) &= p(c), \quad \text{if } R_i(c) = \emptyset \\ P(c) &= p(c) \cdot \oplus_{e \in R_i(c)} P(e), \quad \text{if } R_i(c) \neq \emptyset, \end{aligned} \tag{2.1}$$

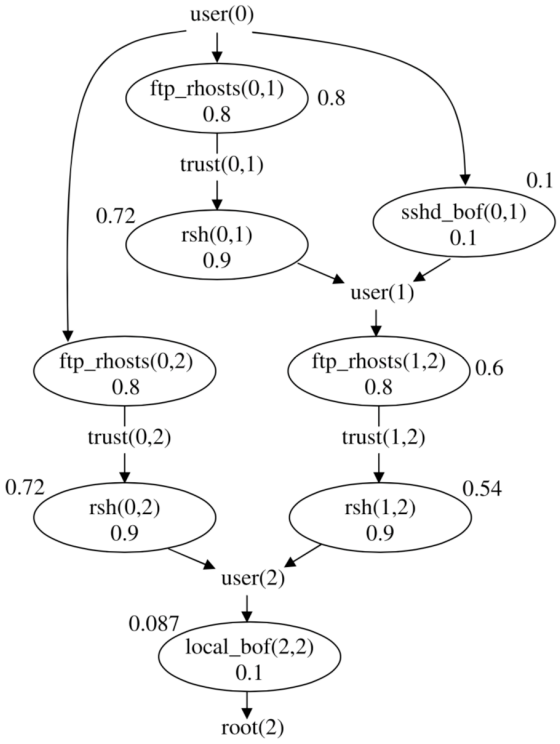


Fig. 2.3 Plain BAG example[114]

where $\oplus_{e \in R_i(c)} P(e)$ *is the probability of the union of exploits in* $R_i(c)$ *and is computed assuming the exploits are independent.*

The acyclic attack graph $G = (E \cup C, R_r \cup R_i)$ together with the individual scores defines a *plain BAG*. An example of using this formalism for a BAG is shown in Figure 2.3; here ovals are used to represent exploits, and the conditions are the plaintexts on the graph. The numerical values inside the ovals are the individual score assignments, whereas the values outside the ovals are the cumulative scores. An alternative example with a description of nodes but with no calculation can be found in [7].

Note that attack graphs can in general have cycles but plain BAGs are defined with acyclic attack graphs.

AND/OR BAGs

The second formalism of BAG is defined by Ou [85] and is used in MulVAL [86].

Definition 2.4.4 *A Bayesian attack graph is defined as a directed acyclic graph* $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ *where nodes* $\mathscr{V}$ *are connected by edges* $\mathscr{E}$*. The set of nodes is comprised of three types of nodes,* $\mathscr{V} = V_l \cup V_a \cup V_o$*, and edges are defined as* $e_{ij} \in E, e_{ij} = e(v_i, v_j)$ *where each edge e defines a mapping from node* v_i *to node* v_j*.*

The sets of nodes are defined as follows:

- V_l: the *leaves* in the graph, having no parents, and represent specific configurations and conditions in the network; this includes information about programs running on a host, network connection information in the form of HACLs (host access control lists), and the existence of vulnerabilities.

- V_a: the AND nodes, which have requisite conditions *all* of which have to be fulfilled in order to be accessed. In other words there is a conjunctive relationship between the parents of such a node. This set of nodes is used to represent specific actions that can be taken when the conditions are fulfilled; this can be something like movement between hosts when an attacker has fulfilled the prerequisite of access to one machine and there exists a configuration node for access between the two nodes, or could be the remote exploit of a specific vulnerability given remote access and the existence of the vulnerability as prerequisites.

- V_o: the OR nodes, which have requisite conditions of which *at least one* must be fulfilled in order to be accessed. There is a disjunction between the parents of such a node. These are specific micro-states in the network that define something about an attacker's position in the system, for example: the ability to execute arbitrary code on a specific host, or network access to a specific host. A macro-state for an attacker would be an enumeration of these nodes demonstrating the privilege they have with respect to every host on the network.

Vulnerabilities in the network have a chance to be exploited when their preconditions are fulfilled, and by exploiting a vulnerability an attacker achieves a specific state. This state, once reached, may then afford the attacker a privilege level on the network that is a

requirement for another exploit, or node. A chain of nodes in the network connected in this way represents an attack path or route.

We define the *access probability* $P(v)$ on the node v as the likelihood of the node being reached in an attack situation.

Definition 2.4.5 *For a given BAG* $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ *and a* local probability *function* $p : \mathscr{V} \to [0,1]$, *the access probability* $P : \mathscr{V} \to [0,1]$ *is defined recursively using the access probabilities of all parents to the node in conjunction with the* local probability *by*

$$P(v) = \begin{cases} p(v) & \text{if } v \in V_l \\ p(v) \prod_{v' \in pa(v)} P(v') & \text{if } v \in V_a \\ p(v)\big[1 - \prod_{v' \in pa(v)} (1 - P(v'))\big] & \text{if } v \in V_o \end{cases} \tag{2.2}$$

where $pa(v)$ *represents the parent set of the node* $v \in \mathscr{V}$, $pa(v) := \{v' \in \mathscr{V} | (v', v) \in \mathscr{E}\}$.

The access probability has a slightly different interpretation depending on the specifics of the node. For $v \in V_o$, $P(v)$ represents the probability that the attacker will achieve the state described by node v. For $v \in V_a$, $P(v)$ represents the probability that an attacker will travel along that specific route to reach the goal state that follows. For $v \in V_l$, $P(v)$ represents the probability of successful exploitation if the node defines a vulnerability, or is the probability that a specific entry-route will be used.

Remark 2 *Access probabilities* P *defined in* (2.2) *assume that probabilities* $P(v_{pa})$ *are independent from each other and takes the product of these probabilities to find the access probability for their child node. This assumption is only true if the graph of the BAG does not have loops. Otherwise,* $P(v)$ *in* (2.2) *will only be an approximation of true access probabilities that can be computed using joint distributions to reflect the dependencies between the related events. One of these exact methods is Variable Elimination discussed in Section 4.2.2.*

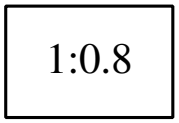


Fig. 2.4 LEAF probability

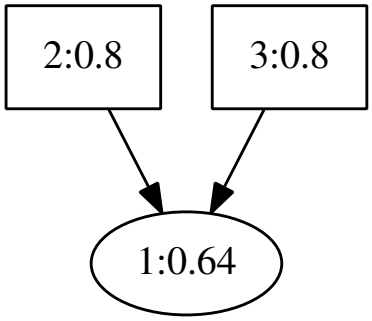


Fig. 2.5 AND probability

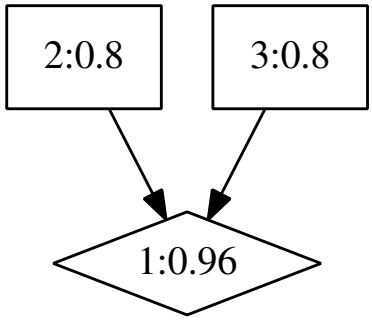


Fig. 2.6 OR probability

In Figures 2.4, 2.5 and 2.6 we demonstrate how this formalism works for simple examples. The LEAF node naturally is calculated by setting the access probability as equal to its local probability as it can have no dependencies. In Figure 2.4 this is simply 0.8.

AND nodes, drawn here as ellipses, are calculated using the above formula for conjunctive probability. In the case of figure 2.5 there are two LEAF node requirements for the node to be accessed, both with 0.8 as their probabilities. The access probability for this simple example is calculated as follows (note that we set $p(v_1) = 1$ for these examples, this is explained at the end of the section);

$$\begin{aligned} P(v_1) &= p(v_1) \prod_{pa} P(v_{pa}) \\ &= P(v_2) \times P(v_3) \\ &= p(v_2) \times p(v_3) = \qquad\qquad 0.8 \times 0.8 \\ &= 0.64 \end{aligned}$$

For OR nodes, drawn as diamonds on the graph, the disjunctive formula is used. This means that if either of the leaf nodes are *True* then the requirements are fulfilled and the node is

accessible. For figure 2.6 this is done practically as follows;

$$\begin{aligned} P(v_1) &= p(v_1)(1 - \prod_{pa} 1 - P(v_{pa})) \\ &= (1 - (1 - P(v_2)) \times (1 - P(v_3))) \\ &= 1 - ((1 - p(v_2) \times (1 - p(v_3)) \\ &= 1 - ((1 - 0.8) \times (1 - 0.8)) \\ &= 0.96 \end{aligned}$$

For clarity a more complex but realistic example is shown in Figure 2.7. This is describing multiple attack routes to be able to execute code on a server; nodes 7 and 8 describe the presence of a vulnerability on some other host on the network and a firewall rule that allows access through a specific port from the internet respectively. These in conjunction permit net access to the server represented by node 3. With net access and exploitation of one of the two vulnerabilities on the server at node 10, remote exploitation of the server's vulnerability can occur at node 2, allowing code execution on the machine at node 1. Alternatively the presence of a vulnerability that allows exploitation externally creates another path to achieving the execution of arbitrary code via node 4, enabled by attacker access and vulnerability existence at nodes 5 and 11.

For simplicity the local probability of non-LEAF nodes is set as 1; this makes sense as the AND nodes represent an assessment as to the likelihood of fulfilling all the requirements to perform an action and as such when these are fulfilled an attacker can access the node if they desire, and similar reasoning can be applied to OR nodes. These local probabilities could be set to values other than one to increase the fidelity of the model on an individual case basis, for example to allow for attacker skill in the model. With these local probabilities set, the access probability of node 1 can be decomposed into a calculation of LEAF probabilities:

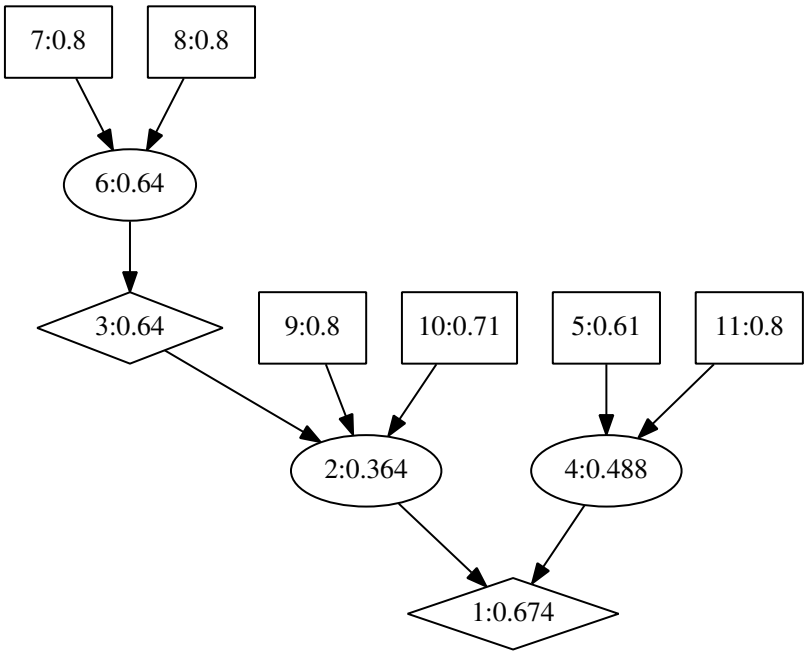


Fig. 2.7 A more complex BAG

$$
\begin{aligned}
P(v_1) &= (P(v_2) \cup P(v_4)) \\
&= (P(v_3) \cap P(v_9) \cap P(v_{10})) \cup P(v_4) \\
&= ((P(v_6)) \cap P(v_9) \cap P(v_{10})) \cup P(v_4) \\
&= ((P(v_7) \cap P(v_8)) \cap P(v_9) \cap P(v_{10})) \\
&\quad \cup P(v_4) \\
&= ((P(v_7) \cap P(v_8)) \cap P(v_9) \cap P(v_{10})) \\
&\quad \cup (P(v_5) \cap P(v_{11})) \\
&= ((0.8 \times 0.8) \times 0.8 \times 0.71) \cup (0.61 \times 0.8) \\
&= 1 - (1 - (0.8 \times 0.8) \times 0.8 \times 0.71) \times (1 - \\
&\quad (0.61 \times 0.8)) \\
&= 0.674
\end{aligned}
$$

2.4.2 Relation Between the Two Formalisms

In the following proposition, we show that the AND/OR definition of BAGs is more general than plain BAGs, as it abstracts away the type of nodes being exploits or conditions. Instead, it puts emphasis on their role in the computation of access probabilities.

Proposition 2.4.6 *Any plain BAG modelled as in Definitions 2.4.2-2.4.3 can be transformed into a BAG modelled as in Definitions 2.4.4-2.4.5.*

Proof 2.4.7 *Suppose we have a plain BAG with the acyclic attack graph* $G = (E \cup C, R_r \cup R_i)$. *Define the set of leaf nodes as* $V_l := \{c \in C \,|\, R_i(c) = \emptyset\}$, *the set of OR nodes* $V_o := C \backslash V_l$, *and the set of AND nodes* $V_a := E$. *Take* $\mathscr{V} = V_l \cup V_a \cup V_o$ *and* $\mathscr{E} = R_r \cup R_i$. *Then* $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ *is an attack graph satisfying all the requirements of Definition 2.4.4. Note that attack graphs of AND/OR BAGs are not necessarily bipartite, which makes them more general than plain BAGs.* ■

2.5 Computation of Access Probabilities

The main approach for computing access probabilities of all nodes is to translate the model into a Bayesian network (BN) and apply off-the-shelf techniques developed in the literature for BNs. We first provide the translation of the BAG into a BN in Section 2.5.1 and then discuss *variable elimination* as one of the techniques for performing probability computations over BNs in Section 4.2.2.

2.5.1 BAG Translation to a Bayesian Network

Definition 2.5.1 *A Bayesian network (BN) is a tuple* $\mathfrak{B} = (\mathscr{V}, \mathscr{E}, \mathscr{T})$. *The pair* $(\mathscr{V}, \mathscr{E})$ *is a directed acyclic graph representing the structure of the network. The nodes in* $\mathscr{V}$ *are (discrete or continuous) random variables and the arcs in* $\mathscr{E}$ *represent the dependence relationships among the random variables. The set* $\mathscr{T}$ *contains conditional probability distributions (CPD) in forms of tables or density functions for discrete and continuous random variables, respectively.*

In a BN, knowledge is represented in two ways: qualitatively, as dependencies between variables by means of a directed acyclic graph; and quantitatively, as conditional probability distributions attached to the dependence relationships. Each random variable $v_i \in \mathscr{V}$ is associated with a conditional probability distribution $\text{Prob}(v_i|pa(v_i))$.

Proposition 2.5.2 *Any BAG $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ as in Definition 2.4.4 with local probability function $p : \mathscr{V} \to [0,1]$ in Definition 2.4.5 can be translated into a BN $\mathfrak{B} = (\mathscr{V}, \mathscr{E}, \mathscr{T})$. The random variables in $\mathscr{V}$ are all Boolean and the probability tables in $\mathscr{T}$ are constructed as follows. For all $v \in V_l$,*

$$Prob(v = 1) = p(v) \quad \text{and} \quad Prob(v = 0) = 1 - p(v). \tag{2.3}$$

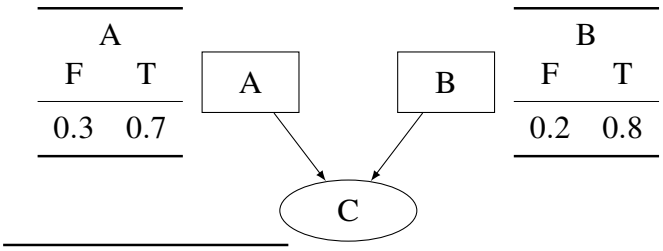
For all $v \in V_a$, let $pa(v) = \mathbf{1}$ indicate that all variables in $pa(v)$ take value equal to one. Then,

$$\begin{cases} Prob(v = 1|pa(v) = \mathbf{1}) = p(v), \\ Prob(v = 1|pa(v) \neq \mathbf{1}) = 0, \\ Prob(v = 0|pa(v) = \mathbf{1}) = 1 - p(v), \\ Prob(v = 0|pa(v) \neq \mathbf{1}) = 1. \end{cases} \tag{2.4}$$

For all $v \in V_o$, let $pa(v) = \mathbf{0}$ indicate that all variables in $pa(v)$ take value equal to zero. Then,

$$\begin{cases} Prob(v = 1|pa(v) = \mathbf{0}) = 0, \\ Prob(v = 0|pa(v) = \mathbf{0}) = 1 \\ Prob(v = 1|pa(v) \neq \mathbf{0}) = p(v), \\ Prob(v = 0|pa(v) \neq \mathbf{0}) = 1 - p(v). \end{cases} \tag{2.5}$$

Then if the BAG $\mathscr{G}$ does not have any loops, we get $P(v) = Prob(v = 1)$ for all $v \in \mathscr{V}$ with access probabilities P defined in (2.2).



C			
A	B	F	T
F	F	1.0	0.0
F	T	1.0	0.0
T	F	1.0	0.0
T	T	0.4	0.6

Fig. 2.8 A simple BAG with the associated probability tables constructed according to Proposition 2.5.2. Local probabilities are $p(A) = 0.7$, $p(B) = 0.8$, and $p(C) = 0.6$.

Figure 2.8 illustrates the construction of probability tables for an AND node. In this figure, the local probabilities are $p(A) = 0.7, p(B) = 0.8$, and $p(C) = 0.6$. The probability tables for A and B are constructed according to (2.3) and for C according to (2.4).

2.5.2 Visualisation

Several attempts have been made to use attack graphs to visualise network security. Xie et al. [119] present a visualisation that results in a single greyscale image. First, they decompose the attack graph into two layers, one representing the network connectivity between hosts and the other detailing all the possible attack scenarios. With these two layers created, they calculate the chances of the success for each attack, and from this the reachability of each host from each of the other hosts can be calculated. An adjacency matrix is then produced to visualise the accessibility of the hosts on the network.

Homer et al. [38] present several techniques for reducing attack graphs to improve visibility. Extraneous portions of the graph that do not contribute to understanding are trimmed and similar attack sequences and structures are identified and grouped together in order to reduce clutter. Lee et al. [63] develop classes of attack that improve the readability

Table 2.5 Complexity scores and their local probabilities.

Vector Score	CVSS Version	Local Probability
Low/L	2,3	0.71
Medium/M	2	0.61
Unknown	-	0.61
High/H	2,3	0.35

of attack graphs for machines as an extension of the multiple prerequisite attack graphs by Ingols, Lippmann and Piwowarski [45].

Chu et al. develop a tool for attack graph visualisation [19] called NAVIGATOR. This can be used to investigate a variety of different attacks and the effects they have on networks. It can also be used for the planning of network structures and defence mechanisms, and as a way of displaying network topology.

2.5.3 Prior Generation

In order to calculate the access probabilities for nodes in a BAG, an initial set of local probabilities must be provided. These local prior probabilities should be as accurate as possible in order to improve the accuracy of the probabilities across the graph. For testing and demonstration purposes we use a simple technique where we determine the ease of access for a vulnerability. The CVSS vector [108, 73] for the vulnerability is collected from NIST's National Vulnerability Database (NVD) and the Access Complexity (CVSSv2) or the Attack Complexity (CVSSv3) is used to define the probability of transitioning to a state. This is on a scale of Low, Medium and High for version 2, and Low and High for version 3, with High meaning there is a great deal of skill or timing required to exploit the vulnerability and as such is associated with the lowest probability scoring. A demonstration of how these values could inform the local probabilities is shown in Table 2.5. The local probability values shown are based on expert opinion of the likelihood that an attacker would successfully exploit a vulnerability with the corresponding complexity score and are given as examples. These values could be modified to model an expert attacker using higher probabilities, or an attacker with little expertise making use of published exploits.

The local probabilities are taken from the contribution that the NVD gives to a vector score when calculating the whole CVSS score. While this is a useful approximation, it is very abstract and ignores a great deal of the information that can be gleaned from the information available about the vulnerabilities. Doynikova and Kotenko [22] demonstrate a more complex method for achieving more accurate results from CVSS data than simply using the complexity score, while also modelling attacks that do not rely on vulnerability exploitation through the use of the CAPEC list (Common Attack Pattern Enumeration and Classification), a taxonomy of different attack patterns described using the MITRE schema [8]. Cheng et al. model dependency relationships of the base metrics in the vectors and attempt to combine them in such a way that a user can weigh specific aspects for their local probability assignment [16].

Scoring with respect to time has also been demonstrated, for example Kaluarachchi et al. propose use of the 'Risk Factor Model' [95] as a way of improving the accuracy of their transition probabilities; rather than relying on CVSS scores alone they use an equation involving the age of the vulnerability to predict the probability of exploitation. Bozorgi et al. [11] also incorporate time into their model by training classifiers to predict if and when an exploit will occur for an individual vulnerability. Machine learning is used instead of the 'magic numbers' in CVSS. They use the Open Source Vulnerability Database (OSVDB) and the MITRE Common Vulnerabilities and Exposures (CVE) database, from which they extract high-dimensional feature vectors to be used in support vector machines (SVM). With an offline analysis they showed that the vulnerability reports contained useful information as to whether a vulnerability would be exploited, with an accuracy of around 90%. From this they show the highest weighted features. They suggest using their classifier score as a replacement for the exploit-ability metric in CVSS as it is grounded in a statistical model rather than opinion.

2.5.4 Probability Propagation and Cycles

Once the prior probabilities have been assigned, they must be propagated across the graph in order to calculate the access probabilities for every node, through the use of a Bayesian

network or a Markovian process. Bayesian network based network analysis has been widely studied in the context of attack graphs (e.g. [27, 66, 91, 102, 120]). In particular, Choi et al. propose a method for deleting network edges from a Bayesian network in order to generate models through approximate inference [17]. Muñoz-González et al. present an exact method for inference in BAGs using the junction tree algorithm [78]. This method is attractive due to its exact nature, but unfortunately is very limited in its application due to how it scales. This is caused by the requirement for tables to be generated based on the cliques created to start the calculations, and for large graphs these tables can become extremely large. It is better to have a trade-off in the accuracy of the method to reduce the space required, in order to allow scalability for the large graphs that are expected from enterprise networks. Furthermore, this technique cannot be applied to graphs with cycles, meaning that in order to use it a graph may first need to be altered. The authors go on to present an approximate technique in [77] using loopy belief propagation. The results of this scale well, linearly with respect to the number of nodes, while achieving a reasonable level of accuracy. The drawback to using this method, unlike stochastic simulation, is that there is no guarantee of convergence to the correct value.

A number of Markovian approaches have been taken to generate Bayesian attack graphs and facilitate vulnerability analysis and the design of optimal defence strategies. Jha et al. use a model checker to automatically generate attack graphs annotated with probabilities and analyse their vulnerabilities using Markov Decision Process (MDP) algorithms [50]. Mace et al. used a similar approach to find the optimal data collection strategies for accurate Bayesian attack graph input parameters (e.g. conditional probability tables) [68]. In [90] Pièvre-Cambacédès et al. model attack trees as Boolean logic Driven Markov Processes (BDMP), suggesting they are dynamic and inherit readability and appropriation of attack trees but with mathematical properties reducing combinatorial problems and processing. Continuous Time Markov chains have been applied by Jhawar et al. to the Bayesian attack graph approach in order to analyse attack defence graphs, that is, attack graphs which define the modelling of defences in their specification. Wang et al. in [114], estimate attack states and define a cost-benefit heuristic to automatically infer optimal defences for attack graphs

integrated with Hidden Markov Models while Miehling et al. [75] and Zhisheng et al. [42] assume the defender can only partially observe the attacker's capabilities at any given time, thereby modelling Bayesian Attack Graphs as partially observable Markov decision processes (POMDPs). In this sense a resulting defence strategy is both reactive and anticipatory.

Dealing with cycles in Bayesian attack graphs has also been tackled (e.g. [121, 51]). Kłopotek et al. who use Markov Chains, suggesting the state of a variable is not influenced by itself but rather the future state is influenced by the past one [57]. Doynikova and Kotenko consider the processing of three simple types of cycle in Bayesian attack graphs [22]. Two of the three types contain cycles between nodes at the same level in the graph structure. These are either removed once the probabilities to reach each node for the first time have been calculated, or enumerated as separate paths. The third type contains cycles between nodes at different levels of the graph structure. In this case, the cycle is simply removed under the backtracking assumption, that is an attacker does not come back to a node already exploited. This assumption that the attacker will not backtrack, also referred to as the *monotonicity principle* is widely used in the literature as a way of removing cycles from graphs. While the assumption can be used well for simple cycles, more complex cycles require a different approach due to all nodes affecting the probabilities in the cycle (this is shown in Section 4.4.2). Furthermore, modifying the graph in order to perform calculations can introduce other problems, for example re-computation of probabilities after a minor change to network structure will be invalid as possible paths for the attacker may have been trimmed. Other work also exists that attempts to remove this assumption, for example in [2], Aguessy et al. present a Bayesian network-based extension to attack graphs, called a Bayesian Attack Model (BAM), which is capable of handling cycles by breaking them. The authors argue that using the backtracking assumption to break cycles suppresses possible attacker actions which cannot be known a priori. To keep all possible paths, the only way to break cycles is to enumerate all paths starting from every possible attack source. In other words, unfolding the cyclic graph structure to an equivalent acyclic graph structure such that each node appears exactly once in each path. This process causes a combinatorial explosion in the number of nodes whilst the inference algorithm is shown to remain efficient only for networks of up to 70 hosts. Homer

et al. suggest their approach correctly handles both cycles and shared dependencies in attack graphs, that is the probabilities along multiple paths leading to a node are dependent on each other [39]. The authors suggest enumerating all paths is unnecessary if data flow analysis is applied to the cyclic nodes enabling the same probabilities to be evaluated as on the unfolded graph. The data flow analysis process uses dynamic programming and other optimizations to avoid increasing computational complexity. The algorithm is limited by the number of nodes and paths within cycles which must be considered when calculating probability values and which can cause evaluation time to be exponential in the worst case. The evaluation is based on the number of nodes and vulnerabilities per node and does not consider how the number of cycles impacts computation. Wang et al. made a number of crucial observations about cyclic attack graphs and proposed a customized probabilistic reasoning method that can handle cycles in the calculation [114]. However, when combining probabilities from multiple attack paths, the method uses a formula that assumes the multiple probabilities are independent. Such dependency needs to be accounted for to prevent a distortion of results.

There are some examples of stochastic simulation techniques for attack graphs as well. One is by Noel and Jajodia [84]; they use probabilistic logic sampling to compare different security fixes for a network. However, this is performed by hand and as such it cannot be generally applied. Their use case compares several security controls that could be added to the network. This is achieved by examining the resulting distributions estimated when the changes are applied to the graph, in a manner similar to the sensitivity analysis we present in Chapter 5. This requires excessive computation and also requires analysis of the resulting distributions by hand. Baiardi and Sgandurra use Monte Carlo simulations in their Haruspex tool [6]. This tool is a fully featured program that uses attack graphs and threat agents to model security. It is an application for this type of graph, incorporating many different elements, but does not analyse different methods for simulation.

2.6 A Note on Terminology

Throughout the literature, two terms are frequently used to denote differences in analysis techniques for attack graphs: statics and dynamic. Static analysis denotes the propagation of probabilities through the attack graph in a one-off calculation that does not include evidence and does not have any updating of beliefs [54, 67, 78]. This is analogous to the method we introduce in Chapter 4, and we refer to the process as the calculation of *access probabilities*. Dynamic analysis refers to the updating of probabilities with new beliefs, for example incorporating evidence into the attack graph [90, 78, 91]. This dynamic approach is the equivalent in Bayesian inference of forward and backward propagation resulting in new *posterior* probabilities. We refer to this process as *inference*, and show sampling-based techniques for probability propagation and inference in Chapter 5.

Chapter 3

Classifying Vulnerabilities with a Neural Network for Simplified Remediation

This chapter is on the classification of vulnerabilities using machine learning and its application. First, we discuss the available data and problems with the data and vulnerability reporting in general that we aim to solve. Specifically, we aim to reduce the volume of the output of a vulnerability scan by grouping all vulnerabilities into families that have the same remediation process, to prevent the administrator from being overwhelmed by unnecessarily complex lists of specific vulnerabilities. This enables an efficient reporting process where once the important vulnerabilities have been identified, a task we tackle in later chapters, the vulnerabilities can be reported in a straightforward manner in terms of their remediation steps. We also address all of the common criticisms of the National Vulnerability Database (NVD) dataset with our approach, which are described in the related work. We then look at our machine learning pipeline using a neural network (NN) as a classifier and explain how this can be used as one of the tools to simplify and understand complex vulnerability scans, in fulfilment of Contribution 2 in the Abstract. We demonstrate the training and use of the model, showing that it can achieve an accuracy of over 99% making it

a good candidate for automating the process of labelling vulnerabilities. Next, we discuss a filtering technique that we show can be used to improve the accuracy of the process even further. We go on to examine the related work in this area both in terms of using machine learning for vulnerability data and relevant machine learning literature for text-based classification problems. Finally, we discuss the usage of the classifier as well as other work that remains to be done both to improve the classifier and to apply machine learning techniques to vulnerability data for other purposes.

3.1 Problem Definition

The aim of this chapter is to show a method for simplifying the patching process for users of vulnerability scanners, and to address the problems with the NVD dataset. Currently, after a vulnerability scan is run on a network, the user receives a long report in the form of a list of vulnerabilities present on the network. Even for networks with just a few hosts present this list can be very long, and larger networks will report thousands of items. The length of the list makes reasoning and prioritising actions very hard, and there is no meaningful grouping of the vulnerabilities. The report will also only inform the user of the vulnerability present rather than what is needed for remediation. The dataset used to populate these lists is also criticised for having some known problems (see Section 3.4 for further discussion). These problems are: the lack of good documentation for vulnerabilities, incorrect chronology, multiple entries for one vulnerability, and missing entries. We describe and demonstrate a machine learning approach to vulnerability labelling that addresses the problems of the dataset and improves the reporting of vulnerabilities to users by instead reporting the required action to fix the vulnerability.

3.2 Building a Neural Network to Classify Vulnerabilities

The aim of our classifier is to group vulnerabilities into families such that the vulnerabilities within the family share a common remediation process. The reason for this is that with a large network with a variety of different technologies and active software, the results of a vulnerability scan will be a large list of complex identifiers and vulnerability names. This list can be simplified into actions required to fix the vulnerability, and if several vulnerabilities share a method of fixing, the list will become much more concise as well as being simpler to read and implement. This process can also be applied just to improve readability making it applicable to small networks as well. These networks may also not have a dedicated system administrator, or have one that is not confident with the vulnerability landscape. In this situation it is much more valuable to instruct the reader that a certain software needs to have a patch downloaded rather than list many complicated exposures that exist on their system.

The many problems with the dataset, discussed in the previous section, can also be overlooked for this application. In fact, sorting the vulnerabilities in the way we propose acts as a fix for some of the problems and a mitigation of the others. The problem of chronology will be removed by ignoring the temporal component of individual vulnerabilities altogether, and the lack of documentation is remedied by grouping many vulnerabilities into classes that can be explained in terms of their remediation. The problem of multiple inclusion is also removed as a vulnerability included multiple times should be moved into the same group, removing the duplication. Finally, the incomplete inclusion problem is also partially rectified, as long as a vulnerability from the same class as an excluded vulnerability is found by the scanner then the reported vulnerability class will be the same. This is not unlikely as many of the remediation processes are simply to apply patches to software, and so if any exposure is found in the software the remediation will fix all vulnerabilities of that class.

The methodology for this process will be supervised learning with a NN architecture. Supervised learning is a common way of applying machine learning principles to make the process of putting items into categories automatic. It requires a labelled data-set that is split with most being used to train the model and the rest used to test it. The training data is input into the model, after feature extraction, and the model makes a prediction as to which

category the input belongs to. This is compared with the ground-truth of the label for the input data, and the model is adjusted accordingly. Once trained, the model can be evaluated using the testing data not seen by the model in the training phase, but that is also labelled so the model's prediction can be compared with the truth and the accuracy of the model can be calculated.

Our choice of using a NN as the predictive model for the process is due to the demonstration of impressive performance that requires only simple models on text classification by Nam et al. and Jo et al. [79, 52]. Nam et al. compare a single layered NN against several more complex state-of-the-art techniques (*Backpropagation for Multi-Label Learning* and *Binary Relevance* using various architectures) on six different datasets and found the best performance with their single layer NN. Gawron et el. and Huang et al. [29, 43] also demonstrate their suitability to the NVD data, and so we use this model for the process.

3.2.1 Data Structures and Labelling

The data used in this process is downloaded from the NVD [80]. A large portion of the data is not helpful for sorting the vulnerabilities into common families and so is removed from the data frame. For example, categorical data and numerical data do not allow for the deduction of the identity of the software or component that is vulnerable or any information on remediation and so are removed from the data-set. The remaining columns of data are: "Name", "Family", "Dependencies", "Description" and "Affected".

The data is labelled according to XQ's vulnerability family system: 130,000 vulnerability entries are labelled with one of 286 family identities. The NVD data is labelled by hand by XQ's security experts. The family identity values are largely comprised of patching pieces of software and operating systems, with 199 of the labels involving patching and 87 not. The three most common labels are "Fedora Linux Patching", "AIX Patching", and "Debian Linux Patching", with 26,176 entries, 11,388 entries and 10,277 entries respectively. Examples of vulnerability families that do not involve patching are "Malware/Backdoor Detected", "Weak Encryption Ciphers Permitted" and "Users With Unchanged Passwords". These labels allow for a much simpler interpretation of vulnerabilities when they are reported, for example if

Table 3.1 Examples of vulnerabilities and their families.

Identity	Vulnerability	Family Label
CVE-2014-2441	Oracle VM VirtualBox Graphics DriverWDDM	→ Oracle Virtualbox Patching
CVE-2016-3612	Oracle Virtualbox Information Disclosure Vulnerability-010716	→ Oracle Virtualbox Patching
-	Greenbone Security Manager (GSM) / Greenbone OS (GOS) Detection (HTTP)	→ Web Server Detected
CVE-2012-4964	Samsung Printer SNMP Hard-coded Community String Authentication Bypass Vulnerability	→ Insecure SNMP Configuration

your Virtualbox application is out of date then it becomes vulnerable to several vulnerabilities. Instead of a list of the relevant vulnerabilities being reported, the user is informed that there is a patch required for their Virtualbox if the vulnerabilities have been sorted into families.

Table 3.1 shows the labelling of some vulnerabilities along with the original CVE and name entries. A full list of the vulnerability families used in the results of this chapter can be found in Appendix C

3.2.2 Preprocessing

The labelled data first needs to be processed before it can be used as the input for a NN, as shown in Figure 3.1. This is because the data we are using is comprised of textual descriptions and terms, but the model requires numerical vectors as input. The first step to preprocessing the data is to *tokenize* the sentences we have in the data columns remaining after we remove all the unnecessary data. This splits the data into specific words, removing the punctuation. After this, for every vulnerability in our data we have an associated list of words. For the labels in the data, the unique entries for the labels are arranged in a list that is then associated to a vector. We have a set of labels, L, that each element of our data must be associated with. The label vector of a specific vulnerability entry, $\mathbf{y}$, is then a vector with L dimensions: $\mathbf{y} \in \{0,1\}^L$ where $\sum_i \mathbf{y}_i \equiv 1$. As such, where $y_i = 1$ the vulnerability belongs to the class L_i. (Our objective, then, is finding a model m to act on a vector input $\mathbf{x}$ built from the textual data

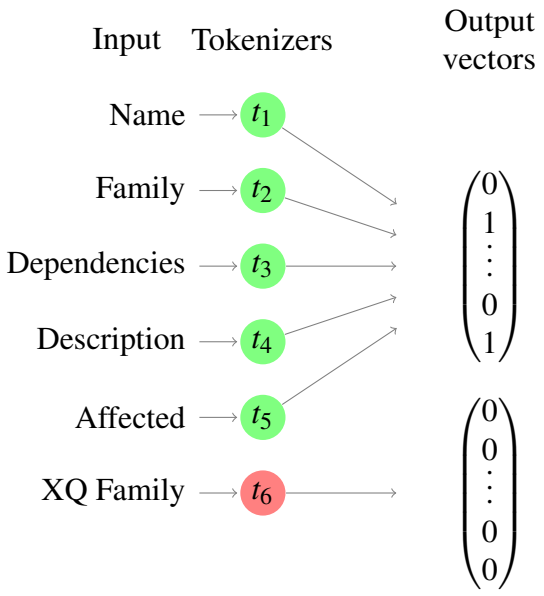


Fig. 3.1 Vulnerability data processed by tokenizers to create a vector that can be used in the NN, including the labelled family.

in a vulnerability entry to achieve $m : \mathbf{x} \rightarrow \mathbf{y}$.) This method for labelling the vulnerabilities using a single positive class that is high, with all other classes being low, is called one-hot encoding [37]. We use this encoding as the data is categorical but the NN requires numerical data as input. If a simple integer encoding were to be used (*Class A* is labelled 1, *Class B* is labelled 2 etc.) then the machine learning algorithm might infer that the categories are related, and that *Class B* is worth double what *Class A* is worth [13]. As such we use one-hot encoding to ensure that all the classes have the same numerical value.

After tokenizing we must *stem* the words that make up our data. Stemming is a process performed by a stemming algorithm that reduces a derived, inflected or otherwise modified word into its root form. The reasoning for this is that the same term used in different contexts should count towards one word count, rather than having a word count for each form of the word. An example would be if we were counting the number of times a description contains the word *exploit*. A stemmer would change any occurrences of *exploited*, *exploiting*, *exploits*, and any other variations back into the common root '*exploit*' ensuring they all contribute towards the same total. The algorithm we use for the stemming process is the Lancaster stemmer, due to its speed for a large dataset as well as the fact that it is a very aggressive

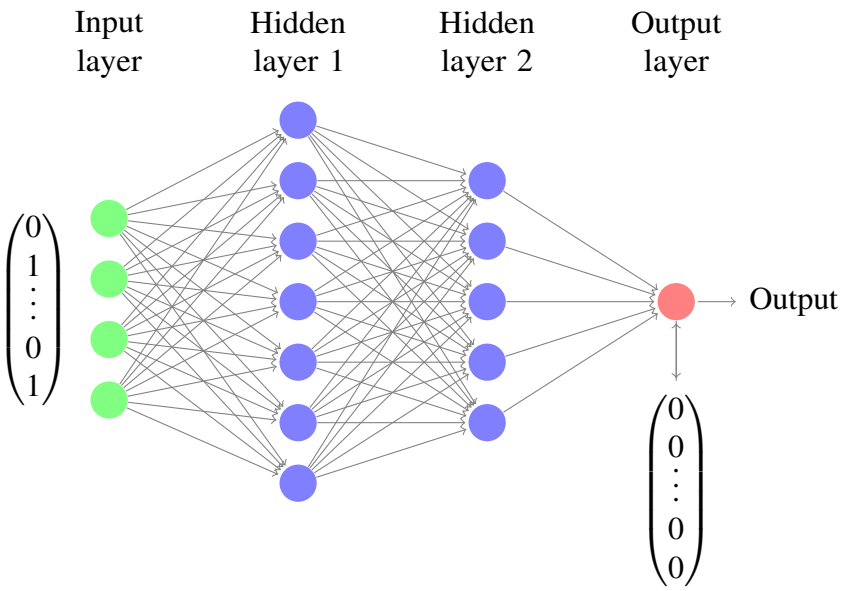


Fig. 3.2 Architecture of the NN.

stemmer that will significantly reduce the set of words used, a property which is desirable for the very sparse data of vulnerability descriptions and names.

In order to further reduce the dimensionality of our inputs, any terms that are in 90% or more of the documents are not included in our final dictionary, as they will not contribute much or at all to distinguishing between entries. This is similar to a *term frequency-inverse document frequency* (TF-IDF) approach where the frequency of a term is multiplied by the inverse of how many times the term appears in all documents combined (Christian et al. demonstrate the process in [18]). Our simpler process was originally meant to be replaced by TF-IDF as Huang et al. [43] show it works well with NVD data, but using this simpler technique produced very impressive accuracy and also remains very easy to interpret. With a final dictionary, or *bag-of-words* (BoW), a database entry can be encoded as a vector using the same tokenizers and stemmers, then occurrences of words that are present in the overall BoW are identified and the corresponding entry in the vector is set as 1. This process is shown in Figure 3.1.

3.2.3 NN Architecture and Training

In order to have as much speed and scalability as possible, it is desirable to have fewer layers in a NN. However, the fewer layers that are present the harder it becomes to model complex relationships. Text-based preprocessing already has an amount of higher dimensionality features [53] and so the network does not need to be as deep, and Nam et al. [79] perform better than state-of-the-art deep learning approaches with just a single layer for large multi-class text classification problems across six different sets of data. However, due to the simplicity of our preprocessing, we will be using two layers.

The network structure is shown in Figure 3.2. The input layer is comprised of the BoW representation of a vulnerability, a vector with a 1 at every point where the vulnerability contains a word in the BoW (a one-hot encoding of the vulnerability text). This vector feeds forwards into the next layer, hidden layer 1, that is originally set to have a randomly sampled weight with mean 0, along with hidden layer 2. These fully connected layers take the previous layer as well as the weightings of their neurons to calculate the layer's activation. The output vector is then compared with the ground-truth of the label vector and an error is calculated. We measure the loss using the softmax cross-entropy loss function (for definitions for both the softmax function and cross-entropy loss refer to Bishop [10] Chapters 2 and 4 respectively, or see Section 3.3.1 for the loss) and perform stochastic gradient descent using the Adam (adaptive moment estimation) [55] optimizer to tune the weights of the hidden layers. For a full description of the workings of NNs, along with a mathematical definition, we refer to Goodfellow and Benigo [33].

In order to prevent overfitting we employ the regularization technique called *dropout* [105]. Overfitting occurs when a model becomes too closely tuned to the training data-set such that its predictions incorporate the noise of the data rather than modelling the relationships in the data. When the model is presented with unseen data it may have a higher error rate due to the new data having different noise. Dropout is the process of taking only a subset of all the units for a training step. In other words, given a dropout probability, for each training step a random sample from 0 to 1 is taken for every unit and if the sample is lower than the dropout probability that unit is omitted from the training step. This means that

each unit of the network will see a different subset of the training data. The speed of training is also increased as a direct result of using dropout, as each step will require the training of fewer units.

Hyperparameters are values used in the model that are not model parameters (parameters that are modified by the training process). The dropout probability is an example of a hyperparameter. In order to select the values for the hyperparameters in the model, as well as compare different regularization techniques to choose the most effective, we use a process called randomised grid search cross-validation [9]. A grid of values for each hyperparameter is constructed and the performance of the model with random selections of those values is evaluated using cross-validation. Using this process we compare values for dropout probability alongside common regularization techniques L1 [111] and L2 [82] regularization. L1 and L2 both add a term to the cost function of the model, with the difference between the two methods being how the extra term is calculated. For this model, the dropout process was the only method that had a positive effect on the cross-validated error, so L1 and L2 were not used in the final model.

Imbalanced data. The data we are using has a very high variation in terms of the representation of classes. Many classes have thousands of samples, but there are some classes with 50 or fewer. In order to discourage the network from only making predictions from the highly represented classes we resample from points in the minority. The points are chosen randomly, and this process is known as *random oversampling*. We oversample the minority classes rather than undersampling the larger ones as recommended by Japkowicz and Shaju [49], who discover that random oversampling is much more useful. This could be extended further by creating synthetic data as part of a more complex oversampling algorithm, like SMOTE [15], but we use simple random oversampling as it has been shown to be effective and robust [65].

Finally we also use *early stopping* as another form of regularization [92]. During training, at regular intervals, the model is evaluated by running it on the validation set and measuring the mean error. If this mean error is higher than the mean error the previous time the model

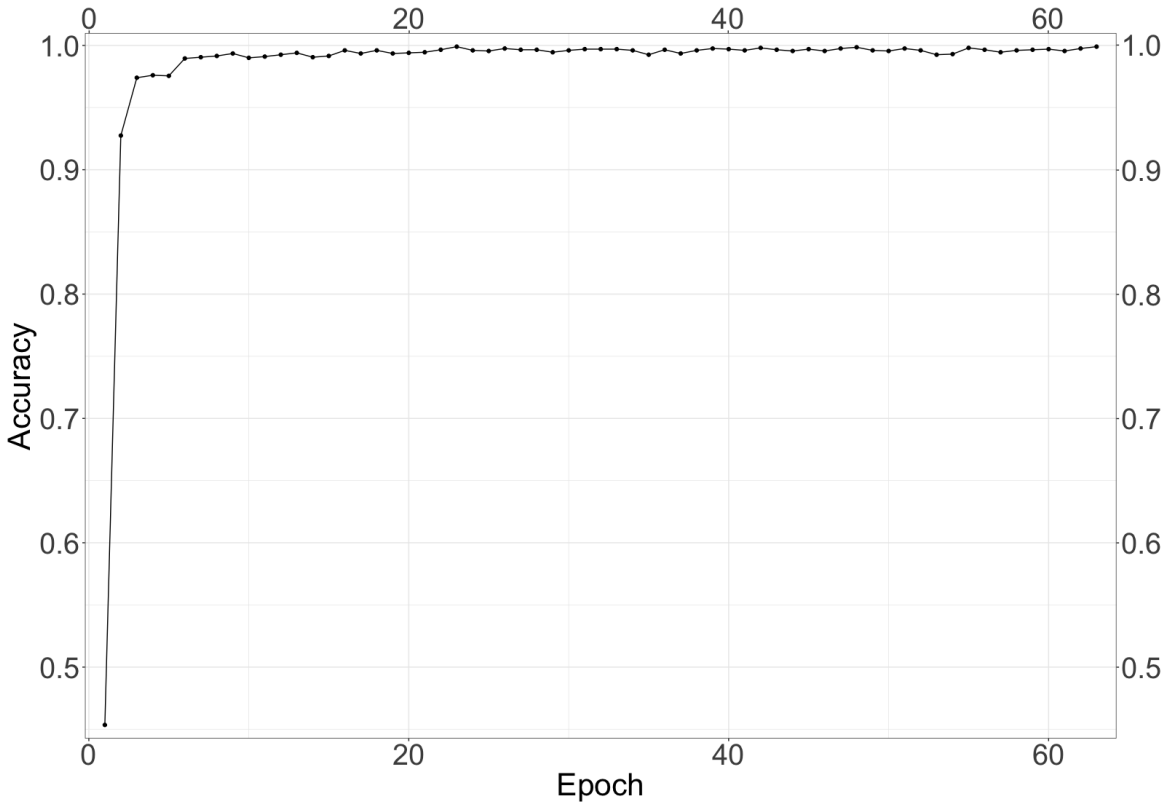


Fig. 3.3 Accuracy of the model while training.

was evaluated, training should be stopped and the earlier model should be used. This reduces the chance of overfitting.

3.3 Performance and Results

3.3.1 Initial Testing

In order to demonstrate the practicality and performance of our solution, we train the model in the process described in the previous section. Training the network on an Apple MacBook using the CPU takes 63 seconds, when using a split of 90% of the data for training and the remaining 10% for validation, and so using a more powerful machine, especially one with a GPU, would make the network very quick to train. After training, we achieve a prediction

Table 3.2 Precision, recall and F-1 scores for the model as global metrics and as per-class averages.

Metric	Global	Class average
Precision	0.9967	0.9914
Recall	0.9967	0.9979
F-1	0.9967	0.9935

accuracy of 99.90%, with a loss of 9.883×10^{-6}. The accuracy is defined as

$$ACC = \frac{TP+TN}{TP+TN+FP+FN} \tag{3.1}$$

with TP and TN being true positives and true negatives, and FP and FN being false positives and false negatives respectively. The accuracy of the model is evaluated against the validation dataset that the model has not seen before, and quickly reaches around 96% then slowly trends upwards to 99.90% as seen in Figure 3.3. This accuracy, however, only corresponds to a binary evaluation of the model in that it ignores the certainty of the model and instead simply checks if the highest probability the model gives as output is in the correct class. Ideally, we also want the model to give a very high probability when selecting the correct class, or on the few occasions that an incorrect class is chosen a very low certainty should be given. To analyse this we look at the loss of the model.

The loss we use is the categorical cross-entropy loss (CCE), calculated with

$$CCE = -log\left(\frac{e^{s_p}}{\sum_j^C e^{s_j}}\right) \tag{3.2}$$

where C is the set of classes in the data, s_p is the score assigned to the positive class by the network and s_j is the score assigned to the j^{th} class. Using this, we can evaluate how certain the model is in selecting the correct classes. Figure 3.4 shows the loss reducing as the model is trained.

As the model is a classification model, we also examine the *precision*, *recall* and *F-1* scores of the model, shown in Table 3.2. The precision, or positive predictive value (PPV), is

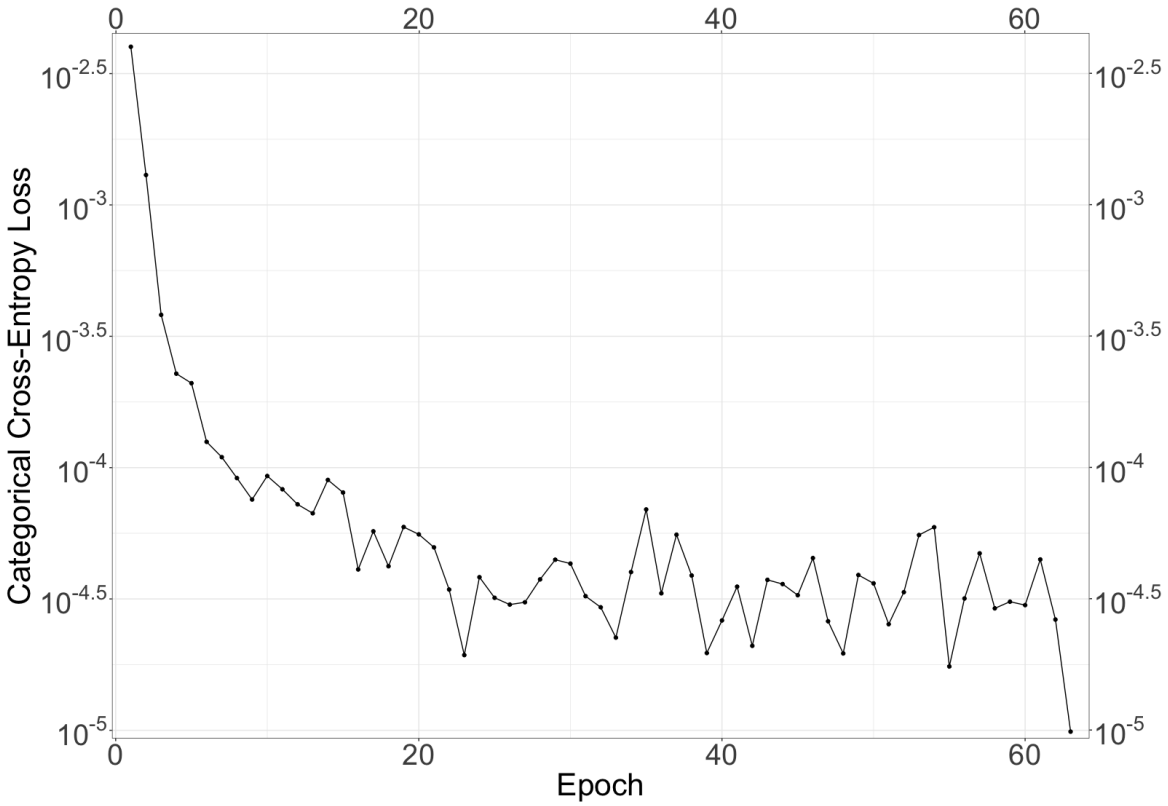


Fig. 3.4 Categorical cross-entropy loss of the model while training.

calculated as

$$PPV = \frac{TP}{TP+FP} \tag{3.3}$$

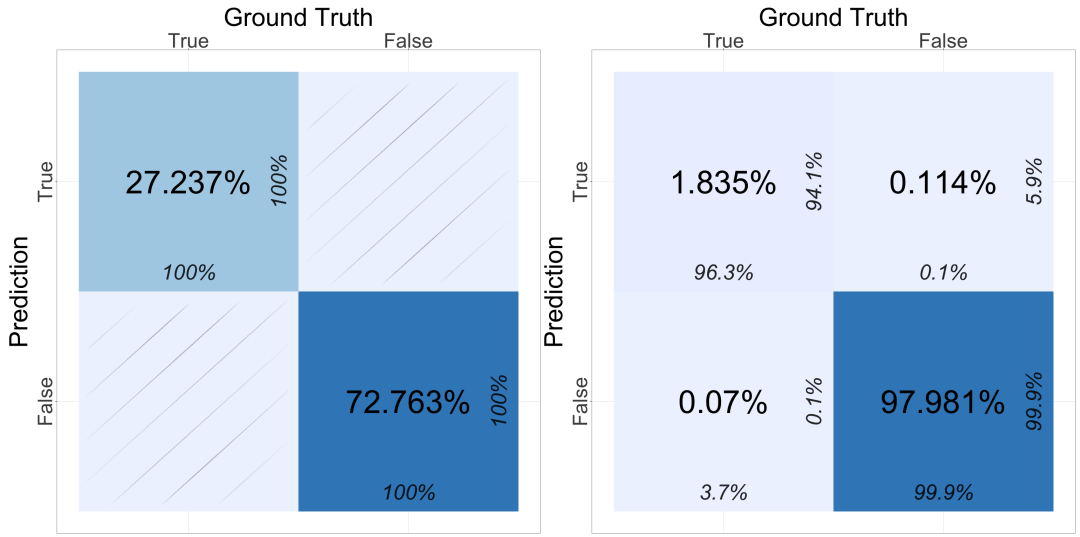
and is a measure of the ability of the model to not incorrectly label a negative sample as positive. Recall, or the true positive rate (TPR), is calculated as

$$TPR = \frac{TP}{TP+FN} \tag{3.4}$$

and can be understood as the amount of positives the model will correctly identify from the total amount of positives. Finally, the F-1 score is calculated as

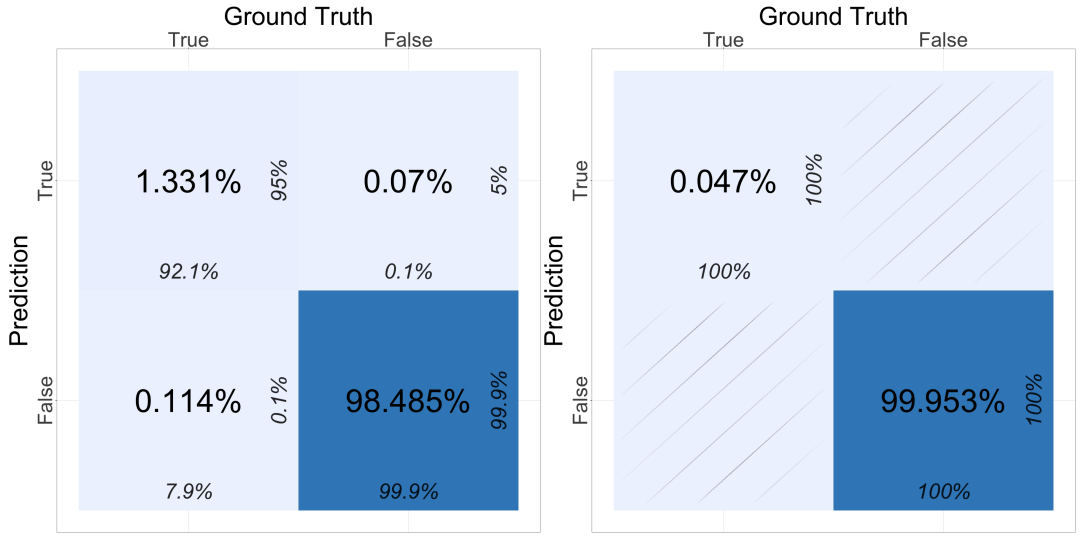
$$F_1 = 2 \times \frac{PPV \times TPR}{PPV + TPR} \tag{3.5}$$

which is the same as the harmonic mean of the precision and the recall of the model. For this multi-class classification problem we calculate the scores in two ways: globally where the overall true positives, false positives and false negatives for the entirety of the data are used, and a per-class average, where each metric (precision, recall, F-1) is calculated for each class and the values are averaged.



(a) Class with the most instances (highest TP)

(b) Class other classes are most frequently incorrectly assigned to (highest FP)



(c) Class most frequently incorrectly assigned to a different class (highest FN)

(d) Class with the fewest instances (highest TN)

Fig. 3.5 Normalised confusion matrices for the four classes with highest values for True Positive *(a)*, False Positive *(b)*, False Negative *(c)* and True Negative *(d)*

The normalised confusion matrices for four of the classes can be seen in Figure 3.5. A confusion matrix is a graphical representation of the model's ability to correctly assign a class to a piece of new data. On the diagonal of the matrix are the correct assignments; in the case of a two class classifier these are assigning *true* to new data that belongs to the class and *false* to new data that does not correspond with that class (TP and TN respectively). The other squares in the matrix are for incorrect predictions, separated into data that does belong to the class but is incorrectly assigned to a different class (FN), and data that belongs to a different class but is incorrectly assigned to the class we are looking at (FP). Due to the quantity of classes in our classifier, the complete confusion matrix is too large to plot in a way that is intelligible. Instead, we plot the boundary cases for each of the four conditions. Figure 3.5a shows the normalised confusion matrix for the class with the highest *support*, or the most commonly occurring class in the dataset. Impressively, although the class takes up 27% of all the data (as shown in the True, True cell), none of the data points belonging to other classes were assigned this class, demonstrating that our model does not 'cheat' by defaulting to assigning the most common class. On the other end of the spectrum, Figure 3.5d shows the confusion matrix for the class with the lowest support. Again, all instances for this class were correctly identified and assigned even though the class had such a small representation. Because of this result, our technique of random oversampling has enabled the model to recognise a class that had very little presence in the training dataset.

Figures 3.5b and 3.5c show the classes that caused the classifier to produce the most errors. Figure 3.5b is the confusion matrix for the class with the highest number of false positives; in other words this class was the class that the classifier most commonly assigned data points to when making a mistake. The class itself does not have a particularly high support (1.835% of the data), so the errors do not arise from a problem with the balance of the data. The quantity of data points incorrectly assigned to this class (0.1%) is also not an especially high portion of the total error of the model and so we can conclude that there is no specific problem with this class. In Figure 3.5c, the confusion matrix shows the assignments for the class that is most frequently miscategorised. The class has a lower value for support (1.3%), but it still has much higher representation than the class with the lowest support.

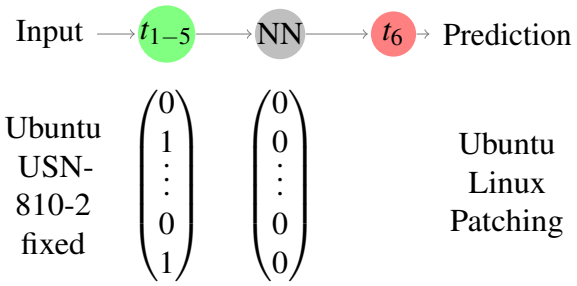


Fig. 3.6 Using the tokenizers and NN to create a prediction vector and class from an input vulnerability.

Even with this low representation, the classifier correctly assigns data to this class 92% of the time

With these results achieved, the model is clearly a good candidate for automating the classification of vulnerabilities into families.

3.3.2 Evaluation and Improvement

Due to the high importance of the accuracy of the model, once it was trained it was used in a probationary fashion for three months, being used to classify the vulnerabilities but also with the vulnerabilities checked by hand. Once the model has completed training and validation, it is used to predict the classes of new vulnerabilities added to the vulnerability database as shown in Figure 3.6. A vulnerability and all associated data is downloaded and passed into the same tokenizers used to train the network, and a one-hot vector input is created. This is put across the input of the neural network, which then outputs a label vector with the location of the maximum value in the network corresponding to the label predicted. This vector can then be processed in reverse by the label tokenizer to print the name of the class. Over this period of time approximately 4,500 new vulnerabilities were added and the model continued to perform well, with an accuracy of 99.6%. While this is still an impressive result, we would like as close to 100% accuracy as possible so no vulnerabilities are incorrectly reported to users. In order to boost the performance further, we apply a filter on the output that requires a minimum amount of certainty for the vulnerability to be classified. If the certainty is too low then instead the vulnerability is left and can be sorted by hand. In this way, the vast

Table 3.3 Improving classifier accuracy by filtering output.

Minimum Certainty	Accuracy	Un-classified
0.5	0.9963	9
0.6	0.9965	32
0.7	0.9967	45
0.8	0.9969	88
0.9	0.9973	160
0.95	0.9975	220
0.98	0.9985	486
0.99	0.9989	1163
0.995	0.9994	2154

majority of vulnerabilities can be sorted automatically and so the time requirement for sorting is significantly reduced. Table 3.3 shows the accuracy improving as stricter certainty filters are used. The number of unclassified vulnerabilities is from the complete dataset as this would then be the total number of vulnerabilities that required processing by hand were the model to be used with that parameter. As such, if an accuracy of 99.94% was desired, only outputs with a certainty of 0.995 would be used and thus 2,154 vulnerabilities of the 135,000 vulnerability database would be left to classify by hand. Even with this very strict requirement for certainty there is a 98.5% reduction in workload for the vulnerabilities to be classified.

Before this model, thousands of vulnerabilities needed to be assigned a class by hand in order to simplify vulnerability scan reports. With the help of XQ, the model has been trained and implemented, running alongside hand sorting the vulnerabilities for three months. The assignments given by hand were compared to those given by the NN classifier. The classifier achieved a prediction accuracy of 99.86% on the new data.

Overfitting. The very high level of accuracy of the model can be a cause for concern in some cases as reaching an accuracy of 100% can be a sign of overfitting. We believe this is not the case in this situation for several reasons. First, the performance of the model remains at a similarly high level to the training accuracy when applied to the validation dataset and when used over the probationary period. This suggests that the model is generalised and performs well on new data. We also use filtering to boost the accuracy, but this does not

change the workings of the model and so while the performance of the overall technique is close to 100% accurate, the actual model is less so. Finally, the task of categorising the vulnerabilities is not an especially complex one and so very high scores are more reasonable.

3.4 Machine Learning for Vulnerability Data and Related Work

Machine learning tools have already been applied to vulnerability data in various ways. Prediction of future vulnerabilities is a common use case for machine learning in the field. Zhang et al. [124] use the Weka tool for GUI development of machine learning models to test a variety of machine learning techniques on the vulnerability dataset in order to predict the feature TTNV or time to next vulnerability. Edkrantz and Said [23] use support vector machines in a similar way, but try to predict when an exploit will be available for a vulnerability, as well as investigating the NVD and an exploit database for correlations. Bozorgi et al. also use support vector machines to predict time to exploit, but go on to use their predicted metrics to create a ranking of vulnerabilities based on their calculated statistics [11]. For their preprocessing and feature generation, the majority of features they extract are from a bag-of-words representation.

For investigation of the vulnerability data, as opposed to prediction, there is less work. Lin et al. in 2017 use the Weka tool to try to discover associations between the vulnerabilities in order to infer some underlying attributes of vulnerabilities. However, they ultimately do not uncover a meaningful relationship [64]. Gawron et al. use both Naive Bayes and NNs in order to map new vulnerabilities to parts of a CVSS vector (specifically availability, integrity, confidentiality and attack range), and find that a reasonable level of accuracy can be achieved using either, although the NN approach has an over-fitting problem [29]. The authors use dropout to combat the over-fitting problem. Overall, the NN achieves slightly better results with the final validation dataset. Huang et al. also use and recommend a deep NN approach to vulnerability categorisation [43]. They claim that support vector machines and Naive Bayes classifiers are less appropriate than an artificial NN approach due to the

nature of the majority of the data in the NVD. The high dimensionality that occurs due to the text based feature space, in conjunction with the sparsity of the resulting characteristics of a vulnerability mean a NN approach will be more successful.

A large factor in the lack of investigation is likely due to the consensus that the NVD is not always very accurate, being at best an ad-hoc measure [12], but at worst unsuitable for statistical methods due to a lack of precision in the data [89, 124]. Ozment observes that the NVD has four large drawbacks that make it unsuitable for use in statistical analysis: "chronological inconsistency, incomplete inclusion, multiple entries for a single detection event, and lack of documentation" [89]. This opinion is held by the experts at XQ Digital Resilience and is shared among many security professionals [56].

Applying machine learning techniques like neural networks to text-based classification problems has been demonstrated to be an effective technique by much of the literature. These techniques have not yet been applied to grouping vulnerabilities by remediation process. We have shown in this chapter that using a neural network based on the state-of-the-art [79], vulnerabilities can be accurately classified and so this process can be used in place of hand labelling vulnerability families.

3.5 Discussion and Further Work

With the impressive accuracy of this process, we recommend that a simple neural network can be used to classify new vulnerabilities in a supervised learning environment. As discussed earlier, there are many benefits to classifying vulnerabilities in this way including the simplifying of vulnerabilities and remediation for ease of patching, fixing the problems of multiple inclusion and incomplete inclusion, and mitigating the effects of the problems of chronology and lack of documentation.

3.5.1 Achieving Perfection

While high accuracy is useful, ideally we want no vulnerabilities to be misclassified. Out of the incorrect classifications that we did find, all belonged to under-represented classes. We

demonstrate one method of improving the accuracy by only using categorisations that have a high certainty. A future improvement to this work would be to further address this problem. Due to all the incorrect classifications being under-represented classes, a more complex over-sampling technique could be applied with synthetic data, using an algorithm like SMOTE as discussed earlier. Alternatively, all under-represented classes could be grouped into a single 'by-hand' class, allowing the model to train to recognise vulnerabilities that are not from one of the common classes. We speculate that using one of these methods, you could achieve effectively 100% accuracy for the NN predictions while significantly reducing the man-hour requirement of the labelling.

3.5.2 Unsupervised Learning

In terms of general application of machine learning to vulnerability data there is still much to be done. We have demonstrated that supervised techniques are appropriate for labelling, but unsupervised learning still remains a question. After the inconclusive results of Lin et al.'s machine learning based data exploration [64], vulnerability data and any patterns within it still have yet to be fully understood or analysed. An unsupervised classification could demonstrate some unknown parallels between vulnerabilities, or the quantitative data could be linked to the qualitative aspects of the definition of a vulnerability to discover a predisposition of certain types of software or technologies to specific security problems, to allow a more pro-active approach to building secure new technologies.

Chapter 4

Probability Propagation for Cyclic BAGs

In this chapter we investigate cycles in attack graphs, and propose a general novel approximation technique for BAG calculation that can be used on any graph without modification, in fulfilment of Contribution 3 in the Abstract. This chapter is based on our paper "Cyclic Bayesian Attack Graphs: A Systematic Computational Approach" published in IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications [70]. Attack graphs are commonly used to analyse the security of medium-sized to large networks. Based on a scan of the network and likelihood information of vulnerabilities, attack graphs can be transformed into Bayesian Attack Graphs (BAGs). These BAGs are used to evaluate how security controls affect a network and how changes in topology affect security. A challenge with these automatically generated BAGs is that cycles arise naturally, which make it impossible to use Bayesian network theory to calculate state probabilities. In this chapter we provide a systematic approach to analyse and perform computations over cyclic Bayesian attack graphs. Our approach first presents an interpretation of Bayesian attack graphs based on combinational logic circuits with probabilistic inputs, which facilitates an intuitively attractive systematic treatment of cycles. We prove properties of the associated logic circuit

and go on to investigate the occurences of cycles, as well as using variable elimination as an exact method for comparisons. We then present an algorithm that computes state probabilities without altering the attack graphs (e.g., remove an arc to remove a cycle). Moreover, our algorithm deals seamlessly with all cycles without the need to identify their types. A set of experiments using synthetically created networks demonstrates the scalability of the algorithm on computer networks with hundreds of machines, each with multiple vulnerabilities. The algorithm is also evaluated against variable elimination, and is then shown to calculate correct results for examples in the literature.

4.1 Introduction

In this Chapter we exclusively consider compact attack graphs, as these are the most common type of attack graph in the literature and are best suited for modelling how individual vulnerabilities affect the security of the network, as discussed in Section 2.4. An attack graph is a representation of a system and its vulnerabilities in the form of a directed acyclic graph. It models how a system's vulnerabilities can be leveraged during a single attack to progress through a network. These attack graphs are very well suited for modelling network insecurity due to vulnerabilities that exist in software and protocols, however there are limitations for evaluating certain forms of attack, especially those that are not dependent on a specific vulnerability. For example, attacks that do not rely on certain software being installed but instead attack a system's resources are not easily modelled, like in the case of denial-of-service attacks that flood a server with legitimate requests. Another example is a password attack, where an attacker manages to discover a password prior to attacking a network due to a scam or carelessness by a user or administrator. With user account information an attacker may be able to bypass certain access controls entirely, creating a new path of access not found on the attack graph.

In recent years, several authors have pursued to combine Bayesian statistics with attack graphs to automatically prioritise network vulnerabilities from a probabilistic view point, resulting in Bayesian Attack Graphs[2, 44, 77, 96, 102, 20, 22, 39]. The literature discusses a variety of considerations when generating BAGs [75, 120], including using the Common Vulnerability Scoring System (CVSS) to associate probabilities with vulnerabilities [26]. This approach has received uptake in practical security analysis systems, in particular in MulVAL [86], which automatically generate BAGs from network scans and CVSS information.

Once a BAG has been generated it can be analysed in various ways. One approach is an exclusively static analysis, in order to identify the weakest areas of a network as well as identify quantitatively the risk that a certain asset will be compromised in the case of an attack. Further extensions to this kind of analysis include the introduction of attack profiles to modify the probabilities; for example this could be done using the attack complexity metric in the CVSS base vector to determine the ease of an attack. One approach to this is detailed by Cheng et al. [115] along with a method to include dependency relationships between vulnerabilities. Another application of the BAG is as a dynamic risk assessment tool where an administrator can model new security controls and their effects on a network, as well as dynamically analyse a deployed network's most likely attack paths, that can be updated dependent on information from an intrusion detection system [91].

The majority of the techniques used to calculate probabilities for BAGs require that they do not contain 'cycles' but are allowed to have 'loops' [26, 66, 75]. Such acyclic BAGs follow the *monotonicity principle*, that an attacker will never return to a previous state. However, networks that arise in practice when using tools such as MulVAL routinely contain cycles. These cycles arise naturally, as we will illustrate in Section 4.2.1 for a canonical example.

To deal with cycles in BAGs, the existing literature suggest to remove edges from the graph to prevent the attacker backtracking [26, 98, 22]. There are a number of practical drawbacks to this approach, especially when carried out outside the analysis algorithm, thus altering the model. For instance, removing an edge can make reasoning about the graph for a cyber security professional confusing, if the graph is examined by hand to identify specific routes and there are edges missing for the calculations. In addition, avoiding cycles that occur

when generating the BAG could be impractical and take a “substantial amount of time" [107] to ensure an edge is not cycle-causing whenever a new edge is being added to the graph.

In this chapter we propose a systematic computational approach for analysing cyclic BAGs, combining formalising models (attack graphs, BAGs and variants) and their properties throughout. We integrate the resolution of cycles in the algorithm for computing state probabilities of the BAG. The benefit of our approach is twofold. First, we establish a more formal base for the discussion of attack graphs, (cyclic) BAGs and solution techniques for BAGs, something that has not been strongly developed in the literature thus far. Secondly, we provide a single unified solution algorithm for BAGs that resolves the problem of cycles for any cyclic BAG, without altering the attack graph.

Our most significant contribution is a single unified solution to the static analysis of any BAG that does not modify the graph in any way while being able to run on graphs containing cycles. This can be used to properly analyse security threats to a network and correctly prioritise remediation steps. Moreover, when applied to acyclic BAGs, our solution provides exactly the same results identical with the outcome of approaches in the literature that deal only with the acyclic case, thus is a generalisation of these approaches.

More specifically we first introduce a running example involving the attack graph formalisms developed in Section 2.4.1. We then introduce a novel interpretation and formalization of attack graphs using combinational logic circuits that helps us reason about cycles more effectively. Combinational circuits allows one to capture intuitively the notion of subsequent visits to the same state, which we use to reason about cycles. We show that the types of cycles studied in the literature correspond to different ways in which probabilities change as a function of the visit count of a state.

Based on the formalization and the associated intuition, we derive an algorithm that handles cycles in BAGs regardless of their type in a natural manner. Due to the reasons mentioned above, the algorithm does not explicitly identify the edges in the attack graph that should be removed. Instead, it integrates the identification of cycles with the solution algorithm and then terminates the recursion. The algorithm is suitable for solving state

probabilities in the BAG, and gives results for acyclic BAGs that are identical with the output of traditional solvers for acyclic BAGs.

To study the scalability of our algorithm, we generate synthetic BAGs of various size, in a manner similar to [77]. We conclude that the algorithm can be used with BAGs of size 15000 nodes with reasonable computation time, implying it can scale to be used with networks of at least 750 hosts each with 5 vulnerabilities.

The rest of this chapter is organised as follows. Section 4.2 introduces the network architecture that will be used as a running example throughout the chapter. It also provides the motivation for our contributions. Section 2.5 introduces Bayesian networks and relates the process of Variable Elimination for Bayesian networks to the calculation of probabilities in acyclic BAGs. Section 4.3 shows how an attack graph can be interpreted as a deterministic combinational circuit with probabilistic inputs. Section 4.5 presents our unified solution for dealing with both cyclic and acyclic BAGs, and section 4.6 details the experiments run for our solution on both common and simulated examples. Finally, our conclusions are presented in Section 4.7.

4.2 Motivation and Problem Formulation

4.2.1 Running Example

As a motivating example for this work, we will consider a network architecture that could be used in a small enterprise situation. It is an example that has been used in the literature [88, 39]. The architecture is shown in Figure 4.1. The network comprises of a Database server on the internal network. This can be accessed through an internal firewall by both the user Workstations and the Webserver, that exists in the demilitarized zone subnet. This Webserver connects to the Internet via an external firewall.

For this scenario, we suppose a vulnerability scan has been run on the network, revealing three vulnerabilities that are present. There is a MySQL vulnerability on the Database server, an Apache vulnerability on the Webserver, and an Internet Explorer vulnerability on the Workstations. The workstations are modelled as a single host and imagined to be all similarly

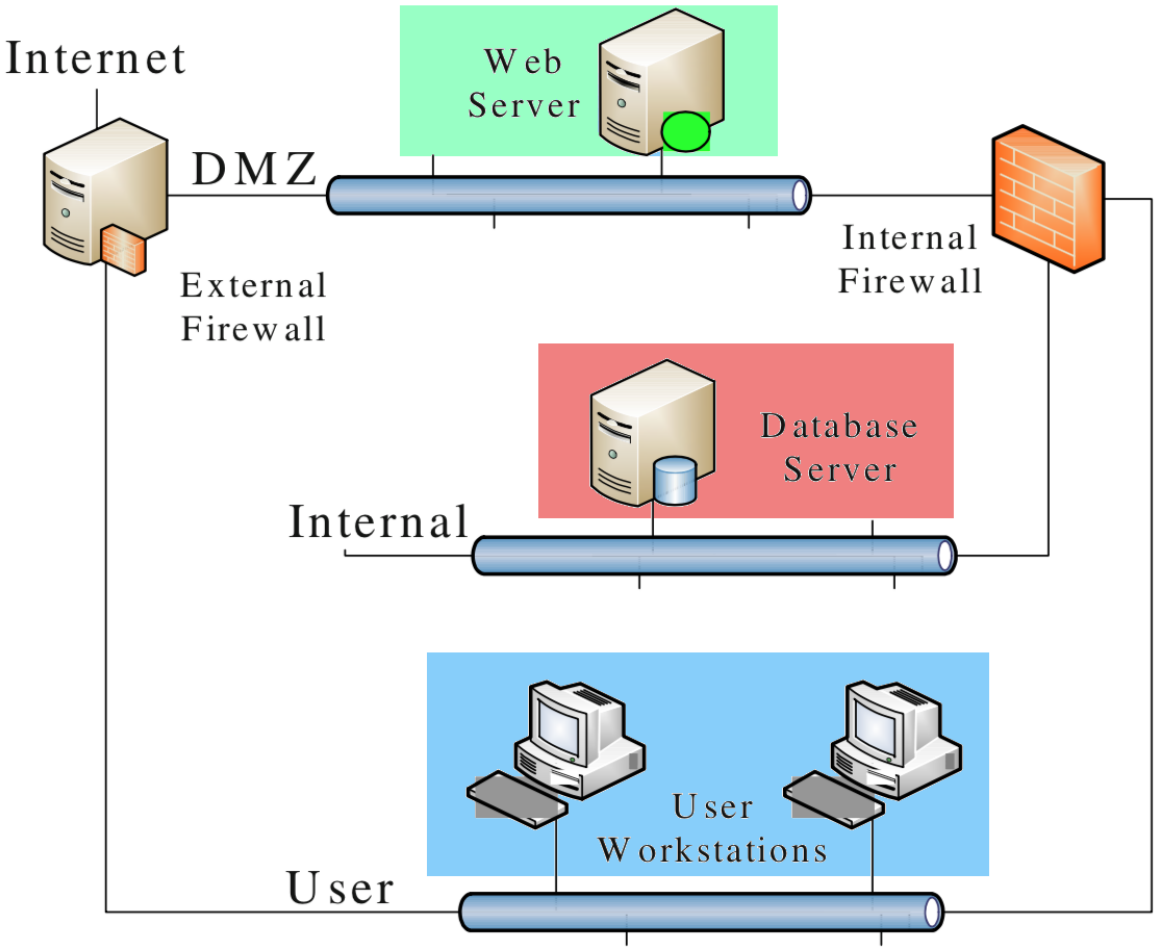


Fig. 4.1 Network architecture used as a running example.

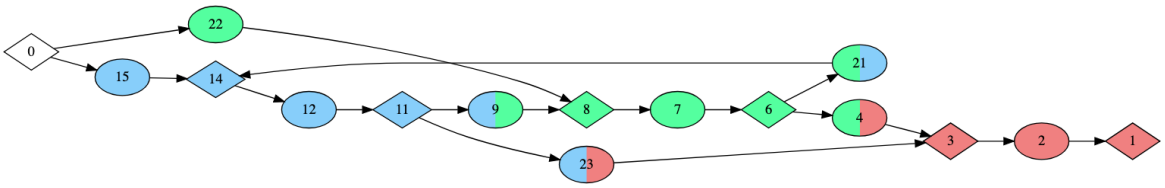


Fig. 4.2 An excerpt of the BAG of the running example. Node colours correspond with the components of the network presented in Figure 4.1.

set up and patched. This is not necessarily the case however, and will depend on the specific setup for the organisation. For this example we consider only an external attacker attempting to gain access to hosts on the network, and as such include an origin node representing external connections for hosts that are accessible from outside the network. Internal threats may also be evaluated by including origin nodes connecting to hosts that an internal threat may have access to. For example if an employee were to attempt to access confidential information on the Database server, a node indicating access of that employees privilege level (user, administrator, root) could be added to the corresponding Workstation node. We consider root access to the Database server to be the goal of an attacker in this scenario, but probabilities calculated for other nodes, for example user level access to a Workstation, can also be considered when evaluating network security.

Figure 4.2 is the corresponding BAG for this situation, with the corresponding MulVAL node labels listed in Listing 4.1. It is comprised of two node types; diamonds (OR nodes) represent a state that an attacker can be in, such as a certain level of privilege with respect to a host, and ellipses (AND nodes) are actions, such as exploiting a vulnerability or connecting to a host. Section 2.4.1 gives a formal definition of these nodes. For clarity, all Leaf nodes are removed so that the different routes to the Database server are easier to see, and the remaining nodes have been numbered so they may be referred to (a description of each node on the full graph can be found in Appendix A along with a technical explanation for the existence of the cycle and information about the vulnerabilities in the scenario). The directed edges are dependencies, an OR node can be reached if any of the parents are reached, whereas an AND node can only be reached if every parent has been reached. The colours correspond to the colours in Figure 4.1; blue is a node that relates to the Workstations, green to the Webserver, and red to the Database server. Node 0 in white represents an attacker from the Internet, to model an external attacker attempting to infiltrate the network. This node is assigned the probability 1 in any calculations as we assume that an attack is taking place and want to know the likelihood of an attacker reaching an important host in the network. Note that when using MulVAL, we do not specify any goal nodes and as such can explore the probability an attacker will reach any node in the graph. For larger graphs with a known goal node, the `-g`

option can be invoked in MulVAL to generate an attack graph with only paths relevant to that node.

Listing 4.1 MulVAL labels for Figure 4.2

```
0, "attackerLocated(internet)"
1, "execCode(dbServer,root)"
2, "RULE 2 (remote exploit of a server program)"
3, "netAccess(dbServer,tcp,'3306')"
4, "RULE 5 (multi-hop access)"
6, "execCode(webServer,apache)"
7, "RULE 2"
8, "netAccess(webServer,tcp,'80')"
9, "RULE 5"
11, "execCode(workStation,userAccount)"
12, "RULE 2"
14, "accessMaliciousInput(workStation,user, IE)"
15, "malicious website"
21, "compromise of website"
22, "RULE 6 (direct network access)"
23, "RULE 5"
```

The using the graph as an overview, it is clear there are two options available to an attacker to begin an attack. They can either attempt to compromise the Web Server (travelling through node 22 in green) or attempt to exploit a User Workstation (node 15 in blue).If the attacker were to compromise the Web Server, once obtaining privileges on the machine they can then progress to the Database Server within the internal network by continuing through node 4, or they could connect to the Workstation and exploit the vulnerability there. In a similar manner the attacker, having achieved a certain privilege level on a Workstation, can attack the Database Server by moving from node 11 to node 23, or can move from node 11 to node 9 and attempt to compromise the Web Server. As such, the logic shown in the

attack graph agrees with the reality of the network situation. While in this simple example there is seemingly little reason to attack either the Workstations or the Web Server from the other, if another goal were to be added that, for example, required credentials stored on the Workstations but was only accessible from the Web Server, it would be important that these paths are represented in the graph to fully understand all possible attacker paths.

A feature of this graph is the presence of *loops* that has been already studied in the literature. For instance, the sequence of nodes $(11,9,8,7,6,4,3,23,11)$ forms a loop (cf. Definition 2.4.1). Another important aspect of the graph is the presence of *cycles*. For instance, the sequence of nodes $(14,12,11,9,8,7,6,21,14)$ forms a cycle due to node 21 joining back to node 14. This cycle represents the fact that there are two different ways to gain access to a Workstation, and that once access is achieved a future state also allows a Workstation to be compromised. A user can simply access a malicious website, shown on the route along nodes 15 and 14, or alternatively an attacker could exploit the vulnerability on the Webserver (nodes 22, 8, 7, 6) and then compromise a website from there using the Webserver to gain access to the workstation.

Because this cycle can be entered into from multiple nodes (either 14 or 8), calculating the probability of an attacker reaching node 14, or indeed any other node in the cycle, cannot be done using basic approaches to solving BAGs. The *monotonicity principle*, which states that an attacker will always increase their privilege [5], cannot be used to remove the edge (between 21 and 14) either. This is because it is possible for an attacker to reach node 14 for the first time using the edge (21 to 14) that needs to be removed, if they enter the cycle travelling from node 22 to node 8. As such there is no loss of privilege and the graph cannot be simplified. Our approach performs the computation on the graph without the need for any simplification.

A more complete discussion of the scenario and attack graph, including the specific vulnerabilities, can be found in Appendix A.

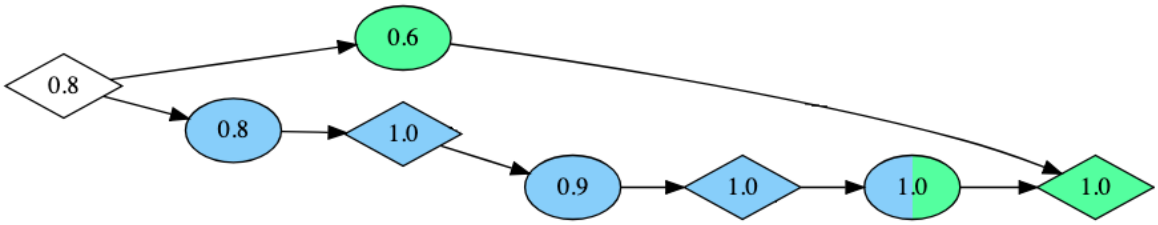


Fig. 4.3 A subgraph of the running example indicating a loop.

4.2.2 Variable Elimination

Based on the results of Section 2.5.1, the computation of access probabilities is equivalent to the computation of marginal probabilities $\text{Prob}(v = 1)$ in the associated BN. *Variable elimination* (VE) is a simple and general algorithm developed in the literature that computes exact values of these marginal probabilities (e.g., [58]). Given that the structure of the graph $(\mathscr{V}, \mathscr{E})$ models the independence between random variables associated with the nodes, we can obtain the joint distribution of the variables as the product of the conditional probability tables $\prod_{v' \in \mathscr{V}} \text{Prob}(v'|pa(v'))$. Then the marginal probabilities are computed by taking the sum over the unwanted variables:

$$\text{Prob}(v) = \sum_{v' \neq v, v' \in \{0,1\}} \prod_{v' \in \mathscr{V}} \text{Prob}(v'|pa(v')). \tag{4.1}$$

VE provides a procedure for the computation of the sum-product in (4.1). The main goal of the algorithm is to specify at each iteration which tables to multiply and which variable to sum over. The reader may refer to Koller and Friedman [58, Chapter 9] for a detailed discussion on VE.

Proposition 4.2.1 *The access probabilities of Definition 2.4.5 are equal to the marginal probabilities of the BN of Definition 2.5.1 obtained by variable elimination in the case that the BAG does not have any loops.*

In the case that the BAG does have loops, the access probabilities will not necessarily be equal to the marginal probabilities calculated using VE. This can be shown by taking the first loop from Figure 4.2 and using the local probabilities shown in Figure 4.3, then

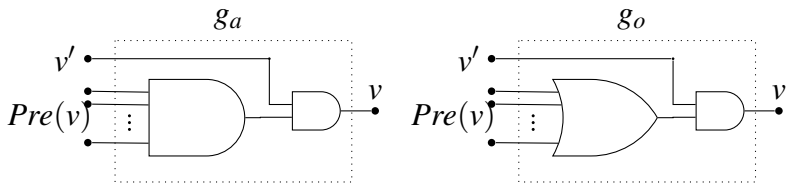


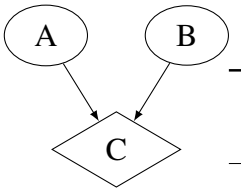
Fig. 4.4 Logic gate representation of AND and OR nodes.

marginalizing node 8 with VE and calculating the access probability for the same node using (2.2) of Definition 2.4.5. The former gives a probability of 0.7104 for node 8 being True; the latter gives 0.7795. This discrepancy is due to nodes 15 and 22 being assumed to be independent while having a common ancestor (cf. Remark 2). In other words the probability at node 0 is being allowed to contribute more than it should to the final result, hence the higher calculated probability. This level of discrepancy is already studied in the literature: VE is known to give the exact values while other computational approaches give approximate values for BAGs with loops [78].

4.3 Combinational Circuits with Probabilistic Inputs

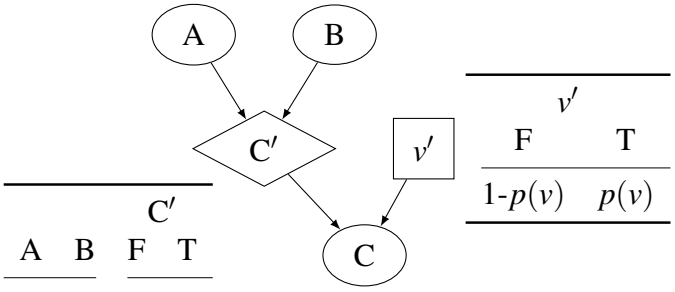
As one of the main contributions of this chapter, we look at the attack graph from a different perspective and model it as a deterministic combinational circuit with probabilistic inputs. This interpretation paves the way towards including and analysing cycles directly in the computation of access probabilities over attack graphs. Combinational circuits are mainly studied in the literature [94, 93] from the perspective of constructing a certain distribution on the output of the circuit by applying random inputs.

Let us enlarge the attack graph $(\mathscr{V},\mathscr{E})$ with the set of nodes $\mathscr{V} = \{v_1, v_2, \ldots, v_n\}$ to an augmented attack graph $(\bar{\mathscr{V}},\bar{\mathscr{E}})$ with nodes $\bar{\mathscr{V}} := \{v_1, v_2, \ldots, v_n, v'_1, v'_2, \ldots, v'_n\}$ and edges $\bar{\mathscr{E}} := \mathscr{E} \cup \{(v'_1, v_1), \ldots, (v'_n, v_n)\}$. The augmented graph is obtained by adding one node v' for each node $v \in \mathscr{V}$ and connecting it directly to v. The added node v' has the role of modeling local probabilities at node v and renders the behaviour of this node non-probabilistic. Assuming a delay in the computation of the value of the node, we get one of the following



		C	
A	B	F	T
F	F	1	0
F	T	1-$p(v)$	$p(v)$
T	F	1-$p(v)$	$p(v)$
T	T	1-$p(v)$	$p(v)$

Fig. 4.5 A graph using definitions 2.4.4-2.4.5, with local probability in OR node C.



		C′	
A	B	F	T
F	F	1	0
F	T	0	1
T	F	0	1
T	T	0	1

	v′	
F		T
1-$p(v)$		$p(v)$

		C	
C′	v′	F	T
F	F	1	0
F	T	1	0
T	F	1	0
T	T	0	1

Fig. 4.6 The graph of Figure 4.5 transformed into a graph with probabilities only on leaves.

two equations for each node:

$$v(k+1) = g_a(pa(v)(k), v')$$

or

$$v(k+1) = g_o(pa(v)(k), v'),$$

where $v(k+1)$ and $pa(v)(k)$ indicate respectively the value of node v at $k+1^{\text{st}}$ iteration and the values of the parent nodes of V at k^{th} iteration. The two functions g_a and g_o correspond to the *AND* node or the *OR* node respectively, and are depicted in Figure 4.4 using logic gates. Note that the Leaf nodes can be treated as either AND nodes or OR nodes.

A demonstration of enlarging an attack graph to make internal nodes behave in a non-probabilistic way can be seen in Figures 4.5 and 4.6. Figure 4.5 is an AND/OR BAG, with the probability table for the OR node shown to the right. We can move the local probability to a Leaf node, and as such all the internal probability tables simply become logical AND and OR tables. This is shown in Figure 4.6, where the equivalent BAG is shown, and the left and central probability tables can be recognised as logical OR and AND truth tables respectively, with the rightmost probability table containing the local probability on a leaf of the graph.

Theorem 4.3.1 *The behaviour of an attack network can be modelled as a combinational circuit with probabilistic inputs. The value of the variables are changing according to the equation*

$$\begin{cases} v_1(k+1) = g_1(pa(v_1)(k), v_1') \\ v_2(k+1) = g_2(pa(v_2)(k), v_2') \\ \quad \vdots \\ v_n(k+1) = g_n(pa(v_n)(k), v_n'), \end{cases} \tag{4.2}$$

where $k = 0, 1, 2, \ldots$ models the progression of an attacker in gaining access to the nodes or satisfying conditions along the time axis, $g_i \in \{g_a, g_o\}$ for all i, with g_a, g_o defined according to Figure 4.4. The nodes v_i' take values $\{0,1\}$ according to the local probabilities. The notation (k) is used for the k^{th} iteration.

Access probabilities are equivalent to the computation of reachability probabilities over the augmented graph:

$$\text{Prob}(v = 1) = \text{Prob}\{v(k) = 1 \text{ for some } k\}, \tag{4.3}$$

where the probability is computed over all combination of values of $\{v'_1, v'_2, \ldots, v'_n\} \in \{0,1\}^n$. Unlike previous formalisms that are not able to handle cycles, our new interpretation can easily encode cycles without the need for any modification.

Next we prove a property of this interpretation that helps in developing our algorithm for the computation of reachability probabilities in (4.3).

Proposition 4.3.2 *The system of equations* (4.2) *is monotonically increasing, i.e.,* $v_i(k+1) \geq v_i(k)$ *for any k and i and any instantiation of* $\{v'_1, v'_2, \ldots, v'_n\}$.

Proof 4.3.3 *Fix an instantiation of* $\{v'_1, v'_2, \ldots, v'_n\}$. *First we show that the function* g_a *is monotonically increasing, which is the property that* $g_a(w, v') \geq g_a(w', v')$ *for any* w, w' *with* $w \geq w'$ *element-wise. Note that* $g_a(w', v') = 0$ *if an element of* w' *or* v' *is zero, which means the inequality holds. If all elements of* w' *and* v' *are one, then all elements of w is also one, which means* $g_a(w, v') = g_a(w', v') = 1$ *and the inequality holds.*
Next we show that the function g_o *is monotonically increasing, which is the property that* $g_o(w, v') \geq g_o(w', v')$ *for any* w, w' *with* $w \geq w'$ *element-wise. This holds due to the identity* $g_o(w, v') = 1 - g_a(1 - w, 1 - v')$ *that the OR gate is the complement of the AND gate:*
$w \geq w' \Rightarrow 1 - w \leq 1 - w'$
$\Rightarrow g_a(1 - w, 1 - v') \leq g_a(1 - w', 1 - v')$
$\Rightarrow 1 - g_a(1 - w, 1 - v') \geq 1 - g_a(1 - w', 1 - v')$
$\Rightarrow g_o(w, v') \geq g_o(w', v')$.
Now the claim follows inductively from the fact that initially $v(k) = 0$ *for* $k = 0$, *and functions* g_a *and* g_b *are non-negative and monotonically increasing.* ■

Theorem 4.3.4 *The solution of* (4.2) *converges to a unique steady state in finite time, i.e., there is a time instance* k^* *such that* $v_i(k^* + 1) = v_i(k^*)$ *for any i and any instantiation of* $\{v'_1, v'_2, \ldots, v'_n\}$.

Proof 4.3.5 *According to Proposition 4.3.2, the system of equations* (4.2) *is monotonically increasing. Moreover, the space of variables in this equation is finite; each variable belongs to* $\{0,1\}$ *and the set of possible values for the vector with elements* $v_i(k), i = 1,2,\ldots,n$ *is* $0,1^n$*. In total there are* 2^n *possibilities for such a vector. Since the system of equations* (4.2) *is monotonically increasing, the system starts at some vector* $v_i(0)$*,* $i = 1,2,\ldots,n$ *and increases at each time step (according to the order defined on the space of vectors). Therefore, there is a finite number of indices k such that* $(v_1(k+1), v_2(k+1), \ldots, v_n(k+1)) \neq (v_1(k), v_2(k), \ldots, v_n(k))$*. The first index k where* $(v_1(k+1), v_2(k+1), \ldots, v_n(k+1)) = (v_1(k), v_2(k), \ldots, v_n(k))$ *gives the converged index of the system (after this index, the values of* v_i *do not change anymore). Note that this index is a function of values of* $(v'_1, v'_2, \ldots, v'_n)$*. By taking the maximum of such indices for different values of* $(v'_1, v'_2, \ldots, v'_n)$*, we get the index* k^* *in the statement of Theorem 4.3.4.* ■

Theorem 4.3.6 *For an acyclic BAG with the time instance* k^* *defined in the previous theorem,* $Prob(v(k^*) = 1)$ *is the same as the probability computed via Variable Elimination on the BAG. Furthermore, if the BAG does not have loops, this quantity is the same as access probabilities in* (2.2)*.*

Proof 4.3.7 *According to Equality 4.3, access probabilities are equivalent to the computation of reachability probabilities over the augmented graph;* $Prob(v = 1) = Prob\{v(k) = 1$ *for some* $k\}$*. Based on Theorem 4.3.4, we get that* $Prob(v(k) = 1$ *for some* $k) = Prob(v(k^*) = 1)$*. Therefore,* $Prob(v = 1) = Prob(v(k^*) = 1)$*, which means* $Prob(v(k^*) = 1)$ *is the same as the probability computed via Variable Elimination on the BAG. If the BAG does not have loops, the result of variable elimination is the same as access probabilities in* (2.2)*.* ■

Reachability probabilities $\text{Prob}(v(k^*) = 1)$ are well-defined on combinational circuits regardless of the existence of cycles. Therefore, the above theorem gives a nice direction for generalizing computations to cyclic BAGs. In the next section, we provide an algorithmic computation of probabilities while replacing the joint distributions with product of marginal distributions, similar to (2.2).

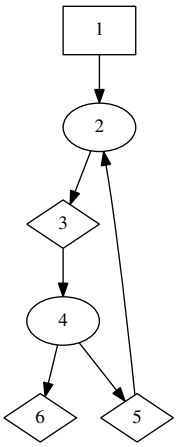


Fig. 4.7 Cycle of type 1.

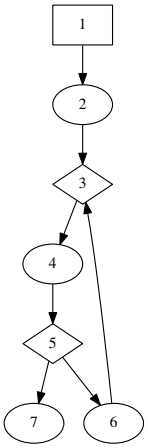


Fig. 4.8 Cycle of type 2.

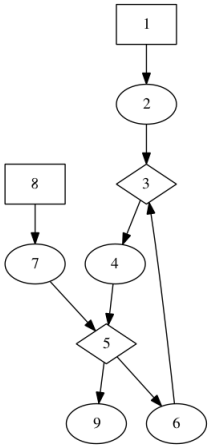


Fig. 4.9 Cycle of type 3.

4.4 Attack Graphs with Cycles

As we discussed in Section 4.2, most of the literature on BAG probability computations have focused on acyclic attack graphs. This constraint ensures the probabilities on the nodes become the chance that an attacker reaches the node at all. In other words, paths that are enabled by access to a node should not increase the probability that that same node is reached. We presented a running example and showed that cycles can occur in a number of situations. Thus, this property should be extended for each node on the pathway calculations.

In the following, we first discuss types of cycles mentioned in the literature and the methods currently used to deal with these cycles. Next we put these types of cycles into the perspective of our computational algorithm and show how they can be interpreted over the associated combinational circuit. We emphasise that our solution does not modify the graph in any way while being able to run on graphs containing cycles. Our solution deals directly with the cycles and integrate cycle resolution with the computation of state probabilities without the need for identifying or differentiating the cycle types.

4.4.1 Handling Cycles

There are three main types of cycles in an attack graph [26, 22], and Figures 4.7 to 4.9 shows an example of what each type of cycle could look like based on Definition 2.4.4. Cycles are

assigned a type according to the required process for calculating probabilities on the graph with that cycle. The first two types of cycle can be trivially dealt with by ignoring the cycle or removing an edge. However, as discussed in [26], there is no trivial edge removal solution for dealing with cycles of type 3. We now discuss each type of cycle in detail.

The first cycle type, Figure 4.7, can be demonstrated as removable. This is because node 2 has two prerequisites and one of them, node 5, is only fulfilled given that node 2 has been accessed. As such node 2 can never be true and the cycle will never occur and does not aid our understanding or modelling of the graph.

Figure 4.8 shows the second type of cycle, one that cannot be trivially removed like the first. Node 3 can be fulfilled by either 2 or 6 meaning that node 7 is reachable. Since node 6 can only be accessed after node 3 has been accessed, it should not contribute to the likelihood of reaching nodes 3, 4, 5 or 7. In essence the edge from node 6 to 3 could be removed from the graph, and this is the necessary step in current acyclic BAG techniques.

An important addition to this discussion is that while edge removal does work for the second type of cycle, it is perhaps not the preferred solution. Firstly removing the edge can make the graph less understandable from an engineering perspective, as logically if reaching a state allows access to another element that was a prerequisite to a previous state then that route is possible. Also, more importantly, removing an edge, the edge from node 6 to node 3 in our example, removes information that could be important in future. If it was desired that a new vulnerability be added to the graph, for example in such a way that it would transform Figure 4.8 into Figure 4.9, then a legitimate route would now be missing from the graph and from any new calculations. As such, preserving the structure of the graph is important for both future analysis and understanding.

The third type of cycle, Figure 4.9, shows a cycle that cannot be ignored or fixed through edge removal. It is the same in structure as the Type 2 cycle but a node in the cycle has an extra way of being accessed, meaning there are now multiple routes to reach node 3. This type of cycle should be dealt with by imagining probabilities are populations of attackers and as such the quantities on the nodes should represent unique attacker numbers ensuring we do not double count attackers moving through the graph. Practically for this simplistic example

this will mean the probability at node 6 is the disjunction of attackers coming from node 7 and the attackers reaching node 4 for the first time, i.e. the initial population at node 1 minus any attackers lost moving through nodes 2 and 3.

It is also possible for nested cycles to exist; for large cycles involving many nodes, a subset of these nodes may comprise a different cycle. In general nested cycles are also complex to deal with. If a cycle of Type 3 contained a subset of nodes that also formed a Type 3 cycle then naturally neither the subset nor the larger cycle could be removed from the graph. However, if a Type 3 cycle were to contain a Type 1 or 2 cycle in a subset of it's nodes, the smaller cycle could no longer be removed in the usual way due to the nodes in the cycle having an effect on the other nodes in the larger cycle. As such the complexity for dealing with a graph can increase with nested cycles.

4.4.2 Cycles in Combinational Circuits

Using the combinational circuit paradigm introduced in Section 4, we can formally describe the different types of cycle. The finiteness of k^* mentioned in Theorem 4.3.4 enables us to unfold the logic circuit k^* number of times. Let us denote

$$pa(v_i) = \{v_{i1}, v_{i2}, \ldots, v_{i,m_i}\}.$$

The unfolding is done sequentially by replacing each $v_{ij}(k)$ in the right-hand side of (4.2) with function $g_z(pa(v_z)(k-1), v'_z)$ where v_z is the node associated with v_{ij}. Starting this process from k^* and repeating it k^* times gives us a full circuit as

$$\begin{cases} v_1(k^*) = f_1(v'_1, v'_2, \ldots, v'_n) \\ v_2(k^*) = f_2(v'_1, v'_2, \ldots, v'_n) \\ \quad \vdots \\ v_n(k^*) = f_n(v'_1, v'_2, \ldots, v'_n), \end{cases} \tag{4.4}$$

where f_i's are associated to the unfolded circuit with a directed graph that does not have any cycles. Unfortunately, the procedure of unfolding and probability computation over (4.4) is

computationally intense but it is very helpful in giving an automatic characterisation of cycle types in BAGs discussed in the literature [26, 22].

Cycles of Type 1 are seen when the steady-state value of a node is zero: $v_i(k^*) = 0$ for some i and for any instantiation of $\{v'_1, v'_2, \ldots, v'_n\}$. This means that the node cannot be reached and can be safely eliminated from the analysis. Any incoming edges or outgoing edges to this node can also be eliminated. If this elimination results in breaking a specific cycle, that cycle is of Type 1.

Cycles of Type 2 need more elaboration and are defined with respect to nodes that $v_i(k^*) = 1$. Let k_i^* be the earliest time that the value of node v_i becomes one:

$$k_i^* = min_k\{k, v_i(k) = 1\} \text{ for } i \text{ with } v_i(k^*) = 1. \tag{4.5}$$

It is obvious that k_i^* depends on the instantiation of $\{v'_1, v'_2, \ldots, v'_n\}$ and is upper bounded by k^*. The computation of prob$(v_i(k^*) = 1)$ requires unfolding (4.4) for k_i^* times. Nodes with the property that $v_j(k_i^*) = 0$ can safely be removed together with their outgoing and incoming edges. These are the nodes that do not have any influence on node v_i. If such elimination results in breaking a cycle, that cycle is of Type 2. Note that the elimination is valid when we only need to compute access probabilities of v_i (the definition is with respect to a particular node).

The previous two types of cycles require properties that should hold for any instantiation of $\{v'_1, v'_2, \ldots, v'_n\}$. Cycles of Type 3 are the ones that do not fit in the definition of cycles of Type 2. Formally, for a given node v_i, a cycle is of Type 3 if there exists a node v_j on the cycle and an instantiation of $\{v'_1, v'_2, \ldots, v'_n\}$ such that $v_j(k_i^* - 1) = 1$. This means node v_j on the cycle can influence the access probability of node v_i, thus cannot be removed.

As discussed above, cycles of Type 3 cannot be removed and requires a particular attention when performing the probability computations. In the next subsection, we demonstrate the computation on the running example and provide the full algorithm in Section 4.5.

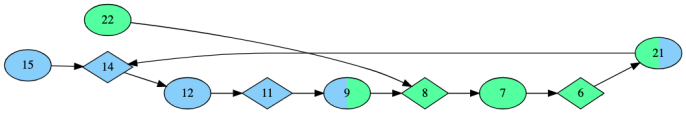


Fig. 4.10 The cycle of the BAG presented in Figure 4.2.

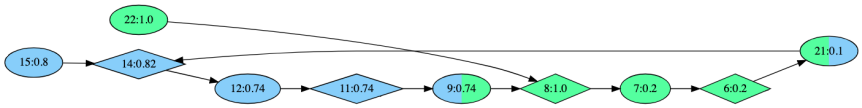


Fig. 4.11 The cycle of the BAG presented in Figure 4.2 with computed access probabilities.

4.4.3 Calculating Probabilities for the Running Example

Here we discuss how a cycle's probabilities should be calculated, as is implemented in the algorithm in the following section. Figure 4.10 is an excerpt from the larger attack graph of the running example presented in Section 4.2.1. It is a simple example of how a Type 3 cycle (see Figure 4.9) can exist in a real attack graph. This excerpt is a trivial demonstration of the simplest occurrences of cycles in real attack graphs. The cycle occurs because of the multiple routes that can be taken to reach node 14, where an Internet Explorer vulnerability on the Workstations can be exploited using an HTML document. A user may visit a malicious website, represented by the route from node 15 to 14, or alternatively the Webserver could be targeted first, and used to provide the HTML document once it has been compromised, shown in the route through nodes 8, 7, 6, 21 and then 14.

The cycle is further complicated by the fact that the Webserver can be accessed directly without going through the Workstations, in the route from node 22 to 8. Without the edge $e_{22,8}$, the edge causing the cycle ($e_{21,14}$) could be safely trimmed as the probability of the attacker reaching node 14 does not increase due to node 21 as node 14 is a prerequisite for node 21 to be reached. Node 22 entering the cycle part way through, however, will increase the probability of reaching node 14 at some point in an attack as node 15 being accessed is no longer a requirement.

The result of calculating the probability of reaching each node, performed by disregarding nodes that have already contributed, can be seen in Figure 4.11. In order to calculate probabilities within the cycle, all the parent nodes are collected exhaustively. Their contribution

to the probability of the node in question is then performed according to the relationships defined by the graph, as in Definition 2.4.5, with the caveat that any node in the parent set may only contribute once to the calculation. In this way, calculating the probability of node 12 on Figure 4.10 will involve the likelihood of any attacker reaching node 14 from node 15, and also the likelihood of an attacker reaching node 14 from node 21, but with the removal of node 15's contribution to the probability of reaching node 21 as node 15 has already been included. In this way each nodes probability can be calculated without the removal of any edge, as a node causing a loop in one place may contribute to probabilities elsewhere on the graph and as such should only be disregarded in specific calculations where it's effects have already been calculated. The ability for a node to be present in multiple paths but contribute differing amounts demonstrates the idea that a recursive algorithm that identifies each node's contribution to a path would be a correct solution to this problem, preventing any node from contributing multiple times to the same path.

4.5 Calculation on Cyclic BAGs

4.5.1 Algorithmic Inference

We have created an algorithm to propagate probabilities through an attack graph and thus generate a BAG based on the probabilities assigned at the leaves of the graph. The algorithm, shown in Algorithm 1, works by detecting all attack paths that lead to a node. It moves through each step in each attack path collapsing cycles by the method previously discussed (prevention of multiple instances of the same node from contributing to the probability multiple times) and calculating probabilities using the recursive conjunction and disjunction functions. Probabilities on a path are calculated fully for each node as the chance of a node being reached, but will not necessarily reflect its contribution to the proceeding nodes. In essence this means that when a node is being calculated, all the nodes that contribute to this probability are identified and only allowed to contribute to the final calculation once, thus ensuring that no nodes in a loop inflate the final probability by being counted multiple times.

The order of calculation for the nodes does not matter and can be performed in a random order as contributions to a probability are explicitly calculated for each node every time.

The input to the algorithm is the graph with local probabilities. The `pop(int)` function used in the *Disjunction* and *Conjunction* functions in Algorithm 1 is a list function that returns the item, in this case a probability, at the given list index then removes it from the list.

This algorithm achieves a calculation for node probabilities that makes sense given the context of network security without the normal requirement for removing edges from the graphs. This means that attack routes on graphs can be better understood while also improving the versatility of graphs as extra portions can be added and the new nodes can be calculated correctly as no edges have been removed.

4.5.2 Complexity of the Algorithm

The complexity of Algorithm 1 can be calculated as $O(n \times \max_v |Pre(v)|)$ where n is the number of nodes in the attack graph. In the worst case, when every added node is required to be present in the calculation of every other node, the complexity of the algorithm is $O(n^2)$. If the cyclicity of the graph is known, then the complexity becomes $O(n(cn + \max_v |Pre(v)|))$ where $0 \leq c \leq 1$ and c is the portion of nodes that are in at least one cycle.

4.5.3 Selection of Local Probabilities

In order to infer the access probabilities for the nodes passed to the algorithm, an initial set of local probabilities must be provided. These were originally generated from a simplistic evaluation of the ease to exploit a vulnerability (with non leaf local probabilities set to 1 as discussed earlier in Section 3.2). In order to determine the ease of access, the CVSS vector [108, 73] for the vulnerability is collected from NIST's National Vulnerability Database (NVD). Currently the Access Complexity (CVSSv2) or the Attack Complexity (CVSSv3) is used to define the probability of transitioning to a state. This is on a scale of Low, Medium and High for version 2 and Low and High for version 3, with High meaning there is a great deal of skill or timing required to exploit the vulnerability and as such is associated with

Input: Attack Graph; nodes v in V with local probability $p(v)$
Output: Bayesian Attack Graph; nodes v in V with access probability $P(v)$
for $v \in V$ **:**
  $P(v) =$ `RecursiveProbability(`v, v`)`
def `RecursiveProbability(`*node v, origin node* v_{origin}`)`**:**
  if $v \in V_l$ **:**
    return $p(v)$
  else:
    for $v_{pa} \in V_{parents}$ **:**
      if v_{pa} *is* v_{origin} **:**
        append 0 to p_list
      elif v_{pa} *has been visited already* **:**
        if $v_{pa} \in V_l$ **:**
          append $p(v_{pa})$ to p_list
        else:
          append 0 to p_list
      else:
        append `RecursiveProbability(`v_{pa}, v_{origin}`)` to p_list
    if $v \in V_a$ **:**
      return *Conjunction(p_list)*
    if $v \in V_o$ **:**
      return *Disjunction(p_list)*

`Disjunction(`Probability List P`) /* Calculating the disjunctive probabilities for OR nodes */`
Input: List of probabilities
Output: Disjunction of the probabilities
p = P.pop(0)
if *length of P is larger than 1* **:**
  recursive_p = `Disjunction(`P`)`
  p = p + recursive_p - recursive_p $\times$ p
elif *length of P is equal to 1* **:**
  return p + *P[0]* - $p \times$*P[0]*
else:
  return p
return p

`Conjunction(`Probability List P`) /* Calculating the conjunctive probabilities for AND nodes */`
Input: List of probabilities
Output: Conjunction of the probabilities
p= P.pop(0)
if *length of P is larger than 1* **:**
  recursive_p = `Conjunction(`P`)`
  p = recursive_p $\times$ p
elif *length of P is equal to 1* **:**
  return $p \times$ *P[0]*
else:
  return p
return p

Algorithm 1: Propagating probabilities through the attack graph. The conjunctive and disjunctive functions correspond to calculating probabilities on OR and AND nodes, respectively.

Table 4.1 Complexity scores and their local probabilities.

Vector Score	CVSS Version	Local Probability
Low/L	2,3	0.71
Medium/M	2	0.61
Unknown	-	0.61
High/H	2,3	0.35

the lowest probability scoring. A demonstration of how these values could inform the local probabilities is shown in Table 4.1.

The local probabilities are taken from the contribution that the NVD gives to a vector score when calculating the whole CVSS score. While this is a useful approximation, it is very abstract and ignores a great deal of the information that can be gleaned from the information available about the vulnerabilities. A discussion of this can be found in Section 2.5.3.

4.6 Experimental Results

4.6.1 Simulating Networks

In order to test this algorithm and other procedures on data, we simulate a variety of networks by producing attack graphs with a range of adjustable variables using our own simulator, which can also generate attack graphs with cycles. The graph is built out of nodes generated with a Leaf:AND:OR ratio of 50:40:10 in order to model the fact that approximately half a common attack graph comprises of configuration Leaf nodes, and there are fewer nodes representing states than there are representing attack steps (this is due to the fact that multiple different AND nodes can lead to the same OR node i.e. many actions can lead to the same state). The specific numbers are derived from the average ratios on our attack graphs built from real networks, but with the OR node count rounded up to increase the model complexity to represent more of a worst-case scenario in terms of computation.

The simulator then builds a random attack graph using these nodes with a specifiable quantity of cycles; it is given a percentage of cycles to artificially add, and ensures that the given percentage of OR nodes are involved in cycles (as this is where cycles originate, from

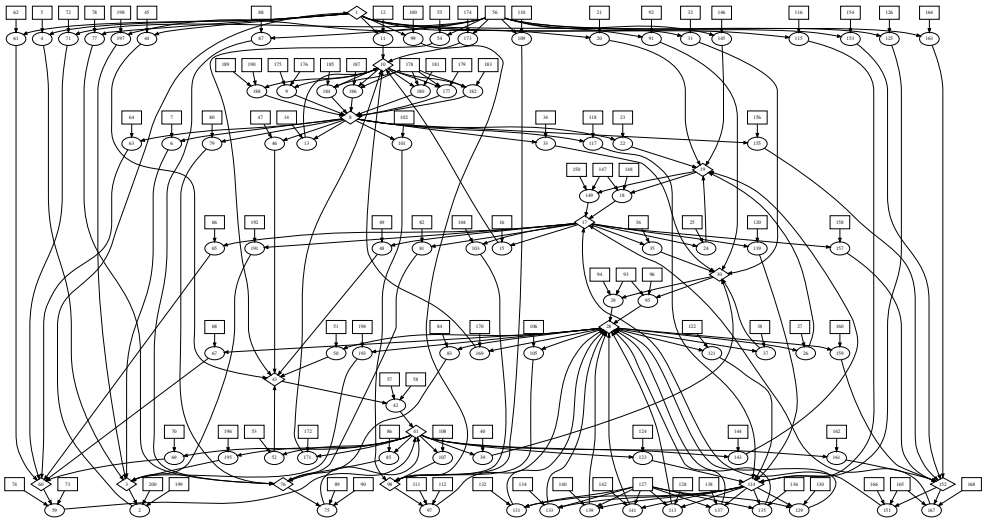
the state of privileged access that allows potential future access to a vulnerability that has already been exploited). Finally the probabilities for the vulnerabilities are sampled randomly from the Low/Medium/High levels shown in Table 4.1. A real attack graph can be seen in Figure 4.12a alongside a simulated attack graph of the same size, in Figure 4.12b. In general we find that the simulated graphs resemble actual attack graphs with very high consistency. Using the same value for cyclicity and copying the size of a real graph, a virtual graph created using our process will have only up to a $\pm 5\%$ difference in the number of edges. This fact, alongside comparison by eye and the similar run times given when running Algorithm 1 on real and simulated graphs, indicates that the graph simulator is a good approximation for running on real attack graphs, while having the benefit of being able to produce graphs of any size without requiring a time consuming vulnerability scan or a network being set up.

Simulation Algorithm and Design

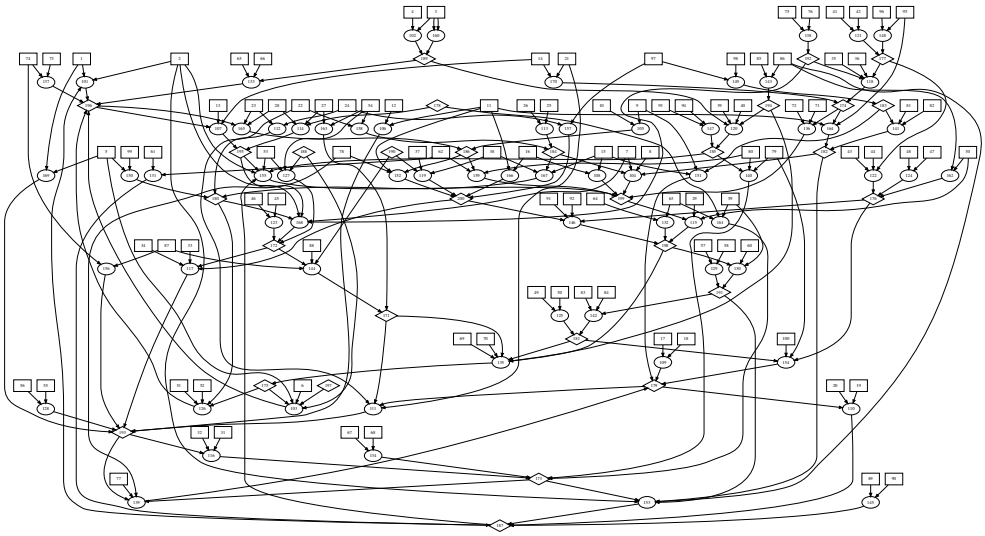
The simulator can generate a graph with the numbers of each node type explicitly declared and an exact quantity of cycles, however in general it is only provided with a graph size and a value for cyclicity. The full process is described in Algorithm 2. The function `RandomProb()` returns a randomly sampled probability for a node, `RandomSelect()` makes a random selection of one node from a list, and `CollectParents()` collates all of a node's predecessors. The algorithm generates numbered nodes for each node type up to the specified graph size. The AND nodes are then assigned to a randomly selected OR node, as an OR node will always lead to at least one state, and a random amount of Leaf nodes, between two and four as in general AND nodes have two requirements but may have up to four. The OR nodes are then randomly assigned to be parents of AND nodes, as some exploits require a specific state of privilege in order to be executed. For each edge added it is first evaluated to check if it causes a cycle, and if it does it is only added if there are cycles left in the cycle counter. Once all of the edges of the base graph have been added, cycles are then randomly included and the cycle counter decremented until the counter reaches 0. The attack graph is then written to a file.

4.6.2 Application to Simulated Networks

In order to test the practicality of the algorithm, it is implemented in Python in order to run it on a variety of simulated graphs, along with like-real networks and common examples in literature. Attack graphs were simulated in increasing sizes, from 500 to 15000 in step sizes of 500, and in four different groupings; '0% cyclic' where there are no cycles in the graph, '5% cyclic' where five percent of OR nodes are involved in cycles, '25% cyclic', and '100% cyclic' where every node, excluding Leaf nodes, is included in a cycle. Every situation was



(a) A real 200 node attack graph



(b) A simulated 200 node attack graph

Fig. 4.12 Real *(top)* and simulated *(bottom)* attack graphs with 200 nodes

Input: Number of nodes N, cyclicity C
Output: Attack graph of size N and cyclicity C
$leaves$ = max(2,(N*0.5))
$ands$ = max(1,(N*0.4))
ors = max(1,(N*0.1))
$cycles = ors \times C/100$
$n = 1$
for $i \in leaves$ **:**
    $leafDict[n]$ = Node$(n, LEAF$,RandomProb())
    n += 1
for $j \in ands$ **:**
    $andDict[i]$ = Node$(n, AND, 1)$
    n += 1
for $k \in ors$ **:**
    $orDict[i]$ = Node$(n, OR, 1)$
    n += 1
$nodes = ors \cup ands \cup leaves$
$edges$ = []
$l = 1$
for $node \in ands$ **:**
    orNode = RandomSelect(ors)
    append edge $(node, orNode)$ to $edges$
    for *i in range(*RandomInt(*2,4))* **:**
        if $l < leaves$ **:**
            append edge $(l, node)$ to $edges$
    else:
        append edge (RandomInt(1,$leaves$), $node$) to $edges$
for $node \in ors$ **:**
    for *i in range(*RandomInt(*3))* **:**
        andNode = RandomSelect($ands$)
        if *(node,andNode)* $\notin edges \wedge$ *andNode* $\notin$ *node.Parents* **:**
            if $node \notin$ collectParents($andNode$) **:**
                append edge $(node, andNode)$ to $edges$
            elif $cycles > 0$ **:**
                append edge $(node, andNode)$ to $edges$
                $cycles$ -= 1
while $cycles > 0$ **:**
    $orNode$ = RandomSelect(ors)
    $parentList$ = CollectParents($orNode$)
    remove Leaf nodes from $parentList$
    $cycleNode$ = RandomSelect($parentList$)
    if $(orNode, cycleNode) \notin edges$ **:**
        append edge $(orNode, cycleNode)$ to $edges$
        $cycles$ -= 1
return AttackGraph($nodes, edges$)

Algorithm 2: Process to generate attack graphs of specifiable size and cyclicity. The function `CollectParents()` is a recursive function that populates and returns a list of all the predecessors of the given node.

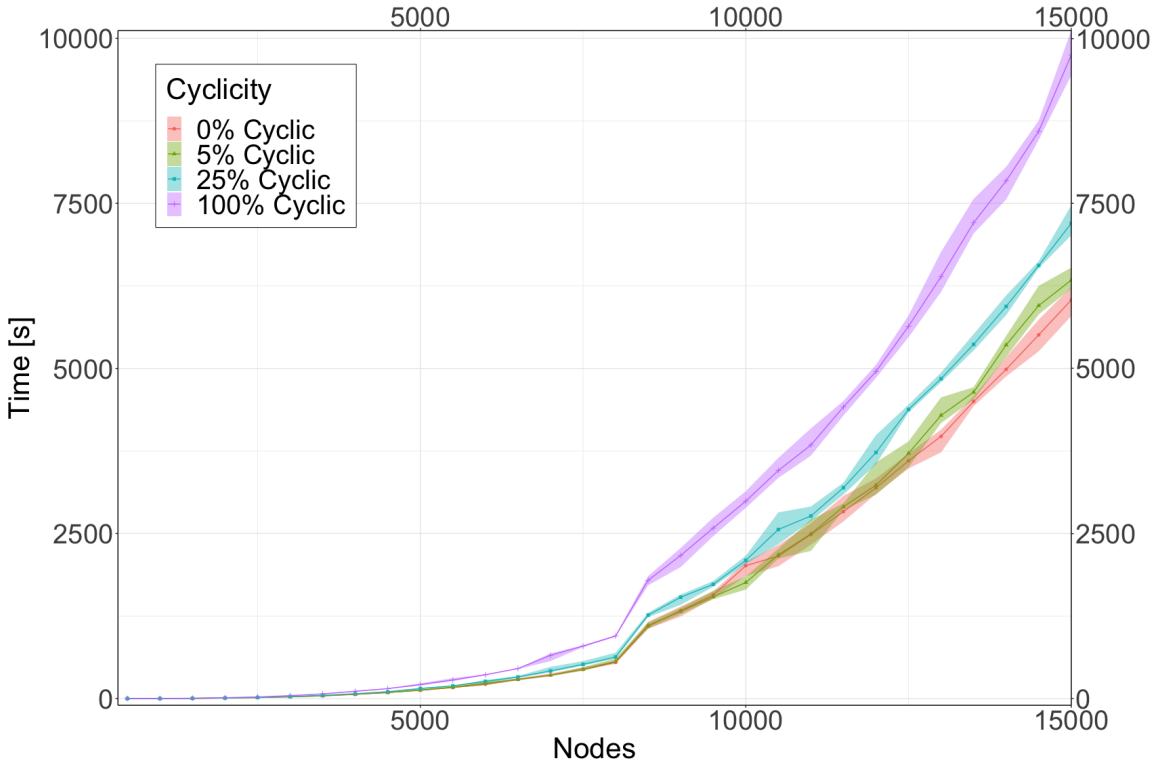


Fig. 4.13 Time performance of Algorithm 1.

simulated five times, and the computation time required to complete the algorithm was timed. These results are shown in Figure 4.13. Plotted are the average values for each amount of nodes, as well as range bars for the minimum and maximum values. The reasoning for the choices of groupings is as follows. The '0% cyclic' group, or acyclic group, allows direct comparison to standard literature methods for acyclic graphs. '5% cyclic' aligns with the average cyclicity across our real networks, while '25% cyclic' is the maximum cyclicity we have found in real graphs, rounded up to the nearest 5, so these groups represent how well we expect the algorithm to perform in general along with an approximate realistic worst case. Finally '100% cyclic' shows the largest possible penalisation for including cycles in computation.

As can be seen, the algorithm can generate probabilities for graph sizes of at least 15000 nodes in the worst possible case in a moderate time frame, around 155 minutes. Thus the algorithm is suitable for use on medium-large sized networks, especially if modeling techniques like consolidating similar work stations into singles nodes is used. By way of

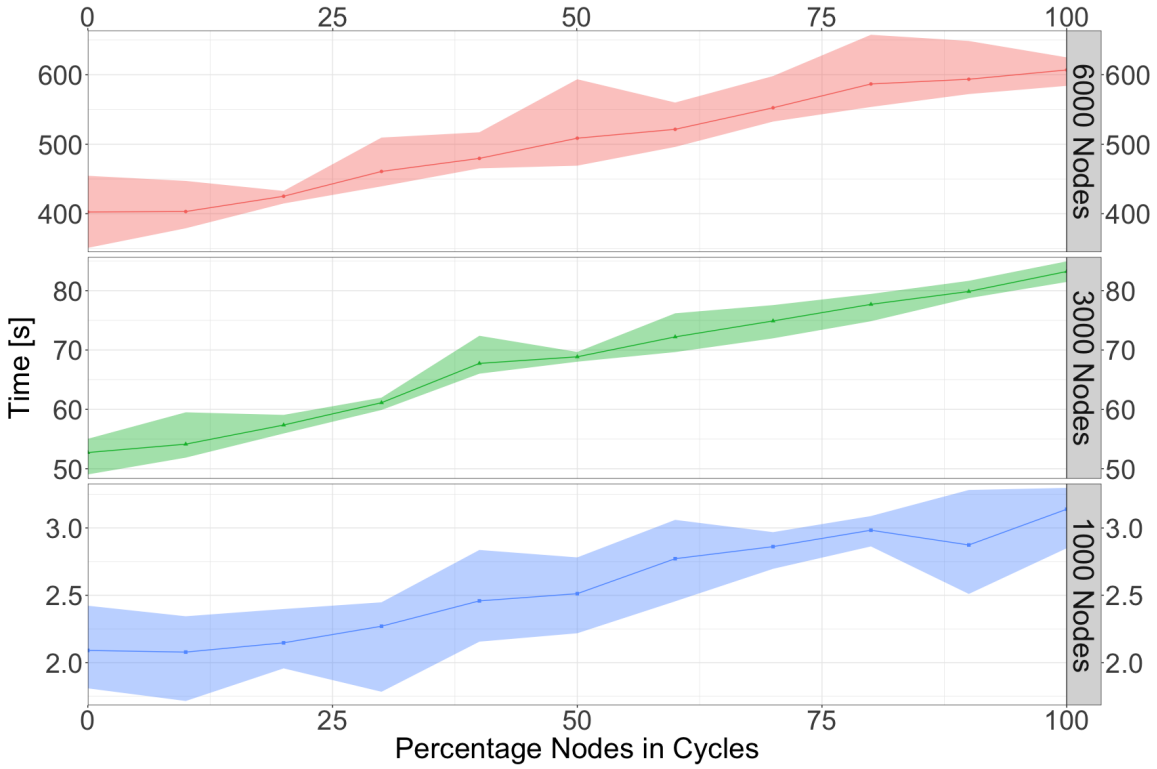


Fig. 4.14 Impact of cyclicity on the computation time.

comparison, an enterprise with 300 machines, each with an average of 5 vulnerabilities, would create a graph of around 6000 nodes. In a worst case situation regarding cycles, the probabilities could be calculated in approximately 600 seconds.

The effect of increasing the percentage of cycles at fixed node quantities was also examined, with results displayed in Figure 4.14. Graphs were simulated with 1000, 3000 and 6000 nodes, with the percentage cycles being part of cycles is increased from 0 to 100 in steps of 10. The contribution of the cycles to the computation time increases with the quantity of cycles on the graph, up to an approximate 75% increase in computation time in the worst case in the ranges of nodes that we experimented with. This can be seen in both Figures 4.13-4.14 by comparing the results for 0% cyclicity with 100%. Including cycles seems like a somewhat expensive addition to graphs; however when any changes are to be modelled there will be no requirement to recompute the graph and as such the upfront cost will mean that only small portions of the graph will have to be computed after changes are

made as the structure of the graph will be correct. In other words the upfront increase in time significantly reduces the cost of future adaptations and analyses of the graph.

4.6.3 Application to Realistic Networks

To confirm the applicabality of the results to realistic scenarios, we have implemented the algorithm on a collection of realistic networks. These are networks designed to replicate common real-life networking implementations for testing and modelling purposes. They are set up as a combination of virtual and real assets that function as an enterprise network, with different numbers of hosts and different vulnerabilities on each machine. These network scenarios return the same scan results that one would expect from running a vulnerability scan on the real network equivalent of the scenario.

We have considered three networks with respectively 10, 15, and 18 hosts; the first network comprises of a Linux database and workstation, an iOS device, a peripheral device that uses SMB and a Windows server, with all the remaining hosts being Windows workstations. The workstations have various commonly used applications installed (Chrome, iTunes, JDK) and are in various patching states. The other networks have similar enterprise topologies, see Appendix B for more details. Each network is scanned using a modified version of the OpenVAS vulnerability scanner [35], and then an attack graph is generated for each using MulVAL [86]. The number of nodes in the attack graphs generated from these networks is 1053, 2234 and 2342; all have naturally occurring cycles. This gives us attack graphs that are roughly 5% cyclic. The mean runtime of our algorithm is 3, 11, and 12 seconds, respectively. This runtime confirms the quantities reported in Figure 4.13: the simulations for attack graphs with 5% cyclicity and 1000 and 2000 nodes have mean runtimes of 2 and 18 seconds respectively when run on the same machine used for the realistic networks.

4.6.4 Evaluation on Examples from Literature

To demonstrate the validity of our method we have applied Algorithm 1 on two common attack graphs from the literature; an acyclic graph generated from a scenario presented by

Wang et al. [114], and the cyclic example in Figure 4.2 that was created by Ou et al. [85] and had probabilities calculated by Homer et al. by creating a larger equivalent graph that is acyclic [39]. Once the acyclic example was converted from a plain BAG into an AND/OR BAG, our algorithm was run on both examples. The results were identical to the expected results in the other papers, demonstrating that this algorithm gives correct results for common network scenarios and generalises them to the larger class of cyclic BAGs.

4.6.5 Acyclic Example

The network scenario, in Figure 4.15, is moving from a Workstation to root access on a Database server through a Firewall and possibly via a File server depending on the attack path. There are three services running on Machine 1, the file server, and two services running on Machine 2, the target Database server. An attack graph has already been generated for this scenario but using a different schema, shown in Figure 4.16. Here it can be seen that there are three possible paths for achieving the goal of root access to Machine 2; the attacker can either edit the trusted host list on Machine 2 to gain enough access to the host in order to run the buffer overflow attack, or the attacker can attempt to reach the same privilege by changing the relationship between Machine 1 and Machine 2, either by initially editing Machine 1's trusted host list or by attempting a buffer overflow attack against Machine 1.

First, an AND/OR BAG was generated from the plain BAG in Figure 4.16 using the principles discussed in the Section 2.4.2. This involves moving all less than one local probabilities to the leaf nodes allowing easy logical calculation of the marginal probabilities at each node. The algorithm was then run on the new attack graph with the same probabilities associated with the vulnerabilities as allocated in the original example to generate a new Bayesian attack graph. This new graph, Figure 4.17, shows the same probabilities for each part of the graph. This demonstrates that the algorithm is correct for this common acyclic graph example.

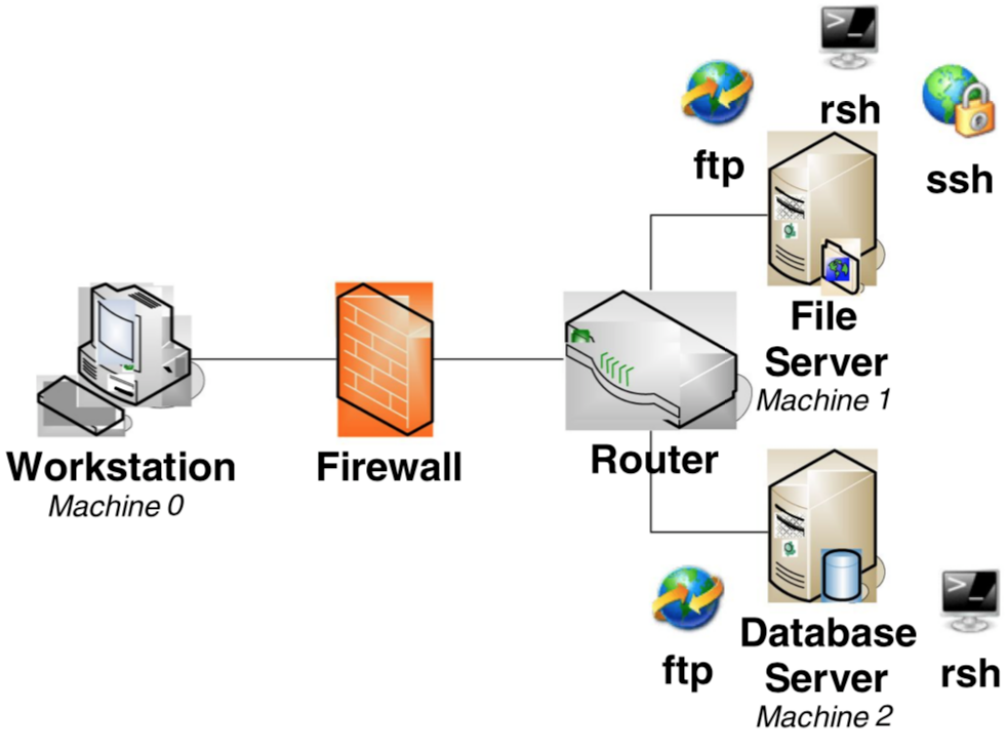


Fig. 4.15 Example of a network taken from [114].

4.6.6 Cyclic Example

The probabilities are calculated and shown in Figure 4.18 by applying Algorithm 1 to the running example presented in Section 4.2.1 and described in Appendix A. This graph has all the nodes displayed, including the leaf nodes that were trimmed for clarity earlier.

4.6.7 Comparison with Exact Methods

In order to verify and compare our results on acyclic graphs, we implemented the Variable Elimination algorithm of Section 4.2.2 on the simulated graphs that are acyclic. Due to the limitations of Variable Elimination, we implemented it on a series of small simulated graphs from 2 to 26 nodes, and compared the results with our algorithm. The average error of the quantities computed by our algorithm is ± 0.011.

As can be seen in Figure 4.19, Variable Elimination despite being exact scales very poorly with one run on a 24 node graph taking approximately the same amount of time as a 11500

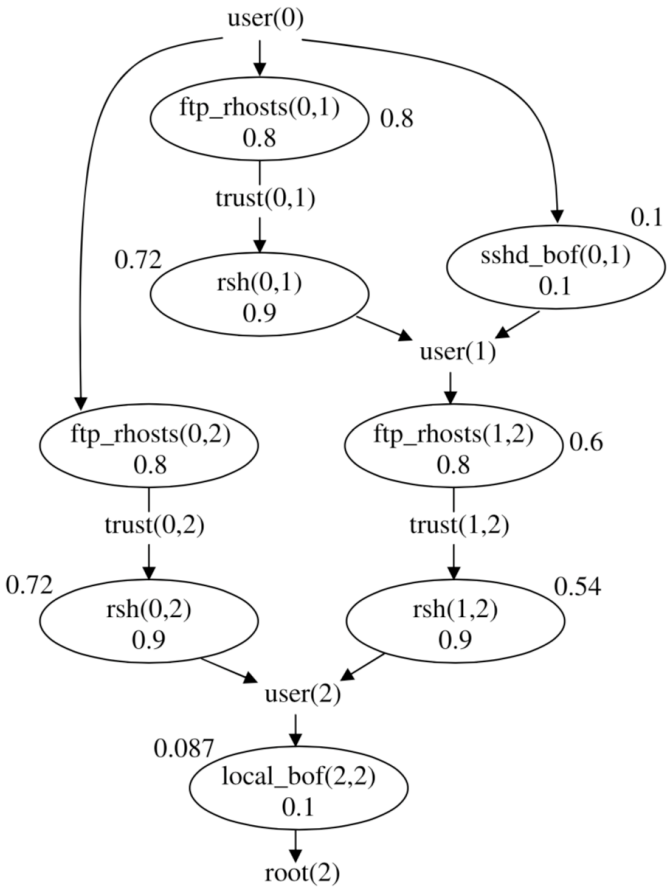


Fig. 4.16 BAG of the network of Figure 4.15 which is acyclic [114].

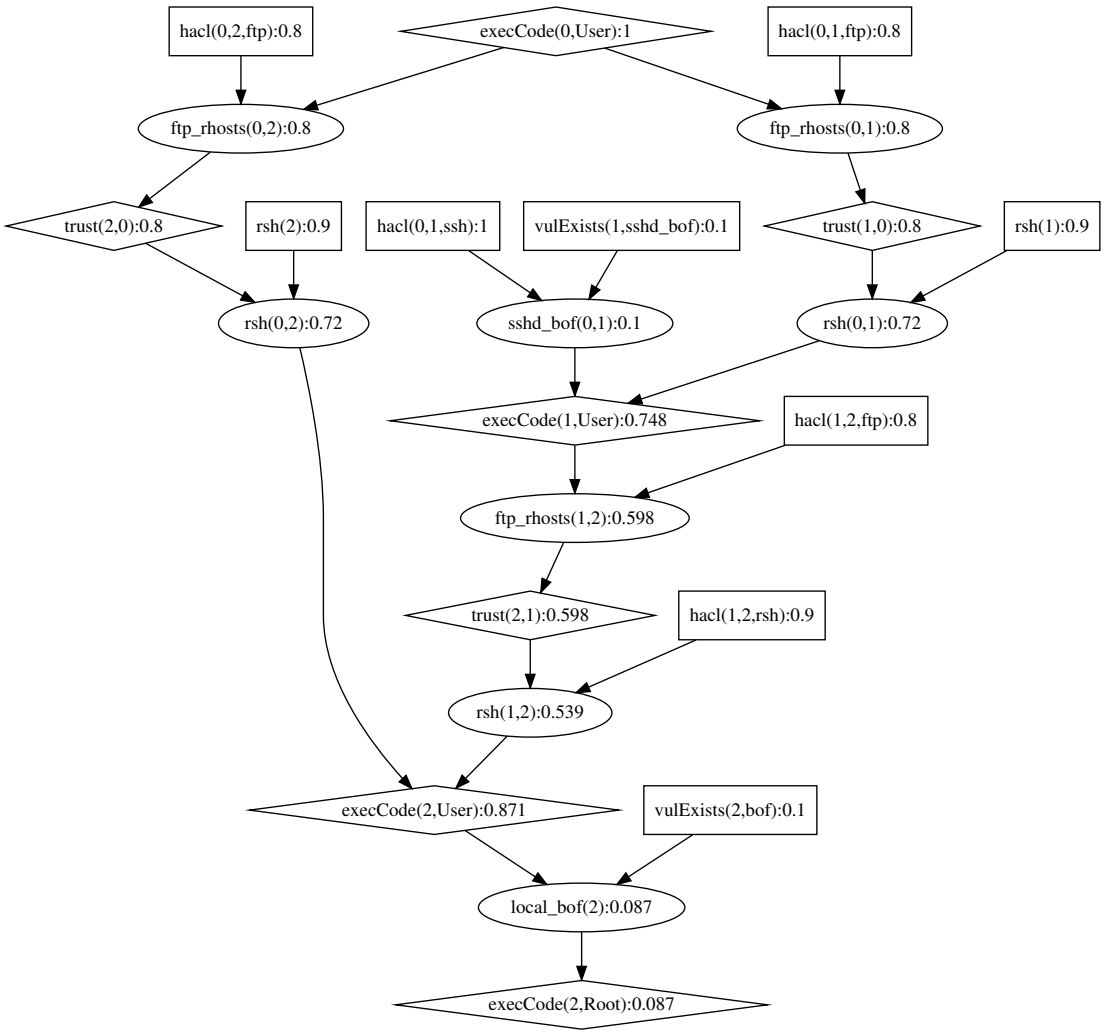


Fig. 4.17 Converted graph from Figure 4.16 using the equivalence shown in Section 2.4.2

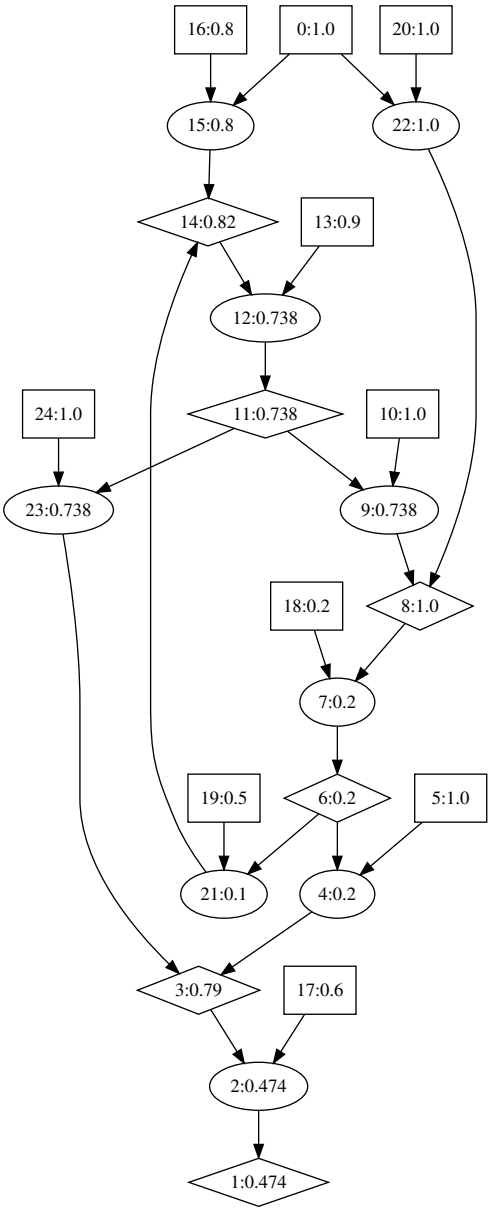


Fig. 4.18 Results of Algorithm 1 applied to the cyclic running example.

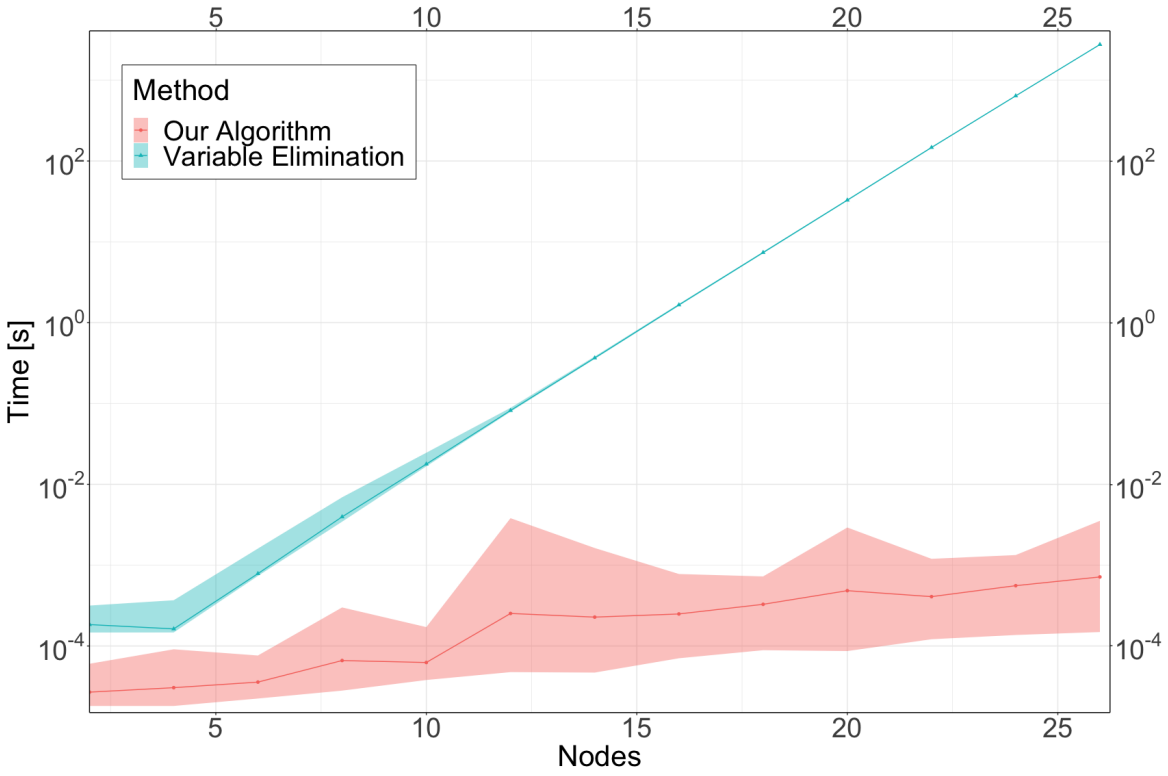


Fig. 4.19 Comparing computational time of our algorithm with Variable Elimination on acyclic graphs.

node run with our algorithm. Due to this poor scaling, the fact that Variable Elimination cannot run on cyclic graphs, and the low average error from our algorithm, our approach is a much better choice for all practical applications.

4.7 Conclusion

In this chapter we have created and demonstrated a systematic approach to analyse Bayesian attack graphs, including those with cycles. Since cycles naturally arise in BAGs that are generated from scanning software (e.g., using MulVAL), it is imperative to establish practical approaches to handle cyclic BAGs. We presented a formal treatment of the problem domain and introduced a solution algorithm that can be applied to any BAG, cyclic or acyclic. This results in a method by which Bayesian attack graphs can be automatically evaluated with respect to what states are available to an attacker and how easily they are reached. Our solution deals directly with the cycles and integrate cycle resolution with the computation of state probabilities without the need for identifying or differentiating the cycle types. Our approach *does not* alter the attack graph by removing edges to make it acyclic. Instead, we preserve all the information in the graph, and no potential attack routes are lost when new data is added to the graph.

Our computational approach is currently restricted to single state probabilities and cannot compute joint probabilities for multiple nodes. A solution that allows the computation of joint probabilities is a next step to further advance the presented approach to cyclic BAGs. Future work will also involve extending the local probability assignment to have a more meaningful value; a temporal metric can be included given that the longer a vulnerability is known about the more likely an exploit has been published for it, increasing the ease of an attack. Scaleable solution algorithms then need to be identified to automate the analysis of these graphs to prioritise fixing of vulnerabilities and identifying most vulnerable network hosts, with regard to their criticality to an enterprise. Finally, although the proposed algorithm can handle large models, for yet larger networks more approximate solutions could be considered.

Chapter 5

Stochastic Simulation Techniques for Inference and Sensitivity Analysis of Bayesian Attack Graphs

A vulnerability scan combined with information about a computer network can be used to create an attack graph, a model of how the elements of a network could be used in an attack to reach specific states or goals in the network. These graphs can be understood probabilistically by turning them into Bayesian attack graphs, making it possible to quantitatively analyse the security of large networks. In the event of an attack, probabilities on the graph change depending on the evidence discovered (e.g., by an intrusion detection system or knowledge of a host's activity). Since such scenarios are difficult to solve through direct computation, we discuss and compare three stochastic simulation techniques for updating the probabilities dynamically based on the evidence and compare their speed and accuracy. From our experiments we conclude that likelihood weighting is most efficient for most uses. We also consider sensitivity analysis of BAGs, to identify the most critical nodes for protection of the network and solve the uncertainty problem in the assignment of priors to nodes. Since sensi-

tivity analysis can easily become computationally expensive, we present and demonstrate an efficient sensitivity analysis approach that exploits a quantitative relation with stochastic inference. These processes enable us to quantitatively determine the vulnerabilities that have the largest effect on the security of important hosts in a network, and so allow for the recommendation of a prioritisation of which vulnerabilities to fix, improving the efficiency of the vulnerability remediation process.

5.1 Introduction

Attack graphs are models of how vulnerabilities can be exploited to attack a network. They are directed graphs that demonstrate how multiple vulnerabilities and system configurations can be leveraged during a single attack in order to reach states in the network that were previously inaccessible to the attacker. An example of such a state would be root privilege on a database that contains sensitive information.

BAGs are particularly promising as a dynamic risk assessment tool where an administrator models new security controls and their effects on a network. A network's most likely attack paths and most vulnerable hosts can be dynamically analysed, and this can be updated dependent on information from an intrusion detection system [91].

Well-defined (that is, acyclic) BAGs can be solved using computational techniques that are well-known from the theory of Bayesian Networks [81]. In recent work systematic approaches have also been proposed for BAGs that have loops and cycles, e.g., [70]. However, direct computational approaches become prohibitively slow if the number of nodes in the BAG is large, and can have large space requirements due to an increase in the size of the cliques in the graphs and their probability tables. Therefore, it becomes important to consider stochastic simulation (Monte-Carlo) techniques.

In this chapter we focus on performing inference and sensitivity analysis on BAGs using stochastic simulation. We do this for dynamic scenarios that do not lend themselves for exact computation, namely scenarios that include observed evidence in the BAGs. We discuss

how any evidence or alterations to the network can be included in the BAG analysis. That is, we create a dynamic model of the security of the network that can be used to deduce an attacker's most likely next move and their route thus far, as well as quantitatively evaluate and compare the effectiveness of different security controls and changes to the network.

While inference for BAGs using stochastic simulation has been performed by others to investigate potential uses [84, 6], there has to date not been a comparison of different techniques' performances on BAGs. In this chapter we employ three stochastic simulations techniques, probabilistic logic sampling (PLS), likelihood weighting (LW), and backward simulation (BS). We evaluate the performance of these techniques in their speed and accuracy as well as how they perform with different quantities of evidence to be included in the graph and different sizes of graph.

The primary outcome of our work is a recommendation of the most efficient simulation technique to use for inference in attack graphs. The recommendation is to use likelihood weighting, which performs well for both low and high evidence scenarios. Moreover, we establish a quantitative relation between stochastic inference and sensitivity analysis of BAGs. We discuss how the methods for including evidence in the graphs can also be used to measure the graphs sensitivity to each vulnerability in the network, and develop a fast approach to calculate these sensitivities without requiring many simulations or any analysis of distributions.

The rest of this chapter is organised as follows. Section 5.2 introduces the running example and lists the problem statements that is the motivation for the work. Section 5.3 introduces and discusses the three sampling techniques that are implemented and their accuracy. Section 5.4 then evaluates the performances of these techniques with regard to accuracy, amount of evidence and size of graph. Section 5.5 introduces our measure of sensitivity and its importance. Finally Section 5.6 compares this work with the current literature available and Section 5.7 presents our conclusions.

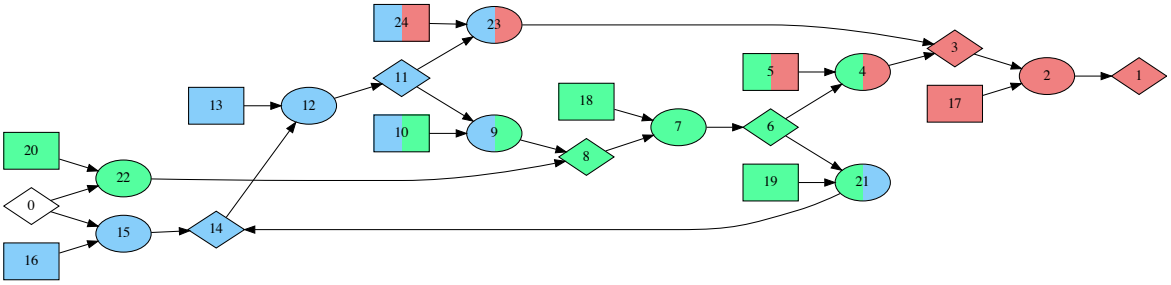


Fig. 5.1 The complete BAG of the small enterprise network presented in Figure 4.1.

5.2 Motivation and Problem Statement

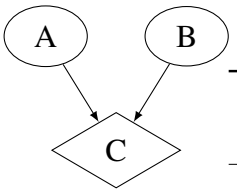
We consider a small enterprise network as a standard example used in the literature [88, 39, 70] to motivate and demonstrate the use of the sampling techniques discussed in this chapter. A full description of the example network, as well as the creation of the corresponding attack graph and the definitions of each of the nodes in the graph can be found in Section 4.2.1. The complete attack graph, including the LEAF nodes, is shown in Figure 5.1.

The following problem statements describe the three main goals that are to be achieved through use of inference on attack graphs.

Problem 1 (Access Probabilities) *Consider an attack graph* $(\mathscr{V},\mathscr{E})$ *with local probability function* $p : \mathscr{V} \to [0,1]$. *Compute* $Prob(X_k = 1)$ *for any* $k \in \mathscr{V}$. *This quantity is called the* access probability *of the node* k *and is simply denoted by* $P(X_k)$. *It will give the likelihood that an attacker will reach node* k *in an attack and will depend on the local probability function* p *and the structure of the attack graph.*

Problem 2 (Inference) *Suppose some evidence of an attack is known in the form of* $\mathscr{Z} = \mathbf{z}$, *where* $\mathscr{Z} \subset \mathscr{V}$ *is the set of random variables associated to the nodes for which we have the respective evidence values* $\mathbf{z}$. *Compute the likelihood that the attacker gain access to node* $k \in \mathscr{V}$ *given such an evidence:* $P(X_k|\mathscr{Z} = \mathbf{z})$.

Problem 3 (Sensitivity Analysis) *The local probability function* $p : \mathscr{V} \to [0,1]$ *is often estimated based on prior knowledge or data on the network. Compute the sensitivity of access probabilities* $P(X_k)$ *and conditional probabilities* $P(X_k|\mathscr{Z} = \mathbf{z})$ *to the values* $p(v)$



		C	
A	B	F	T
F	F	1	0
F	T	1-$p(v)$	$p(v)$
T	F	1-$p(v)$	$p(v)$
T	T	1-$p(v)$	$p(v)$

Fig. 5.2 A small attack graph, with local probability in OR node C.

for any $v \in \mathcal{V}$. If $p(v)$ has a distribution, compute an interval for these quantities with a confidence bound.

We provide stochastic simulation techniques to answer Problems 1 and 2 in Section 5.3, and present a novel solution to Problem 3 in Section 5.5.

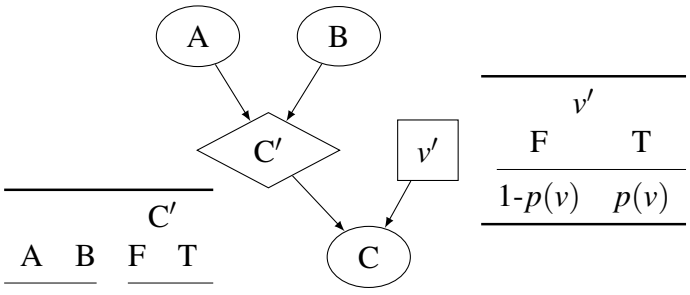
5.3 Sampling Techniques

5.3.1 Graph Decomposition

In order to simplify the process of simulation for attack graphs, we can move all stochastic behaviour to LEAF nodes and in doing so make the rest of the graph purely deterministic. This can be achieved by enlarging the attack graph and moving local probabilities of non-LEAF nodes onto a new LEAF node with the same local probability. This process can be seen in Figures 5.2-5.3, where the node C will have the same behaviour in both figures. This is only required when non-LEAF nodes have local probabilities which is not the case for graphs we create but is found in the literature. Full demonstration of the equivalence is in our previous work [70].

5.3.2 Generating Samples

Using this formalism of BAGs a single attack can be modelled as the array of LEAF nodes being allocated values corresponding to ‘achieving’ something in an attack such that if the



		C′	
A	B	F	T
F	F	1	0
F	T	0	1
T	F	0	1
T	T	0	1

		C	
C′	v′	F	T
F	F	1	0
F	T	1	0
T	F	1	0
T	T	0	1

v′	
F	T
1-$p(v)$	$p(v)$

Fig. 5.3 The equivalent graph of Figure 5.2 with probabilities only on leaves.

node is given the value 1 then the exploit has worked or a condition has been met, and if the value is 0 then an exploit or condition has failed. With these values the reach of the attack can be calculated, as the internal (non-LEAF nodes) in the graph are all deterministically dependent on the LEAF nodes. In this way, with a specific distribution of LEAF nodes a state on the graph is either accessible or inaccessible. With the example of Figure 5.1, a single attack configuration would equate to all the rectangular LEAF nodes being set as *y* or *n*, determining the states of the rest of the nodes in the graph. The attackers ability to reach an important state, like the ability to execute code on the database server at node 1, becomes either *y* or *n*.

We prepare the graph by assigning the LEAF nodes a series of prior distributions based upon factors like the ease of exploitation of a vulnerability. We then sample from these to create a single attack simulation with the LEAF nodes being 1 or 0 according to a random sample of their distribution, and all other nodes being assigned values deterministically from their tables.

5.3.3 Probabilistic Logic Sampling

For our attack graphs, probabilistic logic sampling (PLS) is performed by first sampling a configuration of LEAF nodes. A random number is generated between 0 and 1, if the number generated is less than the prior probability assigned to the LEAF node then the node is assigned a 1 (or y), if it is greater then the node is set at 0 (n). This is repeated for all LEAF nodes to create the configuration. When the configuration has been generated it can be used to prescribe states to the rest of the nodes in the graph. These states are then recorded as an array of 1s and 0s. This process is then repeated until N_s configurations have been generated and evaluated. The recorded arrays can be used to estimate the probability distributions of the nodes in the graph, with $\frac{N(X=1)}{N_s}$ being the estimated probability that an attacker will gain access to a particular node:

$$P(X) \approx \left(\frac{N(X=1)}{N_s}, \frac{N(X=0)}{N_s} \right) \tag{5.1}$$

The simplest way to include evidence with this technique is by discarding any samples that do not conform to the evidence provided. As such one is left with a subset of the original N_s simulations and can calculate the new probabilities in a similar way to equation (5.1).

In order to estimate the probability distribution of the k^{th} variable with regard to the new evidence, $P(X_k|\mathscr{Z} = z)$, using N_s samples with PLS we use algorithm 3 modified from [81]. Here $\mathscr{Z}$ is the variables or nodes that we have evidence for, $sp(X_k)$ is the state space of variable X_k, and $\mathbf{z}$ is the evidence that has been provided for these nodes. This would be in the form of a list of nodes that we know have been accessed created by an intrusion detection system, or a list of nodes that we are modelling as not accessed if we are comparing different security controls and their affect on the network.

5.3.4 Likelihood Weighting

Likelihood weighting (LW) is a method to deal with the problems of PLS for dealing with evidence, namely the inefficiency of generating samples that will be discarded if they conflict with evidence. Instead, for LW, only non-evidence variables are sampled from and as such

Input: Set of nodes $\mathscr{V} = \{1,2,\ldots,n\}$, local probabilities $p : \mathscr{V} \to [0,1]$, number of simulations N_s, evidence $\mathscr{Z} = \mathbf{z}$
Output: Conditional likelihoods $P(X_k|\mathscr{Z} = \mathbf{z})$ for all $k \in \mathscr{V}$
1. Let $(X_1,...,X_n)$ be the associated Boolean random variables.
2. Initialise $N(X_k = x_k) = 0$ for all $x_k \in \{0,1\}$ and all $k \in \mathscr{V}$.
3. **for** $j = 1$ *to* N_s **:**
   a) **for** $i = 1$ *to* n **:**
      Sample a state x_i for X_i using $P(X_i|pa(X_i) = \pi)$, where π is the configuration sampled for $pa(X_i)$
   b) If $\mathbf{x} = (x_1,...,x_n)$ is consistent with $\mathbf{z}$, then $N(X_k = x_k) := N(X_k = x_k) + 1$ for all $k \in \mathscr{V}$, where x_k is the sampled state for X_k
return $Prob(X_k = x_k|\mathbf{z}) \approx \dfrac{N(X_k = x_k)}{\sum_{x \in sp(X_k)} N(X_k = x)}$

Algorithm 3: Performing PLS to approximate a distribution given some evidence.

no simulations are discarded. However this approach causes sampled variables to ignore evidence that is not present in their ancestors, and so an extra weighting has to be introduced. This weighting is equivalent to the probability a certain state will arise given the evidence provided.

Essentially we want to sample from the following distribution,

$$\begin{aligned} P(\mathscr{V},\mathbf{z}) = &\prod_{X \in \mathscr{V} \setminus \mathscr{Z}} P(X|pa(X)', pa(X)'' = \mathbf{z}) \\ &\times \prod_{X \in \mathscr{Z}} P(X = e|pa(X)', pa(X)'' = \mathbf{z}) \end{aligned} \tag{5.2}$$

where $pa(X)''$ are parent nodes that have been instantiated by evidence, and $pa(X)'$ have not. By fixing the evidence variables then taking the sample, we instead are using

$$P_s(\mathscr{V},\mathbf{z}) = \prod_{X \in \mathscr{V} \setminus \mathscr{Z}} P(X|pa(X)', pa(X)'' = \mathbf{z}) \tag{5.3}$$

So to rectify this we weigh each sample taken using

$$w(\mathbf{x},\mathbf{z}) = \prod_{Z \in \mathscr{Z}} P(Z = z|pa(X) = \pi) \tag{5.4}$$

where π is the configuration of the parents specified by **x** and **z**. In order to estimate $P(X_k|\mathscr{Z} = z)$ using N_s samples, we use Algorithm 4 as defined in [81].

Input: Set of nodes $\mathscr{V} = \{1,2,\ldots,n\}$, local probabilities $p : \mathscr{V} \to [0,1]$, number of simulations N_s, evidence $\mathscr{Z} = \mathbf{z}$
Output: Conditional likelihoods $P(X_k|\mathscr{Z} = \mathbf{z})$ for all $k \in \mathscr{V}$
1. Let $(X_1,...,X_n)$ be the associated Boolean random variables.
2. **for** $j = 1$ *to* N_s **:**
  a) $w := 1$
  b) **for** $i = 1$ *to* n **:**
    - Let **x'** be the configuration of $(X_1,...,X_{i-1})$ specified by **e** and previous samples
    - **if** $X_i \notin \mathscr{Z}$ **:**
      Sample a state x_i for X_i using $P(X_i|pa(X_i) = \pi)$, where $pa(X_i) = \pi$ is consistent with **x'**
    else:
      $w := w \cdot P(X_i = z_i|pa(X_i) = \pi)$, where $pa(X_i) = \pi$ is consistent with **x'**
  c) $N(X_k = x_k) := N(X_k = x_k) + w$, where x_k is the sampled state for X_k
return $P(X_k = x_k|z) \approx \frac{N(X_k=x_k)}{\sum_{x \in sp(X_k)} N(X_k=x)}$

Algorithm 4: Performing likelihood weighting to approximate a distribution given some evidence

This is an improvement on PLS as it removes the inefficiency of discarding evidence, instead requiring the calculation of a weight for each simulation. A large number of samples may still be required, however, if the evidence provided is unlikely and therefore the difference between equations 5.2 and 5.3 is large. This means the weighting would in general be very small and as such reaching an amount of error that is not too large may require a large computational time.

5.3.5 Backward Simulation

The final technique is based on the Backward Simulation (BS) method devised by Fung and Del Favero [28]. The primary difference between this and other techniques is that simulation runs originate at the known evidence and the simulation is run backwards. Once this process has terminated the remaining nodes are forward sampled in the standard way. The reason for this is to rectify the slow convergence caused by unlikely evidence.

The backward sampling procedure begins at an evidence node and samples for the parents of the node using the distribution

$$P_s(pa(X_i)') = \frac{P(X_i|pa(X_i)'pa(X_i)'')}{Norm(i)}, i \in \mathscr{V}_b, \tag{5.5}$$

where $pa(X_i)'$ are the uninstantiated parents of X_i and $pa(X_i)''$ are the instantiated parents. The normalising constant $Norm(i)$ is calculated as

$$Norm(i) = \sum_{y \in XP(pa(X_i)')} P(X_i|y, X_{pa(X_i)''}). \tag{5.6}$$

with $XP(pa(X_i)')$ as the set of all possible cases for the uninstantiated parents of node X_i. For our case of n uninstantiated parents with binary values, this set has size 2^n. Once all the ancestors of evidence nodes have been sampled, the forward sweep samples the remaining nodes as

$$P_s(X_i) = P(X_i|pa(X_i)), \quad \text{for all } i \in \mathscr{V}_f. \tag{5.7}$$

The weight for the simulation can be computed as the product of the normalisation constants along with the likelihood of nodes that were set by backwards sampling but were not sampled from themselves

$$w(\mathbf{x}, \mathbf{z}) = \prod_{i \in \mathscr{V} \setminus \mathscr{V}_S} P(X_i|pa(X_i)) \prod_{j \in \mathscr{V}_b} Norm(j). \tag{5.8}$$

As a form of likelihood weighting, BS is designed to cope better with very low-likelihood evidence. A large part of the computational cost of the algorithm comes from the calculation of the normalisation constants, which grows exponentially with the number of predecessor nodes. We would expect this technique to perform similarly to likelihood weighting for few evidence nodes but be an improvement when there are many nodes, as is demonstrated in the paper presenting the technique [28]. However the structure of the graph is of great importance and as such it is difficult to know beforehand which of the techniques will perform better for the application of BAGs.

5.3.6 Confidence Bounds and Convergence

Since all these techniques are sampling from the same distribution once the corrective factors are applied, the standard error can be calculated in a similar way for each. As each trial is random and independent from the last, using the central limit theorem it can be shown that

$$\sigma_{p(x)} = \sqrt{\frac{P\{x\}(1-P\{x\})}{N}} \tag{5.9}$$

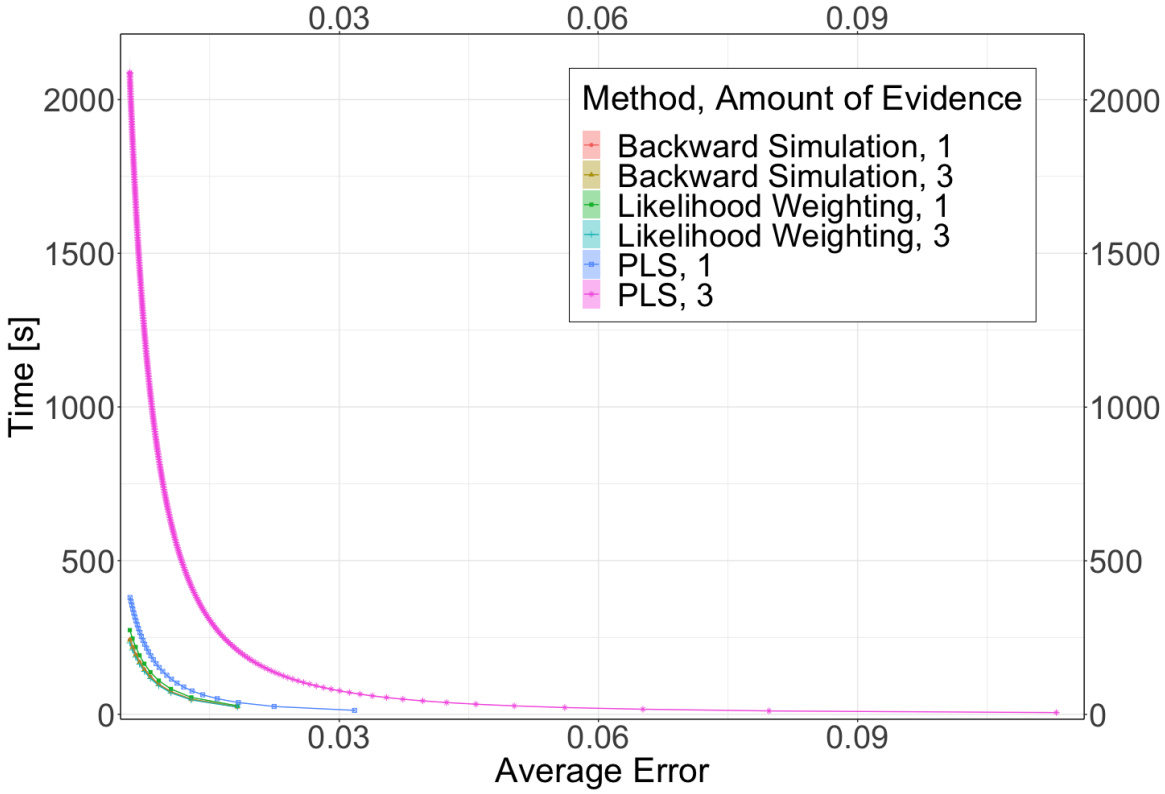


Fig. 5.4 Time against average node error for all techniques for one and three evidence nodes.

All three of the methods also have a guarantee of convergence as shown by Jensen and Nielsen [81] for PLS and LW, and Fung and Del Favero [28] for BS. As all three techniques are forms of importance sampling they inherit the convergence properties of importance sampling, formally proven by Geweke [32].

5.4 Comparison

For the comparison of these techniques, each is first run on a 200 node attack graph from a small enterprise network with varying amounts of evidence. Figure 5.4 shows the increase in time (in wall clock seconds) required for improving the accuracy of results for situations when one and three evidence nodes have been included (the average time over thirty runs has been plotted; the error bars are too small to be drawn for this graph). As can be seen even with just one piece of evidence PLS performed poorly compared to the other methods, with three evidence nodes taking considerable amounts of time and runs with more than three evidence nodes timing out. The other two methods are run with five and ten evidence nodes provided, and the results for this can be seen in Figure 5.5, again with the average result over thirty runs plotted. The minimum and maximum values are shown by the transparent ribbon. While these results are close, interestingly LW does outperform BS at higher quantities of evidence.

Figure 5.6 shows the convergence of each technique on a probability for one of the goals in the network, with the ribbon showing the error of the estimate. LW and BS converge equally quickly with three pieces of evidence but BS does converge faster when only one evidence is used. The techniques are also run across graph sizes of 100 nodes to 1500, again with 30 runs per graph size, with 1, 5 and 10 evidence nodes used. The same technique for generating realistic attack graphs is described in Section 4.6.1, with the full algorithm that is used listed in Algorithm 2. The results of these runs are shown in Figure 5.7 with the points showing the average run time and the ribbon showing the maximum and minimum of the runs. PLS runs slightly worse than the other two techniques for one evidence node but performs very poorly for the other evidence levels so is omitted from the graphs for clarity. The stopping criteria for a run is an error of ± 0.02 per node.

BS performs best with one evidence node, and gives similar results for both 5 and 10 pieces of evidence. LW performs about as well at all evidence levels for graphs below 1000 nodes, but performs best with 10 evidence nodes for graphs larger than 1000.

The methods were also run on graphs of sizes 100 to 5000, with a stopping criteria of an error of ± 0.04 per node. The results for these runs are displayed in Figure 5.8. Figure

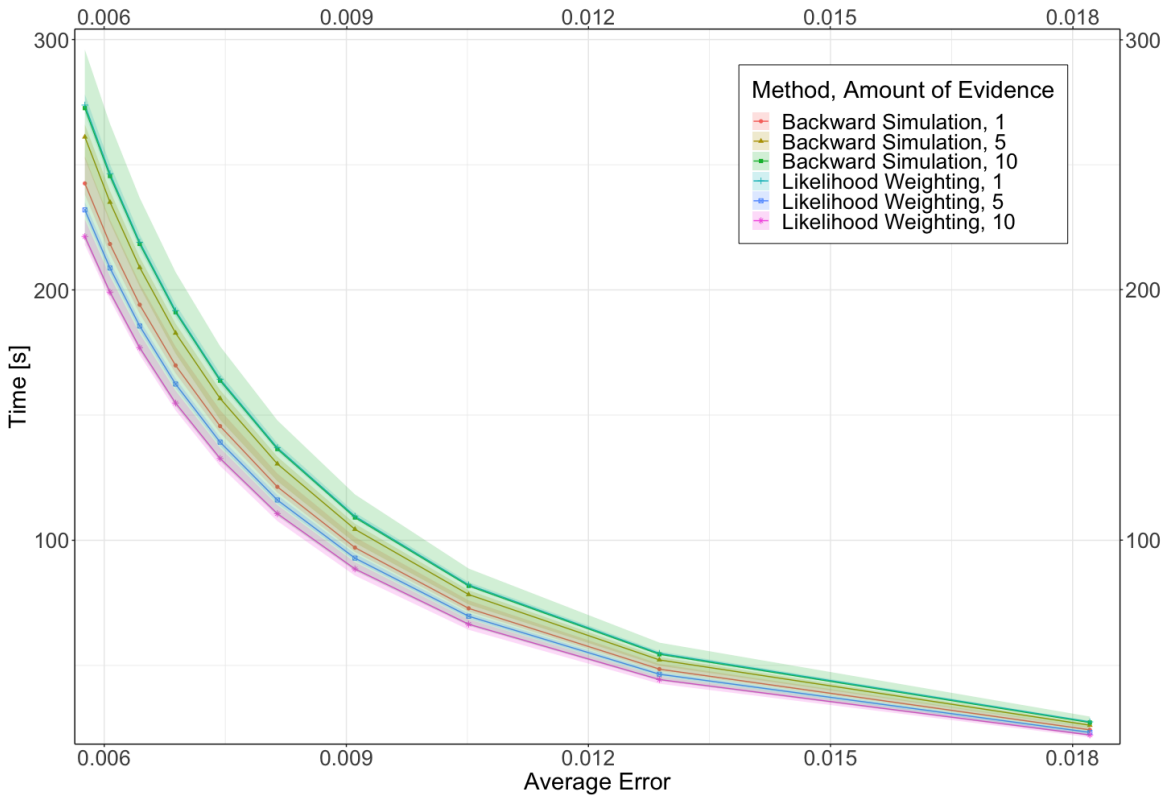


Fig. 5.5 Time against average node error for BS and LW for different numbers of evidence nodes.

5.7 shows that time does not increase by much with an increase in graph size in a few circumstances, and Figure 5.8 even shows an improvement in the time taken for an increase in graph size (for example, 'Backward Simulation, 1' drops below the value for the previous simulation size at 3,650 Nodes in Graph).This variability is due to a method performing better or worse depending on the evidence nodes selected and the shape of the network that was generated.

Given these results, likelihood weighting is the best technique for belief updating in Bayesian attack graphs as it not only performs slightly better overall across different evidence levels and graph sizes but also is easier to implement than backward simulation. Probabilistic logic sampling should only be used if no evidence is expected most of the time.

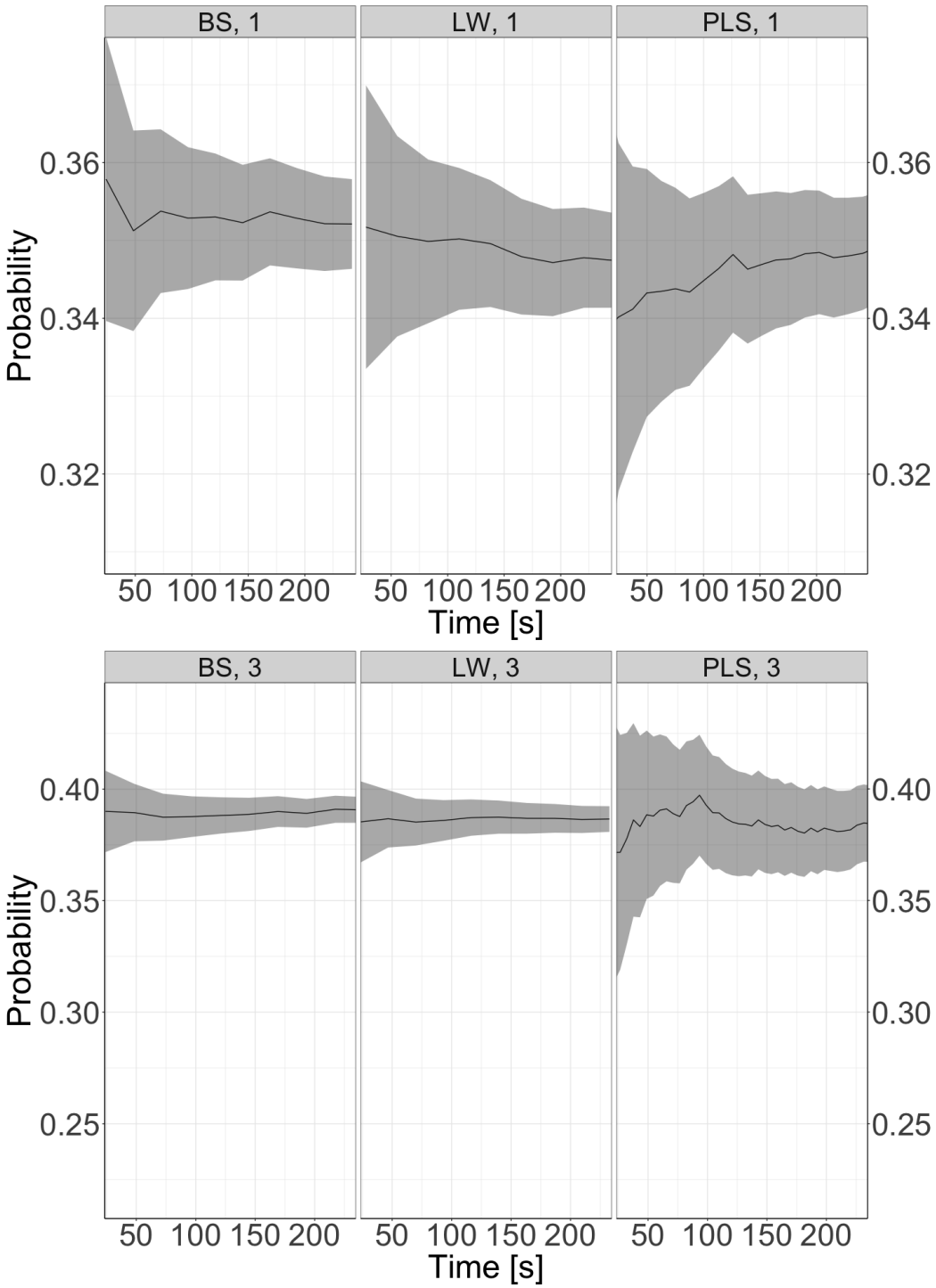


Fig. 5.6 Convergence of goal node probability for all three techniques with one evidence node *(top)* and three evidence nodes *(bottom)*

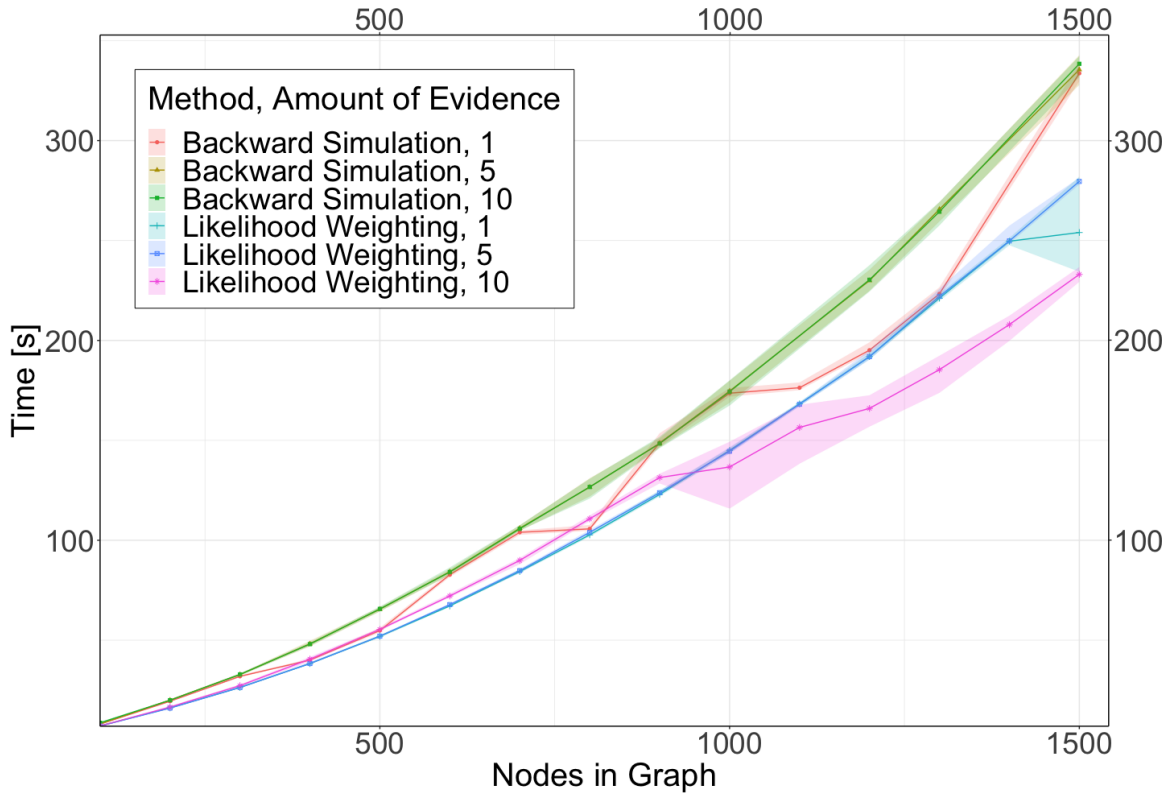


Fig. 5.7 Time required for increasing graph sizes for LW and BS for different amounts of evidence.

5.5 Sensitivity Analysis

The prior probabilities for the vulnerabilities on the LEAF nodes can be generated via different methods of varying complexity. For example Doynikova and Kotenko [22] use various parts of the CVSS vector and Cheng et al. [16] model the relationships of parts of the metrics to give them different weights and improve the accuracy of the probabilities. All these techniques however draw from the data available for a vulnerability which is often incomplete and quickly becomes outdated. Sensitivity analysis is important for the overall analysis of BAGs as it considers the impact of the original assignment of the probability.

To evaluate the sensitivity of the graph to the LEAF nodes, each node can be assigned a uniform probability distribution in turn rather than a single probability. A distribution can be generated for one or several goal nodes in the network with respect to each LEAF node; this is done by sampling from the LEAF nodes' uniform distribution, then generating a

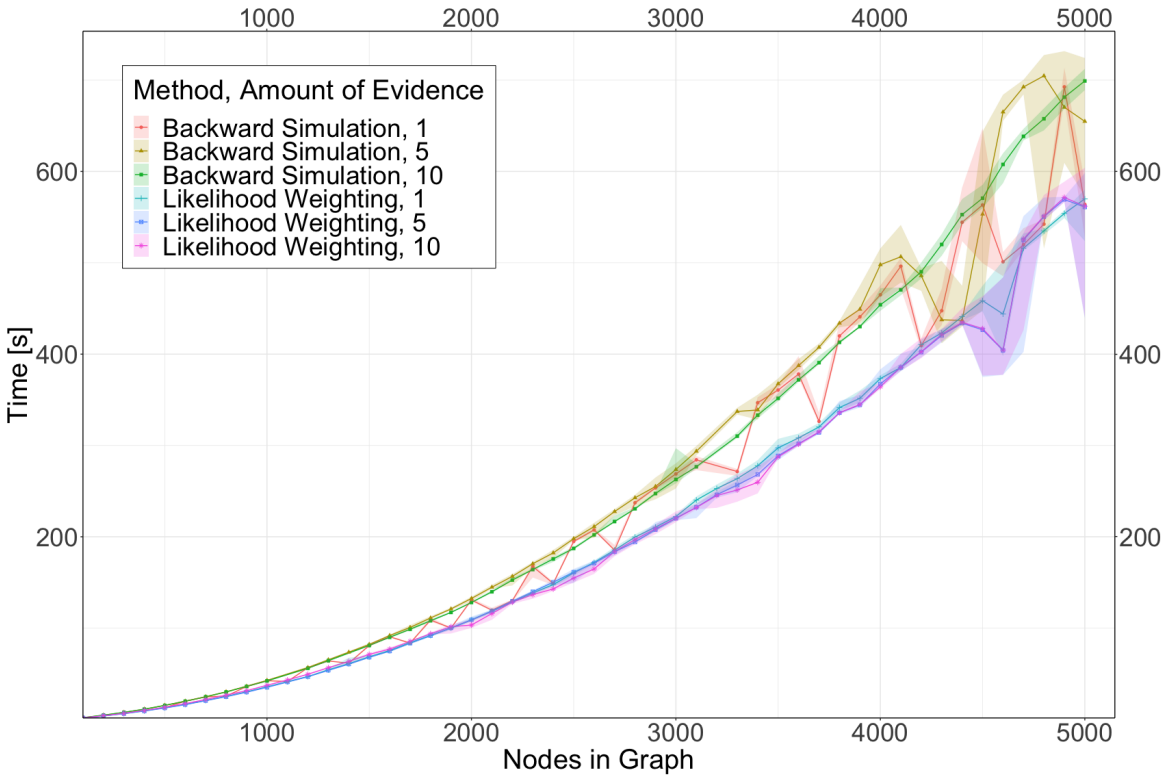


Fig. 5.8 Time required for larger graph sizes for LW and BS for different amounts of evidence.

sample of the entire network as before. The change in the probability density of the access probability of the goal node in the network from Figure 5.1 is shown in Figure 5.9. The wider the distribution the more sensitive the goal node is to the probability applied at the LEAF node, if the LEAF node probability does not affect the goal node at all the probability density would be entirely concentrated at the goal probability value that is calculated when there are only single values for all LEAF nodes.

As such, the network is more sensitive to changes at node 17 or 16, whereas nodes 5 and 10 do not have much of an effect on the goal probability as shown by their narrow probability densities. This type of sensitivity analysis has been performed by others, as discussed in Section 5.6, however, in what follows we propose an alternative technique that requires much less computation and gives more usable results.

Theorem 5.5.1 *In any BAG, access probability of any node* $P(X_k = x_k)$ *is a polynomial function of the local probabilities* $p(v)$, $v \in \mathscr{V}$. *Moreover, for any* $v \in \mathscr{V}$, $P(X_k = x_k)$ *is*

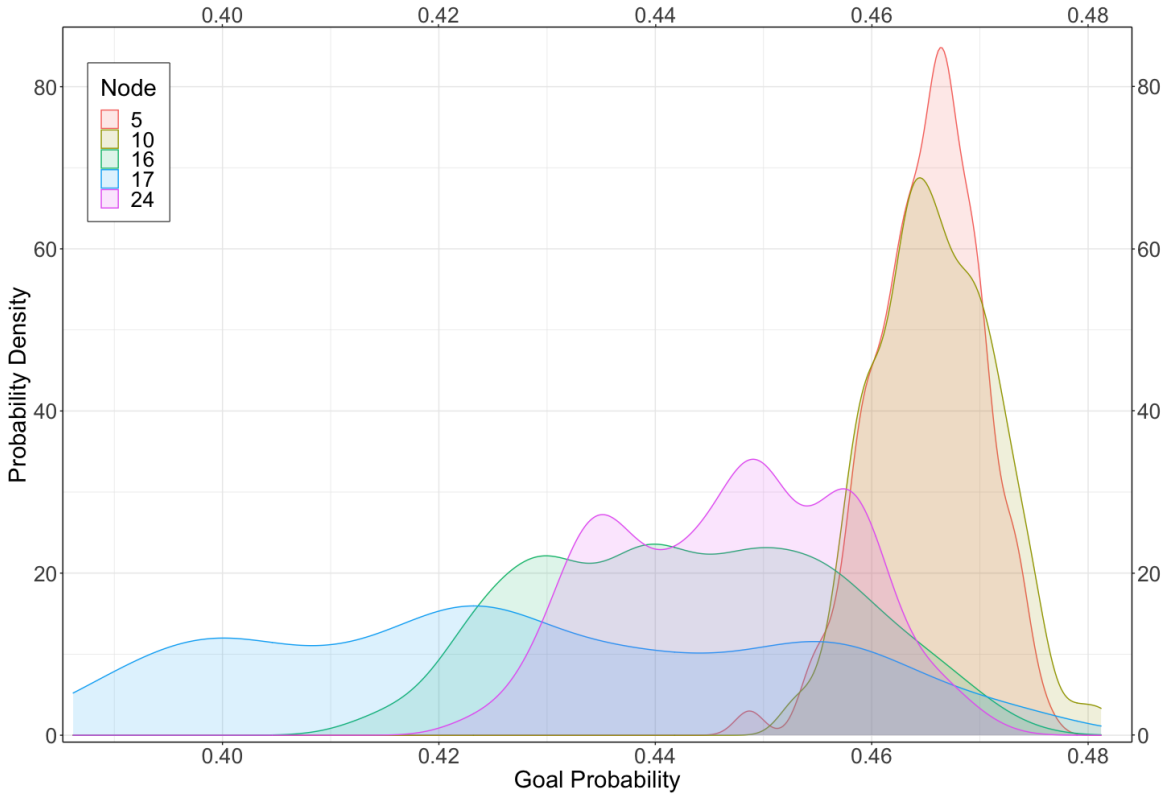


Fig. 5.9 Probability density of goal node when a uniform distribution is used for various leaf nodes, demonstrating their sensitivity.

a linear function of $p(v)$ *if* $p(v')$ *is fixed for all* $v' \neq v$. *The sensitivity of* $P(X_k = x_k)$ *with respect to any local probability* $p(v)$ *is*

$$\left| \frac{\partial}{\partial p(v)} P(X_k = x_k) \right| = |P(X_k = x_k | v = 1) - P(X_k = x_k | v = 0)|, \tag{5.10}$$

which is always in the interval $[0,1]$. *More generally, for any evidence* $\mathscr{Z} = z$,

$$\begin{aligned} &\left| \frac{\partial}{\partial p(v)} P(X_k = x_k | \mathscr{Z} = z) \right| \\ &\qquad = |P(X_k = x_k | \mathscr{Z} = z, v = 1) - P(X_k = x_k | \mathscr{Z} = z, v = 0)|. \end{aligned} \tag{5.11}$$

Both sensitivities in (5.10)-(5.11) *can be estimated using the efficient simulation techniques presented in this chapter.*

Proof 5.5.2 *The proof is inductive by sequentially conditioning the access probability to the LEAF nodes in the BAG. The access probability $P(X_k = x_k)$ can be written as*

$$\begin{aligned} P(X_k = x_k) &= P(X_k = x_k|v = 0)P(v = 0) + P(X_k = x_k|v = 1)P(v = 1) \\ &= P(X_k = x_k|v = 0)(1 - p(v)) + P(X_k = x_k|v = 1)p(v). \end{aligned} \tag{5.12}$$

Note that $P(X_k = x_k|v = 0)$ and $P(X_k = x_k|v = 0)$ are independent from $p(v)$. This gives linear dependency to $p(v)$ and makes $P(X_k = x_k)$ a polynomial function of all local probabilities. Moreover, the sensitivity with respect to $p(v)$ is: Sensitivity $= |P(X_k = x_k|v = 1) - P(X_k = x_k|v = 0)|$.

The sensitivities calculated in this manner are shown in Table 5.1, and using this sensitivity value allows quick evaluation of the importance of each node without the extensive computation or the required analysis of the probability distribution that is necessary to generate and interpret Figure 5.9. The information remains the same, however, with node 17 being the most important followed by 16 then 24, while nodes 5 and 10 have very little impact on the goal node probability. Both techniques for investigating sensitivity are also timed, with a ± 0.02 node error, and the 'on/off' technique took 59s while the original method took 1581s. With this result significantly reducing computational cost for sensitivity analysis, future work should examine the process of using this information to improve the accuracy of attack graphs even with imprecise CVSS scores.

Table 5.1 Sensitivities calculated using 'on/off' evidence.

Leaf Node	**Sensitivity**
17	0.7780
16	0.4388
24	0.3526
5	0.0225
10	0.0081

5.5.1 Sensitivity Analysis for Network Security

This process for evaluating the sensitivity of the Bayesian network to the LEAF nodes on the network can also be used to improve network security. Firstly, if the goal nodes on the network are known and sensible prior probabilities have been assigned to the graph, the process can be used as a tool for prioritisation of network vulnerabilities, as the LEAF nodes with the highest sensitivities will be the vulnerabilities that have the largest effect on the insecurity of the goal nodes in the network. With multiple goal nodes, this process will incorporate the fact that one vulnerability can allow routes to multiple different goal nodes and so may be a higher priority vulnerability.

The sensitivity analysis process can also be applied to modelling network topologies and controls and quantifying a change's effect on security. For example, any node in the attack graph that represents a connection between two of the hosts or an open port could be turned on or off with sensitivity analysis to investigate the effect that blocking various types of network traffic would have on network security. In order to fully understand the benefit, the cost of making the restriction should also be evaluated and so an appropriate cost model should be included in the analysis. An example of using a sensitivity analysis with a risk and cost analysis can be found in [84], where the authors investigate the cost and benefit of blocking ssh and rsh traffic between two hosts, compared to not changing the network.

5.6 Related Work

Baiardi and Sgandurra use Monte Carlo simulations in their Haruspex tool [6]. This tool is a fully featured program that uses attack graphs and threat agents to model security. It is an application for this type of graph, incorporating many different elements, but does not analyse different methods for simulation. Another example of stochastic simulation techniques for attack graphs is by Noel and Jajodia [84]. They use PLS to compare different security fixes for a network. However this is performed by hand and as such it cannot be generally applied. Their use case compares several security controls that could be added to the network. This is achieved by examining the resulting distributions estimated when the changes are applied

to the graph, in a manner similar to that shown in Figure 5.9 as a sensitivity analysis. As discussed in Section 5.5, this requires more computation and also requires analysis of the resulting distributions.

Muñoz-González et al. present an exact method for inference in BAGs using the junction tree algorithm [78]. This method is attractive due to its exact nature, but unfortunately is very limited in its application due to how it scales. This is caused by the requirement for tables to be generated based on the cliques created to start the calculations, and for large graphs these tables can become extremely large. It is better to have a trade-off in the accuracy of the method to reduce the space required, to allow scalability for the large graphs that are expected from enterprise networks. They go on to present an approximate technique in [77] using loopy belief propagation. The results of this scale well, linearly with respect to the number of nodes, while achieving a reasonable level of accuracy. The drawback to using this method, unlike stochastic simulation, is that there is no guarantee of convergence to the correct value.

5.7 Conclusion

In this chapter we have presented and compared three techniques that can be generally applied to inference of any Bayesian attack graph. We make the recommendation that for most purposes the likelihood weighting process is a good choice to analyse an attack graph when any amount of evidence is presented, in a timely fashion. We also demonstrate a test of sensitivity for the graph that can be very quickly calculated and does not require any complex analysis of distributions or prior sampling of node distributions. This can be used both as remediation for the high uncertainty in LEAF node prior probabilities, as well as an easy prioritisation of vulnerabilities in light of their importance to a series of goal nodes.

Chapter 6

Attack Graph Generation Platform

In this chapter we discuss the process of building attack graphs from vulnerability scans. We first introduce the tools and technologies used in this process and then introduce and explain our fully containerised automated pipeline for the automatic generation of attack graphs from vulnerability scans, in fulfilment of Contribution 6 in the Abstract. While attack graph generators are available, the most commonly used generator, MulVAL [86], has many prerequisites and does not run out of the box. Furthermore the scans that can be processed need to be generated by proprietary software. We incorporate the generator in a complete platform that can be run on any system, and include a scan translator to allow for the use of open-source vulnerability scans, making attack graph generation accessible to anyone. The graphs can also optionally be rendered using the graphing container, allowing for a customisable visual output alongside the normal attack graph files. A readily deployed software pipeline that enables users to generate attack graphs and visualise their network security from a vulnerability scan is a fast and effective route to understand a network's security, and enables a full computational analysis of the vulnerabilities in the network by making the attack graph available for processing using the methods described in Chapters 4 and 5.

6.1 Introduction

In order to replicate any of the literature on attack graphs, the first requirement is the ability to generate an attack graph. While there are several attack graph generators available online, there is currently only one non-commercial generator that is open-source [1], which is MulVAL. MulVAL, or "Multi-host, multi-stage Vulnerability Analysis Language", is an open-source application used for the generation of dependency attack graphs. It is also the generator that is used predominantly in the attack graph literature, and is praised in Yi et al.'s overview of attack graph generation software [123]. As such it is the natural choice of engine for an attack graph generation platform.

MulVAL, however, has many prerequisites and dependencies for installation, and also requires some specific changes to the operating system in order to run. These can clash with existing settings, and prevent users from being able to run the generator, as can be seen from the many requests for technical help online. It also only accepts specific scan types that do not come from an open-source scanner. As such, it is a contribution to the field to build a platform that easily and automatically processes scans, converting them where necessary, and that runs on any operating system without difficulty.

6.2 Design of the Platform

6.2.1 Requirements

In order to enable further research for attack graphs and their visualisation, as well as provide a simple and consistent method for network vulnerability analysis using attack graphs, the platform must fulfil various requirements.

Functional Requirements

1. The platform must be able to process a variety of vulnerability scan formats as input, including at least one open source format.

2. The platform should be able to be deployed on all major operating systems
3. Generated attack graphs should be rendered to allow visual inspection of results.
4. The resulting attack graph should be output in an easily used format for further analysis and research.
5. The visualisation process should be modifiable to allow custom visualisations and the incorporation of other data.

Nonfunctional Requirements

1. The platform should be easily run by anyone.
2. The platform should be able to be run and stopped using a single command.
3. The databases used in the platform and the software packaged with it should be automatically updated.
4. Once a vulnerability scan is provided, processing should be performed automatically and no further user input should be required.

6.2.2 Use Cases

The platform is designed with three primary use cases in mind, shown in Figure 6.1. A system administrator using the platform should be able to generate an attack graph for their network after using one of a variety of vulnerability scanners. This attack graph should be visualised to allow them to examine it, as well as output in a format that allows them to use analysis techniques to determine high priority vulnerabilities. An attack graph researcher has a similar use case, but instead of using a visualisation will want to export the attack graph to then run their proposed form of attack graph analysis (for example by transforming it into a Bayesian network). Finally the visualisation researcher would use the customisable visualisation module to design and implement their own visualisation techniques, whether

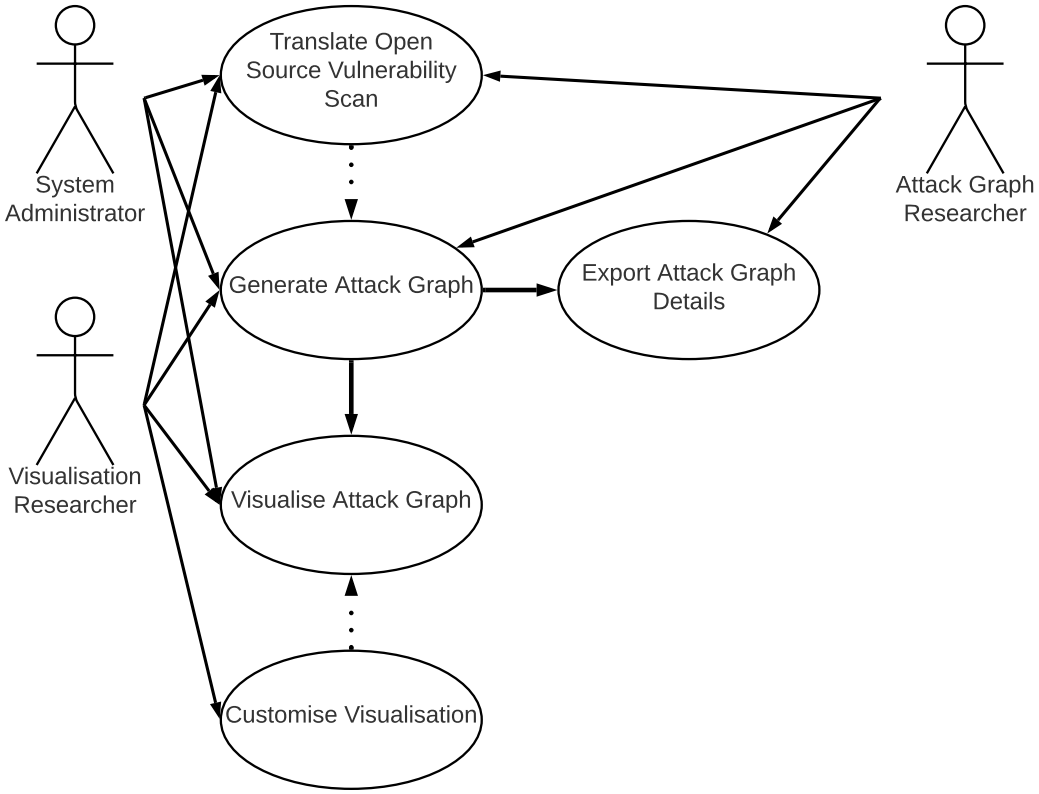


Fig. 6.1 Use cases of the platform

that involves bringing in a custom data source to add information to the graphs or designing a new visualisation module to examine specific properties of the graph more closely.

6.2.3 Layout

The overall layout of the workflow for the platform to enable each use case is shown in Figure 6.2. A vulnerability scan from either an open source scanner or a proprietary scanner is taken as input. From this input, an attack graph is generated, and is then rendered into an image using the default settings or by incorporating custom processes and optional supplementary information. The rendered attack graph is then given as output, along with files that describe the attack graph.

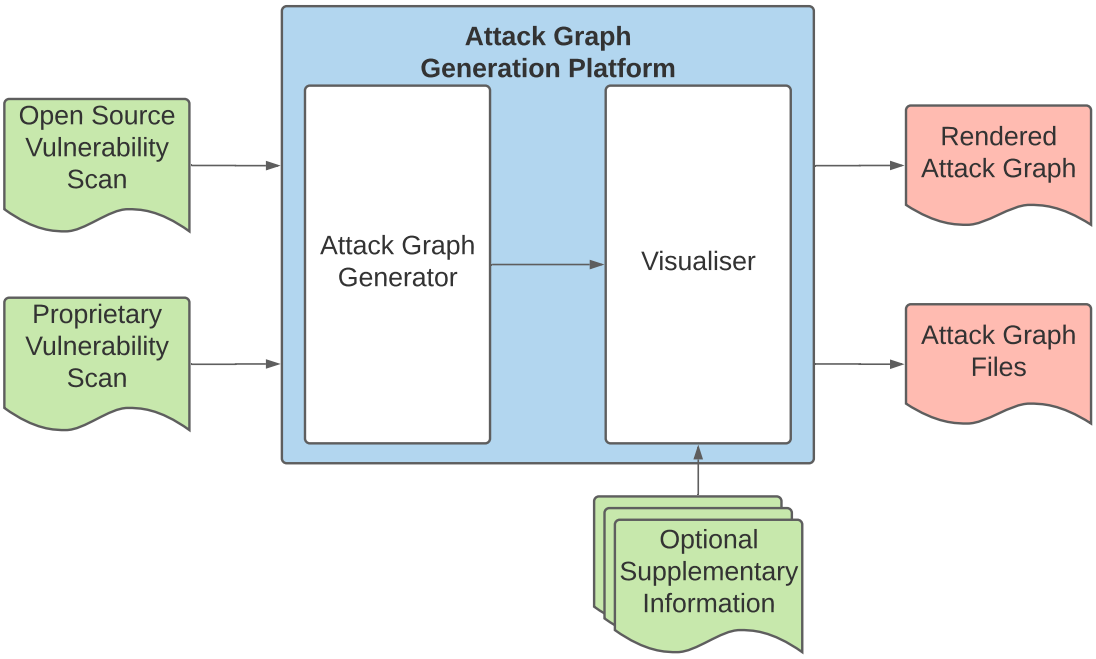


Fig. 6.2 Layout of the platform's workflow

6.3 Software

6.3.1 Docker

In order to achieve both the goal of making attack graph generation available to as many people as possible, and making the installation and use simple, some form of virtualisation should be used. Virtualisation is the process of running an isolated user space within a real machine that behaves as if it were a real machine itself. As such, if a program can be installed, set up and run in a virtual environment then any machine that can install the virtual environment can use it to run the program. Therefore, by creating an environment with our desired attack generator pre-installed and runnable, anyone that is able to install the virtual environment software can generate attack graphs.

To achieve this goal we use the tool Docker [74], due to its ability to run on all major operating systems (Windows, macOS, and Linux) as well as its popularity. Docker is a virtualisation tool that allows any requisites for an application to be included in a single

package called a container. All the necessary aspects of an operating system needed to run an application can be included in a container, thus removing the need to run and maintain multiple virtual machines to allow interaction between applications. Because many of the programs we are using require specific set-ups in order to run, Docker is a useful tool to allow us to run these programs with minimal overheads. In order to correctly install and set up our software we can also make changes to system paths and files without needing to alter the host machine.

A Docker application is constructed by executing a series of instructions on an image. The image of an operating system is taken and virtualised, and then a `Dockerfile` is written to modify the image from its starting point. As such, no matter what the host operating system or setup is, anyone installing the application will have the same image downloaded and the same instructions and modifications performed to create the final container, and so the application will run identically no matter the environment.

We use an installation of Ubuntu version 16.04, available from [112] as the basis of the Docker container.

6.3.2 Docker Compose

Docker Compose is a tool that allows the creation of applications built from otherwise separate Docker containers. This means that all the containers can be controlled through a single command, and the containers can communicate on a Docker network. This is how we build and control the platform, and allows the installation and setup of each of the Docker containers to be completely automated. Every part of the platform can therefore be built and controlled from one interface, and any output from a container can be processed by another container.

6.3.3 MulVAL

MulVAL [86][85], or "Multi-host, multi-stage Vulnerability Analysis Language", is an open-source application for the generation of dependency attack graphs. There are many solutions

available but MulVAL is one of the most popular for use in research, is open-source, and is praised in Yi et al.'s overview of attack graph generation software [123].

MulVAL takes in network scans, only Nessus and OVAL adapters are available to convert into the necessary Datalog format, and generates an attack graph from the vulnerabilities and network information contained in the scans. In order to make the whole process open-sources, as well as widen the number of scans that can be processed, we construct our own adaptor for OpenVAS reports. It can also generate a PDF of the complete attack graph but for most non-trivial cases this PDF will be much too large to be useful. These graphs can then be analysed using the Bayesian network based techniques discussed in Chapters 4 and 5 of this thesis for a probabilistic interpretation, and can also be visualised in order to gain some basic insights into the overall structure and security of the network the scan comes from. We build this visualisation process into the platform.

The code for MulVAL is freely available online. However there are numerous software prerequisites and setups that are required in order to run MulvVAL, as well as several required operating system settings. Listing 6.1 is the dockerfile that the reader can use to build a container in order to have a working version of MulVAL. Once the container is running, the user can run `docker ps` to discover the name of the container, and then execute `docker exec -it <container_name>` in order to enter the container in interactive mode with a pseudo terminal. Once the container has been entered, the standard MulVAL usage instructions [87] can be used to generate attack graphs - first by running the adapters on an input file and then by running the graph generation script `graph_gen.sh` with any desired options on the output of the adapters.

Listing 6.1 Docker container set up enabling MulVAL to run on any operating system

```
FROM ubuntu:16.04

# Set MySQL credentials
RUN echo mysql-server mysql-server/root_password password root |
    debconf-set-selections && \
echo mysql-server mysql-server/root_password_again password root |
    debconf-set-selections

# Install dependencies
RUN apt-get update && apt-get install -y \
autoconf \
bison \
flex \
```

```
g++ \
graphviz \
libpcre3 \
libpcre3-dev \
libcurl3 \
libcurl4-openssl-dev \
make \
mysql-server \
openjdk-8-jdk \
texlive-font-utils \
vim \
wget \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*

# Use XSB workarounds, install XSB, add keywords
WORKDIR /usr/lib/jvm/java-8-openjdk-amd64/include
RUN cp linux/jni_md.h .
ENV JAVA_HOME /usr/lib/jvm/java-8-openjdk-amd64
WORKDIR /root
RUN wget http://xsb.sourceforge.net/downloads/XSB360.tar.gz && \
tar -xzf XSB360.tar.gz && \
rm XSB360.tar.gz
WORKDIR /root/XSB/build
RUN ./configure && ./makexsb
ENV PATH ${PATH}:/root/XSB/bin/
ENV XSBHOME='~/XSB/'

# Install MulVAL
WORKDIR /root
RUN mkdir mulval && \
mkdir mulval/bin && \
mkdir mulval/bin/adapter && \
mkdir mulval/bin/metrics
ADD mulval.tar.gz /root/mulval/
WORKDIR /root/mulval
ENV MULVALROOT /root/mulval/
ENV PATH ${PATH}:${MULVALROOT}/bin:${MULVALROOT}/utils
RUN make

# Create input directory
RUN mkdir /input && cp /root/config.txt /input/
WORKDIR /input
VOLUME ["/input"]

# Populate MySQL with NVD data and change file permissions
WORKDIR /root
RUN echo "jdbc:mysql://localhost:3306/nvd\nroot\nroot" > \
config.txt
RUN echo "#!/usr/bin/env␣bash" > \
createDatabase.bash && \
echo 'service mysql restart\nsleep 5\nmysql -uroot -proot -e "create␣
   database␣nvd"' >> \
createDatabase.bash && \
echo "nvd_sync.sh\nexit␣0" >> \
createDatabase.bash && \
```

```
chmod +x createDatabase.bash && \
./createDatabase.bash
RUN rm -rf nvd_xml_files
RUN echo "#!/usr/bin/env␣bash" > \
startSql.bash && \
echo "service␣mysql␣start\nexit␣0" >> \
startSql.bash && \
chmod +x startSql.bash

# Prepare container for the compose network and set entrypoint
ENV SCPT='./mulval/utils/nessus_translate.sh'
RUN echo $SCPT
RUN tail -n +2 "$SCPT" > "$SCPT.tmp" && mv "$SCPT.tmp" "$SCPT"
RUN sed -i '1s/^/#!\/bin\/sh\ncp ~\/config.txt .\/config.txt\n/' "
    $SCPT"
RUN chmod 775 $SCPT
ADD scripts.tar.gz /root/
RUN chmod 775 /root/scripts/openvas_translate.py
RUN chmod 775 /root/scripts/process_dot.py
ENV PATH /root/scripts:${PATH}
RUN python /root/scripts/get-pip.py
RUN pip install mysql-connector-python mysql-connector-python
RUN pip install pydot
RUN pip install graphviz
RUN chmod 775 /root/scripts/compose.sh
RUN chmod 775 /root/scripts/compose-generate.sh
WORKDIR /input
ENTRYPOINT /root/createDatabase.bash && /root/startSql.bash && compose.
    sh
```

6.3.4 OpenVAS and CyberScore

Open Vulnerability Assessment System [35] (OpenVAS) is a software package that is used for vulnerability discovery and management. It is developed by Greenbone Networks and is freely available and open-source. It functions as a complete agentless vulnerability scanner with the ability to run both authenticated and unauthenticated scans, test protocols implemented in the network, and run custom vulnerability tests using user-written scripts sing the NASL language (cf. Section 2.2.3 for an explanation and discussion of these terms). While we do not install OpenVAS in our platform for attack graph generation and inspection, it is relevant to the building of the pipeline as it is an open-source method for performing vulnerability scanning, and it is not currently supported by the MulVAL attack graph generation engine. As such, by modifying MulVAL to include an adaptor for OpenVAS

scans we enable an entirely open-source path for attack graph generation - from scanning to generation, processing and inspection.

Figure 6.3 shows the architecture of OpenVAS. The OpenVAS scanner, as shown in the figure, is fed by a library of Network Vulnerability Tests, or NVTs, that are updated daily. There are currently over 93,000 network vulnerability tests [36] (updated May 2021) available for use automatically within OpenVAS's built-in repository. These NVTs are scripts written in NASL, or Nessus Attack Scripting Language, that automatically attempt to exploit a vulnerability in order to detect if it is present. They are executed against each scan target by the scanner when a network vulnerability scan is started, and can also detect what software, services and policies are available on a machine. We also use XQ's CyberScore vulnerability scanner, which is built from OpenVAS and has a custom library of NVTs developed by XQ (as well as having different reporting mechanisms and a different front-end and installation procedure). We have successfully tested our platform against OpenVAS and CyberScore scan reports, as well as reports from Nessus [110] and reports written in the Open Vulnerability Assessment Language (OVAL). OVAL is an open-source community effort to create a standard language and formatting for the reporting of machine states and information [14]. This means we have a breadth of input options including two open-source options.

6.3.5 Gephi

Gephi [31] is primarily a network visualisation tool that is open-source. It is written in Java, has a Java API, and can be run in a docker container. It is therefore a good choice for us to use for generating easily digestible images to explain the attack graphs that are the output of the MulVAL graph generator container, as it is open-source and well documented, as well as running on all common operating systems. We include Gephi in our platform to enable visualisation of the attack graphs that are generated, but only implement a few simple processing techniques to demonstrate simple use-cases of the Gephi container with attack graphs (cf. Section 6.4.3). Gephi comes with a parser for the DOT [34] graph description

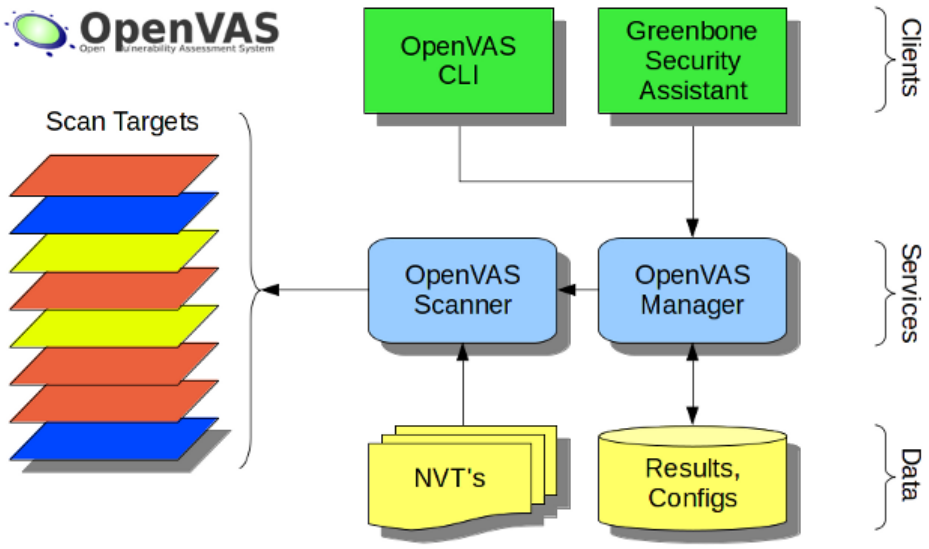


Fig. 6.3 Structure of the OpenVAS scanner software, using Network Vulnerability Tests (NVTs) that are updated daily [113]

language, and so we can use the MulVAL container to process the attack graphs into DOT files and then import them into Gephi.

We demonstrate three usage examples for visualisation of the information that is produced by MulVAL, that we package with the platform. The platform can be used to implement and test other visualisation techniques in an automated open-source pipeline. By way of an example, Homer et al. [38] discuss two visualisation techniques that could be implemented within the Gephi platform in order to improve the readability of attack graphs. They discuss identifying less important areas of large graphs that can be trimmed for user clarity, and the concept of finding instances of similar attacks and grouping them into single virtual nodes. Gephi can also be used interactively, and so a process like that described by Noel and Jajoda [83], where attack graph complexity is reduced through the use of a visual aggregation hierarchy, could be implemented to simplify user interaction with the attack graph.

6.3.6 CVEs

CVEs (common vulnerabilities and exposures) are collected in many different databases. One of the most popular, the NVD or National Vulnerability Database, is maintained by NIST (the National Institute for Standards and Technology). The database stores metadata

concerning each vulnerability including it's CVSS (Common Vulnerability Scoring System) score. CVSS scores are a way to label and sort vulnerabilities, as well as convey information about them in the form of vectors that encode things like the ease of exploitation and severity if exploited. Refer to Section 2.2 for a full discussion of vulnerabilities and an explanation of their scoring.

6.4 Development and Architecture of the Platform

In order to demonstrate the construction and intended use of our platform, we consider the case of the processing of an OpenVAS scan. There are two reasons for using this specific case: it is the most complex of the cases as there is no native parser for OpenVAS scans within MulVAL, and it is the use case that is completely open-source and so we anticipate it being the most common one. Alternative use cases would be importing a scan from the Nessus scanning suite or using a scan that uses OVAL notation, which are similar processes but both have an adaptor included in MulVAL so do not require any preprocessing. Other than the preprocessing step, the workflow for any scan will be identical.

The code for the platform is available from [69]. The only prerequisites for running the platform are that Docker and Docker Compose must be installed. With these two programs present, the platform can be setup and run by running the `setup.sh` script. Once installed, the platform can be brought online by running `run.sh`. Once the platform is running, any files placed into the input folder will be automatically processed. To take the platform offline run the `stop.sh` script. Each script comes with several runtime options (verbose mode, setup options, updating of databases) which can be explored by appending `-h` or `-help` when running the script.

6.4.1 OpenVAS Translation

The first requirement of the platform to be able to process OpenVAS scans is to translate them for MulVAL to be able to use them. In order to achieve this, MulVAL is put into a Docker container that has a Python script that loads in each OpenVAS report and processes

it into a form MulVAL can use. The script works by iterating through each host result in the xml blob that is loaded into Python. From each result it takes the IP address of the host, along with extra information like the vulnerability CVE IDs and the ports and protocols used. With this information it creates a CVE dictionary. With the dictionary built, the script uses the same MySQL database as MulVAL to correctly name and sort the vulnerabilities, as well as removing vulnerabilities that have been incorrectly included by the scans, and then all the information is rebuilt into a file with the correct format. By reusing the database, we ensure that all of the vulnerabilities will be understood by MulVAL, and we can also package the translator and MulVAL generator into a single container for more space efficiency and easier setup.

6.4.2 Processing of Dot Files

After MulVAL has generated an attack graph in the form of a dot file, the files needs to be processed before being imported to the graphing software. This is because the attack graph contains a lot of extra nodes and information that is not necessary to plot the topographical structure of the network or where the vulnerabilities are located. Note that this information is needed for the future analysis of the attack graphs which means that the original dot is also saved alongside a new processed dot. A python script was written to load in the dot file and understand the graph, meaning that removing nodes does not leave trailing pathways on the graph as the edges and nodes that are removed are replaced with edges between the nodes that were either side of the unnecessary node.

6.4.3 Gephi Graph Processing

We now demonstrate three usage examples of the Gephi container. It is important to note that this container is intended to be accessed by the user to run custom visualisation programs (cf. Section 6.3.5) and that these examples are primarily meant as a non-exhaustive show case of the possibilities of processing attack graphs within Gephi.

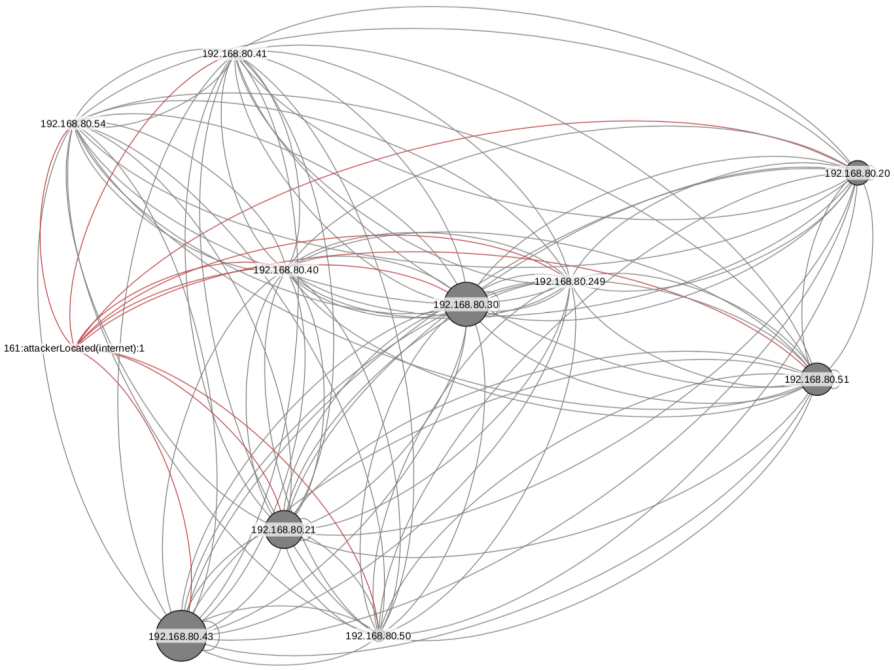


Fig. 6.4 Plot of the host's connections with scaled nodes

The processed dot file is loaded into Gephi to generate a graph. This is done automatically using Gephi's Java API. The Java application that was created runs analysis on the nodes in the graph and uses this information to create different graphs. The first iteration is the simplest interpretation of the data, and simply plots the connections between the hosts and the external attacker, with the size of the node corresponding to the number of vulnerabilities found on the host; this can be seen in Figure 6.4. Processing in this way allows the user to identify the protocols and connections present in the network they have scanned, as a first step to understanding any insecurity in their network that comes from its architecture. We colour the edges that originate form an external source as red, to allow the user to identify which hosts are immediately vulnerable to outside attacks.

The next iteration includes certain vulnerabilities as nodes to demonstrate one way of incorporating vulnerability data into the visualisation. This can be customised, along with the labelling of these vulnerabilities; for example Figure 6.5 shows a set up where any high priority vulnerability is added as a vulnerability node, and a node with an exceptionally high rating is labelled with its CVE to allow easy look-up of important vulnerabilities.

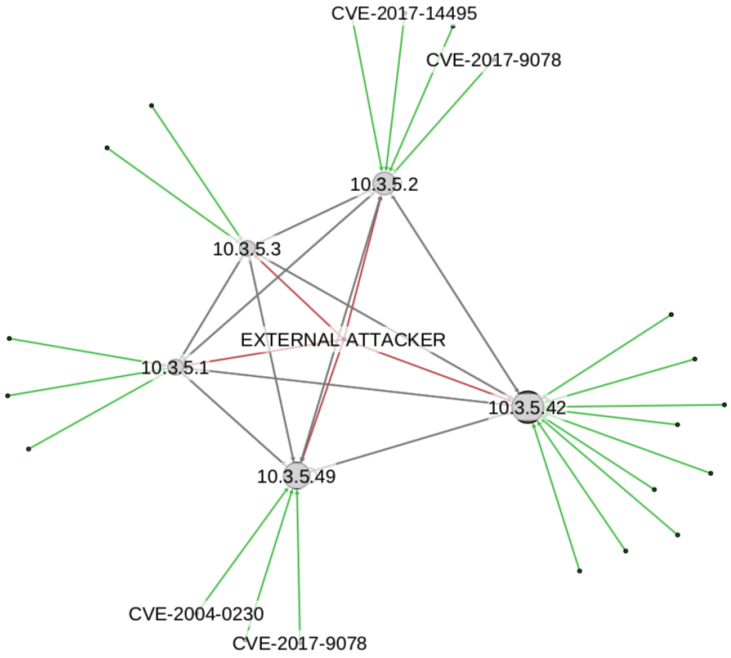


Fig. 6.5 Plot with vulnerabilities included and high priorities labelled

The last thing we demonstrate is the ability to label vulnerabilities using a custom database and dataset. This would be a way to include the output of the classifier described in Chapter 3 into the visualisation, and in doing so the required actions for each host can be labelled on the graph. For this implementation we use XQ's labelling system, but in order to use a custom dataset a different database should be included in the `docker-compose.yml` file. In order to accomplish this a new container was added that downloads an SQL file, then loads it into a MariaDB database in the container. Both this container and the Java container are put on the same Docker network, to allow communication, and the Java application then lists all the CVE's present and requests any relevant information from the database. With this information the vulnerabilities can be labelled and categorised using XQ's tags and information. This is shown in Figure 6.6.

6.4.4 Architecture

Fig. 6.6 Plot using XQ's categorisation

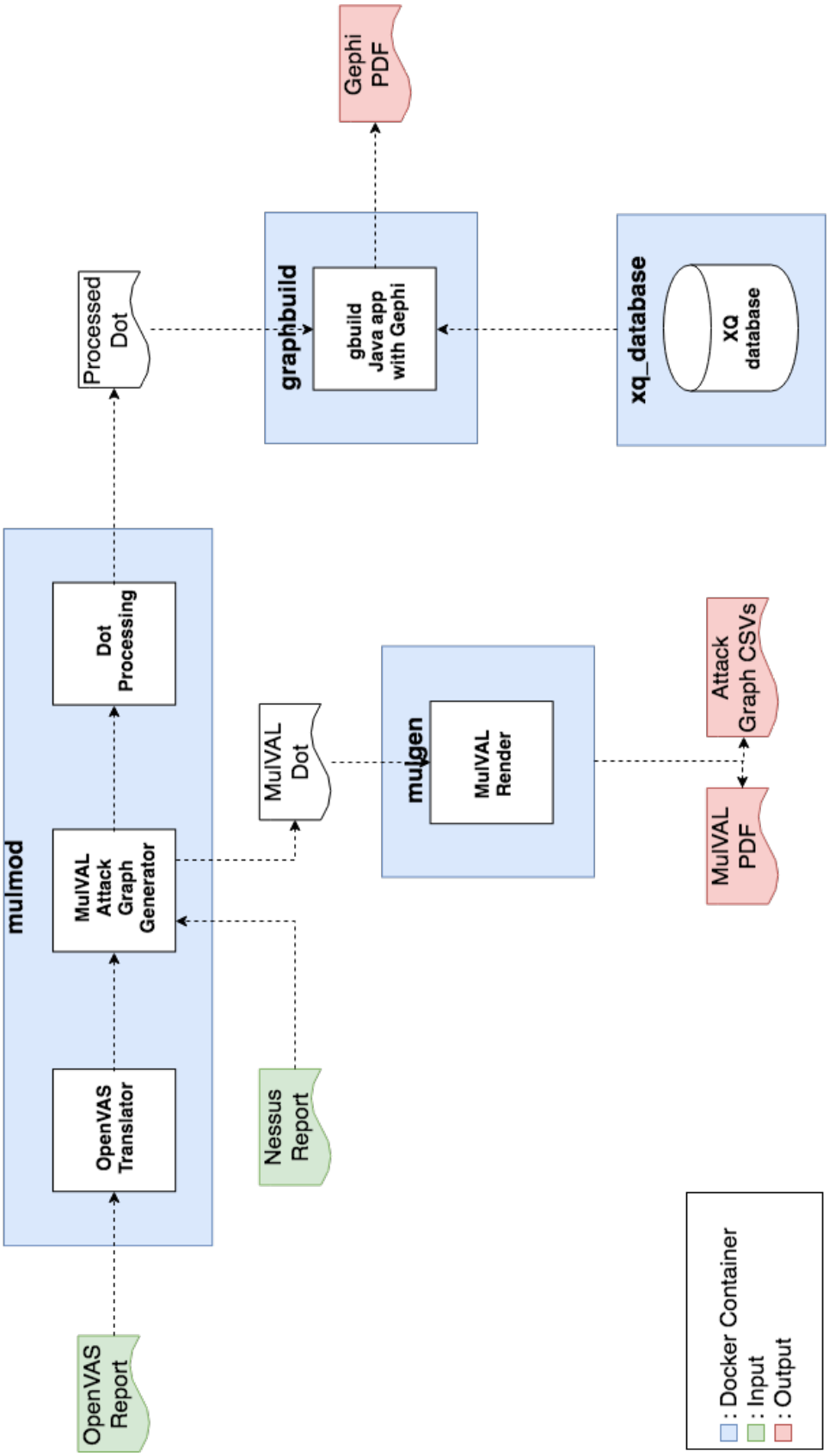


Fig. 6.7 Architecture of the Platform

Listing 6.2 Docker Compose YAML configuration of the platform

```
version: '2'
services:
mulval:
image: scimitar/mulmod:latest
volumes:
- ./input:/input

mulgen:
entrypoint: compose-generate.sh
image: scimitar/mulmod:latest
volumes:
- ./input:/input

xqdb:
image: scimitar/xq_db:latest
container_name: xqdb
restart: always
# expose:
#   - "3306"
ports:
- "127.0.0.1:3306:3306"
environment:
MYSQL_ROOT_PASSWORD: root

graphbuild:
depends_on:
- xqdb
image: scimitar/graphbuild:latest
volumes:
- ./input:/input
```

Figure 6.7 shows the architecture of the platform with all the components combined. The description of the containers in the Docker Compose YAML file is shown in Listing 6.2. Note that the 'xqdb' container is the container that connects to Gephi for the custom labelling of graphs and can either be replaced by a custom database to use other labels for vulnerabilities, or can be removed to use the standard NVD labelling system. This entire platform is fully automated and can be setup, updated, run and stopped using single commands as described earlier. Once the platform is running, the high level processing steps are shown in pseudocode in Algorithm 5.

Input: A vulnerability scan output from OpenVAS, Nessus, or written in OVAL
Output: Attack graph as xml, csv and dot files, PDF visualisation
if *Docker not installed* OR *Docker Compose not installed* **:**
    return Error message
    end procedure
start *mulmod* container
start *mulgen* container
start *graphbuild* container
start *database* container
Loop start
    Wait for new file scan
    Read scan
    In *mulmod*:
        if *scan in OpenVAS format* **:**
            run openvas_translate
            store in parsed_scan
        else:
            run MulVAL native translator
            store in parsed_scan
        run graph_gen script with parsed_scan as argument
        process output with dot_processor
    In *mulgen*:
        use renderer on mulmod output
        store attack graph csv files
        store attack graph pdf file
    In *graphbuild*:
        run Gephi import controller
        parse data
        if *database container running* **:**
            create dictionaries using labelling from *database*
        else:
            create dictionaries using name tags in input file
        append data and dictionaries to GraphAPI
        run graphing modules
        store visualisation pdf files
Loop end

Algorithm 5: Pseudocode for the working of the pipeline, after it has been brought online with `run.sh`

The platform takes, as input, either OpenVAS XML files or Nessus files. These files contain all the information required to generate an attack graph: the hosts on the network, their vulnerabilities, the topography of the network, and the allowed protocols and open ports inside it. The most common input type will be the OpenVAS style of report generated by CyberScore, and for MulVAL to process these files they first need to be translated. This is done by the OpenVAS translator which parses the files and generates new Datalog files that MulVAL can use. Next MulVAL uses the Datalog files to generate an attack graph; this is in the DOT format, which is a graph description language.

This DOT file is then processed by two different containers; the mulgen container takes the file and generates MulVAL's standard graph rendering outputs in the form of a PDF and several CSVs that describe the attack graph, and the mulmod container processes the DOT file to allow it to be plotted using Gephi's API in the graphbuild container.

After the platform has processed and created the attack graph, it is ready to be further analysed using more complex techniques, for example, by turning it into a probabilistic Bayesian network model as we demonstrate in Chapters 4 and 5.

6.5 Demonstration of the Platform

In order to demonstrate the use of the platform with larger real network examples, we use it to process vulnerability scans from several realistic network examples. Here we show the largest network, with others shown in Appendix B. For this example we use a complex enterprise network with several workstations, some servers, and a collection of heterogeneous peripheral devices. The network includes a server running TWiki that all the workstations can access for collaboration, an outdated Windows XP machine, a machine for web development and a Red Hat MRG machine. The full host inventory can be seen in table 6.1.

The scan is placed inside the input folder for the platform, where it is automatically translated by the OpenVAS translator. It is then put into MulVAL, which generates the dot file for the attack graph of the network. This is then processed and can be plotted as an attack

Table 6.1 Hosts and software for the 2342 node realistic network.

Type	Amount	Software
Old Windows Machine	1	Windows XP, Flash Player, JavaFX, Adobe Air, Wireshark
Windows Workstation	4	Internet Explorer, JDK, Office, DirectX, Edge
Windows Workstation	3	LiveMeeting, Edge
Windows Workstation	1	Internet Explorer, Office, Chrome, ExpressionWeb, JScript
Windows Server 2008	1	-
SMB Device	2	-
Ubuntu Machine	2	Pidgin, Chrome, Firefox, Apport, Python, Jasper, OpenSSL,
Red Hat Enterprise Machine	1	Enterprise MRG, Evince Libxml2, Poppler
Linux Database Server	1	-
TWiki Web Server	1	TWiki, PCRE, PHP, Samba
Remote Login Machine	1	OpenSSH

graph. The attack graph is shown in Figure 6.9. Due to the size of the graph, Figure 6.8 is an excerpt showing a privilege escalation on a host with many connections.

6.6 Conclusion

In this chapter we have described the development and structure of an attack graph platform that is fully containerised and automated and allows for the processing of vulnerability scans to be completely open-source and widely available on any platform. The pipeline enables the construction of regular attack graphs using the MulVAL engine, as well as serving as a platform to run and automate the visualisation of the resulting attack graphs. The visualisation container in the platform can be used as a way of incorporating the classifier designed in Chapter 3, and the output of the platform can be used with our Bayesian network techniques described in Chapters 4 and 5.

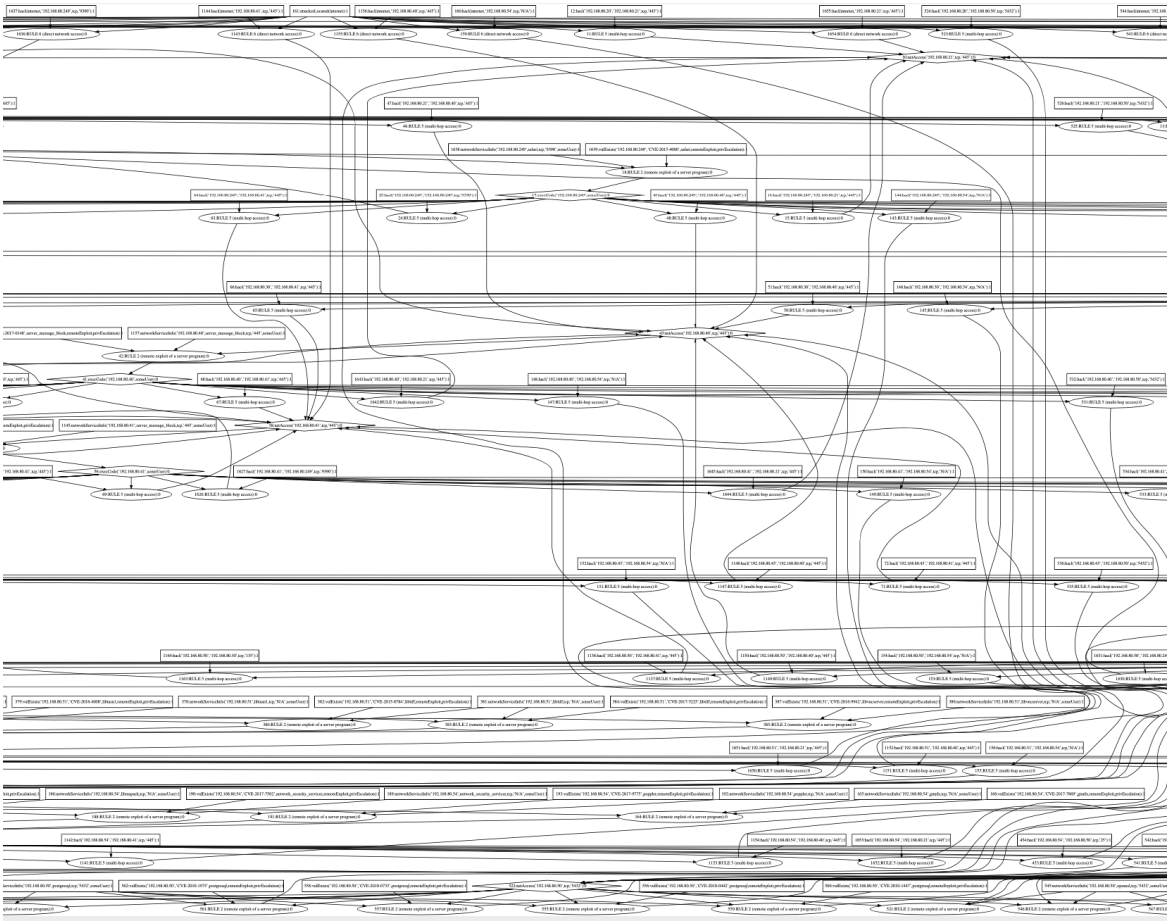


Fig. 6.8 Excerpt of 2342 node attack graph

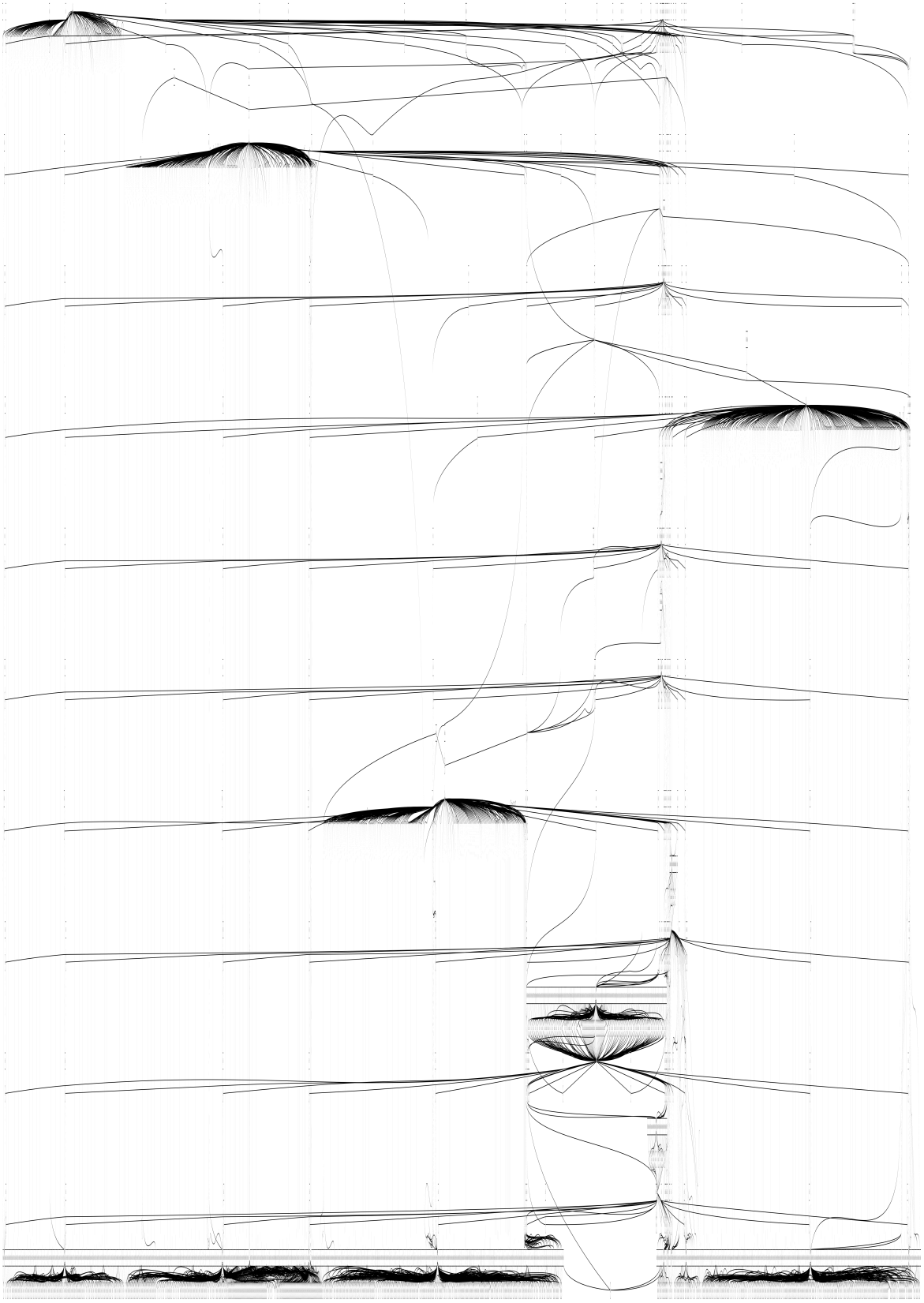


Fig. 6.9 Attack graph of 2342 node example

Chapter 7

Discussion and Conclusion

In this chapter we summarise this thesis and the contributions within. We examine some workflows that incorporate the methods and processes we discuss. We then look at the limitations of the work and of the area as a whole, and finally we discuss any open research problems in the field, as well as motivating new avenues of research as a result of this work.

7.1 Thesis Summary

In this thesis we have examined vulnerability scans and the difficulties in using a vulnerability scan to efficiently improve the security of a network. The overall objective of this work is to maximise the efficiency by which vulnerability scans are used once they have been created, and to enable further research with the same aim. We use a number of different approaches for different parts of the process of applying a scan to a network. Each approach we use improves the speed or simplicity for the use of analysing vulnerability scans to improve the security of a network.

We introduce a single unified formalism for Bayesian attack graphs in Chapter 2, with the aim of allowing future work to build from one shared set of definitions. We also demonstrate how another common formalism can be transformed into this one, meaning that past work that has been developed for that formalism can be used with our contributions and platform.

In Chapter 3 we examine the step from results to remediation, where a user is presented with the outcome of a scan and must then proceed to implement this result into the security of their network. We design and demonstrate a classifier using a neural network that can automatically assign vulnerabilities into classes with similar remediation processes, so that when a report is being generated for a user, instead of a list of complex and unintuitive identifiers, the user can be provided with a series of remediation steps. This process also significantly reduces the verbosity of the output as many vulnerabilities share remediation processes. This contribution improves the efficiency of directly applying scanning data to network security. We address the common problems associated with the National Vulnerability Database and enable use of the database to be simpler and sufficiently accurate.

Chapter 4 introduces the first of our contributions to the Bayesian attack graph space. Bayesian attack graphs can be used to learn the probabilities of reaching and exploiting security problems in a network, and so from this we can discover how we should prioritise vulnerabilities to have a maximal effect of reducing the probability of success in an attack. In this way, a scan can be used more efficiently by creating an attack graph model and fixing the most important vulnerabilities that are found. This chapter shows our interpretation of attack graphs with cycles, and we design an algorithm that can calculate access probabilities for BAGs that can run on any graph without identifying or removing cycles.

Our work with BAGs continues in Chapter 5, where we investigate how new information, like events triggered by an intrusion detection system, can be incorporated into an attack graph to calculate the security risk as an attack occurs. We implement and compare three stochastic simulation approaches to find the most efficient method for performing inference for BAGs. We also show a method for sensitivity analysis that significantly reduces computation times, and can be used to investigate the effects of prior assignment of BAGs, as well as enabling the modelling of how introducing new security controls could effect a network.

We also develop an accessible open-source platform for the generation and visualisation of attack graphs, in Chapter 6. The platform is modular and fully containerised, can run on any operating system that can run Docker, and can be customised through the use of alternative databases for labelling, and custom graphing files to perform different visualisation

techniques. In developing the platform we significantly lower the barrier for entry to process scans and generate attack graphs.

7.1.1 Combining Methods

The practical contributions and processes we discuss in this thesis (vulnerability classification in Chapter 3, access probability calculation in Chapter 4, inference and sensitivity analysis in Chapter 5 and attack graph generation and visualisation in Chapter 6) can be used separately, but also may be combined in various ways to achieve different analyses of a vulnerability scan. There are many ways to go about this, but we will discuss three potential routes that we anticipate to be the most common.

Security audit An audit of an organisation's security is something that is required by many; it can be used by insurance companies to inform their premiums, it can be requested by a company to ensure their supply-chain is secure, and it can be used by the organisation itself to quantitatively measure security for a comparison with past measurements and to improve the current security. Once a vulnerability scan has been run, passing it through our attack graph platform will generate the corresponding attack graph. This graph can then be evaluated using Algorithm 1 to discover the most exposed hosts, the likelihood of a breach of data, or to prioritise the vulnerabilities. This information can be reported as it is in the insurance and supply-chain cases, or can then be processed using the visualisation container of the platform and the data from the vulnerability classifier, so that the most impactful vulnerabilities can be easily remedied.

Network monitoring Several of the methods can be combined to allow for an online monitoring of a network and its risks using information gathering tools such as intrusion detection systems or network monitoring systems. Again, after passing a vulnerability scan of the network through the attack graph generation platform, the Bayesian attack graph can be plotted using an interactive instance of Gephi using

the visualisation container after it has been generated by the stochastic simulation BAG calculation process. Then, when new evidence arrives, this evidence can be incorporated into the model in real-time and the graph can be adjusted in the Gephi instance to observe and react to an attack that is in progress. This same workflow can be used to examine new security controls, by generating new evidence that corresponds to the security control (for example holding a node at 0 if it corresponds to a communication protocol that will be blocked by the security control) the effect on the probabilities on the graph can be investigated.

Robust network design These techniques can also be used before a network is even put in place. Using virtualisation, hosts can be simulated in various network scenarios and then the networks can be scanned, and transformed into attack graphs. Using a process for probability propagation the security of various designs of the network can be analysed and used to inform the network design procedure. Furthermore, use of the sensitivity analysis discussed in Section 5.5 can allow different network controls to be directly compared in relation to their effect on the security of any of the hosts on the network.

7.2 Limitations

One of the major limitations within this work, and within the Bayesian attack graph field as a whole, is the limitation of graph size. We investigate graphs of up to 15,000 nodes in Chapter 4, which is adequate as a solution for even very large enterprises. However, with the advent of new technologies, including 5G, fog computing, and distributed supply-chain networks, the number of hosts and therefore nodes in an attack graph may become large enough such that calculating the probabilities on the attack graph takes a prohibitively long amount of time.

Another limitation of the BAG field is the imprecise nature of the prior assignment of vulnerability probabilities. While more complex methods do exist that incorporate more of the available data, the prior values for the likelihood of exploiting a vulnerability still

remain based on arbitrary values assigned in the NVD. While we do demonstrate an improved approach for investigating the effect of these assignments using the sensitivity analysis of Section 5.5, the actual effect of these prior assignments of the goal node probabilities remains unknown. Furthermore, there is no method currently available for generating these priors from empirical data.

A limitation of the classifier is the requirement for hand-labelling. Ideally, the process of discovering the labels would be automated as well, as this would mean the model could react to new software and new vulnerability types that are not preexisting in the dataset. Currently, if a new piece of software is developed and a vulnerability is found, a new family would need to be created and added to the dataset manually, slowing the process and using more resources.

7.3 Future Research Directions

As a result of the experience of conducting the research documented in this thesis, and as a continuation to the work done, we motivate a number of potential areas of future research.

7.3.1 Data Collection

Originally, one of the aims of this thesis was to collect anonymised data from XQ Cyber-Score security scans, in order to create a large and empirical dataset about the presence of vulnerabilities and the frequency of their exploit. This dataset still does not exist, but could be used to great effect within the attack graph field. Such a dataset could be combined with attack vector datasets, like the way in which the Amazon web services' honeypot database has been used [4] to more quantitatively evaluate current attack probabilities for various applications. For example, a new method for prior generation, based on empirical attack evidence, could be developed. This would be a significant improvement to the validity and accuracy of BAGs generated using these new prior values. The data could also be used to identify specific architectures that are more prone to attack, and so based off a vulnerability scan an alternative network architecture could be suggested to users.

7.3.2 Data Driven Techniques

Building from our work in Chapter 5, other data driven techniques for inference on BAGs should be investigated. A good starting point would be to investigate some of the variations of Gibbs samplers for Bayesian inference, as originally proposed by Geman and Geman [30]. Gibbs sampling is performed by sampling a configuration that is consistent with the evidence that has been given, and then stochastically altering the states of the variables on the graph before resampling. Implementing and testing various techniques could allow for improved computational efficiency, and allow BAG generation to scale with the growing size of networks.

7.3.3 Visualisations of Graphs

Another avenue of research is that of the visualisation aspect of reporting. We demonstrate a method for improving the textual reporting of vulnerabilities using a classification system, but have only implemented simple visualisations into our platform as a proof-of-concept. Other visualisation techniques should be explored, for example the interactive visualisation that displays the reachability of hosts within the network described by Williams et al. [118]. The possibilities for improving attack graph readability through visualisation techniques remains relatively unexplored.

7.3.4 Unsupervised Classifier

The possibility of using unsupervised machine learning techniques to categorise vulnerabilities also remains unknown. If possible, an unsupervised online model could be used to automatically classify vulnerabilities by remediation processes without the requirement for human labelling of the training dataset, as well as allowing the model to automatically react to and include new categories when they arise in the incoming data.

References

[1] Abraham, S. and Nair, S. (2015). Predictive cyber security analytics framework : A non-homogenous markov model for security quantification. *Computer Science and Information Technology*, 4.

[2] Aguessy, F., Bettan, O., Blanc, G., Conan, V., and Debar, H. (2016). Bayesian attack model for dynamic risk assessment. *CoRR*, abs/1606.09042.

[3] Allegretti, A. (2018). Cost of wannacry cyber attack to the nhs revealed. https://news.sky.com/story/cost-of-wannacry-cyber-attack-to-the-nhs-revealed-11523784. Accessed: 2021-05-28.

[4] Almohannadi, H., Awan, I., Al Hamar, J., Cullen, A., Disso, J. P., and Armitage, L. (2018). Cyber threat intelligence from honeypot data using elasticsearch. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 900–906. IEEE.

[5] Ammann, P., Wijesekera, D., and Kaushik, S. (2002). Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 217–224, New York, NY, USA. ACM.

[6] Baiardi, F. and Sgandurra, D. (2013). Assessing ICT risk through a Monte Carlo method. *Environment Systems and Decisions*, 33(4):486–499.

[7] Barik, M., Sengupta, A., and Mazumdar, C. (2016). Attack graph generation and analysis techniques. *Defence Science Journal*, 66(6):559–567.

[8] Barnum, S. (2008). Common attack pattern enumeration and classification (CAPEC) schema description. *Cigital Inc, http://capec. mitre. org/documents/documentation/-CAPEC_Schema_Descr iption_v1*, 3.

[9] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2).

[10] Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

[11] Bozorgi, M., Saul, L. K., Savage, S., and Voelker, G. M. (2010a). Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 105–114. ACM.

[12] Bozorgi, M., Saul, L. K., Savage, S., and Voelker, G. M. (2010b). Beyond heuristics: Learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 105–114, New York, NY, USA. ACM.

[13] Brownlee, J. (2020). Why one-hot encode data in machine learning? https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/. Accessed: 2021-05-15.

[14] Center for Internet Security (2021). Open vulnerability assessment language. https://oval.cisecurity.org/. Accessed: 2021-05-15.

[15] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357.

[16] Cheng, P., Wang, L., Jajodia, S., and Singhal, A. (2017). *Refining CVSS-Based Network Security Metrics by Examining the Base Scores*, pages 25–52. Springer International Publishing.

[17] Choi, A., Chan, H., and Darwiche, A. (2012). On Bayesian network approximation by edge deletion. *CoRR*, abs/1207.1370.

[18] Christian, H., Agus, M. P., and Suhartono, D. (2016). Single document automatic text summarization using term frequency-inverse document frequency (tf-idf). *ComTech: Computer, Mathematics and Engineering Applications*, 7(4):285–294.

[19] Chu, M., Ingols, K., Lippmann, R., Webster, S., and Boyer, S. (2010). Visualizing attack graphs, reachability, and trust relationships with navigator. In *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*, VizSec '10, page 22–33, New York, NY, USA. Association for Computing Machinery.

[20] Dantu, R., Loper, K., and Kolan, P. (2004). Risk management using behavior based attack graphs. In *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, volume 1, pages 445–449.

[21] Deloitte Financial Advisory (2019). The performance of small and medium sized businesses in a digital world. https://www2.deloitte.com/content/dam/Deloitte/es/Documents/Consultoria/The-performance-of-SMBs-in-digital-world.pdf. Accessed: 2021-05-15.

[22] Doynikova, E. and Kotenko, I. (2017). Enhancement of probabilistic attack graphs for accurate cyber security monitoring. In *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation*, pages 1–6.

[23] Edkrantz, M. and Said, A. (2015). Predicting cyber vulnerability exploits with machine learning. In *SCAI*, pages 48–57.

[24] FireMon (2013). Firemon risk analysis and attack simulation. https://www.firemon.com/products/risk-analyzer/. Accessed: 2021-05-15.

[25] FIRST (2019). Common vulnerability scoring system v3.1: Specification document. https://www.first.org/cvss/v3.1/specification-document. Accessed: 2021-05-15.

[26] Frigault, M., Wang, L., Jajodia, S., and Singhal, A. (2017). *Measuring the Overall Network Security by Combining CVSS Scores Based on Attack Graphs and Bayesian Networks*, pages 1–23. Springer International Publishing, Cham.

[27] Frigault, M., Wang, L., Singhal, A., and Jajodia, S. (2008). Measuring network security using dynamic Bayesian network. In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, pages 23–30, New York, NY, USA. ACM.

[28] Fung, R. and Del Favero, B. (1994). Backward simulation in Bayesian networks. In *Uncertainty Proceedings 1994*, pages 227–234. Elsevier.

[29] Gawron, M., Cheng, F., and Meinel, C. (2017). Automatic vulnerability classification using machine learning. In *International Conference on Risks and Security of Internet and Systems*, pages 3–17. Springer.

[30] Geman, S. and Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 721–741.

[31] Gephi Consortium (2021). Gephi. https://gephi.org/. Accessed: 2021-05-17.

[32] Geweke, J. (1989). Bayesian inference in econometric models using monte carlo integration. *Econometrica: Journal of the Econometric Society*, pages 1317–1339.

[33] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

[34] Graphviz (2021). The dot language. https://graphviz.org/doc/info/lang.html. Accessed: 2021-05-17.

[35] Greenbone (2006). Open vulnerability assesment scanner. https://www.openvas.org/. Accessed: 2021-05-15.

[36] Greenbone (2021). Greenbone security manager: NVTs. https://secinfo.greenbone.net/nvts. Accessed: 2021-05-15.

[37] Harris, D. and Harris, S. L. (2010). *Digital design and computer architecture*. Morgan Kaufmann.

[38] Homer, J., Varikuti, A., Ou, X., and McQueen, M. A. (2008). Improving attack graph visualization through data reduction and attack grouping. In Goodall, J. R., Conti, G., and Ma, K.-L., editors, *Visualization for Computer Security*, pages 68–79, Berlin, Heidelberg. Springer Berlin Heidelberg.

[39] Homer, J., Zhang, S., Ou, X., Schmidt, D., Du, Y., Rajagopalan, S. R., and Singhal, A. (2013). Aggregating vulnerability metrics in enterprise networks using attack graphs. *Journal of Computer Security*, 21(4):561–597.

[40] Hong, J. B., Kim, D. S., Chung, C.-J., and Huang, D. (2017). A survey on the usability and practical applications of graphical security models. *Computer Science Review*, 26:1–16.

[41] Hu, H., Liu, J., Zhang, Y., Liu, Y., Xu, X., and Tan, J. (2020). Attack scenario reconstruction approach using attack graph and alert data mining. *Journal of Information Security and Applications*, 54:102522.

[42] Hu, Z., Zhu, M., and Liu, P. (2017). Online algorithms for adaptive cyber defense on Bayesian attack graphs. In *Proceedings of the 2017 Workshop on Moving Target Defense*, pages 99–109. ACM.

[43] Huang, G., Li, Y., Wang, Q., Ren, J., Cheng, Y., and Zhao, X. (2019). Automatic classification method for software vulnerability based on deep neural network. *IEEE Access*, 7:28291–28298.

[44] Huangfu, Y., Zhou, L., and Yang, C. (2017). Routing the cyber-attack path with the Bayesian network deducing approach. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 5–10.

[45] Ingols, K., Lippmann, R., and Piwowarski, K. (2006). Practical attack graph generation for network defense. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 121–130.

[46] Ipsos MORI (2019). Cyber security breaches survey. https://www.ipsos.com/ipsos-mori/en-uk/cyber-security-breaches-survey-2019. Accessed: 2021-05-15.

[47] Jajodia, S. and Noel, S. (2009). Topological vulnerability analysis: A powerful new approach for network attack prevention, detection, and response. In *Algorithms, Architectures and Information Systems Security*, pages 285–305. World Scientific.

[48] Jajodia, S., Noel, S., and O'berry, B. (2005). Topological analysis of network attack vulnerability. In *Managing cyber threats*, pages 247–266. Springer.

[49] Japkowicz, N. and Stephen, S. (2002). The class imbalance problem: A systematic study. *Intelligent Data Analysis*, 6(5):429–449.

[50] Jha, S., Sheyner, O., and Wing, J. (2002). Two formal analyses of attack graphs. In *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*, pages 49–63.

[51] Jiang, G. and Ren, M. (2018). Cyclic Bayesian network for software project iterative process. In *2018 International Conference on Mathematics, Modelling, Simulation and Algorithms*, MMSA'18, pages 413–417.

[52] Jo, H., Kim, J.-H., Kim, K.-M., Chang, J.-H., Eom, J.-H., and Zhang, B.-T. (2017). Large-scale text classification with deep neural networks. *Journal of Information Science and Technology Computing*, 23(5):322–327.

[53] Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. In *European Conference on Machine Learning*, pages 137–142. Springer.

[54] Kaynar, K. (2016). A taxonomy for attack graph generation and usage in network security. *Journal of Information Security and Applications*, 29:27–56.

[55] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[56] Kinzer, S. (2015). Why relying on the nvd is not good for open-source security tools. https://www.veracode.com/blog/managing-appsec/why-relying-nvd-not-good-open-source-security-tools. Accessed: 2021-05-15.

[57] Kłopotek, M. A. (2006). Cyclic Bayesian network: Markov process approach. *Studia Informatica: Systems and Information Technology*, 1.

[58] Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press.

[59] Kordy, B., Mauw, S., Radomirović, S., and Schweitzer, P. (2010). Foundations of attack–defense trees. In *International Workshop on Formal Aspects in Security and Trust*, pages 80–95. Springer.

[60] Kordy, B., Pietre-Cambacedes, L., and Schweitzer, P. (2014a). DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review*, 13-14:1 – 38.

[61] Kordy, B., Pouly, M., and Schweitzer, P. (2014b). A probabilistic framework for security scenarios with dependent actions. In Albert, E. and Sekerinski, E., editors, *Integrated Formal Methods*, pages 256–271, Cham. Springer International Publishing.

[62] Kumar, R. and Stoelinga, M. (2017). Quantitative security and safety analysis with attack-fault trees. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 25–32. IEEE.

[63] Lee, J., Moon, D., Kim, I., and Lee, Y. (2018). A semantic approach to improving machine readability of a large-scale attack graph. *The Journal of Supercomputing*.

[64] Lin, Z., Li, X., and Kuang, X. (2017). Machine learning in vulnerability databases. In *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, volume 1, pages 108–113.

[65] Ling, C. X. and Li, C. (1998). Data mining for direct marketing: Problems and solutions. In *Knowledge Discovery from Data (KDD)*, volume 98, pages 73–79.

[66] Liu, Y. and Man, H. (2005). Network vulnerability assessment using Bayesian networks. In *Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security 2005*, volume 5812, pages 61–72. International Society for Optics and Photonics.

[67] Luckett, P., McDonald, J. T., and Glisson, W. B. (2017). Attack-graph threat modeling assessment of ambulatory medical devices. *arXiv preprint arXiv:1709.05026*.

[68] Mace, J. C., Thekkummal, N., Morisset, C., and van Moorsel, A. (2017). ADaCS: A tool for analysing data collection strategies. In *Computer Performance Engineering*, EPEW'17, pages 230–245.

[69] Matthews, I. (2021). Attack graph generation platform. https://github.com/Isaac-Matthews/ag-gen-platform. Accessed: 2021-07-05.

[70] Matthews, I., Mace, J., Soudjani, S., and van Moorsel, A. (2020). Cyclic bayesian attack graphs: A systematic computational approach. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 129–136.

[71] Matthews, I., Soudjani, S., and van Moorsel, A. (2021). Stochastic simulation techniques for inference and sensitivity analysis of bayesian attack graphs. In *Science of Cyber Security*, pages 171–186, Cham. Springer International Publishing.

[72] Mauw, S. and Oostdijk, M. (2006). Foundations of attack trees. In Won, D. H. and Kim, S., editors, *Information Security and Cryptology - ICISC 2005*, pages 186–198, Berlin, Heidelberg. Springer Berlin Heidelberg.

[73] Mell, P., Scarfone, K., and Romanosky, S. (2006). Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6).

[74] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[75] Miehling, E., Rasouli, M., and Teneketzis, D. (2015). Optimal defense policies for partially observable spreading processes on Bayesian attack graphs. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 67–76, New York, NY, USA. ACM.

[76] Morgan, S. (2020). Cybercrime to cost the world $10.5 trillion annually by 2025. https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/. Accessed: 2021-05-28.

[77] Muñoz-González, L., Sgandurra, D., Barrere, M., and Lupu, E. (2017). Exact inference techniques for the analysis of Bayesian attack graphs. *IEEE Transactions on Dependable and Secure Computing*.

[78] Muñoz-González, L., Sgandurra, D., Paudice, A., and Lupu, E. C. (2016). Efficient attack graph analysis through approximate inference. *CoRR*, abs/1606.07025.

[79] Nam, J., Kim, J., Loza Mencía, E., Gurevych, I., and Fürnkranz, J. (2014). Large-scale multi-label text classification — revisiting neural networks. In Calders, T., Esposito, F., Hüllermeier, E., and Meo, R., editors, *Machine Learning and Knowledge Discovery in Databases*, pages 437–452, Berlin, Heidelberg. Springer Berlin Heidelberg.

[80] National Institute of Standards and Technology (2021). National vulnerability database. https://www.https://nvd.nist.gov/. Accessed: 2021-05-15.

[81] Nielsen, T. D. and Jensen, F. V. (2009). *Bayesian networks and decision graphs*. Springer Science & Business Media.

[82] Nigam, K., Lafferty, J., and McCallum, A. (1999). Using maximum entropy for text classification. In *IJCAI-99 workshop on machine learning for information filtering*, volume 1, pages 61–67. Stockholom, Sweden.

[83] Noel, S. and Jajodia, S. (2004). Managing attack graph complexity through visual hierarchical aggregation. In *Proceedings of the 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, VizSEC/DMSEC '04, page 109–118, New York, NY, USA. Association for Computing Machinery.

[84] Noel, S., Jajodia, S., Wang, L., and Singhal, A. (2010). Measuring security risk of networks using attack graphs. *International Journal of Next-Generation Computing*, 1(1):135–147.

[85] Ou, X., Boyer, W. F., and McQueen, M. A. (2006). A scalable approach to attack graph generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 336–345, New York, NY, USA. ACM.

[86] Ou, X., Govindavajhala, S., and Appel, A. W. (2005). Mulval: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 8–8, Berkeley, CA, USA. USENIX Association.

[87] Ou, X., Govindavajhala, S., and Appel, A. W. (2013). Mulval readme. http://people.cs.ksu.edu/~xou/argus/software/mulval/readme.html. Accessed: 2021-05-15.

[88] Ou, X. and Singhal, A. (2011). *Attack Graph Techniques*, pages 5–8. Springer New York, New York, NY.

[89] Ozment, J. A. (2007). *Vulnerability discovery & software security*. PhD thesis, University of Cambridge.

[90] Pietre-Cambacedes, L. and Bouissou, M. (2010). Beyond attack trees: Dynamic security modeling with Boolean logic driven Markov processes (BDMP). In *2010 European Dependable Computing Conference*, pages 199–208.

[91] Poolsappasit, N., Dewri, R., and Ray, I. (2012). Dynamic security risk management using Bayesian attack graphs. *IEEE Transactions on Dependable and Secure Computing*, 9(1):61–74.

[92] Prechelt, L. (1998). Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer.

[93] Qian, W., Riedel, M. D., Bazargan, K., and Lilja, D. J. (2009). The synthesis of combinational logic to generate probabilities. In *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, pages 367–374.

[94] Qian, W., Riedel, M. D., Zhou, H., and Bruck, J. (2011). Transforming probabilities with combinational logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(9):1279–1292.

[95] Rajasooriya, S. M., Tsokos, C. P., Kaluarachchi, P. K., et al. (2017). Cyber security: Nonlinear stochastic models for predicting the exploitability. *Journal of Information Security*, 8(02):125.

[96] Ramaki, A. A., Khosravi-Farmad, M., and Bafghi, A. G. (2015). Real time alert correlation and prediction using Bayesian networks. In *2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*, pages 98–103.

[97] Ryder, T., Prangle, D., Golightly, A., and Matthews, I. (2021). The neural moving average model for scalable variational inference of state space models. In *Uncertainty in Artificial Intelligence*.

[98] Saha, D. (2008). Extending logical attack graphs for efficient vulnerability analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 63–74, New York, NY, USA. ACM.

[99] Salter, C., Saydjari, O. S., Schneier, B., and Wallner, J. (1998). Toward a secure system engineering methodolgy. In *Proceedings of the 1998 Workshop on New Security Paradigms*, pages 2–10.

[100] Sawilla, R. and Ou, X. (2007). *Googling attack graphs*. Defence R & D Canada-Ottawa.

[101] Schneier, B. (1999). Attack trees. *Dr. Dobb's journal*, 24(12):21–29.

[102] Sembiring, J., Ramadhan, M., Gondokaryono, Y. S., and Arman, A. A. (2015). Network security risk analysis using improved mulval Bayesian attack graphs. *International Journal on Electrical Engineering and Informatics*, 7(4):735.

[103] Sheyner, O. M. (2004). Scenario graphs and attack graphs. Technical report, Carnegie-Mellon University School Of Computer Science.

[104] Shirey, R. (2007). Internet security glossary, version 2. IETF. Technical report, RFC 4949 (Informational).

[105] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

[106] Stoneburner, G., Hayden, C., and Feringa, A. (2004). Nist special publication 800-27 rev a. *Engineering Principles for Information Technology Security (a Baseline for Achieving Security). National Institute of Standards and Technology, Gaithersburg*.

[107] Swiler, L. P., Phillips, C., Ellis, D., and Chakerian, S. (2001). Computer-attack graph generation tool. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 307–321 vol.2.

[108] Team, C. (2019). Common vulnerability scoring system v3.1: Specification document. *First. org*.

[109] Team Firmex (2016). The 10 most expensive data breaches in corporate history. https://www.firmex.com/resources/blog/the-10-most-expensive-data-breaches-in-corporate-history/. Accessed: 2021-05-28.

[110] Tenable (1998). Nessus vulnerability scanner. https://www.tenable.com/products/nessus/nessus-professional/. Accessed: 2021-05-15.

[111] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B*, 58(1):267–288.

[112] Ubuntu (2021). Ubuntu xenial for docker. https://github.com/tianon/docker-brew-ubuntu-core/blob/4b7cb6f04bc4054f9ab1fa42b549caa1a41b7c92/xenial/Dockerfile. Accessed: 2021-05-15.

[113] Wagner, J.-O. (2018). Openvas 8 structure. http://openvas.org/img/OpenVAS-8-Structure.png. Accessed: 2021-05-15.

[114] Wang, L., Islam, T., Long, T., Singhal, A., and Jajodia, S. (2008). An attack graph-based probabilistic security metric. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 283–296. Springer.

[115] Wang, L., Jajodia, S., and Singhal, A. (2017). *Network Security Metrics*. Springer.

[116] Wang, S., Zhang, Z., and Kadobayashi, Y. (2013a). Exploring attack graph for cost-benefit security hardening: A probabilistic approach. *Computers & Security*, 32:158–169.

[117] Wang, S., Zhang, Z., and Kadobayashi, Y. (2013b). Exploring attack graph for cost-benefit security hardening: A probabilistic approach. *Computers & Security*, 32(Supplement C):158 – 169.

[118] Williams, L., Lippmann, R., and Ingols, K. (2008). *An Interactive Attack Graph Cascade and Reachability Display*, pages 221–236. Springer Berlin Heidelberg, Berlin, Heidelberg.

[119] Xie, A., Cai, Z., Tang, C., Hu, J., and Chen, Z. (2009). Evaluating network security with two-layer attack graphs. In *2009 Annual Computer Security Applications Conference*, pages 127–136. IEEE.

[120] Xie, P., Li, J. H., Ou, X., Liu, P., and Levy, R. (2010a). Using Bayesian networks for cyber security analysis. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 211–220.

[121] Xie, P., Li, J. H., Ou, X., Liu, P., and Levy, R. (2010b). Using Bayesian networks for cyber security analysis. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 211–220.

[122] Yager, R. R. (2006). Owa trees and their role in security modeling using attack trees. *Information Sciences*, 176(20):2933–2959.

[123] Yi, S., Peng, Y., Xiong, Q., Wang, T., Dai, Z., Gao, H., Xu, J., Wang, J., and Xu, L. (2013). Overview on attack graph generation and visualization technology. In *2013 International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, pages 1–6.

[124] Zhang, S., Caragea, D., and Ou, X. (2011). An empirical study on using the national vulnerability database to predict software vulnerabilities. In *International Conference on Database and Expert Systems Applications*, pages 217–231. Springer.

[125] Zhong, S., Yan, D., and Liu, C. (2009). Automatic generation of host-based network attack graph. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 1, pages 93–98.

Appendix A

Full Example

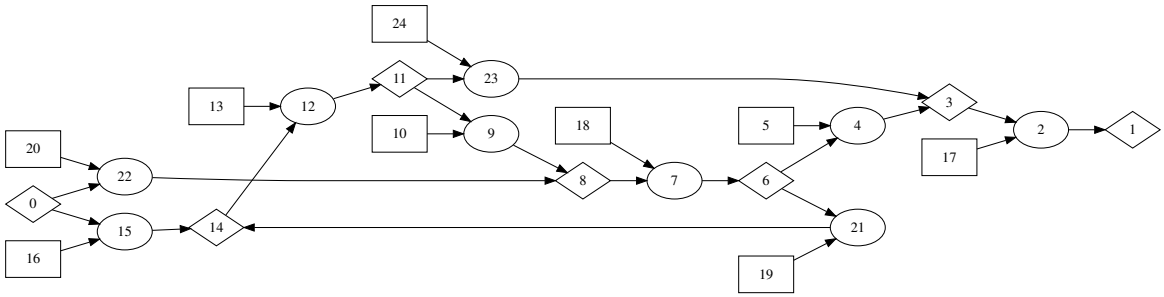


Fig. A.1 The BAG of the running example including leaf nodes.

The complete attack graph for the running example scenario can be seen in Figure A.1, with the labels for the nodes written out below. An important note is the reason the cycle exists: the state at node 14, whereby a user on one of the Workstation machines access a malicious website, can be reached via two means. Firstly the user can simply visit a malicious website allowing the attacker to exploit CVE-2009-1918 that is in Internet Explorer on the Workstation. Alternatively, an attacker that has achieved the ability to execute code on the Web Server (node 6) can serve the user of a Workstation machine an HTML document that exploits CVE-2009-1918 and thus also achieves the state on node 14.

This cycle is made more complex due to the ability to access the Webserver without passing through the Workstations originally (visiting nodes 22 and 8). Because of this, reaching node 14 via node 21 will not always mean that the attacker is travelling backwards,

and as such the monotonicity principle does not apply and the probability of reaching node 14 becomes more challenging to calculate.

The vulnerabilities in this scenario are as follow:

- CVE-2009-1918[1] on the Workstations - Internet Explorer vulnerability that allows an attacker to execute arbitrary code on the machine after the user accesses a website with purposely malformed elements that trigger memory corruption
- CVE-2006-3747[2] on the Webserver - Apache vulnerability that can be exploited to execute arbitrary code using crafted URLs and requires network access to exploit
- CVE-2009-2446[3] on the Database Server - MySQL vulnerability where an authenticated user can cause a denial of service and possibly execute arbitrary code

Listing A.1 MulVAL labels for Figure A.1

```
0, "attackerLocated(internet)"
1, "execCode(dbServer,root)"
2, "RULE 2 (remote exploit of a server program)"
3, "netAccess(dbServer,tcp,'3306')"
4, "RULE 5 (multi-hop access)"
5, "hacl(webServer,dbServer,tcp,'3306')"
6, "execCode(webServer,apache)"
7, "RULE 2"
8, "netAccess(webServer,tcp,'80')"
9, "RULE 5"
10, "hacl(workStation,webServer,tcp,'80'"
11, "execCode(workStation,userAccount)"
12, "RULE 2"
13, "vulExists(workStation,'CVE-2009-1918',IE,remoteExploit,
```

[1] https://nvd.nist.gov/vuln/detail/CVE-2009-1918
[2] https://nvd.nist.gov/vuln/detail/CVE-2006-3747
[3] https://nvd.nist.gov/vuln/detail/CVE-2009-2446

```
privEscalation)"
14, "accessMaliciousInput(workStation,user, IE)"
15, "malicious website"
16, "visit of malicious website"
17, "vulExists(dbServer,'CVE-2009-2446',mySQL,remoteExploit,
privEscalation)"
18, "vulExists(webServer,'CVE-2006-3747',apache,remoteExploit,
privEscalation)"
19, "visit of compromised website"
20, "hacl(internet, webServer, tcp, '80')"
21, "compromise of website"
22, "RULE 6 (direct network access"
23, "RULE 5"
24, "hacl(workStation,dbServer,tcp,'3306')
```

Appendix B

Realistic Network Examples

B.1 200 Nodes

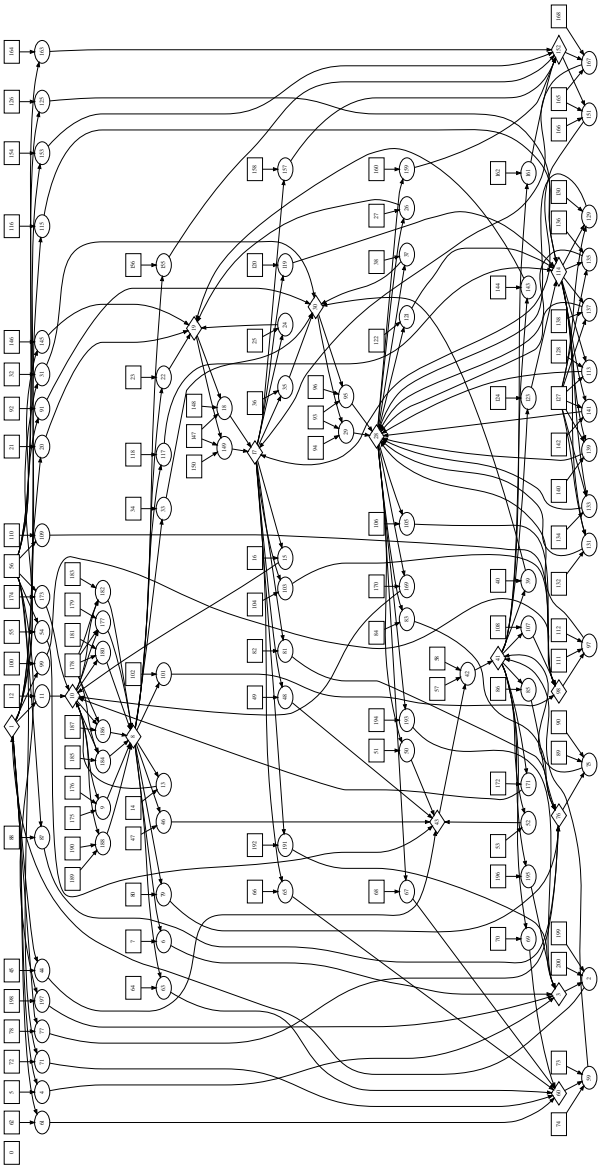


Fig. B.1 Attack graph of 200 node example

B.2 1053 Nodes

This network is a simple small enterprise setup, with several workstations, some servers, and a collection of peripheral devices. The full host inventory can be seen in table B.1.

Table B.1 Hosts and software for the 1053 node realistic network.

Type	Amount	Software
Windows Workstation	4	Internet Explorer, JDK, iTunes, Office
Windows Server 2008	1	-
SMB Device	1	-
Linux Machine	2	Pidgin, Chrome, Firefox, Samba
Linux Database Server	1	-
iOS Machine	1	Apple TV

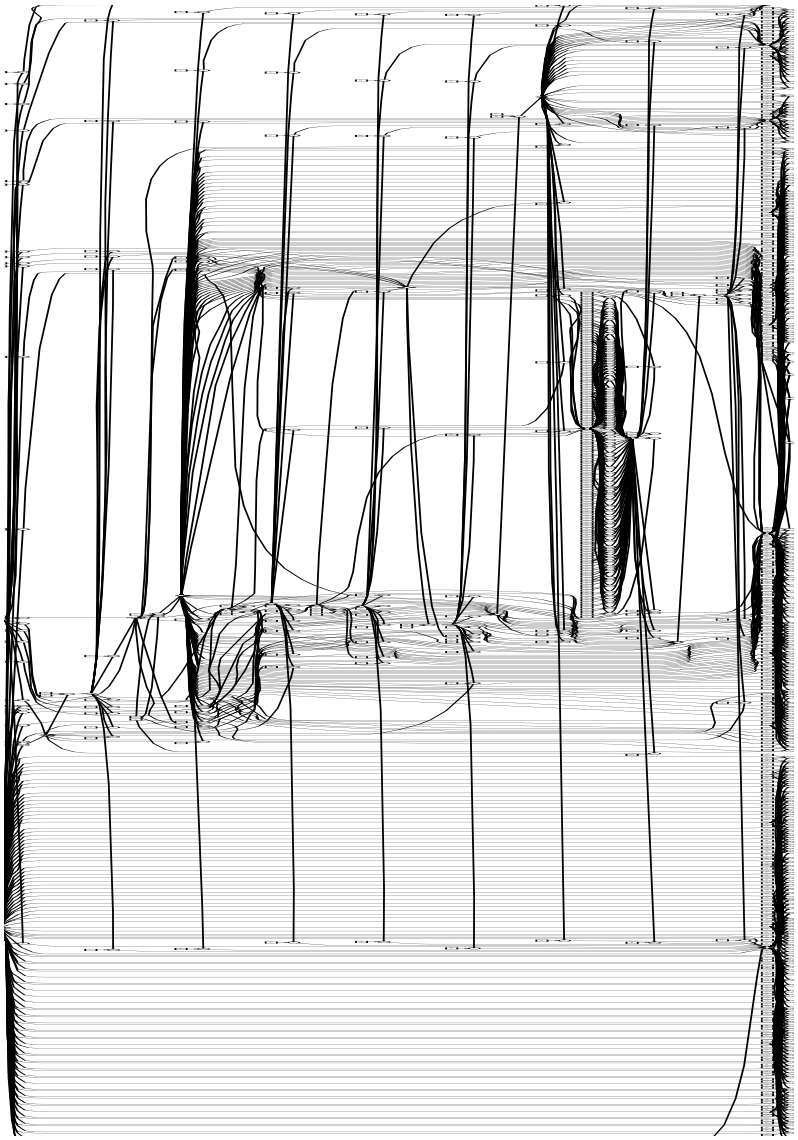


Fig. B.2 Attack graph of 1053 node example

B.3 2234 Nodes

This network is a more complex enterprise example, and includes a server running TWiki that all the workstations can access for collaboration. The full host inventory can be seen in table B.2.

Table B.2 Hosts and software for the 2234 node realistic network.

Type	Amount	Software
Windows Workstation	4	Internet Explorer, JDK, Office, DirectX, Edge
Windows Workstation	3	LiveMeeting, Edge
Windows Server 2008	1	-
SMB Device	2	-
Ubuntu Machine	2	Pidgin, Chrome, Firefox, Apport, Python, Jasper, OpenSSL, Libxml2, Poppler
Linux Database Server	1	-
TWiki Web Server	1	TWiki, PCRE, PHP, Samba
Remote Login Machine	1	OpenSSH

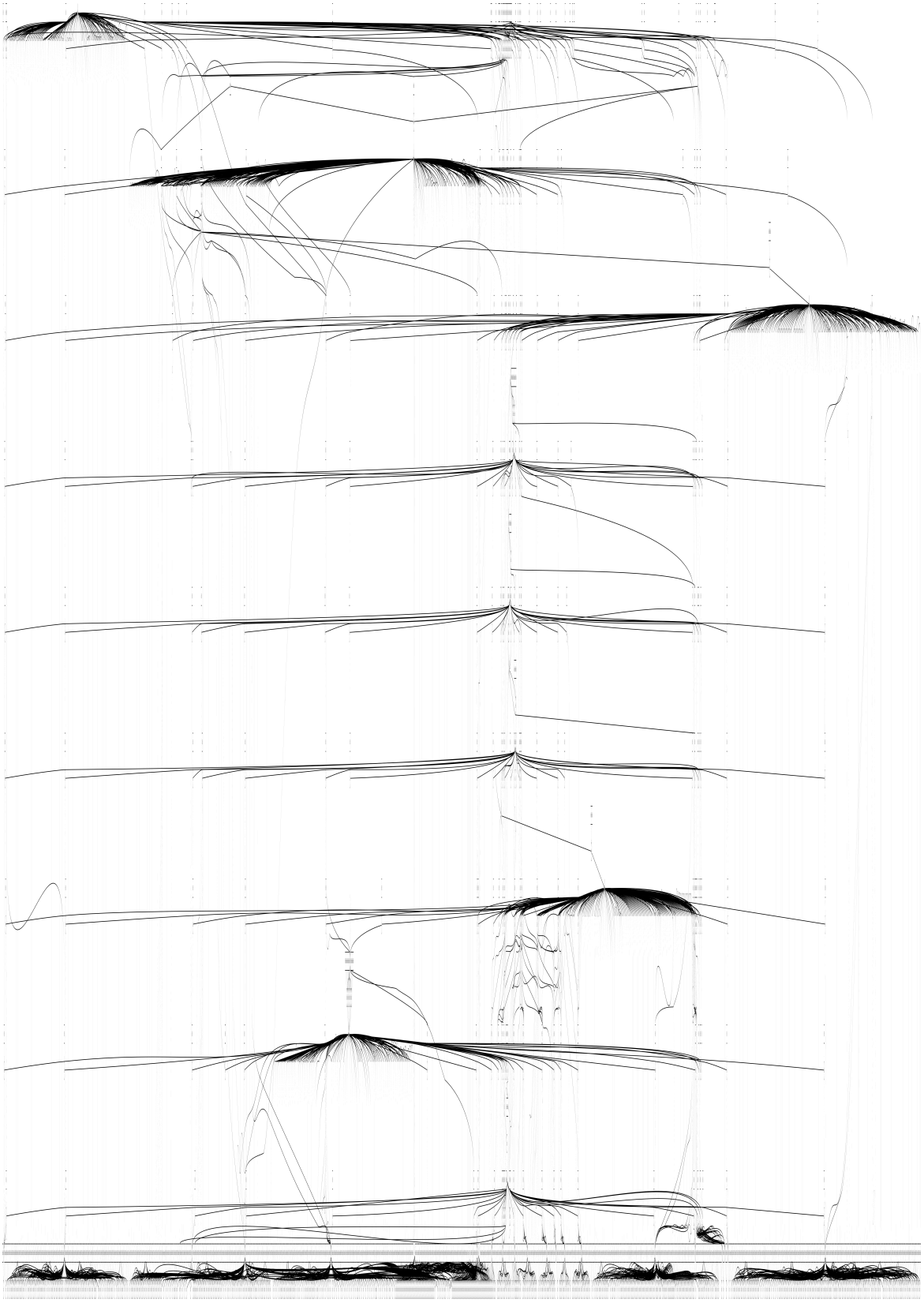


Fig. B.3 Attack graph of 2234 node example

Appendix C

Classifier Families

Below is the listing of each of the possible families, or classes, that a vulnerability can be sorted into.

Listing C.1 All family labels used by the vulnerability classifier

```
"7-Zip Patching"
"Adobe Acrobat Patching"
"Adobe AIR Patching"
"Adobe ColdFusion Patching"
"Adobe Creative Cloud Patching"
"Adobe Digital Editions Patching"
"Adobe Dreamweaver Patching"
"Adobe Flash Media Server"
"Adobe Flash Patching"
"Adobe Illustrator Patching"
"Adobe InDesign Patching"
"Adobe Photoshop Patching"
"Adobe Reader Patching"
"Adobe Shockwave Patching"
"AIX Patching"
"Amazon Linux Patching"
```

```
"Anonymous Cipher Suites Permitted"
"Anonymous FTP Enabled"
"AOLserver"
"Apache ActiveMQ"
"Apache Ambari Patching"
"Apache Cassandra Patching"
"Apache CouchDB Patching"
"Apache Hadoop Patching"
"Apache Solr Patching"
"Apache Struts Patching"
"Apache Subversion Patching"
"Apache Tika Server Patching"
"Apache Tomcat"
"Apache Tomcat Patching"
"Apache Traffic Server Patching"
"Apache Web Server"
"Apache Web Server Patching"
"Apple iCloud Patching"
"Apple iOS Patching"
"Apple iTunes Patching"
"Apple OSX Patching"
"Apple Quicktime Patching"
"Apple Safari Patching"
"Asterisk Patching"
"ASUSTOR ADM"
"Avast AntiVirus Patching"
"Axis Web Camera Patching"
"BIND"
"Bugzilla Patching"
```

```
"Cacti Patching"
"CentOS Linux Patching"
"Cipher Zero Authentication Bypass Permitted"
"Cisco AnyConnect Client Patching"
"Cisco Firepower"
"Cisco Patching"
"Cisco Prime Infrastructure Patching"
"Cisco UCS Patching"
"Cisco WebEx Patching"
"Citrix NetScaler Patching"
"Citrix XenServer Patching"
"ClamAV Patching"
"Comodo Internet Security Patching"
"CouchDB Patching"
"CUPS Patching"
"Debian Linux Patching"
"Dell NetVault Patching"
"Dell Remote Access Controller"
"Deprecated Versions Permitted"
"Discourse Patching"
"DistCC Patching"
"DNS Server Detected"
"DNS Zone Transfer Permitted"
"Dokuwiki"
"Dropbear SSH Patching"
"Drupal"
"Drupal Patching"
"Elasticsearch"
"Elasticsearch Patching"
```

```
"Enhydra Multiserver"
"EPESI CRM Patching"
"Exim Patching"
"F5 Patching"
"Fedora Linux Patching"
"FileZilla Patching"
"Firefox Patching"
"Fortigate Patching"
"Foxit PhantomPDF Patching"
"Foxit Reader Patching"
"FreeBSD Patching"
"Gentoo Linux Patching"
"GeoVision IP Camera"
"GeoVision IP Camera Patching"
"Git Patching"
"Google Chrome Patching"
"Google Sketchup Patching"
"Grandstream VOIP Phone"
"Graylog"
"Host scanned"
"HP Onboard Administrator Patching"
"HP Systems Insight Manager Patching"
"HP Systems Management Homepage Patching"
"HP Version Control Agent Patching"
"HP Version Control Repository Manager Patching"
"HPUX Patching"
"HSTS Not Configured"
"HTTP Public Key Pinning Not Configured"
"Huawei Switch Patching"
```

```
"IBM DB2 Patching"
"IBM WebSphere Patching"
"ICMP Timestamps Enabled"
"ImageMagick Patching"
"Insecure Encryption Protocols Supported"
"Insecure SNMP Configuration"
"Insecurely Configured Windows Service Path"
"Insecurely-Configured etcd Service"
"Insecurely-configured FTP Server"
"Insecurely-configured Windows Service"
"Internet Explorer"
"IPMI Management Service"
"IPMI2 Password Hash Accessible"
"ISC Bind Patching"
"JBoss Patching"
"Jenkins Patching"
"Joomla"
"Joomla Patching"
"Juniper Patching"
"JustSystems Ichitaro Patching"
"Kibana"
"Kibana Patching"
"LibreOffice Patching"
"Lighttpd Patching"
"LimeSurvey Patching"
"Logstash Patching"
"Lotus Notes Patching"
"Mageia Linux Patching"
"Malware / Backdoor Detected"
```

```
"Mandrake Linux Patching"
"Mandriva Linux Patching"
"MariaDB"
"MariaDB Patching"
"McAfee ePolicy Orchestrator Patching"
"McAfee VirusScan Enterprise Patching"
"MD2 Authentication Enabled"
"MediaWiki"
"MediaWiki Patching"
"Memcached Patching"
"Microsoft Exchange"
"Microsoft Exchange Patching"
"Microsoft IIS Patching"
"Microsoft Office Patching"
"Microsoft SharePoint Server Patching"
"Microsoft SQL Server"
"Microsoft SQL Server Patching"
"Microsoft Visio Patching"
"Microsoft Visual Studio Patching"
"Microsoft Windows Patching"
"Mobotix Camera"
"MongoDB"
"MongoDB Patching"
"Mongoose Webserver Patching"
"Moodle Patching"
"Mozilla SeaMonkey Patching"
"Mozilla Thunderbird Patching"
"MS SQL Server"
"MySQL"
```

```
"MySQL / MariaDB"
"MySQL Patching"
"Nagios Patching"
"Nextcloud Patching"
"NFS Exports Easily Accessible"
"Nginx Patching"
"No Authentication Mode Enabled"
"Node.js Patching"
"Null Usernames Permitted"
"Open Xchange Patching"
"OpenOffice Patching"
"OpenSSH Patching"
"OpenSSL"
"OpenSSL Detected"
"OpenSSL Patching"
"OpenSUSE Linux Patching"
"OpenVPN Access Server"
"Opera Patching"
"Operating System - End of Life"
"Oracle Database Patching"
"Oracle GlassFish Patching"
"Oracle Java Patching"
"Oracle Linux Patching"
"Oracle Virtualbox Patching"
"ownCloud"
"ownCloud Patching"
"Paessler PRTG Patching"
"Palo Alto Patching"
"Perfect Forward Secrecy Not Supported"
```

```
"pfSense"
"pfSense Patching"
"PHP"
"PHP Patching"
"PHPMailer Patching"
"phpMyAdmin"
"phpMyAdmin Patching"
"PHPUnit Patching"
"Pidgin Patching"
"POP3 Mail Server Detected"
"Portainer Patching"
"Postgresql"
"PostgreSQL"
"PostgreSQL Patching"
"PrestaShop Patching"
"ProFTPD Patching"
"Puppet Patching"
"PuTTY Patching"
"QNAP Patching"
"RealNetworks RealPlayer Patching"
"RealVNC Patching"
"Red Hat Enterprise Linux Patching"
"Red Hat Linux Patching"
"Redis Patching"
"Redis Server"
"Ruby On Rails Patching"
"Ruby Patching"
"Samba"
"Samba Patching"
```

```
"Scientific Linux Patching"
"Simple Machines Forum Patching"
"Skype Patching"
"Slackware Linux Patching"
"SLES Linux Patching"
"SMB"
"SMB Signing Not Enabled"
"Solaris Patching"
"Sophos Anti-Virus Patching"
"Splunk Patching"
"Squid Patching"
"SquirrelMail Patching"
"SSH"
"SSL Certificate Expired"
"SSL Certificate Signed With Weak Signature Algorithm"
"SSL Certificate With Expired Certificate In Chain"
"SSL Certificate With Small Public Key"
"SSLv2 and SSLv3 Deprecated"
"SuSE Linux Patching"
"Sybase ASA"
"Symantec BackupExec Patching"
"Symantec Endpoint Protection Client Patching"
"Symantec Endpoint Protection Manager Patching"
"Symantec MessagingGateway Patching"
"Symantec PGP Desktop Patching"
"Symantec WebGateway Patching"
"Symfony Patching"
"TeamViewer Patching"
"Telnet Service Detected"
```

```
"TikiWiki Patching"
"Trend Micro Internet Security Patching"
"Trend Micro InterScan Patching"
"Trend Micro OfficeScan Patching"
"TWiki Patching"
"Twonky Media Server Patching"
"TYPO3"
"TYPO3 Patching"
"Ubuntu Linux Patching"
"Unencrypted Network Service"
"UPnP Service Detected"
"Users Never Logged On"
"Users With Unchanged Passwords"
"vBulletin Patching"
"VirtualBox Patching"
"VLC Patching"
"VMware ESXi Patching"
"VMWare Fusion Patching"
"VMware Player Patching"
"VMware vSphere Client Patching"
"VMware Workstation Patching"
"VNC Patching"
"VNC Server"
"vsftp Patching"
"Weak Encryption Algorithms"
"Weak Encryption Ciphers Permitted"
"Weak MAC Algorithms"
"Web Server Detected"
"Webmin Patching"
```

```
"Winamp Patching"
"Windows Autorun Enabled"
"WinZip Patching"
"Wireshark Patching"
"WordPress"
"Wordpress Patching"
"WS FTP Patching"
"WU-FTPd Patching"
"Zimbra Patching"
```