Distributed Log Analysis for Scenario-based Detection of Multi-step Attacks and Generation of Near-optimal Defense Recommendations

vorgelegt von
MSc.
Kerem Kaynar
geb. in Ankara, Türkei

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 19. September 2017

Berlin 2017

Zusammenfassung

Die Erkennung verwandter, laufender Aktionen von Angreifern ist bedeutend für eine umfassende Situationsbewertung der Netzwerksicherheit und die Bestimmung der wirksamsten, reaktiven Gegenmaßnahmen für ein Netzwerk. Die Log-Einträge, die von den Software generiert werden, die auf das Netzwerk laufen, können zu diesem Zweck verwendet werden, da sie die Spuren von bösartigen Aktivitäten enthalten können, die innerhalb des Netzwerks auftreten. Primäre Log-Einträge, die Sicherheitsalarme wie die Ausnutzungen der Schwachstellen angeben, können verwendet werden, um die Angriffspfade zu berechnen, die von einem potenziellen Angreifer wahrscheinlich verfolgt werden. Angriffsgraphen werden eingesetzt, um die Angriffspfade darzustellen. Einer der Hauptbeiträge dieser Doktorarbeit ist der Vorschlag eines verteilten Algorithmus zur Generierung der Angriffsgraphen, der das Skalabilitätsproblem beseitigen kann, das in der Angriffsgraphberechnung für auch mittlere Netzwerke inhärent ist. Sekundäre Log-Einträge sind nicht direkt mit Sicherheitsalarmen verknüpft. Die Verwendung dieser Log-Einträge kann jedoch einen besseren Einblick in die Aktivitäten der Angreifer ermöglichen. Wir tragen zur Verarbeitung der sekundären Log-Einträge durch die Generierung von Verhaltens-Malware-Signaturen und die Übereinstimmung von diesen Malware-Signaturen an die sekundären Log-Einträge bei. Die instantiierten Malware-Signaturen werden mit den berechneten Angriffsgraphen integriert. Die Skalabilitätsprobleme, die durch das hohe Volumen an sekundären Log-Einträge verursacht werden, werden durch die Nutzung einer streambasierten Big Data-Infrastruktur gemildert. Die Bestimmung der entsprechenden Reaktionen auf die anhaltenden Angriffsszenarien wird durch die Anwendung einer Optimierungsmethode ausgeführt, die eine speziell gestaltete Kandidatenauswahlfunktion und die berechneten Angriffsgraphen verwendet.

Abstract

Detecting related, ongoing actions of attackers is significant for providing a complete situational assessment of security and determining the most effective reactive defense measures for a network. The logs generated by software running across the network can be used for this purpose, since they may contain the traces of malicious activities occurring inside the network. Primary logs that indicate security alerts such as vulnerability exploits can be used to compute the attack paths that are likely being followed by the attackers. Attack graphs are utilized to represent the attack paths. One of the main contributions of this thesis work is the proposal of a distributed attack graph generation algorithm eliminating the scalability problem inherent in attack graph computation for even medium scale networks. Secondary logs are not directly related to security alerts. However, the usage of these logs can provide more insight into the activities of the attackers. We contribute to the secondary log processing by generating behavioural malware signatures and matching them to the secondary logs. The instantiated malware signatures are integrated with the computed attack graphs. The scalability problems caused by the high volume of secondary logs are alleviated by utilizing a stream-based Big Data infrastructure. Response to the ongoing attack scenarios is performed by applying an optimization method that utilizes a specifically designed candidate selection function and the computed attack graphs.

Acknowledgments

I would like to thank Prof. Dr.-Ing. Sahin Albayrak for his precious support for this work from the creation to the completion.

Likewise, I would like to thank Dr.-Ing. Karsten Bsufka and Dr. Brijnesh Johannes Jain for their precious comments, suggestions and support while forming this work.

I would like to thank all my colleagues at the DAI-Labor and GT-ARC for their support.

Most importantly, I would like to thank my family for their support and everything.

Contents

List of Figures

List of Tables

1 Motivation

Nowadays, almost all corporates' security administrator teams deploy perimeter security appliances and anti-virus software in order to protect their information networks against malicious software originating from outside or inside their networks. As the size of the corporate networks, the variability of the software used in these networks and the interconnections among the networks increase, managing the network security by using isolated security applications becomes more difficult. The need for integration of different security applications to create a global security analysis framework for a corporate network arises. Such a global security analysis framework can be used to assess the online security situation of the network by detecting ongoing attacks.

The global security analysis framework for a network investigates the existence of attacks directed to the network, their potential propagation paths and payloads. It tries to react to ongoing attacks on time and also find the sources of attacks to determine the identity of the attackers. To achieve these aims, the global security analysis framework can use the logs/events collected from various log/event sources on the network. The logs/events can be obtained from the alerts generated by the security sensors and the execution traces of various software on the network hosts, among others. They should be processed in near real-time to detect and react to ongoing attacks which can be performed simultaneously by one or more groups of coordinated attackers.

The concept of an attack scenario that collects related atomic attack actions that are performed by a single group of coordinated attackers plays an important role for global, network-wide security analysis. Atomic attacks are generally represented by vulnerability exploits. Atomic attacks employed in an attack scenario serve as a whole to the aim of compromising common targets. An attack scenario shows the development of related attacks and can be utilized to find the possible sources of attacks. It is generally represented by an attack graph that shows vulnerability exploits and attacker privileges used to apply these exploits and gained after successfully exploiting the vulnerabilities. There can be more than one attack scenario targeted to a network during the same time interval.

The aim of this thesis work is to devise a system for detecting ongoing attack scenarios for a target network via analyzing logs/events that are generated by the software installed on the network machines. The system provides also support for determination of near optimal defense measures to eliminate as much as possible the propagation of attacks performed in the scope of the detected attack scenarios. Support for analyzing large-scale networks is provided in the system via designing and implementing the components of the system in such a way that allows the execution of them over distributed environments.

1.1 Why is the Attack Scenario Detection Process Important?

An attack scenario represents a set of related atomic attacks performed by a coordinating group of attackers. Detecting the attempts of related atomic attacks and the relations among them give clues about which privileges the attackers have already obtained on the target network hosts or any hosts that can reach to the target network hosts. The determination of the privileges already obtained by the attackers makes prediction of future attack attempts that can be performed by these attackers possible. The possible propagation paths for the currently active attacks can also be predicted. This future attack and propagation path predictions can be utilized in selecting the most effective defense measures among the available ones, which can stop further propagation of the currently active attacks and render possible future attack attempts infeasible. The more focused the predicted possible future attack attempts and propagation paths are, the more effective and accurate the recommended defense measures are and it becomes possible to recommend a more targeted defense measure configuration that can stop further attacks with less number of defense measures and side effect on the current operation of the network.

Another benefit of the determination of the privileges already obtained by the attackers is that the infected network machines which are not detected by isolated, local security sensors deployed across the network can be detected. Because, the attack scenario detection process requires/employs global analysis methods that can hypothesize the security events missed by local security sensors. The hypothesized events can give clues about additional infected machines on the network, the infection mechanisms applied to them by the attackers and the effects of the malicious activities on these machines. This allows to determine the appropriate actions in order to quarantine these machines and revert the effects of the malicious activities on them. This contributes to the elimination of further propagation of attacks.

1.2 What is Important in the Attack Scenario Detection Process?

The most important point in the attack scenario detection process is to recognize the related activities and goals of the attackers accurately as early as possible. This is not so easy under all circumstances, since at the early stages of the attacks, the collected logs are so dispersed that constructing an attack scenario by relating them can be very difficult. Most probably, the attacker performs some network reconnaissance activities (such as port scanning or fingerprinting) at these early stages, so the generated logs may be spread across the whole region that is accessible remotely by the attacker. For instance, the attacker may perform a random IP scan and get port or operating system (OS) information about some hosts on the Demilitarized Zone (DMZ) of the corporate network. At these early stages, the attacker has most probably no idea about which paths she has to follow to reach the target hosts in her mind, so she may have to perform some random trials. For an insider attacker, the situation may be similar. She may have to find an intrusion point through a series of access-controlled sub-networks in order to reach her target. Therefore, the collected logs may not differentiate specific attack paths at this stage.

After taking some preliminary attack steps, the attacker starts to utilize the vulnerabilities she finds, when there is such an opportunity. Exploiting vulnerabilities, gaining privileges on software and obtaining more information can cause the attacker to be more selective in her future actions to reach her goals. After that point, the traces that her attacks left on the software on the network can be easily correlated to differentiate specific attack paths. It is crucial to extract these paths indicating ongoing attack scenarios in a timely manner. This is important, because the security administrator shall employ the required counter-measures in a timely manner in critical situations. In order to provide support for timely attack scenario detection even for large-scale networks, the attack scenario detection process can be performed in a fully distributed manner. In this case, the distributed agents spread across the target network need to combine their results and deduce accurate attack scenarios in collaboration with each other.

Another important point in the attack scenario detection process is to filter false and unrelated alarms as much as possible. These alarms may have been generated as the side effects of a malicious software, or they may be generated because of the imperfectness of the security applications. Even, they may be generated as a result of the deliberate, deceptive actions performed by the attackers. These alarms can be eliminated by extracting the cohesive, highly connected set of actions from the collected alarms/logs. This can be achieved by defining and searching for possible relationships among the alarms/logs at a fine-grained level.

1.2.1 Timely Detection and Prevention of Attack Scenarios

Distributed computation techniques can be utilized to provide for timely detection of attack scenarios. The aggregation and correlation of the collected logs/events are performed by using a distributed, real-time middleware in order to determine the related sequences of attack actions on time. Since an attack graph is used to represent an attack scenario, the computation of attack graphs is also performed in a distributed manner.

The process of defense measure recommendation takes place after detecting an ongoing attack scenario. The recommended defense measures should incur minimal cost and eliminate maximum number of attack paths in an attack scenario that allow the attacker gain privileges on critical network resources. The defense measure recommendation problem can be considered as an optimization problem. The near-optimal defense measures should be found fast enough to prevent further propagation of ongoing attacks. For this purpose, prominent optimization algorithms can be enriched with network security domain-specific functionalities and heuristics.

1.2.2 Accurate Attack Scenario Detection

The logs collected from the software on the target network are correlated to detect ongoing attack scenarios. The logs can be of two types. A primary log is a log directly related with a security (violation) event. It is generally generated by security software. A secondary log is a log or network trace that does not directly indicate a security event. It is generally generated by software other than security software. A collection of more than one secondary log can indicate a security event.

Primary logs mostly indicate infection vectors, namely vulnerability exploit attempts. They can also indicate the existence of a malware on a network host (trojan, virus, e.g.). A collection of secondary logs mostly indicate the actions taken by an attacker, after infecting the target software. These actions can seem innocent when individually analysed, but as a collection they can indicate the usage of a number of privileges by the attacker for malicious purposes. In order to provide for accurate determination of ongoing attack scenarios, both primary and secondary logs should be utilized. Specific signature or anomaly-based techniques can be devised and used to find the security events implied by a collection of secondary logs. The found security events can be correlated with the ones directly indicated by the primary logs in order to obtain accurate attack chains.

2 Approach

We try to detect ongoing attack scenarios using the logs generated by various applications (operating systems, server/client applications, security sensors, etc.) running on the target network. An attack scenario is designated by a partial attack graph. An attack graph contains as nodes vulnerability exploits (infection vectors) and privileges gained by an attacker on applications running on the network. Each node of an attack graph can be in alarmed status or not. A node being in alarmed status means that the existence of the node (vulnerability exploit or privilege) is implied by a primary log or a collection of more than one secondary log. Vulnerability exploit nodes are implied by primary logs. Privilege nodes indicate the usage of the corresponding privileges by an attacker for malicious purposes and can be implied by a primary log or a collection of secondary logs. Section 2.1 contains detailed information about attack graphs, their generation methods and the contributions of this thesis in this subject.

In order to find the privileges used by an attacker for malicious purposes, we should correlate the collected secondary logs in addition to analysing the primary logs. A signature-based technique from the domain of malware behaviour analysis is developed and applied to correlate the secondary logs. We generate and use behavioural malware signatures in this context. By using the behavioural malware signatures, we try to provide for more resistance to the evasion techniques utilized by the attackers to hide their malware from detection. Section 2.2 describes behavioural malware signatures and the contributions of this thesis in generating them.

The volume of secondary logs can be huge for even a moderately sized network. The matching of secondary logs to the generated behavioural malware signatures should be performed in near real-time to determine alarmed privileges on time. The high volume of secondary logs can be a bottleneck in this respect. We use a distributed, stream-based Big Data framework to provide for near real-time processing of the collected secondary logs. The related log correlation algorithms are designed and implemented over this framework. The correlated logs are used to form attack scenarios in the form of attack graphs, each of which represents a group of related (coordinated) actions taken by the attackers. The approach of detecting ongoing attack scenarios by correlating logs and contributions of this thesis in this subject are described in Section 2.3.

We treat the (near optimal) defense measure recommendation problem as a single objective optimization problem. The criteria forming the objective for the optimization problem are to minimize the total cost of the applied defense measures and to maximize the number of the attack paths on the input attack graph which are eliminated by the applied defense measures. We propose a new candidate selection function to be used in the execution of the selected optimization algorithm. By using this candidate selection function, near optimal sets of defense measures can be found in a reasonable time even for large attack graphs. The near optimal defense measure recommendation problem and contributions of this thesis in this respect are described in Section 2.4.

2.1 Representing Relations among Vulnerability Exploits and Privileges

An attack graph shows the possible paths that an attacker can follow to intrude into a network and gain certain target privileges, given a set of initially satisfied attacker privileges as input. It generally represents the relationships among various vulnerability exploits that can be employed by the attacker and the privileges gained by the attacker as a result of exploiting these vulnerabilities successfully. The term vulnerability is treated in different ways in attack graph generation literature. In some cases, the existence of a bug in a specific version of a software/hardware product that allows an attacker to compromise the software/hardware product is considered as a vulnerability. In other cases, an inappropriate configuration setting for a software/hardware product may be considered as a vulnerability. In this work, the vulnerabilities defined in National Vulnerability Database (NVD) [50] are used. In this respect, a vulnerability can be considered as a bug in a specific software/hardware product that allows an attacker to affect the confidentiality, integrity or availability of data managed by the computer system containing the software/hardware product.

The number of vulnerabilities on the target network, the reachability conditions among the vulnerable software instances on the network and the level of detail in vulnerability modeling are the main factors determining the size of the state space for the attack graph computation process. States in this context correspond to nodes in the resulting attack graph, which generally represent attacker privileges and vulnerability exploits. When the state space becomes too large, the popular serial attack graph generation algorithms, running on a single computer and generally employing hashes or other in-memory data structures to hold the already traversed parts of the state space, may become ineffective.

In order to compute vulnerability-based attack graphs, the pre- and post-conditions of possible vulnerabilities shall be generated in advance. The preconditions of a vulnerability determine the necessary privileges that an attacker must own to exploit the vulnerability, while the postconditions are the privileges gained by the attacker, after successfully exploiting the vulnerability. The determination of which facts can be a privilege is an issue that should be handled before the attack graph building process. The level of detail of these facts affects the chaining of vulnerability exploits in the attack graph building process. If the level of detail is very high, it becomes difficult to chain unrelated vulnerabilities, because pre- and postconditions of vulnerabilities are more specific (for instance, specific to an application's module name and version). The detailed, fine-grained extraction of the pre-and post-conditions for vulnerabilities may become an error-prone, cumbersome task, if it is performed manually.

In order to compute attack graphs, a model of the target network shall be created, which contains the network topology, information about the software installed on the network hosts and the necessary network configuration information to compute the reachability conditions among the network hosts. The reachability conditions are affected by the filtering rules and security policies employed across the network, among others. The information about the software installed on the network hosts can be used to determine the vulnerabilities of them. The information sources contained by the software are also important, since they can contain information referring to other software. As examples, the cookies file of a specific web browser can be treated as an information source of the web browser, a database storing sensitive (login) information and managed by a database server can be considered as an information source of the database server.

2.1.1 Main Contributions in the Context of Attack Graphs

Our main contribution in the context of attack graphs is related to the *Attack Graph Core Building* phase and we try to improve the time and space complexity of the problem by applying a parallel and distributed search algorithm to cope with the potential state explosion problem in the resulting attack graphs.

A hyper-graph is used to represent the reachability conditions among the networked software applications, in which a hyper-vertex represents a networked software application and a hyper-edge indicates a collection of networked software applications which can access each other using a specific network protocol. One of the main contributions of this thesis work is to perform reachability hyper-graph partitioning to guide the distributed search performed to generate an attack graph. Distributed search is performed on a multi-agent platform and the tasks assigned to each agent is determined according to the results of partitioning the reachability hyper-graph. Therefore, the tasks of the determination of the attacker privileges on networked software applications are distributed over agents in such a way that an agent can perform attacker privilege determination on the networked software applications that are closely connected. The main aim of this kind of task distribution among the agents is to minimize the required communication among the agents. (Communication is needed among the agents for exchanging knowledge about the current privileges of the attacker and using this knowledge to decide the exploitability of a vulnerability. In order to compute the exploitability of a vulnerability on host h, agent a requires information about the privileges already gained by the attacker on the hosts that can reach host h. This information may have been stored in another agent and should be transferred to agent a.) The list of networked software applications handled by an agent can also be changed dynamically at run-time.

Another contribution of this thesis is to provide a virtual shared memory abstraction across distributed agents to avoid multiple expansion of nodes (multiple expansion of the attacker privileges) during the generation of an attack graph. The algorithm described in [40] is used to provide a virtual shared memory abstraction over a distributed multi-agent system. One of the important parameters of a virtual shared memory abstraction is the initial allocation of memory pages (or, memory objects) to the distributed agents. This is performed by partitioning the reachability hyper-graph, so that the more closely connected any two networked software applications, the more likely that the privileges that can be obtained on them are stored in the same memory page and thus handled by the same agent without having to transfer memory pages from other agents.

2.2 Representing Malicious Activities after Vulnerability Exploits

The relations among the vulnerability exploits (infection vectors) and attacker privileges are computed and displayed in the form of attack graphs. The existence of a vulnerability exploit can be deduced by analysing the primary logs generated mainly by intrusion detection/prevention systems. The existence of an attacker privilege can be deduced by analysing the primary logs or tracking and correlating the activities indicated by the secondary logs. These activities may represent the malicious activities performed by the attacker, after exploiting a vulnerability (infecting a system). In this thesis work, these activities are correlated by using the *behavioural malware signatures* derived by processing the:

- malware reports (behavioural profiles) downloaded from the web site *https//:malwr.com*,
- malware reports generated manually for sample malicious files by using a dynamic malware analyser called ANUBIS [39].

The number of malware variants has been increasing at a higher rate than before due to the rich set of publicly available malware generation and obfuscation tools on the Internet. Static malware detection methods utilizing malware signatures in the form of byte sequences are insufficient today to detect possibly a huge number of different variants of the same malware. Packing, binding, encryption, instruction resequencing, using syntactically different but semantically equivalent instructions, nop (no operation) instruction insertion, etc. are among the well-known malware obfuscation techniques that can be used to evade detection by static malware analysers.

An obvious alternative to static malware detection is dynamic malware detection in which an input binary is executed in a controlled environment (sandbox) and a behavioural profile is generated for the binary. A behavioural profile may contain system call sequences executed by the binary, however detection via using such a profile is also prone to malware obfuscation techniques (system call reordering, insertion or change without changing the semantics of malware, etc.). In order to eliminate the effects of these obfuscation techniques, a behavioural profile may be composed of a set of high level operations performed by the input binary. These operations can be registry record read/change, file creation/read/modification, process creation/termination, service start-up, setting up an HTTP connection, sending an e-mail, etc. There is generally no temporal dependencies among the high level operations contained in a behavioural profile.

Malware behavioural profiles consisting of high level operations can be clustered and a malware behavioural signature can be generated for each cluster to be used in the online malicious activity detection process. One of the main problems in this behavioural signature generation process is caused by the insertion of deceptive high level operations into the malware binaries by the attackers. Each variant of a malware binary may perform a number of misleading (tricky) high level operations to hide its main purpose. In order to eliminate the adverse effects of deceptive high level operations in malware behavioural signature generation, we propose a graph similarity-based approach to the processing of malware behavioural profiles. We provide support for the distributed computation of the applied graph similarity algorithm by utilizing graph partitioning.

2.2.1 Main Contributions in the Context of Behavioural Malware Signatures

Our main contributions in this thesis work for malware behavioural clustering and signature generation are:

- We represent the behavioural profile for the malware binary as a malware graph. A node of the malware graph represents a malicious activity performed by the malware. An edge represents a relation between the values of the fields of the source and target malicious activity. By this way, we account for the dependencies among the malicious activities performed by the malware.

- We apply a specific graph similarity algorithm to find the structurally and contextually similar parts of a set of malware graphs. These similar parts serve as the malware behavioural signatures. Contextual similarity refers to the similarity in terms of the field values of the malicious activities in the malware graphs. Structural similarity refers to the similarity of the patterns of connections among the malicious activities in the malware graphs. The aim of applying graph similarity to the malware graphs is to eliminate deceptive malicious activities (behavioural components) as much as possible.

- In order to support for distributed generation of malware behavioural signatures, we utilize graph partitioning to separate the highly connected sub-graphs of a malware graph and compare them with the highly connected sub-graphs of other malware graphs by using our specific graph similarity algorithm.

2.3 Correlating Network-wide Logs: Attack Scenario Detection

The concept of an attack scenario that collects related atomic attack actions that are performed by a single coordinated group of attackers plays an important role for the network security analysis. An attack scenario serves as a whole to the compromise of common targets. It shows the development of attacks and can be utilized to find the possible sources of attacks. There can be more than one attack scenario targeted to a network during the same time interval.

In this thesis work, the attacks targeted to a network are grouped into attack scenarios by collecting and analysing logs generated throughout the network. Two types of logs are defined: primary and secondary logs. A primary log indicates directly a malicious activity performed by an attacker. A secondary log is a normal event that does not indicate a malicious activity by itself. However, a collection of more than one secondary logs can indicate a malicious activity as a whole.

The main challenge to construct attack scenarios is to summarize the huge number of collected logs into a number of groups that contain correlated logs and indicate malicious activities. For each malicious activity, a number of security events are generated and then appropriate attack scenarios are created and updated with the security events. The previous works use different log correlation models to generate security events and create/update attack scenarios. Some of them use pre-defined or statistical similarities among the attributes of the logs. Some others use pre-defined attack scenario templates and groups the logs according to the components of the templates. Even some others define pre- and postconditions for atomic attacks (vulnerability exploits) to generate attack graphs representing attack scenarios. Attack graph-based approaches are more effective for attack scenario detection, since they can detect new (zero-day) attack scenarios. In attack graph-based approaches, the existence of a vulnerability exploit node is deduced by analysing primary logs. The existence of an attacker privilege can be deduced by analysing both primary and secondary logs. When the existence of a vulnerability exploit or an attacker privilege is deduced by a collection of logs, the corresponding node on the attack graph is used to create/update attack scenarios.

There are important points to be improved in previous works that employ attack graph-based approach to detect ongoing attack scenarios. The ones that are addressed in this thesis work are:

- The previous attack graph-based approaches generally do not process secondary logs, because of their possible huge volume. However, some of these logs can collectively indicate a malicious activity and can provide for the creation/update of the nodes on the attack scenarios that are referred to by the malicious activity.

- The previous attack graph-based approaches generally mark the nodes of the pre-computed full attack graph which are implied by the collected primary logs as alarmed. Then, they find the (shortest) paths among the alarmed nodes over the full attack graph. This gives rise to the traversal of the parts of the full attack graph that are unrelated with the alarmed nodes. When we consider a huge full attack graph with some alarmed nodes scattered on the attack graph far away from each other, these approaches may have to traverse a huge portion of the attack graph that do not contain alarmed nodes in order to find paths among the alarmed nodes.

2.3.1 Main Contributions in the Context of Attack Scenario Detection

We provide a method to construct attack scenarios by processing the collected logs. Our method processes the secondary logs in addition to the primary logs in order to provide for more accurate attack scenario detection. The processing of the collected primary and secondary logs is performed over a real-time, stream-based Big Data framework by using specifically developed distributed algorithms (application topologies). The primary logs are used to prove the existence of the vulnerability exploits. We generate behavioural malware signatures for the specific categories of malicious activities such as mass mailing worms, trojans, adwares, etc. The secondary logs are matched to these signatures. If these matchings are successful, we conclude that the matched secondary logs collectively indicate a malicious activity. After that, by using the indicated malicious activities, we prove the existence of the specific privileges that could have been possibly used by an attacker for malicious purposes. These privileges are attacker privilege nodes on the attack graph.

We construct attack scenarios by using partial attack graphs which are dynamically created/expanded during the log correlation/processing. The nodes in partial attack graphs, whose existence are proved by the primary or secondary logs, are found. These nodes are called *alarmed nodes*. The expansion of a partial attack graph is performed using the pre-computed full attack graph. We determine the nodes to be expanded in a partial attack graph by using their alarmed status and their distance to the alarmed nodes, among other factors. By this way, we can also include the unalarmed nodes, which are *most likely* missed by IDSs, in the expansion process. With this partial attack graph-based approach, we try to eliminate as much as possible the traversal of the parts of the full attack graph that are unrelated with the received logs during attack scenario construction.

2.4 Responding to Attacks

With the growing need for connectivity inside and among the enterprise networks, it becomes ever more challenging to ensure the confidentiality, integrity and availability of data. Therefore, the accurate assessment of the interactions among the possible network defense or *network hardening* measures becomes crucial. This assessment is dependent on the target network configuration and plays a crucial role in determining the optimal sets of hardening measures for the target network. An application of a specific hardening measure may eliminate the need for a set of other measures, if the measure can subsume the utility gained by the application of those measures. In addition to the role of the target network configuration, the selected defense criteria to minimize or maximize are also important to determine an optimal network hardening plan.

In this thesis work, the problem of finding the optimal hardening measure set by using an attack graph is defined as a single-objective optimization problem, which is then solved using the simulated annealing algorithm. The objective function is a linear combination of the two optimization criteria that represent the minimization of the total cost of the applied hardening measures and the maximization of the number of the eliminated attack paths. Their relative weights would depend on the target network configuration and preferences that can be adjusted by the network security administrators.

2.4.1 Main Contributions in the Context of Network Hardening

The main contribution of this thesis work is the proposed candidate selection function used in the optimization process, where each candidate solution represents a set of hardening measures. The aim of the candidate selection function is to prevent the optimization process from being caught in local optima easily and help the optimization process find the near-optimal solutions in a reasonable time frame.

3
State of the Art

This section explains the approaches proposed by the prominent works in the literature in the context of four main subjects:

- behavioural malware signatures
- ongoing attack scenario detection by correlating alerts and logs
- security information and event management systems
- optimal network hardening (defense) measure recommendation.

The works related with each subject are discussed in the separate subsection for the subject. The state of the art for the generation of attack graphs is described in Section 5, which describes the taxonomy for the whole attack graph generation process proposed in this thesis work (by referring to the related past works).

3.1 Behavioural Malware Signatures

In this section, we describe the related work for dynamic (behavioural) malware analysis and malware clustering approaches which use the results of dynamic malware analysis. The proposed method for malware clustering in this thesis work is based on high-level, behavioural, dynamic malware analysis results in order to eliminate the possible negative effects of well-known malware obfuscation techniques, such as instruction (system call) reordering, adding nop instructions, instruction (system call) renaming etc. on other malware analysis methods. Actually in [46], a binary obfuscation and rewriting method to emphasize the insufficiency of static malware analysers is proposed. The method allows to change the binaries of malwares to hide their existence from analysis by both pattern-matching- based and semantics-aware static malware detectors. The main obfuscation method proposed in [46] is based on the famous 3-satisfiability NP-complete decision problem (3SAT) in Boolean logic. Opaque constants that cannot be determined in a reasonable time by static malware analysers are created. To determine each bit of the value of a constant, the malware analyser must solve a specific instance of the 3SAT problem. The authors propose techniques that incorporate 3SAT-based opaque constants in binaries in order to hide the target of jump, call and branching instructions, the reached data locations and usage patterns, and also the locations of loaded shared (or dynamic link) library functions.

Yin et al. [94] use a QEMU-based emulator as an environment to test, monitor and analyse the information access patterns and information processing behaviour of input unknown sample codes. Sensitive information is introduced into the environment in a test case and marked with a taint. During the execution of the test case, the propagation of the taint among the operating system objects and resources is recorded as a taint graph. Taint graphs represent the results of the dynamic malware analysis.

In [88], input unknown sample code is executed in a controlled simulation environment, called CWSandbox, instruction-by-instruction. As the results of this execution, the processes spawned, file entries changed, registry entries changed by the sample code are automatically derived. CWSandbox uses Windows API hooking and DLL injection to catch the implicit and explicit DLL function calls made by the input sample and analyse the DLL function call parameters. By this way, a behavioural profile for the input sample is created and reported. The usage of Windows API hooking and DLL injection mechanisms also prevents the malicious code from detecting the controlled simulation environment, namely prevents the malware from realizing being tracked. It is also stated that CWSandbox can detect the network connections created by malicious code to download new malicious applications.

Bayer et al. [39] propose a malware clustering system based on malware behaviour. Malware samples are fed into a sandbox ANUBIS supporting dynamic taint analysis and advanced network analysis capabilities to obtain system call traces. The system call traces are converted into a more abstract representation called behavioural profiles. A behavioural profile is comprised of the operating system (OS) objects and operations performed by a malware sample. It also contains the dependencies between OS operations. An OS object can be a file, registry key, synchronization object (mutex) or a network socket, among others. An OS operation is an abstraction over the system calls having a similar functionality.

In [39], the features for the clustering algorithm are generated from the behavioural profiles by using the string representations of OS objects and operations and string concatenation. However, string concatenation can not accurately represent the graph structure inherent in a behavioural profile. The order of string representations of the OS objects and operations in the concatenated string may cause inaccurate distance computations among the behavioural profiles. It may also not eliminate deceptive (tricky) OS operations and objects in the behavioural profiles when generating signatures. Complete-linkage hierarchical clustering algorithm is used for clustering the behavioural profiles. To avoid quadratic amount of distance computations among the feature sets, the locality sensitive hashing algorithm is used.

Apel et al. [5] compare various distance measures to measure their efficiency in clustering malware behavioural traces. Among the compared measures are the graph edit distance, approximated edit distance, normalized compression distance with different compression algorithms and Manhattan distance. Collected malware samples are analysed with CWSandbox and the behavioural malware traces for them are obtained. A malware trace contains system call sequences. The distance measures are compared in terms of three qualitative criteria: ensuring appropriateness by creating minimum number of malware clusters with diverse shared behaviour, ensuring sensitivity to system call reordering and having acceptable runtime performance.

Rieck et al. [66] propose a malware behaviour analysis framework combining clustering and classification approaches. The clustering approach is used to find novel classes of malware from the samples with similar behaviour. The classification approach is used to match unknown malware binaries to known (previously determined) classes of malware. The proposed framework uses the CWSandbox tool to determine the system call sequences performed by a binary. These sequences are embedded into a vector space denoted by the proposed malware instruction set. The distances among the generated vectors are computed during the clustering algorithm. The proposed method may not handle system call reordering employed for malware obfuscation, since it uses system call sequences. It also can not catch the dependencies among the different system call sequences performed by a malware binary.

In [8], a malware classification system is proposed, which uses the changes in the behaviour of an operating system incurred by malware. It uses the performance monitor data, frequency of single system calls and pairs of system calls as features. For each malware sample execution, the feature data are collected for a tunable amount of time before and after the execution time of the malware sample as two separate data sets. For each feature data set, the normalized changes in the mean and variance values are computed. The normalized changes are given as input to a decision tree-based, supervised classifier algorithm to classify malware samples. Decision trees are given as input to a specific random forest algorithm for training them. Feature selection is applied during the execution of the random forest algorithm. The method is prone to the malware obfuscation techniques employing system call reordering and does not catch dependencies among the system call parameters.

Borojerdi et al. [62] introduce a malware classification tool, named MalHunter, which uses high-level semantic behavioural sequences instead of only system calls. System calls in malware traces are grouped into behavioural sequences. After that, using the normalized edit distance metric, the behavioural sequences are clustered. For each cluster, a number of (possibly more than one) malware behaviour patterns are formed. The tool does not account for the relations among the different behavioural sequences of a single malware trace. So, it is incapable of distilling deceptive behavioural sequences from a malware trace. (Deceptive behavioural sequences can be added by a malware writer to hide the real behaviour of the malware and they are mostly isolated from other malware-mission-specific behaviour.)

In [47], a malware classification method based on network traces of malware is proposed. The network traces of a malware are converted into network flows. A behavioural malware profile in the form of a graph is formed using the network flows. The graph contains the network flow information as nodes and the dependencies between the different network flows as edges. The dependencies are created using the source and destination IP addresses. The authors define a set of features on malware behavioural profile graphs. These features are processed by a supervised classification algorithm. To create a training set for the supervised classification algorithm, the authors have to collect the network traces for a set of malware samples and label each of them with the appropriate malware family name.

3.2 Attack Scenario Detection Using Logs

There are various works in the literature concentrating on the management and correlation of events, alerts, network traces and other kinds of logs generated throughout a computer network. The aim is generally to determine and evaluate the global security status of the network. Most of these works correlate intrusion alerts to generate attack scenarios. The attack scenarios are generally in the form of attack graphs representing multi-step attacks.

In [26], the authors propose to use dependency graphs extracted from the management data stored in management systems for event correlation. After normalizing the events received from different streams, the pre-generated dependency graph is used to determine the cause of these events. A breadth-first search strategy starting from each event received in a specified time interval is employed and the managed objects in the dependency graph that are reachable by a number of received events greater than some threshold value are determined. The breadth-first search instances can continue up to reaching a specified search depth. A basic breadth-first strategy does not eliminate the traversal of the large sub-graphs of the dependency graph to find the managed objects.

In [17], global candidate attacker plans are created by applying intrusion alert correlation based on a predefined attack base created using LAMBDA attack specification language. An attack is defined in this language by specifying its prerequisites, consequences, scenario and detection information. The detection information contains the definition of the alert types related with the attack. The intrusion alert correlation process includes the application of the correlation rules between pairs of attacks in the attack base and the instantiation of the variables in the rules with the data of actual intrusion alerts. The application of the correlation rules between pairs of attacks in the attack base corresponds to performing a *template matching* over all possible attacks and their relationships with a set of (sliding window of) received intrusion alerts.

Ning et al. [48] propose a method for building attack scenarios using hyper-alert types and hyper-alerts. A hyper-alert type represents an attack possibly having a corresponding signature on a particular intrusion detection sensor. Each hyper-alert type has its prerequisites and consequences. On the other hand, a hyper-alert is a collection of received intrusion alerts that are related with the same hyper-alert type. By using the *prepare for* relations among the hyper-alerts, hyper-alert correlation graphs are created. In [49], the authors introduce attack strategy graphs. An attack strategy graph is composed of a collection of related hyper-alert correlation graphs and denotes an attack scenario. Although our approach of representing attack scenarios with partial attack graphs is somehow similar to the approach of these two past works, there is no indication of processing secondary logs in these two works. The methods employed by them only process intrusion alerts.

In [14] a language, called CAML (Correlated Attack Modelling Language), for modelling multi-step cyber attacks in a way that enables attack scenario recognition by recognition engines (for ex. intrusion detection systems (IDS)) is proposed. CAML consists of modules linked to each other with predicates. Each module defines a template for a specific step of a multi-step attack scenario. The work of [14]) resembles to the work of ([17], since they both use attack scenario templates and match intrusion alerts to template elements.

Wu et al. [89] propose a distributed, collaborative intrusion detection framework that collects intrusion alerts from individual sensors and aggregates them in order to decrease the false positive and false negative alarm rates. The collaborative intrusion detection system (CIDS) proposed in this work is comprised mainly of the individual event detectors (sensors) placed on different locations on the target network, CIDS manager process and response strategy determiner. In the CIDS manager, there is an inference engine related to each different host on the target network. These are local inference engines. Also, there is a global inference engine that is responsible from collating the results of the local inference engines and inferring the occurrence of global (depending on more than one network host) intrusion attempts. There is a group of rule objects associated with each inference engine. Each rule object can be represented with a graph or a Bayesian network. Each rule object also represents a specific attack class (for ex. buffer overflow attack). An inference engine tries to match the translated events coming to its inference queue to its rule objects. The work of [89] uses an attack template approach similar to the works of [48]. However, an inference engine with its Bayesian network-based rule objects improves the cause-consequence-based hyper-alert correlation graphs in [48] by incorporating probabilistic reasoning.

Noel et al. [54] propose an intrusion event correlation method based on the distances among the exploits on an all-paths attack graph for a target network. The attack graph for the target network is built off-line by taking into account the network vulnerability information and exploit models derived using the information on the public vulnerability databases on the Internet. After the attack graph is built, the distances of the shortest paths between each pair of exploits on the attack graph are computed and stored. When an intrusion event is received, it is mapped to the corresponding exploit on the attack graph and checked for correlation with the previous intrusion events. Two intrusion events are correlated, if the distance between their corresponding exploits on the attack graph is not infinite.

In [78], a comprehensive framework defining relevant phases for intrusion alert correlation is proposed. The aim of the framework is to reduce the number and increase the abstraction level of the intrusion alerts. It tries to achieve this aim by constructing meta-alerts from the raw alerts by applying specific algorithms in alert correlation phases. By this way, an alert tree is produced whose leaves are single alerts and intermediate nodes and root node are meta-alerts. Some of the alert correlation phases defined by the framework are normalization, pre-processing, fusion, thread reconstruction, attack session reconstruction, focus recognition, multi-step attack analysis, verification, impact analysis and prioritization.

Wang et al. [85] propose a novel algorithm for alert correlation and attack prediction, which is called queue graphs. The main factor causing the development of this algorithm is a need to eliminate the disadvantages of the nested loop-based intrusion alert correlation methods developed so far. An obvious disadvantage of the nested loop-based alert correlation is the introduction of a sliding time window for the processing of the alerts and the comparison of a new alert with all the previous alerts for correlation. Even if in-memory indices are used for locating the previously received alerts, some of these indices must be deleted according to a sliding time window because of memory limitations.

In contrary to the nested loop-based approach, the queue graph approach takes only the latest alert corresponding to each exploit into account. In an all-paths attack graph consisting of exploit and security condition (privilege) nodes, each exploit node contains a queue of length one to store the latest intrusion alert related to the exploit node. When a new alert is received, its corresponding exploit is found in the attack graph, the alert is enqueued to the queue of the exploit and a breadth-first search is started backwards in the attack graph. When hypothesized alerts are not generated (in the basic correlation algorithm), the search stops, when an empty queue is reached. Since the backwards breadth-first search are performed over the whole all-paths attack graph, it may be very time-consuming because of the possible traversal of large parts of the attack graph that are unrelated to the received events, if the all-paths attack graph is large.

In [75], an approach that uses hyper-alert type graphs to generate hyper-alert correlation graphs, similar to those in [48], is described. The authors decide not to use vulnerability-based attack graphs containing also network topology information. The basic correlation algorithm uses in-memory indices among the attribute subsets of hyper-alerts, if these hyper-alerts satisfy the *prepares for* relation among themselves. The *prepares for* relation implies an inherent cause-consequence relationship and is the precondition for the existence of in-memory indices among the attribute subsets of the corresponding hyper-alerts. There is another graph to be used in the generation of the correlation graph. This graph is called attack-type graph whose nodes are hyper-alert types and edges represent the *prepares for* relation between hyper-alert types. There are also backward and forward indices stored in the hyper-alert types of the attack-type graph. These indices represent the relations among the different attribute combinations of the hyper-alert types. To generate the correlation graph, attack-type graph is traversed using these in-memory indices. Attack type graphs can be considered as network-independent attack templates.

The framework proposed in [87] consists of three main phases. These are evidence collection and aggregation, evidence graph manipulation and hierarchical reasoning using the generated evidence graph. Evidences are collected from different IDSs and network traffic captures and then converted into a normalized format. The evidence aggregation procedure tries to reduce the number of the evidences by eliminating duplicate evidences and providing support for many-to-one (DDoS) and one-to-many (network scanning) attack types. It uses basic comparison relations among the attributes of different evidences. The evidences generated as a result of the aggregation procedure are called hyper-evidences.

The evidence graph manipulation phase creates the evidence graph using the hyper-evidences. Each node in this graph represents a network host and each edge represents a hyper-evidence. There are two important attributions (labelling functions), one for nodes and one for edges. The most important node attribute is the functional state that can have values of *Attacker, Victim, Stepping Stone and Affiliated*. Each functional state value for each node has also a real strength value between 0 and 1. The most important edge attribute is the edge priority that is computed by considering the importance (weight) and relevancy (whether the attack is successful or not) of the corresponding hyper-evidence.

The hierarchical reasoning framework uses the generated evidence graph to generate attack scenarios in the form of attack groups. It contains two layers: local reasoning and global reasoning. Local reasoning computes and updates the values for each function state for each node in the evidence graph dynamically. It uses rule-based fuzzy cognitive maps. Global reasoning finds the attack groups by computing the structural similarities among the evidence graph parts using the function states for the nodes and priority attributes for the edges. The secondary evidences are not accounted in the generation of the attack scenarios.

In [64], the authors propose both an off-line and an online approach to intrusion alert correlation. The off-line approach tries to determine the features of a pair of attack types that play important roles in the formation of the *causality* relationships among the instances of these attack types. The off-line approach benefits from the Bayesian networks containing alerts as nodes. The probabilistic dependency values among the nodes of the Bayesian network are calculated by considering the specific features of alerts, such as the source and destination IP addresses and ports. These probabilistic dependency values are calculated from the raw alert stream, after performing some low-level alert correlation.

The online approach uses the correlation and relevance tables generated by the off-line approach to identify the correlated alerts and construct the attack scenarios on-the-fly from the raw alert stream. In addition to being history-based (uses the results of off-line approach), the online approach can also dynamically adapt the correlation probabilities between any two alert types according to the alert occurrence probability values measured from the current alert stream.

Roschke et al. present [68] an attack graph-based approach to intrusion alert correlation. Specific methods for mapping the received intrusion alerts to the attack graph nodes, alert aggregation, building alert dependency graphs and selecting suspicious subsets of alerts from the created alert dependency graphs are proposed. Each of these methods are parametrized to adjust the accuracy and speed of the overall correlation algorithm. The authors define several alternatives for the mapping of the received alerts to the attack graph nodes. Each of these alternatives requires different constraints to be satisfied among the specific fields of a received alert and the attack graph node to be matched. The alert dependency graphs are also built using the attack graph. The authors also define several alternative functions that determine the existence of dependencies among the matched alerts according to the locations of the corresponding nodes of the matched alerts in the attack graph. For instance, one of these alternative functions looks for the existence of a path of length less than some threshold value between the attack graph nodes of two matched alerts. If such a path exists, then an edge connecting the matched alerts is created on the alert dependency graph.

Yen et al. [92] propose a system that tries to detect the malicious activities and policy violations inside an enterprise network by using the *dirty* logs generated by various network devices. The volume of the generated logs in the used network in the performed experiments is beyond 1TB per day. The proposed system first normalizes the collected log data by normalizing their timestamps to UTC, detecting dynamically and statically assigned IP addresses, performing IP-to-hostname mapping and detecting dedicated hosts. Then, for each host a feature vector is created periodically. Destination, policy, host and traffic-based features are defined. After feature vectors are formed for the hosts, principal component analysis (PCA) is applied to the vectors. The resulting vectors created by PCA are clustered by using a modified K-means algorithm that does not need the number of clusters to be specified in advance. The distance between any two vector is computed by the L1 distance metric. The clustering results are manually examined to identify the possible malware infections and policy violations.

In [72], a network incident and malicious activity detection/response tool, called Reasonets, is proposed, which employs anomaly-based intrusion and case-based malicious activity detection methods. The anomaly-based intrusion detection method forms a model for regular network traffic by training over the normal network traffic data collected for the target network. Machine performance metrics such as the average usage of CPU and memory are also collected and used in training. For the case-based malicious activity detection method, a number of cases each representing a malicious activity (DoS, spyware, worm, etc.) are created by using the network traffic incidents obtained from the execution traces of various malware. Matching of the incidents collected from the target network to the generated cases is performed by using fuzzy logic methods and accounting for the incidents previously matched to the cases.

Marchal et al. [45] develop a network security monitoring system that utilizes four types of log data to detect malicious activities inside the target network. These types of log data are the DNS replies, HTTP packets, IP-flow data and honeypot data. For the instances of each type of data, a score is computed that indicates its maliciousness level. The determination of the values of the elements used to compute a score and the relations among these values are not based on formal, concrete methods. The selection method for the values of the weight vectors for the score elements is not described. The effects of these weight values on the maliciousness are also not explained. The authors measure the performance of five big data frameworks (Hadoop, Hive, Pig, Spark and Shark) with different attack scenarios representative of the computation of the maliciousness scores for different types of log data. It is done to determine the best big data framework for their score computations. However, there is no description about the implementation of the score computation algorithms over the mentioned big data frameworks.

In [95], a method for detecting stealthy malware activities by inferring triggering relations among the network events is proposed. The existence of a triggering relation between two network events is derived by creating pairwise features from the individual features of the events and using these pairwise features in specific machine learning algorithms. The creation of a pairwise feature uses the relation between the same individual features from the two network events. This may be considered as insufficient, since there can be a relation between two different individual features from the network events. As an example there may be a relation between the source IP feature of the first event and the target IP feature of the second event. The labelling of the created pairwise feature values with the existence of a triggering relation is performed manually or by using pre-defined rules. The machine learning algorithms used are naive Bayes, Bayesian classifier and Support Vector Machine classifier.

As a result, in our approach, attack scenarios are represented by partial attack graphs which are created and expanded dynamically. The nodes to be expanded in each iteration are selected according to their alarmed status, the ratio of their alarmed neighbours and the number of previous expansions applied to them. With this approach, during the attack scenario generation process, we try to eliminate the traversal of the large portions of the input all-paths attack graph that are irrelevant to the received logs (alarmed nodes). We also process secondary logs to determine additional alarmed privilege nodes in the partial attack graphs to make the attack scenario generation process more accurate.

3.3 Security Information and Event Management Systems

Security Information and Event Management (SIEM) systems collect and store the logs and security events generated by software running on the devices in a network, perform log trend/statistical analysis and apply specific event correlation rules to the collected data in order to detect attacks targeted to the network. They execute online during the normal operation of the target network and can recommend defense measures as responses to the intrusion attempts directed to the network. They can also perform network forensics analysis. SIEMs act as an information fusion system which interprets the collected log and event data in order to derive high-level security incidents and reports the derived information to the network security administrator together with recommended reactive defense measures.

A general-purpose information fusion system that requires a small amount of a priori network configuration information and can be utilized in cyber attack tracking is proposed in [73]. The main input to the system is a collection of sensor messages which can be any raw data generated by any hardware or software application on the target network. The (attack) tracking models are also provided as plug-ins to the system. The architecture of the models are defined to be composed of concepts and their containing features. Each feature contains a set of association constraints that are evaluated against sensor messages. Constraints can also be defined among the concepts. The different ways of the formation and update of the attack tracks by using the models and the input sensor messages are described in the paper. The measurement and reporting functions that can be performed on the detected tracks are provided as plug-ins to the system.

The applicability of the data mining methods for the analysis of security events is discussed in [27]. Given a list of malware incident log data, the authors apply A-priori data mining algorithm to identify patterns in the log data. This gives rise to the determination of the possible factors affecting (causing) malware infection. In addition, malware permanence and propagation analysis are performed by using the number of computers infected by the same malware and the date information in the input malware incident log data. Clustering algorithms are utilized for these analysis.

In [79], a parallel and distributed Complex Event Processing (CEP) rules engine that can be used as core event correlation engine in a SIEM system is proposed. In the proposed engine, a CEP rule can be divided into sub-rules and each sub-rule can be deployed on a number of SIEM physical nodes. The operations in a sub-rule can also be deployed on different nodes. The proposed engine makes the detection of slow and stealthy attacks possible, even when there is a huge volume of security event data generated on the target network. The performance of it is evaluated on a simulated environment that mimics the Olympic Games IT infrastructure.

Cheng et al. [13] present multi-core and parallel processing support for the security event correlation framework of SIEM systems. A plugin-based multi-core functionality is added to the SIEM system implemented by them. A parallel version of the k-means algorithm is implemented and integrated into the SIEM system via the multi-core functionality. The k-means implementation is used for event clustering (correlation).

The fundamentals of the SIEM systems and the challenges of forming them are discussed in [6]. A SIEM system is a fundamental component of a security operations centre. It consists of collecting, normalizing, storing and correlating large amounts of security event data generated by a large number of different event sources, such as anti-virus software, firewalls, proxies, application software and operating systems. The correlation of normalized events are generally performed by using pre-defined rules. These systems also contain network forensics analysis functionality.

One of the challenges arising in the formation of a SIEM system is related to the utilization of fine-grained, detailed rules to detect malicious behaviour. With the increase of the generated events, this utilization becomes much more resource-intensive necessitating the use of distributed, big data analysis methods. Huge amounts of generated events give also rise to problems in storing them in permanent storage. The lack of obtaining precise configuration information for the target enterprise network is another challenge. This may lead to increased false alerts produced by SIEM systems.

In [41], the authors propose a programming abstraction called semantic room that can be used to facilitate the implementation of SIEM systems based on collaborative information sharing among different institutions. Each semantic room has a different objective such as detection of port scanning activity or botnets. It has a number of gateways that are used to collect information from different institutions, normalize, aggregate and filter them and send them to the complex event processing component of the semantic room. The complex event processing component of the semantic room can be implemented with the Esper event correlation engine in a centralized manner or with Apache Storm in a distributed manner. In the Apache Storm implementation, a number of Esper engines can be used as Storm bolts.

The authors of [41] implement a port scan activity detection and a fraud monitoring semantic room. In the port scan activity detection semantic room, inter-domain stealthy TCP SYN port scans are detected. In the fraud monitoring semantic room, unauthorized point of sales, tampered ATM and counterfeit banknotes information supplied by different banks are used to detect the fraud locations in Italian territory. The authors also propose the use of specific privacy-preserving mechanisms to avoid the linkage between the raw data sent by the institutions to the semantic rooms and the results extracted and distributed by the semantic rooms. These mechanisms are used to ensure the contracts provided by the semantic rooms that are used to avoid the leakage of sensitive data of the institutions.

3.4 Optimal Network Hardening Measure Recommendation

There is considerable work in the literature on optimal network hardening against attack paths. In one of the earliest works on the subject [60], Phillips and Swiller propose a greedy network hardening algorithm based on *d-optimal* shortest attack paths, where *d* is the error parameter. A node in the employed attack graph represents a security state of the network in terms of attacker privileges. A defense can be related with the elimination of any node in the attack graph. However, the elimination of any node in their attack graph structure may not always be enforceable by the network administrator.

Sheyner et al. [69] concentrate on eliminating all attack paths to the goal states. The authors prove that the computation of the minimum atomic attack sets that prevent the intruder from reaching her goals is NP-complete by giving a reduction from the minimum cover problem. They propose finding critical atomic attack sets with a greedy algorithm that is polynomial on the size of the attack graph. However, a critical atomic attack set may be related with any security state node in the attack graph whose negation may not always be enforceable by the network administrator.

In [53], hardening measures are defined in terms of the negation of only *initial security states* which are always enforceable by the network administrator. A forward and a backward analysis are employed to convert the input attack graph into an exploit dependency graph by removing the cycles in the attack graph. The goal states are expressed as a conjunctive normal form of terms containing the initial conditions. The time complexity of the generation of the conjunctive normal form is exponential in the number of the initial conditions in the attack graph, since all combinations of the initial conditions' negation status can appear as a term in this form. Also, the initial conditions are considered independent of each other. The existence of a hardening measure that can negate more than one initial condition is not considered in [53].

An adaptive intrusion response system using a graph of intrusion goals (I-Graph) is proposed in [23]. An I-Graph is formed by using the vulnerability descriptions and service dependency graph for the target network. The system can be used with any number of detectors that generate security alerts (evidences). The alerts are mapped to the nodes of the I-Graph. For each node to which an alert is mapped, a confidence value is computed by using the confidence values of its children. The nodes for which responses are deployed are determined according to their confidence values. The responses deployed are determined by computing an effectiveness and disruptiveness value for them. These values are updated by accounting for the previous success conditions for the responses. This gives rise to the adaptive selection of responses by improving the responses deployed against similar attacks over time.

The proposed work in [86] improves the work of [53] by applying *one-pass* backwards modified breadth-first search to the input attack graph in order to both remove logic loops and obtain the logic expression of the negated goals as a disjunctive normal form of the negated initial conditions.

Dewri et al. [21] define the optimal network hardening problem as a multi-objective optimization problem. The objectives are minimizing the total cost and residual damage. Their residual damage model computes a damage value for each node in the attack graph. Minimizing the residual damage is not as effective as maximizing the number of eliminated attack paths, which is one of the optimization objectives in our work. This is because the former gives priority to the decreasing of the residual damage on all the conditions and not to the elimination of attackers' possibility of reaching the goal conditions. Our standpoint is that the latter fact should come first and the residual damage minimization should be performed afterwards.

In [21], the authors also apply a genetic algorithm to solve the optimization problem, but they do not take into account the interactions among the hardening measures for the generation of new candidate hardening solutions (new generations in the genetic algorithm). Instead, the candidate solutions seem to be generated simply by random crossover and mutation processes in their genetic algorithm, independent of such interactions. These interactions may indicate a number of common attack paths that can be eliminated by various hardening measures, which can be utilized to prevent the optimization process from being caught in local optima easily and help speed up the optimization process.

In [33], breadth-first search is applied starting from the initial conditions, after assigning effective costs to them. The effective cost of an initial condition is defined as its negation cost divided by its parity. The parity is taken as the number of dependent exploits of the initial condition. Only the initial conditions with the most effective costs are considered for each attack path to the goal conditions, which can hinder obtaining optimal solutions in most cases. In [12], an attack graph is first converted into a Reduced Ordered Binary Decision Diagram (ROBDD). The computation of the optimal solution is performed by recursively applying Shannon decomposition rule to the nodes of the ROBDD starting from the source node. The time complexity is $O(n)$, where n is the number of nodes in the ROBDD. The number of nodes in an ROBDD is dependent on the variable ordering of the ROBDD and the variables are the initial conditions in the attack graph. Therefore, the time complexity is exponential in the number of initial conditions.

Poolsappasit et al. [61] utilize both single and multi-objective genetic optimization algorithms to find the optimal hardening solution for a network. However, similar to [21], the candidate solutions are generated randomly without accounting for the commonality of the effects induced by the hardening measures on the target network. In [1], the assumption of independently negating each initial condition in an attack graph is removed by considering the inter-dependencies between the hardening measures. A forward search starting from the initial conditions is performed to determine the suboptimal sets of hardening measures on each of the traversed conditions and exploits. A parameter to limit the number of these sets stored on each condition and exploit is introduced in order to refrain from their exponential growth. The effectiveness of the obtained hardening measure sets is highly dependent on this parameter. Ma et al. [42] present a mathematical model employing attack graphs for representing the optimal network hardening problem as a non-restraint optimization problem with penalty. A parallel, genetic algorithm is used to solve the resulting optimization problem.

In [83], a solution for hardening a network against multi-step intrusions by using attack graphs is proposed. It represents the specified critical resources as a logic proposition of the initial security states. After simplifying the proposition, the minimum-cost hardening measure set in terms of negating initial security states is determined. In [81], the network hardening problem using attack graphs is formally defined and the applicability of graph-theoretic, heuristic approaches to the problem is discussed.

In [82], the authors define formally the concept of a hardening action, the interdependencies among the hardening actions and the concept of a hardening strategy that can combine multiple, possibly interdependent hardening actions into a composite representing a defense plan. They also propose a near-optimal approximation algorithm for the network hardening problem that employs attack graphs and scales linearly with the size of the graphs.

Yigit et al. [93] propose a heuristic method to find a cost-effective network hardening solution with a limited budget. The method uses as input an attack graph that contains only the possible attack paths which can reach pre-specified critical resources in the target network. The exploit or initial security state contributing most to the elimination of these attack paths with least cost is selected at each step of the method, until the total cost exceeds the allocated budget.

4
System Overview

We define an attack scenario as a related group of atomic attacks performed by a coordinated group of attackers to achieve specific target compromises on a network. The main aim of the system proposed in this thesis work is to find ongoing attack scenarios for the target network in near real-time using the logs collected throughout the network. Vulnerability exploits are used to represent atomic attacks.

One of the inputs to the proposed system is the stream of logs generated across the target network. They are the log records generated by various software running on the network hosts. The other inputs are the pre-computed full attack graph for the target network and the pre-generated behavioural malware signatures. The system tries to output the attack scenarios that have been/are being followed by the attackers on the target network by using its inputs. An overview of the attack scenario construction (detection) system proposed in this thesis work is shown in Figure 4.1.

The logs are collected by the log collection module that is composed of distributed software components each of which is responsible from fetching the logs recorded by the software running on a single network host. The collected logs are first preprocessed to normalize them, eliminate duplicates among them and classify them as primary or secondary. The normalized primary and secondary logs are processed by using different methods. The aim of the log processing phase is to determine and output the possible attacker privileges and vulnerability exploits that are indicated (implied) by the normalized logs generated so far. In log processing results' correlation phase, the attacker privileges and vulnerability exploits output by the log processing phase are matched to the nodes on the full attack graph.

The matched nodes are used to create attack scenarios in the form of partial

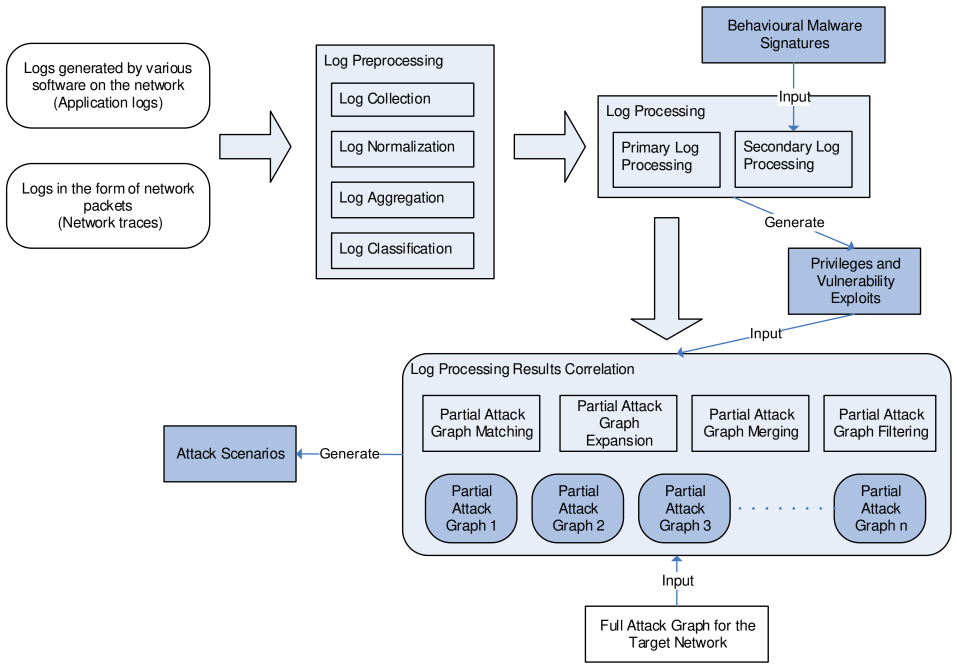

Logs generated by various software on the network (Application logs)
Logs in the form of network packets (Network traces)
Log Preprocessing
Log Collection
Log Normalization
Log Aggregation
Log Classification
Behavioural Malware Signatures
Input
Log Processing
Primary Log Processing
Secondary Log Processing
Generate
Privileges and Vulnerability Exploits
Input
Log Processing Results Correlation
Partial Attack Graph Matching
Partial Attack Graph Expansion
Partial Attack Graph Merging
Partial Attack Graph Filtering
Partial Attack Graph 1
Partial Attack Graph 2
Partial Attack Graph 3
Partial Attack Graph n
Attack Scenarios
Generate
Input
Full Attack Graph for the Target Network

Fig. 4.1: System overview

attack graphs. We search each matched node on the already created partial attack graphs. If a node n having the same content as the content of the matched node exists on a partial attack graph, we mark node n on the partial attack graph as alarmed. If we do not find a node having the same content as the content of the matched node on any partial attack graph, we create a new partial attack graph, add a copy of the matched node to the newly created partial attack graph and mark the copy node as alarmed. It should be noted that a partial attack graph must contain at least one alarmed node.

The partial attack graphs (representing attack scenarios) are expanded by using the full attack graph. The nodes to be expanded can be alarmed or not and are selected according to their distances to the alarmed nodes, among other factors. After the expansion of the partial attack graphs is finished, the partial attack graphs are searched for a possibility to merge them together.

Actually, there is a pipelined, continuous execution among all the processes (log preprocessing, log processing and log processing results correlation) performed by the system as shown in Figure 4.1. All the processes are executed over a real-time, stream-based Big Data framework (Apache Storm). A distributed execution (application) topology comprised of Storm components (spouts and bolts) is designed and implemented. The spouts and bolts are distributed processing agents defined in the Apache Storm framework. The spouts are responsible from collecting data from data streams. The bolts process the data transferred to them by spouts. The data is processed by executing user-defined algorithms. The users can define any application topology via using Apache Storm API (Application Programming Interface) that is composed of connected spouts and bolts in order to execute distributed algorithms over Storm.

The spouts defined by the proposed system are responsible for collecting, filtering, normalizing and classifying the logs. They transfer the resulting normalized logs to the bolts that are responsible from log processing. Log processing bolts finds the attacker privileges and vulnerability exploits indicated by the normalized logs and transfers the found items to the bolts that are responsible from correlating log processing results to detect attack scenarios. The detected attack scenarios are in the form of partial attack graphs and also stored in a database. Before storing the attack scenarios in a database, they are post-processed to filter the irrelevant nodes in them. The irrelevant nodes are the nodes which are not existing on a neighbourhood of an alarmed node that is of length less than a specific threshold value. (The irrelevant nodes are far away from the alarmed ones and may be generated as a result of imperfectness of the algorithms employed in the proposed system.) When the system user wants to obtain the attack scenarios detected so far, she can send a signal to the backend system shown in Figure 4.1 via a user interface.

The proposed system has four main components that are listed below:

- *Attack graph generation* component responsible for generating the full attack graph for the target network in a distributed manner
- *Behavioural malware signature generation* component responsible for generating malware signatures that are matched against the logs collected for the target network
- *Attack scenario detection* component responsible for processing the collected logs to create/update partial attack graphs representing ongoing attack scenarios
- *Defense recommendation* component responsible for computing near-optimal defense measures to react to the detected attacks and stop further propagation of the attacks indicated in the detected attack scenarios

Generating attack graphs forms the basis for detecting ongoing attack scenarios in the scope of this thesis work. The next section describes the taxonomy for the attack graph generation process proposed in this thesis work. Afterwards, each component of the proposed system listed above is described in detail in its own section by referring to the main phases of the execution of the system illustrated in Figure 4.1.

5 A Taxonomy for Attack Graph Generation Process

In this section, we describe the *attack graph generation process taxonomy proposed in this thesis work* by also referring to the prominent works on the attack graph literature. The different phases of the attack graph generation process are explained and a set of classification criteria is proposed for the applicable models/methods for each phase. During the explanation of each phase, the models/methods implemented in the related past works in the scope of this phase are also discussed. Additionally, different uses of attack graphs are described by discussing the approaches proposed by the related past works. The contents of this section are mostly taken from [36] which was also written by the author of this thesis work.

The activities performed during the whole attack graph generation process can be classified into three high-level phases, as illustrated in Figure 5.1. The first phase, reachability analysis, mainly considers the computation of the reachability conditions among the target network hosts. The modelling phase considers how to model the individual attack templates and the attack graph structure. In the core building phase, the possible attack paths are determined and some paths are possibly pruned to construct the attack graph. The uses of attack graphs cover the operations performed on the constructed attack graphs for network security analysis and are detailed in the corresponding subsection. The remainder of this section provides a systematic description of the proposed classification scheme for the applicable models/methods in each phase with brief examples of each classification criterion from the related literature.

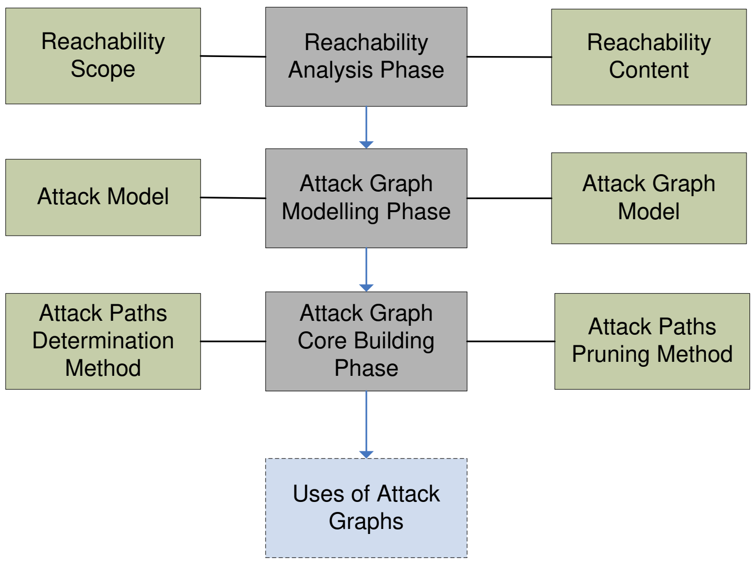

Reachability Scope
Reachability Analysis Phase
Reachability Content
Attack Model
Attack Graph Modelling Phase
Attack Graph Model
Attack Paths Determination Method
Attack Graph Core Building Phase
Attack Paths Pruning Method
Uses of Attack Graphs

Fig. 5.1: Attack graph generation phases and related classification criteria

5.1 Reachability Analysis Phase

Reachability analysis phase mainly investigates the reachability conditions within the target network, which, in a simplistic viewpoint, determines whether two given hosts can access each other. It can also indicate more detailed information, such as which applications of the two hosts can access each other, which protocols can be used for the communication between the two hosts, etc. The reachability conditions among the network hosts are mainly represented with a single reachability matrix whose rows and columns indicate the network hosts. An entry of a reachability matrix can simply be a boolean indicating the existence of a reachability (accessibility) between the corresponding two hosts, or any complex data structure.

The two main classification criteria for the reachability information are the *reachability scope* and *reachability content*. The reachability scope determines the scope of the network hosts among which the reachability conditions are computed, before the attack graph core building process. The reachability content determines the network security objects (entities) that are accounted for in the computation of the reachability information. A detailed classification scheme for the reachability information is given in Figure 5.2.

Reachability Scope The possible values for the reachability scope classifi-

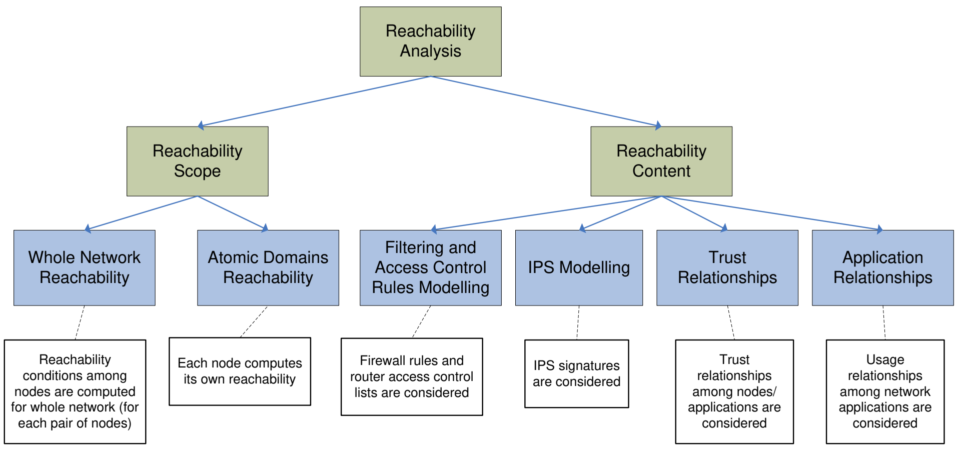

Reachability Analysis
Reachability Scope
Reachability Content
Whole Network Reachability
Atomic Domains Reachability
Filtering and Access Control Rules Modelling
IPS Modelling
Trust Relationships
Application Relationships
Reachability conditions among nodes are computed for whole network (for each pair of nodes)
Each node computes its own reachability
Firewall rules and router access control lists are considered
IPS signatures are considered
Trust relationships among nodes/ applications are considered
Usage relationships among network applications are considered

Fig. 5.2: Reachability information classification

cation criterion according to Figure 5.2 are:

1. *Whole Network Reachability*: Single step reachability condition for each pair of hosts on the network is computed. A single step reachability condition denotes direct reachability at any network layer between the two hosts (without any intermediate hosts).

2. *Atomic Domains Reachability*: Each host (or in the general case, a group of hosts) computes single step reachability conditions for the hosts in its neighbourhood in the target network topology.

Actually, most of the past works related to attack graph generation compute whole network reachability as an input to the attack graph core building process. One exception is [11], where the authors propose using atomic domains, each of which contains information about one network host and its directly connected hosts, to generate attack graphs. This ensures that there is no need to generate the whole attack graph from scratch, when one part of the target network topology changes. Only the information in the related atomic domains should be updated. The protection domain abstraction in the TVA tool in [52] and [34] can also be considered as an *Atomic Domains Reachability* abstraction, since it encodes single step reachability conditions for the network hosts in a protected domain, which does not contain any connectivity limitations among its contained network hosts. A host in a protection domain may have recorded the reachability conditions with the hosts in other protection domains.

Reachability Content The possible values for the reachability content classification criterion according to Figure 5.2 are:

1. *Filtering and Access Control Rules Modelling*: Firewalls' filtering rules and routers' access control rules are accounted for in the computation of the reachability information.
2. *Intrusion Prevention System (IPS) Modelling*: Signatures defined in the intrusion prevention sensors are taken for the computation of the reachability information into account.
3. *Trust Relationships*: Trust relationships among the network hosts are accounted for in the computation of the reachability information.
4. *Application Relationships*: Usage relationships among the networked applications (service usages, etc.) are accounted for in the computation of the reachability information.

The four items enumerated above represent the most commonly used network security entities that affect the reachability information computation. The use of relational predicates to represent trust relationships, running services and existing vulnerabilities on the network hosts is proposed in [69]. The authors model the connectivity among the network hosts by processing the access control rules and the vulnerability signatures on the network intrusion prevention sensors. All of the possible values for the reachability content classification criterion are eligible for this work.

In [32], a reachability matrix is computed by accounting for the service usages, network and application layer filtering rules on the security boundary devices. *Filtering and access control rules modelling* and *application relationships* values in the reachability content classification criterion are exemplified by this work. The NetSPA tool that is developed in [32] is improved in [31] by the same group from MIT Lincoln Laboratory. Most notably, the improvements account for the rules in personal and proxy firewalls and the signatures in intrusion prevention systems as additional reachability content in network reachability computation.

5.2 Attack Graph Modelling Phase

The classification associated with the attack graph modelling phase takes into account the attack model and attack graph model. The attack model can be considered as a model for forming an *attack template* describing the elements of a number of attacks, the conditions (required/gained attacker capabilities) for the attack elements and the relations among the elements and the conditions. An attack template defines the utilization logic for a number of attacks. An attack graph model defines a structure used to represent *attack instances* (successfully applied attacks) and the connections among them.

5.2.1 Attack Model

An attack model defines the elements and the utilization logic, in terms of required/gained attacker capabilities, of one or more attacks via attack templates. Attack templates can include high-level, abstract adversary and threat models or low-level vulnerability exploit models. Threat models can be formed by defining the relations among the vulnerability exploit models. As an example, a specific type of worm that can exploit two different vulnerabilities for infiltration and privilege escalation in its life-cycle can be modelled with a threat model combining the exploitation models of these two vulnerabilities. Adversary models can be formed by incorporating various attacker behavioural profiles and combining a number of threat models according to these profiles. As an example, a specific hacker can be modelled with an adversary model accounting for the exploits that are at her disposal. The adversary model can be refined to determine the transitions among the appropriate threat models by considering the capability of the hacker to hide her existence from the defense applications.

The possible values identified for the attack modelling classification criterion are as follows, as also depicted in Figure 5.3.

1. *Manually-defined Attack Templates*: In this case, attack templates manually formed by security experts are used.
2. *Intrusion Alert-based Attack Templates*: In this case, attack templates are formed by using intrusion alerts' meta information and forming relations among the alerts by utilizing specific alert correlation algorithms executing on these meta information.

3. *Text Processing-based Attack Templates*: In this case, attack templates are formed by applying text processing methods to the information contained in some specific vulnerability, weakness or attack databases.

The attack templates formed in each of the above cases define a collection of related attacks with their pre- and postconditions and relations among them. The pre-and postconditions are generally templates for representing network states and attacker privileges that are independent of any specific network.

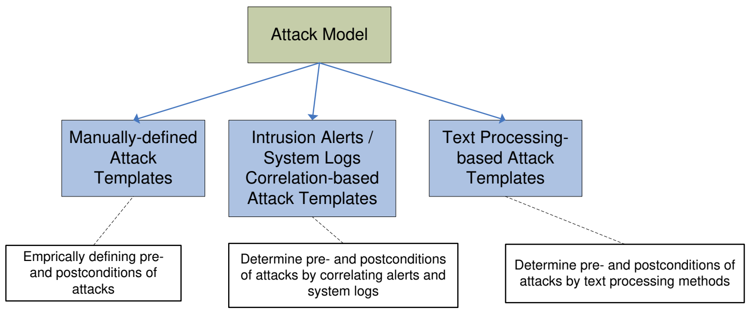

Attack Model
Manually-defined Attack Templates
Intrusion Alerts / System Logs Correlation-based Attack Templates
Text Processing-based Attack Templates
Emprically defining pre- and postconditions of attacks
Determine pre- and postconditions of attacks by correlating alerts and system logs
Determine pre- and postconditions of attacks by text processing methods

Fig. 5.3: Attack model classification

In [60], attack templates, each of which defines a collection of atomic attacks as edges among attack stages in a directed graph, are described. There is no indication of semi-automatic or automatic generation of attack templates. In [67], the described attack model defines the vulnerabilities of the network hosts, exploits applied to the hosts and access level of the attacker on the hosts as variables. It also defines the relations among the variables. All these definitions are encapsulated in the *manually-defined attack templates*. The vulnerabilities and access levels serve as pre- and postconditions for the exploits.

In [76], the *concepts* take the form of atomic attacks. Their pre- and postconditions are defined by the *capabilities*. In this work, there is no indication of semi-automatic or automatic generation of pre- and postconditions of the concepts and capabilities. In [69], the authors model atomic attacks as rules that describe how an attacker (intruder) can change the state of the network or add to her knowledge new facts about the network state. These rules form pre- and postcondition relationships with the exploits. In this work, there is also no indication of semi-automatic or automatic generation of pre- and postconditions of vulnerability exploits defined as atomic attacks, so the attack model can be considered as a collection of *manually-defined attack templates* taking the form of rules.

Ning et al. [49] propose a model and an algorithm for the extraction of hyper alert types from intrusion alerts. A hyper alert type represents an attack template containing information about the prerequisites and consequences of the constituent attacks based on the attribute values of the related intrusion alerts. The model described in this work exemplifies the utilization of *intrusion alert-based attack templates* for modelling attacks.

Kotenko et al. [38] considers vulnerability exploits as low-level attacker actions (atomic attacks). The vulnerability descriptions in Open Source Vulnerability Database (OSVDB) are processed by specific text processing methods to extract pre- and postconditions for the vulnerabilities. An attack model representing the utilization logic for the atomic attacks are formed by using *text processing-based attack templates*, where each attack template defines pre- and postconditions for one atomic attack.

In [10], the authors define an abstraction, called attack pattern, over vulnerabilities. The attack patterns are generated by applying *text processing methods* over the attack pattern enumeration and classification information in CAPEC database ([9]), which are then used to form attack templates each of which describes the utilization logic for a set of vulnerabilities with similar weakness types. The attack patterns can represent threat models in this context. In [29], attack scripts that are generated by security experts are combined in the form of directed graphs. The vulnerability descriptions existing in NVD are processed by a specific text mining method using the keywords in the attack scripts to match the vulnerabilities to the nodes of the directed graphs representing combined attack scripts. Attack scripts can be considered as threat models.

5.2.2 Attack Graph Model

An attack graph model defines how to represent the attack instances on the target network hosts and the connections among them. Attack graph models can be classified in terms of their structures as given in Figure 5.4 and listed below.

1. *Single level*: There is only one type of graph comprising the resulting attack graph, possibly containing more than one layer.
2. *Multi-level*: There is a hierarchical structure of different types of graphs, all of which form the resulting attack graph output together.

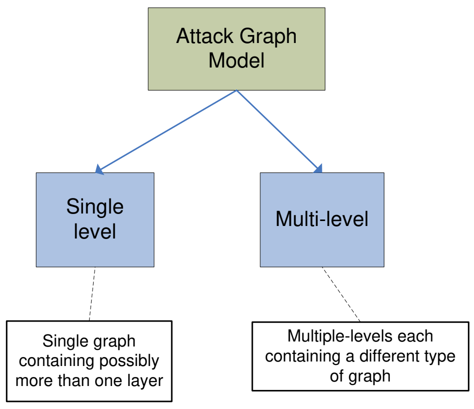

Attack Graph Model
Single level
Multi-level
Single graph containing possibly more than one layer
Multiple-levels each containing a different type of graph

Fig. 5.4: Attack graph model classification

Ammann et al. [3] introduce a *single level* attack graph model where the nodes represent the hosts on the target network and the edges represent the highest access level that can be obtained by an attacker attacking from the source hosts to the target hosts. Ou et al. [57] employ a *single level* attack graph model containing derivation and fact nodes which define the applied exploits and facts about the network state in order. In [32], multiple-prerequisite attack graph structure, which is also based on a *single level* attack graph model, is introduced. This structure models the attacker privileges and reachability conditions as nodes in the attack graph. In [96], a *single level* attack graph model based on virtual performance nodes is introduced. A virtual performance node indicates the negative effects of the attacker activities on the network performance. Privilege nodes are also used in this model.

Dacier et al. [20] propose a *multi-level* attack graph model consisting of a privilege and an intrusion process state graph. A privilege graph represents the privileges (capabilities) gained by the attacker with its nodes, and the atomic attacks to gain these privileges with its edges. An intrusion process state graph summarizes a privilege graph to determine the attack scenarios as a collection of individual atomic attack paths in the privilege graph. An attack scenario can be considered to represent a summary of a single path composed of atomic attacks in the privilege graph. Another *multi-level* attack graph model is proposed in [4], where a number of layers may be created dynamically during the attack graph core building mechanism. Each layer stores different nodes that can be the vulnerabilities on the specific network hosts, attacker privileges, etc.

Ning et al. [49] propose a *multi-level* attack graph model consisting of a hyper alert correlation and an attack strategy graph. A hyper alert correlation graph is formed by using the correlated intrusion alerts for the target network. An attack strategy graph is formed by generalizing (parts of) the hyper alert correlation graphs. In [38], the authors propose an attack graph model that contains a hierarchical structure composed of three levels representing low-level (atomic) attack instances, attack purposes and stages constituting attack scenarios and combination of the attack scenarios. In [90], a two-tier attack graph model is proposed, where the higher level is formed by host access graphs that are built using sub-attack graphs at the lower level.

5.3 Attack Graph Core Building Phase

Attack graph core building phase refers to the core algorithm used to construct the attack graphs. In this phase, some of the attack paths may also be pruned during the formation of the resulting attack graph. Figure 5.5 shows the classification criteria proposed in this thesis work for the methods that are applied in the attack graph core building phase.

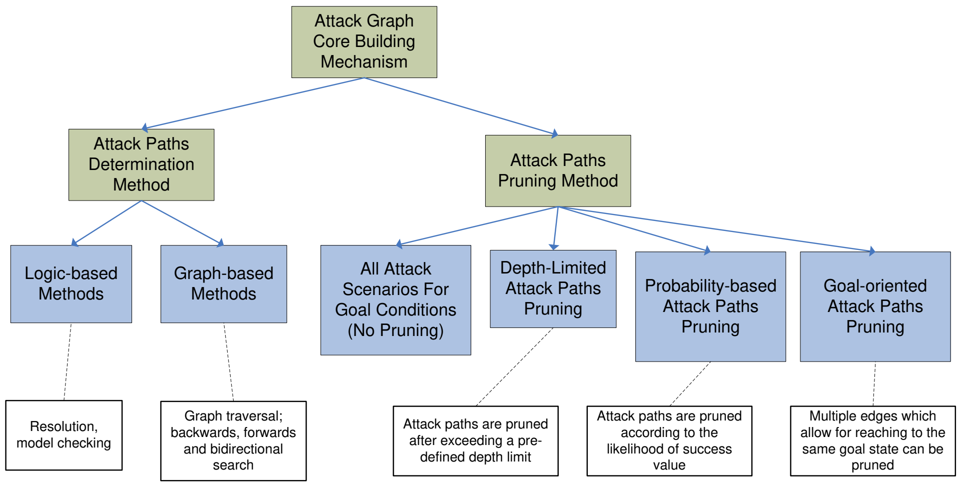

Attack Graph Core Building Mechanism
Attack Paths Determination Method
Attack Paths Pruning Method
Logic-based Methods
Graph-based Methods
All Attack Scenarios For Goal Conditions (No Pruning)
Depth-Limited Attack Paths Pruning
Probability-based Attack Paths Pruning
Goal-oriented Attack Paths Pruning
Resolution, model checking
Graph traversal; backwards, forwards and bidirectional search
Attack paths are pruned after exceeding a pre-defined depth limit
Attack paths are pruned according to the likelihood of success value
Multiple edges which allow for reaching to the same goal state can be pruned

Fig. 5.5: Attack graph core building mechanism classification

An attack graph core building mechanism can be considered from two different perspectives according to the proposed classification scheme. One of them is the attack paths determination method and the other is the attack paths pruning method, which are described next.

5.3.1 Attack Paths Determination Method

The attack paths determination method indicates the main algorithmic approach to the attack graph core building process. The possible methods for the attack paths determination are:

1. *Logic-based Methods*: The attack paths are created by using logic deduction methods (resolution, model checking, etc.). The network states are represented with facts and the exploits are represented with relational predicates over these facts.

2. *Graph-based Methods*: The attack graph building problem is seen as a graph traversal problem and the attack paths are created during backwards, forwards or bidirectional graph search. The graph-based attack graph building algorithms employ some form of searching procedure to generate the nodes and edges of the attack graph on the fly. This searching process may sometimes approach to the logic-based deduction methods, especially when attack templates (with variables indicating network security states) containing facts and predicates are used and instantiated during the search.

In [67], attack graphs are generated by applying model checking on a state machine that represents possible network states as facts and atomic attacks as logic predicates. The user can specify desired security conditions as temporal logic formulas and an attack scenario is generated by applying model checking on the state machine to find the counter examples to the specified logic formulas. The model checking tool developed in [67] is improved in [69] to find all the counter examples to a given security condition. In [57], logic deduction is applied to reach from the initial facts to the goal facts representing the attacker privileges. All of these works use *logic-based methods* to generate the attack paths on the resulting attack graph.

In [60], a backwards search from the goal states (attacker's goal privileges) is performed to generate an attack graph. The search process employs a unification mechanism to instantiate the attack templates using the provided attacker profile and the target network configuration information. This work exemplifies the utilization of *graph-based methods* (graph search) to generate the attack paths. In [4], a breadth-first search with a specific marking procedure to generate the resulting attributes (attacker privileges) in a multi-layer graph is applied. The marking of the attributes is used to determine the termination condition of the algorithm and is used by the additional algorithms (e.g., finding minimal attack paths). Also in [3] a *graph-based method* is used to generate an attack graph showing only the highest access levels that can be obtained when attacking from a host to other hosts with a direct exploit. After that, a transitive closure on this graph is computed to reflect the effects of the indirect application of exploits. In [10], a search-based algorithm is used to build an attack graph via unifying attributes (attacker privileges) to the pre-defined attack patterns. This unification mechanism resembles to that in [60], but in [10] the authors also define some basic simplifications that can be applied during the unification process to stop the further update of instantiated unsatisfiable attack patterns earlier during the search. Ma et al. [43] apply a bidirectional search in parallel (forwards from the initial privileges and backwards from the goal privileges) to generate an attack graph.

5.3.2 Attack Paths Pruning Method

The attack paths pruning methods aim to avoid the combinatorial state explosion problem that can occur in the attack graph building process. Possible approaches for the attack paths pruning are:

1. *All Attack Scenarios For Goal Conditions (No Pruning)*: No pruning method is applied.

2. *Depth-Limited Attack Paths Pruning*: Attack paths are pruned, when their depth values exceed some predetermined threshold. By this way, it is assumed that the attack paths containing a number of exploits more than the predetermined threshold value are less likely to be followed by an attacker.

3. *Probability-based Attack Paths Pruning*: Each edge (generally exploit) and each node (generally network state, attacker privilege) is assigned a probability of successful occurrence. By using these values, a cumulative probability of success value is computed for each attack path during attack graph building. This cumulative probability value indicates the likelihood of the attack path to be followed by an attacker. The attack paths whose cumulative probability of success value decrease below some predetermined threshold value are not extended further.

4. *Goal-oriented Attack Paths Pruning*: There may be more than one attack path between any two network states, one of which is a goal state (attacker's goal). Some of these paths can be eliminated by removing

corresponding redundant edges. This process can be performed by taking into account the sub-goals (critical network states) instead of the end goals of the attacker.

Different graph structures and algorithms have been proposed in the literature to avoid the exponential time and space complexity (depending on the network size) that can be encountered in the attack graph building process. In [3], the authors propose a method for attack graph computation based on highest access levels obtained on the network hosts. However, the resulting attack graph is not a full attack graph, namely it does not express all the possible paths that an attacker can use to intrude into the target network. It only shows the example worst-case attack scenario among each host pair in the network, which can be utilized by an attacker. The proposed attack paths pruning method in this work can be considered as a *goal-oriented attack paths pruning* method, since it eliminates some paths *reaching the goal states* according to the importance of the access levels gained as a result of applying the exploits on the paths.

Dapeng et al. [44] utilize both *depth-limited* and *probability-based attack paths pruning* methods to reduce the effects of the combinatorial state explosion problem that can be encountered when generating full attack graphs.

Bhattacharya et al. [7] propose a *goal-oriented attack paths pruning* method, which is used to identify the attack paths leading to the determined goal states by removing redundant (useless) edges from an input exploit dependent attack graph.

In [91], a *probability-based attack paths pruning* method containing the computation of three different types of probability values is introduced. The three different probability value types are used to denote the likelihood of successful exploitation for the vulnerabilities, the effects of inaccurate network configuration information collected before the attack graph building process and the likelihood of utilization of the attack paths by an attacker.

In [43], the authors apply a *simple depth limitation policy* to the bidirectional graph search method they use for the attack graph building. The attack paths whose lengths are greater than a given threshold are no longer expanded.

5.4 Using Attack Graphs for Network Security

Once an attack graph is generated, it can be used for a variety of purposes with positive or negative effects. This section is mainly concerned with the use of attack graphs for increasing the security level of the network, e.g. by determining the network applications presenting higher security risk values and optimal security measures based on the generated attack graphs. The major application areas of the attack graphs for network security can be listed as follows, which are also depicted in Figure 5.6.

1. *Network Security Metrics Computation*: Attack graphs can be used to derive network security metrics used for global security assessment of the target network. These metrics can be used to perform security risk analysis for the target network. Each node (generally indicating a network state) and each edge (generally indicating a vulnerability exploit) on the attack graph can be assigned a probability of occurrence. A node can also be assigned a possible damage value, if the corresponding network state for the node indicates the compromise of some information source for a network host. From these probability and damage values, the cumulative risk values are computed for each network state (node) on the attack graph.
2. *Counter-measure Recommendation*: Attack graphs can be used for recommending near optimal security defense counter-measures.
3. *Near Real-time Security Analysis*: Attack graphs can be used for online security situational awareness and detecting ongoing attack scenarios by performing high-level correlation and aggregation of the intrusion alerts, network traces and system logs. The detected attack scenarios can be used to perform future attack predictions and determine reactive defense measures.
4. *Network Design Generation*: An attack graph can be used to generate optimal firewall rules and IDS/IPS locations automatically for the target network to effectively counteract the attack paths on the attack graph.

In fact, all of the application areas for attack graphs described above can be employed in practice, if we assume that the attack graphs are updated

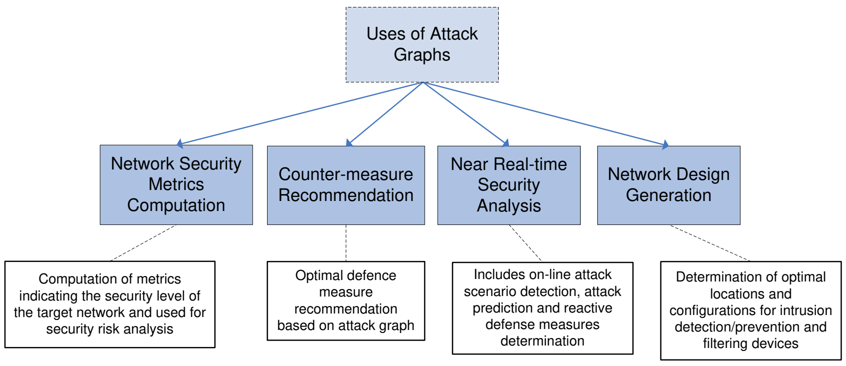

Uses of Attack Graphs
Network Security Metrics Computation
Counter-measure Recommendation
Near Real-time Security Analysis
Network Design Generation
Computation of metrics indicating the security level of the target network and used for security risk analysis
Optimal defence measure recommendation based on attack graph
Includes on-line attack scenario detection, attack prediction and reactive defense measures determination
Determination of optimal locations and configurations for intrusion detection/prevention and filtering devices

Fig. 5.6: Application areas of attack graphs for network security

continuously with the updates of the target network configuration. Network security metrics computation can provide an indication of the critical attack paths and the associated weakest links. For this purpose, the goal attacker privileges supplied as input to the attack graph core building algorithm shall reflect the privileges that can be obtained on critical network resources. Optimal counter-measure recommendation can also find practical usage for determining proactive defense recommendations. It can also use the attack graphs generated by accounting for the goal privileges pointing to critical network resources. Network design generation can find practical use in locating the intrusion detection/prevention systems and firewalls optimally in the target network. It can also be used to determine firewall and access control rules, if necessary support for resolving conflicting rules and processing different custom rule formats is provided.

Near real-time security analysis can be utilized in practice in two ways. In the first way, a full attack graph is generated and stored off-line and used for intrusion alert/log correlation online. In the second way, there is no off-line, statically generated full attack graph. Instead, partial and depth-limited attack graphs are generated online using the intrusion alerts/logs as inputs. The partial attack graphs are expanded with the reception of additional related intrusion alerts/logs. Independent of whether a full attack graph is statically generated or not, the correlated intrusion alerts on the full attack graph or partial attack graphs are used to predict future attacks and determine reactive defense measures. Actually, attack graphs can be utilized in collaborative intrusion detection systems in similar ways.

Dacier et al. [20] propose a security evaluation and a *network security metrics computation* framework based on attack graphs, which can be used to perform risk analysis and *counter-measure recommendation* to evaluate and increase the security level of the target network. In [60], a *counter-measure recommendation* method is introduced, which takes into account the cost values for recommended defense measures and total budget for the target network to spend for defense measures. Sheyner et al. [69] propose a *counter-measure recommendation* approach using the resulting attack graphs. Minimal and minimum atomic attack sets, which prevent the attacker from reaching her goals, are computed via utilization of the algorithms developed for the minimum cover problem. In [3], the generated attack graphs are used for *counter-measure recommendation* and predicting future attacks in near real-time.

A *counter-measure recommendation* method that represents the conjunction of negated goals as a logic proposition of the initially satisfied conditions on the attack graph and disables the appropriate initially satisfied conditions according to the obtained representation is proposed in [86]. In [12], a solution to the minimum cost network hardening problem based on Reduced Ordered Binary Decision Diagrams (ROBDDs) is presented. The solution is used for determining the *counter-measure recommendations* that incur minimum cost and minimize the number of attack paths that can be utilized by an attacker. In [33], a heuristic-based search solution approach to the minimum cost network hardening problem is proposed.

The works in [24] and [25] interpret attack graphs as Bayesian networks to provide a general *network security metrics computation* framework. In [61], the authors propose network security risk evaluation and mitigation plan assessment methods based on Bayesian attack graphs they formally define.

Templeton et al. [76] describe online attack scenario detection as a possible utilization method for the attack graphs generated off-line to support *near real-time security analysis*. It is also mentioned that the detected attack scenarios can be used for online future attack prediction for the target network. In [49], the authors propose computing a similarity measure between two attack (strategy) graphs that can be utilized in *near real-time security analysis*. It can be used during intrusion alert correlation to identify and hypothesize attacks missed by the intrusion detection/prevention sensors. The identification of the missed attacks can lead to more accurate detection of ongoing attack scenarios for the target network. In [59], Bayesian networks are generated from attack graphs by incorporating specific uncertainties defined in the context of *near real-time security analysis*. One of these uncertainties is related to the false positive and negative intrusion alerts.

An integrated tool called TVA, which can recommend near optimal defense measures using the resulting attack graphs is introduced in [34]. It also provides support for the determination of the near optimal locations for the intrusion detection/prevention devices on the target network based on attack graphs (*network design generation*).

5.5 Comparison of Past Works on Attack Graphs

In this section, a comparative analysis of the past works related to the attack graph generation and usage is provided. The analysis is presented in tabular format for easy reference according to the classification criteria proposed in the containing section. Additional comments related to possible improvements for these works are also included. In Table 5.1, the list of the past works is shown with their corresponding category for the classification criteria for reachability analysis.

Tab. 5.1: Classification of past works according to the reachability analysis criteria

Reachability Scope		**Reachability Content**			
Whole Network Reachability	**Atomic Domains Reachability**	**Filtering and Access Control Rules Modeling**	**IPS Modeling**	**Trust Relationships**	**Application Relationships**
All works mentioned in this section except: [32], [31], [11], [52], [34]	[32], [31], [11], [52], [34]	[69], [32], [31], [52], [34]	[31]	[69], [52], [34]	[32], [31], [52], [34]

Atomic domains reachability can provide an inherent support for the development of distributed attack graph core building algorithms. It can also eliminate the need for the whole reachability recomputation, when an update in the target network configuration (topology, filtering rules, etc.) occurs. Only the related atomic domain's reachability information is recomputed in this case. Its disadvantage can be the increased reachability information processing time during the attack graph core building process, since the connections among the atomic domains should be computed (at least searched inside a statically formed table) and traversed during this process.

Obtaining the IPS locations and signatures and using them to compute reachability gives more precise results compared to just using the filtering and access control rules and trust relationships. It can eliminate the attack paths employing exploits that are blocked by the IPS signatures. However, it is better to introduce some uncertainty measure in this process or use exploits and vulnerabilities together in the generation of attack graphs, because there can be a set of exploits for a vulnerability, some of which are blocked and some of which are not blocked by the corresponding IPS signatures. Using only vulnerability-based attack graphs without incorporating exploits may not give complete attack paths in this situation.

It is important to incorporate the application relationships on the target network as detailed as possible in the generation of the reachability conditions. Some specific information sources, which are contained by applications (software installations) and can point to other applications, can be defined and incorporated into the reachability graph generation process. An example of such an information source may be a database table managed by a web application and storing user credentials for a specific client application. When an attacker obtains file access right on the database table by, for instance, applying an SQL injection attack to the web application, she can read the user credentials for a completely different application and access this client application. This kind of relationship is not included by any reachability computation process for the past works. It may not be derived by just using vulnerability scanners and host-based asset managers. Cookies are another example for information sources. If an attacker can access the cookie store of a web browser, she can get *authorization right* on the web applications pointed to by clear-text cookies *directly*, without using any vulnerability.

Tab. 5.2: Classification of past works according to the attack graph modelling criteria

Attack Model			**Attack Graph Model**	
Manually-defined Attack Templates	**Intrusion Alert/System Log Correlation-based Attack Templates**	**Text Processing-based Attack Templates**	**Single Level**	**Multi-level**
[60], [67], [76], [69]	[49]	[38], [10], [29]	[3], [57], [32], [96]	[20], [4], [49], [38], [90]

In Table 5.2, the categorization of the past works according to the attack model and attack graph model is shown. Generating manually-defined attack templates is an error-prone task performed by human experts. Intrusion alert/system log correlation-based attack template generation may utilize some machine learning algorithms to perform the correlation of the collected logs and alerts. At the raw alert/log level, they can use rules generated by human experts to relate raw data fields. This kind of attack template generation may also introduce some false cause-consequence relationships among atomic attacks. For text processing-based attack templates to be performed, there should be some descriptive texts for vulnerabilities and exploits. These texts can be found in the public weakness and vulnerability databases on the Internet. For this kind of attack model generation to be more comprehensive, the information on more than one text source can be aggregated.

Multi-level attack graph models are more flexible than single level ones, since they can provide an inherent support for distributed attack graph core building algorithms, advanced (hierarchical) visualization of attack graphs and easy re-computation of attack graphs when the network configuration is updated. In essence, with multi-level attack graph model, the attack graphs are abstracted in more than one layer and only the required layers are processed for the intended attack graph computation process. It can decrease the time and space complexity of the attack graph-related processes, such as reachability computation, attack graph core building, etc. by concentrating only on the layers intended to be processed.

Tab. 5.3: Classification of past works according to the attack graph core building mechanism criteria

Attack Paths Determination Method		**Attack Paths Pruning Method**			
Logic-based Methods	**Graph-based Methods**	**No Pruning**	**Depth-Limited Attack Paths Pruning**	**Probability-based Attack Paths Pruning**	**Goal-oriented Attack Paths Pruning**
[67], [69], [57]	[60], [4], [3], [10], [43]	[60], [67], [4] [69], [57], [32], [31], [10]	[44], [43]	[44], [91]	[3], [7]

In Table 5.3, the classification of the past works according to the attack paths determination and paths pruning methods is shown. The logic-based methods employing model checking suffer from the scalability issue for network sizes greater than 10. The logic-based method proposed in [57] is somewhat different in the sense that its specific reasoning engine simulates the instantiations of attack rule templates in parallel with a specific tabled execution. The best worst-case run-time complexity obtained by the graph-based methods employing the monotonicity assumption is quadratic on the number of network hosts.

To decrease the number and expansion level of the attack paths, some of the past works propose pruning the attack paths according to specific criteria. Probability-based attack path pruning can provide more accurate pruning points for the attack paths than depth-limited attack paths pruning. Goal-oriented attack path pruning considers the importance of vulnerability exploits in the context of the (sub)goals of the attackers. For probability-based and goal-oriented attack path pruning methods, there must be some source providing an ordering of the likelihood of success values and importance values (in the context of the goals) for the vulnerability exploits. These values are dynamic in their nature, so temporal factors can also be accounted in the computation of them to obtain more precise attack paths. All these three attack paths pruning methods can be used together.

Tab. 5.4: Classification of past works according to the uses of attack graphs and a list of commercial tool suites

Uses of Attack Graphs				
Network Security Metrics Computation	**Countermeasure Recommendation**	**Near Real-time Security Analysis**	**Network Design Generation**	**Complete Commercial Tool Suites**
[20], [60], [24], [25], [84], [61]	[20], [60], [69], [3], [86], [12], [33], [52], [34], [61]	[76], [49], [59]	[52], [34]	([52], [34] - Cauldron), [71], [63]

Table 5.4 presents the categorization of the past works in terms of their uses of attack graphs. It also shows some commercial tool suites. Topological vulnerability analysis (TVA) tool described in [52] and [34] forms the basis of the integrated security suite Cauldron developed by George Mason University's Center for Secure Information Systems (CSIS). It incorporates network configuration extraction, vulnerability and exploit models, multi-step attack graph generation and visualization and optimal defense measures computa-

tion. The works of MIT Lincoln Laboratory about attack graphs ([32], [31], [52], [34]) are quite important for the TVA project. To extract software configuration for the target network, vulnerability scanners and host-based asset managers are used. Vulnerability and exploit models are derived by aggregating the information on the popular product and vulnerability databases on the Internet such as NVD [50] and OSVDB [56], etc. Attack graph generation utilizes the derived pre- and postconditions for the vulnerabilities. An attack simulation scenario can be given as input to the attack graph generation process. It can determine the start and end (goal) security states for the target network. The achieved worst-case time complexity for attack graph generation algorithms is on the quadratic order of the number of network hosts.

In TVA, there is an extensive support for the visualization of attack graphs. An attack graph can be divided into hierarchical layers according to specific criteria. The most important criterion is based on the protection domain abstraction. The hosts in a protection domain can access each other directly, without any connectivity limitations. The TVA tool can generate near optimal defense measures for network hardening by using the vulnerability (exploit) patches. There is also a support for determining the optimal places for the firewalls and intrusion detection/prevention devices on the target network based on the generated attack graphs.

Skybox Risk Control [71] is another commercial integrated vulnerability management/analysis tool suite that combines the capabilities to discover vulnerabilities daily, prioritize risks automatically and derive defense measures using the configuration of the target network. The commercial tool suite RedSeal [63] continuously analyses the configuration of the target network and determines the critical weaknesses and configuration flaws that can be utilized by an attacker to compromise the critical resources of the target network. The tool also performs risk analysis and provides interactive and continuously updated visualizations of the target network and security infrastructure for the target network.

In Table 5.5, the worst-case time complexities for the employed algorithms, the maximum number of network security-related objects used in the performed experiments and the distinguishing features for each of the past works are presented. Some of the entries in these tables are omitted, since no information about the entry is found in the related past work.

Tab. 5.5: Complexity measures, max network size used in experiments and significant features of past works

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[20]	-	-	Multi-level attack graph model composed of privilege and intrusion process state graphs, network security evaluation framework for risk analysis and defense measure recommendation
[60]	exponential in the number of network hosts	-	Introduction of manually-defined attack (scenario) templates as an attack model and unification mechanism to instantiate the attack templates using the attacker profile and network configuration information
[67]	exponential in the number of network hosts	4 network hosts and 15 vulnerabilities	Model-checking based network vulnerability analysis that finds a counter example to a given safety property
[76]	-	-	Definition of attack scenario templates using an attack specification language, called JIGSAW and generation of new templates from old ones via application of specific mutation methods

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[74]	exponential in the number of network hosts	2 network hosts and 5 attack templates	Realization of attack template concept and unification mechanism introduced in [60] by implementing an attack graph generation tool which can eliminate redundant nodes, edges and also cycles in the resulting attack graphs
[69]	exponential in the number of network hosts	5 network hosts and 8 atomic attack templates	Development of model-checking based analysis to find all counter examples to a given safety property and defense measure recommendation by finding minimal and minimum atomic attack sets that prevent the attacker from reaching his goals
[4]	$O(n^5)$ where n is the number of network hosts	3 network hosts and 6 vulnerabilities (atomic attacks)	Introduction of the monotonicity assumption and application of a breadth-first search with a specific marking procedure to the attack graph building problem

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[49]	-	-	Introduction of hyper alert correlation and attack strategy graphs which are generated from correlated intrusion alerts and hyper alert types representing possible attack templates
[70]	exponential in the number of network hosts	4 network hosts and 30 vulnerabilities	Presentation of the toolkit implementing the attack graph generation and analysis algorithms proposed in [69]
[3]	$O(n^3)$ where n is the number of network hosts	87 network hosts	Introduction of an attack graph whose nodes represent network hosts and edges represent highest-level access privileges that can gained by an attacker attacking from the source hosts to the target hosts of the edges
[57]	$O(n^2)$ where n is the number of network hosts	1000 network hosts and 100 vulnerabilities per host	Computation of attack graphs based on logic programming (reasoning)

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[38]	$O(V^n n!)$ where n is the number of network hosts and V is the number of vulnerability instances in the network	10 network hosts and 16 vulnerabilities	Modeling attack scenarios in a hierarchical structure
[32]	$O(n^2)$ where n is the number of network hosts	over 50,000 hosts and over 1.5 million ports	Introduction of reachability groups in the computation of the reachability matrix and introduction of the multiple-prerequisite attack graph structure containing attack prerequisites (reachability conditions) as nodes
[86]	exponential in the number of nodes in the attack graph	around 20 attack graph nodes using 3 network hosts and 9 vulnerabilities	Representation of the conjunction of the negated goals as a logic proposition of the initially satisfied conditions on the attack graph by disjunctive normal forms and using these forms to compute minimum cost defense recommendations
[44]	-	6 network hosts and 7 vulnerabilities	Utilization of depth limitation and likelihood of success value for an attack path to decide to expand it further

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[12]	exponential in the number of nodes in the attack graph	around 10 attack graph nodes	Conversion of an attack graph into a reduced ordered binary decision diagram and application of depth-first search and Shannon decomposition on this diagram to provide minimum cost defense recommendations
[7]	$O(m)$ where m is the number of nodes in the resulting attack graph	3 network hosts and 4 atomic attacks	Definition and removal of useless edges from attack graphs to get more scalable representations
[33]	polynomial in the number of nodes in the attack graph	over 40 attack graph nodes	Evaluation of various heuristics for selecting the initially satisfied conditions to be patched for minimum cost defense recommendation
[24]	-	3 network hosts and 5 vulnerabilities, almost 20 attack graph nodes	Introduction of conditional probabilities among vulnerabilities which are not connected in the attack graph
[25]	-	3 network hosts and 5 vulnerabilities, almost 20 attack graph nodes	Incorporation of temporal factors to compute successful exploit probabilities for vulnerabilities

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[84]	$O(n^2)$ where n is the number of nodes in the attack graph	3 network hosts, 5 vulnerabilities and around 20 attack graph nodes	Definition of the likelihood of an attack graph node being reached by an attacker for the first time in the attack graph and provision of a specific search algorithm to compute these likelihood values even in the context of cyclic attack graphs
[11]	-	5 network hosts and 4 vulnerabilities	Introduction of an atomic domain containing reachability information among a network host and its neighbours, division of the whole network reachability information into atomic domains
[31]	$O(n^2)$ where n is the number of network hosts	over 40000 network hosts, 52 personal firewalls each with at most 10000 filtering rules when the number of network hosts is held at 251	Introduction of the reverse reachability concept and support for modeling personal firewalls, proxy firewalls, intrusion prevention systems, client-side attacks and zero-day exploits

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[10]	$O(An^3)$ where A is the number of possible attack patterns and n is the number of network hosts	around 1000 network hosts each containing around three vulnerabilities	Definition of an abstraction over vulnerabilities, called an attack pattern which summarizes vulnerabilities that can be utilized with a common method of exploit
[52], [34]	$O(n^2)$ where n is the number of network hosts	over 40000 network hosts each with around 5 vulnerabilities	Presentation of a fully-functional automated network security testing tool suite which integrates attack simulation with attack graph construction, includes novel methods for the visualization of attack graphs and algorithms for placing firewalls and IDS/IPS devices optimally in the target network and determining optimal security counter-measures
[90]	$O(n^2)$ where n is the number of network hosts	4 network hosts and 9 vulnerabilities	Introduction of a two-tier attack graph model consisting of a sub-attack graph and a host access graph
[91]	$O(n^2)$ where n is the number of network hosts	6 network hosts and 17 vulnerabilities	Introduction of a probability-based attack paths pruning method by using three distinct types of probability values

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[96]	$O(n^2)$ where n is the number of network hosts	5 network hosts and 6 vulnerabilities	Definition of an attack graph model considering negative effects of the attacks on the performance of the target network
[43]	$O(n^{(L-1)/2})$ where n is the number of network hosts and L is the maximum search depth	5 network hosts and 5 vulnerabilities	Application of a bidirectional search method to compute an attack graph
[59]	-	4 network hosts and 4 vulnerabilities	Determination and modularization of domain-specific uncertainties in conditional probability tables of a Bayesian network which is used for near real-time security analysis
[29]	-	3 attack scripts each with around 10 extracted keywords and 53967 vulnerabilities defined in NVD ([50])	Introduction of an attack graph generation method which uses text-mining methods on the vulnerability definitions in NVD ([50]) to find the keywords defined in attack scripts generated by security experts

Past Work	**Worst-case Time Complexity Measure**	**Maximum Network Security Object Counts In Experiments**	**Significant Features**
[61]	$O(n^3)$ for attack graph generation and $O(2^n)$ for computing marginal probabilities, where n is the number of network hosts	9 network hosts and 13 vulnerabilities	Introduction of Bayesian attack graphs and usage of them to compute risk values and assess the effectiveness of specific mitigation plans

6 Attack Graph Generation

In this section, the basic problems related to attack graph generation are discussed and the solutions to these problems proposed in this thesis work are described. The solutions are generally related to attack graph modelling and attack graph core building mechanism. The contents of this section are mostly taken from [37] which was also written by the author of this thesis work.

6.1 Basic Problems in Attack Graph Generation

The problems that should be tackled for full or partial attack graph generation may arise in the initial preparation process or during the attack graph core building process. Of particular importance is the initial preparation process, since the structure of the resulting preparation data directly affects the complexity of the attack graph core building process. In determining the basic problems, the role of the vulnerabilities and privileges gained on the network hosts should be considered, as most of the attack graph generation algorithms proposed in the literature refer to these constructs in one way or other. Six basic problems are determined in this thesis work for attack graph generation: privilege determination, vulnerability data processing, network configuration information collection, reachability analysis, attack graph structure determination, and attack graph core building mechanism. These are detailed further in the following subsections.

6.1.1 Privilege Determination

An attack graph usually contains the privileges gained on the target network hosts by an attacker. Therefore, the determination of what can be a privilege shall be performed before the attack graph core building process. Example privileges include access levels (e.g., user, root), file access/modification rights, and memory access/modification rights. One can design privileges based on the applications installed on the network hosts, e.g., file modification rights on browser cookies or system files. When the detail level of the determined privileges increases, the precision of the resulting chains of the vulnerability exploits in the generated attack graphs increases, but the time and space requirements of the attack graph core building process may also grow.

6.1.2 Vulnerability Data Processing

An attack graph is generally built by finding the chains of the vulnerabilities, possibly on different network hosts, that can be exploited by a potential attacker one after the other. For an attacker to exploit a vulnerability, her current privileges on the attacking, target or any specific intermediate host must satisfy the preconditions of the vulnerability. Also, if the attacker succeeds in exploiting the vulnerability, she gains additional privileges on the attacked (target) host determined by the postconditions of the exploited vulnerability. Therefore, the pre- and postconditions of a vulnerability determine in order the exploit success for the vulnerability at a specific point during attack graph core building process and the privileges gained by an attacker after successful exploitation of the vulnerability.

Vulnerabilities are collected and managed in specific databases publicly accessible on the Internet. One of these databases is called National Vulnerability Database (NVD) [50]. (Another database is Open Source Vulnerability Database (OSVDB) [56].) However, the vulnerability descriptions in NVD are not completely machine-readable, impeding the easy parsing of the vulnerabilities and extraction of adequately detailed pre- and postconditions for them. Without applying complex text processing algorithms, one may not extract detailed pre- and post conditions which account for the application types, for instance memory access precondition on a *web browser*. For preconditions, only the connectivity and authorization requirements could be derived by applying basic XML parsing on the NVD data. The situation is similar for postconditions, as one may not determine application-type-based, direct or indirect postconditions by applying only XML parsing to the NVD data. For an example vulnerability exploited by an SQL injection attack, the postconditions derived by basic XML parsing may most likely not contain the indirect privileges gained by an attacker on the back-end database (indirect postconditions of the vulnerability), and instead these indirect privileges may be falsely represented as direct privileges on the web application on which the vulnerability exists.

An alternate solution to this problem may be to utilize the weakness descriptions in addition to the vulnerability descriptions. A weakness is an abstraction over one or more related vulnerabilities. An example weakness is insufficient input control for special characters. The weaknesses are independent of specific products, while the vulnerabilities are product-specific. Therefore, the number of weaknesses is far less than that of vulnerabilities, one can determine pre- and postconditions for the weaknesses semi-automatically or manually and relate these conditions with products, when processing the vulnerabilities. The Common Weakness Enumeration (CWE) database [19] currently describes around 1000 weaknesses, as of mid-2014. They are hierarchically organized and publicly accessible.

6.1.3 Network Configuration Information Collection

In order to perform network reachability analysis in the attack graph generation process, the information about the target network configuration shall be obtained. The configuration information can include the following: the topology of the target network, the applications (software or hardware installations) on the network hosts, the employed filtering and access control rules, the intrusion detection/prevention system configurations and the trust relations among the network hosts. The more network configuration information is obtained, the more accurate the computed attack graphs will be. This information directly affects the derivation of the reachability conditions among the target network hosts.

The vulnerabilities of the network hosts are determined by using the information about the applications installed on the hosts and the vulnerability databases on the Internet such as NVD [50] and OSVDB [56]. Determining the applications on the hosts includes the determination of both the networked and local applications. The networked applications, services, etc. can be determined by using network vulnerability scanners. However, the network vulnerability scanners may be blocked by the firewall rules on the target network. Then, for each protection domain (subnet), a separate vulnerability scanner session can be performed as exemplified in [52] and [34]. The examples of network vulnerability scanners are [77], [55], [28], [65]. The existence of the local applications is determined via asset detectors (host-based application/vulnerability scanners). The examples of host-based vulnerability scanners are [2] and [58]. However, these scanners may also be blocked by the personal firewalls and anti-virus scanners. Network discovery tools and vulnerability scanners can be used to extract network topology. One example network discovery tool is [51].

Network and host-based application and vulnerability scanners can provide valuable information; however, it may require a long time to gather the vulnerabilities in a network with thousands of hosts and millions of vulnerabilities. In this case, some assumptions may be utilized to summarize the application and vulnerability information for a group of hosts in a subnet. For instance, it may be assumed that the network administrator enforces a specific operating system and office software for all the hosts in a specific subnet. These assumptions may save time, since they can eliminate the need for running host-based scanners for all the hosts in a specific subnet and (some scanning plug-ins of) network-based scanners for all the subnets.

Collecting and processing the access control and filtering rules and intrusion prevention system configurations is also a cumbersome process, since there is no widely accepted common format for them. It can also be extremely difficult to gather and interpret these rules in large networks. One may have to resolve the conflicts among a large number of such rules to compute the reachability conditions correctly. For the intrusion prevention systems, finding that one or more exploits for a vulnerability are blocked by the intrusion prevention signatures may not mean that this vulnerability can not be utilized by an attacker for the target network. There can be an exploit that benefits from the vulnerability and is not blocked by the signatures. Therefore, more elaborate work may be needed to cross check the signatures with the well-known exploit and vulnerability databases. Processing the intrusion detection system configurations is much more difficult, since they may employ specific machine learning (anomaly detection) algorithms and the configuration options are related to these algorithms. The trust relations are also stored in custom formats for different types of applications, which may hinder simple processing.

There are also information sources like database tables, cookies and password files which can include sensitive user information for the applications totally irrelevant with their containing applications. For instance, a database table in a database server can be managed by a web application and may include the user credentials for this web application. Therefore, the information sources (assets) are also one of the main factors determining the reachability conditions among the network hosts. They can be changed from one installation to another for a specific software product. They can also be changed in time for a specific software installation (application). Although there are off-the-shelf tools to collect information sources of specific types from a target network, there is often no standard format for the content of different information sources of the same type.

6.1.4 Reachability Analysis

The attack graph core building process usually utilizes network reachability data to check for the target hosts' reachability by an attacker from the current attacking host. The network reachability data is mostly represented as a reachability matrix, where the columns and rows include the hosts in the network and each entry represents the reachability condition between the host on the corresponding row and the host on the corresponding column. Each entry in the reachability matrix may be a boolean or indicate the protocols that can be used between the two corresponding hosts to reach each other. A reachability matrix can be used to represent any type of connection among the hosts; physical, network, transport or application-level connection. Its space complexity is on the order of the square of the number of hosts in the network. In fact, it is not necessary to use a matrix to represent the reachability conditions. A (hyper)graph structure can also be used to represent them. By this way, the space complexity can be reduced.

The main factors affecting the reachability conditions are the filtering rules and access control lists defined on the (security) boundary devices in the target network, e.g. firewalls, routers. In fact, a reachability matrix or graph encodes the connectivity conditions among the hosts that may well be a precondition for the exploit of a specific vulnerability. The reachability analysis is usually performed before the attack graph core building process and the resulting reachability matrix or graph is put into a compact form according to specific algorithms to accelerate the attack graph core building process. It is important to determine the common patterns of the reachability conditions for the target network and find near optimal grouping schemes for the reachability conditions according to these patterns. This can minimize the redundant information in the resulting reachability matrix and speeds up the traversal of the reachability conditions during the attack graph core building process by minimizing the number of look-ups to the reachability matrix or graph.

6.1.5 Attack Graph Structure Determination

The space complexity of a full attack graph may easily reach an exponential order in the number of hosts in the target network, if each permutation of the possible vulnerability exploits on the hosts are recorded. To refrain from this exponential space complexity, there are various attack graph structures proposed in the literature. Although some of these structures are not as expressive as a full attack graph at the end, the others may have the same expressive power as a full attack graph by representing all possible attack paths. Generally, privileges and vulnerability exploits are used as the attack graph elements. However, in some works in the literature, other kinds of graph elements are introduced to reduce the space complexity of a full attack graph and the time complexity of building attack graphs. The resulting state space of an attack graph is also important in post-processing of the attack graph (defense measures recommendation, risk analysis, etc.). For the attack graph post-processing activities, the coverage of an attack graph is also significant. The attack graph shall contain the necessary states (nodes) and edges that can critically affect the decisions of a specific post-processing activity.

6.1.6 Attack Graph Core Building Mechanism

For both partial and full attack graph generation, the initial and target privileges possessed by the attacker can be given as input. For full attack graph generation, each possible attack path from the initial to the target privileges are found. For partial attack graph generation, this is not the case. Only a number of critical (shortest) attack paths may be found. Such specific partial attack graph generation problems may be formulated as artificial intelligence planning problems. The full attack graph generation process may be formulated as a general graph traversal problem, since it has to find all the attack paths. In essence, most of the attack graph generation algorithms proposed in the literature use some form of searching algorithm to find the corresponding nodes in the resulting attack graph. Some of them introduce domain-specific improvements to the basic prominent search algorithms existing in the artificial intelligence literature.

6.2 Attack Graph Modelling

Attack graph modelling is concerned with attack template modelling, attack graph structure determination and network modelling. The attack template modelling comprises the representation of pre- and postconditions for the vulnerabilities. It also includes a mechanism for the derivation of these conditions for specific vulnerabilities by using the information in public vulnerability and weakness databases. The determination of the attack graph structure includes deciding which types of nodes and edges can be found in an attack graph. The network modelling aims to determine an appropriate representation for the network assets (e.g., software/hardware applications running on the network hosts). In the following subsections, each of these sub-parts is described in detail.

6.2.1 Attack Template Modelling

Attack template modelling determines a model for the vulnerabilities and their pre- and postconditions and also suggests a method for creating instances of them. The attack template model used in our system is shown in Figure 6.1, with the formal definitions of its components following next.

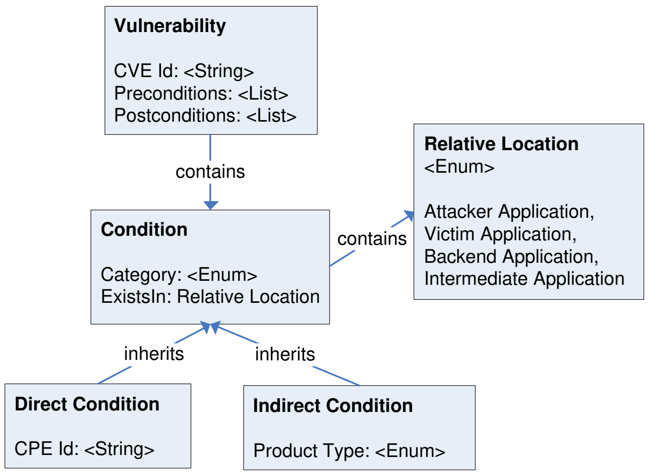


Fig. 6.1: Attack template model

Definition 6.2.1. A condition represents a right that can be gained on a software application. It is a two element tuple $< Category, ExistsIn >$, where *Category* represents the right gained on the software application and *ExistsIn* represents the location of the software application on which the right is gained. The location is determined relative to the attacker and victim software application. It can be the attacker software application, victim software application, a backend application of the victim software application or an intermediate software application that is located between the attacker and victim application and can intercept the traffic between them.

A condition is independent from any IP network and does not specify an IP address in its definition. For the conditions, in the first place, the possible values for the category (right) of a condition shall be determined. The level of detail for the condition categories shall be adjusted carefully to adequately express the pre- and postconditions of the vulnerabilities and also to eliminate the chaining of the unrelated vulnerabilities in the resulting attack graphs. The proposed scheme for the determination of the condition categories is illustrated in Figure 6.2. The condition categories are hierarchically organized.

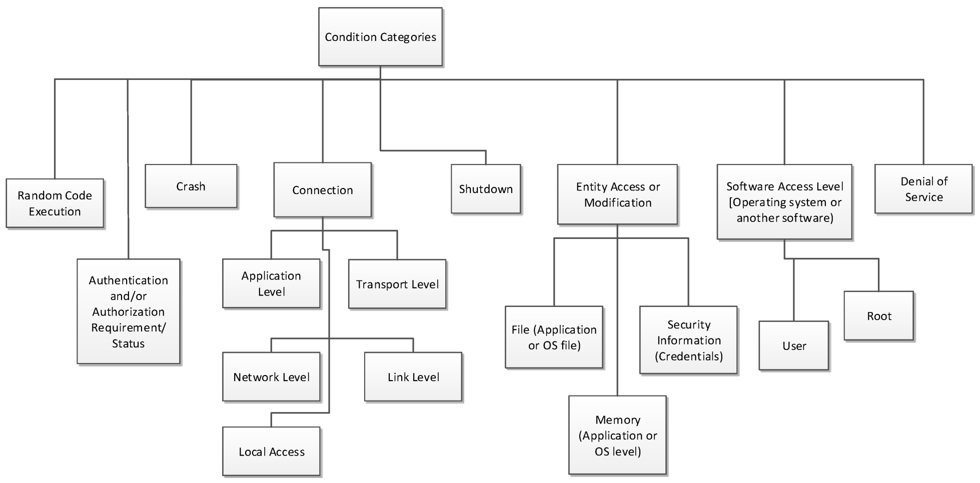

Condition Categories
Random Code Execution
Authentication and/or Authorization Requirement/ Status
Crash
Connection
Application Level
Transport Level
Network Level
Link Level
Local Access
Shutdown
Entity Access or Modification
File (Application or OS file)
Security Information (Credentials)
Memory (Application or OS level)
Software Access Level (Operating system or another software)
User
Root
Denial of Service

Fig. 6.2: Pre and Postcondition Categories

Definition 6.2.2. A direct condition specifies an additional element to a condition tuple. This element is named *CPEId* and represents the CPE product identifier [15] of the software application on which the condition category is gained.

Definition 6.2.3. An indirect condition specifies an additional element to a condition tuple. This element is named *ProductType* and specifies the product type of the software application on which the condition category is gained.

The product types are defined in the proposed system. A product type can be mail server, mail client, web server, web client, ftp client, database server application, etc. indicating the technology class of the software application. We have manually derived product types for around 10000 CPE identifiers related to the mostly used products in the information technology sector. The derived product type-CPE identifier matchings are used in the attack graph building process.

Definition 6.2.4. A vulnerability in thesis work is defined as a single CVE entry defined in the CVE database ([18],[50]). It is represented by a three element tuple $< CVEId, Preconditions, Postconditions >$. A vulnerability is identified by its unique CVE [18] identifier stored in the $CVEId$ element. It has a list of preconditions that denote the list of the required attacker privileges to exploit the vulnerability. The preconditions are stored in the list $Preconditions$. There is a conjunction relation among the preconditions. A vulnerability also has a list of postconditions that denote the privileges obtained by the attacker after successfully exploiting the vulnerability. The postconditions are stored in the list $Postconditions$. There is a disjunctive relation among the postconditions. A precondition and a postcondition can be a direct or indirect condition.

There are over 70000 vulnerabilities described in NVD [50]. Our method for generating the pre- and postconditions for the vulnerabilities parses the vulnerability descriptions in NVD to find the base pre- and postconditions for them. In this context, the *Access Vector* and *Authentication* fields of a vulnerability record are utilized to extract the postconditions for the vulnerability. The *Impact Type* field of a vulnerability record is used to extract the preconditions for the vulnerability.

The vulnerability descriptions in NVD contain some useful information to enrich the pre- and postconditions for the vulnerabilities. However, they are not in a form that allows the automatic generation of the pre- and postconditions easily and mostly, they are not detailed enough to derive the pre- and postconditions in an adequate granularity. Most of the descriptions are very brief explanations of how to use the corresponding vulnerabilities. Based on these considerations, our method for generating the pre- and postconditions for the vulnerabilities utilizes also the weaknesses defined in the CWE database [19] and the vulnerability-weakness mappings provided by NVD in order to enrich the base pre- and postconditions generated by using only the NVD data and make them more accurate.

We manually generate the pre- and postconditions for the weaknesses existing on a specific weakness hierarchy formed by CWE. *Research Concepts (CWE-1000)* hierarchy is used. The relatively small number of weaknesses (on the order of 100) allows manual pre- and postcondition generation according to the proposed attack template model. In order to derive the preconditions for the weaknesses, the *Description*, *Applicable Platforms*, *Modes of Introduction* and *Enabling Factors for Exploitation* fields of the CWE weakness records are examined manually. In order to derive the postconditions for the weaknesses, the *Description*, *Applicable Platforms* and *Common Consequences* fields of the CWE weakness records are examined manually. An example can be given by using the SQL injection weakness with CWE identifier CWE-89. The manually derived precondition for the SQL injection weakness is:

1. *Transport Level Connection to an Application with Product Type Web Server on the Victim Side.*

The manually derived postconditions for the SQL injection weakness are:

1. *Authorization on an Application with Product Type Web Server on the Victim Side*,
2. *File Modification on an Application with Product Type Database Server on the Backend Side.*

6.2.2 Attack Graph Structure Determination

Attack graph structure determines the nodes and the edges of the generated attack graphs. The proposed attack graph structure is shown in Figure 6.3.

An attack graph can contain four types of nodes. The formal definitions of the different types of nodes in an attack graph and attack graph are given below.

Definition 6.2.5. A privilege node represents an attacker privilege on a software application on a network host and is a six element tuple $< IPAddress$, $CPEId$, $ApplicationName$, $Category$, $InEdges$, $OutEdges >$. $IPAddress$ denotes the IP address on which the software application is running. $CPEId$ is the CPE [15] identifier of the product related to the software application.

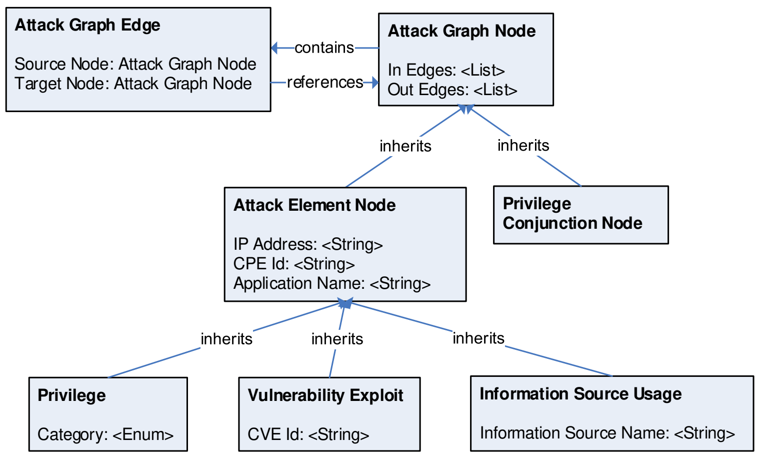


Fig. 6.3: Attack graph structure

$ApplicationName$ is the name of the software application. $Category$ represents the category of the condition possessed by the attacker on the software application. Possible condition categories are shown in Figure 6.2. $InEdges$ and $OutEdges$ are the lists holding references to the in and out attack graph edges connected to the privilege node.

Definition 6.2.6. A privilege conjunction node denotes a conjunction connector for a number of privilege nodes in an attack graph. It is a two element tuple $< InEdges, OutEdges >$. $InEdges$ and $OutEdges$ are the lists holding references to the in and out attack graph edges connected to the privilege conjunction node.

Definition 6.2.7. A vulnerability exploit node represents an exploit of a vulnerability on a software application on a network host by an attacker. It is a six element tuple $< IPAddress, CPEId, ApplicationName, CVEId, InEdges, OutEdges >$. $IPAddress$, $CPEId$, $ApplicationName$ elements are defined the same as in Definition 6.2.5. $CVEId$ is the unique identifier of the exploited vulnerability and is defined in the CVE [18] database. $InEdges$ and $OutEdges$ are the lists holding references to the in and out attack graph edges connected to the vulnerability exploit node.

Definition 6.2.8. An information source usage node represents an access and usage of an information source (cookie file, DNS table, database table, etc.) on a software application on a network host by an attacker. It is a six element tuple $< IPAddress, CPEId, ApplicationName, ISName, InEdges, OutEdges >$. $IPAddress$, $CPEId$, $ApplicationName$ elements are defined the same as in Definition 6.2.5. $ISName$ is the name of the used information source inside the software application. $InEdges$ and $OutEdges$ are the lists holding references to the in and out attack graph edges connected to the information source usage node.

Definition 6.2.9. An attack graph is a graph $G = (N, E)$, where N denotes the set of nodes and E denotes the set of edges of the graph G. $n \in N$ can be a privilege, privilege conjunction, vulnerability exploit or information source usage node. $e \in E$ is a two element tuple $< SourceNode, TargetNode >$, where $SourceNode$ denotes the source and $TargetNode$ denotes the target node for the edge e. The possible node types for $SourceNode$ and $TargetNode$ are $< Pr, Ve >$, $< Pr, Pc >$, $< Pr, Isu >$, $< Pc, Ve >$, $< Pc, Isu >$, $< Ve, Pr >$ and $< Isu, Pr >$, where Pr denotes a privilege, Ve denotes a vulnerability exploit, Pc denotes a privilege conjunction and Isu denotes an information source usage node.

The edges of an attack graph just denote the relationships among different types of nodes. There can be an edge:

1. from a privilege to a vulnerability exploit or an information source usage indicating that the existence of only the privilege is sufficient and necessary for an attacker to exploit the vulnerability or the information source usage,

2. from a privilege to a privilege conjunction indicating that the privilege is one of the necessary preconditions for an attacker to exploit the vulnerability or information source usage targeted by the privilege conjunction,

3. from a privilege conjunction to a vulnerability exploit or an information source usage indicating that the acquisition of the privileges connected to the privilege conjunction by an attacker gives rise to the vulnerability exploit or the usage of the information source,

4. from a vulnerability exploit or an information source usage to a privilege indicating that the exploit of the vulnerability or usage of the information source gives rise to the acquisition of the privilege by an attacker.

An example attack graph is shown in Figure 6.4.

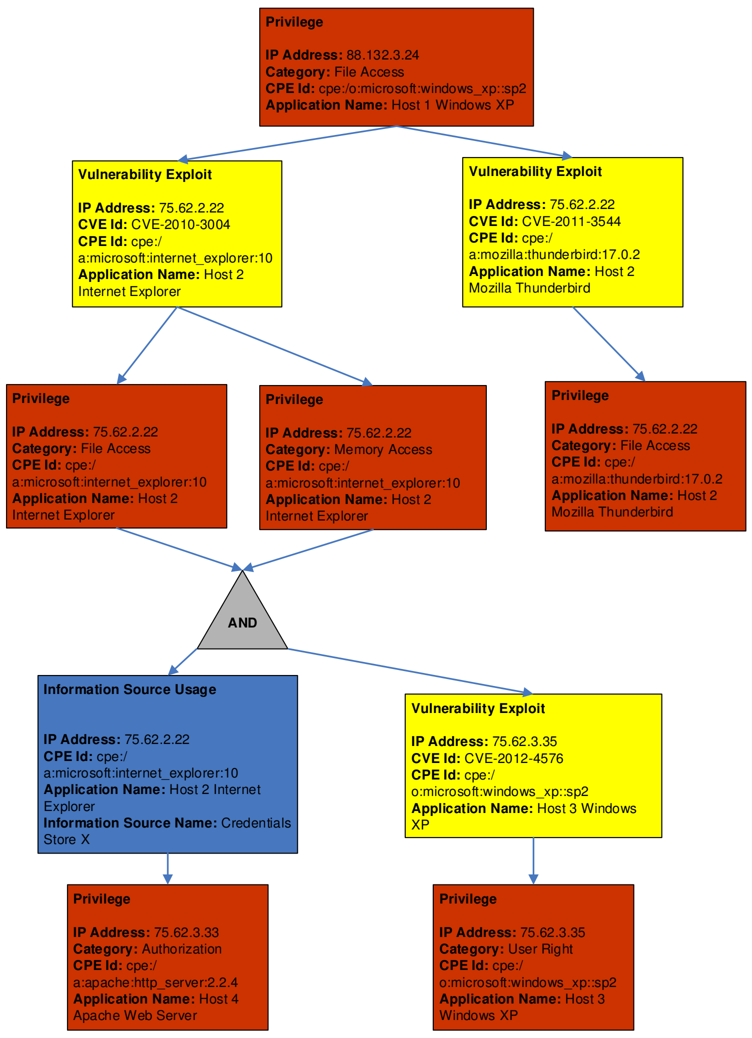

Privilege
IP Address: 88.132.3.24
Category: File Access
CPE Id: cpe:/o:microsoft:windows_xp::sp2
Application Name: Host 1 Windows XP
Vulnerability Exploit
IP Address: 75.62.2.22
CVE Id: CVE-2010-3004
CPE Id: cpe:/ a:microsoft:internet_explorer:10
Application Name: Host 2 Internet Explorer
Vulnerability Exploit
IP Address: 75.62.2.22
CVE Id: CVE-2011-3544
CPE Id: cpe:/ a:mozilla:thunderbird:17.0.2
Application Name: Host 2 Mozilla Thunderbird
Privilege
IP Address: 75.62.2.22
Category: File Access
CPE Id: cpe:/ a:microsoft:internet_explorer:10
Application Name: Host 2 Internet Explorer
Privilege
IP Address: 75.62.2.22
Category: Memory Access
CPE Id: cpe:/ a:microsoft:internet_explorer:10
Application Name: Host 2 Internet Explorer
Privilege
IP Address: 75.62.2.22
Category: File Access
CPE Id: cpe:/ a:mozilla:thunderbird:17.0.2
Application Name: Host 2 Mozilla Thunderbird
AND
Information Source Usage
IP Address: 75.62.2.22
CPE Id: cpe:/ a:microsoft:internet_explorer:10
Application Name: Host 2 Internet Explorer
Information Source Name: Credentials Store X
Vulnerability Exploit
IP Address: 75.62.3.35
CVE Id: CVE-2012-4576
CPE Id: cpe:/ o:microsoft:windows_xp::sp2
Application Name: Host 3 Windows XP
Privilege
IP Address: 75.62.3.33
Category: Authorization
CPE Id: cpe:/ a:apache:http_server:2.2.4
Application Name: Host 4 Apache Web Server
Privilege
IP Address: 75.62.3.35
Category: User Right
CPE Id: cpe:/ o:microsoft:windows_xp::sp2
Application Name: Host 3 Windows XP

Fig. 6.4: An Example Attack Graph

6.2.3 Network Modelling

The network model is used to model the target network topology and installed software configuration on the target network. It includes the network hosts as the main elements. The network hosts are connected to each other via their contained network interfaces. Each network interface has an IP address and a reference to the communication link, which connects it to another network interface contained in another network host. Our network model is illustrated in Figure 6.5 and formally defined next.

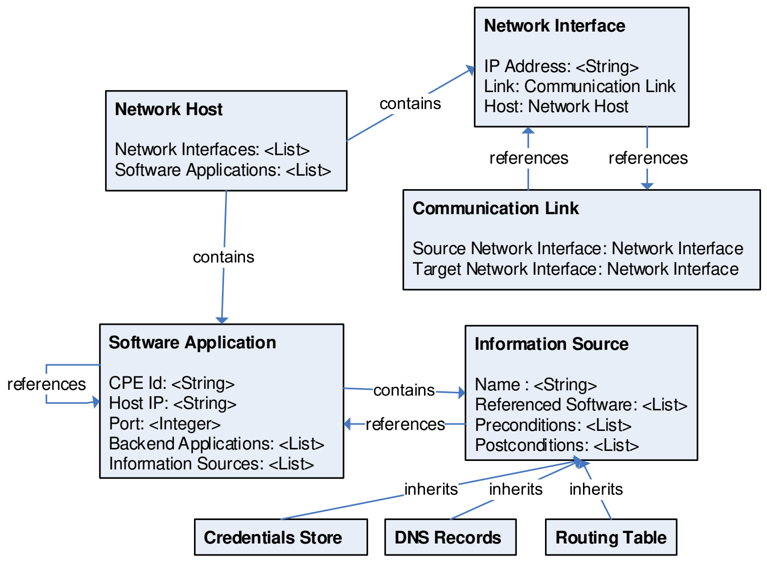


Fig. 6.5: Network model

Definition 6.2.10. A network host is a two element tuple $< NetworkInterfaces, SoftwareApplications >$, where $NetworkInterfaces$ is a list of the network interfaces contained by the network host and $SoftwareApplications$ is a list of the software applications installed on the network host.

Definition 6.2.11. A network interface denotes a OSI Layer 3 interface on a network host and is a three element tuple $< IPAddress, Link, Host >$. $IPAddress$ is the IP address associated with the network interface. $Link$ denotes the communication link connected to the network interface. $Host$ denotes the network host containing the network interface.

Definition 6.2.12. A communication link connects two network interfaces and is a two element tuple $< SourceNI, TargetNI >$. $SourceNI$ refers to the source network interface, $TargetNI$ refers to the target network interface connected to the communication link. The designation of the source or target network interface is performed randomly and does not impose any restriction on the data transfer direction between the network hosts containing the network interfaces.

Definition 6.2.13. A software application is a five element tuple $<$ $CPEId$, $HostIP$, $Port$, *BackendApplications*, $InformationSources$ $>$. $CPEId$ denotes the software product identifier (CPE identifier [15]), $HostIP$ denotes the IP address on which the software application is serving (running) and *Port* denotes the port on which it is serving. *BackendApplications* refers to the software applications whose services are used by this software application. $InformationSources$ is a list of information sources contained by the software application such as credentials stores, cookies, DNS tables, routing tables and databases.

A software application object stores references to the other software applications used by the software application in its backend. For instance, a web server application may reference a database server application as its backend application. If an attacker uses a specific vulnerability on this web server application, she can gain privileges on the backend database server application without even using a vulnerability on the database server application. The backend software applications are accounted for during the attack graph core building process.

Definition 6.2.14. An information source denotes a sensitive data store that is contained by a software application and can be accessed and used by an attacker. It is represented by a three element tuple $<$ $ReferencedSoftware$, *Preconditions*, *Postconditions* $>$. In order to use an information source, an attacker should satisfy the preconditions that are stored in the list *Preconditions* for the information source. After successfully benefiting from the information source, the attacker gains the postconditions that are stored in the list *Postconditions* for the information source. The postconditions are gained on the software applications referenced by the information source that are stored by the element $ReferencedSoftware$.

An attacker can gain privileges on a host by benefiting from the information sources on a software application on a different host, e.g., without using any vulnerability defined in NVD. As an example, an attacker gaining access to the cookies file on a web browser on one host may gain authorization rights to a web application on another host by using the information stored in the cookies file.

6.3 Proposed Attack Graph Core Building Mechanism

The attack graph core building process is treated as a search problem in this thesis work, which is solved in a distributed multi-agent environment. For this purpose, a reachability hyper-graph is first formed and then, a modified parallel version of the classical depth-first search algorithm is applied by taking the reachability conditions stored in the hyper-graph into account. One of the main issues in parallel searching in a distributed memory environment is the difficulty of eliminating duplicate attack graph node expansions. To eliminate this problem, a virtual shared memory abstraction is implemented to provide memory coherency over the distributed search agents.

6.3.1 Reachability Hyper-graph and Partitioning

Reachability determines the accessibility conditions among the software applications installed on the target network. The filtering rules on the firewalls, access control lists on the routers, security policies applied among the software applications are among the factors that determine the reachability conditions. All these factors are taken into account to create our reachability hyper-graph, in which a hyper-vertex indicates a software application. Each hyper-vertex has an associated weight value that indicates the density of the workload related to the software application. The computation of this workload accounts for the number of the attack graph nodes for the software application that will be created by the search agents.

A hyper-edge indicates a collection of source and target software applications such that the source applications can directly access the target applications, if they satisfy specific conditions. These conditions, allowing direct reachability among the software applications, are stored in the hyper-edge. Such conditions can include network protocols, port numbers, user credentials, etc. A hyper-graph is more efficient in terms of storage space than a reachability matrix or graph.

The resulting reachability hyper-graph is partitioned to determine the initial tasks of each distributed search agent responsible for generating the attack graph. Each search agent shall be responsible for determining the attacker privileges, vulnerability exploits and information source usages for a number of software applications. The software applications are allocated to the search agents with hyper-graph partitioning in such a way that the number of messages transferred among the search agents, when building the resulting attack graph, is minimized and the workload of the search agents are balanced. Any hyper-graph partitioning algorithm can be used to partition the reachability hyper-graph, e.g. [35].

As stated in [35], hyper-graph partitioning is a significant problem with many application areas, including VLSI design, efficient storage of large databases on disks and data mining. The problem is to partition the vertices of a hyper-graph into k roughly equal parts, such that a certain objective function defined over the hyper-edges is optimized. A commonly used objective function is to minimize the number of hyper-edges that span different partitions. These hyper-edges are cluttered across different partitions.

In [35], the authors propose a hyper-graph partitioning method based on a greedy k-way partition refinement algorithm. They successively coarsen the given hyper-graph into smaller representative hyper-graphs, partition the smallest representative hyper-graph into k roughly equal parts and then refine the partitions by traversing the representative hyper-graphs back to the original hyper-graph. The partition refinement algorithm is performed by following a greedy approach that selects the vertices to be moved across the partitions by considering the greatest positive reduction (gain) in the objective function caused by the vertex movements. The greedy approach also accounts for the load balancing constraints among the partitions in order to create roughly equal partitions. Hyper-vertex weights are considered in load balancing.

The main aim of the reachability hyper-graph partitioning is to achieve the load balancing in terms of the hyper-vertex weights and minimize the number of the hyper-edges spanning across the search agents. We assign large weight values to the hyper-vertices (software applications) related to the initial attacker privileges, since main work density in the attack graph computation will be related to (i.e., will be in the neighbourhood of) these hyper-vertices. The minimization of the cluttered edges across the search agents provides for the minimization of the transferred messages among the search agents during the attack graph computation. An example reachability hyper-graph is depicted in Figure 6.6.

The software applications that have a common color in Figure 6.6 can directly access each other. After partitioning, the reachability hyper-graph contains minimum number of cluttered hyper-edges and the total weight of the hyper-vertices contained in each partition is comparable (load balancing). The hyper-vertices (software applications), which are related to the initially

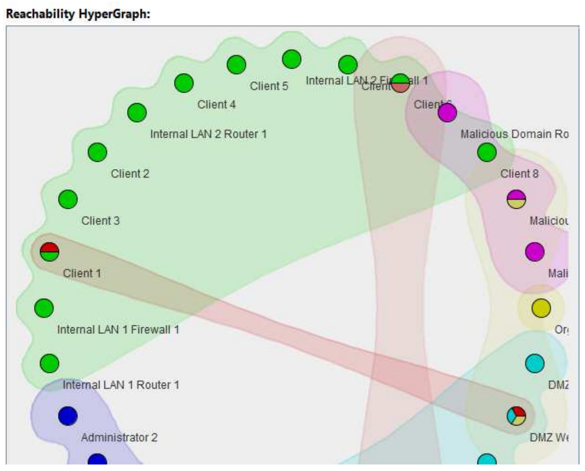


Fig. 6.6: An Example Reachability Hyper-graph

satisfied attacker privileges, are assigned higher weight values than the other hyper-vertices, so that the workload can be distributed to the search agents almost equally.

6.3.2 Virtual Shared Memory Abstraction

The main aim of the virtual shared memory abstraction is to provide memory coherency across all the distributed search agents. The *Dynamic Distributed Memory Manager Algorithm* described in [40] is employed in this paper. In this algorithm, there is no centralized memory manager. Each distributed search agent stores information about the attacker privileges for the software applications that are assigned to it in its local memory. This information at least comprises the expansion status of the corresponding privileges, namely whether they are used to exploit some vulnerabilities or benefit from information sources to generate additional privileges.

As a distributed search agent is performing, it may need to expand an attacker privilege and request information (expansion status) about the attacker privilege, which is stored in another agent's local memory. In this case, it has to transfer the corresponding information into its local memory. Some messages are transferred among the search agents according to the algorithm described in [40] to transfer the corresponding memory page to the local memory of the requesting search agent. If the corresponding information indicates that the associated privilege has already been expanded, then the requesting agent does not expand it. Otherwise, the requesting agent pushes the privilege to its search stack to expand it later, sets its expansion status to true and invalidates the copies of the corresponding memory page in other agents by sending messages to them.

In this proposed method, it is important to decrease the memory read/write faults, thereby memory page transfers. For this, it is crucial to determine a reasonable initial configuration for the virtual shared memory. At the initialization of the virtual shared memory, one must carefully select which memory pages contain which memory objects (privileges) and which search agents store which memory pages locally. In this paper, this is performed by using the results of the reachability hyper-graph partitioning. Hyper-vertices in the reachability hyper-graph, representing software applications on which the privileges are defined, are distributed as equally as possible to the search agents. The number of hyper-edges cluttered across the search agents is minimized by the partitioning process to minimize the number of transferred messages among the search agents. In summary, the distribution of the memory objects to the search agents' local memories is performed according to the reachability conditions among the software applications related to the memory objects. These reachability conditions are used by the search agents during the attack graph building process.

6.3.3 Parallel, Shared Memory-Based Depth-first Search

In this thesis work, a parallel, shared memory-based depth-first search method is proposed as the core attack graph building algorithm. Each distributed search agent performs this algorithm. Actually, each distributed search agent performs a stack-based depth-first search starting by expanding the initially satisfied attacker privileges assigned to it. The decision of expanding a privilege during the search is determined by examining the boolean expansion status of it stored in the virtual shared memory. If an agent generates a privilege, it first queries the shared memory to learn whether the privilege has already been expanded or not. If the privilege has not already been expanded, the agent sets its expansion status to true and pushes the privilege to its search stack to expand it later.

When a search agent has no privilege to expand in its search stack at a certain time, it starts to request one or more privilege from other search agents. When a search agent requests privilege from the search agent x, the search agent x sends a random number of privileges that are stored on the bottom half of its search stack to the requesting search agent, if it has sufficient privileges in its stack. If the request has resulted in no privilege transfer, then the requesting search agent tries other search agents. If no privilege transfer is occurred after a specified number of trials for each other search agent in a cyclic manner, then the requesting search agent gives up and enters into a passive state, but does not terminate. When all the search agents enter into the passive state, the distributed search algorithm is assumed to be terminated.

Parallel, Distributed Search Performed By Search Agents

The proposed parallel, distributed shared memory based depth-first search algorithm is given in Figure 6.7. It is executed by each distributed search agent on the multi-agent platform. Before performing the search algorithm, the reachability hyper-graph has already been generated and the virtual shared memory pages have already been initialized with the hyper-graph partitioning results.

During the distributed search algorithm, each agent uses the previously computed reachability hyper-graph to determine which target software applications are reachable from the software application related to the currently processed privilege (FindContainingEdges() and FindTargetSoftwareApps()). Then, the vulnerabilities and information sources on each of these target software are fetched and their exploitability/usability are checked by calling the function CheckExploitability(). For each exploitable vulnerability and usable information source, the new privileges gained by the attacker are computed by calling the function FindGainedPrivileges(). The expansion status of all the gained privileges stored in the shared memory are updated by calling the function UpdateGainedPrivilegesInMemory().

The flow chart for the parallel, distributed shared memory-based depth-first search algorithm performed by each search agent is also given in Figure 6.8.

Checking Exploitability of Vulnerabilities and Information Sources

During the parallel, distributed shared memory-based depth-first search algorithm, the exploitability of each vulnerability/information source in the software applications currently reachable by an attacker is checked via matching the already expanded (searched) privileges in shared memory to the preconditions of the vulnerability/information source. This is performed by the function CheckExploitability(). The pseudocode for the CheckExploitability() function is shown in Figure 6.9.

```
procedure PerformDFS($RHG, IPRGS$)    ▷ Reachability hyper-graph (RHG) and initial attacker privileges (IPRGS) are inputs
    $MainStack$ ←CreateMainStack()    ▷ Create the search main stack
    for all $initialPrivilege \in IPRGS$ do
        $initialPrivilegeStatus$ ←new PrivilegeStatus()
        $initialPrivilegeStatus.setExpanded(true)$
        WriteToSharedMemory($initialPrivilege, initialPrivilegeStatus$)
        $MainStack.push(ip)$
        $foundPrivileges.add(initialPrivilege)$
    end for
    while $true$ do
        if $MainStack.size() > 0$ then    ▷ Continue to the search while there are privileges on the main stack
            $currentPrivilege \leftarrow MainStack.pop()$
        else
            $receivedPrivileges$ ←GetWorkFromOtherAgents()    ▷ Requests privileges from other agents to expand
            if $receivedPrivileges.size() == 0$ then
                break
            else
                $MainStack.push(receivedPrivileges)$
                $foundPrivileges.addAll(receivedPrivileges)$
                continue
            end if
        end if
        $vertex$ ←FindVertexForPriv($currentPrivilege, RHG$)
        $edges$ ←FindContainingEdges($vertex, RHG$)
        $gainedPrivileges$ ←new List()
        for all $edge \in edges$ do
            $targetSoftwareApps$ ←FindTargetSoftwareApps($edge$)
            for all $targetSoftwareApp \in targetSoftwareApps$ do
                for all $vulnerability \in targetSoftwareApp.vulnerabilities()$ do
                    $requiredPrivileges$ ←CheckExploitability($vulnerability, currentPrivilege,$
                        $targetSoftwareApp$)
                    if $requiredPrivileges != null$ then ▷ Vulnerability can be exploited by an attacker
                        $vulnGainedPrivileges$ ←FindGainedPrivileges(
                            $vulnerability, currentPrivilege, targetSoftwareApp$)
                        $gainedPrivileges.addAll(vulnGainedPrivileges)$
                        UpdateAttackGraph($vulnerability,$
                            $requiredPrivileges, vulnGainedPrivileges, targetSoftwareApp$)
                    end if
                end for
                for all $infoSource \in targetSoftwareApp.infoSources()$ do
                    $requiredPrivileges$ ←CheckExploitability($infoSource, currentPrivilege,$
                        $targetSoftwareApp$)
                    if $requiredPrivileges != null$ then    ▷ Information source can be used by an attacker
                        $infoSourceGainedPrivileges$ ←FindGainedPrivileges(
                            $infoSource, currentPrivilege, targetSoftwareApp$)
                        $gainedPrivileges.addAll(infoSourceGainedPrivileges)$
                        UpdateAttackGraph($infoSource, requiredPrivileges,$
                            $infoSourceGainedPrivileges, targetSoftwareApp$)
                    end if
                end for
            end for
        end for
        UpdateGainedPrivilegesInMemory($gainedPrivileges, MainStack, foundPrivileges$)
    end while
end procedure
```

Fig. 6.7: Parallel, distributed shared memory-based depth-first search algorithm

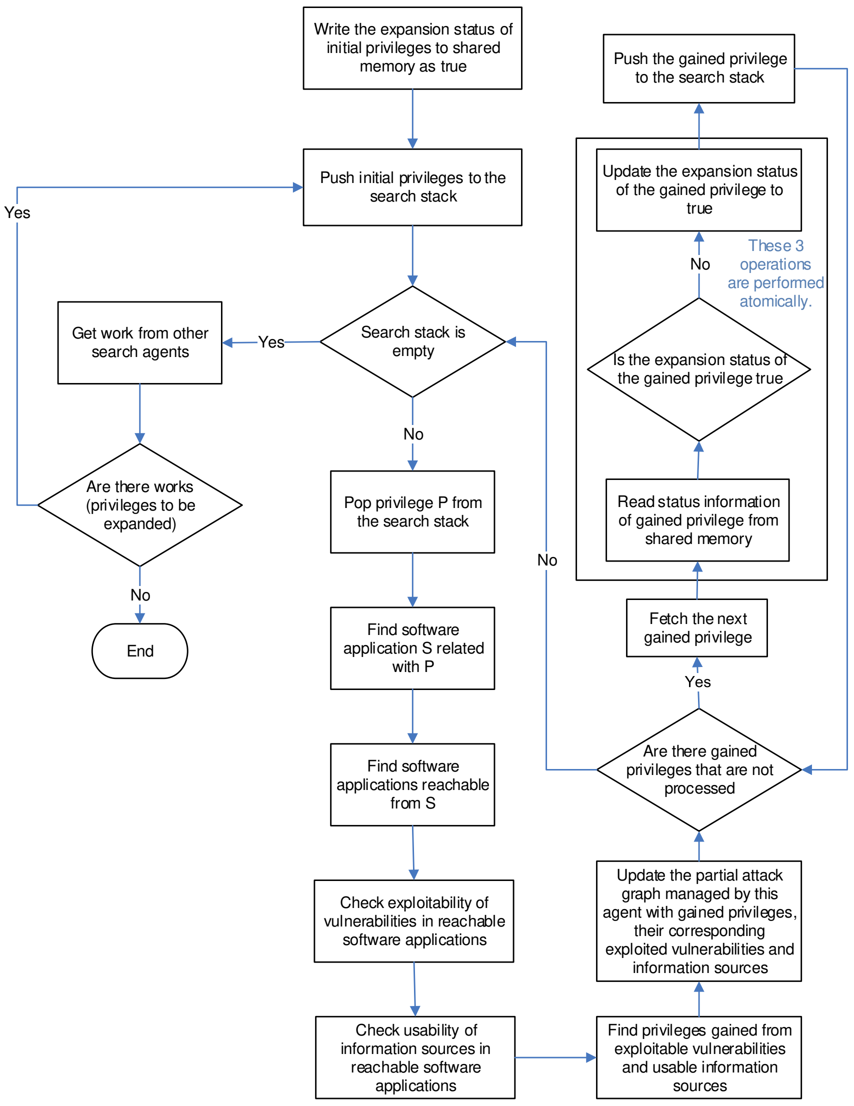

Write the expansion status of initial privileges to shared memory as true
Push initial privileges to the search stack
Yes
Get work from other search agents
Yes
Search stack is empty
No
Are there works (privileges to be expanded)
No
End
Pop privilege P from the search stack
Find software application S related with P
Find software applications reachable from S
Check exploitability of vulnerabilities in reachable software applications
Check usability of information sources in reachable software applications
Find privileges gained from exploitable vulnerabilities and usable information sources
Update the partial attack graph managed by this agent with gained privileges, their corresponding exploited vulnerabilities and information sources
Are there gained privileges that are not processed
No
Yes
Fetch the next gained privilege
Read status information of gained privilege from shared memory
Is the expansion status of the gained privilege true
No
Update the expansion status of the gained privilege to true
These 3 operations are performed atomically.
Push the gained privilege to the search stack

Fig. 6.8: Flow chart for the parallel, distributed shared memory-based depth-first search algorithm

```
foundPrivileges ← new Set()                          ▷ Global variable across the search agent
function CheckExploitability(SP, CP, TSA)    ▷ SP can be a vulnerability or information source,
CP is the current privilege, TSA is the target software application
    requiredPrivileges ← FormPrivileges(SP.preConditions(), CP.softwareApp, TSA)
    if not (requiredPrivileges contains CP) then
        return null;
    end if
    for all requiredPrivilege ∈ requiredPrivileges do
        if not (foundPrivileges contains requiredPrivilege) then
            requiredPrivileges ←ReadFromSharedMemory(requiredPrivilege)
            if requiredPrivilege.expanded == false then  ▷ If the privilege is not expanded, then it
means that it is not generated by any search agent until now.
                return null
            end if
        end if
    end for
    return requiredPrivileges
end function
```

Fig. 6.9: Checking exploitability of a vulnerability or an information source by an attacker

In the function CheckExploitability(), first, the required privileges for the exploitation of the input vulnerability or information source are found by calling the function FormPrivileges(). The FormPrivileges() function makes the given preconditions of the input vulnerability or information source specific to the given source and target software application by adding the information (IP address, CPE [15] name, etc.) of the source and target software application to them. The second parameter of the FormPrivileges() function is the source (attacking) software application and the third parameter is the target (attacked) software application. For instance, if the condition is an indirect condition and specifies a product type instead of a CPE name on the attacker relative location, then the FormPrivileges() function makes this condition more specific by attaching the CPE name of the source software application to it. However, this is possible, only if the source application has the same product type specified in the condition.

If the currently traversed privilege, given as parameter, does not exist in the required privileges, then the CheckExploitability() function returns null, indicating that the satisfaction conditions (preconditions) of the input vulnerability or information source are unrelated to the currently traversed privilege. A required privilege is satisfied, if and only if it is found in the *global list foundPrivileges* or it has already been written into the virtual shared memory (its expanded status is true). (*foundPrivileges* is a global set used in both algorithms shown in Figure 6.7 and Figure 6.9.) If at least one of the required privileges is not satisfied, then the function CheckExploitability() returns null indicating that the input vulnerability or information source can not be exploited/used by the attacker now.

If the required privileges returned by the function CheckExploitability() is not null, the vulnerability or the information source can be exploited/used by an attacker. The newly gained privileges after the exploitation are computed from the postconditions of the exploitable vulnerability or usable information source by calling the function FindGainedPrivileges() in Figure 6.7. The details of the process of computing the newly gained privileges are explained next.

Computing Gained Privileges and Updating Status of Them in Shared Memory

When computing the newly gained privileges, *relative locations for the postconditions* of the exploitable vulnerabilities and usable information sources are accounted. If the relative location for a postcondition refers to the backend application of the parameter target software application, then the backend software applications used by the target software application are found and the corresponding gained privileges on the backend software applications are created by using the postcondition. By this way, the attacker can gain privileges on a (backend) software application by using another application's vulnerability instead of using a vulnerability of the (backend) software application. Newly gained privileges are pushed to the main search stack used by the search agent, if they have not been already expanded. The pseudo-code for the function FindGainedPrivileges() is shown in Figure 6.10.

```
function FINDGAINEDPRIVILEGES($SP, CP, TSA$)  ▷ SP can be a vulnerability or information source, CP is the current privilege, TSA is the target software application
    $gainedPrivileges \leftarrow EmptyList()$
    $BA \leftarrow RelativeLocation.BackendApplication$   ▷ Account for relative location value of backend application for postconditions
    for all $postCondition \in SP.postConditions()$ do
        if $postCondition.ExistsIn == BA$ then
            for all $backendSoftwareApp \in TSA.backendSoftwareApps()$ do
                $gainedPrivileges.addAll($FormPrivileges($postCondition,$
                    $CP.softwareApp, backendSoftwareApp$))
            end for
        else
            $gainedPrivileges.addAll($FormPrivileges($postCondition, CP.softwareApp, TSA$))
        end if
    end for
    return $gprgs$
end function
```

Fig. 6.10: Finding the privileges gained by an attacker after exploiting a vulnerability or an information source

After finding the privileges gained by the attacker, the expansion status of these privileges are checked, appropriate actions are performed according to the checked status and the status information is updated by calling the function UpdateGainedPrivilegesInMemory(). The pseudo-code for this function is shown in Figure 6.11. The expansion status of each gained privilege stored in the shared memory is checked. If the gained privilege has not already been expanded, its expansion status is updated to true in the shared memory and the gained privilege is pushed into the search stack. The two shared memory operations, namely reading the expansion status from the shared memory and updating it with new status object, are performed atomically by the function ReadAndUpdateSharedMemory(). More than one agent can not enter into this function simultaneously.

```
1: function UpdateGainedPrivilegesInMemory($gainedPrivileges, MainStack, foundPrivileges$)
2:     for all $gainedPrivilege \in gainedPrivileges$ do
3:         $newGainedPrivilegeStatus \leftarrow$ new PrivilegeStatus()
4:         $newGainedPrivilegeStatus.setExpanded(true)$
5:         $oldGainedPrivilegeStatus \leftarrow$ ReadAndUpdateSharedMemory(
6:             $gainedPrivilege, newGainedPrivilegeStatus$)
   ▷ ReadAndUpdateSharedMemory is an atomic operation that updates the status of its input
   privilege and returns its old status.
7:         if $oldGainedPrivilegeStatus.expanded == false$ then
8:             $MainStack.push(gainedPrivilege)$
9:         end if
10:        $foundPrivileges.add(gainedPrivilege)$
11:    end for
12: end function
```

Fig. 6.11: Updating the expansion status of the privileges gained by an attacker in shared memory

Updating Partial Attack Graphs

The partial attack graph computed by a search agent is updated with the newly gained privileges, exploited vulnerabilities and used information sources in UpdateAttackGraph() function during the distributed search algorithm as shown in Figure 6.7. The partial attack graph is updated according to the attack graph structure described in Section 6.2.2 as follows: If a vulnerability is exploited or an information source is used by an attacker, a vulnerability exploit node or an information source usage node is created by using the meta-information (IP address, CPE [15] name, etc.) of the target software application and added to the partial attack graph. If there is one required privilege, given as parameter, then an edge from the required privilege to the vulnerability exploit node or information source usage node is added in the partial attack graph for the search agent. Otherwise, a privilege conjunction is created and added to the partial attack graph. An edge from each of the required privileges to the privilege conjunction node is added in the partial attack graph. Also, an edge from the privilege conjunction node to the vulnerability exploit node or information source usage node is added in the partial attack graph. Lastly, an edge from the created vulnerability exploit node or an information source usage node to each of the privileges is added, which are gained by the attacker as a result of successfully exploiting/using the vulnerability exploit or information source. The pseudo-code for the function UpdateAttackGraph() is shown in Figure 6.12.

```
$partialAttackGraph \leftarrow$ new AttackGraph()          ▷ Global variable across the search agent code
procedure UpdateAttackGraph($SP, REQPS, GPS, TSA$)          ▷ SP can be a vulnerability or information source, REQPS is the required privileges, GPS is the gained privileges, TSA is the target software application
    if $SP$ instanceof $Vulnerability$ then
        $exploitNode \leftarrow$ CreateVulnerabilityExploitNode($SP, TSA$)
    else
        $exploitNode \leftarrow$ CreateInformationSourceUsageNode($SP, TSA$)
    end if
    if $REQPS.size() > 1$ then
        $privilegeConjunction \leftarrow$ new PrivilegeConjunction()
        $partialAttackGraph.addNode(prjc)$
        for all $requiredPrivilege \in REQPS$ do
            $partialAttackGraph.addEdge(requiredPrivilege, privilegeConjunction)$
        end for
        $partialAttackGraph.addEdge(privilegeConjunction, exploitNode)$
    else
        if $REQPS.size() == 1$ then
            $partialAttackGraph.addEdge(REQPS.get(0), exploitNode)$
        end if
    end if
    for all $gainedPrivilege \in GPS$ do
        $partialAttackGraph.addEdge(exploitNode, gainedPrivilege)$
    end for
end procedure
```

Fig. 6.12: Updating the partial attack graph for each search agent after exploitation of a vulnerability or usage of an information source

Merging Partial Attack Graphs

After all search agents finish computing their partial attack graphs, the partial attack graphs are merged by one of the search agents, which is designated as the leader search agent. The other agents send their computed partial attack graphs to the leader agent and the leader agent performs the procedure, shown in Figure 6.13, to merge them and form a single attack graph.

The attack graph merge algorithm starts by merging the same privileges existing in the different partial attack graphs. If a privilege exists in more than one partial attack graph, only one instance of it can contain descendant nodes. Because, we expand each privilege only once during the distributed search algorithm via holding its expansion status in virtual shared memory. Therefore, if there is an instance of a privilege in a partial attack graph that contains descendant nodes, this instance should exist in the resulting attack graph computed by the leader search agent. Other instances of the privilege are removed from the attack graph. The incoming edges of the other instances (existing privileges) are updated to target the instance containing the descendant nodes by calling the UpdateInEdgesOfExistingNode() function. This function updates the target nodes of the incoming edges of its first parameter with the value of the second parameter.

After eliminating the duplicate privileges in the resulting attack graph, the merging algorithm eliminates the duplicate vulnerability exploit and information source usage nodes (exploit nodes) contained by the resulting attack graph. If there exist same instances of an exploit node in the resulting attack graph, their out edges are merged (combined). If we find an exploit node named *exploitNode* during the traversal of the *exploit node list of the resulting attack graph*, for which the same node has already existed, we first find the outgoing neighbour nodes of the already existing exploit node. We add an edge from the *exploitNode* to each outgoing neighbour node. Then, we update the target nodes of the incoming edges of the already existing exploit node with the value of *exploitNode* by calling the UpdateInEdgesOfExistingNode() function. We remove the already existing exploit node from the resulting attack graph. As a summary, we collect the incoming/outgoing neighbours of the same exploit nodes, make one of these exploit nodes refer to all of these incoming/outgoing neighbours and remove the other exploit node from the attack graph.

6.3.4 Complexity Analysis

The complexity of the distributed search algorithm, shown in Figure 6.7, is considered in terms of the message transfer and execution time complexity. The maximum number of the messages transferred among the distributed search agents is dominated by the number of the memory page faults encountered by them, since we try to provide for comparable workload for each search agent during the execution of the search algorithm.

The number of the possible privileges that can be obtained on a software application is constant. If virtual shared memory abstraction is used without any specific memory initialization scheme (e.g. reachability hyper-graph

```
1: function MergePartialAttackGraphs($PGS$) ▷ PGS is the partial attack graphs generated by the search agents
2:     if $PGS.size() == 0$ then
3:         return new AttackGraph()
4:     end if
5:     $attackGraph \leftarrow PGS.removeFirst()$ ▷ Update privileges for the attack graph
6:     $privilegeHashMap \leftarrow$ FormHashMap($attackGraph.privileges()$) ▷ Hash of privileges mapped by their identifiers
7:     for all $partialAttackGraph \in PGS$ do
8:         for all $privilege \in partialAttackGraph.privileges()$ do
9:             $existingPrivilege \leftarrow privilegeHashMap.get(privilege.id())$
10:            if $existingPrivilege != null$ then
11:                if $(privilege.outEdges.size() > 0$ then
12:                    if $existingPrivilege.outEdges.size() == 0)$ then ▷ Remove the existing privilege from the attack graph and add the privilege and its subtree in place of existing privilege
13:                        $attackGraph.addNodeWithItsSubTree(privilege)$
14:                        UpdateInEdgesOfExistingNode($existingPrivilege, privilege$)
15:                        $attackGraph.removeNode(existingPrivilege)$
16:                        $privilegeHashMap.put(privilege.id(), privilege)$
17:                    end if
18:                end if
19:            else
20:                $attackGraph.addNodeWithItsSubTree(privilege)$
21:                $privilegeHashMap.put(privilege.id(), privilege)$
22:            end if
23:        end for
24:    end for
    ▷ Remove duplicate vulnerability exploit and information source usage nodes for the attack graph
25:    $exploitHashMap \leftarrow$ new HashMap()
26:    $exploits \leftarrow attackGraph.vulnerabilityExploits$
27:    $exploits.addAll(attackGraph.informationSourceUsages)$
28:    for all $exploit \in exploits$ do
29:        $existingExploit \leftarrow exploitHashMap.get(exploit.id())$
30:        if $existingExploit != null$ then
31:            for all $outNeighbour \in existingExploit.outNeighbourNodes()$ do
32:                $attackGraph.addEdge(exploit, outNeighbour)$
33:            end for
34:            UpdateInEdgesOfExistingNode($existingExploit, exploit$)
35:            $attackGraph.removeNode(existingExploit)$
36:        end if
37:        $exploitHashMap.put(exploit.id(), exploit)$
38:    end for
39:    return $attackGraph$
40: end function
```

Fig. 6.13: Merging the partial attack graphs by the leader search agent

partitioning), then the maximum number of the memory page faults encountered by the search agents is $O(E)$, where E denotes the number of edges among the directly reachable software applications. This is because, each edge among the directly reachable software applications is traversed maximum C times totally by all the agents during attack graph building, where C denotes the maximum total number of privileges related to a software application and is constant. Actually, in the computation of the attack graph there can be at most $O(E * C)$ privileges directly obtainable from a specific privilege. Since each privilege has its expansion status stored in the virtual shared memory, each of the edges between a privilege and its directly obtainable privileges is traversed exactly once. Memory page transfers can occur, when going from one privilege to each of its directly obtainable privileges during the attack graph building. C is constant, so the maximum number of memory page faults encountered by the search agents is $O(E)$. In the worst-case, this complexity is $O(N^2)$, where N denotes the number of network interfaces/hosts containing software applications.

If virtual shared memory abstraction is used with memory initialization based on reachability hyper-graph partitioning, then the maximum number of memory page faults encountered by the search agents is $O(N * H)$, where H is the number of cluttered hyper-edges (whose contents are distributed into more than one search agent) after reachability hyper-graph partitioning, N denotes the number of network interfaces/hosts. Each privilege is processed only once and can have at most H containing hyper-edges, which can be stored in the memory of other search agents. Memory page transfers can occur, when going from one privilege to each of its directly obtainable privileges which are related to the software applications stored in the cluttered hyper-edges. It should be noted that this complexity reasoning is made under the assumption of comparable workload for each search agent during the execution of the search algorithm.

According to [40], the worst-case number of messages for locating the owner of a single page K times is $O(P + K * log(P))$, where P is the number of processors (the search agents in our case). Therefore, the worst-case complexity of the number of messages transferred among the search agents in our algorithm is $O(P + N * H * log(P))$, where H denotes the number of cluttered hyper-edges obtained after reachability hyper-graph partitioning as described above.

When considering the execution time complexity of the distributed search algorithm, since a virtual shared memory abstraction is applied over the distributed search agents, each edge in the resulting attack graph is traversed only once as in popular serial depth-first or breadth-first search-based algorithms running on one processor, by controlling the memory coherent expansion status for the privileges. The worst-case time complexity of the popular serial depth-first or breadth-first search-based attack graph building algorithms is $O(N^2)$, where N denotes the number of the network interfaces/hosts (when considering constant maximum number of privileges per network interface/host). If we assume that the number of the search agents is P and account for the load balancing as a result of the reachability hyper-graph partitioning algorithm, then the number of the network interfaces/hosts that should be handled by one agent is $O(N/P)$. Therefore, the worst-case execution time complexity of one search agent's execution of the modified depth-first search-based algorithm is $O(N^2/P^2)$.

The execution time complexity of the merging algorithm used to merge the partial attack graphs (generated by the search agents) by the leader search agent and shown in Figure 6.13 is $O(P * N * log(N))$, if we assume that getting a value from a hash map takes $O(log(N))$ time.

As a result, the time complexity of the overall distributed attack graph generation algorithm is:

$$O((N^2/P^2) + P + N * H * log(P) + P * N * log(N)) \tag{6.1}$$

If $P^3 < N/log(N)$, then the last term in Equation 6.1 which is $P * N * log(N)$ is smaller than N^2/P^2. Also, if $H < N/(P^2 log(P))$, then the third term in Equation 6.1 which is $N * H * log(P)$ is smaller than N^2/P^2 and the time complexity of the overall distributed attack graph generation algorithm becomes bounded by $O(N^2/P^2)$, namely the first term in Equation 6.1. It is easy to satisfy the above two *if* conditions for enterprise networks with a large number of network interfaces/hosts. For instance, for a network composed of 1000 network interfaces/hosts (N), we can take P as at most 6 to satisfy the first *if* condition. If we take P as 4, H should be at most 104 to satisfy the second *if* condition. This means that there can be at most 104 cluttered (cut) hyper-edges on the reachability hyper-graph. If we collect the network interfaces/hosts on the same subnet in a single group (in one reachability hyper-edge) as in the previous works such as [32], we will use most probably around 10 hyper-edges for this purpose. It will remain around 90 reachability hyper-edges that can be used to encode the accessibility relations enforced by the firewall rules. If we use one hyper-edge for representing all the outside accessibility relations of a single network interface/host, we will be able to store the outside accessibility relations for around 90 network interfaces/hosts. This number will be more than enough for a realistic network with 1000 network interfaces/hosts. Under these conditions, we can compute an attack graph around 16 (P^2) times faster than the serial, depth-first search-based algorithm in theory.

6.3.5 Experiments

The experiments used to evaluate the performance of the proposed attack graph building mechanism are performed by using the open-source multi-agent platform JIAC V [30], developed at DAI-Labor at TU Berlin. Each search agent in JIAC V multi-agent environment can be considered to execute in its own thread. JIAC V agents communicate via TCP sockets in the employed experimental configuration.

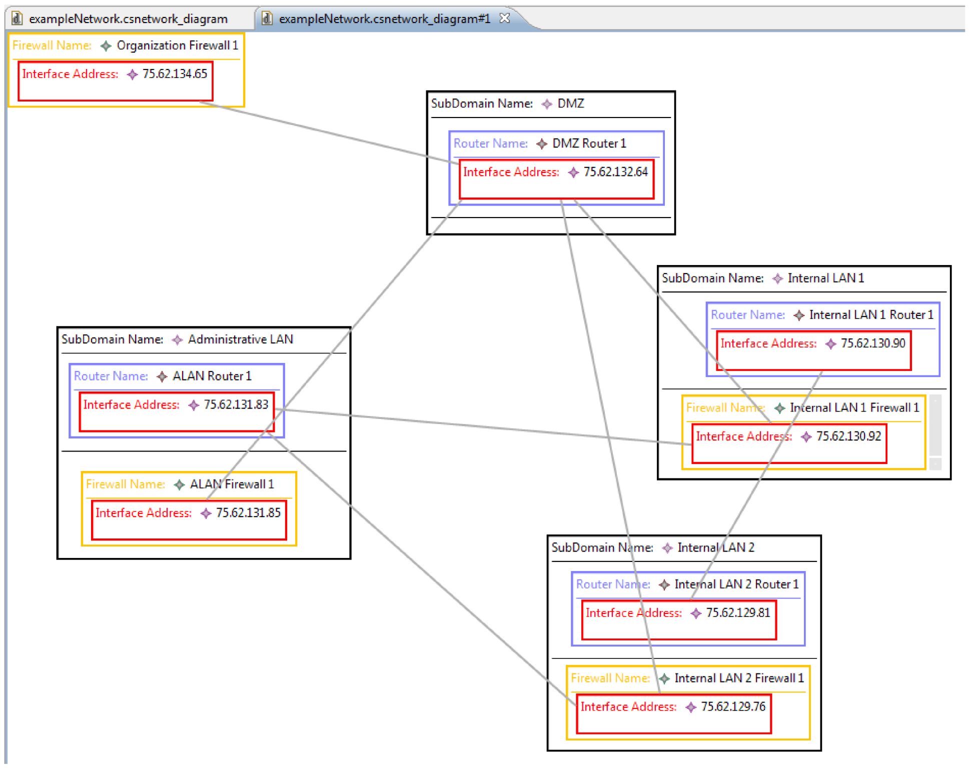
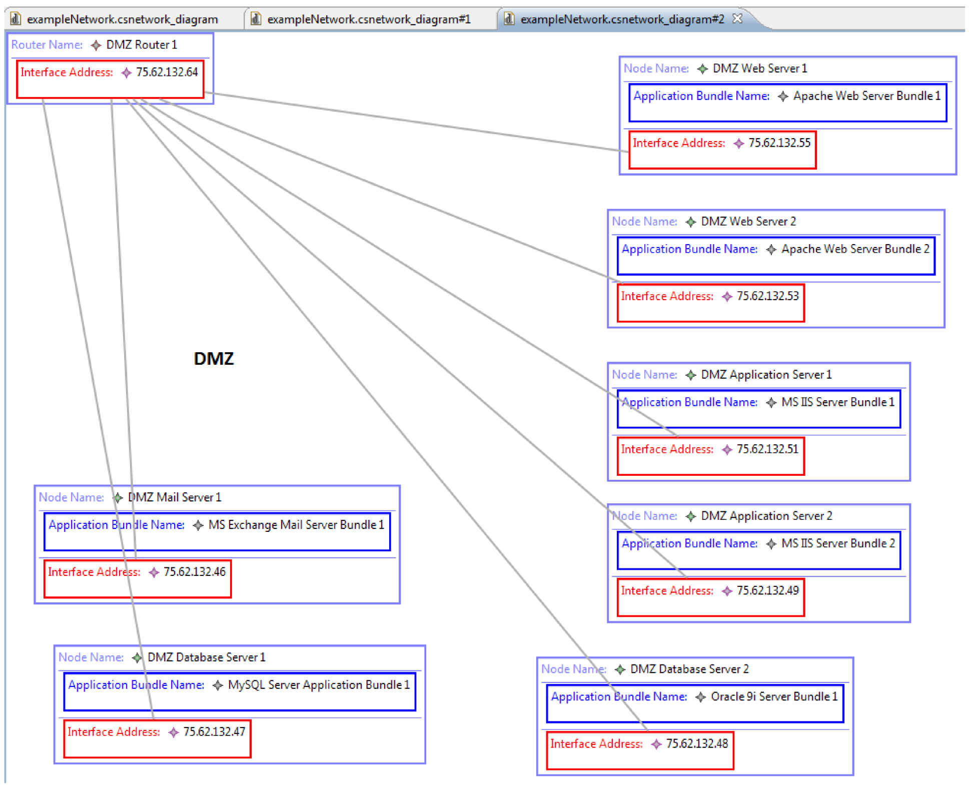

exampleNetwork.csnetwork_diagram
exampleNetwork.csnetwork_diagram#1
Firewall Name: Organization Firewall 1
Interface Address: 75.62.134.65
SubDomain Name: DMZ
Router Name: DMZ Router 1
Interface Address: 75.62.132.64
SubDomain Name: Internal LAN 1
Router Name: Internal LAN 1 Router 1
Interface Address: 75.62.130.90
Firewall Name: Internal LAN 1 Firewall 1
Interface Address: 75.62.130.92
SubDomain Name: Administrative LAN
Router Name: ALAN Router 1
Interface Address: 75.62.131.83
Firewall Name: ALAN Firewall 1
Interface Address: 75.62.131.85
SubDomain Name: Internal LAN 2
Router Name: Internal LAN 2 Router 1
Interface Address: 75.62.129.81
Firewall Name: Internal LAN 2 Firewall 1
Interface Address: 75.62.129.76

Fig. 6.14: An example target network for attack graph generation

The original network (network domain) to be tested (for which the attack graph is to be generated) is given in Figure 6.14. It consists of four subdomains: DMZ, Administrative LAN, Internal LAN 1 and Internal LAN 2.

The elements of the DMZ sub-domain are given in Figure 6.15 as an example.


exampleNetwork.csnetwork_diagram | exampleNetwork.csnetwork_diagram#1 | exampleNetwork.csnetwork_diagram#2

DMZ

Router Name: DMZ Router 1
Interface Address: 75.62.132.64

Node Name: DMZ Web Server 1
Application Bundle Name: Apache Web Server Bundle 1
Interface Address: 75.62.132.55

Node Name: DMZ Web Server 2
Application Bundle Name: Apache Web Server Bundle 2
Interface Address: 75.62.132.53

Node Name: DMZ Application Server 1
Application Bundle Name: MS IIS Server Bundle 1
Interface Address: 75.62.132.51

Node Name: DMZ Application Server 2
Application Bundle Name: MS IIS Server Bundle 2
Interface Address: 75.62.132.49

Node Name: DMZ Mail Server 1
Application Bundle Name: MS Exchange Mail Server Bundle 1
Interface Address: 75.62.132.46

Node Name: DMZ Database Server 1
Application Bundle Name: MySQL Server Application Bundle 1
Interface Address: 75.62.132.47

Node Name: DMZ Database Server 2
Application Bundle Name: Oracle 9i Server Bundle 1
Interface Address: 75.62.132.48

Fig. 6.15: Elements of DMZ sub-domain in the example target network

There is a malicious external domain, consisting of a single sub-domain, connected to the domain of the example network. Each host found in the Internal LANs and Administrative LAN in the target network domain contains the following applications: MS Windows 7 gold, MS Outlook 2007, MS Office 2010, MS Internet Explorer 10. Two of the web servers found in DMZ in the target network domain contain Apache HTTP Server 2.4.3, the other two contain MS IIS Server 6.0. One of the database servers in DMZ contains Oracle9i Database Server 9.0.1.2, the other contains MySQL Database Server 5.1. The mail server in DMZ contains MS Exchange Server 2010. The vulnerabilities of each of these applications (with their pre- and postconditions) are stored in a specific vulnerability database managed in our system.

The aim of these experiments is to compare the performance of the distributed search algorithm, proposed as the core attack graph building mechanism in this thesis work, with the performance of the serial search-based attack graph building algorithms, which has worst-case time complexity $O(N^2)$, where N denotes the number of network interfaces/hosts in the target network. To the best of our knowledge, no attack graph building algorithm proposed so far in the literature attains a worst-case time complexity better than $O(N^2)$. Also, there has been no distributed attack graph building algorithm proposed so far. By increasing the count of the network hosts (except the routers and firewalls) in the Internal LANs in the target network domain and in the external malicious network domain in each experiment iteration, the performance of the serial (depth-first) search-based and the proposed distributed attack graph building algorithm are compared. The proposed distributed attack graph building algorithm is executed with two to four search agents running on a computer with Intel X7550 quad-core processor. The configurations of the firewalls are held constant across the experiment iterations. The running times of both algorithms with respect to the total number of the hosts in the resulting network (the target network domain combined with the external malicious network domain) are shown in Table 6.1.

Tab. 6.1: Running times of serial DFS and the proposed distributed, parallel algorithm

Network Size	Attack Graph Size	Running Time (sec.)			
(Host Count)	(Privilege Count)	Serial DFS	Dist. (2 Agents)	Dist. (3 Agents)	Dist. (4 Agents)
18	352	1.85	5.95	5.93	5.99
27	680	2.68	6.67	6.83	6.85
36	1,008	3.82	7.71	7.30	7.99
54	1,664	5.19	9.41	9.25	10.47
90	2,976	10.72	13.09	12.24	12.79
126	4,288	19.59	21.11	18.25	16.79
162	5,600	32.35	29.73	27.36	25.63
198	6,912	50.49	38.69	36.48	34.65
243	8,552	78.29	62.65	48.85	46.66
288	10,192	124.23	86.32	79.45	68.55
333	11,832	171.35	122.63	105.41	88.68
387	13,438	231.86	167.87	122.28	100.22
441	16,754	295.57	200.52	151.82	121.71
495	19,936	385.36	261.82	198.21	145.25

In Table 6.1, the attacker privilege count in the resulting attack graph for each target network is also shown. An attacker privilege is a type of attack graph node and represents a specific right that is gained by an attacker on a software application on a network host as described in Section 6.2.2.

Therefore, the number of the attacker privilege nodes in the resulting attack graphs represent their size roughly. Since attack graph building algorithms are mostly based on the traversal of the privileges and aim to eliminate the duplicate expansion of the privileges as much as possible, the number of the privilege nodes in the resulting attack graphs also serves as an indication of the workload that must be performed by the attack graph building algorithms. By indicating the workload for each experiment, we compare the performance of the serial algorithm with our distributed algorithm in the face of increasing workload. In each of these experiments, there is only one initial attacker privilege that is supposed to be gained on one of the malicious nodes. This initial privilege triggers the attack graph building process. The page size for the virtual shared memory is set so that each page can contain all the privileges associated with 16 network hosts. Additionally, for each target network, the average number of the vulnerabilities per network host is almost 10.

When the benefit (time decrease) gained from parallelism (increasing the number of search agents executing the attack graph building process) exceeds the time increase caused by the message transfers among the agents, the proposed distributed attack graph building algorithm becomes desirable. For instance, starting from the experiment where the target network has 162 hosts (resulting attack graph has 5600 privileges), the proposed algorithm running with the 2-agent configuration gives better performance in terms of time with respect to the serial, graph search-based attack graph building algorithm. Similar condition is true for the execution of the proposed algorithm with the 3- and 4-agent configurations starting from the experiment where the target network has 126 hosts. The performance gain by the distributed computation of attack graphs goes up to 40 percent or more using only 4 agents for networks of at least 250 hosts. Figure 6.16 further depicts the performance gains obtained by the distributed algorithm in terms of the execution time.

We can also conclude from the experiment results that while the number of the network hosts increases, the execution time for the 4-agent configuration becomes much better than the 3-agent configuration and the execution time for the 3-agent configuration becomes much better than the 2-agent configuration. Namely, as the number of the network hosts increases, the performance gain obtained with a specific number of agents compared to the performance gain obtained with less number of agents increases. The benefit of the distributed algorithm becomes more obvious.

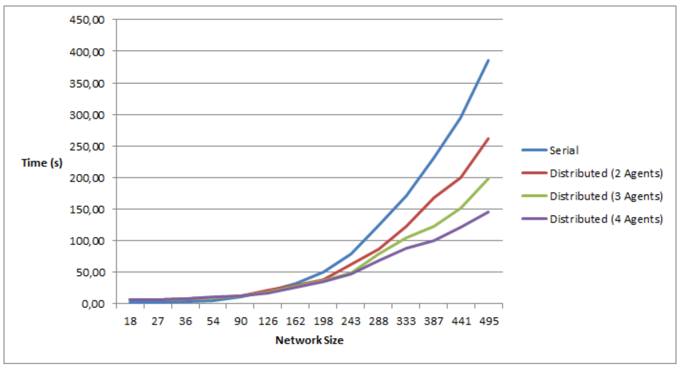


Fig. 6.16: Performance comparison of the serial and proposed distributed attack graph building algorithms in terms of execution time

7 Generation of Behavioural Malware Signatures

The increased importance of the utilization of as much data as possible generated by the software running throughout a network in assessing the security situation of the network gives rise to the proliferation of developments in Security Event and Information Management (SIEM) tools. The first SIEM tools using only manually generated rules to correlate software logs for detecting the malicious activities occurring inside the network is being updated to contain advanced detection capabilities that are anomaly and/or signature-based. The trend in developing signature-based detection techniques is towards using automatically generated behavioural malware signatures instead of content-based (byte sequence-based) ones. This supports the detection of zero-day malware which can be essentially packed, polymorphic or metamorphic variants of existing malware and perform semantically equivalent with respect to the existing malware, but can have totally different syntactic structures.

A behavioural signature characterizes the effects of a family of malware on a target operating system (OS) in terms of the OS operations performed by the malware on specific OS objects. The examples of these OS objects are the files, registry keys/values, processes, services, threads, memory areas, mutexes and network sockets. The OS operations performed by a malware sample file and the affected OS objects can be extracted by executing the file in a dynamic software (or malware) analysis tool. Such a tool basically executes the file in a virtual machine (sandbox) environment, collects the system (or API) calls performed by the file and optionally abstracts the system calls into high level behavioural artifacts in order to generate a concise, system-independent behavioural profile for the file. The behavioural profiles for the malware sample files are clustered to generate behavioural signatures, after converting them into a format that can be processed by specific clustering algorithms. For each cluster, a behavioural signature is generated. Namely, the outputs of the dynamic software analysis tools serve as the main inputs for generating malware behavioural signatures.

A behavioural signature can also be generated without executing the malware sample files in a sandbox. For this purpose, static software analysis methods can be utilized. Control and data flow diagrams generated after disassembling a malware sample file can be used with the import address tables to generate a system (API) call graph for the file. The call graphs for different malware files are compared by using a specific similarity measure and the comparison results are used to cluster them. The signatures can be created in the form of a graph of system calls for each cluster.

Anti-virus vendors receive hundreds of thousands of files per day, many of which can be indeed malicious (the variants of existing malware). The possibly huge amount of malicious files appeared each day necessitates the development and usage of a scalable and incremental clustering and signature generation method for them. Additionally, the generated behavioural signatures can be used in the context of security information and event management. For this purpose, the format of the generated signatures should allow the implementation of matching among the signatures and software logs easily. For instance, to represent a signature with a graph structure holding conditions on its edges that specify the to be satisfied relationships among the fields of the corresponding graph nodes can introduce the need for complex and resource-intensive algorithms to match the logs into the signatures. In addition to the conditions among the fields of a single node, the conditions on the edges should also be satisfied in this case.

The generation of the behavioural signatures by processing sample malicious files requires the determination of:

- A method for the generation of the behavioural artifacts (system calls, call graphs, high-level artifacts representing the OS state changes, etc.) of a malicious file,
- A similarity (distance) computation method to measure the similarity (distance) between the artifacts of a pair of malicious files,
- A clustering (unsupervised) or classification (supervised) method to cluster or classify the malicious files by using the computed similarity

(distance) values,

- A method for generating signature(s) for each of the resulting clusters or classes,
- A method for labeling the generated signatures (determining the family labels).

The main contribution of this thesis work in this context is to propose a method for clustering malware files that is incremental and has linear average time complexity. The method also generates a behavioural signature for each malware cluster in a form that can easily be used in SIEM tools for detecting the malicious system activities. These activities can be detected by computing the matchings among the collected system/application logs and the generated behavioural signatures. There is no condition defined on any component of the signature that specifies a relationship among the values of components' fields. The proposed method for signature generation uses the system (or API) calls generated via dynamic analysis of each sample malicious file as input. The system calls performed by a file is divided into groups according to their generic operation (read, write, create, etc.) and OS object (registry key, file, memory area, etc.) type. The computation of the similarity between the system calls of a pair of files is performed via application of a modified form of Jaccard distance (weighted Jaccard distance) between the corresponding system call groups. The proposed clustering algorithm is tree-based and the signatures are computed by a simple depth-first traversal of the generated tree.

The malware behavioural signature generation system proposed in this thesis work introduces an incremental, tree-based, scalable clustering algorithm to group malware having similar behavioural artifacts. For each of the resulting clusters, a signature that can be easily used to match against the software logs collected by a SIEM tool is generated. The workflow for the system is shown in Figure 7.1.

The system takes the system (or API) calls performed by a malware sample as input. The input can be generated by executing the malware sample in any virtual sandbox (or dynamic malware analyser) like Cuckoo sandbox [16] or Anubis [39]. (We use the behavioural reports downloaded from the web site *https://malwr.com* for sample malware.) The system calls are processed to generate the behavioural artifacts for the sample. A behavioural artifact consists of operating system (OS) operations of a specific type performed on OS objects of a specific type. There are different types of operations

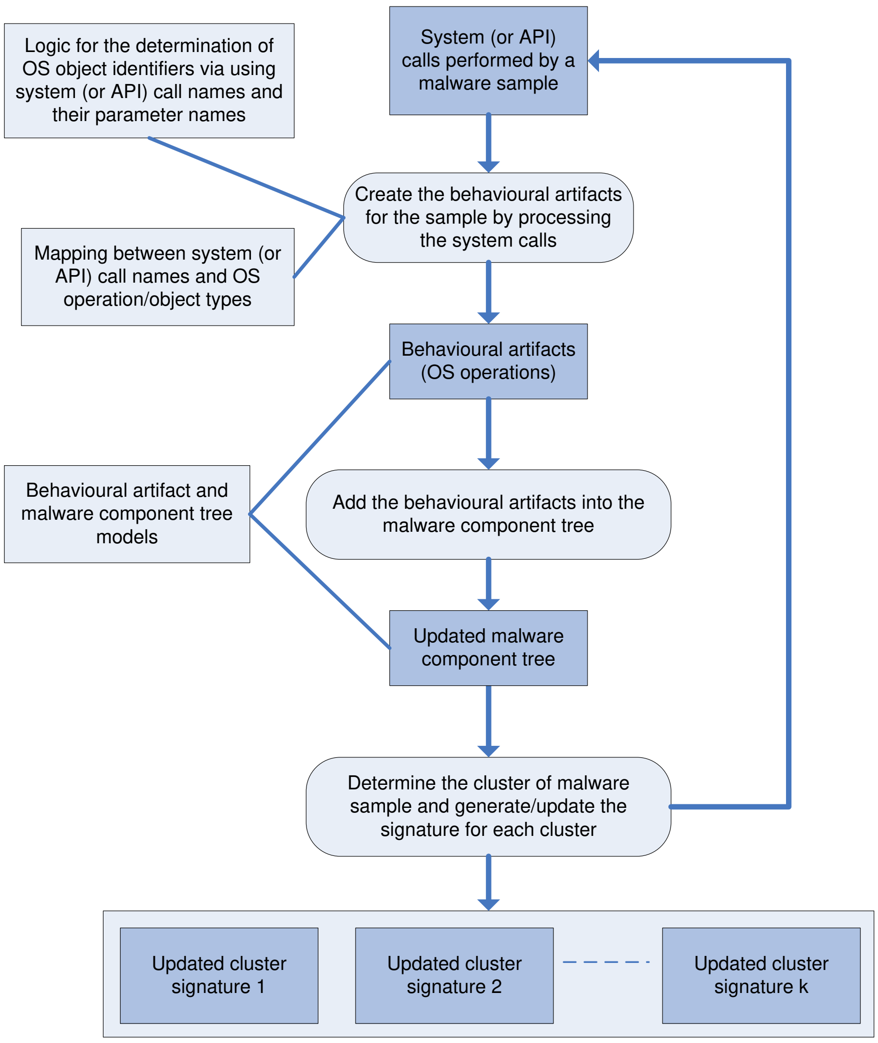

Logic for the determination of OS object identifiers via using system (or API) call names and their parameter names
System (or API) calls performed by a malware sample
Create the behavioural artifacts for the sample by processing the system calls
Mapping between system (or API) call names and OS operation/object types
Behavioural artifacts (OS operations)
Behavioural artifact and malware component tree models
Add the behavioural artifacts into the malware component tree
Updated malware component tree
Determine the cluster of malware sample and generate/update the signature for each cluster
Updated cluster signature 1
Updated cluster signature 2
Updated cluster signature k

Fig. 7.1: Workflow for the malware behavioural signature generation system

like read, write, create, delete, execute and terminate operations that can be performed on different types of OS objects like registry keys, files (including directories), section objects (memory areas), processes, services, network sockets and synchronization objects such as semaphores. Each malware sample has the following behavioural artifacts: registry entry read, registry entry write, registry entry create, directory read, directory write, directory create, directory delete, file execute, memory area read, memory area write and network socket write. Not all OS operation type-object type combinations are used in the proposed system, rather some combinations are merged (e.g., process create and process execute are merged into file execute) and some are eliminated to increase performance and decrease the effect of name variations (e.g., operations for synchronization objects which mostly have differing names across malware variants), some are not used, since they are not meaningful (e.g., registry key execute). Also, the behavioural artifact *file execute* is associated with the execution of a file in a specific directory, where the name of the file is not important.

The behavioural artifacts of the malware samples are organized in a tree structure which is called the malware component tree. The addition of a behavioural artifact to the tree can include the merging of the artifact with another artifact previously added into the tree, or the insertion of the artifact as a separate node. During the addition, the artifact is compared to the existing artifacts on the tree which have the same OS operation and object type. After all the behavioural artifacts of a malware sample are added to the malware component tree, its cluster is determined and the behavioural signatures for the affected clusters are updated. The following subsections details the components of the workflow illustrated in Figure 7.1. Section 7.1 describes the behavioural artifact and malware component tree models. The generation of the behavioural artifacts from the system (or API) calls performed by a malware sample is discussed in Section 7.2. The generation of the malware component tree during incremental clustering of the malware samples is described in Section 7.3. Section 7.4 describes the generation of the behavioural signatures for the malware clusters based on a depth-first traversal on the malware component tree. In Section 7.5, the experiments performed to measure the performance and accuracy of the proposed malware behavioural signature generation system are described.

7.1 Behavioural Artifact and Malware Component Tree Models

This section comprises the description of the formal data models of the objects used to represent the behavioural artifacts and malware component tree. In this context, two models are described. The first model is the behavioural artifact model and the second one is the malware component tree model. The description of the behavioural artifact model includes the formal definitions of an OS object, OS object occurrence, OS operation and a behavioural artifact as given below:

Definition 7.1.1. An OS object is a two-element tuple $\langle Type, Identifier \rangle$ indicating a data structure implemented by an operating system and used by software to facilitate their activities such as file-system, process and network management. The $Type$ field denotes whether the OS object is a registry key, file (directory), memory area used by a process or network socket. The $Identifier$ field denotes the unique identifier for the OS object. Its value is determined according to the type of the OS object. If the type of the OS object is *registry key*, the $Identifier$ field is set to the name of the registry key indicated by the OS object. If the type is *file*, the $Identifier$ field is set to the absolute location of the directory of the actual file indicated by the OS object. If the type is *network socket*, the $Identifier$ field is set to the concatenation of the protocol and destination port of the network socket related to the OS object.

Definition 7.1.2. An OS object occurrence is a two-element tuple $\langle osObject, occurrenceCount \rangle$ defining the count of occurrences of an OS object in any related context. The $osObject$ field denotes an OS object, the $occurrenceCount$ field indicates the occurrence count for the OS object in the defined context.

Definition 7.1.3. An OS operation is a constant denoting the type of the operation that is performed on a set of OS objects. Its possible values are *read, write, create, delete, execute.*

Definition 7.1.4. A behavioural artifact is a three-element tuple $\langle osObjectType, osOperation, osObjectOccurrences \rangle$ indicating a concept which groups a number of instances of the same OS operation on a set of OS objects of the same type. The $osObjectType$ field denotes the type of all the OS objects operated upon by the corresponding operation indicated by the *osOperation* field. *osObjectOccuurrences* is a list of OS object occurrence objects each holding an OS object with a different identifier. Example behavioural artifacts are registry entry read, registry entry write, directory (file) read, directory write, memory area read, memory area write and network socket write. For instance, the behavioural artifact *registry entry read* of a malware sample contains the identifiers of the registry keys read by the sample.

The description the malware component tree model includes the formal definitions of a malware component and malware component tree as given below:

Definition 7.1.5. A malware component is a three-element tuple $\langle behaviouralArtifact, childComponents, malwareSampleNames\rangle$ that encapsulates a behavioural artifact in the context of a tree structure. The $behaviouralArtifact$ field represents the corresponding behavioural artifact and the $childComponents$ field contains references to the child malware components of this malware component. The $malwareSampleNames$ field stores the names of the malware sample files contributing to the formation of the behavioural artifact of this malware component.

Definition 7.1.6. A malware component tree is a tree defined by a three-element tuple $\langle nodes, edges, rootNode\rangle$, where $nodes$ is the set of nodes and $edges$ is the set of edges of the tree. A node of the tree is a malware component and an edge of the tree is a two-element tuple $\langle sourceNode, targetNode\rangle$, where $sourceNode$ indicates the source malware component and $targetNode$ indicates the target malware component for the edge. The root malware component of a malware component tree is indicated by the $rootNode$ field. The $behaviouralArtifact$ field of the root malware component is set to null.

Only one malware component tree is generated during the clustering of the malware samples. A layer of this tree is considered to be composed of the malware component nodes having the same depth value (starting from 0). The tree is built in such a way that requires the placement of the behavioural artifacts in a layer that have the same OS operation and object type. The tree has 11 layers corresponding to the following types of behavioural artifacts: registry entry read, registry entry write, registry entry create, directory read, directory write, directory create, directory delete, file execute, memory area read, memory area write and network socket write. In first layer of the malware component tree, there are *registry entry read* behavioural artifacts, in the second layer there are *registry entry write* behavioural artifacts, and so on. In the last layer of the tree, there are *network socket write* behavioural artifacts. An example malware component tree is shown in Figure 7.2.

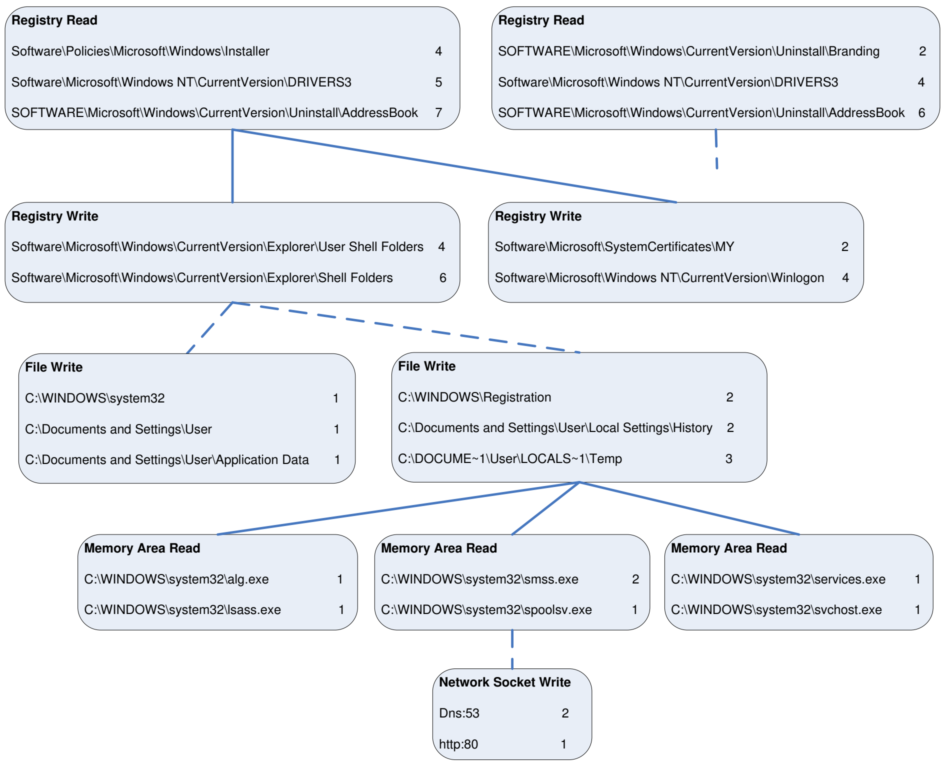

Registry Read
Software\Policies\Microsoft\Windows\Installer 4
Software\Microsoft\Windows NT\CurrentVersion\DRIVERS3 5
SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\AddressBook 7
Registry Read
SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\Branding 2
Software\Microsoft\Windows NT\CurrentVersion\DRIVERS3 4
SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\AddressBook 6
Registry Write
Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders 4
Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders 6
Registry Write
Software\Microsoft\SystemCertificates\MY 2
Software\Microsoft\Windows NT\CurrentVersion\Winlogon 4
File Write
C:\WINDOWS\system32 1
C:\Documents and Settings\User 1
C:\Documents and Settings\User\Application Data 1
File Write
C:\WINDOWS\Registration 2
C:\Documents and Settings\User\Local Settings\History 2
C:\DOCUME~1\User\LOCALS~1\Temp 3
Memory Area Read
C:\WINDOWS\system32\alg.exe 1
C:\WINDOWS\system32\lsass.exe 1
Memory Area Read
C:\WINDOWS\system32\smss.exe 2
C:\WINDOWS\system32\spoolsv.exe 1
Memory Area Read
C:\WINDOWS\system32\services.exe 1
C:\WINDOWS\system32\svchost.exe 1
Network Socket Write
Dns:53 2
http:80 1

Fig. 7.2: An example malware component tree

7.2 Generation of Behavioural Artifacts

The behavioural artifacts for a malware sample are generated by processing the system (or API) calls performed by the sample. Firstly, the following 11 behavioural artifacts that have no content (no OS object occurrences) are created for the sample: registry entry read, registry entry write, registry entry create, directory read, directory write, directory create, directory delete, file execute, memory area read, memory area write and network socket write. During the processing of the system calls, the OS object occurrences are generated. According to the results of a keyword-based searching on the name of a system call, the behavioural artifact to which this system call can contribute is found. For instance, if the name of a system call contains both the keywords *Reg* and *Read*, then it is decided that this system call specifies an OS object for the *registry read* behavioural artifact. The keywords to be searched are determined according to the format of the system (or API) calls for the supported operating systems.

After finding the behavioural artifact to be contributed to by the current system call, which parameter of this system call specifies an identifier for an OS object is determined. This determination is also performed in a hard-coded way according to the the format of the system (or API) calls for the operating systems supported by the proposed system. A behavioural artifact should not contain multiple OS object occurrences containing the same OS object. To ensure this, the OS object identifiers are stored and checked for duplicates during the processing of the system calls for a sample. The occurrence count for each of the created OS object occurrences for each behavioural artifact is set to 1. This indicates that only one malware sample performs the operation indicated by the behavioural artifact on the OS object contained by an OS object occurrence.

Another point to note is that for file operations, only the absolute path of the directory containing the related file is used as the corresponding OS object identifier. For registry operations, the user identifiers are found and eliminated from the registry key names. Additionally, operating systems mostly generate handle identifiers for objects such as files, threads and processes, when creating them via specific system calls and utilize these identifiers instead of names or locations of the objects in the subsequent system calls operating on these objects. Therefore, the proposed system stores a mapping between the handle identifiers and the object names or locations during the processing of the system calls. When a handle identifier is referenced, the system finds the corresponding object name or location and determines an OS object identifier from this data.

7.3 Incremental Clustering of Malware

In this thesis work, a tree-based, incremental malware clustering algorithm is proposed. The algorithm uses the generated behavioural artifacts for the input malware samples and has linear time complexity on the number of the input malware samples. The clustering is performed via the construction and update of the single malware component tree. The behavioural artifacts for each malware sample are used to update the nodes of the malware component tree. At the beginning, the tree has only one malware component which is the root node. When the system (or API) calls for a malware sample are input to the system, the behavioural artifacts for the sample are generated. Then, the generated artifacts are added to the malware component tree in the form of malware components by using the algorithm shown in Figure 7.3.

The addition of the behavioural artifacts of a malware sample to the malware component tree is conducted via the $AddBehaviouralArtifacts()$ procedure as follows: At the beginning of the procedure, the root node of the tree is the current node ($currentMalwareComponent$). The first behavioural artifact to be added is compared against all the existing behavioural artifacts in the first layer of the tree. The existing artifact which has the maximum similarity to the first behavioural artifact is tried to be found. If there is no existing artifact in the first layer or the maximum similarity between the existing artifacts and the first behavioural artifact is smaller than a pre-specified threshold value ($MalwareComponentSimilarityThreshold$), then the first behavioural artifact is added as a child of the root (current) node of the tree and is made the current node. Otherwise, a common behavioural artifact ($maxCommonArtifact$) is formed by merging the first behavioural artifact of the malware sample and the existing artifact in the first layer of the tree which is most similar to it. The malware component ($maxSimilarChildNode$) containing this existing artifact is updated with the common behavioural artifact ($maxCommonArtifact$) and is made the current node. The procedure then proceeds with the second behavioural artifact of the malware sample and so on.

Figure 7.4 illustrates the addition of the behavioural artifacts of a malware sample to the malware component tree. A behavioural artifact of the malware

```
$MalwareComponentSimilarityThreshold \leftarrow 0.8$
procedure AddBehaviouralArtifacts($BehaviouralArtifacts, MalwareSampleName,$
$MalwareComponentTree$) ▷ Inputs are the behavioural artifacts for a malware sample, the name of the malware sample and the malware component tree.
    $currentMalwareComponent \leftarrow MalwareComponentTree.rootNode$
    for all $behaviouralArtifact \in BehaviouralArtifacts$ do
        $newMalwareComponent \leftarrow newMalwareComponent()$
        $newMalwareComponent.behaviouralArtifact \leftarrow behaviouralArtifact$
        $newMalwareComponent.malwareSampleNames.add(MalwareSampleName)$
        $addedToCurrentNodeAsChild \leftarrow false$
        $maxSimilarityValue \leftarrow MalwareComponentSimilarityThreshold$
        $maxSimilarChildNode \leftarrow null$
        $maxCommonArtifact \leftarrow null$
        for all $childMalwareComponent \in currentMalwareComponent.childComponents$ do
            $\langle commonArtifact, similarityValue \rangle \leftarrow$
                ComputeSimilarityBetweenMalwareComponents($childMalwareComponent,$
                $newMalwareComponent$)
            if $similarityValue \geq maxSimilarityValue$ then
                $addedToCurrentNodeAsChild \leftarrow true$
                $maxSimilarityValue \leftarrow similarityValue$
                $maxSimilarChildNode \leftarrow childMalwareComponent$
                $maxCommonArtifact \leftarrow commonArtifact$
            end if
        end for
        if $addedToCurrentNodeAsChild = true$ then
            $maxSimilarChildNode.behaviouralArtifact \leftarrow maxCommonArtifact$
            $maxSimilarChildNode.malwareSampleNames.add(MalwareSampleName)$
            $currentMalwareComponent \leftarrow maxSimilarChildNode$
        else
            $currentMalwareComponent.childComponents.add(newMalwareComponent)$
            $currentMalwareComponent \leftarrow newMalwareComponent$
        end if
    end for
end procedure
```

Fig. 7.3: The algorithm for the addition of the behavioural artifacts of a malware sample to the malware component tree

sample is merged with an existing artifact x on the corresponding layer of the tree which is most similar to it, if the similarity between the artifacts is greater than the threshold $MalwareComponentSimilarityThreshold$. If the merging takes place, the artifacts on the next layer which will be compared to the next artifact of the malware sample are selected as the artifacts of the children of the malware component holding the artifact x. This is shown by straight vertical lines that are not dashed and have no label.

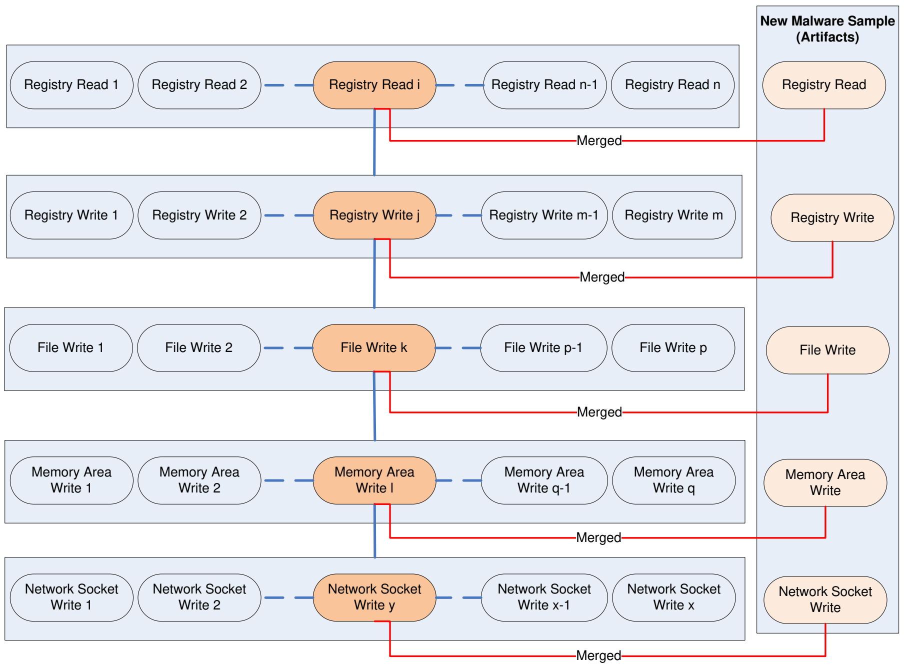


Fig. 7.4: The illustration of the addition of the behavioural artifacts of a sample malware to the malware component tree

The computation of the similarity between two malware components, so to say between their contained behavioural artifacts, is performed by the $ComputeSimilarityBetweenMalwareComponents()$ function. This function also computes the common behavioural artifact for the behavioural artifacts contained by the input malware components. The common behavioural artifact contains the union of the OS objects contained by the two input behavioural artifacts. An OS object in the common behavioural artifact is encapsulated in an OS object occurrence object whose occurrence count value is determined by summing the occurrence count values of the corresponding OS object occurrences in the input behavioural artifacts. The pseudo-code for the $ComputeSimilarityBetweenMalwareComponents()$ function is shown in Figure 7.5.

```
function ComputeSimilarityBetweenMalwareComponents(
$MalwareComponent1, MalwareComponent2$)    ▷ Malware components to be compared are the inputs.
    $behaviouralArtifact1 \leftarrow MalwareComponent1.behaviouralArtifact$
    $behaviouralArtifact2 \leftarrow MalwareComponent2.behaviouralArtifact$
    $commonBehaviouralArtifact \leftarrow newBehaviouralArtifact()$
    $commonBehaviouralArtifact.osObjectType \leftarrow behaviouralArtifact1.osObjectType$
    $commonBehaviouralArtifact.osOperation \leftarrow behaviouralArtifact1.osOperation$
    $intersectionCount \leftarrow 0$
    $totalCount \leftarrow 0$
    $intersectingOSObjectOccurrences \leftarrow newList()$
    for all $osObjectOccurrence1 \in behaviouralArtifact1.osObjectOccurrences$ do
        $osObjectId1 \leftarrow$toLowerCase($osObjectOccurrence1.osObject.Identifier$)
        $intersectedOSObjectOccurrence \leftarrow null$
        for all $osObjectOccurrence2 \in behaviouralArtifact2.osObjectOccurrences$ do
            $osObjectId2 \leftarrow$toLowerCase($osObjectOccurrence2.osObject.Identifier$)
            if $osObjectId1 = osObjectId2$ then
                $intersectingOSObjectOccurrences.add(osObjectOccurrence2)$
                $intersectedOSObjectOccurrence \leftarrow osObjectOccurrence2$
            end if
        end for
        $copyOsObjectOccurrence1 \leftarrow copy(osObjectOccurrence1)$
        if $intersectedOSObjectOccurrence \neq null$ then
            $intersectionCount{+}{=}copyOsObjectOccurrence1.occurrenceCount$
            $intersectionCount{+}{=}intersectedOSObjectOccurrence.occurrenceCount$
            $copyOsObjectOccurrence1.occurrenceCount{+}{=}$
                $intersectedOSObjectOccurrence.occurrenceCount$
        end if
        $totalCount{+}{=}copyOsObjectOccurrence1.occurrenceCount$
        $commonBehaviouralArtifact.osObjectOccurrences.add(copyOsObjectOccurrence1)$
    end for
    for all $osObjectOccurrence2 \in behaviouralArtifact2.osObjectOccurrences$ do
        if not $intersectingOSObjectOccurrences.contains(osObjectOccurrence2)$ then
            $copyOsObjectOccurrence2 \leftarrow copy(osObjectOccurrence2)$
            $commonBehaviouralArtifact.osObjectOccurrences.add(copyOsObjectOccurrence2)$
            $totalCount{+}{=}copyOsObjectOccurrence2.occurrenceCount$
        end if
    end for
    $similarityValue \leftarrow intersectionCount/totalCount$
    return $\langle commonBehaviouralArtifact, similarityValue \rangle$
end function
```

Fig. 7.5: The computation of the similarity between the behavioural artifacts contained by two malware components

The $ComputeSimilarityBetweenMalwareComponents()$ function computes the similarity between the behavioural artifacts by summing the occurrence count of the OS objects contained by both artifacts and then dividing this value to the total occurrence count value for both artifacts. This similarity computation would be similar to the Jaccard distance computation, if all the occurrence count values were kept as 1.

Each path of the malware component tree forms a cluster. The malware sample names stored on the leaf node for each path indicate the malware samples belonging to the cluster corresponding to this path. The paths (the clusters) of the tree can be found by a simple depth-first traversal on the tree.

During the insertion of a behavioural artifact x into the malware component tree, the $AddBehaviouralArtifacts()$ procedure compares the artifact x to F existing artifacts. If we set the value of the threshold $MalwareComponentSimilarityThreshold$ to 1.0, then F denotes the number of different behavioural artifacts which are contained by the malware samples and have the same OS operation and object type as x. This number is actually far smaller than the number of malware samples inserted into the tree. It is bounded by the number of clusters (malware families) generated by the procedure. If we decrease the value of the threshold $MalwareComponentSimilarityThreshold$, the value of F also decreases. When we increase the accuracy of the proposed clustering according to a reference clustering (that can be obtained by majority voting across the decisions/labellings of different anti-virus products) by adjusting the threshold $MalwareComponentSimilarityThreshold$, the value of F becomes closer to the number of clusters generated by the reference clustering. Therefore, the time complexity of the insertion of the behavioural artifacts of all the malware samples into the tree (namely, the clustering of all the malware samples) becomes closer to $O(NF)$, where N is the number of all malware samples. This complexity is much less than $O(N^2)$. The important question is how much closer the value of F can become to the number of clusters (different malware families) generated by the reference clustering. We try to answer this question in Section 7.5 by computing the accuracy and performance of the proposed method in different experimental settings.

7.4 Generation of Behavioural Signatures

A behavioural signature is generated for each cluster represented by a path on the malware component tree. The behavioural signature for a cluster is simply composed of all the malware component nodes on the path corresponding to the cluster. The signatures are generated by applying a simple depth-first search on the malware component tree. The generated signatures can be easily used by a SIEM tool in matching the collected software logs against the behavioural artifacts contained by the components of the signatures. A log can be matched to an OS object occurrence in a behavioural artifact by checking the OS object identifier of the occurrence and OS operation type of the artifact against the corresponding information in the log. In this matching process, the occurrence counts in the matched OS object occurrence objects should also be taken into account, since they indicate the count of malware which operate upon the OS objects contained by the OS object occurrence objects. If a log of the software can be matched to an OS object occurrence with a higher occurrence count value, the probability of this software being malicious is higher. If a number of logs can be matched to a portion of the OS object occurrences in the behavioural artifacts contained by a signature, then the signature can be instantiated. The lower bound for the value of this portion can be pre-specified.

7.5 Experiments

There are two sets of experiments performed to measure the accuracy and efficiency of the malware behavioural clustering system proposed in this thesis work. All of the experiments are conducted on a single server with Intel Xeon E5520 quad-core 2.26 GHz processor and 16GB memory. The behavioural reports for almost 3000 malware sample files collected from the web site *https://malwr.com* form the basis for both sets of experiments. They are all Windows operating system malware having various file extensions (.pdf, .exe, .doc, .xls, etc.). The malware families related to these files are found by feeding the MD5 hashes of them to the VirusTotal web site [80]. These files are related to a wide range of malware families defined by several off-the-shelf anti-virus products.

The first set of experiments measures the accuracy of the proposed malware clustering system with respect to the selected 12 off-the-shelf anti-virus products. The collected malware sample files are used for this purpose. The malware family labels assigned to each file by each of the 12 off-the-shelf anti-virus products are found by referring to the VirusTotal web site. If more than half of these 12 anti-virus products label a file as clean, the file is removed from the list of collected sample files. At the end, 237 malware files remain in the sample files list. These 237 files are used to measure the accuracy of the proposed clustering system.

A reference clustering is needed for the 237 malware files to compute the accuracy of the proposed clustering system. The accuracy of the proposed clustering system is computed by calculating the precision and recall value for each experimental setting. The precision and recall values are calculated as in [39]. We calculate a precision value P_j for each cluster C_j (j starting from 1) generated by the proposed system as follows:

$$P_j = \max_{1 \leq i \leq t} |C_j \cap T_i| \tag{7.1}$$

where t is the total number of reference clusters, T_j is the j^th reference cluster. The overall precision value P is calculated as follows:

$$P = (\sum_{1 \leq i \leq m} P_i)/n \tag{7.2}$$

where m is the total number of the malware clusters generated by the proposed system and n is the total number of the malware files clustered (237 in our case). We calculate a recall value R_j for each reference cluster j (starting from 1) as follows:

$$R_j = \max_{1 \leq i \leq m} |C_i \cap T_j| \tag{7.3}$$

The overall recall value R is calculated as follows:

$$R = (\sum_{1 \leq i \leq t} R_i)/n \tag{7.4}$$

It is not needed to determine descriptive labels (such as Trojan.Zeus, Adware, etc.) for the reference clusters to compute the overall precision and recall values for the proposed system.

The reference clustering for the selected 237 malware files is created by accounting for the majority voting method on the decisions of the selected 12 anti-virus products on the similarity of each pair of malware files. Two malware files are supposed to be similar, if more than half of the 12 anti-virus products decide that they should be in the same cluster. The decision of an anti-virus product for a pair of malware files is determined by comparing the labels assigned to the files by the anti-virus product. If the labels match after removing the malware variant indicators (for instance, in the label Trojan.Zbot.a, a is supposed to be the variant indicator) that are generally found at the end of the labels, then the two files should be in the same cluster according to this anti-virus product. The removal of the malware variant indicators is performed differently for the labelling schemes used by different anti-virus products.

The reference clustering is formed as follows: First, each malware file is put into a separate reference cluster. Then, the malware files are processed pairwise. If for a pair of malware files, more than half of the selected anti-virus products decide that the files should be in the same cluster, then the reference cluster for the second file is changed as the reference cluster for the first file in the pair. At the end of the processing of all the file pairs, if any reference cluster is found to contain no files, it is deleted. For the 237 malware files used in the experiments, the reference clustering contains 119 clusters.

The only parameter to the proposed clustering system is the threshold $MalwareComponentSimilarityThreshold$ which determines the minimum similarity value between two behavioural artifacts to merge them into a single behavioural artifact during the addition of the behavioural artifacts into the malware component tree. The details about the addition of the behavioural artifacts of a malware file into the malware component tree are given in Section 7.3. The definition of the threshold $MalwareComponentSimilarityThreshold$ is shown in the algorithm in Figure 7.3. We compute the overall precision and recall values for the proposed clustering system for different values of the threshold $MalwareComponentSimilarityThreshold$. Table 7.1 shows the results of this computation.

The overall precision and recall values for each of the selected 12 anti-virus products are also calculated. The maximum, average and minimum precision values for all of these anti-virus products are 0.85, 0.72 and 0.53. The maximum, average and minimum recall values for all of these anti-virus products are 0.79, 0.53 and 0.20. When we examine the results for the proposed clustering system in Table 7.1, we realize that the precision value for the proposed system remains over the average precision value for the anti-virus products for the values of the similarity threshold ($MalwareComponentSimilarityThreshold$) greater than approximately

Tab. 7.1: Overall precision and recall values for the proposed clustering system

Similarity Threshold	Overall Precision	Overall Recall
0.9	0.81	0.55
0.8	0.78	0.62
0.7	0.75	0.67
0.6	0.72	0.67
0.5	0.67	0.68
0.4	0.64	0.68
0.3	0.62	0.68
0.2	0.57	0.69
0.1	0.53	0.76

0.6. When we increase the similarity threshold value to 0.7, the recall value for the proposed system remains almost constant at 0.67, but the precision value increases to 0.75. When we continue to increase the value of the similarity threshold, the recall value for the proposed system decreases sharply. As a result, it is concluded that the most appropriate value for the similarity threshold can be selected at around 0.7. At this value for the similarity threshold, the precision and recall values for the proposed system are sufficiently greater than the average precision and recall value for the selected 12 anti-virus products. Also, there are 127 clusters generated by the proposed system at this value for the similarity threshold. It is not much greater than the number of clusters generated for the same malware data set (119 clusters) by the reference clustering described above.

When we examine the previous works in the literature, we realize that some of them attain better results for the precision and recall of their malware clustering methods. However, the malware sample files in the datasets used by the experiments in all of these works are mostly labelled as malware by almost all of the anti-virus products. For instance, many previous works use the CWSandbox [88] example dataset which contains sample files that are labelled as malware by nearly all the popular anti-virus products. After our examinations, we conclude that a measure can be defined for quantifying the difficulty of clustering the malware sample files in a dataset. This measure can compute the ratio of the malware labels to all the labels (clean plus malware labels) assigned to the files in the dataset by the popular anti-virus products. The more the value of this measure for a dataset, the less difficult to cluster the malware sample files in this dataset. For our dataset containing 237 malware sample files, the value of this measure is 0.62 which is computed by taking into account the labels assigned to the sample files by the selected 12 anti-virus products. We believe that without the specification of a value for this measure, it is not meaningful to compare the precision and recall values for two different malware clustering systems.

The second set of experiments measures the efficiency of the proposed malware clustering system in terms of the execution time. The 237 malware sample files are augmented (multiplied) by copying them at each experiment iteration for this purpose. The proposed malware clustering system is used to cluster the multiplied numbers of malware sample files and the execution times for the clustering process are collected for each iteration. The files are fed into the proposed system incrementally. When passing from experiment iteration x to $x + 1$, the number of files are augmented and only the newly created files are fed into the system for incremental clustering. For instance, when passing from the first to the second iteration, only the newly created 2370 sample files are fed into the system incrementally, even if there are 3555 files in total in the second experiment. During the experiments for measuring the efficiency of the proposed malware clustering system, the value of the similarity threshold ($MalwareComponentSimilarityThreshold$) is held constant at 0.7. Table 7.2 shows the execution time for the proposed clustering process for different number of malware sample files added incrementally.

Tab. 7.2: Malware clustering execution times for the proposed malware clustering system

Total Number of Malware Files	Number of Added Malware Files	Incremental Execution Time for Clustering (seconds)
1185	-	352
3555	2370	774
8295	4740	1614
17775	9480	3726
36735	18960	7661
74655	37920	15012
150495	75840	32812
302175	151680	60616

It can be deduced from Table 7.2 that the increase in the execution time with the number of malware files which is doubled in each iteration is lower than quadratic increase. Actually, it is almost linear in the number of malware files. Since the collected sample malware files are copied in each iteration to augment the malware sample set, the malware families and therefore the upper bound for the breadth of the malware component tree do not change across the iterations. When we double the number of malware files across the iterations, the execution time for the incremental clustering is roughly doubled in each iteration (actually, it becomes slightly more than double of the execution time of the previous iteration). These results support the complexity reasoning for the proposed incremental clustering algorithm in Section 7.3. This reasoning indicates that the execution time complexity of the proposed algorithm is $O(NF)$, where N is the number of malware files to be clustered in each iteration and F is bounded by the number of malware families generated by the proposed clustering system (which is actually an upper bound on the breadth value of the malware component tree). When the number of malware families is held constant, the time complexity of the proposed clustering algorithm changes almost linearly with the number of malware files (with the linearity constant almost 1) as can be deduced from Table 7.2. (In this discussion, it should be noted that the number of the clusters (128) generated by the proposed clustering system when the similarity threshold value is set to 0.7 is comparable to the number of the clusters (119) generated by the reference clustering for the same data set and it is not changed across the experiment iterations shown in Table 7.2.) In addition, when we consider the number of malware files received per day by the anti-virus companies is on the order of 100Ks, we believe that the computed execution time for clustering 150K malware samples (151680 samples) is reasonable for the generation, update and deployment of malware signatures before any of these malware propagate to a wide range of machines on the Internet.

The utilization of an incremental, tree-based algorithm for clustering malware files seems to have significant advantages over using clustering algorithms requiring the execution of the algorithm on all the malware files collected so far with each introduction of a set of new malware files. The proposed incremental clustering algorithm has also another advantage. It does not require the computation of the similarity (distance) values between each pair of malware files to be clustered. It allows even around 150K malware files to be clustered in a reasonable time interval incrementally. The behavioural signature for the malware files belonging to a cluster is generated by a simple depth-first search on the clustering (malware component) tree. The signatures are generated in a format that easily allows to match them against the software logs collected across a target network (in the context of SIEM tools). They do not contain any conditions that need to be satisfied among the fields of their components, which may necessitate the usage of more complex and resource-intensive signature matching algorithms during the software log analysis.

The proposed malware behavioural clustering system now supports the clustering of only the malware files that are specific to the Windows operating system, since it only provides support for the processing of the Windows system and API calls for now to generate the behavioural artifacts. As a future work, appropriate extensions allowing to generate behavioural artifacts from system (or API) calls specific to the other operating systems can be implemented and integrated into the system.

8
Attack Scenario Detection Using Network-wide Logs

The main aim of the overall system proposed in this thesis work is to detect ongoing attack scenarios from the collected logs by creating dynamically expanded partial attack graphs. The inputs to the attack scenario detection process are the full attack graph for the target network, the generated behavioural malware signatures and the logs collected across the target network. The logs are composed of the entries in the log files of various software running on the hosts of a target network. We first apply log preprocessing to convert the logs collected from different software into a common, normalized format and then eliminate the spurious, duplicate logs. The remaining logs are then classified as primary or secondary. A primary log directly indicates an attack. It can be a vulnerability exploit attempt among others. A secondary log can not indicate an attack *by itself*, but a collection of more than one secondary log can indicate an execution of a malware and in this case, each secondary log of the collection represents a malicious activity.

After log preprocessing, we apply different processing to the primary and secondary logs to determine the attack scenario elements indicated by them. The attack scenario elements can be matched directly to the attack graph nodes. There are three types of attack scenario elements that are defined in the system:

- Usage of a privilege by an attacker which correspond to privilege nodes in the attack graphs,
- Vulnerability exploits by an attacker which correspond to vulnerability exploit nodes in the attack graphs,

- Usage of an information source (e.g. accessing a credentials store, reading cookie files) which correspond to information source usage nodes in the attack graphs.

The attack scenario elements are used to mark the appropriate nodes in the partial attack graphs as *alarmed nodes* during the correlation of the log processing results. The alarmed nodes in the partial attack graphs are used to form attack scenarios by applying partial attack graph filtering and determine the nodes in the partial attack graphs to be further expanded in each correlation phase during the life cycle of the system. After expansion of the partial attack graphs, they are checked pairwise to see if they can be merged.

As stated above, we process the software logs collected throughout the target network in streaming mode to detect the ongoing attack scenarios targeted to the network. Four phases of operations are defined in this scope: log collection, log preprocessing, log processing and log processing results' correlation. The algorithms implemented in the context of these phases are executed continuously in a pipelined manner. These algorithms are implemented over Apache Storm framework containing a spout layer and three layers of bolts. The log processing framework architecture is shown in Figure 8.1.

The spouts are responsible for collecting and preprocessing the logs. The first two bolt layers are responsible for primary and secondary log processing (malicious activity detection). The last layer of bolts are responsible for correlating the log processing results. In this section, we describe each of the operational phases in detail.

8.1 Log Collection

We have implemented software to collect the logs generated by various software. We can collect:

- Apache web server logs,
- Linux, BSD system logs,
- Microsoft IIS logs,
- Windows system and application logs,

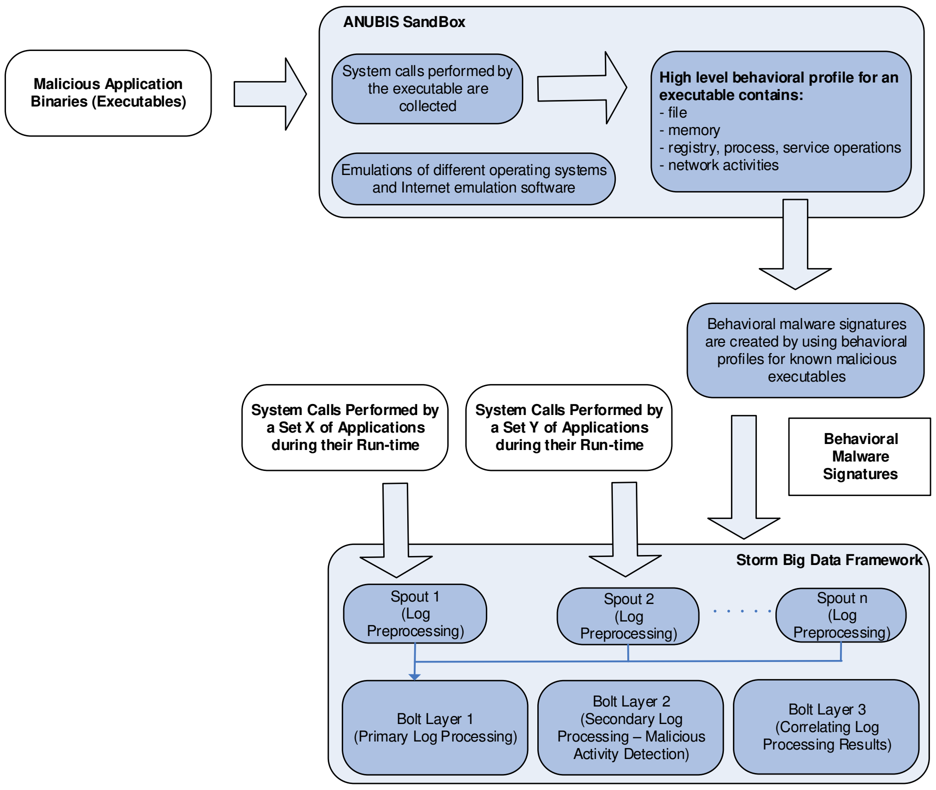

Malicious Application Binaries (Executables)
ANUBIS SandBox
System calls performed by the executable are collected
Emulations of different operating systems and Internet emulation software
High level behavioral profile for an executable contains:
- file
- memory
- registry, process, service operations
- network activities
Behavioral malware signatures are created by using behavioral profiles for known malicious executables
System Calls Performed by a Set X of Applications during their Run-time
System Calls Performed by a Set Y of Applications during their Run-time
Behavioral Malware Signatures
Storm Big Data Framework
Spout 1 (Log Preprocessing)
Spout 2 (Log Preprocessing)
Spout n (Log Preprocessing)
Bolt Layer 1 (Primary Log Processing)
Bolt Layer 2 (Secondary Log Processing – Malicious Activity Detection)
Bolt Layer 3 (Correlating Log Processing Results)

Fig. 8.1: Log processing framework architecture

- Alerts generated by Snort intrusion detection system.

The log collection process is performed by a number of spouts implemented over Apache Storm framework and requires (log collector) agents to be installed on the network hosts. The collected logs are preprocessed inside the spouts as described in the next section.

8.2 Log Preprocessing

Log preprocessing is responsible for normalizing the collected logs, eliminating the duplicates among them and classify them. First, the collected logs are converted into a common, proprietary format with the normalization procedure. A normalized log can be represented by a tuple containing the following fields: source and target IP address, source and target port, source and target application path, source and target process identifier, source and target thread identifier, source and target user name, processed data type, processed data size, processed data path, processed data creation and modification time, log activity type, log timestamp and references to other normalized logs aggregated by the normalized log.

As can be understood from the normalized log representation, it can indicate a network event (e.g. TCP connection establishment, UDP data transfer) or an event local to a single network host. A normalized log can also be related to a single processed data block. This processed data can be a modified registry record, a file transferred over the network, a modified file located on the file-system of a network host, among others. A normalized log also contains references to its aggregated normalized logs. The timestamp field for a normalized log is set to the minimum timestamp of its aggregated logs.

The log activity type field for a normalized log indicates the main system operation that the log indicates. The possible values for this field are hard-coded in the system. It can be network data transfer, network connection establishment, data update, shared library load, file read/update/delete/execute, registry key read/update/delete, memory area read/update, vulnerability exploit among others. It should be noted that some of the fields of a normalized log may have no (empty) value.

After normalization, log aggregation takes place. The first operation performed by the aggregation is eliminating the duplicate normalized logs. Two normalized logs are assumed to be duplicate, if all of their initialized fields are equal and the difference between their timestamps is less than some threshold value. After eliminating duplication, the aggregation tries to merge the related logs indicating network traces into network flows. It constructs the network flows by tracking TCP sessions. Off-the-shelf tools can also be used to construct the network flows. A network flow is also represented by a normalized log instance recording general characteristics for the corresponding network traffic such as total bytes/packets transferred in each direction, bytes/packets transferred per second in each direction, etc.

After aggregation, normalized log classification takes place. The classification operation classifies the normalized logs, whose log activity type denotes a vulnerability exploit, as primary logs. The other normalized logs are classified as secondary logs. Each operation of the log preprocessing phase is performed by the spouts implemented over Apache Storm framework.

8.3 Log Processing

After a spout has completed its log preprocessing operations for a log in streaming node, the log is transferred to one of the first layer of bolts implemented on the Apache Storm framework. The log is processed differently by the bolts in this layer according to whether it is a primary or secondary log.

8.3.1 Primary Log Processing

A primary log denotes directly a vulnerability exploit. Therefore, primary log processing creates a vulnerability exploit element from each primary log in a straightforward manner. Each of the created vulnerability exploit element represents an attack scenario element and has the same fields as *Vulnerability Exploit* class in the proposed attack graph model in this thesis work. The attacker privileges required to benefit from the vulnerability exploits indicated by the created elements are also fetched from the full attack graph for the target network. These attacker privileges are included also in the attack scenario elements created by the primary log processing. The created attack scenario elements (vulnerability exploit and attacker privilege elements) are transferred directly to the third (last) layer of bolts by jumping over the second layer of bolts. They are used to create/update the attacker privilege and vulnerability exploit nodes in the partial attack graphs in log processing results' correlation phase.

8.3.2 Secondary Log Processing

A secondary log can not directly indicate a vulnerability exploit (or attack). However, a collection of more than one secondary log can indicate an execution of a malware. This indication is deduced by matching the secondary logs to the behavioural malware signatures. If a collection of secondary logs indicate an execution of a malware, then each secondary log in the collection corresponds to a single malicious activity that is a component of the behavioural signature of the malware.

If a log received by a first layer bolt from a spout is a secondary log, then the bolt first finds the malicious activities (behavioural artifacts) inside all the behavioural malware signatures that can be matched to the log. The matching is true, if and only if the values of the corresponding fields of the log and any operating system (OS) object of the behavioural artifact are equal. A precomputed set of hash maps formed by components (path components) of OS object identifiers is utilized to perform the matching. The behavioural artifacts that are matched to the received log are found by finding the components of the OS path identifier contained by the log and querying the precomputed hash maps with these components. Afterwards, the bolt transfers the secondary log and the information about its matched behavioural artifact to the appropriate second layer bolt which is responsible for collecting the logs that can be related to different malware instances. The communication among the first layer and second layer bolts for the secondary log processing phase is multiplexed according to the source IP address stored in the received logs. Namely, the logs having the same source IP address are processed by the same second layer bolt. This ensures that the malicious activities occurring at an IP address are collectively processed by the same second layer bolt to determine the existence of a malware executing at this IP address.

When a second layer bolt receives a number of logs and the information about their matched malware behavioural artifacts, it checks whether these logs together with some of the previously received logs satisfy the behavioural signature for the malware containing the behavioural artifacts. Namely, the bolt checks whether there is adequate number of appropriate logs that have been matched to the OS objects of the behavioural artifacts of the malware. In order to check this, the bolt stores information about the previous log-signature matchings in its local memory.

If a malware signature is satisfied by a collection of secondary logs, then the attacker privileges providing for the (malicious) activity indicated by each secondary log matched to the signature are determined. An attack scenario element is created from each determined attacker privilege and is sent to a third (last) layer bolt.

The derivation of the values of the fields of an attacker privilege from the fields of a secondary log is hard-coded in the system. For instance, if a secondary log indicates that a file on a web server that has an executable (application) path $ep1$ and is running on a machine with an IP address $ip1$ is modified (activity type field), then an attacker privilege is created with its category field set to file modification, IP address field set to $ip1$ and application name field set to a value derived from the string $ep1$. The CPE [15] identifier field of the created attacker privilege is also filled by using the string $ep1$ and *CPE identifier-software application path mappings* prepared for each target network host before the attack scenario detection process begins.

8.4 Log Processing Results' Correlation

Primary log processing generates vulnerability exploit attempts and attacker privileges as alarmed attack scenario elements. Secondary log processing generates attacker privileges as alarmed attack scenario elements. The aim of the log processing results correlation phase is to correlate the generated alarmed attack scenario elements in attack scenarios each of which is represented by a partial attack graph. The third (last) layer of bolts implemented over Apache Storm framework are responsible for performing this correlation. It should be noted that the full attack graph for the target network is used as an input for creating/updating the partial attack graphs.

The main algorithm for the correlation phase is shown in Algorithm 8.2. It essentially defines a loop that continues up to receiving a finish signal (*GetUpdatesForFinishSignal()*) from the graphical user interface or elsewhere. In the loop, a specified number of the alarmed attack scenario elements that are generated as a result of (primary and secondary) log processing are fetched and processed. If an alarmed attack scenario element can not be matched to any node of the currently existing partial attack graphs, a new partial attack graph is created with the attack scenario element as its root node. Otherwise, the nodes that are matched to the alarmed attack scenario element are marked as *alarmed*. This is exemplified in Figure 8.3. The function that is used to match the nodes of the partial attack graphs to the attack scenario elements is the *FindVertexInGraph()* function. It is a simple function that compares the non-null values of the appropriate fields of the attack scenario element and the nodes of the partial attack graphs.

```
1: procedure GeneratePartialAttackGraphs($fullAttackGraph$, $expansionDepth$,
   $expansionBreadth$)   ▷ The full attack graph ($fullAttackGraph$) for the target network. Expansion
   depth ($expansionDepth$) and breadth ($expansionBreadth$) for privileges to be expanded.
2:     $partialAttackGraphs \leftarrow$ EmptyList()
3:     $finishSignal \leftarrow FALSE$;
4:     while $finishSignal == FALSE$ do
5:         $alarmedPrivilegesAndExploits \leftarrow$ FetchResultsOfLogProcessing()
6:         for all $alarmedEntity \in alarmedPrivilegesAndExploits$ do
7:             $alarmedEntityFound \leftarrow FALSE$
8:             for all $partialAttackGraph \in partialAttackGraphs$ do
9:                 $partialAttackGraphNode \leftarrow$ FindVertexInGraph($partialAttackGraph$,
10:                    $alarmedEntity$)
11:                $partialAttackGraphNode.alarmed \leftarrow TRUE$
12:                $alarmedEntityFound \leftarrow TRUE$
13:            end for
14:            if $alarmedEntityFound == FALSE$ then
15:                $newPartialAttackGraph \leftarrow$ EmptyGraph()
16:                AddVertexToGraph($newPartialAttackGraph$,, Copy($alarmedEntity$))
17:                AddToList($partialAttackGraphs$, $newPartialAttackGraph$)
18:            end if
19:        end for
20:        ExpandPartialAttackGraphs($partialAttackGraphs$, $fullAttackGraph$,
21:            $expansionDepth$, $expansionBreadth$)
22:        MergePartialAttackGraphs($partialAttackGraphs$)
23:        GetUpdatesForFinishSignal($finishSignal$)
24:        if a signal has been received from graphical user interface to show attack scenarios then
25:            $attackScenarios \leftarrow$ FilterPartialAttackGraphs($partialAttackGraphs$)
26:            SendToGUI($attackScenarios$)
27:        end if
28:    end while
29: end procedure
```

Fig. 8.2: Generating partial attack graphs using evidence processing results

After each of the fetched alarmed attack scenario elements are processed, the partial attack graphs are expanded by using the input full attack graph for the target network (ExpandPartialAttackGraphs()). The expansion breadth is also an input for the correlation algorithm and is not changed during the algorithm execution. It should be determined so that it accounts for the maximum number of the consecutive vulnerability exploit attempts that can be missed by a *single* IDS/IPSs located at the gateway of a network (or a subnet). The expansion depth is also an input which is not changed during the algorithm execution. It should account for the maximum number of the consecutive vulnerability exploit attempts that can be missed by a *chain* of IDS/IPSs located inside the network.

The partial attack graphs are expanded by extending the privilege nodes

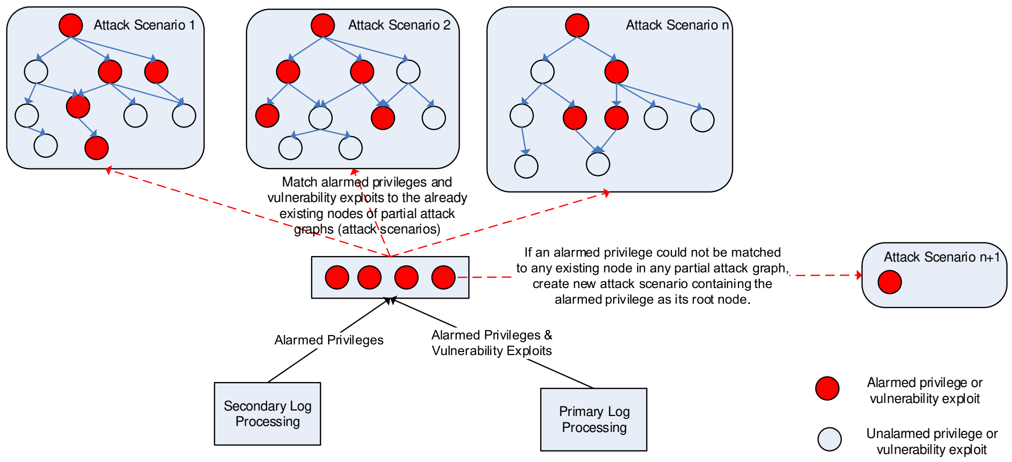

Attack Scenario 1
Attack Scenario 2
Attack Scenario n
Match alarmed privileges and vulnerability exploits to the already existing nodes of partial attack graphs (attack scenarios)
If an alarmed privilege could not be matched to any existing node in any partial attack graph, create new attack scenario containing the alarmed privilege as its root node.
Attack Scenario n+1
Alarmed Privileges
Alarmed Privileges & Vulnerability Exploits
Secondary Log Processing
Primary Log Processing
Alarmed privilege or vulnerability exploit
Unalarmed privilege or vulnerability exploit

Fig. 8.3: Matching alarmed attack scenario elements to the nodes of the partial attack graphs

of them. The extension of a privilege node gives rise to the creation of vulnerability exploit nodes and further privilege nodes. The privileges to be extended in the partial attack graphs in each iteration of the loop are determined by using the ratio of their *alarmed neighbour nodes* in the partial attack graphs, among other factors. After expansion of the partial attack graphs in each iteration of the main loop, the partial attack graphs are tried to be merged together (MergePartialAttackGraphs()). Some of the partial attack graphs can be merged and yield a single partial attack graph as a result of this process.

When the user wants to display the constructed attack scenarios, the algorithm filters the current partial attack graphs to form a list of attack scenarios (FilterPartialAttackGraphs()). A single attack scenario is generated from each partial attack graph by eliminating *unalarmed nodes* of the partial attack graph during filtering, which are far away from any alarmed node in the graph. A threshold value for path lengths is used in this procedure.

8.4.1 Partial Attack Graph Expansion

Partial attack graphs are expanded by extending the selected privileges inside them during each iteration of the main loop in the correlation algorithm, as shown in Figure 8.2. The full attack graph is used to expand the partial attack graphs. The pseudo-code of the expansion algorithm is shown in Figure 8.4. At the start of the algorithm, the privileges to be extended in each partial attack graph need to be determined. For this purpose, a score function is executed for each privilege of a partial attack graph to compute its score (ComputeScoreForPrivileges()). The score value for a privilege is affected by the alarmed status of the privilege and the ratio of the alarmed neighbour privileges of the privilege. The score function is shown in Equation 8.1.

$$Score(p) = p_a * 0.5 + (p_{an}/p_n) * 0.5 \tag{8.1}$$

where p_a is 1 if privilege p is an alarmed privilege and otherwise 0. p_{an} denotes the count of the alarmed neighbour privileges of privilege p and p_n denotes the count of all the neighbour privileges of privilege p. The privilege conjunctions, vulnerability exploits and information sources are ignored in neighbour computation.

A privilege in a partial attack graph can be expanded at most once. We give precedence to the alarmed privileges or privileges which have alarmed neighbour privileges, when selecting the privileges to be expanded. After computing the scores for all the privileges in a partial attack graph that have not been expanded so far, they are sorted according to their scores in a decreasing manner (SortPrivilegesByScore()). A random number of privileges are selected from the front of the sorted privilege list (GetListElements()) for expansion. The selected privileges are marked as being expanded.

In order to expand a privilege in a partial attack graph, the input full attack graph for the target network is used. The privilege is found on the full attack graph (FindVertexInGraph()). A sub-graph of the full attack graph rooted at the privilege with the specified expansion depth and breadth are extracted (ExtractSubGraphFromGraph()) and added to the corresponding partial attack graph at the point of the privilege (AddSubGraphToGraph()). The privileges existing in the extracted sub-graph of the full attack graph are marked so that they will not be extracted in the future requests. Actually, the function AddSubGraphToGraph() ignores the marked, already extracted privileges, when extracting a sub-graph from the full attack graph.

```
procedure ExpandPartialAttackGraphs(partialGraphs, fullAttackGraph, expansionDepth, expansionBreadth)    ▷ A list (partialGraphs) of partial attack graphs to be expanded. The full attack graph (fullAttackGraph) for the target network. Expansion depth (expansionDepth) and breadth (expansionBreadth) for the privileges to be expanded.
    for all $partialGraph \in partialGraphs$ do
        ComputeScoreForPrivileges(partialGraph)
        allPrivileges ←VertexList(partialGraph)
        notExpandedPrivileges ←EmptyList()
        for all $privilege \in allPrivileges$ do
            if $privilege.expanded == FALSE$ then
                AddToList(notExpandedPrivileges, privilege)
            end if
        end for
        sortedPrivilegeList ←SortPrivilegesByScore(notExpandedPrivileges, 'descending')
        randomNumber ←Random(Size(sortedPrivilegeList))
        expandedPrivileges ←GetListElements(sortedPrivilegeList, 0, randomNumber)
        for all $expandedPrivilege \in expandedPrivileges$ do
            $expandedPrivilege.expanded \leftarrow TRUE$
            fullGraphExpandedPrivilege ←FindVertexInGraph(fullAttackGraph,
                expandedPrivilege)
            subGraph ←ExtractSubGraphFromGraph(fullAttackGraph,
                fullGraphExpandedPrivilege, expansionDepth, expansionBreadth)
            AddSubGraphToGraph(partialGraph, expandedPrivilege, subGraph)
        end for
    end for
end procedure
```

Fig. 8.4: Expanding partial attack graphs

8.4.2 Partial Attack Graph Merging

After expansion of the partial attack graphs, they are checked pairwise to see if they can be merged. If a partial attack graph contains a node n with the same content as the root node of another partial attack graph, then the second partial attack graph is added as a sub-graph inside the first partial attack graph at the location of node n. The pseudo-code for the partial attack graph merging algorithm is shown in Figure 8.5. When merging one partial attack graph inside another, each edge of the first partial attack graph is traversed and checked if there exists a vertex in the second attack graph with the same content as the source/target vertex of the edge. If there exists not, a new vertex that has the same content with the source/target vertex of the edge is created and added to the second graph. A new edge between a pair of vertices that correspond to the source and target vertices of the original edge is created and added to the second graph, if such an edge does not exist already in this graph.

After merging a partial graph inside a second, the first graph is marked as being *merged*. When there is no possibility of merging any two attack graphs, the partial attack graphs that are not marked as being merged are added to the result graph list. The graphs in this list are the new partial attack graphs

```
1: procedure MergePartialAttackGraphs($partialGraphs$)    ▷ A list ($partialGraphs$) of partial attack graphs. Opportunities for merging any number of graphs from $partialGraphs$ are searched.
2:     $resultGraphs \leftarrow$ EmptyList()
3:     $mergeHappened \leftarrow TRUE$
4:     while $mergeHappened == TRUE$ do
5:         $mergeHappened \leftarrow FALSE$
6:         for all $partialGraph1 \in partialGraphs$ do
7:             for all $partialGraph2 \in partialGraphs$ do
8:                 if ($partialGraph1$ != $partialGraph2$) and ($partialGraph2.merged == FALSE$) then
9:                     $partialGraph1Vertices \leftarrow$ VertexList($partialGraph1$)
10:                    $partialGraph2Root \leftarrow$ RootVertex($partialGraph2$)
11:                    if $partialGraph1Vertices$ contains a vertex with the same content as $partialGraph2Root$ then
12:                        for $partialGraph2Edge \in$ EdgeList($partialGraph2$) do
13:                            $sourceVertex1 \leftarrow$ FindVertexInGraph($partialGraph1$,
14:                                SourceVertex($partialGraph2Edge$))
15:                            if $sourceVertex1 == Null$ then
16:                                AddVertexToGraph($partialGraph1$, SourceVertex($partialGraph2Edge$))
17:                                $sourceVertex1 \leftarrow$ SourceVertex($partialGraph2Edge$)
18:                            end if
19:                            $targetVertex1 \leftarrow$ FindVertexInGraph($partialGraph1$,
20:                                TargetVertex($partialGraph2Edge$))
21:                            if $targetVertex1 == Null$ then
22:                                AddVertexToGraph($partialGraph1$, TargetVertex($partialGraph2Edge$))
23:                                $targetVertex1 \leftarrow$ TargetVertex($partialGraph2Edge$)
24:                            end if
25:                            AddEdgeToGraph($partialGraph1, sourceVertex1, targetVertex1$)
26:                        end for
27:                        $partialGraph2.merged \leftarrow TRUE$
28:                        $mergeHappened \leftarrow TRUE$
29:                    end if
30:                end if
31:            end for
32:        end for
33:    end while
34:    for all $partialGraph \in partialGraphs$ do
35:        if $partialGraph.merged == FALSE$ then
36:            AddElementToList($resultGraphs, partialGraph$)
37:        end if
38:    end for
39:    $partialGraphs \leftarrow resultGraphs$
40: end procedure
```

Fig. 8.5: Merging partial attack graphs

that will be used during the next iteration of the main loop of the correlation algorithm shown in Figure 8.2.

8.4.3 Partial Attack Graph Filtering

The aim of the partial attack graph filtering is to eliminate the unrelated nodes from the partial attack graphs and construct the final attack scenario for each partial attack graph. During filtering, a basic BFS (Breadth First Search) is applied on a copy of each partial attack graph and the *unalarmed nodes* that are closer to any alarmed node more than a specific threshold value are marked. After BFS finishes, the unmarked and unalarmed nodes are deleted from the copy of the partial attack graph. The resulting copy of the partial attack graph constitutes an attack scenario.

8.5 Experiments

Two different sets of experiments are conducted in the context of the attack scenario detection process. The first set of experiments measures the performance of the secondary log processing process. The secondary log processing is performed by the first and second layer of bolts in the Apache Storm framework by matching the collected logs to the generated behavioural malware signatures. The second set of experiments illustrates a sample case study including the processing of both primary and secondary logs.

8.5.1 Experiment Set 1

The aim of the experiments described in this section is to measure the performance of the signature-based secondary log processing process. A set of sample networks each containing a different number of hosts is used in the experiments. For each experiment, a number of hosts in the used network is infected by a randomly selected malware by injecting the log records generated by the corresponding malware into the corresponding host's log records. The time interval between the consecutive activities performed by each malware instance is also varied across the experiments by varying the time interval between the injections of the consecutive log records of the malware instance to the corresponding host's logs.

Each component (behavioural artifact) of a behavioural malware signature should be matched to a collection of appropriate logs in order to be accounted as a matched signature. The performance of the signature-based secondary log processing process is evaluated in terms of the average delay between the injection/generation of the last log record for each malware instance and the reporting of the matched signature for the malware instance by the proposed system.

The experiments are conducted by using two PCs each having Intel Corei7-2620M 2.70GHz CPU and 8GB RAM. Each experiment is conducted at least ten times and the average performance (delay for malware detection) values are computed. The parameters for each experiment are:

- the number of the hosts in the used network
- the number of the hosts in the used network which are infected by a malware instance
- the time interval between the consecutive activities performed by each malware instance

Benign log records are interleaved with the log records performed by malware instances for each experiment. The count of the benign system calls generated by one host per second is held at constant across the experiments. This constant value is equal to roughly 100. The benign logs are generated by using the log records generated during 3-day operation of an experiment PC (client computer) which is assumed to be not infected by any malware. This experiment PC is not connected to Internet and is checked against possible malware infection with external tools and found to be clear. To generate the benign logs, also some popular, built-in Windows executables (e.g., notepad.exe, mspaint.exe) are executed in this experiment PC. While creating the benign logs for a host in the used network for each experiment, a number of the benign logs collected from the experiment PC are selected randomly and the IP address for the host is attached to each of the randomly selected logs.

In Table 8.1, the results of the conducted experiments are shown. In each of the experiments, all the malware instances are detected by the secondary log processing process. The network size in terms of the host count and the time interval between malicious log records (injected activities) adversely affect the average delay for the malware detection. This effect is caused by the increased number of the benign log records interleaved among the malicious

Tab. 8.1: Performance measurement of the secondary log processing through several experiments

Network Size (Host Count)	Infected Host Count	Time Interval Between Malicious Log Records (ms)	Average Delay for Malware Detection (ms)
100	10	400	622
100	10	1200	1676
100	10	3600	4386
100	10	10800	14479
100	20	400	964
100	20	1200	2338
100	20	3600	7738
100	20	10800	20196
200	20	400	1983
200	20	1200	5604
200	20	3600	15833
200	20	10800	46508
200	40	400	2890
200	40	1200	8329
200	40	3600	23067
200	40	10800	70981
400	40	400	6018
400	40	1200	19875
400	40	3600	62382
400	40	10800	190208
400	80	400	9349
400	80	1200	31972
400	80	3600	96650
400	80	10800	288526

ones. The effect is not very huge though. The number of infected hosts (malware instances) in the used network also adversely affects the average delay for the malware detection, but the effect is smaller than that for the previous case. In this case, the log records generated by additional malware instances on newly infected hosts can interleave among the malicious log records of a specific malware instance on a previously infected host.

Even if the network size, the time interval between consecutive malicious log records and the number of malware instances injected into the used network adversely affect the performance of the secondary log processing, the resulting performance is adequate for near real-time detection of ongoing malicious activities. When the network contains 400 hosts, the time interval between the malicious log records is 400 milliseconds and 80 hosts are infected, the average malware detection delay is almost 10 seconds. By considering this detection delay value, we can conclude that the advance of malware on the propagated hosts can be hindered. When we consider that the average number of the log records generated by a malware instance is on the order of 100 and there is 400 milliseconds of time interval between each consecutive activity of the malware instance, then the total time for the malware instance to complete its activities is almost 40 seconds at minimum. Even if the malware instance has jumped to a new network host, it can be detected before completing even half of its activities. (It should be noted that this is just an informal reasoning that do not take the delays imposed by the logging system of different software and network delays into account.)

8.5.2 Experiment Set 2

The aim of the experiments described in this section is to show the results generated by the proposed system for a sample network. The network is shown in Figure 8.6. Two different attack scenarios are conducted on the network simultaneously and the attack scenarios generated by the proposed system are examined. The experiments are conducted on a network simulation environment in a single PC with Intel Corei7-2620M 2.70GHz CPU and 8GB RAM. The sample network is modelled on the simulation environment.

The first attack scenario conducted on the sample network comprises the infection of the IP devices with addresses 10.0.2.2 and 10.0.2.15 with the Stuxnet (version b2) worm via USB flash disks. It is assumed that the host-based intrusion sensors on these IP devices can not detect the exploit attempts for the vulnerabilities used by the Stuxnet worm. Therefore, there

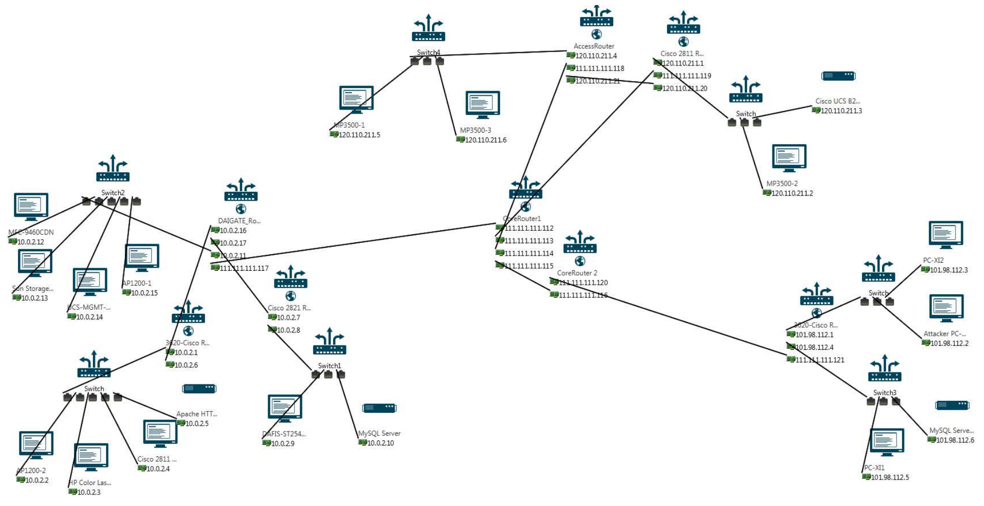


Fig. 8.6: Sample network for attack scenario detection system evaluation

is no primary logs (alerts) collected related to these infections. The logs generated by the execution of the Stuxnet malware are created (simulated) by using the dynamic malware analyzer (ANUBIS) report for the Stuxnet (version b2) worm. These logs serve as the secondary logs for the execution of the proposed system, since they are not related to any security alert (vulnerability exploit or virus detection, etc.).

The second attack scenario conducted on the sample network comprises the infection of the IP devices with addresses 10.0.2.5 and 10.0.2.9 with the Zbot Trojan (Trojan.Zbot.B) via connecting to the IP device with address 101.98.112.2 (Attacker PC). The Zbot Trojan exploits the vulnerability with CVE identifier CVE-2008-0726 on the IP device with address 10.0.2.9. It exploits the vulnerability with CVE identifier CVE-2009-2994 on the IP device with address 10.0.2.5. It is assumed that these exploit attempts are detected by the network-based intrusion detection sensor deployed on the IP device DAIGATE_Router containing addresses 10.0.2.16 and 10.0.2.17. Therefore, we have collected primary logs (alerts) for the exploit attempts in this scenario. The other logs generated by the execution of the Zbot Trojan are created (simulated) by using the dynamic malware analyzer (ANUBIS) report for the corresponding version of the Zbot Trojan. These logs serve as the secondary logs.

We create (simulate) benign logs in addition to the logs related to the malware executions. In order to create the benign logs, we first execute well-known non-malicious programs (Windows Explorer, Internet Explorer, Mozilla Firefox, Google Chrome, Adobe PDF Reader, calc.exe, notepad.exe, etc.) on a clean Windows system and collect the log records generated by the execution of these programs. We copy a random subset of these log records for each IP device on the sample network and set the source IP address of each log record trace to the IP address of the device it is related to. There are almost 10000 benign logs for each IP device. At the end, we randomly mix the benign logs for the infected devices with the logs related to the malware executions.

Our system generates two attack scenarios in the form of partial attack graphs. The first attack scenario detected by our system consists of the attack elements related to the devices on the sample network infected by the Stuxnet worm. It is shown in Figure 8.7.

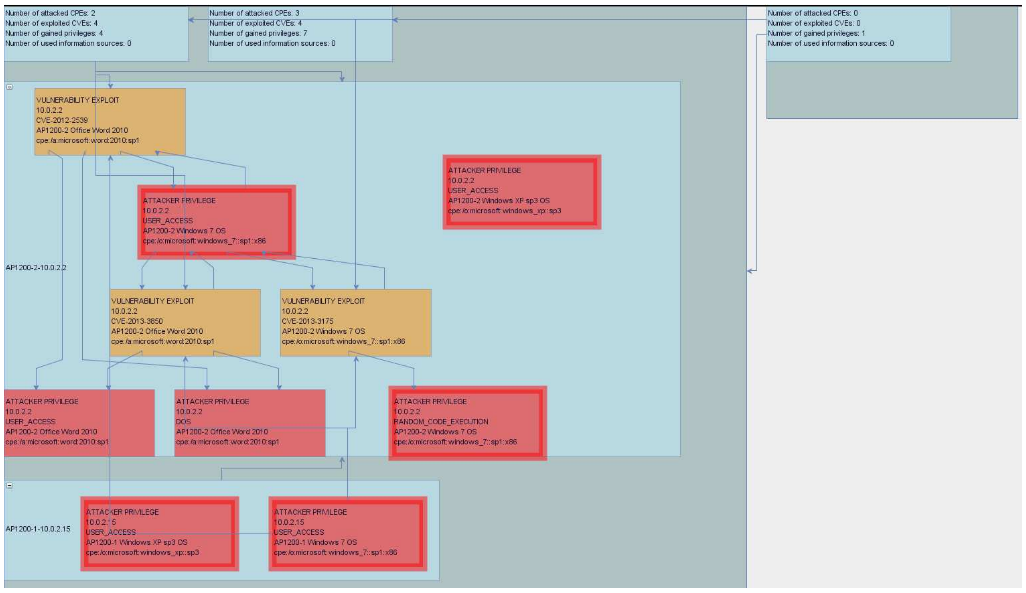

Number of attacked CPEs: 2
Number of exploited CVEs: 4
Number of gained privileges: 4
Number of used information sources: 0
Number of attacked CPEs: 3
Number of exploited CVEs: 4
Number of gained privileges: 7
Number of used information sources: 0
Number of attacked CPEs: 0
Number of exploited CVEs: 0
Number of gained privileges: 1
Number of used information sources: 0
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2012-2539
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
USER_ACCESS
AP1200-2 Windows XP sp3 OS
cpe:/o:microsoft:windows_xp::sp3
ATTACKER PRIVILEGE
10.0.2.2
USER_ACCESS
AP1200-2 Windows 7 OS
cpe:/o:microsoft:windows_7::sp1:x86
AP1200-2-10.0.2.2
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2013-3850
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2013-3175
AP1200-2 Windows 7 OS
cpe:/o:microsoft:windows_7::sp1:x86
ATTACKER PRIVILEGE
10.0.2.2
USER_ACCESS
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
DOS
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
RANDOM_CODE_EXECUTION
AP1200-2 Windows 7 OS
cpe:/o:microsoft:windows_7::sp1:x86
AP1200-1-10.0.2.15
ATTACKER PRIVILEGE
10.0.2.15
USER_ACCESS
AP1200-1 Windows XP sp3 OS
cpe:/o:microsoft:windows_xp::sp3
ATTACKER PRIVILEGE
10.0.2.15
USER_ACCESS
AP1200-1 Windows 7 OS
cpe:/o:microsoft:windows_7::sp1:x86

Fig. 8.7: Devices infected with Stuxnet worm on the sample network

The attack elements surrounded by red, tick rectangles are the alarmed elements. All the alarmed elements are attacker privileges. Since there is no primary log (alert) collected for the first attack scenario, the existence of the alarmed attacker privileges are inferred by matching the secondary logs to the behavioural malware signatures generated by our system. Two instances of the same malware signature are created for this scenario, one for the IP device with address 10.0.2.2 and the other for the IP device with address 10.0.2.15. The corresponding malware signature is used (generated) by the system to detect the Stuxnet worm (version b2) and also other similar malware. The attack scenario detected by the system and shown in Figure 8.7

also contains attack elements that are not alarmed, but are related to the alarmed ones. These unalarmed attack elements complete the attack paths that are indicated by the alarmed ones and give clues about the future attack actions that can be performed by the attackers.

The second attack scenario detected by the system consists of the attack elements related to the devices infected by the Zbot Trojan. It is shown in Figure 8.8.

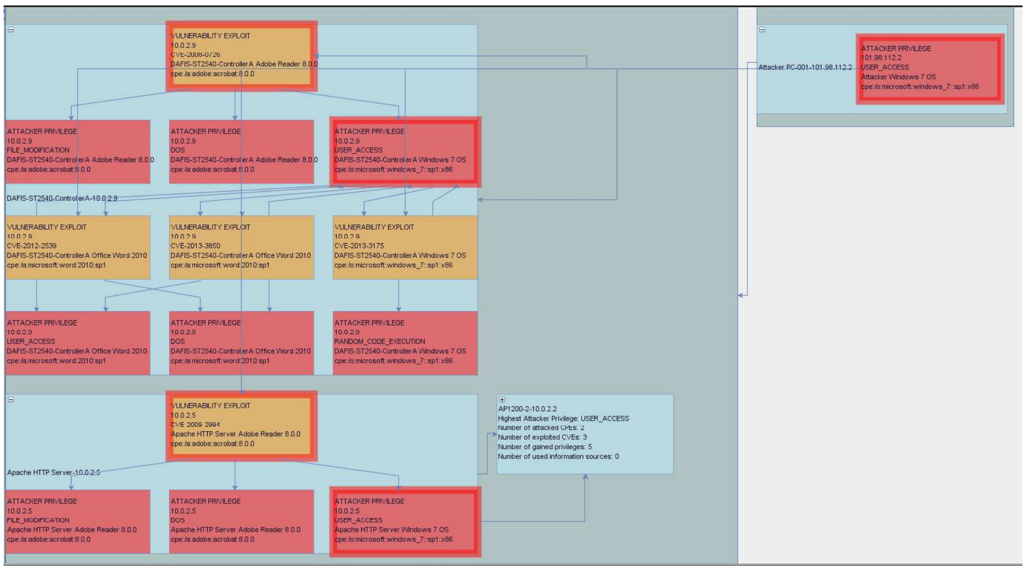

VULNERABILITY EXPLOIT 10.0.2.9 CVE-2008-0726 DAFIS-ST2540-ControllerA Adobe Reader 8.0.0 cpe:/a:adobe:acrobat:8.0.0
ATTACKER PRIVILEGE 10.0.2.9 FILE_MODIFICATION DAFIS-ST2540-ControllerA Adobe Reader 8.0.0 cpe:/a:adobe:acrobat:8.0.0
ATTACKER PRIVILEGE 10.0.2.9 DOS DAFIS-ST2540-ControllerA Adobe Reader 8.0.0 cpe:/a:adobe:acrobat:8.0.0
ATTACKER PRIVILEGE 10.0.2.9 USER_ACCESS DAFIS-ST2540-ControllerA Windows 7 OS cpe:/o:microsoft:windows_7::sp1:x86
DAFIS-ST2540-ControllerA-10.0.2.9
VULNERABILITY EXPLOIT 10.0.2.9 CVE-2012-2539 DAFIS-ST2540-ControllerA Office Word 2010 cpe:/a:microsoft:word:2010:sp1
VULNERABILITY EXPLOIT 10.0.2.9 CVE-2013-3850 DAFIS-ST2540-ControllerA Office Word 2010 cpe:/a:microsoft:word:2010:sp1
VULNERABILITY EXPLOIT 10.0.2.9 CVE-2013-3175 DAFIS-ST2540-ControllerA Windows 7 OS cpe:/o:microsoft:windows_7::sp1:x86
ATTACKER PRIVILEGE 10.0.2.9 USER_ACCESS DAFIS-ST2540-ControllerA Office Word 2010 cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE 10.0.2.9 DOS DAFIS-ST2540-ControllerA Office Word 2010 cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE 10.0.2.9 RANDOM_CODE_EXECUTION DAFIS-ST2540-ControllerA Windows 7 OS cpe:/o:microsoft:windows_7::sp1:x86
VULNERABILITY EXPLOIT 10.0.2.5 CVE-2009-2994 Apache HTTP Server Adobe Reader 8.0.0 cpe:/a:adobe:acrobat:8.0.0
Apache HTTP Server-10.0.2.5
ATTACKER PRIVILEGE 10.0.2.5 FILE_MODIFICATION Apache HTTP Server Adobe Reader 8.0.0 cpe:/a:adobe:acrobat:8.0.0
ATTACKER PRIVILEGE 10.0.2.5 DOS Apache HTTP Server Adobe Reader 8.0.0 cpe:/a:adobe:acrobat:8.0.0
ATTACKER PRIVILEGE 10.0.2.5 USER_ACCESS Apache HTTP Server Windows 7 OS cpe:/o:microsoft:windows_7::sp1:x86
AP1200-2-10.0.2.2 Highest Attacker Privilege: USER_ACCESS Number of attacked CPEs: 2 Number of exploited CVEs: 3 Number of gained privileges: 5 Number of used information sources: 0
Attacker PC-001-101.98.112.2
ATTACKER PRIVILEGE 101.98.112.2 USER_ACCESS Attacker Windows 7 OS cpe:/o:microsoft:windows_7::sp1:x86

Fig. 8.8: Devices infected with Zbot Trojan on the sample network

The attack elements surrounded by red, tick rectangles are the alarmed elements. The existence of alarmed vulnerability exploit attempts are inferred by using the primary logs (alerts) generated by the network-based intrusion detection sensor deployed on DAIGATE_ROUTER device. The existence of the alarmed attacker privilege on device Attacker PC with address 101.98.112.2 is inferred by utilizing the fact that it is a precondition for the alarmed vulnerability exploits. Namely, the existence of this attacker privilege is also inferred by using the primary logs. However, the existence of the other alarmed attacker privileges are inferred by matching the collected secondary logs to the behavioural malware signatures. It is important to note that all the alarmed attack elements, whether they are generated by processing the primary or secondary logs, form a connected graph segment in this attack scenario. This proves the existence of the related actions performed by the attacker in the context of the attack scenario. For the second attack scenario detected by the system, again two instances of the same malware signature are created, one for the device with IP address 10.0.2.5 and the other for the device with IP address 10.0.2.9. The malware signature in this case is used (generated) by the system to detect the Zbot Trojan (Trojan.Zbot.B) and also other similar malware.

9
Responding to Attack Scenarios

In the context of defense recommendation, the aim of this thesis work is to find the near optimal set(s) of available hardening (defense) measures that can be used to respond to the ongoing attack scenarios. The proposed method to finding near optimal network hardening solution(s) takes as input an *acyclic* attack graph that represents an attack scenario for the target network. A set of goal states on the attack graph selected by the network security administrator is also given as input. The goal states are actually selected from the attacker privilege nodes on the input attack graph. The method additionally takes as input the hardening measures available to the network security administrator. Each hardening measure can have the effect of eliminating some of the incoming edges of one or more weakness exploit nodes in the input attack graph. *A weakness exploit node in an attack graph can be a vulnerability exploit or information source usage node.* The set of incoming edges that can be eliminated by each hardening measure is computed before the optimization process begins. This computation can be performed by considering the contents of the hardening measures. Each measure has also a relative cost value specified by the network security administrator.

The proposed method first builds a *hardening measure graph*, which shows the effects of the available hardening measures on the input attack graph according to the target network configuration. The structure and construction of this graph are explained in Section 9.1. After the hardening measure graph is built, the simulated annealing algorithm is applied with the defined optimization criteria to find the near optimal network hardening solution(s). In generating the candidate set of hardening solutions during each iteration of the optimization algorithm, a specific candidate selection function utilizing the hardening measure graph is used. Section 9.2 details the optimization algorithm, criteria and the proposed candidate selection function.

9.1 Hardening Measure Graph

The hardening measure graph is formed by using the input *acyclic* attack graph and the hardening measures available to the network security administrator. The formal definition of a hardening measure and hardening measure graph is given below.

Definition 9.1.1. A hardening measure contains information about the effects of a defense measure occurring when it is applied to a target network. It is defined as a five element tuple $\langle Cost$, $EliminatedAttackGraphEdges$, $PathCount$, $InEdges$, $OutEdges\rangle$ in the context of an attack graph with the goal security states specified. $Cost$ specifies the relative cost for applying the hardening measure. $EliminatedAttackGraphEdges$ is a list storing references to the attack graph edges eliminated by the application of the hardening measure. $PathCount$ denotes the number of the eliminated attack paths to the goal security states after applying the hardening measure. Each attack path is a sequence of the weakness exploits on a path of the attack graph ending in one of the goal security states. $InEdges$ and $OutEdges$ are lists holding references to the incoming and outgoing edges in the hardening measure graph connected to this hardening measure. It should be noted that the definition of the content peculiar to specific defense measures (specific firewall rule addition/deletion, specific vulnerability patch) in the context of the hardening measure definition is not in the scope of this thesis work.

Definition 9.1.2. A hardening measure graph is a graph $G = (N, E)$, where N denotes the set of the nodes and E denotes the set of the edges of the graph G. $n \in N$ represents a hardening measure. $e \in E$ is a three element tuple $\langle SourceNode$, $TargetNode$, $IntersectingPathCount\rangle$, where $SourceNode$ denotes the source and $TargetNode$ denotes the target hardening measure node for the edge e. $IntersectingPathCount$ denotes the number of the attack paths that are eliminated by both the target hardening measure node and the source hardening measure node. The direction of the edge indicates that the target hardening measure can negate attack graph elements located at higher depths in the attack graph than the elements negated by the source hardening measure.

An example hardening measure graph is shown in Figure 9.1.

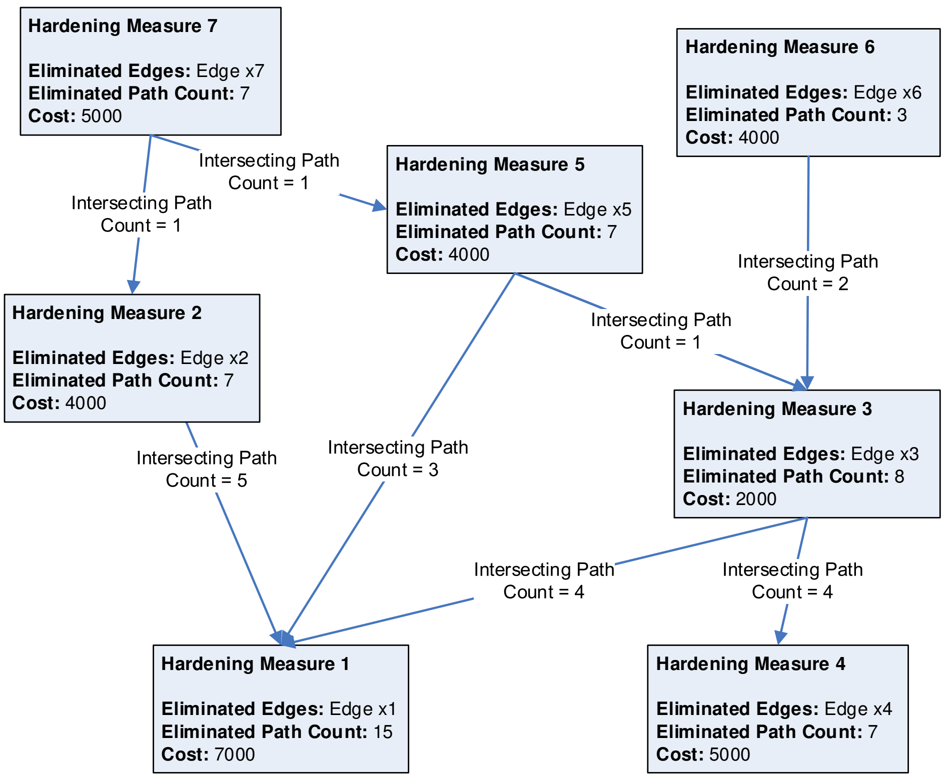

Hardening Measure 7
Eliminated Edges: Edge x7
Eliminated Path Count: 7
Cost: 5000
Hardening Measure 6
Eliminated Edges: Edge x6
Eliminated Path Count: 3
Cost: 4000
Hardening Measure 5
Eliminated Edges: Edge x5
Eliminated Path Count: 7
Cost: 4000
Hardening Measure 2
Eliminated Edges: Edge x2
Eliminated Path Count: 7
Cost: 4000
Hardening Measure 3
Eliminated Edges: Edge x3
Eliminated Path Count: 8
Cost: 2000
Hardening Measure 1
Eliminated Edges: Edge x1
Eliminated Path Count: 15
Cost: 7000
Hardening Measure 4
Eliminated Edges: Edge x4
Eliminated Path Count: 7
Cost: 5000
Intersecting Path Count = 1
Intersecting Path Count = 1
Intersecting Path Count = 2
Intersecting Path Count = 1
Intersecting Path Count = 5
Intersecting Path Count = 3
Intersecting Path Count = 4
Intersecting Path Count = 4

Fig. 9.1: Example hardening measure graph

```
function GenerateHardeningMeasureGraph(AttackGraph, AvailableMeasures, GoalStates)
    hardeningMeasureGraph $\leftarrow$ CreateEmptyGraph()
    for all measure $\in$ AvailableMeasures do
        measureNode $\leftarrow$ CreateMeasureNode()
        measureNode.pathCount $\leftarrow 0$
        measureNode.cost $\leftarrow$ measure.getCost()
        hardeningMeasureGraph.addNode(measureNode)
    end for
    for all goalState $\in$ GoalStates do
        measuresStack $\leftarrow$ CreateEmptyStack()
        PerformDFSOnAttackGraph(goalState,
            hardeningMeasureGraph, measuresStack)
    end for
    return hardeningMeasureGraph
end function

procedure PerformDFSOnAttackGraph(CurrentNode, HardeningMeasureGraph,
MeasuresStack)
    isExploitNode $\leftarrow$ IsExploitNode(CurrentNode)
    for all incomingEdge $\in$ currentNode.incomingEdges() do
        if isExploitNode then
            measureList $\leftarrow$ FindMeasuresNegatingEdge(incomingEdge)
            UpdateHardeningMeasureGraph(HardeningMeasureGraph,
                measureList, MeasuresStack)
        end if
        sourceNode $\leftarrow$ incomingEdge.getSourceNode()
        if isExploitNode then
            MeasuresStack.push(measureList)
        end if
        PerformDFSOnAttackGraph(sourceNode,
            HardeningMeasureGraph, MeasuresStack)
        if isExploitNode then
            MeasuresStack.pop()
        end if
    end for
end procedure

procedure UpdateHardeningMeasureGraph(HardeningMeasureGraph, MeasureList,
MeasuresStack)
    for all measure $\in$ MeasureList do
        measure.pathCount + +
        for all stackedMeasureList $\in$ MeasuresStack do
            for all stackedMeasure $\in$ stackedMeasureList do
                edge $\leftarrow$ FindMeasureEdge(HardeningMeasureGraph, measure, stackedMeasure)
                if edge == null then
                    edge $\leftarrow$ CreateMeasureEdge(measure, stackedMeasure)
                    edge.intersectingPathCount $\leftarrow 0$
                    HardeningMeasureGraph.addEdge(edge)
                end if
                edge.intersectingPathCount + +
            end for
        end for
    end for
    for all stackedMeasureList $\in$ MeasuresStack do
        for all stackedMeasure $\in$ stackedMeasureList do
            stackedMeasure.pathCount + +
        end for
    end for
end procedure
```

Fig. 9.2: Generating the hardening measure graph

The function for the generation of a hardening measure graph is shown in Figure 9.2. It first creates the hardening measure nodes and adds them to the empty hardening measure graph. Then, it performs a modified depth-first search (DFS) algorithm on the input attack graph starting from each goal security state (goal attacker privilege). The search is performed by the recursive PerformDFSOnAttackGraph() procedure. During the search, the edges between the hardening measures on the hardening measure graph are created. The list of the hardening measures negating the incoming edges to the exploit nodes on the currently traversed attack path are stored in $MeasuresStack$ during the search.

The hardening measures that negate the current incoming edge are found by calling the function FindMeasuresNegatingEdge(). The exact internal operation of this function is specific to the contents of the hardening measures and is out of the scope of this work. From an external point of view, it is responsible for identifying the edges negated by each hardening measure, caches the identified edges and uses the cached information to return the set of hardening measures that negate a given edge. Then, an edge from each of these hardening measures to each stacked hardening measure in $MeasuresStack$ is created, if it does not exist, by the UpdateHardeningMeasureGraph() function call. The intersection path counts of these edges are incremented by 1, as well as the eliminated attack path counts for these hardening measures and stacked measures.

A call to the recursive PerformDFSOnAttackGraph() function traverses each attack graph edge that are reachable from the currently processed goal state. This traversal is performed in the reverse direction of the attack graph edges. The number of traversals over an edge e in the attack graph is limited by the number of edges residing between the edge e and the goal states. This gives an upper bound of $O(E)$, where E denotes the number of the edges in the attack graph. Therefore, in total, $O(E^2)$ edge traversals are performed by this function. The function UpdateHardeningMeasureGraph() has $O(HG)$ time complexity, where H denotes the number of available hardening measures and G denotes the maximum number of elements that can be contained by $MeasuresStack$ at any time. G is bounded by $O(HN)$, where N denotes the number of the nodes in the attack graph. Therefore, the time complexity of a call to the recursive PerformDFSOnAttackGraph() function can be computed as $O(E^2H^2N)$. Finally, the time complexity of the GenerateHardeningMeasureGraph() function becomes $O(E^2H^2N^2)$, since the number of the goal states is bounded by $O(N)$.

9.2 Finding the Optimal Solution

The simulated annealing optimization algorithm is applied to find the near optimal hardening measure set(s) (optimal solution(s)). The simulated annealing-based optimization process utilizes the hardening measure graph to guide the generation process of the candidate solutions and compute the scores for the candidate solutions. Each candidate solution generated during the execution of the simulated annealing is a set of a number of disjoint hardening measures that can be applied by the target network's security administrator. At the end of the optimization process, an adjustable number of near-optimal solutions can be obtained. The objective function considers two optimization criteria:

- Minimize the total cost of the applied hardening measures in the solution.
- Maximize the number of the eliminated attack paths to the goal states on the attack graph.

The score for a candidate solution is computed according to the two optimization criteria as follows:

$$Score(solution) = \alpha * T + \beta * E \tag{9.1}$$

$$T = 1 - (Cost(solution) / \sum_{i=0}^{m} Cost(measure_i)) \tag{9.2}$$

$$E = Eliminatedpathcount(solution)/p \tag{9.3}$$

where α and β are adjustable weights of the two optimization criteria. $Cost(solution)$ is the cost of the candidate solution computed by summing the costs of the hardening measures contained in the candidate solution. m is the total number of the hardening measures available to the network administrator, p is the total number of the attack paths to the goal states in the input attack graph. $Eliminatedpathcount(solution)$ gives the number of the eliminated attack paths to the goal states in the input attack graph, when the *solution* is solely applied. It is computed by using the information in the hardening measure graph as shown in Figure 9.3. First, the total number of the eliminated attack paths for the hardening measures in the candidate solution is computed by using the corresponding information in the

```
function ComputeEliminatedAttackPathCount($HardeningMeasureGraph, CandidateSolution$)
    $pathCount \leftarrow 0$
    for all $hardeningMeasure \in CandidateSolution$ do
        $pathCount \leftarrow pathCount +$
            $hardeningMeasure.pathCount$
        $incomingEdges \leftarrow$
            FindIncomingEdges($hardeningMeasure$,
            $HardeningMeasureGraph$)
        for all $incomingEdge \in incomingEdges$ do
            $sourceMeasure \leftarrow incomingEdge.getSource()$
            if $CandidateSolution$ contains($sourceMeasure$) then
                $pathCount \leftarrow pathCount -$
                    $incomingEdge.intersectingPathCount$
            end if
        end for
    end for
    return $pathCount$
end function
```

Fig. 9.3: Computing eliminated attack path count for a candidate solution

hardening measure nodes. Then, the intersecting path counts between any two hardening measures in the candidate solution are subtracted from this amount. The intersecting path counts are found by using the information in the corresponding edges of the hardening measure graph.

We use the simulated annealing algorithm [22] to perform the optimization, starting with a random, accepted initial solution. During the simulated annealing, candidate solutions are tried one after the other to find the solution with the maximum score. If the score of the currently tried candidate solution is greater than the score of the last accepted solution, the candidate solution is accepted. Otherwise, the candidate solution is accepted according to some probability value, which is dynamically changing with a temperature value. As commonly applied in simulated annealing, this temperature is kept high at the start of the algorithm and even a candidate solution with a score much lower than the score of the last accepted solution may be accepted. The temperature is then decreased slowly during the algorithm, eventually allowing only the candidate solutions with a score greater than or slightly lower than the score of the last accepted solution being accepted.

The candidate solutions tried during the optimization process form a tree whether they are accepted or not. The edges of this tree represent the trial order of the candidate solutions. This tree actually comprises the state space. As stated in [22], it is crucial in simulated annealing to design the candidate generation (selection) function in such a way that the candidates that have the potential of being local optima are placed in shallower portions in the candidate tree. By this way, such local optima candidates are evaluated at the early stages of the algorithm with high temperature and they can be separated from each other. If this principle is not followed, there is a high chance for the simulated annealing to be caught in local optima at the later stages of the algorithm and not be able to explore better solutions.

We introduce a customized candidate solution selection function for the simulated annealing algorithm. The intuition at the core of this function can be explained as follows:

- If the majority of the attack paths eliminated by a hardening measure are also eliminated by the other already-applied hardening measures in the candidate solution, then the application of this hardening measure can provide additional benefit only if the cost of the measure is very low in comparison with the other measures.
- Conversely, if a hardening measure has a small number of intersecting eliminated attack paths with the other already-applied measures, then the application of this hardening measure has greater potential to provide additional benefit, if its cost is not too high.
- At the start of the simulated annealing algorithm where the temperature is high, we form the candidate solutions each of which contains hardening measures that have a small number of intersecting eliminated attack paths. This way, we try to evaluate all the potential local optima regions in the state space as early as possible.
- At the later stages of the algorithm where the temperature is low, we perform fine-tuning by allowing hardening measures in a candidate solution that have high number of intersecting eliminated attack paths, by also taking the cost of these hardening measures into account.

The proposed candidate selection (generation) function is illustrated in Figure 9.4. At the early stages of the optimization process, the candidate solutions are composed of hardening measures that have less number of common (intersecting) eliminated attack paths and that do not incur very high cost. By using $Threshold1$, the following points are ensured in forming the next candidate solution, after selecting a random hardening measure from all the available ones:

- If a selected measure has a number of common eliminated attack paths with the measures in the candidate solution that is greater than $Threshold1$, then:

```
$Threshold1 \leftarrow 0.1$
$Threshold2 \leftarrow 0.01$
$CostRatioThreshold \leftarrow 1.5$
    $StepCountForThresholdValueChange \leftarrow 100$
**function** GenerateCandidate($LastAcceptedSolution, HardeningMeasureGraph$)
        ▷ Last accepted solution in the optimization process and hardening measure graph are inputs
    $candidateSolution \leftarrow$ **Copy**($LastAcceptedSolution$)
    $availableHardeningMeasures \leftarrow HardeningMeasureGraph.getNodes()$
    $measuresTotalCost \leftarrow$ **TotalCost**($availableHardeningMeasures$)
    $shuffledHardeningMeasures \leftarrow$ **Shuffle**($availableHardeningMeasures$)
    $random1 \leftarrow$**GenerateRandomNumber**($shuffledHardeningMeasures.size()$)
    **for** i:=0 **do** $random1$
        $hardeningMeasure \leftarrow shuffledHardeningMeasures[i]$
        $maxOutgoingEdge \leftarrow$ **FindOutgoingEdgeWithMaximumValue**($hardeningMeasure, candidateSolution$)
        $intersectingPathCountRatio \leftarrow 0$
        $intersectingPathCountRatio \leftarrow maxOutgoingEdge.intersectingPathCount/$
            $hardeningMeasure.pathCount$
        **if** $intersectingPathCountRatio > Threshold1$ **then**
                ▷ At the start of the algorithm, eliminate hardening measures with a high number of intersecting eliminated attack paths. At later stages, apply this kind of measures for fine-tuning using the next else clause.
            **if** $candidateSolution$ **contains**($hardeningMeasure$) **then**
                **Remove**($hardeningMeasure, candidateSolution$)
            **else**
                $targetMeasure \leftarrow maxOutgoingEdge.getTargetNode()$
                **Add**($hardeningMeasure, candidateSolution$)
                **Remove**($targetMeasure, candidateSolution$)
            **end if**
        **else**
                ▷ At the start of the algorithm, apply hardening measures with a small number of intersecting eliminated attack paths. At later stages, eliminate this kind of measures to decrease the wasted optimization steps and reserve steps for fine-tuning using the next else clause.
            **if** $intersectingPathCountRatio > Threshold2$ **then**
                **if** $hardeningMeasure.cost/measuresTotalCost < CostRatioThreshold$ **then**
                    **Add**($hardeningMeasure, candidateSolution$)
                **else**
                    $random2 \leftarrow$**GenerateRandomNumber**($1.0$)
                    **if** $random2 > 0.5$ **then**
                        **Add**($hardeningMeasure, candidateSolution$)
                    **end if**
                **end if**
            **else**
                **Remove**($hardeningMeasure, candidateSolution$)
            **end if**
        **end if**
    **end for**
    **if** total number of optimization steps is divisible by $StepCountForThresholdValueChange$ **then**
        $Threshold1 \leftarrow Threshold1 + 0.1$
        $Threshold2 \leftarrow Threshold2 + 0.01$
        $CostRatioThreshold \leftarrow CostRatioThreshold - 0.1$
    **end if**
    **return** $candidateSolution$
**end function**
```

Fig. 9.4: Candidate selection (generation) function used during the simulated annealing algorithm

- if it already exists in the candidate solution, it is removed from the candidate solution.
- if it does not already exist in the candidate solution, it is added to the candidate solution, and the measure that has the maximum number of common attack paths with the selected measure is removed from the candidate solution. This is performed to form candidate solutions composed of different hardening measures with small number of common attack paths.

The value of $Threshold1$ is increased with the number of optimization steps. So, at the later stages of the algorithm, its effects gradually disappear. At the early stages of the optimization process, $Threshold2$ is used to increase the addition of hardening measures in the candidate solutions that have a small number of intersecting eliminated attack paths. Its value is kept as nearly one-tenth of the value of $Threshold1$ and is also increased with time, however with a pace slower than that of $Threshold1$.

With the increase of $Threshold2$ at the later stages of the algorithm, the elimination of the hardening measures that have small number of intersecting eliminated attack paths increases. Also at the later stages of the algorithm, the addition of very low-cost hardening measures into the candidate solutions that have large number of intersecting eliminated attack paths increases. This facilitates just fine-tuning the candidate solutions at the later sages of the algorithm. The application and removal of hardening measures according to their intersecting path counts during the earlier and later stages of the algorithm are illustrated for a small scenario in Figure 9.5 and Figure 9.6 in order. (It is assumed that the selected hardening measures shown at the left side of the figure have not already existed in the candidate solution.)

The value of $Threshold2$ is used as the basis for the addition (application) of a hardening measure in the candidate solution. A hardening measure is applied, if the ratio of its cost to the average cost of all the available measures is below a specific threshold named $CostRatioThreshold$. The value of $CostRatioThreshold$ is decreased with the number of optimization steps. Therefore, the measures with small number of intersecting attack paths are applied at the start of the algorithm, if they do not have very high cost. Also, the measures with large number of intersecting attack paths are added to the candidate solutions at the later stages of the algorithm, if they have low cost (the value of $CostRatioThreshold$ is low at the later stages of the algorithm.). If the measures do not satisfy these cost restrictions, they can be applied with random probability ($random2$).

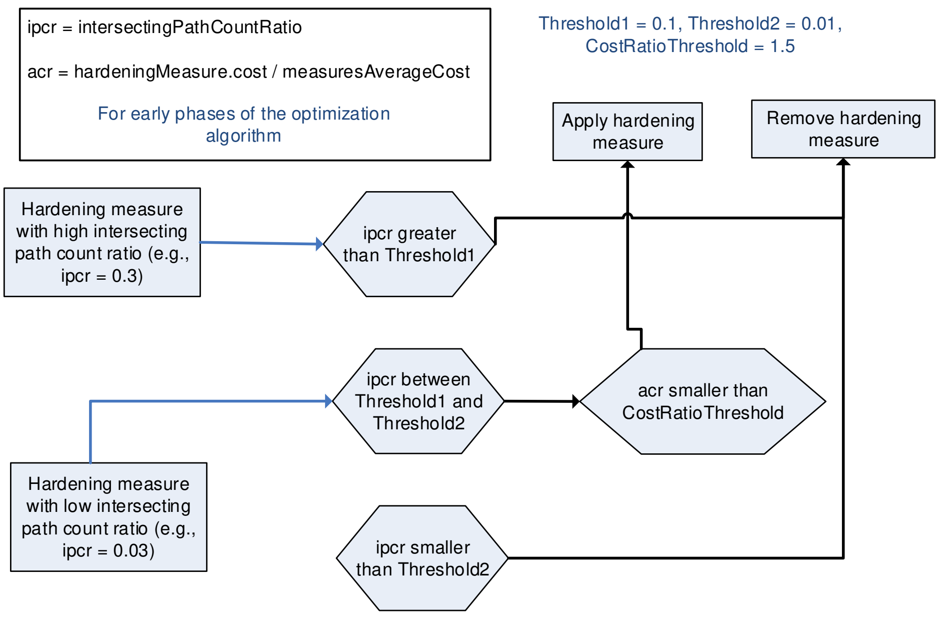

ipcr = intersectingPathCountRatio
acr = hardeningMeasure.cost / measuresAverageCost
For early phases of the optimization algorithm
Threshold1 = 0.1, Threshold2 = 0.01, CostRatioThreshold = 1.5
Apply hardening measure
Remove hardening measure
Hardening measure with high intersecting path count ratio (e.g., ipcr = 0.3)
ipcr greater than Threshold1
ipcr between Threshold1 and Threshold2
acr smaller than CostRatioThreshold
Hardening measure with low intersecting path count ratio (e.g., ipcr = 0.03)
ipcr smaller than Threshold2

Fig. 9.5: Example of application and removal of hardening measures during early phases of the optimization algorithm

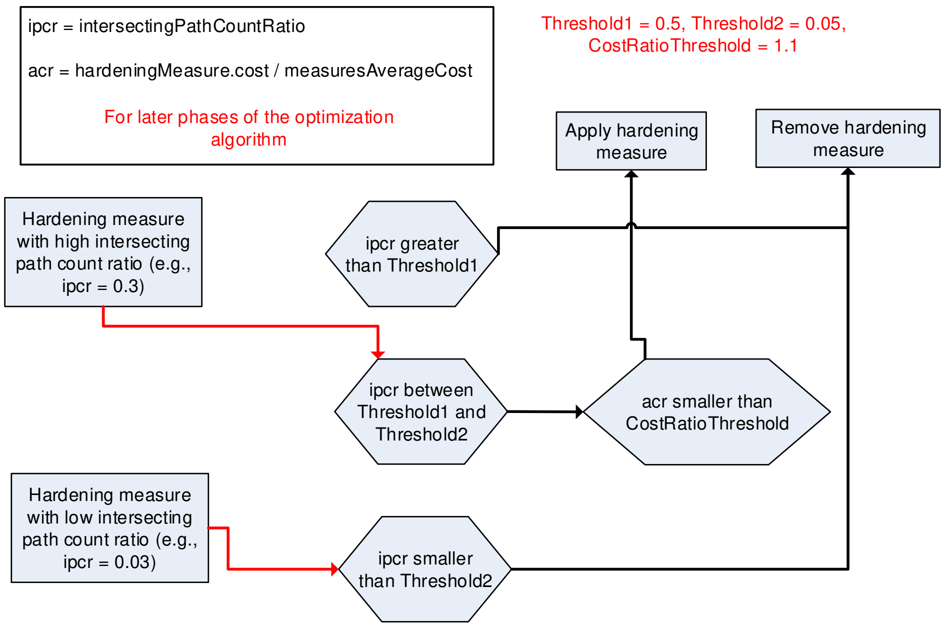

ipcr = intersectingPathCountRatio
acr = hardeningMeasure.cost / measuresAverageCost
For later phases of the optimization algorithm
Threshold1 = 0.5, Threshold2 = 0.05, CostRatioThreshold = 1.1
Apply hardening measure
Remove hardening measure
Hardening measure with high intersecting path count ratio (e.g., ipcr = 0.3)
ipcr greater than Threshold1
ipcr between Threshold1 and Threshold2
acr smaller than CostRatioThreshold
Hardening measure with low intersecting path count ratio (e.g., ipcr = 0.03)
ipcr smaller than Threshold2

Fig. 9.6: Example of application and removal of hardening measures during later phases of the optimization algorithm

In summary, at the start of the algorithm, we form a candidate solution from the hardening measures that are disjoint in terms of the eliminated attack paths. This way, at the beginning, we try to evaluate and compare all the candidate solutions that have a potential of being located close to the distinct local optimal solutions in the state space. At the later stages of the algorithm, simulated annealing allows less exploration in the solution state space, as the temperature is decreased. We utilize this fact by fine-tuning the solutions with the application of additional, very low-cost hardening measures that may have high number of intersecting eliminated attack paths with other already-applied hardening measures. Thus, we avoid wasting time with dense exploration in the state space at the later stages of the algorithm, since the algorithm would reject the majority of the candidate solutions in this case.

9.3 A Small Case Study

We explain the flow of the proposed candidate generation approach with a small case study in this section. The sample attack graph shown in Figure 9.7 is used as the input attack graph for the optimization process. The goal security states are determined as the bottommost two attacker privileges on IP addresses *75.62.3.33* and *75.62.3.35* in order. The hardening measures are the ones given in the computed hardening measure graph shown in Figure 9.8.

Each hardening measure node stores information about its cost, the eliminated attack graph edges and the count of the eliminated attack paths after its application. For instance, hardening measure 1 eliminates edge 10 on the attack graph. The attack paths on the attack graph eliminated by this measure are then as follows:

- (vulnerability exploit 1, vulnerability exploit 3)
- (vulnerability exploit 2, vulnerability exploit 3)
- (vulnerability exploit 3)

It should be noted that each attack path is a sequence of vulnerability exploits on a specific path on the attack graph and the last vulnerability exploit on the sequence must have an edge to any one of the goal security states. As another example, hardening measure 4 eliminates edge 2 and the following attack paths on the attack graph:

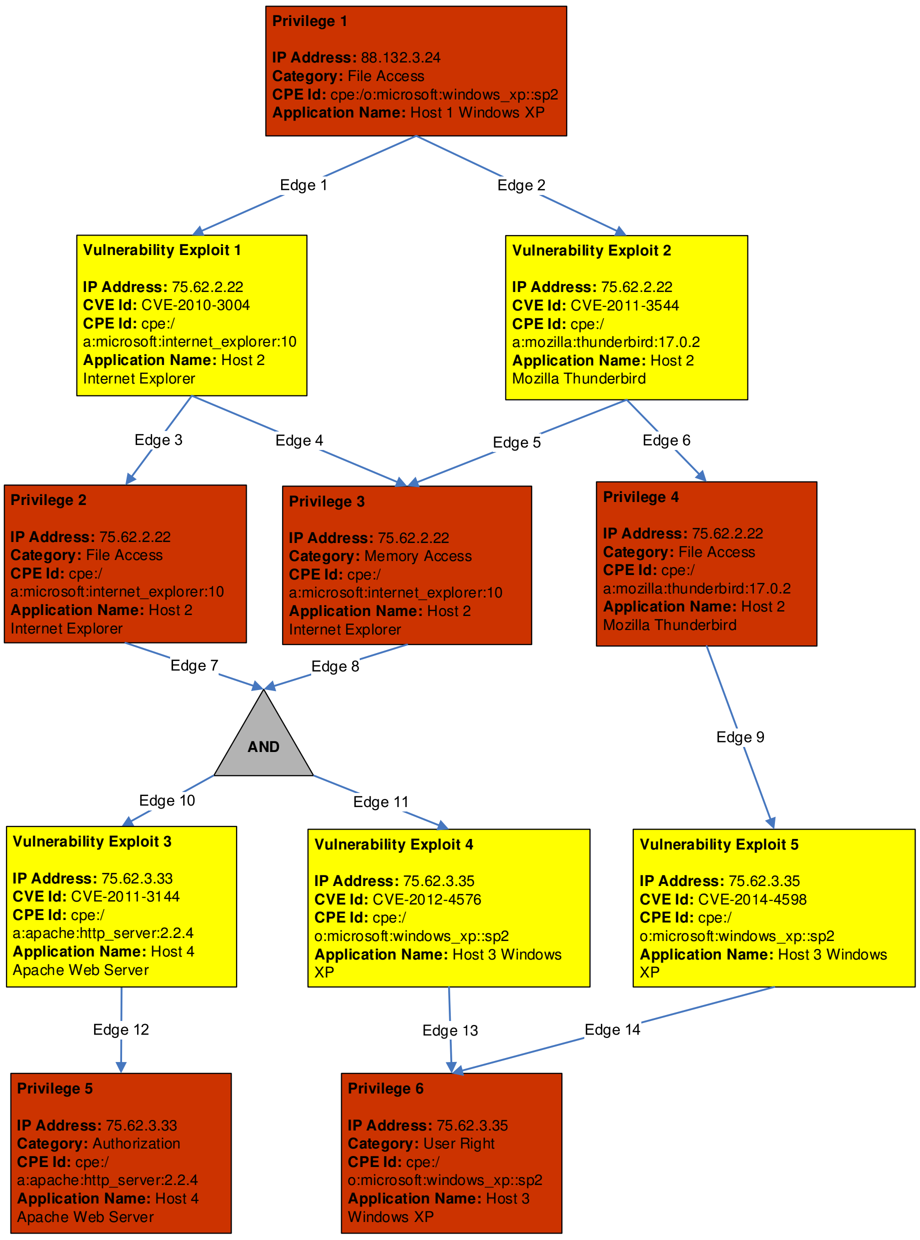

Privilege 1
IP Address: 88.132.3.24
Category: File Access
CPE Id: cpe:/o:microsoft:windows_xp::sp2
Application Name: Host 1 Windows XP
Edge 1
Edge 2
Vulnerability Exploit 1
IP Address: 75.62.2.22
CVE Id: CVE-2010-3004
CPE Id: cpe:/a:microsoft:internet_explorer:10
Application Name: Host 2 Internet Explorer
Vulnerability Exploit 2
IP Address: 75.62.2.22
CVE Id: CVE-2011-3544
CPE Id: cpe:/a:mozilla:thunderbird:17.0.2
Application Name: Host 2 Mozilla Thunderbird
Edge 3
Edge 4
Edge 5
Edge 6
Privilege 2
IP Address: 75.62.2.22
Category: File Access
CPE Id: cpe:/a:microsoft:internet_explorer:10
Application Name: Host 2 Internet Explorer
Privilege 3
IP Address: 75.62.2.22
Category: Memory Access
CPE Id: cpe:/a:microsoft:internet_explorer:10
Application Name: Host 2 Internet Explorer
Privilege 4
IP Address: 75.62.2.22
Category: File Access
CPE Id: cpe:/a:mozilla:thunderbird:17.0.2
Application Name: Host 2 Mozilla Thunderbird
Edge 7
Edge 8
AND
Edge 9
Edge 10
Edge 11
Vulnerability Exploit 3
IP Address: 75.62.3.33
CVE Id: CVE-2011-3144
CPE Id: cpe:/a:apache:http_server:2.2.4
Application Name: Host 4 Apache Web Server
Vulnerability Exploit 4
IP Address: 75.62.3.35
CVE Id: CVE-2012-4576
CPE Id: cpe:/o:microsoft:windows_xp::sp2
Application Name: Host 3 Windows XP
Vulnerability Exploit 5
IP Address: 75.62.3.35
CVE Id: CVE-2014-4598
CPE Id: cpe:/o:microsoft:windows_xp::sp2
Application Name: Host 3 Windows XP
Edge 12
Edge 13
Edge 14
Privilege 5
IP Address: 75.62.3.33
Category: Authorization
CPE Id: cpe:/a:apache:http_server:2.2.4
Application Name: Host 4 Apache Web Server
Privilege 6
IP Address: 75.62.3.35
Category: User Right
CPE Id: cpe:/o:microsoft:windows_xp::sp2
Application Name: Host 3 Windows XP

Fig. 9.7: Sample input attack graph for the case study

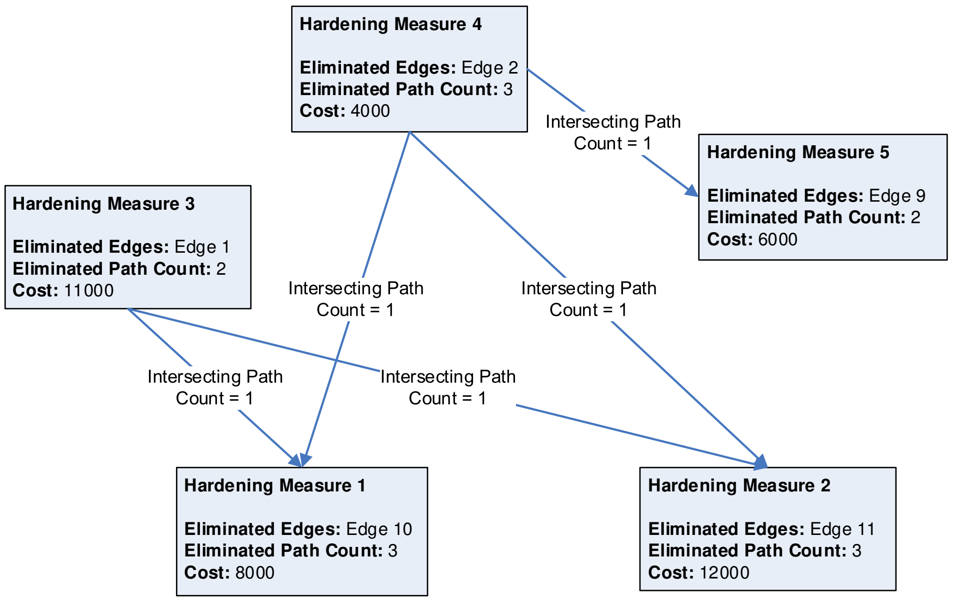


Fig. 9.8: Hardening measure graph for the case study

- (vulnerability exploit 2, vulnerability exploit 3)
- (vulnerability exploit 2, vulnerability exploit 4)
- (vulnerability exploit 2, vulnerability exploit 5)

At the earlier stages of the optimization process, the candidate solutions are formed from the hardening measures that have small number of common eliminated attack paths. This is accomplished by using the $Threshold1$ and $Threshold2$ thresholds. For instance, if we assume that the initial candidate solution is the set of all available hardening measures, then by using $Threshold1$, we eliminate the measures with large number of common eliminated attack paths in the following candidates. If we assume that the initial candidate solution is the set of not all but some hardening measures, the same is again true. In this case, by using $Threshold2$, we also add the measures that have small number of common eliminated attack paths in the following candidates.

At the earlier stages of the optimization process, it is more likely to obtain candidate solutions composed of the nodes not directly linked in Figure 9.8, e.g., we can form the below candidate solutions with higher likelihood:

- hardening measure 1, hardening measure 2

- hardening measure 1, hardening measure 2, hardening measure 5
- hardening measure 3, hardening measure 4
- hardening measure 3, hardening measure 5
- hardening measure 1, hardening measure 5

At the later stages of the algorithm, $Threshold2$ is used to add to the candidate solutions the hardening measures having large number of common eliminated attack paths with the measures in the candidate solutions. The purpose of this is to fine-tune the currently obtained near-optimal solution with the measures that can eliminate additional (may be a few) attack paths with very low cost. For example, the hardening measure set $\{1, 5, 4\}$ can be generated as a candidate solution at the later stages, with the addition of hardening measure 4 to the initial set of $\{1, 5\}$. In this example, we can gain one extra eliminated attack path with the addition of hardening measure 4 with relatively low cost. The eliminated attack path in this case is the one that uses vulnerability exploit 2 and vulnerability exploit 4 on the attack graph.

9.4 Experiments

The main aim of the experiments described in this section is to show the benefits of the proposed candidate selection function applied during the optimization algorithm. The benefits appear in the form of substantially decreased time required for the optimization algorithm to find the near optimal hardening solutions. We have carried out extensive experimentation with varying topologies, exploits and hardening measures as inputs to evaluate and verify our approach. For clarity of presentation, we provide a representative example network depicted in Figure 9.9, which forms the basis of the results provided in this section. However, the conclusions drawn are general and not only specific to this example network. The experiments are performed in a single PC with Intel Corei7-2620M 2.70GHz CPU and 8GB RAM.

The hosts in the network can have various software with a number of vulnerabilities defined in NVD [50]. The software configuration for the hosts and the filtering rules applied by the routers can be changed across the experiments in order to obtain different attack graphs. A full or partial attack graph can

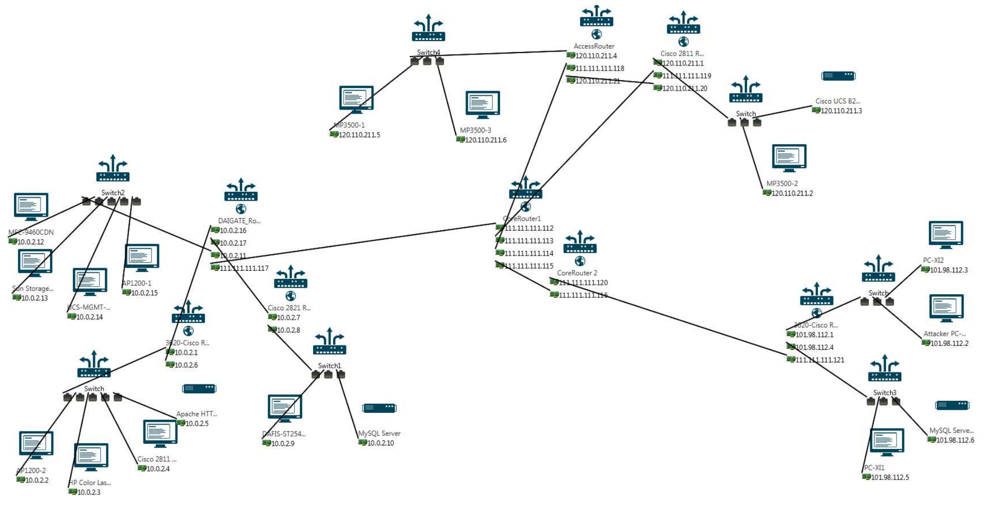


Fig. 9.9: Network used for the experiments

be given as input to the proposed method to find the near optimal network hardening solution. A sample attack graph utilized in the experiments is shown in Figure 9.10. Each area shaded in light blue indicates a network host, which can be collapsed or expanded to simplify the visualization of the attack graphs. Red nodes indicate security states (attacker privileges), orange nodes indicate weakness exploits (vulnerability exploits in the figure) and black nodes indicate conjunction (AND) relations.

The weakness exploit nodes in the attack graph are related with 12 different vulnerabilities. There are 22 exploit nodes on the attack graph having a total of 45 incoming edges. The average out-degree for the nodes in this attack graph is 2.

In our model, an enforceable hardening measure can have effects on the attack graph in terms of negating one or more incoming edges of the weakness exploit nodes. A hardening measure can patch various vulnerabilities and negate all the incoming edges of one or more weakness exploit nodes in the attack graph related to the patched vulnerabilities. It can also negate some specific incoming edges of an exploit node selectively by including filtering rules or according to the application location of the hardening measure inside the target network.

We provide the results of two sets of experiments, both based on the network shown in Figure 9.9. There is one goal security state used as input to the proposed optimization method, common in both sets of experiments. The goal security state is to gain the user access privilege on MySQL server

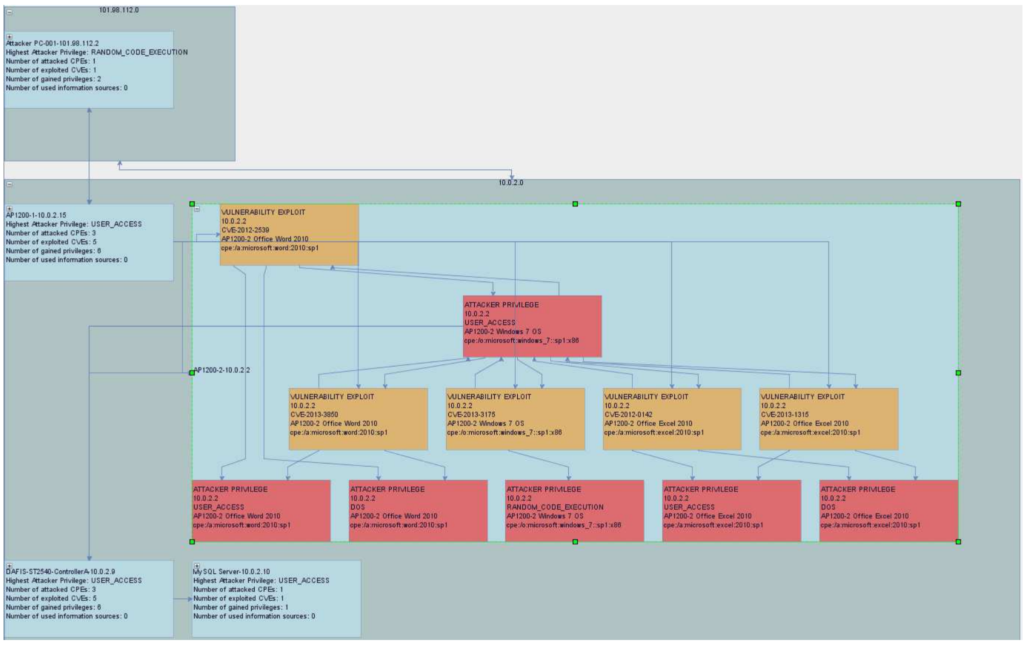

101.98.112.0
Attacker PC-001-101.98.112.2
Highest Attacker Privilege: RANDOM_CODE_EXECUTION
Number of attacked CPEs: 1
Number of exploited CVEs: 1
Number of gained privileges: 2
Number of used information sources: 0
10.0.2.0
AP1200-1-10.0.2.15
Highest Attacker Privilege: USER_ACCESS
Number of attacked CPEs: 3
Number of exploited CVEs: 5
Number of gained privileges: 6
Number of used information sources: 0
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2012-2539
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
USER_ACCESS
AP1200-2 Windows 7 OS
cpe:/o:microsoft:windows_7::sp1:x86
AP1200-2-10.0.2.2
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2013-3850
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2013-3175
AP1200-2 Windows 7 OS
cpe:/o:microsoft:windows_7::sp1:x86
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2012-0142
AP1200-2 Office Excel 2010
cpe:/a:microsoft:excel:2010:sp1
VULNERABILITY EXPLOIT
10.0.2.2
CVE-2013-1315
AP1200-2 Office Excel 2010
cpe:/a:microsoft:excel:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
USER_ACCESS
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
DOS
AP1200-2 Office Word 2010
cpe:/a:microsoft:word:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
RANDOM_CODE_EXECUTION
AP1200-2 Windows 7 OS
cpe:/o:microsoft:windows_7::sp1:x86
ATTACKER PRIVILEGE
10.0.2.2
USER_ACCESS
AP1200-2 Office Excel 2010
cpe:/a:microsoft:excel:2010:sp1
ATTACKER PRIVILEGE
10.0.2.2
DOS
AP1200-2 Office Excel 2010
cpe:/a:microsoft:excel:2010:sp1
DAFIS-ST2540-ControllerA-10.0.2.9
Highest Attacker Privilege: USER_ACCESS
Number of attacked CPEs: 3
Number of exploited CVEs: 5
Number of gained privileges: 6
Number of used information sources: 0
My SQL Server-10.0.2.10
Highest Attacker Privilege: USER_ACCESS
Number of attacked CPEs: 1
Number of exploited CVEs: 1
Number of gained privileges: 1
Number of used information sources: 0

Fig. 9.10: Attack graph used for the experiments

software installed on the host with the IP address 10.0.2.10.

We use the attack graph shown in Figure 9.10 in the first set of experiments and change the availability of different hardening measures as a parameter. In each of these experiments, we determine a specific number of hardening measures and mark them as available to the network security administrator. The attack graph edges negated by a hardening measure are determined randomly. Also, the costs for the hardening measures used in the experiments are selected randomly in a pre-determined range (5000-50000).

We compute the near optimal hardening solution for the used attack graph with both brute force and our simulated annealing-based optimization method for each of the 20 different experiments in the first set. We compare the results in terms of the execution time of both methods and the ratio of the score of the optimal hardening solution obtained by our method to the score of the best solution obtained by the brute force method. The scores for both methods are computed using Equation 9.1, with the values of α and β both set to 0.5. (We have also performed experiments for different values of α and β and the results for these experiments are comparable to the results of the experiments shown in this section.)

Table 9.1 shows the results of the first set of experiments with varying sets of available hardening measures. For each experiment, the count of the hardening measures available to the network security administrator and the number of the edges in the attack graph negated by each available hardening measure are given. The execution time values for the proposed optimization method accounts for the hardening measure graph computation time. The results of each experiment are actually obtained by averaging the results of a number of executions of the proposed optimization method with the selected available hardening measures defined for the experiment.

Tab. 9.1: Comparison of the proposed method with the brute force solution for the first set of experiments

Available Hardening Measure Count	Attack Graph Negated Edge Count	Brute Force Method	Our Optimization Method	
		Execution Time (ms)	Execution Time (ms)	Score Ratio
10	1	434	559	1.0
10	2	452	567	1.0
10	4	479	591	1.0
10	6	497	602	1.0
15	1	2942	758	1.0
15	2	3177	776	1.0
15	4	3358	793	0.998
15	6	3572	807	0.996
17	1	9866	861	1.0
17	2	10063	901	0.996
17	4	10768	937	0.986
17	6	11280	968	0.982
20	1	69448	996	0.993
20	2	71211	1059	0.988
20	4	79374	1092	0.979
20	6	82554	1137	0.972
25	1	N/A	1267	1.117
25	2	N/A	1301	1.086
25	4	N/A	1368	1.067
25	6	N/A	1392	1.055

These experiments show the effectiveness of the proposed candidate solution selection function used during the optimization process. When the number of the available hardening measures for a specific attack graph increases, our method can become more advantageous. For 20 available hardening measures with 1 negated edge count per measure, our method generates an optimal solution with a score that is 0.99 of the score of the optimal solution generated by the brute force method within a fraction (0.01) of the time required by the brute force method. For 25 available hardening measures, the brute force method can not even terminate in a reasonable time interval, indicated by the values of N/A in Table 9.1. The score accounted for the brute force method in these cases is the best score obtained, until manually terminating the execution after a specified time interval. Our simulated annealing-based optimization method is terminated, when the best solution is not improved after 2000 consecutive optimization steps (candidate evaluations).

When the number of the attack graph edges negated by one hardening measure increases for the same attack graph and same number of available hardening measures, finding the optimal solutions becomes more difficult; however, even in such cases, the score ratio of our method to the brute force solution remains above 0.97.

For the second set of experiments, we generate three additional attack graphs for the network shown in Figure 9.9, each with different initial security states (attacker privileges). The average out-degrees for the nodes in the additional attack graphs are chosen to be 4, 6 and 7. The number of exploit nodes in those are 28, 38, 47 respectively. The number of attack graph edges negated by each hardening measure is held constant at 4 for each of the 20 experiments in the second set. Table 9.2 compares the scores obtained by the brute force and the proposed method as in Table 9.1, where the used attack graphs are varied across the experiments. The attack graph used for each experiment is indicated by its degree. It should be noted that the attack graph with average out-degree 2 is the graph shown in Figure 9.10.

Similar to the earlier case, we get near optimal network hardening solutions with comparable scores to the best solutions generated by the brute force method almost hundred times faster for the second set of experiments. Our solutions' scores are between 0.95 and 1.0 of the corresponding best solutions for even very large and complex attack graphs (with average node out-degree 7). For 25 available hardening measures, the brute force method execution can not even terminate in a reasonable time frame, indicated again by N/A values in the corresponding entries of Table 9.2. For such cases, we record the best score obtained by the brute force method, before terminating its execution manually after a specified time interval.

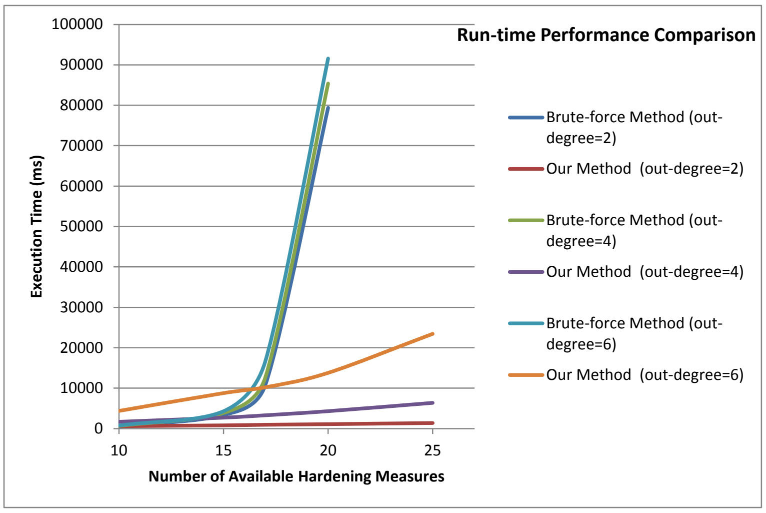
When the average node out-degree for the used attack graphs increases and other parameters remain the same, the accuracy (score ratio) of our method

Tab. 9.2: Comparison of the proposed method with the brute force solution for the second set of experiments

Available Hardening Measure Count	Attack Graph Node Average Out Degree	Brute Force Method	Our Optimization Method	
		Execution Time (ms)	Execution Time (ms)	Score Ratio
10	2	479	591	1.0
15	2	3358	793	0.998
17	2	10768	937	0.986
20	2	79374	1092	0.979
25	2	N/A	1368	1.067
10	4	655	1673	1.0
15	4	3875	2702	0.991
17	4	12663	3284	0.976
20	4	85371	4302	0.957
25	4	N/A	6358	1.037
10	6	731	4366	0.989
15	6	4286	8772	0.982
17	6	16648	10216	0.971
20	6	91559	13783	0.954
25	6	N/A	23433	1.021
10	7	787	8449	0.983
15	7	4426	18040	0.968
17	7	18912	22971	0.960
20	7	97262	31508	0.944
25	7	N/A	47824	1.012

decreases, since the interactions (e.g., common eliminated attack paths) among the available measures can increase substantially with the increasing number of edges in the used attack graphs and finding the optimal solutions becomes harder. However, even with 20 available hardening measures and attack graphs with an average node out-degree of 7, the score ratio of our method remains around 0.94. Also, when the average node out-degree for the used attack graphs increases, the number of available hardening measures for which our method outperforms the brute-force method in terms of execution time increases. For instance, for an attack graph with average node out-degree 4, our method outperforms the brute-force method, when the number of available hardening measures reaches 15. This number of hardening measures increases to 17 for an attack graph with average node out-degree 6. Figure 9.11 compares the run-time performance and accuracy of the brute-force and proposed method by changing the node average out-degree for the input attack graph and the number of available hardening measures. (We increase the attack graph size while increasing the node average out-degree.) The results of Table 9.2 are utilized to create this figure.

When the average node out-degree for the input attack graph (and also the attack graph size) increases, the run-time performance benefit gained by using the proposed method increases without sacrificing the optimality of the obtained solutions significantly. This benefit is more obvious, when the number of available hardening measures increases, from which the network security administrator should select the optimal ones.



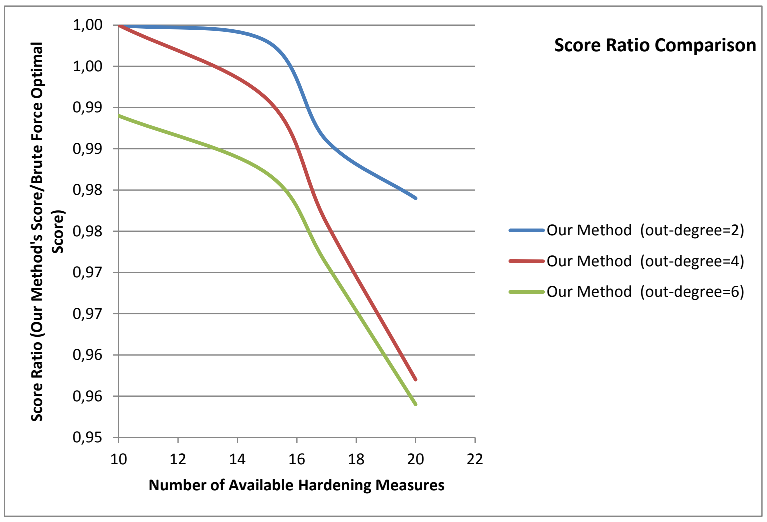


Fig. 9.11: Comparison of the run-time performance and accuracy (in terms of optimal solution score ratio) of the brute-force and proposed method

10 Conclusions and Future Work

In this thesis work, a system for detecting ongoing scenarios of attacks performed on a target network and responding to these attack scenarios is proposed. The system collects and processes the software logs in streaming mode in order to provide timely detection and response. Partial attack graphs are used to represent the attack scenarios. A taxonomy of the applied algorithms and models associated with the reachability analysis, attack graph modelling, and core building phases of the attack graph generation process is developed. The algorithms and models employed by the selected prominent works in the literature related to the attack graph generation process are described by categorizing them according to the presented classification scheme.

A distributed attack graph building algorithm employing a virtual shared memory approach over distributed agents and a reachability hyper-graph partitioning technique to determine the tasks and initial memory configuration of each agent is proposed. Experiments are performed to prove the performance gain obtained as a result of the application of this algorithm. The experiment results demonstrate that the distributed computation of attack graphs can be utilized to overcome the state space explosion problem occurring in the attack graph building process, when the number of hosts and vulnerabilities on the target network grows.

As a future work for the attack graph core building, one can try to apply heuristics-based search algorithms (A*, Iterative Deepening A*, D*, etc.) for the partial attack graph generation, after developing a set of appropriate network security domain-specific heuristics. The target privileges, determined by the network security administrators for partial attack graph generation as input, will be the goal states for the applied heuristics-based search algorithms. These algorithms generally find the shortest paths, however, they may be modified to give a range of critical attack paths to the specified target privileges. In order to use these search algorithms, a cost value must be assigned to each edge to appear on the resulting attack graph. These costs may indicate the likelihood of vulnerability exploits, if the edges on the attack graph represent vulnerability exploits.

Another future work for the attack graph core building may include the assessment of the advantages that can be gained by allowing duplicate privilege expansions at some points in the distributed attack graph building process to refrain from additional memory page transfers among the search agents. Therefore, at certain points, the principle of preserving coherence across the distributed search agents' memories can be relaxed. One other future work can be to develop methods using map-reduce-based approaches in order to solve the attack graph core building problem. Additionally, as a future work, some values may be assigned to the vulnerability exploits and information source usage vertices in an attack graph at run-time to eliminate the computation of the attack paths that are less likely to be utilized by a potential attacker because of difficulty of utilization, less damage level introduced to the target network, etc. In this case, instead of all the possible attack paths to the given target privileges, a number of paths advantageous for the attacker may be computed.

The primary logs (indicating vulnerability exploit attempts) are directly matched to the vulnerability exploit nodes on the computed attack graphs. The secondary logs are matched to the behavioural malware signatures that are generated by using an incremental, tree-based clustering method. The accuracy of the generated signatures are compared against the accuracy of the off-the-shelf anti-virus products' signatures. We have obtained better accuracy and performance by applying an incremental, tree-based malware clustering method to the behavioural signature generation problem. The proposed method clusters malware samples by processing their behavioural analysis reports downloaded from the web site *https://malwr.com*. The reports generated by ANUBIS [39] dynamic malware analyser for sample malicious files can also be processed by the proposed method. The behavioural artifacts for a sample malware are derived by processing its behavioural analysis report, each of which specifies an operating system (OS) operation performed on OS objects of a specific type (read registry key). The behavioural artifacts of all the malware samples are organized into a tree structure called malware tree whose nodes can be formed by merging a number of behavioural artifacts with the same OS operation and object type. The malware tree is traversed to create the behavioural malware signatures. If a new malware is received, it can be incrementally added to the malware tree. The time complexity of the creation of the malware tree is almost linear on the number of the malware samples used to construct the tree. The incremental addition of a new malware sample into the malware tree has constant time complexity.

As a future work, the proposed malware behavioural clustering system can be extended in such a way that allows the derivation of additional fields for the behavioural artifacts (OS operations and objects) of a malware. We currently determine the values of the fields of the behavioural artifacts based mainly on the name and absolute location of the files and registry keys, and the destination port and protocol for the network sockets. However, additional data such as the last access/modification time for the files, the symbolic links to the files, the information (e.g., byte count, packet count, specific flags) about the contents of the data transferred over the network sockets, the memory snapshots etc. that can be obtained by using various dynamic malware analysis tools can be utilized in the determination of the values of the fields for the behavioural artifacts and addition of new fields. The utilization of these kinds of additional data can provide for more accurate similarity computations among the behavioural artifacts of a malware. However, the volume of the information to be processed and the required CPU and memory resources will increase.

We use dynamically created and expanded partial attack graphs to represent the ongoing attack scenarios. The expansion direction (to be expanded nodes) of the partial attack graphs are determined according to the alarmed attacker privileges that are indicated by the primary and secondary logs. A primary log indicates a vulnerability exploit and the precondition privileges of this vulnerability exploit. A collection of secondary logs can indicate a number of attacker privileges, if and only if the logs can be matched to a behavioural malware signature as a collection. The malicious activities contained by the matched signature can be used to compute the attacker privileges indicated by the corresponding secondary logs. These are the privileges giving rise to the performance of the activities indicated by the corresponding secondary logs by the attacker.

In this thesis work, we also describe a new attack model to separate the security state nodes (mostly representing attacker privileges) and weakness exploit nodes in the attack graph and define the effects of an enforceable hardening measure as the negation of one or more incoming edges of the exploit nodes. As our main contribution, we propose a customized candidate selection function for the simulated annealing-based optimization algorithm to obtain the near optimal solutions much faster than the brute force algorithm. We evaluate the effectiveness of the proposed candidate selection function with the experiments, across which the used attack graph and the number of available hardening measures are varied.

As a future work, the proposed candidate selection function can be applied in the scope of a multi-objective optimization solution for the near optimal network hardening solution finding problem. They can be utilized to define effective, custom cross-over and mutation operators in the domain of the genetic algorithms to solve the multi-objective version of the problem. In addition, the proposed attack graph model can be improved to make it more flexible and effective, and it can be used to increase the runtime performance of the search-based optimal network hardening processes by allowing a hardening measure to negate the intermediate exploit nodes in the attack graph.

As a result, we have developed a system that is capable of detecting ongoing attack scenarios by using both the primary and secondary logs accurately. The system is also capable of recommending responses to the detected attack scenarios on time by utilizing a specific candidate selection function inside an optimization algorithm.

Bibliography

[1] M. Albanese, S. Jajodia, and S. Noel. Time-efficient and cost-effective network hardening using attack graphs. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2012.

[2] Symantec Altiris. Endpoint management powered by altiris technology, March 2015. `http://www.symantec.com/endpoint-management`.

[3] P. Ammann, J. Pamula, R. Ritchey, and J. Street. A host-based approach to network attack chaining analysis. In *Computer Security Applications Conference, 21st Annual*, Dec 2005.

[4] P. Ammann, D. Wijesekera, and S. Kaushik. 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, 2002. ACM.

[5] M. Apel, C. Bockermann, and M. Meier. Measuring similarity of malware behavior. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 891–898, Oct 2009.

[6] S. Bhatt, P.K. Manadhata, and L. Zomlot. The operational role of security information and event management systems. *Security Privacy, IEEE*, 12(5):35–41, Sept 2014.

[7] S. Bhattacharya, S. Malhotra, and S.K. Ghsoh. A scalable representation towards attack graph generation. In *Information Technology, 2008. IT 2008. 1st International Conference on*, pages 1–4, May 2008.

[8] R. Canzanese, K. Moshe, and S. Mancoridis. Toward an automatic, online behavioral malware classification system. In *Self-Adaptive and*

Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on, pages 111–120, Sept 2013.

[9] CAPEC. Common attack pattern enumeration and classification, March 2015. `http://capec.mitre.org/`.

[10] F. Chen, J. Su, and Y. Zhang. A scalable approach to full attack graphs generation. In *Engineering Secure Software and Systems*, volume 5429 of *Lecture Notes in Computer Science*, pages 150–163. Springer Berlin Heidelberg, 2009.

[11] F. Chen, C. Wang, Z. Tian, S. Jin, and T. Zhang. An atomic-domains-based approach for attack graph generation, 2009. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.308.4314&rep=rep1&type=pdf`.

[12] F. Chen, L. Wang, and J. Su. An efficient approach to minimum-cost network hardening using attack graphs. In *Information Assurance and Security, 2008. ISIAS '08. Fourth International Conference on*, pages 209–212, Sept 2008.

[13] Feng Cheng, A. Azodi, D. Jaeger, and C. Meinel. Security event correlation supported by multi-core architecture. In *IT Convergence and Security (ICITCS), 2013 International Conference on*, pages 1–5, Dec 2013.

[14] S. Cheung, U. Lindqvist, and M. W. Fong. Modeling multistep cyber attacks for scenario recognition. In *DISCEX (1)*, pages 284–292. IEEE Computer Society, 2003.

[15] MITRE Corporation. Common platform enumeration, January 2015. `https://cpe.mitre.org/`.

[16] Cuckoo. Automated malware analysis - cuckoo sandbox, November 2016. `https://cuckoosandbox.org/`.

[17] F. Cuppens and A. Miège. Alert correlation in a cooperative intrusion detection framework. In *IEEE Symposium on Security and Privacy*, pages 202–215. IEEE Computer Society, 2002.

[18] CVE. Common vulnerabilities and exposures, March 2015. `https://cve.mitre.org/`.

[19] CWE. Common weakness enumeration, March 2015. `http://cwe.mitre.org/`.

[20] M. Dacier, Y. Deswarte, and M. Kaaniche. Quantitative assessment of operational security: Models and tools, March 1996.

[21] R. Dewri, N. Poolsappasit, I. Ray, and D. Whitley. Optimal security hardening using multi-objective optimization on attack tree models of networks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 204–213, New York, NY, USA, 2007. ACM.

[22] R. W. Eglese. Simulated annealing: A tool for operational research. *European Journal of Operational Research*, 46(3):271–281, June 1990.

[23] B. Foo, Yu-Sung W., Y.-C. Mao, S. Bagchi, and E. Spafford. Adepts: adaptive intrusion response using attack graphs in an e-commerce environment. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 508–517, June 2005.

[24] M. Frigault and L. Wang. Measuring network security using bayesian network-based attack graphs. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 698–703, July 2008.

[25] M. Frigault, L. Wang, A. Singhal, and S. Jajodia. 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, 2008. ACM.

[26] Boris G. Integrated event management: Event correlation using dependency graphs, 1998.

[27] R. Gabriel, T. Hoppe, A. Pastwa, and S. Sowa. Analyzing malware log data to support security information and event management: Some research results. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference on*, pages 108–113, March 2009.

[28] GFILanGuard. Gfilanguard network security scanner and patch management, March 2015. `http://www.gfi.com/products-and-solutions/network-security-solutions/gfi-languard`.

[29] B. Han, Q. Wang, F. Yu, and X. Zhang. A vulnerability attack graph generation method based on scripts. In *Proceedings of the Third International Conference on Information Computing and Applications*, ICICA'12, pages 45–50, Berlin, Heidelberg, 2012. Springer-Verlag.

[30] B. Hirsch, T. Konnerth, and A. Hessler. Merging agents and services - the jiac agent platform. In *Multi-Agent Programming*, pages 159–185. Springer US, 2009.

[31] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 117–126, Dec 2009.

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

[33] T. Islam and L. Wang. A heuristic approach to minimum-cost network hardening using attack graph. In *New Technologies, Mobility and Security, 2008. NTMS '08.*, pages 1–5, Nov 2008.

[34] S. Jajodia and S. Noel. Topological vulnerability analysis. In *Cyber Situational Awareness*, volume 46 of *Advances in Information Security*, pages 139–154. Springer, 2010.

[35] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 343–348, New York, NY, USA, 1999. ACM.

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

[37] K. Kaynar and F. Sivrikaya. Distributed attack graph generation. *IEEE Transactions on Dependable and Secure Computing*, 13(5):519–532, Sept 2016.

[38] I. Kotenko and M. Stepashkin. Attack graph based evaluation of network security. In *Proceedings of the 10th IFIP TC-6 TC-11 International Conference on Communications and Multimedia Security*, CMS'06, pages 216–227, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] C. Kruegel, E. Kirda, M. P. Comparetti, U. Bayer, and C. Hlauschek. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 1 2009.

[40] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[41] G. Lodi, L. Aniello, Giuseppe A. Di L., and R. Baldoni. An event-based platform for collaborative threats detection and monitoring. *Inf. Syst.*, 39:175–195, January 2014.

[42] J. Ma and J. Sun. Optimal network hardening model based on parallel genetic algorithm. In *Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on*, pages 546–549, Aug 2012.

[43] J. Ma, Y. Wang, J. Sun, and X. Hu. A scalable, bidirectional-based search strategy to generate attack graphs. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2976–2981, June 2010.

[44] D. Man, B. Zhang, Y. Wu, W. Jin, and Y. Yang. A method for global attack graph generation. In *Networking, Sensing and Control, 2008. ICNSC 2008. IEEE International Conference on*, pages 236–241, April 2008.

[45] S. Marchal, J. Xiuyan, R. State, and T. Engel. A big data architecture for large scale security monitoring. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 56–63, June 2014.

[46] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec 2007.

[47] S. Nari and A. A. Ghorbani. Automated malware classification based on network behavior. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 642–647, Jan 2013.

[48] P. Ning, Y. Cui, and D. S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In Vijayalakshmi Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 245–254. ACM, 2002.

[49] P. Ning and D. Xu. Learning attack strategies from intrusion alerts. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 200–209, New York, NY, USA, 2003. ACM.

[50] NIST. National vulnerability database, March 2015. `http://nvd.nist.gov/`.

[51] Nmap. Nmap security scanner, March 2015. `http://nmap.org/`.

[52] S. Noel, M. Elder, S. Jajodia, P. Kalapa, S. O'Hare, and K. Prole. Advances in topological vulnerability analysis. In *Proc. of the 2009 Cybersecurity Applications & Technology Conf. for Homeland Security*, CATCH '09, pages 124–129, Washington, DC, USA, 2009.

[53] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, pages 86–, Washington, DC, USA, 2003. IEEE Computer Society.

[54] S. Noel, E. Robertson, and S. Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 350–359, Dec 2004.

[55] OpenVAS. Openvas open source vulnerability scanner and manager, March 2015. `http://www.openvas.org/`.

[56] OSVDB. The open source vulnerability database, March 2015. `http://osvdb.org/`.

[57] X. Ou, W. F. Boyer, and M. A. McQueen. 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, 2006. ACM.

[58] OVALdi. Ovaldi - an open-source local vulnerability assessment scanner, March 2015. `http://www.decalage.info/en/ovaldi`.

[59] X. Peng, J. H. Li, O. Xinming, L. Peng, and R. Levy. Using bayesian networks for cyber security analysis. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 211–220, June 2010.

[60] C. Phillips and L. P. Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, pages 71–79, New York, NY, USA, 1998. ACM.

[61] N. Poolsappasit, R. Dewri, and I. Ray. Dynamic security risk management using bayesian attack graphs. *IEEE Trans. Dependable Secur. Comput.*, 9(1):61–74, jan 2012.

[62] H. Razeghi Borojerdi and M. Abadi. Malhunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection. In *Computer and Knowledge Engineering (ICCKE), 2013 3th International eConference on*, pages 430–436, Oct 2013.

[63] Redseal. Redseal proactive network security, March 2015. `http://www.redsealnetworks.com/`.

[64] H. Ren, N. Stakhanova, and A. A. Ghorbani. An online adaptive approach to alert correlation. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'10, pages 153–172, Berlin, Heidelberg, 2010. Springer-Verlag.

[65] Retina. Retina network security scanner unlimited, March 2015. `http://www.beyondtrust.com/Products/RetinaNetworkSecurityScanner/`.

[66] K. Rieck, P. Trinius, C. Willems, and T. Holzaff. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668, dec 2011.

[67] R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 156–165, 2000.

[68] S. Roschke, F. Cheng, and C. Meinel. A new alert correlation algorithm based on attack graph. In *Proceedings of the 4th International Conference on Computational Intelligence in Security for Information Systems*, CISIS'11, pages 58–67, Berlin, Heidelberg, 2011. Springer-Verlag.

[69] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284, 2002.

[70] Oleg Sheyner and Jeannette Wing. Tools for generating and analyzing attack graphs. In F. S. De Boer, M. M. Bonsangue, S. Graf, and W. De Roever, editors, *Formal Methods for Components and Objects*, volume 3188 of *Lecture Notes in Computer Science*, pages 344–371. Springer Berlin Heidelberg, 2004.

[71] Skybox. Skybox risk control, March 2015. `http://www.skyboxsecurity.com/`.

[72] A. C. Squicciarini, G. Petracca, W. G. Horne, and A. Nath. Situational awareness through reasoning on network incidents. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 111–122, New York, NY, USA, 2014. ACM.

[73] A. Stotz and M. Sudit. Information fusion engine for real-time decision-making (inferd): A perceptual system for cyber attack tracking. In *Information Fusion, 2007 10th International Conference on*, pages 1–8, July 2007.

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

[75] G. Tedesco and U. Aickelin. Real-time alert correlation with type graphs. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 173–187, Berlin, Heidelberg, 2008. Springer-Verlag.

[76] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 Workshop on New Security Paradigms*, NSPW '00, pages 31–38, New York, NY, USA, 2000. ACM.

[77] Nessus Tenable. Nessus vulnerability scanner, March 2015. `http://www.tenable.com/products/nessus`.

[78] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer. A comprehensive approach to intrusion detection alert correlation. *IEEE Trans. Dependable Secur. Comput.*, 1(3):146–169, July 2004.

[79] V. Vianello, V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, R. Torres, R. Diaz, and E. Prieto. A scalable siem correlation engine and its application to the olympic games it infrastructure. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 625–629, Sept 2013.

[80] VirusTotal. Virus total, March 2015. `https://www.virustotal.com/`.

[81] L. Wang, M. Albanese, and S. Jajodia. Attack graph and network hardening. In *Network Hardening*, SpringerBriefs in Computer Science, pages 15–22. Springer International Publishing, 2014.

[82] L. Wang, M. Albanese, and S. Jajodia. Linear-time network hardening. In *Network Hardening*, SpringerBriefs in Computer Science, pages 39–58. Springer International Publishing, 2014.

[83] L. Wang, M. Albanese, and S. Jajodia. Minimum-cost network hardening. In *Network Hardening*, SpringerBriefs in Computer Science, pages 23–38. Springer International Publishing, 2014.

[84] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia. An attack graph-based probabilistic security metric. In *Proceeedings of the 22Nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 283–296, Berlin, Heidelberg, 2008. Springer-Verlag.

[85] L. Wang, A. Liu, and S. Jajodia. An efficient and unified approach to correlating, hypothesizing, and predicting intrusion alerts. In *Proceedings of the 10th European Conference on Research in Computer Security*, ESORICS'05, pages 247–266, Berlin, Heidelberg, 2005. Springer-Verlag.

[86] L. Wang, S. Noel, and S. Jajodia. Minimum-cost network hardening using attack graphs. *Comput. Commun.*, 29(18):3812–3824, November 2006.

[87] W. Wang and T. E. Daniels. A graph based approach toward network forensics analysis. *ACM Trans. Inf. Syst. Secur.*, 12(1):4:1–4:33, October 2008.

[88] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

[89] Y. Wu, B. Foo, Y. Mei, and S. Bagchi. Collaborative intrusion detection system (cids): A framework for accurate and efficient ids. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, pages 234–, Washington, DC, USA, 2003. IEEE Computer Society.

[90] A. Xie, G. Chen, Y. Wang, Z. Chen, and J. Hu. A new method to generate attack graphs. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 401–406, July 2009.

[91] A. Xie, Li. Zhang, J. Hu, and Z. Chen. A probability-based approach to attack graphs generation. In *Electronic Commerce and Security, 2009. ISECS '09. Second International Symposium on*, volume 2, pages 343–347, May 2009.

[92] T. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 199–208, New York, NY, USA, 2013. ACM.

[93] B. Yigit, G. Gur, and F. Alagoz. Cost-aware network hardening with limited budget using compact attack graphs. In *Military Communications Conference (MILCOM), 2014 IEEE*, pages 152–157, Oct 2014.

[94] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[95] H. Zhang, D. D. Yao, and N. Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 39–50, New York, NY, USA, 2014. ACM.

[96] Y. Zhao, Z. Wang, X. Zhang, and J. Zheng. An improved algorithm for generation of attack graph based on virtual performance node. In *Multimedia Information Networking and Security, 2009. MINES '09. International Conference on*, volume 2, pages 466–469, Nov 2009.