CAPABILITY MODEL BASED ALERT CORRELATION

by

NAVNEET KUMAR PANDEY

Department of Computer Science & Engineering

Submitted

in fulfillment of the requirements of the degree of

Master of Science by Research

to the

Indian Institute of Technology Delhi

September 2008

CERTIFICATE

This is to certify that the thesis titled "*Capability Model Based Alert Correlation*" being submitted by **Navneet Kumar Pandey** to the Indian Institute of Technology Delhi, for the award of the degree of Master of Science (Research) in Computer Science and Engineering, is a record of bona-fide research work carried out by him under my supervision. The work presented in this thesis has not been submitted to any other university or institute for the award of any other degree or diploma.

Prof. S. K. Gupta
Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Abstract

Most of the existing intrusion detection systems (IDS) often generate large numbers of alerts which contain numerous false positives and non relevant positives. Alert correlation techniques aim to aggregate and combine the outputs of single/multiple IDS to provide a concise and broad view of the security state of network. Capability based alert correlator uses notion of capability to correlate IDS alerts where capability is the abstract view of attack extracted from IDS alerts/alert. To make correlation process semantically correct and systematic, there is a need to identify the algebraic and set properties of capabilities. In this work, the potential algebraic properties of capability are identified in terms of operations, relations and inferences. These properties give better insight to understand the logical association between capabilities which are helpful in making the system modular. A variant of correlation algorithm is presented which uses these algebraic properties. To make these operations more realistic, existing capability model has been extended by adding time-based notion which helps to avoid temporal ambiguity between capability instances. We also propose Attack Capability Modeling language (ACML) used for capability model. It is a specification and description language that has been utilized to express the capability gained by attacker at each step in the intrusion process. These capabilities have been defined using the IDS alerts. The language also provides for the specification of compete attack scenarios in terms of capabilities of the intruder. This, in turn, helps to determine the state of the system in

terms of the extent of infiltration. ACML helps to avoid ambiguity in capability specifications while sharing among developers. We also propose Attack capability modeling framework (ACMF) which forms the basis of a capability model-based semi-automated alert correlation process, which has been used to detect and identify the attack scenarios from IDS alerts. Additionally, the language also has features for customizing the definitions of these structures as well as for customizing the correlation algorithm.

Acknowledgment

I lay this modest work of mine in the feet of Lord Buddha for HE showers HIS infinite wisdom and grace on a petty soul like me.

This report owes its existence to the efforts of many individuals who have left their indelible imprints on the succeeding pages. In presenting this work here, I avail the opportunity to express my thanks with deep sense of gratitude towards my project guide respected Prof. S. K. Gupta, Professor, Department of Computer Science and Engineering, Indian Institute of Technology, Delhi, who always provided moral support and encouraged me to keep working towards my goal. He always stimulated me and provided me with new opportunities to work upon.

I also want to acknowledge Prof. Saroj Kaushik for her lessons of hard work and motivation are second to none. The trust shown in me by Prof. K. K. Biswas to achieve my goals was the driving force behind this work.

For every work whether great or small, there are pillars that support it right from the start. I would like to thank Shaveta leekha and Jingmin Zhou for being the pillars of this work. I am thankful to all my friends and colleagues at Indian Institute of Technology, Delhi for they are the ones who made this tough ride an ever-cherishing experience. I would especially like to mention the names of Prem Piyush Goyal and Komal Pal for proof-reading this work, Gaurav Saxena for providing innovative solutions to various problems, Vikram Goyal and Thomas Kappler for helping me in defining this work in its initial period

and Ashwin S. Rao for his guidance for the brilliant software LATEX and guidance in general. The financial support provided by Advanced Information Systems Security Laboratory (AISSL) is duly acknowledged.
Lastly I would like to thank my parents, former teachers and friends who helped me in innumerable ways to become what I am at present.

Navneet Kumar Pandey

Contents

Chapter 1

Introduction

Even though sustained efforts by information system and network security experts to protect secure systems from exponentially increasing threats, continue the hackers tools now include technology that conventional security tools and services cannot sustain. Even critical systems and networks equipped with highly sophisticated security techniques are vulnerable to blended and multi-stage attacks which use stealth and intelligence to strategically compromise a target, escaping detection and penetrating the defences[**?**].

The security of an information system is compromised when an intrusion takes places. An intrusion can thus be defined as "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource"[**?**]. Nowadays, intrusions are generally associated with attacks launched over the network against remote systems. It is difficult to prevent such attacks by the use of classical information security technologies, such as authentication and cryptography, because system and application software typically contain exploitable weaknesses due to design and programming bugs. It is both technically difficult and economically expensive to build and maintain an information system that is free of security flaws and not susceptible to attacks.

1.1 Motivation

The surveillance and security monitoring of the network infrastructure is mostly performed using Intrusion Detection Systems (IDSs). Event streams are used by IDS in two different ways, according to two different paradigms: anomaly detection and misuse detection. In anomaly detection systems [**?**], [**?**], [**?**], [**?**]),historical data about a system activity and/or specifications of the intended behavior of users and applications are used to build a profile of the normal operation of the monitored system. The intrusion detection system then tries to identify patterns of activity that deviate from the defined profile. It actually analyzes audit traces in the search of a deviating behavior compared to a former behavior considered as normal by the system. The main advantage of this approach, which is the ability to detect previously unknown attacks, comes with a penalizing high number of bogus alerts known as false positives. Misuse detection systems take a complementary approach ([**?**], [**?**],[**?**],[**?**]). Misuse detection tools are equipped with a number of attacks descriptions (or signatures) that are matched against the stream of audit data and look for the evidence modeled attack. These approaches mainly detect modeled attacks, and in some cases variations of them. They have the advantage of being less prone to the generation of false positives, enabling therefore highly focused analysis of the audit data.

Most of the IDSs generate alerts or detect anomalies independent of the others, however some logical connections may exist between them. These alerts signify the attacks. Single step computer attacks are generally described in terms of the single vulnerability exploited in the attack, for example a buffer

overflow in sendmail, a denial-of-service by sending pings to a broadcast IP address, etc. With the increasing frequency of complex and coordinated, multi-stage attacks that take the advantage of various single step attacks on enterprizes and government systems, these single-point/ single step attacks remain of little significance in isolation. Intruders also use patterns of intrusion that are difficult to trace and identify. They frequently use several levels of indirection before breaking into target systems. These more sophisticated multi-stage attacks also have the capability to bypass the security mechanisms and hence motivates the need for better techniques of describing and identifying them. Consequently, methods to understand, predict and identify the attack scenarios are important challenges for computer security research.

Attack scenarios are typically described by sequence of actions the attacker perform to achieve some specific goal. These kinds of attack scenario descriptions are very useful in specifying the attack signatures which facilitate in detecting intrusions/attacks. But these descriptions lack in the ability to generalize beyond the stated scenario or to be utilized as a sub-goal in more complex attacks. An alternative way of describing attacks are uses model of attacks based on the requirements of the abstract components of the attack, where a complete attack is composed of multiple components. Each component requires certain capabilities that must exist for a particular instance of the component to be entailed. Each component may also provide specific capabilities to satisfy the requirements of some other components. The idea is that a series of the attacker's actions has time consecutive association. Specifically, an earlier attack enables or positively affects the later one. We can predict the next most likely step when one attack is achieved successfully. With this

method several unknown attacks can be described as no prior knowledge of complete scenario is required.

In order to minimize false positives(see section2.1.3) and to automatically construct attack scenarios from alerts, many alert correlation techniques have been proposed.

These correlation techniques comes under four categories. In the first category, correlation is done on the basis of feature similarity (e.g., Spice [**?**], the probabilistic alert correlation [**?**]). However they work well for correlating some alerts, but they are unable to identify the causal relationships between related alerts. In the second category(e.g., LAMBDA [**?**]), administrator specifies attack scenarios for the alert correlation. In some of these methods, to generate automatic attack scenarios data mining approach has been used. Apparently, these approaches can not detect unknown attack scenarios. Third category correlation techniques are based on require/provide model in which consequence of an alert satisfies the precondition of another alert. Fourth category belongs to the multiple security systems based approaches [**?**][**?**] in which log of different security tools such as IDSs, firewall log, router log, vulnerability scanners etc. are used for attack correlation [**?**].

Our work is motivated by Zhou et. al.[**?**] which uses capability model for attack correlation. This capability model uses capability as a basic building block and uses it for developing several algorithms in correlation based on alert abstraction and inference rules. Their work also shows that the approach is capable of handling missing attacks and is promising at alert fusion and correlation. However it does not discuss the algebraic operations and relations between capabilities. To make capability model correlation process systematic

and modular, it is essential to identify and understand algebraic properties of capability. This will also help to represent capabilities unambiguously.

As capability is an abstract term, therefore it is needed to customize the model according to network environment and other system preferences. Languages are the best tools to express these preferences. It formalizes the meaning of elements used in language and make it less ambiguous. Language also helps in making the correlation process modular and simple. This makes system easily understandable for even non security expert. This approach helps in facilitating the process flexibility and easy enhancement.

1.2 Objectives

The objectives of this thesis are as follows.

1. **Algebra for capability based attack correlation:** To make a mature capability model, we need to know basic characteristic of Capability in context of attack correlation process. These basic characteristics may include identification of algebraic properties of Capability.
2. **Capability based attack modeling language:** Capability is an abstract term therefore, there is a need of customization of capability. As language is the best way to express user's notion of capability and provides easy and feasible way for customization, it is a good vehicle for customizing capability and for modeling the attack scenarios in terms of capability.

1.3 Contribution

In this thesis, we propose a model of capability in term of algebraic structures that are later used to make correlation process modular. We identify the potential algebraic properties of capabilities in terms of operations, relations and inferences. These properties give better insight to understand the logical association between capabilities which are helpful in making the system modular and systematic. The work also presents variant of correlation algorithm by using these algebraic properties. To make these operations more realistic, existing capability model has been empowered by adding time-based notion which helps to avoid temporal ambiguity between capability instances.

We also propose a novel framework for modeling attack capability using a language. This framework semi-automates the correlation process based on capability model. The framework minimizes the human intervention which is required for processing IDS alerts for the purpose of identifying attack scenarios. The proposed Attack Capability Modeling Language(ACML) is a novel language which provides flexibility in setting up the security concerns as required for the customization of capability model. With the customized model, the system can also be adaptive to optimizations. ACML specifications also enable the utilization and reusability of the experience and knowledge of security professionals in the field of attack correlation.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the background detail and previous work in the area of attack correlation. We

also give brief overview of some recently developed attack language. Brief description about capability model is given in the related work section of this chapter. Chapter 3 contains the proposed capability model. This chapter also contains detail about Indirect and Direct capability, argument for the significance of time parameter and the overview of correlation process using the capability model. Chapter 4 contains the proposed algebra for capability based attack correlation. Chapter 5 provides the proposed Attack Capability Modeling Framework (ACMF) and Attack Capability Modeling Language (ACML). Chapter 6 concludes the this dissertation and outlines the future work.

Chapter 2

Background and Related Work

The major focus of the thesis is design of a systematic model of alert correlation and provide tool to represent this information so that it can be exchanged between security administrators without any ambiguity. This chapter provides a survey in the area of alert correlation. Second part of this chapter gives the overview of attack languages.

2.1 Intrusion Detection

Nowadays as more sensitive data is stored in computer, security is therefore one of the major issues. With the wide development of Internet and Network systems, Intrusion detection become a major threat of all types of organizations. Idea of Intrusion detection was introduced by James Anderson et. al. in the early 1980[**?**]. In their work they define intrusion detection as "Computer Security Threat Monitoring and Surveillance". Anderson defines an intrusion as "any unauthorized attempt to access, manipulate, modify, or destroy information, or to render a system unreliable or unusable". Intrusion detection attempts to detect these types of activities. Amoroso[**?**] defines intrusion detection as the process of identifying and responding to malicious activity targeted at computing and networking resources. An intrusion can thus be

defined as any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource [**?**].

An Intrusion Detection System (IDS) is a security tool that monitors computer systems and networks to detect and alarm on those intrusions that pass through protection mechanisms. Thus, upon receiving the alarms, administrators or automatic response systems can make quick response to those attacks to minimize their damage, and understand their behavior.

Intrusion detection systems can be classified in several ways. It is common to classify an IDS by the detection mode, the audit source, the usage frequency, and the response mechanism [**?**].

2.1.1 Classification of IDS based on Detection Method

Intrusion Detection System(IDS) techniques are classified into two classes based on their detection mechanism.

Misuse detection techniques Misuse detection techniques try to model the attacks on a system as patterns, and then systematically scan the system for the occurance of these patterns. This process involves a specific encoding of previous deemed intrusive or malicious behaviors and actions[**?**].

Misuse detection approaches include rule-based and state transition based systems. Rule-based detection compares the audit data with each encoded intrusion activity (rule) and its detection engine generates alarms whenever there is a match . MIDAS (Multics Intrusion Detection and

Alerting System) [?], Snort[?], and Bro[?] are the IDSs which belong to this category. Ghosh et. al.[?] further classify these into two categories i.e. expert systems[?],[?], model-based reasoning [?],[?].

State transition based detection [?][?] defines attack patterns with state transition diagrams in which nodes represent system states and edge represent actions causing state transitions. Detection engine maps computer state into these transition states to track the current security state of system. It generates alarms for security administrator whenever system reaches the compromised state. USTAT (UNIX State Transition Analysis Tool) [?] comes under this category.

Generally, misuse detection techniques are usually fast and accurate (as compared to anomaly detection), however they are not effective in identifying the novel attacks, whose signatures have not been specified.

Anomaly detection technique Anomaly detection is based on the deviation from normal activity profile (e.g., a user or a system). Any activity that significantly deviates from the normal behavior by a predefined threshold margin is considered to be intrusive. Such profiles can be based on statistical models on the user's historical activities and system call models for the normal execution of programs.

Anomaly detection approaches have been implemented in either learning based or specification based systems. A learning-based anomaly detection system utilizes a training period to learn the normal traffic and use it to recognize anomalous user or program behavior [?][?][?][?]. Specification-based systems take on a specification of the normal traffic

either manually or system generated. Specification based systems can also be classified into two method one which uses statistical models for user or program profiles [**?**][**?**][**?**] and second uses expert systems that use rules for normal behavior to identify possible intrusions [**?**]. For instance, Vigna et. al.[**?**] used illegal sequences of system calls utilized by system to make automated reasoning to generate usage profile.

Based on the subjects being monitored, we divide anomaly detection techniques into two classes[**?**]: user activity based techniques and program behavior based techniques. User activity based approaches create statistical models based on the user's historical data, and raise alerts when the user's activity significantly deviates from the statistical model. Program behavior based approaches create system call models for the normal execution of the programs, and raise alerts when monitored system call sequences deviate from the model.

Examples of anomaly detection systems employing user activity profiles include SRI's IDES [**?**], ADAM [**?**], NIDES [**?**], and W&S [**?**]. In IDES [**?**] and NIDES [**?**], statistics are monitored on subjects such as CPU utilization, frequencies of specific system events, distribution of audit records etc. When the deviation exceeds a pre-defined statistical threshold, it is said to be abnormal and an alarm is raised. W&S[**?**] automatically creates detection rules based on the statistics of historical audit records, and use these rules to detect abnormal user activities. Examples of the systems using program execution profiles include [**?**] [**?**] [**?**] [**?**] [**?**]. These approaches build their system call profile through either program training (monitoring normal program executions), or static

analysis on programs' source codes and execution. Current approaches to building system call models can be roughly classified into two categories: program training based approaches [**?**], and static analysis based approaches [**?**][**?**].

There are certain drawbacks of this technique. 1) Anomalous behavior that is not intrusive is considered as an intrusion (false positive). Anomaly detection suffers from high false alert rate, since it is hard to define normal behaviors. 2) Also intrusive activities such as stealthy attacks that are not anomalous result in false negatives. Even with the proper definitions of normal behaviors, an attacker can still gradually convince the IDS to believe his/her intrusive activities as normal behavior. Therefore, the key issue here is to select the threshold levels to reduce both false positives and false negatives. Anomaly detection systems are computationally expensive because they requires maintaining several system profile metrics.

2.1.2 Classification of IDS based on Audit Source

Based on the audit data processing, intrusion detection belongs to network-based audit data, host-based audit data, and application-based audit data.

1. ***Network-based IDS*** :- Network-based IDS gathers raw network packets as the data source from the secured network for malicious traffic. These types of IDS can scan complete packets or packet headers, or Net flow of network data. There are some networks IDS which use firewall logs as input that contain headers of network packets that have been

blocked by the firewall [snort[?]].

2. ***Host-based IDS*** :- Host-based Intrusion detection system scans operating system's audit data generated log of system calls[?] in a single host. Host based IDSs are implemented in four classes[?] (i) file system monitors which check the integrity of files and directories [?], (ii) logfile analyzers which analyze logfiles for patterns indicating suspicious activity[?][?] (iii) connection analyzers which monitor connection attempts to and from a host, and (iv) kernel based IDSs that detect malicious activity on a kernel level.

3. ***Application-based IDS*** :- Application-based intrusion detection system consumes user space application such as web-server log, mail server log[?]. These kind of IDS are used to protect network daemons.

These techniques have their own limitation. For example the network-based IDS cannot understand what is happening within the host if the network traffic is encrypted, because the network-based IDSs cannot encrypte the payload. Whereas HIDS can understand the encrypted traffic after decrypting in the host, however the host-based IDS can be compromised as a part of the attack. Also HIDS puts a performance cost on the monitored systems.

2.1.3 IDSs Limitation

Current intrusion detection systems suffer from serious well known problems. Following are major shortcomings the among them

1. *Massive alerts*: - Current intrusion detection systems generate enormous amount of alerts that are not caused by real attacks [?][?]. Because of

these huge data overwhelming security officers, these intrusion detection system becomes more of a tool for forensic work for the post attack analysis, rather than as a security monitoring tool.

2. *False positive*: - The worst thing about massive alerts is that most of them belongs to false positive and also these alerts may be mixed with true alerts, which create a challenge to filter out true alerts [**?**][**?**].

3. *Missing Alerts*: - As stated in [**?**] [**?**] [**?**] [**?**] often IDSs do not generate any alert for true attacks. This happens because many attackers frequently change their attack just enough to evade current signatures or tightly written rules in a signature based system so that it will only catch a subset of an attack vector. It happens also in anomaly detection when training data is taken at a compromised state. Another case when false negative happens is when IDS gets overloaded and it starts dropping packets.

4. *Incomplete Information*[**?**]:- IDS alerts usually do not contain sufficient information based on which security administrator can take decision. Main reason behind it is that their work in limited domain. For example, NIDS works in network level which scan the traffic, whereas HIDS scan the activity on a particular host.

5. *Truthless*:- As it is not necessary that all attempted intrusion will be successful, IDS usually are not able to discriminate between a successful attack and failed attack.

6. *Irrelevant positive*: - Most of the times IDS either generate true alert but

these may not belongs to the system context or administrator's interest. For example, an alert warning about a web-based attack for window machine, while system OS belongs to Linux. This happens due to IDS not having enough information about the hosts they are protecting.

7. *Unable to handle Novel attack*:- In most of the cases IDSs are unable to protect from attack variants or novel kind of attack. This happens because either IDS does not have attack signature or specification.

8. *Erroneous activity*: - IDS especially HIDS also generate alerts for erroneous events such as OS failure or software crash which was caused by bugs in the software.

2.2 Attack Correlation

In order to solve the problem with Traditional intrusion detection systems (IDSs) several alert correlation techniques have been proposed. As traditional IDSs concentrates on low-level attacks or anomalies, and raise alerts independently, however there may be logical links between them. Therefore, it is necessary to develop techniques to construct attack scenarios (i.e., synthesize attacker steps to make a larger view) from alerts and provide intrusion analysis.

2.2.1 Types of Attack Correlation

Attack correlation techniques falls in the four categories

Similarity based approaches Similarity based approaches[?] [?] [?] [?] [?] [?] cluster the alerts based on the similarity between alert attributes (e.g., source and destination IP addresses, protocol etc.). A number of the proposed correlation approaches include a multi-phase analysis of the alert stream. For example, the model proposed by Valdes et al. [?] [?] [?] presents a correlation process utilizing an alert similarity metric. Their correlation process is in three steps (i) attack threads aggregation based on attribute like sensor field, attack class, attack name, source, and target in the alerts are the same, (ii) multiple sensors' aggregation in which they cluster with similar metric except sensor field, (iii) higher level aggregation in which they aggregate the alert with relaxing the condition for the similar attack class. In [?], Debar&Wespi propose a system which uses both aggregation and correlation of intrusion detection alerts generated by heterogonous sensors.

One major issue comes in these methods is to define similarity measure. As many attributes are categorical (e.g., TCP/UDP Port numbers) traditional similarity measures [?] [?] may not be suitable. While many researches has proposed their own methods, Julisch et al. [?] [?] use conceptual clustering and generalization hierarchy to aggregate alerts into clusters. On the other side, these techniques have serious drawback like they fail to identify causal relationships between related alerts.

Attack Scenario-based correlation Attack Scenario-based correlation[?] [?] [?] is a knowledge-based method of well-defined scenarios of attacks either defined manually or learned by training datasets. Example

of this technique are STATL[**?**], LAMBDA [**?**], Tivoli approach [**?**], and the data mining approach [**?**]. In these models, their own language is used to express the scenarios which plays an important role in the applicability of the method. A major limitation of such methods is that they are restricted to known attack scenarios. As these models are close to our approach we give the details in the end of this chapter.

Require/provide model Require/provide model (e.g. JIGSAW [**?**], the MIRADOR approach [**?**]) also known as the prerequisites (pre-conditions) and consequences based model [**?**] [**?**] correlates alerts if the consequence of an alert satisfies the precondition of another alert. Several work such as [**?**] [**?**] show that it potentially uncovers the causal relationship between alerts, and not limited to known attack scenarios. In the related work section such methods has described in more detail.

Multiple security systems based approaches Multiple security systems based approaches [**?**] [**?**] [**?**] [**?**] keeps logs of different security tools such as IDSs, firewall log, router log, vulnerability scanners etc and are used for attack correlation [**?**].

2.3 Attack Languages

Attack languages are used to specify/state the signatures of attacks. These signature specifications help IDS in identifying intrusive actions/attacks in audit data stream. Both approaches of IDS i.e. anomaly detection system and

misuse detection system make use of attack languages in specifying and identifying patterns of malicious activities. Anomaly detection systems identify the deviation from the normal behavior in audit data by defining the intended or expected behavior with the help of attack languages where as misuse detection systems use the specification of the signatures of known attacks in attack languages to identify the later in data stream. Attack languages are also used to analyze the relationships among different attacks and to simulate the attacks for testing the system.

Attack languages differ in the set of signatures that can be described. A single attack language cannot specify all aspects of an attack efficiently as they are different in capabilities to express the various attributes/elements of attack. That's why different attack languages have their strength in specifying attack in different environments/domains. There are some desired characteristics of attack languages. Some of them are given here. An attack language:

- should have the capability to specify the temporal ordering of the events because two attack scenarios may constitute the same set of events but may differ in the order of the their occurrence.
- Should have the feature to describe the different parameters of an event
- Should be flexible enough to handle the change in any attack specification without much overhead

Vigna et. al. [**?**] have identified six different classes of attack languages based on their scopes and goals. These attack languages are: event languages, response languages, reporting languages, correlation languages, exploit languages, and detection languages.

1. ***Event languages***:- Events are specified by event languages. Event plays an important role in analyzing security as they serve as input during analysis generally performed by IDS. Some examples of event languages are tcpdump[**?**] packets, BSM(basic security module)[**?**], syslog messages [**?**]etc. Each language provides it's own format for each type of event and features.

2. ***Response languages***:- The response languages specify actions to be predefined in response to detection of intrusion. With the help of response languages, responses can be changed and activated dynamically on progression of attack. Presently there is no standard response language used by IDS that may provide full customizability and flexibility. Functions written in high level languages like C and Java are used by IDS for this purpose. However ADeLe language proposed by Michel et at.[**?**] has features to specify the responses.

3. ***Reporting languages***:- The purpose of Reporting languages is to define format of alerts generated by IDS and exchange procedures for sharing information of interest to intrusion detection and response systems, and to the management systems which may need to interact with them.. (Reporting languages include the specification of attack related information) These languages report about the information related to an attack i.e. source/attack, target/victim, severity of attack, vulnerabilities that are exploited by attacker, type of attack etc which is generally in the form of alerts. Common Intrusion Specification Language (CISL)[**?**] and the Intrusion Detection Message Exchange Format (IDMEF)[**?**]

are examples of reporting languages. CISL is part of the Common Intrusion Detection Framework (CIDF)[**?**]. IDMEF is based on XML and it builds on much of the previous CIDF work.

4. ***Correlation languages***:- Correlation languages are used to define the relationships among attacks to provide complete and coordinated attempt of security breach. They are the major focus of ongoing research in this area. For correlation, these languages use the alerts produced by various IDS in order to provide succinct and precise view of intrusion attack, as most IDS generate large number of alerts that make the job of security analyst harder. Therefore there is a great need of such languages. Examples, of current approaches to the correlation problem are the use of Bayesian networks, as demonstrated in Honey-well's ARGUS system and SRI s EMERALD, event-based reasoning, as demonstrated in UCSB s STATL [**?**], and rule-based reasoning, as in SRI s P-Best [**?**]. For all these correlation approaches to be implemented, correlation languages are required.

5. ***Exploit languages***:- Exploit languages specify the intrusion steps. So these languages are very helpful in testing security of a network and to test the IDS's deployment in secure network by simulating the attacks step by step and by generating various test cases. Exploits are commonly written in general purpose languages like C, C++, perl etc but they may make use of some attack specification languages like NASL[**?**], Custom Attack Simulation Language (CASL)[**?**] for attack scripting.

6. ***Detection Languages***:- The last category is of Detection languages,

which are also termed as attack languages, support detection of intrusion. They provide the mechanism for identifying the instance of attack by matching the input events with complete description of attacks in some format. Examples of detection languages are N-code[**?**], Russel[**?**], REE[**?**] and specification language such as[**?**] and [**?**].

Most of the correlation approaches require the specification of the attack steps or of complete attack scenarios in some format. Therefore, the languages used to specify the attack scenarios are of utmost important. LAMDA, ADeLe and CAML languages described below serve the purpose.

2.3.1 LAMBDA

Development of LAMBDA[**?**] language is part of MIRADOR project[**?**] initiated by French defense agency. LAMBDA is an attack description language. This language is based on logic and uses a declarative approach. This language allows the description of an attack operation in generic form. These generic descriptions are independent of intrusion detection process or computer system. They are later annotated with elements of specific detection process and target computer system.

In this language, an attack is described as a combination of actions, complemented by several statements in relation with the target computer system. Description of attack contains:*Pre-requisites*(A set of conditions to be satisfied in the system target of the attack for this attack to succeed), *Consequences* (effects or outcome of the successful attack), *Scenario*(describes the combination of different actions to perform the complete attack).

From the point of view of an intrusion detection system, it is required to describe separately what operations should be done to detect the occurrence of an attack. So it also allows the description of the detection process i.e., the elementary actions that should be performed in order to detect an attack. The language also describes how these detection actions are combined. Such description is similar to the description of the attack scenario. Finally it has the features to complement detection actions with verification actions. These actions aim at evaluating the impact of an attack on the computer system. These detection and verification aspects provide the language user with means to tailor the description of the attack to the needs of a specific intrusion detection system or a specific environment.

2.3.2 ADeLe

The ADeLe[**?**] language has been developed simultaneously with the Lambda language within the MIRADOR Project. ADeLe is also an attack description language and was designed to model the complete database of known attack scenarios. However, Lambda uses a declarative approach wheras Adele uses a procedural approach. As stated earlier that a single attack language cannot specify all aspects of an attack efficiently as they are different in their capabilities to express the various attributes/elements of attack, ADeLe is concerned with four classes of attack languages: exploit, correlation, response and detection. Language uses the IDMEF format for inter-operability between different IDSs. Attack descriptions allowed by ADeLe are more generic and modular both from the attacker's as well as defender's point of view. In this language, attack description consists of three parts: *Exploit*, *Detection*, and *Response*

part where correlation is covered in *Detection* part. The *Exploit Part* contains pre-condition, stage description code and post-condition. and the *Detection part* is a new language proposed to express the detection of attack. The *Response part* contains the list of functions used for automatic response. Most of the description itself is written in an XML-like code.

2.3.3 CAML

CAML (Correlated Attack Modeling Language)[**?**] is a recent language proposed by SRI International [**?**]. It aims at modeling multi-step attacks, in the modular form, where a module represents an inference step and modules can be linked together to detect multi-step scenarios. The relationships among modules are specified through pre- and post-conditions. Thus, modules can be linked together to recognize attack scenarios.

A module is the basic unit for specifying correlation steps in CAML. A module specification consists of three sections, namely, activity, pre-condition, and post-condition. To support event-driven inferences, the activity section is used to specify a list of events needed to trigger the module. These events are specified using event templates, which describe the requirements for the candidate event instances.The structure of CAML events is based on IDMEF. CAML has been tested within the EMERALD framework. This work is very similar to those of LAMBDA [**?**], JIGSAW [**?**] and P. Ning.[**?**] However Cheung et al.[**?**] have apparently spent a large effort to solve implementation issues (modules reuse and modules interfacing with the EMERALD existing).

2.4 Related Work

2.4.1 JIGSAW

Templeton and Levitt [**?**] originally proposed the first requires/provides model for alert correlation that defines the logical relations using JIGSAW language between different attacks in the same intrusion incident. They have shown that JIGSAW language is able to represent complex attacks and also helps to generalize the unknown attacks by illustrative examples. Rather than attacks as a series of events, they presented it as a set of capabilities that provide support for abstract attack concepts that in turn provide new capabilities to support other concepts. As in the model attack, concepts are defined locally and the model can be constructed without prior knowledge of the attacks. However the method has some limitations. For example, it is hard to practically enumerate the entire precondition and postcondition of attacks and model is likely to generate alert for which those conditions which are not defined. Also as it handles low-level attacks individually, and if attack does not prepare for (or is prepared for by) other alerts, it will fail to correlate it, even if the alert is related to others. Ning et al. [**?**] [**?**] present an alert correlation technique to overcome these problems by combining clustering correlation method with causal correlation method. In parallel, Cuppens et al. [**?**] proposed another approach which is able to deal with missed alerts. Their technique is based on abductive rules. If an alert, which is known consequence of an event that should have generated another alert, is received by the correlation engine and if that other event has not been received by the same correlation engine then, by means of a simple abductive reasoning, the

missed event can be identified. Cuppens et al. [**?**], in parallel, had proposed another approach which is able to deal with missed alerts.

2.4.2 Basic Capability Model

Capability model proposed by Zhou et. al. [**?**] models the capabilities obtained by an attacker from her point of view to form basis for require/provide model and use it for logical alert correlation. They defined capability as a six-tuple: (source, destination, credential, action, service, property), it shows that the capability means that the attacker can access the destination from the source and the access is to perform the action on the property of the service as credential. Their model abstracts the capabilities in a consistent and systematic manner without any ambiguity. Their approach defines all capabilities in different layers of a system abstraction using a single formula. They applied the model to abstract IDS alerts to capabilities not only from successful attacks, but also from unsuccessful attacks. Their framework is comprised of three major components: the capability model and capability sets, the inference rules, and the correlation algorithms. Inference rules represent derive logical relations between different capabilities in terms of inference rules like comparable, resulting etc. which is used in correlation algorithm .This separation of correlation policy (inference rules and correlation algorithms) from modeling IDS alerts brings great flexibility. For example, one can define certain inference rules that connect different capabilities via only the relations of the address fields, e.g., the same source and destination. This will turn alert correlation into simple alert clustering, based on network addresses. In addition, they also showed that different algorithms can work together, where

some act like plug-in to a generic algorithm in order to handle special cases. Thus, the approach can handle very different situations without altering the capability model and capability sets of alerts. Their approach is able to handle missing attacks and capture equivalent effects of different attacks. They have experimented in modeling hundreds of signatures of three popular NIDSs and that it facilitates the task of developing the capability sets based on the model and shown that it works in real-world intrusion detection datasets. However the model is manual and requires human intervention to execute the process. It is important note that the paper does not discuss the algebraic operations and relation between capabilities.

Chapter 3

Capability Model

A multistage attack is a series of attack where each individual attack's post condition serves as the precondition for next stage of attack. However, these attacks are not always linear in nature. At times more than one attack's post condition serves to fulfil the precondition for next step of attack. Therefore it's required to segregate attacks from their pre/post condition for it is not always one to one mapping. The capability modeling provides an elegant way of segregating attacks from their pre/post conditions by modeling the post effects (i.e. intruder and network effects) as the capability gained by the attacker. In a nutshell, capability summarizes these effects irrespectively of which attack caused it but yet at the same time without missing our purpose of identifying other attacks that can be launched whose precondition are fulfilled by the capability gained by attacker. In fact capability is the essence of all network and intruder effect of the attacks launched.

In the capability model, capability represents facilities and accesses that an attacker gains by making a connection. Capability describes the ability of an attacker during an intrusion. An attacker can have many capabilities at a particular instance that may or may not belong to the same intrusion.

3.1 Desired characteristic of Capability Model

- **Expressiveness:** Attributes of capability like service (e.g. software system, network etc.) should be fine-grained appropriately so that capability can be represented atomically and unambiguously. For example granularity for *read* action can be *read*, *list* and *know*. Here, action *know* represents information about file or directory. From this example it is clear that action *know* is different from action *read* however they belong to same category of actions but with different granularity and independent from each other.

- **Inference:** It is not necessary that all attack reported by IDS will be successful. Moreover failure of an attack may also give some capability. We need to consider both cases in the capability model. For example, If attacker's attempt to write in a file have been unsuccessful due to denial of permission, the attacker has gained knowledge about the existence and permission of the file.

- **Abstraction:** Model should also support different abstraction levels so that aggregation of capability becomes feasible which will help to define the possible operations in the capability. For this, it is required to identify relations between attributes value such as hierarchy, independence etc. For example action *write* makes hierarchy in which actions *create* and *modify* are contained.

- **Heterogeneous signature support:** As security is always a primary

focus for every organization consequently various kind of advance monitoring tools have been developed. These heterogenous tools often generate logs in different formats. Capability model should be able to accommodate the capability expressed in different signatures and be able to incorporate supplementary knowledge provided by other monitoring tools like host intrusion detection system, integrity checker etc. Intrusion detection message exchange format [IDMEF] [**?**] provide common signature but still not all IDSs are supporting it. In some cases, HIDS also give information about the failure or success about attack. This information plays important role to define capability, which are hard to capture by NIDS. For example, suppose NIDS prompted about some service exploit and later that service has been stopped then in this case attacker connection will have capability only when service is running and whenever services are stopped, attacker connection will lose that capability.

3.2 Definition of Capability

Let D be a set of network addresses, C be a set of credentials, A be a set of actions, SP be a set of pair of services and their property and $[t_1, t_2]$ is a time interval (where t_1 and t_2 are constant and $t_2 > t_1$).

Capability Capability is a six-tuple

capability = (source, destination, credential, action, (service, property), interval)

where $source \in D$, $destination \in D$, $credential \in C$, $action \in A$, $service \in S$,

property $\in P$, *interval* $\in [t_1, t_2]$ in which capability is valid. It may be noted that we have added the attribute interval to the definition of capability given by Jingmin et. al. [**?**].

Example 3.2.1 *The capability (pushpa, dblab, user1, read,('/etc/passwd', content), From* : $\langle 1997-07-16T19:20:30+01:00\rangle$*) means there is capability from host pushpa (source) to host dblab (destination) with credential user1 for read action of content of the file "/etc/passwd" in interval* $[1997-07-16T19:20:30+01:00, \infty]$.

It is not necessary that capability can be gained only through successful attempts. Failed attempts can also provide some capability. For example, in the case where an attacker wants to read a file or a webpage using http connection and is able to read the file successfully then he will have an information type capability that the file exist and has a reading permission. In the second case where the attacker gets the message "Forbidden" i.e. message for denial message to read the file then the attacker will again have an information capability that file exists and also that file does not have read permission. In the third case where attacker gets the message "Not Found" i.e. message for file does not exist then attacker will have an information type capability that the file does not exist. This example clearly shows that capability can be gained from a failed attack as well.

3.3 Attributes of Capability

As shown in the section 3.2, the definition of Capability contains six attributes. As the service and the property attributes are tightly coupled, therefore we

Table 3.1: Attributes of Capability

Attribute	Description	Example
Source	source address	IP:10.20.3.2, Ethernet:006097981E6B etc..
Destination	destination address	IP:10.20.3.2, Ethernet:006097981E6B etc..
Action	Actions that can be performed by an attacker	read, write, communicate etc.
Credential	Credential using which action can be done	root, system, user etc.
Service	Service used by connection	IIS 3.0, $\backslash etc \backslash passwd$ etc.
Property	Property of service	version, content etc.
Interval	Duration in which a capability will be valid	From:tstamp, Between:tstamp+tt, at:tstamp etc. Where tstamp is time-stamp and tt is it's length

merged them into a single attribute of two tuples. This modification helps us in defining an operation clearly. Except the *interval* attribute, rest of the attributes are almost same as proposed by Zhau et. al. with their details in [**?**]. However we are presenting a brief description in the following subsections. Summary of these attributes is given in Table 3.1.

3.3.1 Source and Destination

Source and destination are the integral components of an attack. Source specifies the address of origin of an attack. Destination specifies the address of the target system or victim under attack. Source and destination components of capability can have IP addresses, ethernet addresses or host names of the attacker and victim. Therefore it is also required to enter the type of addresses with their values. A network address is written in the form *type: value*, e.g.,

IP: 192.168.0.1. Some single-point attacks generate capabilities with exactly one source and one destination. But some network attacks have their impact on the whole network which results in a set of capabilities whose destinations include all the addresses in the network. For example, an attack crashing a router gives the attacker the ability to disconnect the network behind the router from the Internet. In this case, the destination of the capability is the complete set of network addresses behind the router.

Source and destination addresses are also helpful in modeling the attack scenarios by connecting the capabilities through the address relations. All phases of the intrusion could be correlated by the relations between the addresses in the capabilities.

3.3.2 Action

Actions describe the access method of a capability. This component facilitates alert correlation as it helps in correctly analyzing the impact of the attacks in intrusion detection. For example in some capabilities, the attacker obtains the access for only reading arbitrary files but not writing or executing them. So a single concept of access is not suitable and not sufficient to describe such capabilities. This is very crucial while correlating alerts using capability model. Five types of actions are identified for capability model : *Read*, *Write*, *Exec*, *Communicate*, and *Block*. Each action type is associated with certain values which are meaningful for appropriate type of objects and their properties in the systems. Table3.2 shows the action types and their values used in our model. For example, in *Read* action type, *Read* can denote the action to read the content of a file whereas *Know* may denote some knowledge or information

Table 3.2: Actions

Action Type	**Action Value**
Read	read, list, know
Write	create, modify, append, delete
Communicate	send, recv, connect, encrypt, decrypt
Exec	invoke, exec
Block	block(not permitted to run), delay(slow down), spoof(can replace), pause (can be stopped at any time), abort(forcefully terminate), unbolck

about the file gained by the attacker. Similarly in *Write* action type, create specifies the capability of an attacker to create a new file but attacker cannot modify any existing file which is the ability denoted by modify value of write action type.

3.3.3 Credential

Credential is the privilege generally used by the attacker to perform some actions. *Root*, *User*, *Daemon*, *System*, and *None* are the five types of credentials defined for capability model *Root* signifies the privilege of administrative account, *User* signifies the non-administrative interactive account, *Daemon* is non-interactive account for executing background processes, the *System* represents the access require to run the kernel process, and *None* symbolizes no specific privilege. Some actions require or involve only one privilege whereas certain actions may involve group privilege.

3.3.4 Service and Property

Behind every intrusion step, motive of an attacker is to gain access or information of certain objects. These objects are called services. Attributes of

these objects are known as properties. For example, in the case where the attacker intrudes to read the restricted file in a webserver, the webserver will be a service and parameter or attributes such as version, associated port number etc will be treated as properties of the service webserver. Services are categorized as active and passive. Examples of active services are RDBMS, OS etc. whereas example of passive services are file, memory etc. In context of intrusion detection, these services are of six types i.e. (i) Hardware system (ii) Software system (iii) Process (iv) Account (v) OS kernel (vi) File system and each type include sub types. For example, *File system* service includes subtypes *directories*, *files*, *program*, *scripts* etc.

3.3.5 Interval

Time interval is represented by predicate *between* : $[t1, t2]$ which shows that capability will exist from timestamp t1 to timestamp t2. It is also true that some capabilities (e.g. of knowledge type) once gained will always be present with the attacker. To denote such capability *from* predicate is used i.e. $From : t1$. For example if the attacker gaines that the target machine is running on Solaris OS at time t1 then interval of this capability is $From : t1$.

Timestamp can be taken in different formats. In this model following format of timestamp has been used:-

YYYY-MM-DDThh:mm:ss.sTZD

where YYYY = four-digit year,
MM = two-digit month (01=January, etc.),
DD = two-digit day of month (01 through 31),

hh = two digits of hour (00 through 23) (am/pm NOT allowed),

mm = two digits of minute (00 through 59),

ss = two digits of second (00 through 59),

s = one or more digits representing a decimal fraction of a second,

TZD = time zone designator (Z or +hh:mm or -hh:mm).

For example, 2007-07-16T19:20:30.45+01:00 represents the time-stamp which is in +1:00 time zone

3.4 Direct and Indirect Capability

Each Capability can belong to either of the two categories i.e. either direct capability or indirect capability. Direct capabilities can be defined as the capabilities gained by directly using the information or credential gained by some activity. Indirect capabilities, on the other hand, are gained by the knowledge generated by processing the gained information or implication of achieved credential. In other words, direct capabilities are those capabilities which belong to obtaining of direct information or credential after doing some activity. After processing this information or credential, the attacker is able to use it as knowledge or credential to do other activity. Capabilities formed from these inferred knowledge or implicated credentials are called indirect capabilities. For example, if attacker succeeds to make connection for reading a file which contains mail or credit card passwords then the direct capability is being able to read a password file. After reading the password, attacker can

use the password information to read the victims mail or can misuse his/her credit cards. These type of inferred capabilities are indirect capabilities.

3.5 Significance of time parameter

Time parameter which is denoted by the interval is a crucial parameter in reducing the false belief that each capability will last forever during correlation. Some capabilities especially indirect capabilities that depend on service running in target host and may only be valid under certain conditions. It is not necessary that these conditions will always be present in the network or system, for example, in the case where a service is scheduled to run for a specific duration. Therefore, it is clear that these conditions are bound to the validity of a session for capability and cannot be assumed that once gained by attacker they will always be with him/her. Ambiguity due to the assumption that capability once gained will always exist is called temporal ambiguity.

There are various sources of information that may help in specifying the closed time interval of the capability e.g. host integrity checker, HIDS etc. From these sources it can be identified that the capability gained earlier is no longer valid. Other sources may be administrator knowledge especially when some service is allowed for a limited period. For example there is one connection having capability to execute a program in host at time t_1 and later at time t_2(where $t_2 > t_1$) service has been blocked. In this case, the formerly established connection will not have same the capability.

3.6 Correlation process

The correlation process is based on the require/provide model in which capabilities gained from the previous attacks are used to satisfy the prerequisite of subsequent attacks. The model has following components.

3.6.1 H-alert

An H-alert is a three tuple (require, provide, raw) and represents transformed object of alert in terms of capability, where

Require is a set of capabilities that are required for alert to be a true attack.

Provide is a set of gained capabilities after an alert has been generated. Most IDS generate two kinds of alerts for each attack step, one for the incoming traffic in victim host and the other for the outgoing traffic from victim host. Alerts that have been generated for incoming traffic may be either successful or failure. This information is available in outgoing traffic. Attacker may even gain capability in a failed attack therefore the *provide* set contains those capabilities which have been gained by either successful attack or failure attack whichever is applied.

Raw contains other information available in alert message such as time of alert generated, traffic direction etc.

3.6.2 M-Attack

An M attack is a three tuple $(haset, capset, tmpstmp)$ which is a collection of correlated alerts where $haset$ is a set of alerts $(h - alerts)$, $capset$ is a set of

capabilities provided by h-alerts in *haset* and *tmpstmp* is the timestamp of last correlated alert which can be considered as *timestamp* of M.

Capabilities are considered as mandatory and optional (can be ignored while correlation in some conditions) in the *capset*.

In other words, M-attack represents a set of correlated alerts. The correlation process correlates a newly generated alert (H-alert) with these M-attack/M-attacks. Overall correlation algorithm has been explained in section 4.4.1.

3.7 Summary

This chapter explains the modified capability model. Based on the desired characteristic of capability model based (explained in chapter **??**) the capability model has been modified. The modified capability model is six tuple in which time notion has been added. Detail of all the tuples is briefly explained. Adding time notion helps to avoid temporal ambiguity between capability instances which is explained in the significance of time parameter section. Chapter also presented the concept of direct and indirect capability and also shown that capability can be gained by failure attack. Basic correlation process gives a brief overview about how correlation is done.

Chapter 4

Algebra for Capability Model

It is necessary to analyze the properties and the characteristics of the capability model to modularize the whole correlation process. Capability extraction from IDS signatures can be automated by identifying the algebraic properties of capabilities. It gives a better insight and clear demarcation between definitions of capabilities. This also helps to determine the level of granularity in defining the capability. Capability algebra can be divided into three groups i.e. *operations*, *relations* and *inferences*. These are described in the following section. The work in this chapter has publication in WISTP 2008 [**?**].

For comparing two capabilities, it is required to determine the relations between two capabilities, their inferences and relevant operations. It may be noted that the attributes of the capabilities form a hierarchy. We identify the following operations, relations and inferences base on this hierarchy.

4.1 Operations

Operations represent manipulations in the capabilities required in the correlation process. There are four kinds of operations identified for the correlation process. These are described in the following sections.

4.1.1 Join

Join operation merges two capabilities in presence of a join condition (see Algorithm 1). Two capabilities can be joined if both capabilities belong to the same source and destination. Also other attributes should be same except an attribute based on which join operation will be performed. For example capability C_1 *(srcS, dstD, daemon, block, (ftp process, port 80), from:2008-07-16T19:20:30.45+01:00)* is result of join of capabilities C_2 *(srcS, dstD, daemon, block, (ftp process, port 80), between:[2008-07-16T20:10:30.00+01:00, 2008-07-16T21:00:00.00+01:00])* and C_3 *(srcS, dstD, daemon, block, (ftp process, port 80), from:2008-07-16T19:20:30.45+01:00).*

Algorithm 1 Joining two capabilities

Require: Two capabilities C_1 and C_2
Ensure: Resultant capability C_3 if C_1 and C_2 can be joined else $NULL$.
Let $S = (cred, action, (service, property), interval)$
procedure JOIN(C_1,C_2)
if $C_1.src = C_2.src$ and $C_1.dst = C_2.dst$ **then**
if $\forall A_i \in S$ s.t. $C_1.A_i = C_2.A_i$ **then return** C_1
else if $\exists$ an attribute $A_k \in S$ s.t. $C_1.A_k \neq C_2.A_k$ and $\forall A_i \in S - \{A_k, interval\}$,
$C_1.A_i = C_2.A_i$ **then return** C_3 with $C_3.A_k = C_1.A_k \cup C_2.A_k$
else if $C_1.interval$ and $C_2.interval$ overlaps and other attributes are same
then return C_3 with $C_3.interval = C_1.interval \cup C_2.interval$
end if
end if
return NULL
end procedure

Join operation reduces the redundancy which in turn minimizes the number of comparisons (while finding inferences) between h-alert require set and M-attacks Capset (see section 5) during the correlation process.

4.1.2 Split

Split breaks a capability into two capabilities based on the given attribute and its value. For example *(srcS, dstD, userU, modify, (file, content), from:t1)* can be split in *(srcS, dstD, userU, append, (file, content), from:t1)* and *(srcS, dstD, userU, delete, (file, content), from:t1)*. It may be noted that *split* is semantically inverse of *join* operation. *Split* can be performed on the attributes (a) Action (b) Credential (c) Property (d) Time, if their value is composite[1]. After a *split* resultant capabilities would have same values of *src*(source), *dst* (destination) and *service*, however a *split* will not be done on the basis of these attributes.

Algorithm 2 Split a capability into two capabilities for given attribute

Require: Capability C, Attribute A and value of attribute v
Ensure: Resultant capability C_1 and C_2 if C can be split else C
procedure SPLIT(C,A,v)
if $C.A$ is not *composite*[1] **then return** C
else $C_1.A = v$, $C_2.A = reduce(C, A, v)$, $\forall A_i \in S - A$ set $C_1.A_i = C_2.A_i = C.A_i$
where S=(src, dst, cred, action, (service, property), interval)
return C_1 and C_2
end if
end procedure

Split is a special case of *Reduce* (defined in section 4.1.3) in which in which a capability C when split in two capabilities C_1 and C_2 can be obtained back by joining C_1 and C_2. In other words, split is lossless reduction (see Algorithm 2).

[1]Attribute A is composite if it contains multiple values or a value that can be divided into distinct components for eg. *RW* action can be split into R and W actions.

4.1.3 Reduce

Algorithm 3 Reducing a capability

Require: Capability C, Attribute A (must be composite) and v is value of A
Ensure: Reduced capability C_d
Let $S = (src, dst, cred, action, (service, property), interval)$
procedure REDUCE(C,A,v)
Create a new capability C_d with $C_d.A = C_d.A - v$,
$\forall A_i \in S - A$ set $C_d.A_i = C.A_i$, **return** C_d
end procedure

The *Reduce* operation weakens a capability by reducing strength of any of its attribute. For example capability *(srcS, dstD, root, modify, (program, code), from:t1)* can be reduced to *(srcS, dstD, userU, modify, (file, content), from:t1)*. Difference between *split* (Algorithm 2) and *reduce* (Algorithm 3) is that *split* operation always gives two capabilities whereas in the case of *reduce* it is not mandatory that the reduced part will be a capability.

4.1.4 Subtract

The Subtract operation takes two capabilities C_1 and C_2 and returns C_3 which is deduction of capability C_2 from C_1. For example *(srcS, dstD, userU, send, (IIS, Ftp), from:t1)* is result of subtraction of *(srcS, dstD, userU, receive, (IIS, Ftp), from:t1)* from *(srcS, dstD, userU, communicate, (IIS, Ftp), from:t1)*.

Subtract is similar to *reduce* in which minuend capability is reduced by subtrahend capability. For substraction it is necessary that both capabilities have same source and destination and only one attribute is different among the rest (see Algorithm 4).

Algorithm 4 Capability Subtraction

Require: Capabilities C_1 and C_2
Ensure: Resultant capability C_s
  procedure SUBTRACT(C_1,C_2)
    if $C_1.src$=$C_2.src$ and C_1.dst=C_2.dst **then**
      if $\exists$ an attribute $A \in S$ s.t. $C_1.A \neq C_2.A$ and $\forall B \in S - A$, $C_1.B = C_2.B$
        where S=(action, (service, property), interval) **then**
        C_s= Reduce(C_1,A,$C_2.A$)
      else $C_s = C_1$.
      end if return C_s
    end if
  end procedure

4.2 Relation

A relation represents a logical association between two or more capabilities. Following three types of relations are identified for the correlation process.

4.2.1 Overlap

Two capabilities *overlap* if there exists a common capability between them (see Algorithm 5). For example capabilities *(SLab, Dlab, RW, (/home/user1, content), user1, from:t1)* and *(SLab, Dlab, WX, (/home/user1, content), user1, from:t1)* overlap because the capability *(SLab, Dlab, W, (/home/user1, content), user1, from:t1)* is common in both. If any of the attributes between (a) Interval (b) Credential (c) Action and property of service are common in two capabilities, then there is overlapping: .

Algorithm 5 Test two capabilities whether they are overlap

Require: Two capabilities C_1 and C_2
Ensure: *true* or $false$
Let $S = (src, dst, cred, action, (service, property), interval)$
procedure OVERLAP(C_1,C_2)
if ($C_1.interval$ and $C_2.interval$ overlaps) **and** $\forall A_i \in S - \{interval\}$
s.t. $C_1.A_i = C_2.A_i$ **then return** *true*
else if $\exists$ a credential $cred_k$ s.t. $cred_k \in C_1.cred \cap C_2.cred$ **and** $\forall A_i \in S - \{cred\}$
s.t. $C_1.A_i = C_2.A_i$ **then return** *true*
else if $\exists$ an action act_k s.t. $act_k \in C_1.action \cap C_2.action$ **and** $\forall A_i \in S - \{action\}$
s.t. $C_1.A_i = C_2.A_i$ **then return** *true*
else if $\exists$ a property p s.t. $p \in C_1.property \cap C_2.property$ of the same service **and**
$\forall A_i \in S - \{(serivce, property)\}$ s.t. $C_1.A_i = C_2.A_i$ **then return** *true*
else return $false$
end if
end procedure

4.2.2 Independent

Two capabilities are independent if they cannot be joined (see Algorithm 6). In other words, two capabilities are called independent if either both have different source/destination or have different values of more than one attributes among the rest of attributes. For example capabilities *(SLab, Dlab, W, /home/user1, content, user1, from:t1)* and *(SLab, Dlab, X, httpd, (Apache 3.2, apacU), from:t1)* are independent.

4.2.3 Mutually Exclusive

Two capabilities are mutually exclusive if their corresponding attribute's value cannot coexist (see Algorithm 7). Mutually exclusive capabilities are less likely

Algorithm 6 Test two capabilities whether they are Independent

Require: Two capabilities C_1 and C_2
Ensure: *true* or $false$
  procedure INDEPENDENT(C_1,C_2)
    if join(C_1,C_2) is NULL **then return** *true*
    else return $false$
    end if
  end procedure

to belong to the same attack. This information helps in reducing false correlation. For example capabilities *(SLab, Dlab, R,(/etc/passwd, content), user1, from:t1)* and *(SLab, Dlab, X, IIS, Ver4.0, user1, from:t1)* are mutually exclusive.

Algorithm 7 Test two capabilities whether they are Mutual Exclusive

Require: Two capabilities C_1 and C_2
Ensure: *true* or $false$
  procedure MUTUAL-EXCLUSIVE(C_1,C_2)
    if $\exists$ an attribute A s.t. conflict($C_1.A$,$C_2.A$) is *true* **then return** *true*
    else return $false$
    end if
  end procedure

The *conflict set* used in algorithm 7 is a knowledge base having pair of attributes that cannot coexist e.g. service of windows and Linux cannot exist simultaneously in the same IP with the same port.

4.3 Inferences

Inference means causal relationship involved in process of deriving result or making a logical judgment on the basis of known evidence. Inferences identified here are used in comparing the capabilities of require set of h-alert with

capabilities in M-attack's capability set based on require/provide model during the correlation process. Almost all the inferences given in this section are same as given in [**?**].

4.3.1 Comparable Inference

Comparable inference denotes semantic comparability of two capabilities. Two capabilities can be compared only if they hold same type of service and property while other attributes must be same. This inference will be used to correlate two capabilities to construct attack scenario. Capabilities can be correlated only if required capability can be satisfied with some of the capability of M-attack set by *comparable inference* (see Algorithm 8).

Algorithm 8 Test whether C_1 and C_2 can be compared directly

Require: Two capabilities C_1 and C_2
Ensure: *true* or $false$
  procedure COMPARABLE(C_1,C_2)
    if $\forall\ A_i \in \{$ src, dst, cred, action $\}$, $C_1.A_i = C_2.A_i$, with overlapped time interval
      and both have same type of service and property **then return** *true*
    else return $false$
    end if
  end procedure

Service and property belong to the same type when services belong to same category as given in section 3.3.

4.3.2 Resulting Inference

In many cases logical relations between capabilities cannot be represented by *comparable inference* due to strict conditions. One capability is the *resulting*

inference of other if it gives the other capability on its execution. These inferences are nothing but a single step of correlation process and are used in making an attack scenario through multi step correlations (see Algorithm 9).

Algorithm 9 Test whether C_2 is resulting inferable from C_1

Require: Two capabilities C_1 and C_2
Ensure: $true$ or $false$
  procedure RESULTING_INFERABLE(C_1,C_2)
    if exercise of C_1 logically derive C_2 **then return** $true$
    else return $false$
    end if
  end procedure

Administrator knowledge, topology of network are some of the major information sources to identify the capabilities which can be logically derived by exercising a capability.

4.3.3 Other Inferences

Several other inferences are also possible along with the given inferences. For example *compromise inference* and *external inference* as given by Jingmin et. al. [**?**]. Through compromise inference one capability can be inferred from other capability for compromising the destination machine (executing arbitrary program).Capability C_1 can be externally inferred from capability C_2 if C_2 is the capability to execute arbitrary program on destination machine which is the source of C_1.

4.4 Correlating alert using modified capability model

4.4.1 Correlation Algorithm

Correlation algorithm correlates new h-alert (created from alert generated by IDS) with the existing M-attacks. Initially there is a set of M-attacks M. Whenever a new alert comes, then it is abstracted into an h-alert. Correlation algorithm searches a minimal and ordered subset of M-attacks from M such that all the required capabilities of h-alert are satisfied by the capabilities of a subset of M-attacks. The algorithm then combines h-alert with the identified subset of M-attacks in a single M-attack. This M-attack contains all capabilities of selected M-attacks along with the h-alert's provide capabilities. This new M-attack replaces the subset of M-attacks. The whole correlation process is presented in Algorithm 10.

Algorithm 11 shows the search procedure of M-attacks that satisfy the required capabilities of newly generated h-alert.

4.4.2 Case Study

We have extended the existing capability model by adding a new attribute i.e. time. The modified capability improves the correlation by reducing the cases of false correlation and by increasing correlation strength. Following cases arises:

Case1: Case where C_1 *(srcX, dstX, credX,* $\{RW\}$*, (/home/user1, content), intvX)* is required capability of a h-alert and two M-attacks M_1 and

Algorithm 10 Correlate a new h-alert with M-attacks

Require: h-alert h_1 and set of M-attacks M=$\{M_1, \cdots M_n\}$
Ensure: a new M-attacks set M'=$\{M_1$,.... $M_k\}$
procedure CORRELATION ALGORITHM(h_1,M)
Find a minimal and ordered subset M^k of set M (as given in Algorithm 11)
such that h1.requires is satisfied by capabilities in M-attacks of set M^k
if $M^k \neq \phi$ **then** Make new M-attack M_{new} as
$M_{new}.capset = C_M \cup h_1.provide$ where $C_M = \bigcup_i M_i^k.capset$ and $M_i^k \in M^k$,
$M_{new}.haset = h_M \cup h_1$ where $h_M = \bigcup_i M_i^k.haset$ and $M_i^k \in M^k$
and replace all M-attacks in M^k by M_{new}
else Make new M_{new} as $M_{new}.capset = h_1.provide$ and $M_{new}.haset = h_1$
end if
M_{new}.timestamp equal to the timestamp of newly correlated alert.
end procedure

M_2 in M-attack set having capabilities C_2 *(srcX, dstX, credX,* $\{R\}$*, (/home/user1, content), intvX),* C_3*(srcX, dstX, credX,* $\{W\}$*, (/home/user1, content), intvX)* in their *capset* respectively. Using former approach capability C_1 cannot be correlated with either of M-attacks (M_1 or M_2) capability because the action attribute of C_1 cannot directly be compared with that of C_2 or C_3. Therefore, the former approach is unable to correlate it. But in the modified approach whenever C_1 and C_2 are correlated, then C_1 reduces to the capability *(srcX, dstX, credX, W, (/home/user1, content), intvX).* Then it be directly correlate with C_3 which means C_1 is correlates by $M_1 \cup M_2$. It is clear here that enhanced model is able to detect these kinds of true correlations that would have gone undetected in the earlier approach. These kind of cases have been handled in the modified approach because of flexibility by defined operations such as join, split and reduce.

Algorithm 11 Find a minimal and ordered subset M^k of set M

Require: h-alert h_1 and set of M-attacks M=$\{M_1, \cdots M_n\}$
Ensure: M-attacks set M'
1: **procedure** FIND_MIN_ORDERED_SUBSET(h_1,M)
2: Order all M-attacks based on decreasing Timestamp and let cap_{req_set} is set of capabilities in $h_1.require$, $CapM_{sat} = \phi$ and $M_{result} = \phi$
3: **for all** M-attacks $M_i \in \{M_1, \cdots M_n\}$ **do** find subset $cap_{satisfied} \subseteq cap_{req\_set}$ $infer able$ (see section 2) from $M_i.capset$, $CapM_{sat} = CapM_{sat} \cup cap_{satisfied}$
4: **if** $CapM_{sat} = h_{req\_set}$ **then** $M_{result} = M_{result} \cup M_i$ and **return** M_{result}
5: **else if** $cap_{satisfied} \neq \phi$ **then**
6: cap_{req_set} =cap_{req_set}-$cap_{satisfied}$ and $M_{result} = M_{result} \cup M_i$
7: **else**
8: find $cap_{sub} \in cap_{req\_set}$ that can be obtained from $M_i.capset$ by subtract.
9: **if** $cap_{sub} \neq \phi$ **then**
10: $cap_{req\_set} = cap_{req\_set} - cap_{sub}$, and $M_{result} = M_{result} \cup M_i$
11: **end if**
12: **end if**
13: **end for**
14: **return** ϕ
15: **end procedure**

Case2: The require set of an incoming h-alert is satisfied by the *capset* of M-attacks M_1, M_2 and there exists a capability in M_1 which is mutually exclusive of other capability that belongs to M_2. In this case, M_1 and M_2 have no correlation. But the former approach would erroneously correlate these kind of capabilities. Whereas, in the proposed model such capabilities are not correlated because they are mutually exclusive and logically donot belong to the same attack. For example, a capability C_1 *(eth0:12ffdd3453, eth0:12ffee1234, credX,* {*RW*}*, (/home/user1, content), intvX)* belongs to M_1 and other capability C_2 *(srcX, dstX,*

credX, {RW}, (IIS, content), intvX) belongs to M_2. Administrator knowledge, services running in the network, topology of network are the major sources of domain knowledge in identifying the *mutual exclusive* capabilities discussed in section 4.2.3.

Case3: Modified process also handles the correlation conflicts that arise due to temporal ambiguity as explained in section 3.5. For example, in the case where the attacker has a capability to read and write in a host H, then the attacker can also read and write the mail of a user whenever he opens his mail account on that machine i.e. attacker will have the capability of reading/writing mail from a particular user account only for the duration in which the user is logged in. However in this case, there is no upper limit of interval for reading/writing other files. To avoid this ambiguity time attribute has been added with every capability.

Apart from the cases discussed above, there are several other cases where the proposed model helps in making overall process efficient. For example, *Join* operation helps in reducing the redundancy which in turn saves the number of comparisons during the correlation proecess. In this case, there are two capabilities C_1 *(srcX, dstX, credX, {RW}, (/, content), intvX)* and C_2 *(srcX, dstX, credX, {RW}, (/home/, content), intvX)* which can be joined into one capability as they form *contain-ship* relation. Therefore it is clear that if two capabilities of M-attack's cap set are joined then further correlation needs only one comparison instead of two. *Overlapped* and *independent* relations help in defining join condition accurately to test unambiguously that two capabilities can be joined or not.

4.5 Discussion and other issues

In this section, other possible ways of correlation process are discussed. It is clear that join algorithm has significant impact in minimizing the number of comparisons in correlation because it combines the capabilities in M-attacks's capset. However join itself is a costlier operation in terms of time as described below. Following are the alternate methods of doing correlation using various combinations of join and split.

4.5.1 Alternate Method 1

Algorithm 12 (Alternate Method 1) Correlate a new h-alert which is an abstract form of recently came alert with M-attacks

Require: h-alert h_1 and set of M-attacks M=$\{M_1, \cdots M_n\}$
Ensure: a new M-attacks set M'=$\{M_1, \ldots. M_k\}$
procedure CORRELATION ALGORITHMII(h_1,M)
Find a minimal and ordered subset M^k of set M such that h1.requires is
satisfied by capabilities in M-attacks of set M^k using algorithm 11
if $M^k \neq \phi$ **then** Make new M_{new} as
$M_{new}.capset = C_M \cup h_1.provide$ where $C_M = \bigcup_i M_i^k.capset$ and $M_i^k \in M^k$,
$M_{new}.haset = h_M \cup h_1$ where $h_M = \bigcup_i M_i^k.haset$
and $M_i^k \in M^k$ and replace all M-attacks in M^k by M_{new}
else Make new M_{new} as $M_{new}.haset = h_1.provide$ and $M_{new}.haset = h_1$
end if
for all pair of capabilities (C_i,C_j) in $M_{new}.capset$ **do** C_k=join(C_i,C_j)
if $C_k \neq NULL$ **then** replace C_i and C_j by C_k in $M_{new}.capset$
end if
end for
M_{new}.timestamp equal to the timestamp of newly correlated alert.
end procedure

In this method after the correlation, algorithm 12 joins capabilities within

each M-attack i.e. within each M-attack if two or more capabilities can be joined then they are joined to minimize the number of capabilities in capset and remove the redundancy. The minimal set search algorithm is same as algorithm 11.

It may be noted that the method minimizes the number of comparisons while searching for the minimal set of M-attacks because of a lesser number of capabilities in each M-attack's capset.

However join operation is a costlier operation. For example in a M-attack's capset if there are n capabilities then join operation is called for every pair of subset of capabilities which is exponential because the join operation is recursive until no more joins are possible.

4.5.2 Alternate Method 2

In this method capabilities in new h-alert's require set are split into minimal granularity based on their composite attributes.

In this case, we do not use join operation for correlation as it is costly. By using split operation, the granularity of each attribute of every capability will become one. Consequently, this will make the comparisons easier. Also we do not need the subtract operation as all capabilities are in their minimal reduced form. Minimal set search algorithm is same as algorithm 11 except the subtract operation in steps 8,9 and 10.

However in some cases we may end up in split where it may not be required. For example capability containing action RW can be split into two capabilities with action R and W in M-attack's capset. A new required capability with same RW action comes, then we can split it into R and W, which requires two

Algorithm 13 (Alternate Method 2) Correlate a new h-alert which is an abstract form of recently came alert with M-attacks set

Require: h-alert h_1 and set of M-attacks M=$\{M_1, \cdots M_n\}$
Ensure: a new M-attacks set M'=$\{M_1$,.... $M_k\}$
Let $S = \{cred, action, \{service, property\}, interval\}$
procedure CORRELATION ALGORITHMIII(h_1,M)
for all capabilites $C_i \in h_1.require$ **do**
for all attributes $A \in S$ **do**
if A is composite **then** split C_i into minimal granularity based on A
end if
end for
end for
Find a minimal and ordered subset M^k of set M such that h1.requires is
satisfied by capabilities in M-attacks of set M^k using algorithm 11
if $M^k \neq \phi$ **then** Make new M_{new} as
$M_{new}.capset = C_M \cup h_1.provide$ where $C_M = \bigcup_i M_i^k.capset$ and $M_i^k \in M^k$,
$M_{new}.haset = h_M \cup h_1$ where $h_M = \bigcup_i M_i^k.haset$ and $M_i^k \in M^k$
and replace all M-attacks in M^k by M_{new}
else Make new M_{new} as $M_{new}.capset = h_1.provide$ and $M_{new}.haset = h_1$
end if
M_{new}.timestamp equal to the timestamp of newly correlated alert.
end procedure

comparisons. Indirectly we may be increasing the comparisons unintentionally as number of capabilities in the capset of M-attack increase in some cases.

4.5.3 Alternate Method 3

This method is a combination of Alternate Method 1 and Alternate Method 2 which splits the capabilities of h-alert's require set into minimal granules and after correlation, joins the capabilities in the newly formed M-attacks's capset which can be joined.

Algorithm 14 (Alternate Method 3) Correlate a new h-alert which is an abstract form of recently came alert with M-attacks set

Require: h-alert h_1 and set of M-attacks M=$\{M_1, \cdots M_n\}$
Ensure: a new M-attacks set M'=$\{M_1$,.... $M_k\}$
Let $S = \{cred, action, \{service, property\}, interval\}$
procedure CORRELATION ALGORITHMIV(h_1,M)
for all capabilites $C_i \in h_1.require$ **do**
for all attribute $A \in S$ **do**
if A is composite **then** split C_i into maximum granularity based on A
end if
end for
end for
Find a minimal and ordered subset M^k of set M such that h1.requires is
satisfied by capabilities in M-attacks of set M^k using algorithm 11
if $M^k \neq \phi$ **then** Make new M_{new} as
$M_{new}.capset = C_M \cup h_1.provide$ where $C_M = \bigcup_i M_i^k.capset$ and $M_i^k \in M^k$,
$M_{new}.haset = h_M \cup h_1$ where $h_M = \bigcup_i M_i^k.haset$ and $M_i^k \in M^k$
and replace all M-attacks in M^k by M_{new}
else Make new M_{new} as $M_{new}.capset = h_1.provide$ and $M_{new}.haset = h_1$
end if
for all pair of capabilities (C_i,C_j) in $M_{new}.capset$ **do** C_k=join(C_i,C_j)
if $C_k \neq NULL$ **then** replace C_i and C_j by C_k in $M_{new}.capset$
end if
end for
M_{new}.timestamp equal to the timestamp of newly correlated alert.
end procedure

The method wipes out pitfalls of pervious methods as split has been used initially to simplify the comparisons and later on join has been used in each M-attack's capset to minimize the number of capabilities which consequently minimizes the number of comparisons . However this method is more costly than the previous in time complexity.

4.6 Summary

This chapters presents the identified algebraic structures of the capability in context of capability model based attack correlation. These structures give better understanding of capability characteristics. and help in designing the correlation process in a systematic and modular fashion. The algebraic structures are in three classes i.e. operations, relations and inferences. Operations include join, split etc. which represent basic manipulation using one or more capability instance. Relations include overlapped, mutual exclusive, independent relations between capability instances. These relations help in identifying the preconditions to allow specific operations. As the whole system is based on require/provide model therefore to determine whether a capability satisfies a required capability, inferences are used. Inferences include comparable, resulting etc., which enumerate the possible inferences from different real life views. The chapter also gives three derived versions of the correlation process from basic correlation process using these algebraic properties.

Chapter 5

ACML: Capability Based Attack Modeling Language

In this chapter, we propose a novel Attack Capability Modeling framework (ACMF) to semi-automate the whole capability model based correlation process with the help of algebra for attack capability. The framework minimizes the human intervention which is required for processing IDS alerts for the purpose of identifying attack scenarios.

We also propose Attack Capability Modeling language (ACML), which is a specification language of capability model. The objective of developing ACML is to provide flexibility to security concerns in customizing the capability model. With a customized model, the performance of the system can easily be enhanced. ACML specifications help to avoid ambiguity which enables the utilization and reusability of the experience and knowledge of security professionals in the field of attack correlation. With this, the system can also be adaptive to optimization.

In the section next section(3) we explain the Attack Capability Modeling framework (ACMF) and its components. Then section 5.2 presents the detail of Attack Capability Modeling language (ACML) which is major component of the ACMF. The work in this chapter is accepted to publish in the IAS 2008 conference.

Figure 5.1: ACM Framework.

5.1 Attack Capability Modeling Framework

The main objective of the Attack Capability Modeling framework is to detect intrusion scenarios using the capability model. The framework consumes first level security alerts and reports that identified multistep attack scenarios discovered in the alert stream. Figure 5.1 shows an integrated correlation process using capability model. The whole process can be divided into five modules as given below-

1. **Attribute specification** :- In this module, the specifications and taxonomy for capability and its attributes are collected from the user. This module contains two parts as shown in example 5.2.1. First part of the specification contains declarations of the attributes of capability like action, credential etc while the second part contains the relations between these attributes. This specification is first of part of whole program written by user in the ACML (see section 5.2). After passing this specification through syntax-checking and parsing modules, It is provided to the H-alert specification module and the Correlator module.

2. **H-alert specification module** :- The system takes IDS alerts to construct an attack scenario. In this module, alerts which are generated by IDS (in IDMEF format) will be converted into H-alerts. Figure 5.2 shows the details of this module. In this module H-alert db is a relational database which contains the valid H-alerts corresponding to raw alerts. Initially it may be empty and whenever a new H-alert comes in, it is checked against H-alert database. If the corresponding H-alert exists in

the database then its reference id of corresponding H-alert is send to the next module. If it is not in H-alert database, then programmer would have to write a H-alert specification for this raw alert in the ACML. After parsing, the valid H-alert is stored in H-alert db and its reference id is sent to next module.

3. **Attack Scenario specification** :- This module contains specifications of known attack scenarios in terms of capabilities. It comprises of a set of alert IDs (of all alerts that could possibly represent any of the steps in the complete attack) along with the set of require and provide capabilities. Each require capability is tagged with its sequence number which is assigned in chronological order for each multi-step attack scenario. Detail about H-alert specification is given in section5.2.4.

4. **Standard algebraic library** :- This module contains the different algebraic structures defined in chapter 4. Framework contains three different file i.e. each for operations, relations, and inferences algorithm. It contains two relational databases i.e. Conflict knowledge base and Inference knowledge base. These databases is used by the relation and inference library. These libraries are loosely coupled modules. Various APIs is provided in the language to make the customization of these libraries feasible. Detail of this module has been discussed in section 5.3.1.

5. **Correlator module** :- Correlator module contains all four kind of standard algorithms for correlation as shown in section and . However, the objective of making it a separate module is to provide flexibility and plug or embed their own modules. As shown in figure 5.2, This module also

Figure 5.2: H-alert specification module.

requires a attribute specification from the Attribute specification module and H-alert stream from H-alert specification module for correlation. Correlator module also makes use of Attack scenario specifications to identify the threat level of an attack. As an output, this module generates the attack scenarios after correlation.

5.2 Attack Capability Modeling Language

5.2.1 Lexical Details

The lexical structure of a programming language is the set of elementary rules that specify how you write programs in the language. The description of the lexical structure of the Attack Capability Modeling Language (ACML) language specifies the format of comments, set of keywords, delimiters, separators to specify how one program statement is separated from the next and the generic identifiers and constant matching patterns used in identifying instructions during program parsing.

Identifiers in ACML are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ ', reserved words excluded. These identifiers are of two types, first which start with a upper case letter and the other one that start with a lower case letter. For example, *Read* is first type of identifier and *write* is of second type.

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols,

and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like C and Java, including longest match and spacing conventions.

Language allows both kind of comments single line i.e. '`//`' using symbol at the start of the line and multiple line comments (any text between `/*` and `*/` including the delimiter) same as in C++ language.

5.2.2 Data Types

ACML provide several additional built-in data types and literals along with that available in C Language such as String literals *String* (for example "x", where x is any sequence of any characters except " unless preceded by \), IP4Address, IP6Address, Ethernet Address, Port number (WellKnownPorts, RegisteredPort, DynamicPort), Time, Interval etc.

Moreover, language also support user defined types. This feature has a significant role in implementation of generic and abstract model because implementation of such model needs lots of customization which is very hard to incorporate. User can define new data types in declaration and specification section 5.2.3. The language also supports Structure and Union data types (as defined in C-Language) which will build over built-in data types.

5.2.3 Attribute Specification

To make a customized capability model, first step is to define terminology of the attributes of capability. Some instances of these attributes can be semantically interconnected and can form some association. A specification

is required to capture such notion. These specifications contain capability attributes and relations between them. After syntax checking, attributes and their relations are fed to the H-alert and Correlator modules as shown in framework (see figure 5.1).

Following is the syntax of Attribute specification.

```
AttributeSection ::= attributes  AttributeID
‘{’
    actions ":-" ListOfActionNames ‘;’
    credentialGroup ":-" ListofCredGroups ‘;’
    credentialUser ":-" ListofCredUsers ‘;’
    {ServiceDeclaration}
    [AddressDeclaration]
    [IntervalDeclaration]
‘}’
ConnexionsSection ::= connexions
‘{’
    [ActionConnexionDeclaration]
    [CredentialConnexionDeclaration]
    [ServiceConnexionDeclaration]
‘}’
```

Detail of the syntax is given in Appendix 1. In above syntax, optional items are enclosed in square brackets‘[’, ‘]’, items that may appear zero or more times are enclosed in curly braces ‘{’,‘}’, literals start with lower case letters represent keywords whereas those starts with upper case letters are non-terminals, literals enclosed in single quotation marks represent symbols/terminals of the language. Literals enclosed in angular braces represent placeholders for code blocks.

Attribute Specification consists of two parts i.e. (i) attribute definition and (ii) connexion, where attribute definition part comes first, followed by connexion part (see example 5.2.1).

Attribute definition As shown in section 3.2, definition of capability contains five types of attributes i.e. (i) Address (source and destination address), (ii) Action (ii) Credential (iv) Service and property (v) Interval.

However for address and interval attributes, several built-in data types are provided and in addition user can also define its own data types. For example, timestamp and interval are built-in types for interval attribute; user may define rightOpenInterval data type for right open ended interval (see example 5.2.1).

As shown in the Attribute specification syntax, except address and interval attributes, declaration of other attributes is mandatory. Credential group is similar as user group in Linux and it's identifier must start with upper case character, whereas credential user is simple user and it's identifier starts with lower case character. Service is defined along with its attributes. Number of parameters and their type depends on the type of service. For example, for *webserver* service type, five parameters are there i.e. version, permittedAction, portNumber, origin, languageSupport. Following example gives the general outline of specification.

Example 5.2.1 *A sample Attribute specification.*

```
attributes secureLabAttributes
```

```
{
   action :-
       Read(read, list, know, readPermission),
       Write(write, append),
       Modify(changePermission, writeExtendedAttibute),
       Run(block,pause,delay,execute);
   credentialGroup:-
       Admin, User, Guest, Faculty, Seclab, Updaters;
   credentialUser:-
       root(Admin), navneet(User,Updaters), default(Guest);

   service  webserver
   {
       int version;
       action permittedAction;
       RegisterPort portNumber;
       string origin;
       string languageSupport;
   }

    service  fileserver
   {
       int version;
       string supportedEncryption;
       action permittedAction;
       RegisterPort portNumber;
       Time sessionTime;
   }

   address homenet
```

```
    {
        IPAddress mynetwork;
        IPAddress subnetmask;
        IPAddress proxyIP;
        Port proxyPort;
    }

    address externalnet
    {
        IPAddress externalnetid;
        IPAddress subnetMask;
    }

    interval rightOpenInterval
    {
        Time endtime;
    }
}



connexions
{
 ActionConnexions :-
       list<know, block(pause,delay),
       write[modify,append];

 CredentialConnexions :-
       default < user;
```

```
ServiceConnexions :-
      server[webSer,fileSer];

}
```

Connexions Second part of the specification(i.e. connexion part) contains relations and associations between attributes. Following three types of connexions can be defined in this part of specification. These connexions can be defined for service, credential (credential user) and action attributes of capability.

1. *Division*:- For example, *att1 (attA, attB, attC)* means *att1* is union of *attA*, *attB* and *attC*. The attributes *attA*, *attB*, *attC* may not semantically independent. This notation (division) also applies in those cases where it is not clear, whether attributes (i.e. attA, attB attC) posses any dependency among them. For example, let's take a connexion in action attribute is *block* (*pause*, *delay*). Action *delay* shows that if attacker has capability to *pause* a service and also able to *delay* it then attacker can do *block* by first pausing it and then delaying it for infinite time. In this case it is assumed that security administrator is unclear about the relation between *pause* and *delay* action.

2. *Partition*:- att1 [attA, attB] means att1 can be partitioned into two attributes such that deducting attA from att1 will give attB. For example, let's take a connexion in action attribute *write* [*modify*, *append*]. In this case *write* action is the combination of *modify*

and *append* action. In this *modify* and *append* are semantically independent actions.

3. *Inclusion*:- Inclusion means that $att1 < att2$ if attribute att1 semantically include att2. For example, in the Window OS context, *default* credential is semantically contained in *administrator* credential.

5.2.4 H-Alert Specification

Alerts are automatic messages generated when a user requests to be notified of new search results fitting their criteria. Security Alerts provide timely information about current security issues and exploits.

H-alert

H-alert is a abstraction of raw alert generated by various security devices like NIDS, HIDS etc. in terms of capability. H-alert in capability based ACML model consists of:

1. *Require capability set*:- It is a set of capabilities that are required for alert to be a true attack.

2. *Provide capability set*:- It is a set of capabilities that can be seized after an alert has been generated.

3. *Direction*:- It stores the information about packet/alert direction. A NIDS alert differs from a HIDS alert in that it is generated from some network packets that have a direction property.

In the ACM framework, Correlator module requires the alerts to be in the form of capability. Most of the IDS generate alert in IDMEF format. From IDMEF alert generating H-alert specification is simple and require little human intervention as most of values of the IDMEF fields is directly mapped with H-alert's attribute. ACM Language provide flexibility to automatically generate partially filled H-alert template. For each alert there is a corresponding mapping of H-alert template in the plug-in form. This loosely coupled design facilitates easier replacement of the template for customization. The modular design plays important role by providing higher degree of reuse of the framework as generating H-alert template of every alert is tedious task.

Syntax

H-alert specification contains inclusion of external libraries followed by HSet which represents the set of H-alerts. Syntax of this specification is as follows:-

```
{‘#’include ‘<’ Lib_ID ‘>’}
hset Set_name
‘{’
   <Global variables>
   <Function & Predicate definitions>
   halert Name ([Parameters] )
   ‘{’
      alertId ID1 ‘;’
      reqcapbility
      ‘{’
         <Capability_Set>
      ‘}’
```

```
    procability
    '{'
        <Capability_Set>
    '}'
     direction forward / reverse ';'

  '}'
'}'
```

HSet contains zero or more global variable and user defined functions with H-alert definitions. In the function and predicate definition part function contains user defined function and procedures. Language also contains set of predicates on all the attribute of capability (i.e. address, action, credential, service & property, interval) like Subset, Union, Deduct, Overlap, Not_subset, IsComposite, IsSameType etc. Each H-alert will have an AlertID (assigned by IDS) and require and provide capability set. Direction field in H-alert contains direction information i.e. either forward or reverse (As alert is direction sensitive [**?**]).

An H-alert must contain at least one require and one provide capability. *CapabilitySet* is the set of capabilities i.e. one or more *Capability*. Each capability is six tuple along with two additional fields

1. mode

2. optional

In the capability model, capability is categorized into two type i.e.

1. Direct

2. Indirect

Direct capabilities are those which are gained directly by doing intrusion action. Implication of direct capability also provide capability and which is known as indirect capability. Mode field contains the value "direct" or "indirect" based on the type applied. Optional field represents whether this capability is mandatary for correlation. Following is the syntax of capability.

```
Capability ::=
    capability CapID
    '{'
       sourceaddress AddressType  srcID ';'
       destaddress AddressType  destID ';'
       action actionID ';'
       credential credID ';'
       service serviceName ';'
       property propertyID ';'
       mode direct/indirect ';'
       optional true/false ';'
    '}'
```

In the above capability definition syntax, *SourceAddress* and *DestAddress* will contain in-built data types i.e. AddressType which will be validated by the parser. *Direction*, *mode* and *optional* fields can have exactly one of two values as mentioned above. The rest of data types are specified by user in the Attributes Specification Section.

Following example shows the H-alert specification for illegal NFS mount alert generated given in [**?**]. Specification shows that illegal NFS mount requires two capabilities (i) accessLevel i.e. capability to be able to *communicate*

to the target host, and (ii) mountedpartition i.e. able to *execute* the nfs service via nfsd daemon. After required capability is satisfied then it will provide a *canacess* capability which represents the capability to access any file in the target host.

```
#include <IDMEFParser>
hset MIRADOR_Alert_Set
{
 int x;
 halert NFSMount ( struct alert IDMEF_Alert )
 {
   alertId MIR0163;
   reqcapability
   {
     capability accessLevel
     {
        SourceAddress IPAdd sc = IDMEF_Alert.source ;
        DestAddress IPAdd dt = IDMEF_Alert.destination;
        action communicate;
        credential user = belongsTo ( AdminGroup);
        service IP;
        property reply;
        mode direct;
        optional false;
    }
    capability mountedPartition
    {
        SourceAddress IPAdd sc = IDMEF_Alert.source;
        DestAddress IPAdd dt = IDMEF_Alert.destination;
        action execute;
```

```
        credential user = belongsTo(AdminGroup);
        service process = nfsd;
        property version;
        mode direct;
        optional false;
    }
  }

  procapability
  {
    capability canaccess
    {
        SourceAddress IPAdd sc = IDMEF_Alert.source;
        DestAddress IPAdd dt = IDMEF_Alert.destination;
        action read;
        credential user = IDMEF_Alert.User;
        service Dir;
        property content;
        mode indirect;
        optional false;
    }
  }
  direction forward;
 }
}
```

5.2.5 Attack Scenario Specification

Attack scenario specification contains complete specification of attack scenarios in capability terms. Structure of this specification is almost equivalent to alert specification i.e. it also has require and provide capability sections. In contrast to alert specification which can be considered as locally defined without a priori knowledge of complete attack scenario, attack scenario specification contains temporal relationship among the capabilities specified in require capability set along with the information of whole attack process. To represent the relationship among capabilities of require capability set, each capability is tagged with the sequence id based on their temporal order in complete multi-step attack scenario. It also contains the verification part which helps in the identification of security state.

Following is the attack scenario specification syntax:

```
{‘#’include ‘<’ Lib_ID ‘>’}
<Global Declaration>
<Function definitions>
hattacksecnario Name ( [Parameters] )
‘{’
   alertId IDs[] ‘;’
   reqCapbility
  ‘{’
      <CapabilitySet with sequence id>
  ‘}’
   procapability
  ‘{’
      <CapabilitySet>
  ‘}’
```

```
   verification_prdicates()
  ‘{’
      <predicates>
  ‘}’
‘}’
```

Following example shows the illegal file access attack given in [**?**] where an unauthorized user tries to read secret file. To do this, he first creates a file then block the printer by opening paper tray. After giving “lpr-s” command to print this file, he overrides the link with the soft link of the secret file. Then he unblock printer and gets the secret file. Attack scenario Specification corresponding to this attack contains array of AlertId which represents alerts required to trigger this attack. From the specification it is clear that to do IllegalFileAcess attack four capabilities are required i.e capability to *block* and *unblock* printer, capability to *print* file and capability to make soft link in the target host. After doing this attack, attacker will gain access to *read* any file in the target access. Verification of this attack can be done by checking the delay in the printing process which is mentioned in the verification block of specification (As value of source and destination address are same in all capabilities therefore we have removed it from example given below).

```
#include<IDMEFparser>
  halert IllegalFileAccess(struct alert IDMEF_Alert)
  {
    alertId MIR0X,MIR0Y;
    reqcapability
    {
       capability blockPrinter
```

```
   {
     action block;
     credential user = IDMEF_Alert.User;
     service process = cups;
     property version;
     mode direct;
     optional false;
     sequence 01;
 }
capability unblockPrinter
{
     action unblock;
     credential user = IDMEF_Alert.User;
     service process = cups;
     property version;
     mode direct;
     optional false;
     sequence 02;
 }
 capability canPrintlpr
 {
     action print;
     credential user = IDMEF_Alert.User;
     service File;
     property path;
     mode direct;
     optional false;
     sequence 01A;
}
capability makeLink
```

```
    {
          action create;
          credential user = IDMEF_Alert.User;
          service File;
          property softlink;
          mode direct;
          optional false;
          sequence 02A;
    }
  }

  procapability
  {
   capability canaccess
   {
       action read;
       credential user = IDMEF_Alert.User;
       service Dir = "\";
       property content;
       mode indirect;
       optional false;
     }
   }
   direction forward;
  }
  verificiation()
  {
    printdelay();
  }
}
```

Developing specification for multi-step attack scenarios is time-consuming and the quality of the specification depends on the specifier's experience and knowledge. Therefore it is important to identify methods for building new attack templates based on previously defined ones in term of capability. Attack scenarios not only help in identifying attack state but also help to characterize common attack techniques from detection point of view. This specification facilitates in designing the database of known attack scenarios which provide reusability of the experience.

5.3 Standard Library & Knowledge base

5.3.1 Standard Library

Libraries of standard definitions of algebraic structures given in figure 5.1. These structures help in designing the correlation process in a systematic and modular fashion. These structures are divided into three classes i.e. operations, relations and inferences. Operations include join, split etc. which represent basic manipulation using one or more capability instance. Relations include overlapped, mutually exclusive, independent relations between capability instances. These relations help in identifying the preconditions to allow specific operations. The whole system is based on require/provide model therefore Inferences are used to determine whether a capability satisfies a required capability. Inferences include comparable, resulting etc, which enumerate the possible inferences from different real life views. Definition of these operations has been discussed in chapter 4.

To make ACMF more useful, these libraries are made modular and loosely

coupled. To facilitate the customization of algebraic structures defined in these libraries, they are being used as plug-in modules.

5.3.2 Knowledge Base

There are three type of knowledge base in the Framework i.e. (i) Conflict knowledge base, (ii) Inference knowledge base, and (iii) H-alert db. H-alert is entered through H-alert specification module. The other two knowledge bases are entered manually. The descriptions of these knowledge bases are as follows

1. *Conflict knowledge base*:- Mutual exclusive is one kind of relation (see section) which gives knowledge about whether two capabilities can exist together. This relation is used to detect false correlation. As these relations require information from security personnel about the values of different attribute (service/action) which cannot coexist. Conflict knowledge base stores this information which is provided by security officers. The main information source for this knowledge base is network architecture, OS, services running etc. For example at the same time at a particular port SSH and HTTP service cannot run in a server.

2. *Inference Knowledge base*:- Inferences are major component which represent causal relationship on the basis of known evidence. There are three kind of inferences described in section . This information is also filled by security personal from their past experience and system environment.

3. *H-alert db*:- H-alert db contains syntactically and consistent H-alert specified by user using H-alert specification. As discussed in section whenever user will extract H-alert from IDS alert through ACML after parsing and

consistency checking, valid H-alert will be stored in H-alert db. Correlation H-alert will be fetched by its identifier.

5.4 Application of the model

We have obtained the IDS alerts for our experiment from Ning [**?**] which are produced by NIDS RealSecure [RealSecure Network Sensor 6.0] [**?**] on the DARPA 2000 intrusion detection evaluation(IDEVAL) dataset [**?**]. This data is a common benchmark to test IDS and alert correlation models. Basic capability model given in [**?**] also used same dataset and developed the capabilities for all 28 different alert signatures.

The designed language has expressive power to represent the alerts from various NIDS into the basic capability model. The language is tested on the alerts (mentioned above) and is able to reduce the alert into the basic capability model which captures twelve documented multistage intrusions in dataset. The language is generic and can be used for even customized capability model. Language is also providing the flexibility to represent the algebraic model of capability and also support their variants of correlation algorithm. We are also planning to test in other dataset like Honeypot database. Moreover we further plan to include heterogeneous security monitoring devices.

The grammar of ACML is simple, unambiguous and easy to understand even for beginners. We have implemented lexical analyzer, parser and pretty printer for all type of specification of ACML. We are in the process of implanting fully fledged framework for and Graphical user interface for the same.

5.5 Summary

This chapter presented a complete Framework for modelling the capability i.e. ACMF. The chapter also proposed Attack Capability modelling language used for the ACMF. Along with ACML specification module it contains standard algebraic library and correlator module. Standard algebraic library contains the algebraic structures defined in chapter 4. Correlator module contains the correlation algorithms. ACML has three specification parts and each one has been explained with suitable examples. Attribute specification part is first speciation part of the language and is used to capture the user defined taxonomy and its semantics so that security personnel can design the whole system according to their network architecture and their security interests. Second specification part i.e. H-alert Specification part is used to model the raw alerts in the form of capability. Last specification is Attack scenario specification part which gives facility to write the stateful description of attack scenarios in terms of capability, which helps to determine the critical security level of the system. The experimental results against the standard benchmarks display the effectiveness of these specifications.

Chapter 6

Conclusion & Future work

6.1 Conclusion

In this work we enriched the capability model and analyzed it in terms of algebraic structures. Capability model is empowered with time information to avoid temporal ambiguity, which impacts in reducing false correlations. In the algebraic structures, we identified and defined relations between capabilities, operations on capabilities and derived inference rules along with their semantics that have been used in the correlation process. The whole capability model is made systematic, consistent and defined properly with algorithms. Comparison between the previous model and the proposed model is exhibited by demonstrating cases where the correlated alerts are not captured by the old model, but are dealt with our proposed model. We have simplified the whole correlation process by making it modular. This makes the system more understandable for an amateur in security. This approach helps in facilitating the process flexibility. Moreover, the modular approach helps in making the enhancement of the framework quite easy. With this systematic model, the system can be automated and adaptive to optimizations.

We proposed the Attack Capability Modelling Framework(ACMF), which along with alert correlation also tracks the threat level of the security system. The framework consists of the tools for the implementation of the algebraic

structure of capability.

We proposed a novel Attack Capability Modelling Language (ACML) which is one of the major contributions of our work. ACML has been developed with context of ACMF. Along with attribute specification and H-alert Specification, the language also provides for a specification of complete attack scenarios in terms of capabilities of the intruder which helps to determine the critical security level of the system. The details of the syntax and informal semantics of ACML are also provided. One of the major features of ACML is that it includes flexibility for customizing the definitions of these structures as well as for customizing the correlation algorithm. This is essential as most often the systems under scrutiny have customized environments (in terms of topology, OS etc.). In addition, customizing the features according to the specific security interests in the system, the number of false alarms as well as redundant true alarms can be minimized significantly. The whole Framework (ACMF) which provides the language suite has been systematically modularized and also suitable for further extension or customization.

6.2 Limitation of the Model

We have also identified some limitations of the framework. Following are major among them

1. The proposed framework tries to minimize the human intervention in translation of IDS alert into H-alert, however, it is still not fully automatic.

2. Either framework offers the opportunity to share the knowledge base

but still administrator knowledge is required to develop it more context based.

3. Join and Split operations which are used in the correlation algorithm have their own advantages and disadvantages. There is a need to optimize them to make the whole process real-time.

4. Currently the framework tracks only the attacker capability and not the defence capability of system which can be useful in making further decisions of a security system for preventing of intrusion. .

6.3 Future work

Following are the research and development areas in which further work can be done in this direction:

- Part of the future work will be to optimize algorithms and to achieve better performance. One possibility is to optimize the algorithm of join operation and to use that in given alternate correlation algorithm (in section 5). This would help in making the system real time with a low false rate.

- Further research can be done to model the defence capability of security personnel. This defence capability will help the administrator in identifying his position against the attacker's capability.

- At this stage, the language supports Network based IDS alerts (in IDMEF format). Part of the future work will be to incorporate heterogeneous security monitoring tools like Host based IDS, Application logs,

vulnerability information etc.

- One future work is to develop the a fully fledged framework along with a Graphical User Interface(GUI) . One possible extensions is to include a debugger for the language.

References

[] Realsecure intrusion detection system,url-*http://www.iss.net*,ISS Inc.

[] Custom attack simulation language (casl), url-*http://www.virtualblueness.net/nasl.html*, secure networks, January 1998.

[] Darpa 2000 intrusion detection evaluation data set, url-*http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/2000data.html*, 2000.

[] Amoroso, E. G. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, Traps, and Response*. Intrusion.Net Books, 1999.

[] Anderson, D., Frivold, T., Tamaru, A., and Valdes, A. Next-generation intrusion detection expert system (nides), software users manual, beta-update release. Tech. Rep. SRI–CSL–95–07, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, may 1994.

[] Andersson, D., Fong, M., and Valdes, A. Heterogeneous Sensor Correlation: A Case Study of Live Traffic Analysis. *IEEE Information Assurance Workshop* (2002).

[] Aranya, A., Wright, C. P., and Zadok, E. Tracefs: A file system to trace them all. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)* (San Francisco, CA, March/April 2004), USENIX Association, pp. 129–143.

[] Arora, S. Dynamic logging framework for network attacks. Master's thesis, New Delhi,India, 2008.

[] Axelsson, S. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC) 3*, 3 (2000), 186–205.

[] Barbará, D., Couto, J., Jajodia, S., and Wu, N. Adam: a testbed for exploring the use of data mining in intrusion detection. *SIGMOD Rec. 30*, 4 (2001), 15–24.

[] Cannady, J. Artificial neural networks for misuse detection. In *Proceedings of the 1998 National Information Systems Security Conference (NISSC'98) . Arlington, VA* (October 1998), pp. 443–456.

[] Caswell, B., Foster, J., Russell, R., Beale, J., and Posluns, J. Snort 2.0 Intrusion Detection.

[] Cheung, S., Lindqvist, U., and Fong, M. W. Modeling multistep cyber attacks for scenario recognition. In *DARPA Information Survivability Conference and Exposition (DISCEX III)* (Washington, D.C., 2003), pp. 284–292.

[] Cohen, W. W. Fast effective rule induction. In *Proc. of the 12th International Conference on Machine Learning* (Tahoe City, CA, July 9–12, 1995), A. Prieditis and S. Russell, Eds., Morgan Kaufmann, pp. 115–123.

[] Cuppens, F. Managing alerts in a multi-intrusion detection environment. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference* (Washington, DC, USA, 2001), IEEE Computer Society, p. 22.

[] Cuppens, F., Autrel, F., Miège, A., and Benferhat, S. Correlation in an intrusion detection process. In *SECI'02: S Ecurit e des Communications sur Internet* (2002).

[] Cuppens, F., and Miege, A. Alert correlation in a cooperative intrusion detection framework. *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on* (2002), 202–215.

[] Cuppens, F., and Ortalo, R. LAMBDA: A language to model a database for detection of attacks. *Proc. of Recent Advances in Intrusion Detection (RAID 2000)* (2000), 197–216.

[] Dain, O., and Cunningham, R. Fusing a Heterogeneous Alert Stream into Scenarios. *Applications of Data Mining and Computer Security* (2002).

[] Dawkins, J., and Hale, J. A systematic approach to multi-stage network attack analysis. *Information Assurance Workshop, 2004. Proceedings. Second IEEE International* (2004), 48–56.

[] Debar, H., Curry, D., and Feinstein, B. The Intrusion Detection Message Exchange Format (draft-ietf-idwg-idmef-xml-12). *IETF Internet Drafts, work in progress* (2004).

[] Debar, H., Dacier, M., and Wespi, A. Towards a Taxonomy of Intrusion-Detection Systems. *Computer Networks 31*, 8 (Apr. 1999), 805–822.

[] Debar, H., and Wespi, A. Aggregation and correlation of intrusion-detection alerts. In *RAID '00: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection* (London, UK, 2001), Springer-Verlag, pp. 85–103.

[] Deraison, R. The nessus attack scripting language reference guide, September 1999.

[] Eckmann, S., Vigna, G., and Kemmerer, R. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security 10*, 1 (2002), 71–103.

[] Endler, D. Intrusion detection applying machine learning to solaris audit data. In *ACSAC '98: Proceedings of the 14th Annual Computer Security Applications Conference* (Washington, DC, USA, 1998), IEEE Computer Society, p. 268.

[] Feiertag, R., Kahn, C., Porras, P., Schnackenberg, D., Staniford-Chen, S., and Tung, B. A Common Intrusion Specification Language (CISL). *At http://gost. isi. edu/projects/crisis/cidf/cisl_current. txt* (1999).

[] Feng, H. H., Giffin, J. T., Huang, Y., Jha, S., Lee, W., and Miller, B. P. Formalizing sensitivity in static analysis for intrusion detection. *sp 00* (2004), 194.

[] Fink, G., O'Donoghue, K. F., Chappell, B. L., and Turner, T. G. A metrics-based approach to intrusion detection system evaluation for distributed real-time systems. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2002), IEEE Computer Society, p. 17.

[] Forrest, S., Hofmeyr, S. A., and Somayaji, A. Computer immunology. *Commun. ACM 40*, 10 (1997), 88–96.

[] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. A sense of self for unix processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1996), IEEE Computer Society, p. 120.

[] Garvey, T. D., and Lunt, T. F. Model-based intrusion detection. In *Proceedings of the 14:th National Computer Security Conference* (Baltimore, MD, USA, Oct. 1991), NIST, National Institute of Standards and Technology/National Computer Security Center, pp. 372–385.

[] Ghosh, A. K., and Schwartzbard, A. A study in using neural networks for anomaly and misuse detection. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium* (Berkeley, CA, USA, 1999), USENIX Association, pp. 12–12.

[] Giffin, J. T., Jha, S., and Miller, B. P. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 61–79.

[] Gosh, A. K., Wanken, J., and Charron, F. Detecting anomalous and unknown intrusions against programs. In *ACSAC '98: Proceedings of the 14th Annual Computer Security Applications Conference* (Washington, DC, USA, 1998), IEEE Computer Society, p. 259.

[] Gu, G., Fogla, P., Dagon, D., Lee, W., and Skorić, B. Measuring intrusion detection capability: an information-theoretic approach. In *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security* (New York, NY, USA, 2006), ACM, pp. 90–101.

[] Han, J., and Kamber, M. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[] Heady, R., Luger, G., Maccabe, A., and Servilla, M. The architecture of a network level intrusion detection system. Tech. rep., LA-SUB–93-219, Los Alamos National Lab., NM (United States); New Mexico Univ., Albuquerque, NM (United States). Dept. of Computer Science, 1990.

[] Hofmeyr, S. A., Forrest, S., and Somayaji, A. Intrusion detection using sequences of system calls. *J. Comput. Secur. 6*, 3 (1998), 151–180.

[] Ilgun, K. USTAT: A real-time intrusion detection system for UNIX. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy* (Oakland, California, 24–26 May 1993), IEEE Computer Society Press, pp. 16–28.

[] Ilgun, K., Kemmerer, R. A., and Porras, P. A.

[] Javitz, H. S., and Valdes, A. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the IEEE Research in Security and Privacy* (Oakland, CA, MAY 1991), pp. 316–326.

[] JDe Boer, P., and Pels, M. Host-based intrusion detection systems. Tech. Rep. Technical Report:1.10, Faculty of Science, Informatics Institute, University of Amsterdam, 2005.

[] Julisch, K. Mining alarm clusters to improve alarm handling efficiency. *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual* (2001), 12–21.

[] Julisch, K. Clustering intrusion detection alarms to support root cause analysis. *ACM Trans. Inf. Syst. Secur. 6*, 4 (2003), 443–471.

[] Julisch, K., and Dacier, M. Mining intrusion detection alarms for actionable knowledge. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2002), ACM, pp. 366–375.

[] Kaufman, L., and Rousseeuw, P. Finding groups in data. an introduction to cluster analysis. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, New York: Wiley, 1990* (1990).

[] Khosravifar, B., and Bentahar, J. An experience improving intrusion detection systems false alarm ratio by using honeypot. *aina 0* (2008), 997–1004.

[] Kim, G. H., and Spafford, E. H. The design and implementation of tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security* (1994), pp. 18–29.

[] Ko, C., Ruschitzka, M., and Levitt, K. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1997), IEEE Computer Society, p. 175.

[] Kruegel, C., and Vigna, G. Anomaly detection of web-based attacks. *Proceedings of the 10th ACM conference on Computer and communication security* (2003), 251–261.

[] Kumar, S., and Spafford, E. H. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference* (Baltimore MD, USA, 1994), NIST, National Institute of Standards and Technology/National Computer Security Center, pp. 11–21.

[] Lane, T., and Brodley, C. E. An application of machine learning to anomaly detection. In *Proc. 20th NIST-NCSC National Information Systems Security Conference* (1997), pp. 366–380.

[] Lee, W., and Qin, X. Statistical causality analysis of infosec alert data. In *Managing Cyber Threats* (2005), Springer US, pp. 101–127.

[] Lee, W., Stolfo, S. J., and Chan, P. K. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI Approaches to Fraud Detection and Risk Management* (1997), AAAI Press, pp. 50–56.

[] Lindqvist, U., Cheung, S., and Valdez, R. Correlated attack modeling (cam). *Final Technical Report*, AFRL-IF-RS-TR-2003-249 (october 2003).

[] Lindqvist, U., and Porras, P. Detecting computer and network misuse through the Production-Based Expert System Toolset(P-BEST). *Doktorsavhandlingar vid Chalmers Tekniska Hogskola* (1999), 161–189.

[] Lonvick, C. The BSD Syslog Protocol, 2001.

[] Lunt, T. F. A survey of intrusion detection techniques. *Comput. Secur. 12*, 4 (1993), 405–418.

[] Michel, C., and Mé, L. ADeLe: an attack description language for knowledge-based intrustion detection. *Proceedings of the 16th international conference on Information security: Trusted information: the new decade challenge* (2001), 353–368.

[] Microsystems, S. SunSHIELD basic security module guide.

[] Morin, B., and Debar, H. Correlation of intrusion symptoms: An application of chronicles. In *RAID* (2003), Lecture Notes in Computer Science, Springer, pp. 94–112.

[] Morin, B., Me, L., Debar, H., and Ducasse, M. M2D2: A Formal Data Model for IDS Alert Correlation. *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002) 137* (2002).

[] Mounji, A. Languages and Tools for Rule-Based Distributed Intrusion Detection. *PhD Thesis, Facultes Universitaires Notre-Dame de la Paix, Namur, Belgium, September* (1997).

[] Neumann, P., and Porras, P. Experience with EMERALD to Date. *1st USENIX Workshop on Intrusion Detection and Network Monitoring* (1999).

[] Ning, P., Cui, Y., and Reeves, D. Analyzing Intensive Intrusion Alerts Via Correlation. *Recent Advances in Intrusion Detection 2516* (2002), 74–94.

[] Ning, P., Cui, Y., and Reeves, D. Constructing attack scenarios through correlation of intrusion alerts. *Proceedings of the 9th ACM conference on Computer and communications security* (2002), 245–254.

[] Ning, P., Cui, Y., Reeves, D., and Xu, D. Tools and Techniques for Analyzing Intrusion Alerts. *ACM Transactions on Information and System Security 7*, 2 (2004), 273–318.

[] Pandey, N. K., Gupta, S. K., and Leekha, S. Algebra for capability based attack correlation. In *WISTP* (2008), vol. 5019 of *Lecture Notes in Computer Science*, Springer, pp. 117–135.

[] Paxon, V. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, TX, USA, Jan. 1988), USENIX, USENIX Association. Corrected version, §7 overstated the traffic level on the FDDI ring by a factor of two.

[] Porras, P., Fong, M., and Valdes, A. A Mission-Impact-Based Approach to INFOSEC Alarm Correlation. *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)* (2002), 95–114.

[] Porras, P. A., and Neumann, P. G. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proc. 20th NIST-NCSC National Information Systems Security Conference* (1997), pp. 353–365.

[] Qin, X., and Lee, W. Attack plan recognition and prediction using causal networks. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 370–379.

[] Ranum, M., Landfield, K., Stolarchuk, M., Sienkiewicz, M., Lambeth, A., and Wall, E. Implementing a generalized tool for network monitoring. *Information Security Technical Report 3*, 4 (1998), 53–64.

[] Rock, S., Cawsey, A., McAndrew, P., and Bental, D. The MIRADOR project: Metadata Improved Relevance Assessment through Descriptions of Online Resources. *Proceedings of the WWW8 Workshop on Virtual Documents, Hypertext Functionality and the Web. Available online: http://www. cee. hw. ac. uk/mirador/WWW8revised. html* (1999).

[] Roesch, M. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration* (1999), USENIX Association, pp. 229–238.

[] Sebring, M. M., Shellhouse, E., Hanna, M. E., and Whitehurst, R. A. Expert systems in intrusion detection: A case study. In *Proceedings of the 11th National Computer Security Conference* (Baltimore, Maryland, october 1988), NIST, pp. 74–81.

[] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2001), IEEE Computer Society, p. 144.

[] Sekar, R., Guang, Y., Verma, S., and Shanbhag, T. A high-performance network intrusion detection system. *Proceedings of the 6th ACM conference on Computer and communications security* (1999), 8–17.

[] Sekar, R., and Uppuluri, P. Synthesizing Fast Intrusion Detection/Prevention Systems from High-Level Specifications. *USENIX Security Symposium* (1999).

[] Staniford, S., Hoagland, J. A., and McAlerney, J. M. Practical automated detection of stealthy portscans. *J. Comput. Secur. 10*, 1-2 (2002), 105–136.

[] Staniford-Chen, S., Tung, B., and Schnackenberg, D. The common intrusion detection framework (cidf). *Proceedings of the Information Survivability Workshop* (1998).

[] Templeton, S., and Levitt, K. A requires/provides model for computer attacks. *Proceedings of the 2000 workshop on New security paradigms* (2001), 31–38.

[] Vaccaro, M., and Liepins, G. E. Detection of anomalous computer session activity. *sp 00* (1989), 280.

[] Valdes, A., and Skinner, K. An Approach to Sensor Correlation. *Proceedings of RAID 2000*.

[] Valdes, A., and Skinner, K. Probabilistic alert correlation. In *RAID '00: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection* (London, UK, 2001), Springer-Verlag, pp. 54–68.

[] Valeur, F. *Real-time intrusion detection alert correlation.* PhD thesis, Santa Barbara, CA, USA, 2006. Adviser-Richard A. Kemmerer and Adviser-Giovanni Vigna.

[] Valeur, F., Vigna, G., Kruegel, C., and Kemmerer, R. A Comprehensive Approach to Intrusion Detection Alert Correlation.

[] Van Jacobson, C., and McCanne, S. TCPDUMP public repository. *Web page at http://www. cpdump. org* (2003).

[] Vigna, G., Eckmann, S., and Kemmerer, R. Attack Languages. *Proceedings of the IEEE Information Survivability Workshop* (2000).

[] Vigna, G., Robertson, W., Kher, V., and Kemmerer, R. A. A stateful intrusion detection system for world-wide web servers. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference* (Washington, DC, USA, 2003), IEEE Computer Society, p. 34.

[] Wagner, D., and Dean, D. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2001), IEEE Computer Society, p. 156.

[] Warrender, C., Forrest, S., and Pearlmutter, B. A. Detecting intrusions using system calls: alternative data models. *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on* (1999), 133–145.

[] Wu, Y.-S., Foo, B., Mei, Y., and Bagchi, S. Collaborative intrusion detection system (cids): A framework for accurate and efficient ids. *acsac 0* (2003), 234.

[] Xu, D. *Correlation Analysis of Intrusion Alerts*. PhD thesis, North Carolina State University, Raleigh, 2006.

[] Xu, D., and Ning, P. Alert Correlation through Triggering Events and Common Resources. *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC04)* (2004).

[] Yegneswaran, V., Barford, P., and Jha, S. Global Intrusion Detection in the DOMINO Overlay System. *Proceedings of NDSS 2004* (2004).

[] Zhai, Y., Ning, P., Iyer, P., and Reeves, D. Reasoning about complementary intrusion evidence. *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC04)* (2004), 39–48.

[] Zhou, J., Heckman, M., Reynolds, B., Carlson, A., and Bishop, M. Modeling network intrusion detection alerts for correlation. *ACM Transactions on Information and System Security (TISSEC) 10*, 1 (2007).

Publications

1. Pandey, N. K., Gupta, S. K., and Leekha, S. Algebra for Attack Capability. In *Microsoft TechVista 2007)*, October 13, 2007.

2. Pandey, N. K., Gupta, S. K., and Leekha, S. Algebra for capability based attack correlation. In *WISTP (2008)*, vol. 5019 of Lecture Notes in Computer Science, Springer, pp. 117135.

3. Pandey, N. K., Gupta, S. K., and Leekha, S. ACML : Capability Based Attack Modeling Language. In *IAS 08*, Naples Italy, 2008 (Accepted).

Appendix A

Syntax of Attribute specification section)

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in this specification are the following:

ActionConnexions	CredentialConnexions	EAddress
IPAddress	Port	RegisterPort
ServiceConnexions	Time	Typedef_name
action	address	attributes
char	connexions	const
credentialGroup	credentialUser	double
enum	float	int
interval	long	service
short	signed	string
struct	union	unsigned
void	volatile	

The symbols used in ACMLAtt are the following:

{	}	:-
;	,	(
)	=	[
]	<	>

Comments

There are no single-line comments in the grammar.

There are no multiple-line comments in the grammar.

The syntactic structure of Attribute specification

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Attribute*⟩ ::= ⟨*Att-decl*⟩ ⟨*Connexion*⟩

⟨*Att-decl*⟩ ::= `attributes` ⟨*Ident*⟩ { ⟨*ActionTerms*⟩ ⟨*CredentialTerms*⟩ ⟨*ServiceTerms*⟩ ⟨*OptionalTerm*⟩ }

⟨*OptionalTerm*⟩ ::= ⟨*AddressTerms*⟩ ⟨*IntervalTerms*⟩
| ⟨*AddressTerms*⟩
| ⟨*IntervalTerms*⟩
| ϵ

⟨*ActionTerms*⟩ ::= `action :-` ⟨*ListAct-val*⟩ `;`

⟨*ListAct-val*⟩ ::= ⟨*Act-val*⟩
| ⟨*Act-val*⟩ `,` ⟨*ListAct-val*⟩

⟨*Act-val*⟩ ::= ⟨*ActionGP*⟩ `(` ⟨*ListActionItem*⟩ `)`

⟨*ActionGP*⟩ ::= ⟨*UIdent*⟩

⟨*ListActionItem*⟩ ::= ⟨*ActionItem*⟩
| ⟨*ActionItem*⟩ , ⟨*ListActionItem*⟩

⟨*ActionItem*⟩ ::= ⟨*Ident*⟩

⟨*CredentialTerms*⟩ ::= ⟨*CredentialGp*⟩ ⟨*CredentialU*⟩

⟨*CredentialGp*⟩ ::= `credentialGroup :-` ⟨*ListCred-val*⟩ `;`

⟨*ListCred-val*⟩ ::= ⟨*Cred-val*⟩
| ⟨*Cred-val*⟩ , ⟨*ListCred-val*⟩

⟨*Cred-val*⟩ ::= ⟨*UIdent*⟩

⟨*CredentialU*⟩ ::= `credentialUser :-` ⟨*ListCredU-val*⟩ `;`

⟨*ListCredU-val*⟩ ::= ⟨*CredU-val*⟩
| ⟨*CredU-val*⟩ , ⟨*ListCredU-val*⟩

⟨*CredU-val*⟩ ::= ⟨*Ident*⟩ `(` ⟨*ListCred-val*⟩ `)`

⟨*ServiceTerms*⟩ ::= ⟨*ListServiceList*⟩

⟨*ListServiceList*⟩ ::= ⟨*ServiceList*⟩
| ⟨*ServiceList*⟩ ⟨*ListServiceList*⟩

⟨*ServiceList*⟩ ::= `service` ⟨*Ident*⟩ { ⟨*Property-list*⟩ }

⟨*Property-list*⟩ ::= ⟨*ListDec*⟩

⟨*ListDec*⟩ ::= ⟨*Dec*⟩
| ⟨*Dec*⟩ ⟨*ListDec*⟩

⟨*AddressTerms*⟩ ::= ⟨*ListAddressList*⟩

⟨*ListAddressList*⟩ ::= ⟨*AddressList*⟩
| ⟨*AddressList*⟩ ⟨*ListAddressList*⟩

⟨*AddressList*⟩ ::= `address` ⟨*Ident*⟩ { ⟨*ListDec*⟩ }

⟨*IntervalTerms*⟩ ::= ⟨*ListIntervalList*⟩

⟨*ListIntervalList*⟩ ::= ⟨*IntervalList*⟩
| ⟨*IntervalList*⟩ ⟨*ListIntervalList*⟩

⟨*IntervalList*⟩ ::= `interval` ⟨*Ident*⟩ { ⟨*ListDec*⟩ }

⟨*Dec*⟩ ::= ⟨*ListDeclaration-specifier*⟩ ;
| ⟨*ListDeclaration-specifier*⟩ ⟨*ListInit-declarator*⟩ ;

⟨*ListDeclaration-specifier*⟩ ::= ⟨*Declaration-specifier*⟩
| ⟨*Declaration-specifier*⟩ ⟨*ListDeclaration-specifier*⟩

⟨*Declaration-specifier*⟩ ::= ⟨*Type-specifier*⟩
| ⟨*Type-qualifier*⟩

⟨*Type-specifier*⟩ ::= `void`
| `char`
| `short`
| `int`
| `long`
| `float`
| `double`
| `signed`
| `unsigned`
| `RegisterPort`
| `Port`
| `action`
| `string`
| ⟨*Struct-or-union-spec*⟩
| ⟨*Enum-specifier*⟩
| `Typedef_name`
| `Time`
| `IPAddress`
| `EAddress`

⟨*ListInit-declarator*⟩ ::= ⟨*Init-declarator*⟩
| ⟨*Init-declarator*⟩ , ⟨*ListInit-declarator*⟩

⟨*Init-declarator*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ = ⟨*Initializer*⟩

⟨*Type-qualifier*⟩ ::= `const`
| `volatile`

⟨*Struct-or-union-spec*⟩ ::= ⟨*Struct-or-union*⟩ ⟨*Ident*⟩ { ⟨*ListStruct-dec*⟩ }
| ⟨*Struct-or-union*⟩ { ⟨*ListStruct-dec*⟩ }
| ⟨*Struct-or-union*⟩ ⟨*Ident*⟩

⟨*Struct-or-union*⟩ ::= `struct`
| `union`

⟨*ListStruct-dec*⟩ ::= ⟨*Struct-dec*⟩
| ⟨*Struct-dec*⟩ ⟨*ListStruct-dec*⟩

⟨*Struct-dec*⟩ ::= ⟨*ListSpec-qual*⟩ ⟨*ListStruct-declarator*⟩ ;

⟨*ListSpec-qual*⟩ ::= ⟨*Spec-qual*⟩
| ⟨*Spec-qual*⟩ ⟨*ListSpec-qual*⟩

⟨*Spec-qual*⟩ ::= ⟨*Type-specifier*⟩
| ⟨*Type-qualifier*⟩

⟨*ListStruct-declarator*⟩ ::= ⟨*Struct-declarator*⟩
| ⟨*Struct-declarator*⟩ , ⟨*ListStruct-declarator*⟩

⟨*Struct-declarator*⟩ ::= ⟨*Ident*⟩

⟨*Enum-specifier*⟩ ::= `enum` { ⟨*ListEnumerator*⟩ }
| `enum` ⟨*Ident*⟩ { ⟨*ListEnumerator*⟩ }
| `enum` ⟨*Ident*⟩

⟨*ListEnumerator*⟩ ::= ⟨*Enumerator*⟩
| ⟨*Enumerator*⟩ **,** ⟨*ListEnumerator*⟩

⟨*Enumerator*⟩ ::= ⟨*Ident*⟩

⟨*Initializer*⟩ ::= ⟨*Exp*⟩
| { ⟨*Initializers*⟩ }
| { ⟨*Initializers*⟩ **,** }

⟨*Initializers*⟩ ::= ⟨*Initializer*⟩
| ⟨*Initializers*⟩ **,** ⟨*Initializer*⟩

⟨*Exp*⟩ ::= ⟨*Constant*⟩
| ⟨*String*⟩

⟨*Constant*⟩ ::= ⟨*Double*⟩
| ⟨*Char*⟩
| ⟨*Unsigned*⟩
| ⟨*Long*⟩
| ⟨*UnsignedLong*⟩
| ⟨*Hexadecimal*⟩
| ⟨*HexUnsigned*⟩
| ⟨*HexLong*⟩
| ⟨*HexUnsLong*⟩
| ⟨*Octal*⟩
| ⟨*OctalUnsigned*⟩
| ⟨*OctalLong*⟩
| ⟨*OctalUnsLong*⟩
| ⟨*CDouble*⟩
| ⟨*CFloat*⟩
| ⟨*CLongDouble*⟩
| ⟨*Integer*⟩
| ⟨*Time*⟩
| ⟨*IPAddress*⟩
| ⟨*EAddress*⟩

⟨*Connexion*⟩ ::= `connexions` { ⟨*ActionC*⟩ ⟨*CredentialC*⟩ ⟨*ServiceC*⟩ }

⟨*ActionC*⟩ ::= `ActionConnexions :-` ⟨*ListAction-attribute*⟩ `;`

⟨*ListAction-attribute*⟩ ::= ⟨*Action-attribute*⟩
| ⟨*Action-attribute*⟩ , ⟨*ListAction-attribute*⟩

⟨*Action-attribute*⟩ ::= ⟨*Actionval*⟩ [⟨*ListContained-action-attributes*⟩]
| ⟨*Actionval*⟩ (⟨*ListContained-action-attributes*⟩)
| ⟨*Actionval*⟩ < ⟨*Actionval*⟩
| ⟨*Actionval*⟩ > ⟨*Actionval*⟩

⟨*Actionval*⟩ ::= ⟨*Ident*⟩
| ⟨*UIdent*⟩

⟨*ListContained-action-attributes*⟩ ::= ⟨*Contained-action-attributes*⟩
| ⟨*Contained-action-attributes*⟩ ,
⟨*ListContained-action-attributes*⟩

⟨*Contained-action-attributes*⟩ ::= ⟨*Ident*⟩
| ⟨*UIdent*⟩

⟨*CredentialC*⟩ ::= `CredentialConnexions` :− ⟨*ListCredential-attribute*⟩ ;

⟨*ListCredential-attribute*⟩ ::= ⟨*Credential-attribute*⟩
| ⟨*Credential-attribute*⟩ , ⟨*ListCredential-attribute*⟩

⟨*Credential-attribute*⟩ ::= ⟨*Credval*⟩ [⟨*ListContained-Credential-attributes*⟩]
| ⟨*Credval*⟩ (⟨*ListContained-Credential-attributes*⟩)
| ⟨*Credval*⟩ < ⟨*Credval*⟩
| ⟨*Credval*⟩ > ⟨*Credval*⟩

⟨*Credval*⟩ ::= ⟨*Ident*⟩
| ⟨*UIdent*⟩

⟨*ListContained-Credential-attributes*⟩ ::= ⟨*Contained-Credential-attributes*⟩
| ⟨*Contained-Credential-attributes*⟩ ,
⟨*ListContained-Credential-attributes*⟩

⟨*Contained-Credential-attributes*⟩ ::= ⟨*Ident*⟩
| ⟨*UIdent*⟩

⟨*ServiceC*⟩ ::= `ServiceConnexions :-` ⟨*ListService-attribute*⟩ `;`

⟨*ListService-attribute*⟩ ::= ⟨*Service-attribute*⟩
| ⟨*Service-attribute*⟩ , ⟨*ListService-attribute*⟩

⟨*Service-attribute*⟩ ::= ⟨*Serviceval*⟩ `[` ⟨*ListContained-Service-attributes*⟩ `]`
| ⟨*Serviceval*⟩ `(` ⟨*ListContained-Service-attributes*⟩ `)`
| ⟨*Serviceval*⟩ < ⟨*Serviceval*⟩
| ⟨*Serviceval*⟩ > ⟨*Serviceval*⟩

⟨*Serviceval*⟩ ::= ⟨*Ident*⟩

⟨*ListContained-Service-attributes*⟩ ::= ⟨*Contained-Service-attributes*⟩
| ⟨*Contained-Service-attributes*⟩ ,
⟨*ListContained-Service-attributes*⟩

⟨*Contained-Service-attributes*⟩ ::= ⟨*Ident*⟩

Appendix B

Syntax of H-alert specification section

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in this specification are the following:

DestAddress	EAdd	Host
IPAdd	SourceAddress	Typedef_name
action	alertId	auto
backward	capability	char
const	credential	direct
direction	double	enum
extern	false	float
forward	halert	hset
indirect	int	long
mode	optional	procapability
property	register	reqcapability
service	short	signed
sizeof	static	struct
true	typedef	union
unsigned	void	volatile

The symbols used in ACML are the following:

```
{       }       #include
<       >       ;
=       ,       :
(       )       [
]       *       ...
?       ||      &&
|       ^       &
==      !=      <=
>=      <<      >>
+       -       /
%       ++      --
.       ->      ~
!       *=      /=
%=      +=      -=
<<=     >>=     &=
^=      |=
```

Comments

Single-line comments begin with `//`.

Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of H-alert specification

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Program*⟩ ::= ⟨*Include-lib*⟩ ⟨*HAlertSet*⟩

⟨*HAlertSet*⟩ ::= `hset` ⟨*Ident*⟩ { ⟨*Global-parameters*⟩ ⟨*ListHAler*⟩ }

⟨*ListInclude-lib*⟩ ::= ⟨*Include-lib*⟩
| ⟨*Include-lib*⟩ ⟨*ListInclude-lib*⟩

⟨*Include-lib*⟩ ::= `#include` < ⟨*Ident*⟩ >

⟨*Global-parameters*⟩ ::= ⟨*ListDec*⟩

⟨*ListHAler*⟩ ::= ⟨*HAler*⟩
| ⟨*HAler*⟩ ⟨*ListHAler*⟩

⟨*HAler*⟩ ::= `halert` ⟨*HAlert-Parameter*⟩ { `alertId` ⟨*Ident*⟩ `;`
⟨*Require-capability*⟩ ⟨*Provide-capability*⟩ ⟨*ListExtra-parameter*⟩ }

⟨*HAlert-Parameter*⟩ ::= ⟨*Direct-declarator*⟩

⟨*Require-capability*⟩ ::= `reqcapability` { ⟨*ListCap*⟩ }

⟨*Provide-capability*⟩ ::= `procapability` { ⟨*ListCap*⟩ }

⟨*ListCap*⟩ ::= ⟨*Cap*⟩
| ⟨*Cap*⟩ ⟨*ListCap*⟩

⟨*Cap*⟩ ::= `capability` ⟨*Ident*⟩ { ⟨*SrcAddress*⟩ ⟨*DstAddress*⟩ ⟨*Action*⟩
⟨*Credential*⟩ ⟨*Service*⟩ ⟨*Property*⟩ ⟨*Mode*⟩ ⟨*Optional*⟩ }

⟨*SrcAddress*⟩ ::= `SourceAddress` ⟨*NetAddress*⟩

⟨*DstAddress*⟩ ::= `DestAddress` ⟨*NetAddress*⟩

⟨*NetAddress*⟩ ::= ⟨*AddressType-specifier*⟩ ⟨*Init-declarator*⟩ ;

⟨*AddressType-specifier*⟩ ::= `IPAdd`
| `EAdd`
| `Host`

⟨*Credential*⟩ ::= `credential` ⟨*Ident*⟩ ;
| `credential` ⟨*Ident*⟩ = ⟨*Cred-Direct-declarator*⟩ ;

⟨*Action*⟩ ::= `action` ⟨*Ident*⟩ ;

⟨*Service*⟩ ::= `service` ⟨*Ident*⟩ ;
| `service` ⟨*Ident*⟩ = ⟨*Initializer*⟩ ;

⟨*Property*⟩ ::= `property` ⟨*Ident*⟩ ;
| `property` ⟨*Ident*⟩ = ⟨*Initializer*⟩ ;

⟨*Mode*⟩ ::= `mode` ⟨*Modeval*⟩ `;`

⟨*Modeval*⟩ ::= `direct`
| `indirect`

⟨*Optional*⟩ ::= `optional` ⟨*Boolean*⟩ `;`

⟨*Boolean*⟩ ::= `true`
| `false`

⟨*Extra-parameter*⟩ ::= ⟨*Direction*⟩

⟨*ListExtra-parameter*⟩ ::= ⟨*Extra-parameter*⟩
| ⟨*Extra-parameter*⟩ ⟨*ListExtra-parameter*⟩

⟨*Direction*⟩ ::= `direction` ⟨*TDirection*⟩ `;`

⟨*TDirection*⟩ ::= `forward`
| `backward`

⟨*Dec*⟩ ::= ⟨*ListDeclaration-specifier*⟩ `;`
| ⟨*ListDeclaration-specifier*⟩ ⟨*ListInit-declarator*⟩ `;`

⟨*ListDec*⟩ ::= ⟨*Dec*⟩
| ⟨*Dec*⟩ ⟨*ListDec*⟩

⟨*ListDeclaration-specifier*⟩ ::= ⟨*Declaration-specifier*⟩
| ⟨*Declaration-specifier*⟩ ⟨*ListDeclaration-specifier*⟩

⟨*Declaration-specifier*⟩ ::= ⟨*Type-specifier*⟩
| ⟨*Storage-class-specifier*⟩
| ⟨*Type-qualifier*⟩

⟨*ListInit-declarator*⟩ ::= ⟨*Init-declarator*⟩
| ⟨*Init-declarator*⟩ , ⟨*ListInit-declarator*⟩

⟨*Init-declarator*⟩ ::= ⟨*Declarator*⟩
| ⟨*Declarator*⟩ = ⟨*Initializer*⟩

⟨*Type-specifier*⟩ ::= `void`
| `char`
| `short`
| `int`
| `long`
| `float`
| `double`
| `signed`
| `unsigned`
| ⟨*Struct-or-union-spec*⟩
| ⟨*Enum-specifier*⟩
| `Typedef_name`

⟨*Storage-class-specifier*⟩ ::= `typedef`
| `extern`
| `static`
| `auto`
| `register`

⟨*Type-qualifier*⟩ ::= `const`
| `volatile`

⟨*Struct-or-union-spec*⟩ ::= ⟨*Struct-or-union*⟩ ⟨*Ident*⟩ { ⟨*ListStruct-dec*⟩ }
| ⟨*Struct-or-union*⟩ { ⟨*ListStruct-dec*⟩ }
| ⟨*Struct-or-union*⟩ ⟨*Ident*⟩

⟨*Struct-or-union*⟩ ::= `struct`
| `union`

⟨*ListStruct-dec*⟩ ::= ⟨*Struct-dec*⟩
| ⟨*Struct-dec*⟩ ⟨*ListStruct-dec*⟩

⟨*Struct-dec*⟩ ::= ⟨*ListSpec-qual*⟩ ⟨*ListStruct-declarator*⟩ ;

⟨*ListSpec-qual*⟩ ::= ⟨*Spec-qual*⟩
| ⟨*Spec-qual*⟩ ⟨*ListSpec-qual*⟩

⟨*Spec-qual*⟩ ::= ⟨*Type-specifier*⟩
| ⟨*Type-qualifier*⟩

⟨*ListStruct-declarator*⟩ ::= ⟨*Struct-declarator*⟩
| ⟨*Struct-declarator*⟩ , ⟨*ListStruct-declarator*⟩

⟨*Struct-declarator*⟩ ::= ⟨*Declarator*⟩
| : ⟨*Constant-expression*⟩
| ⟨*Declarator*⟩ : ⟨*Constant-expression*⟩

⟨*Enum-specifier*⟩ ::= `enum` { ⟨*ListEnumerator*⟩ }
| `enum` ⟨*Ident*⟩ { ⟨*ListEnumerator*⟩ }
| `enum` ⟨*Ident*⟩

⟨*ListEnumerator*⟩ ::= ⟨*Enumerator*⟩
| ⟨*Enumerator*⟩ , ⟨*ListEnumerator*⟩

⟨*Enumerator*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ = ⟨*Constant-expression*⟩

⟨*Declarator*⟩ ::= ⟨*Pointer*⟩ ⟨*Direct-declarator*⟩
| ⟨*Direct-declarator*⟩

⟨*Cred-Direct-declarator*⟩ ::= ⟨*Initializer*⟩
| ⟨*Direct-declarator*⟩ (⟨*Initializer*⟩)

⟨*Direct-declarator*⟩ ::= ⟨*Ident*⟩
| (⟨*Declarator*⟩)
| ⟨*Direct-declarator*⟩ [⟨*Constant-expression*⟩]
| ⟨*Direct-declarator*⟩ [ ]
| ⟨*Direct-declarator*⟩ (⟨*Parameter-type*⟩)

⟨*Pointer*⟩ ::= *
| * ⟨*ListType-qualifier*⟩
| * ⟨*Pointer*⟩
| * ⟨*ListType-qualifier*⟩ ⟨*Pointer*⟩

⟨*ListType-qualifier*⟩ ::= ⟨*Type-qualifier*⟩
| ⟨*Type-qualifier*⟩ ⟨*ListType-qualifier*⟩

⟨*Parameter-type*⟩ ::= ⟨*Parameter-declarations*⟩
| ⟨*Parameter-declarations*⟩ , ...

⟨*Parameter-declarations*⟩ ::= ⟨*Parameter-declaration*⟩
| ⟨*Parameter-declarations*⟩ , ⟨*Parameter-declaration*⟩

⟨*Parameter-declaration*⟩ ::= ⟨*ListDeclaration-specifier*⟩
| ⟨*ListDeclaration-specifier*⟩ ⟨*Declarator*⟩
| ⟨*ListDeclaration-specifier*⟩ ⟨*Abstract-declarator*⟩

⟨*ListIdent*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ , ⟨*ListIdent*⟩

⟨*Initializer*⟩ ::= ⟨*Exp2*⟩
| { ⟨*Initializers*⟩ }
| { ⟨*Initializers*⟩ , }

⟨*Initializers*⟩ ::= ⟨*Initializer*⟩
| ⟨*Initializers*⟩ , ⟨*Initializer*⟩

⟨*Type-name*⟩ ::= ⟨*ListSpec-qual*⟩
| ⟨*ListSpec-qual*⟩ ⟨*Abstract-declarator*⟩

⟨*Abstract-declarator*⟩ ::= ⟨*Pointer*⟩
| ⟨*Dir-abs-dec*⟩
| ⟨*Pointer*⟩ ⟨*Dir-abs-dec*⟩

⟨*Dir-abs-dec*⟩ ::= (⟨*Abstract-declarator*⟩)
| [ ]
| [⟨*Constant-expression*⟩]
| ⟨*Dir-abs-dec*⟩ [ ]
| ⟨*Dir-abs-dec*⟩ [⟨*Constant-expression*⟩]

⟨*Exp*⟩ ::= ⟨*Exp*⟩ , ⟨*Exp2*⟩
| ⟨*Exp2*⟩

⟨*Exp2*⟩ ::= ⟨*Exp15*⟩ ⟨*Assignment-op*⟩ ⟨*Exp2*⟩
| ⟨*Exp3*⟩

⟨*Exp3*⟩ ::= ⟨*Exp4*⟩ `?` ⟨*Exp*⟩ `:` ⟨*Exp3*⟩
| ⟨*Exp4*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ || ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp5*⟩ ::= ⟨*Exp5*⟩ `&&` ⟨*Exp6*⟩
| ⟨*Exp6*⟩

⟨*Exp6*⟩ ::= ⟨*Exp6*⟩ | ⟨*Exp7*⟩
| ⟨*Exp7*⟩

⟨*Exp7*⟩ ::= ⟨*Exp7*⟩ `^` ⟨*Exp8*⟩
| ⟨*Exp8*⟩

⟨*Exp8*⟩ ::= ⟨*Exp8*⟩ `&` ⟨*Exp9*⟩
| ⟨*Exp9*⟩

⟨*Exp9*⟩ ::= ⟨*Exp9*⟩ == ⟨*Exp10*⟩
| ⟨*Exp9*⟩ `!=` ⟨*Exp10*⟩
| ⟨*Exp10*⟩

⟨*Exp10*⟩ ::= ⟨*Exp10*⟩ < ⟨*Exp11*⟩
| ⟨*Exp10*⟩ > ⟨*Exp11*⟩
| ⟨*Exp10*⟩ <= ⟨*Exp11*⟩
| ⟨*Exp10*⟩ >= ⟨*Exp11*⟩
| ⟨*Exp11*⟩

⟨*Exp11*⟩ ::= ⟨*Exp11*⟩ << ⟨*Exp12*⟩
| ⟨*Exp11*⟩ >> ⟨*Exp12*⟩
| ⟨*Exp12*⟩

⟨*Exp12*⟩ ::= ⟨*Exp12*⟩ + ⟨*Exp13*⟩
| ⟨*Exp12*⟩ − ⟨*Exp13*⟩
| ⟨*Exp13*⟩

⟨*Exp13*⟩ ::= ⟨*Exp13*⟩ `*` ⟨*Exp14*⟩
| ⟨*Exp13*⟩ `/` ⟨*Exp14*⟩
| ⟨*Exp13*⟩ `%` ⟨*Exp14*⟩
| ⟨*Exp14*⟩

⟨*Exp14*⟩ ::= `(` ⟨*Type-name*⟩ `)` ⟨*Exp14*⟩
| ⟨*Exp15*⟩

⟨*Exp15*⟩ ::= ++ ⟨*Exp15*⟩
| −− ⟨*Exp15*⟩
| ⟨*Unary-operator*⟩ ⟨*Exp14*⟩
| `sizeof` ⟨*Exp15*⟩
| `sizeof (` ⟨*Type-name*⟩ `)`
| ⟨*Exp16*⟩

⟨*Exp16*⟩ ::= ⟨*Exp16*⟩ `[` ⟨*Exp*⟩ `]`

| ⟨*Exp16*⟩ `(` `)`

| ⟨*Exp16*⟩ `(` ⟨*ListExp2*⟩ `)`

| ⟨*Exp16*⟩ `.` ⟨*Ident*⟩

| ⟨*Exp16*⟩ −> ⟨*Ident*⟩

| ⟨*Exp16*⟩ ++

| ⟨*Exp16*⟩ −−

| ⟨*Exp17*⟩

⟨*Exp17*⟩ ::= ⟨*Ident*⟩

| ⟨*Constant*⟩

| ⟨*String*⟩

| `(` ⟨*Exp*⟩ `)`

⟨*Constant*⟩ ::= ⟨*Double*⟩
| ⟨*Char*⟩
| ⟨*Unsigned*⟩
| ⟨*Long*⟩
| ⟨*UnsignedLong*⟩
| ⟨*Hexadecimal*⟩
| ⟨*HexUnsigned*⟩
| ⟨*HexLong*⟩
| ⟨*HexUnsLong*⟩
| ⟨*Octal*⟩
| ⟨*OctalUnsigned*⟩
| ⟨*OctalLong*⟩
| ⟨*OctalUnsLong*⟩
| ⟨*CDouble*⟩
| ⟨*CFloat*⟩
| ⟨*CLongDouble*⟩
| ⟨*Integer*⟩
| ⟨*Time*⟩
| ⟨*IPAddress*⟩
| ⟨*EAddress*⟩

⟨*Constant-expression*⟩ ::= ⟨*Exp3*⟩

⟨*Unary-operator*⟩ ::= `&`
| `*`
| `+`
| `-`
| `~`
| `!`

⟨*ListExp2*⟩ ::= ⟨*Exp2*⟩
| ⟨*Exp2*⟩ `,` ⟨*ListExp2*⟩

⟨*Assignment-op*⟩ ::= `=`
| `*=`
| `/=`
| `%=`
| `+=`
| `-=`
| `<<=`
| `>>=`
| `&=`
| `^=`
| `|=`