Abstracting and Correlating Heterogeneous Events to Detect Complex Scenarios

by

Sorot Panichprecha

Bachelor of Science (Thammasat University, Thailand) – 1999
Master of Information Technology (QUT) – 2004

Thesis submitted in accordance with the regulations for
Degree of Doctor of Philosophy

Information Security Institute
Faculty of Science and Technology
Queensland University of Technology

March 2009
Keywords

Intrusion detection, signature-based intrusion detection, event correlation, event abstraction, time uncertainty, multi-step attack detection, unification.
Abstract

The research presented in this thesis addresses inherent problems in signature-based intrusion detection systems (IDSs) operating in heterogeneous environments. The research proposes a solution to address the difficulties associated with multi-step attack scenario specification and detection for such environments. The research has focused on two distinct problems: the representation of events derived from heterogeneous sources and multi-step attack specification and detection.

The first part of the research investigates the application of an event abstraction model to event logs collected from a heterogeneous environment. The event abstraction model comprises a hierarchy of events derived from different log sources such as system audit data, application logs, captured network traffic, and intrusion detection system alerts. Unlike existing event abstraction models where low-level information may be discarded during the abstraction process, the event abstraction model presented in this work preserves all low-level information as well as providing high-level information in the form of abstract events. The event abstraction model presented in this work was designed independently of any particular IDS and thus may be used by any IDS, intrusion forensic tools, or monitoring tools.

The second part of the research investigates the use of unification for multi-step attack scenario specification and detection. Multi-step attack scenarios are hard to specify and detect as they often involve the correlation of events from multiple sources which may be affected by time uncertainty. The unification algorithm provides a simple and straightforward scenario matching mechanism by using variable instantiation where variables represent events as defined in the event abstraction model.

The third part of the research looks into the solution to address time uncertainty. Clock synchronisation is crucial for detecting multi-step attack scenarios which involve logs from multiple hosts. Issues involving time uncertainty have been largely neglected by intrusion detection research. The system presented in this re-
search introduces two techniques for addressing time uncertainty issues: clock skew compensation and clock drift modelling using linear regression.

An off-line IDS prototype for detecting multi-step attacks has been implemented. The prototype comprises two modules: implementation of the abstract event system architecture (AESA) and of the scenario detection module. The scenario detection module implements our signature language developed based on the Python programming language syntax and the unification-based scenario detection engine.

The prototype has been evaluated using a publicly available dataset of real attack traffic and event logs and a synthetic dataset. The distinct features of the public dataset are the fact that it contains multi-step attacks which involve multiple hosts with clock skew and clock drift. These features allow us to demonstrate the application and the advantages of the contributions of this research. All instances of multi-step attacks in the dataset have been correctly identified even though there exists a significant clock skew and drift in the dataset.

Future work identified by this research would be to develop a refined unification algorithm suitable for processing streams of events to enable an on-line detection. In terms of time uncertainty, identified future work would be to develop mechanisms which allows automatic clock skew and clock drift identification and correction.

The immediate application of the research presented in this thesis is the framework of an off-line IDS which processes events from heterogeneous sources using abstraction and which can detect multi-step attack scenarios which may involve time uncertainty.
For my parents and Arayapha.
Contents

Keywords

Abstract

Table of Contents

List of Figures

List of Tables

Declaration

Previously Published Material

Acknowledgements

1 Introduction
  1.1 Motivation ...................................................... 2
  1.2 Research Outcomes ........................................... 4
  1.3 Organisation of the Thesis ................................... 5

2 Intrusion Detection Systems ..................................... 7
  2.1 Intrusion Detection Systems: Architecture, Classifications, and Requirements ...................................................... 8
    2.1.1 Architecture of Intrusion Detection Systems ........................................... 9
    2.1.2 Intrusion Detection System Classifications ........................................... 10
    2.1.3 Intrusion Detection Systems: Requirements and Evaluation Methodologies ...................................................... 13
  2.2 Multi-Step Attack Detection Techniques .......................... 15
    2.2.1 State-based Technique ........................................... 15
2.2.2 Event-based Technique ........................................ 17
2.2.3 Evolution of Multi-Step Attack Detection Techniques ...... 19

2.3 Event Representation and Abstraction .......................... 20
  2.3.1 Canonical Event Representation ............................ 21
  2.3.2 Event Abstraction .......................................... 24

2.4 Time Uncertainty ..................................................... 26
  2.4.1 Clock Synchronisation Mechanisms ........................ 27
  2.4.2 Clock Skew and Clock Drift ................................ 28

2.5 Research Challenges ................................................. 29
  2.5.1 Canonical Event Representation ............................ 29
  2.5.2 Comprehensive Multi-Level Event Abstraction .......... 29
  2.5.3 Multi-Step Attack Specification and Detection Mechanisms 30
  2.5.4 Treatment of Time Uncertainty ............................ 31

2.6 Summary .............................................................. 31

3 Abstract Event Model, and Scenario Specification and Detection 33
  3.1 Motivating Example: Failed Administrator Login Attempts .... 34
  3.2 The Abstract Event System Architecture ............................ 36
    3.2.1 Fundamental Concepts ................................... 37
    3.2.2 Components of the Abstract Event System Architecture .... 38
  3.3 Sensor Events ...................................................... 41
    3.3.1 Data Source Schema ...................................... 41
    3.3.2 Sensor Event Tree .......................................... 44
  3.4 The Abstract Event Model ........................................ 45
    3.4.1 Derived Events ............................................ 46
    3.4.2 Abstract Events ............................................ 47
    3.4.3 Modelling Failed Administrator Login Attempts .......... 50
    3.4.4 Discussion ................................................ 52
  3.5 Time Uncertainty .................................................. 53
    3.5.1 Determining Clock Skew ................................... 54
    3.5.2 Constant Skew Compensation .............................. 55
    3.5.3 Clock Drift Modelling with Linear Regression .......... 56
    3.5.4 Discussion ................................................ 58
  3.6 Scenario Specification and Detection .......................... 59
    3.6.1 Unification Background .................................... 60
    3.6.2 Unification in Scenario Detection ......................... 61
3.6.3 The Scenario Detection Engine .................................. 66
3.6.4 Scenario Specification and Example ............................... 68
3.6.5 Discussion ............................................................ 70
3.7 Summary ................................................................. 72

4 The IDS Prototype ......................................................... 73
4.1 Implementation of the IDS Prototype ................................. 74
  4.1.1 Implementation of the Abstract Event System Architecture .... 75
  4.1.2 Implementation of the Scenario Detection Module .............. 77
4.2 Scenario Specification Language .................................... 78
  4.2.1 Language Syntax .................................................... 78
  4.2.2 Signature Translation ............................................... 82
  4.2.3 Signature Composition ............................................. 84
  4.2.4 Discussion ............................................................ 86
4.3 Implementation Issues and Solutions ................................. 90
4.4 Evaluation Methodologies ............................................. 91
4.5 Summary ................................................................. 93

5 Evaluation ...................................................................... 95
5.1 Existing IDS Evaluation Methodologies and Evaluation Criteria ... 96
  5.1.1 IDS Evaluation Methodologies ...................................... 96
  5.1.2 Dataset Classifications .............................................. 97
  5.1.3 Intrusion Detection Evaluation Criteria ........................... 99
5.2 Remarks on Existing Evaluation Methodologies .................... 103
  5.2.1 Receiver Operating Characteristic Curve Issues ................ 103
  5.2.2 Detection Rates and False Positive Rated Issues ................ 104
  5.2.3 Dataset Generation Issues .......................................... 106
5.3 Evaluation of the IDS Prototype ...................................... 107
  5.3.1 Evaluation Methodology ............................................. 107
  5.3.2 Configuration of the IDS Prototype for System Evaluation .... 108
  5.3.3 Evaluation Objectives ............................................... 109
5.4 The Datasets ............................................................. 110
  5.4.1 The Synthetic Dataset ................................................ 111
  5.4.2 The Scan Of the Month Dataset .................................... 113
5.5 Evaluation Results ...................................................... 121
  5.5.1 Validity of Event Processing ....................................... 122
5.5.2 Platform Independent Attack Representation using the AESA 123
5.5.3 Multi-step Attack Specification .............................. 126
5.5.4 Signature Composition ......................................... 127
5.5.5 Attack detection with no Skew Compensation .............. 129
5.5.6 Evaluation of Multi-Step Attack Detection using Constant
Skew Compensation ..................................................... 131
5.5.7 Evaluation of Multi-Step Attack Detection using Linear Re-
gression ................................................................. 134
5.6 Summary .................................................................. 136

6 Conclusions and Future Work ........................................ 139
6.1 Abstract Event System Architecture ............................. 141
6.2 Scenario Detection Engine using Unification .................. 143
6.3 Resolution of Time Uncertainty .................................... 144
6.4 Intrusion Detection System Framework for Detecting Complex Scenarios .............................................. 145
6.5 Conclusion ................................................................ 147

A Data Source Schema and Abstract Event Model ................ 149
A.1 Data Source Schema ................................................... 149
A.2 Abstract Event Model .................................................. 150
A.2.1 Operating System Event Branch ............................... 151
A.2.2 Application Event Branch ......................................... 153
A.2.3 Network Protocol Branch ......................................... 154

B Signature Operator and SQL Operator Mapping ............... 155
B.1 Identity Operators ....................................................... 156
B.2 String Operators ......................................................... 157
B.3 Set Operators ............................................................. 158
B.4 Time Operators (No Timestamp Compensation) ............ 159
B.5 Time Operators (Timestamp Compensation) .................. 160
B.5.1 Constant Skew Compensation Time Operators .......... 160
B.5.2 Linear Regression Time Operators ............................ 161

C Details of the Datasets ................................................... 163
C.1 The Synthetic Dataset .................................................. 163
C.2 The Scan of the Month Dataset .................................... 164
D  Attack Signatures used in the Evaluation  167
  D.1  Signatures for Attacks in the Synthetic Dataset  . . . . . . . . . . . . . 167
  D.2  Signatures for Attacks in the SOTM 34  . . . . . . . . . . . . . . . . . 168

Bibliography  173
# List of Figures

2.1 Basic intrusion detection system architecture (adapted from ISO/IEC TR 15947:2002 [47]). .................................................. 9
2.2 Time line of multi-step attack detection techniques. ............. 20
2.3 A login entry in the Solaris BSM and Bishop’s format (from [13]). 22
2.4 A log entry in UNIX syslog and ULM (from [2]). .................... 23
2.5 TCP denial of service attack model in CARDS (from [75]). ........ 26
3.1 The Abstract Event System Architecture overview. .............. 39
3.2 Components of the AESA. .................................................. 40
3.3 A subset of the DSS. .......................................................... 43
3.4 Subset of application-based and network protocol-based derived events. 48
3.5 The high level abstract events of the AEM. ......................... 49
3.6 The authentication_event branch of the AEM. ...................... 51
3.7 Graph showing clock skew values with more or less constant drift. 57
3.8 Graph of non-continuous clock drift. ................................... 58
3.9 Architecture of the scenario matching using unification. ........... 67
3.10 Pseudo code for the signature of the failed administrator login sce- nario. ................................................................. 69

4.1 Components of the IDS prototype. ........................................ 74
4.2 Table inheritance example in the DSS database schema. .......... 76
4.3 Signature for the failed administrator login scenario. .............. 81
4.4 Signature-to-SQL translation process. ................................... 82
4.5 Translated SQL statement for the failed administrator login signature. 84
4.6 Two sub-signatures of the failed administrator login attempt scenario. 85
4.7 Composite signature of the failed administrator login attempt sce- nario. ................................................................. 86
List of Tables

3.1 Examples of unification. ........................................... 61
4.1 Mapping between signature operators and SQL clauses. ........... 83
5.1 Number of attack instances in the synthetic dataset. ............... 113
5.2 Number of attack instances in the SOTM34 dataset. ............... 121
5.3 Number of recorded events and sensor events derived from the synthetic dataset and the SOTM34 dataset. .................... 122
5.4 Number of derived events from the SOTM34 dataset and our synthetic datasets .................................................. 123
5.5 Detection results (no skew compensation) for the synthetic and SOTM34 datasets. .................................................. 130
5.6 Detection results after applying constant skew technique to the SOTM34 dataset. .................................................. 132
5.7 Detection results after applying the linear regression technique. ... 135
5.8 Detection results, slope, and y-intercept derived from different size of sampling period. .............................................. 136

B.1 Identity operators and their corresponding SQL clauses. ....... 156
B.2 String operators and their corresponding SQL clauses. .......... 157
B.3 Set operators and their corresponding SQL clauses. ............. 158
B.4 Time operators (no compensation) and their corresponding SQL clauses. .......................................................... 159
B.5 Time operators using constant compensation technique and their corresponding SQL clauses. ..................................... 160
B.6 Time operators using linear regression technique and their corresponding SQL clauses. .......................................... 161
C.1 Number of recorded events and logging duration of the synthetic
dataset. .................................................. 164

C.2 Types of log, number of log entries, and logging duration of the
SOTM 34 dataset. .......................................... 165
Declaration

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed: ...........................................  Date: ......................
Previously Published Material

The following papers have been published or presented, and contain material based on the content of this thesis.


Acknowledgements

I would like to express my gratitude to my supervisory team. I would like to thank my principal supervisor, Adjunct Professor George Mohay, for his support, ideas, and guidance throughout the course of my PhD. I could not have written this thesis without his support and his dedication to his students. I would like to thank my associate supervisor Associate Professor Andrew Clark for his support in both technical and nontechnical matters. Although he is one of the supervisors who has the most number of students under supervision, he still dedicated time to discuss problems with me. I would like to thank Dr. Jacob Zimmermann for providing his technical expertise especially in the unification algorithm. Also, I really appreciate his support in writing and editing the two papers that I have published.

Special thanks go to Mark Branagan for the energy and the number of hours and weekends he spent with me editing my thesis especially for grammatical checking and kept me company during the time of writing this thesis. This thesis would not have been completed in time if I did not have help from him. I would like to also thank him for his valuable friendship, discussion about things that happen in the world, random facts that he provides, and coffee making skills.

I would also like to acknowledge my colleagues at the Information Security Institute (ISI) whom I have had discussion with about both research related and non research related topics. Among many there are some I would particularly like to thank, Jason Smith who helped me develop my understanding of real-world applications of information security and who always gave me a view from another perspective to tackle problems. I would like to thank Mehdi Kiani for providing the phpBB attack used in Chapter 5. I would like to thank Mark Branagan and Andrew Marrington for helping me learning Australian culture and politics. I would like to thank my coffee buddies, Mark Branagan, Andrew Marrington, and James Mackie.
The ISI system administrator team, past and present, Sam Lor, Matt Bradford, and James Mackie, have always been helpful with lab support.

I would like to thank my long-time friend, Prachid Tiyapanjanit, for his friendship, Sunday afternoon tea, and many dinners during the preparation of this thesis.

I would also like to express my gratitude to my parents, for their emotional and financial support for my education. Special thanks my two brothers for their emotional support throughout the years of my PhD.

Finally, I wish to express my deep gratitude to my fiancé, Arayapha Siripanich, for her patience, understanding, and emotional support throughout the course of my PhD.
Chapter 1

Introduction

Computer systems and networks including the Internet have become part of everyday life. Business transactions and critical infrastructure operate on computers and successful attacks against such applications and their hosts can cause major interruptions in service and large scale damage. In addition, applications are designed to serve multiple purposes, hence applications and systems they are running on have become complex. This complexity makes it very difficult, if not impossible, to develop and implement such applications and systems that are secure. Systems continue to contain flaws or security vulnerabilities. Flaws in systems and networks can arise from issues such as misconfiguration and software bugs. Software bugs are the prime vulnerabilities for attackers. Details of software bugs are commonly publicly available, and thus the availability of such details contribute to the opportunity of software being the target of attacks. Incidents caused by the exploitations of software bugs have increased every year [8] and so have the number of known software vulnerabilities [20].

There are of course mechanisms which provide protection against and detection of attacks on computer systems and networks. Technologies such as firewalls provide protection to computers and computer networks from unauthorised network access. Detection mechanisms “fill the hole” left by incomplete prevention mechanisms. Intrusion detection systems (IDSs) provide detection of attacks so that incidents caused by attacks can be resolved in a timely manner. Nevertheless, no system is perfect. Although, both protection and detection mechanisms are commonly deployed, attackers can still launch successful attacks against supposedly
protected computer systems and networks as reported by Computer Emergency Response Teams (CERTs) such as the Australian CERT (AUSCERT) [8]. These attacks are in many cases so-called “multi-step” attacks.

A multi-step attack comprises a group of actions where some of these actions may be legitimate but when combined together constitute malicious activities. A multi-step attack is difficult to detect since detection requires the correlation of events recorded by multiple heterogeneous sources. This presents several problems involving event representation and representation, and event correlation.

This thesis focuses on multi-step attack specification and detection in a heterogeneous environment using an off-line signature-based IDS.

Section 1.1 identifies motivation for this research. Section 1.2 identifies the outcomes of this research. Section 1.3 presents the organisation of this thesis.

1.1 Motivation

This thesis focuses on multi-step attack specification and detection in an environment with heterogeneous components. This research is motivated by the following in particular:

- absence of a framework for the canonical representation of events derived from heterogeneous sources;
- absence of a multi-level abstract event model for heterogeneous events;
- time uncertainty in a heterogeneous environment; and
- the need for simpler multi-step attack specification and detection.

A multi-step attack is difficult to specify and detect. In particular, a multi-step attack often involves multiple hosts and multiple event sources and some attack steps may be legitimate. In order to deal with multiple event sources, a canonical event representation is required. There have been efforts to standardise event representation [2, 13, 26], but none of them have been widely adopted. In current IDSs, recorded events (log entries) are transformed into an internal representation which is native to a particular IDS. The native representation is often hard-coded and is part of an IDS, and extending the native representation poses difficulties. A standard representation for events derived from heterogeneous components would provide great benefits for the advancement of multi-step attack specification. A
canonical event representation would enable interoperability amongst IDSs. Such a representation should be designed and developed independently from a particular IDS so that the event representation is flexible and extensible.

In addition to the need for canonical event representation, there is a need for a multi-level abstract event model. Abstract events enable attack signature writers to describe attack characteristics at a high-level or in a platform independent fashion. A multi-level abstract event model represents relationships between concrete and abstract events. A signature using abstract events provides the ability to specify generic signatures which leads to a less number of signatures to be maintained. For instance, to monitor administrator logins in an environment with multiple hosts running different operating systems would otherwise require multiple platform-specific signatures. Without abstract events, to detect all possibilities of the login event would require at least one signature per login service (local login or remote login) per operating system. In this particular example, a single signature can specify the Login event (abstract event) thus identifying all login services on all operating systems. Current abstract event models employ only a flat structure (one or two levels of abstraction) and provide only limited scope for abstraction. As with the canonical event representation framework described above, the abstract event model should be designed and developed independently from any specific IDS, flexible and extensible.

Reliable timestamps are crucial for multi-step attack detection since timestamps are often one of the attributes used for event correlation, in particular, for specifying the chronological orders of events. However, computer clocks are well known for their unreliability. Hence, time uncertainty is a common problem in a computer network even though some form of clock synchronisation (e.g., NTP) is in place. Time uncertainty problems have been studied to some extent in the often supposedly computer forensics field. However, these problems have been largely neglected by IDS research. Time uncertainty problems and solutions have been explored in this research.

Current multi-step attack specification and detection techniques are complex and difficult to use. For instance, the State Transition Analysis Technique Language (STATL) [33] builds on state-based technique where a signature is expressed as states of the system being monitored and transitions between states. There are three types of states and three types of transitions. Signature writers must carefully determine the types of states and transitions to be used in expressing
each step of an attack which may be non-trivial in complex attacks. A signature written in STATL is counter intuitive since a state represents the status of the system being monitored rather than the characteristics of an attack. An alternative multi-step attack specification and detection technique which is simpler than existing techniques is needed.

In summary, this research investigates problems in multi-step attack specification and detection in a heterogeneous environment. A framework for the canonical representation of events derived from heterogeneous sources is explored. The canonical event representation will be capable of representing recorded events derived from both host-based and network-based sensors. Such a representation will be used as the foundation for an abstract event model. An abstract event model will be explored. The abstract event model enables multiple levels of abstraction of event from heterogeneous platforms. An IDS which uses both the canonical event representation and the abstract event model will be developed. A alternative simple multi-step attack detection mechanism will be explored. Resolutions to address timestamps uncertainty caused by time uncertainty will be investigated.

1.2 Research Outcomes

The outcomes of this research are divided into five areas related to multi-step attack specification and detection in a heterogeneous environment. The outcomes are as follows.

A canonical representation of events derived from heterogeneous sources and a multi-level abstract event model. The proposed canonical event representation scheme provides a flexible and extensible representation of events. The flexibility and extensibility enable the integration of the representation of new event types without re-factoring existing event representation. In addition to the event representation, a system architecture for event transformation has been developed. This system architecture provides event abstraction and lays the foundation for multi-step attack specification and detection. The abstract event model builds upon the canonical representation scheme. The proposed event model provides multi-level abstract representation of events derived from heterogeneous sources. The model provides coverage of application, system, and network events. Abstract events defined in the model enable attack signatures to represent attacks regardless of platform and implementation. This feature is needed for an IDS operating in an
environment with heterogeneous components to avoid writing signatures specific to one system and to add flexibility to the IDS.

**Multi-step attack specification and multi-step attack detection engine.** A multi-step attack detection engine based upon unification for signature detection has been developed. A Python-based signature specification language to interface to the proposed attack detection technique has been developed. The signature language and the engine utilises the canonical representation and abstract event model discussed above.

**Approaches to address time uncertainty.** Techniques to address time uncertainty caused by clock skew and clock drift have been explored. These techniques have been implemented and integrated into the attack detection engine.

**Off-line IDS framework for detecting complex scenarios.** This contribution is the integration of the canonical event representation, abstract event model, attack specification and detection engine, and approaches to address time uncertainty. The framework provides building blocks for future research and development in complex and multi-step attack specification and detection in an environment with heterogeneous components. Taking advantage of the flexibility and extensibility of the proposed canonical event representation and abstract event model, the IDS framework can also be used not only in IDS research but also in computer forensics, network monitoring, and security event monitoring.

**Evaluation.** Current IDS evaluation methodologies have been explored. Evaluation criteria in addition to the traditional criteria, i.e., accuracy and completeness, have been defined. The framework prototype has been evaluated using the Scan of the Month (SOTM) [21] dataset and a synthetic dataset.

### 1.3 Organisation of the Thesis

This thesis proposes solutions to inherent problems in multi-step attack specification and detection in an environment with heterogeneous components.

Chapter 2 identifies past and present research into intrusion detection, event representation, event correlation, event abstraction, and time uncertainty. Challenges in these areas will be identified. Existing event representation and research into signature-based IDS will be reviewed. The review focuses on IDSs that detect multi-step attacks. The limitations of existing IDSs will be identified.

Chapter 3 describes the core concepts developed in this thesis: the Abstract
Event System Architecture (AESA) and the multi-step attack detection engine using the unification algorithm. The AESA provides canonical event representation and abstraction of events derived from heterogeneous sources. The concepts and details of the architecture will be discussed. Issues caused by time uncertainty are identified. Solutions to the time uncertainty issues using two techniques will be discussed. The multi-step attack detection engine uses the unification algorithm for attack detection. The design and construction of the signature specification language are based on the unification algorithm. The unification algorithm and its application to attack detection will be discussed.

Chapter 4 describes the IDS prototype that has been developed based on the AESA, the approaches to address time uncertainty, and the unification algorithm. The detailed implementation of the prototype is described. The signature language syntax and rules for constructing signatures are defined.

Chapter 5 explores IDS evaluation methodologies and presents evaluation results for the IDS prototype. An analysis of the prototype evaluation results is presented and discussed.

Chapter 6 concludes the thesis and reviews future directions for the work described in this thesis.
Chapter 2

Intrusion Detection Systems

In Chapter 1, the research goals and research outcomes of this thesis regarding multi-step attack specification and detection in an environment with heterogeneous components have been defined. This chapter describes the historical evolution of intrusion detection systems (IDSs) and relevant past and present research. This chapter also explores current research into event representation and abstraction. Research challenges in multi-step attack specification and detection will be identified.

The concept of using software to automate system auditing processes was introduced by Anderson in the early 1980s [5]. Such auditing processes include identifying malicious events such as events that breach security policies or events that cause damage to the system or network being monitored. The software that implements the concept is referred to as an intrusion detection system. IDSs are needed due to the fact that it is very difficult, if not impossible, to design and implement a practical system that is provably secure for use in any situation.

In general, an IDS operates by reading audit logs or captured network traffic (henceforth audit logs and captured network traffic are referred to as recorded events) and identifying those events that signify intrusions or attacks. The outputs from an IDS are alerts which provide details of the events that trigger the alerts.

One particularly important aspect of IDS research is event correlation. Event correlation enables an IDS to relate multiple recorded events collected from heterogeneous sources for attack detection. An IDS has to establish relationships between these recorded events. The relationships can be chronological orders (e.g.,
before and after) or value referencing (e.g., attributes of events derived from two different sources are alike). Event correlation is an important part of multi-step attack detection.

Traditionally, IDS performance is identified by the number of attacks detected by the IDS (True Positives or TPs) and the number of false alarms (False Positives or FPs). The false alarms here refer to alarms raised by the IDS when there is no attack. The ultimate goal of IDS development is for the IDS to generate maximum TPs (all attacks are detected) while generating minimum or zero FPs. However, in practice, current IDSs are still far from this goal [67].

This chapter is organised as follows. Section 2.1 describes the foundation of an IDS. The generic architecture and classifications of IDS will be described. Requirements of an ideal IDS and IDS evaluation methodologies will be explored. The evolution of multi-step attack detection techniques are explored in Section 2.2. Section 2.3 explores current canonical event representation and abstract event models. Section 2.4 investigates time uncertainty problems. Section 2.5 identifies research challenges in the area of multi-step attack specification and detection in a heterogeneous environment. Section 2.6 summarises the chapter.

2.1 Intrusion Detection Systems: Architecture, Classifications, and Requirements

This section describes basic concepts of IDS with respect to architecture, classifications, requirements of IDS, and evaluation methodologies.

IDS functions can be simplified into three operations: event collection, analysis, and response. Event collection refers to the aggregation of recorded events from their sources. Analysis refers to the attack detection process. Response refers to the feedback of an IDS when an attack is detected such as raising an alarm. These operations may be implemented as separate software components or integrated into one software. The detail of IDS architecture based on these operations is described in Section 2.1.1.

IDS can be divided into several classes based on different characteristics of an IDS. The classification is based upon detection scope and capabilities of an IDS. The purpose of such classification is to provide categories of IDS so that system implementers can choose the class of IDS that fits with the operational environment. Section 2.1.2 describes the two most common IDS classifications.
The main objectives of IDS research and development are to increase accuracy and completeness. The accuracy of intrusion detection refers to the ability of an IDS to correctly identify malicious events as attacks. However, in some cases, an IDS may falsely report alarms when there is no attack. An alarm generated by a false report is referred to as a false positive. The completeness of intrusion detection refers to the ability of an IDS to detect all instances of attacks in the system or network being monitored. However, there are also other aspects of IDS requirements which should be taken into account. These requirements as well as IDS evaluation methodologies are identified and discussed in Section 2.1.3.

2.1.1 Architecture of Intrusion Detection Systems

The ISO/IEC TR 15947:2002 [47] standard defines a generic architecture of IDSs based on the functions of intrusion detection. The architecture comprises components of the following type: a sensor, an analyser, a response module, and a repository. Figure 2.1 shows the relationships between these components. In practice, these components may be integrated into one piece of software or they may be implemented separately. Multiple instances of each of these modules are possible. Nevertheless, the architecture of an IDS can be simplified into these four components. The functions of each component are described as follows.

An IDS sensor collects recorded events from applications, systems or networks being monitored. There are two methods of collecting recorded events: on-line
and off-line. In the on-line event collection, a sensor records events occurring in the environment being monitored [77, 90]. In the off-line event collection or batch mode, a sensor reads recorded events from files generated by applications, systems, or network traffic capture [33]. After recorded events have been collected, they are converted into an appropriate representation that is recognised by the corresponding IDS analyser. The converted recorded events are, then, sent to the analyser for analysis or stored in the repository for further analysis.

An analyser determines whether an event or a group of events are malicious. Outputs from an analyser consist of information about an attack including details of events that signify an attack. The outputs are forwarded to a response module to take appropriate responses. An analyser may store outputs in a repository for further analysis.

A response module provides an interface to human operators. A response module receives events that signify attacks from the analyser. The most common response of an IDS is to report alarms to human operators so that they can further investigate incidents. A response module may also store the alarms in a repository for human operators to investigate them at a later time. A response module can, in some cases, reconfigure a system to minimise the damage caused by the attack or to prevent the same attack occurring in the future.

A repository is used to store recorded events and IDS alarms. Both recorded events and alarms may be used by an analyser to refute or verify that an attack has actually occurred.

In addition to these four components, some IDS may include a management module (not shown in Figure 2.1) which is used to control operations of IDS components, e.g., start/stop functions and control behaviour of analysers and response modules.

We now explore IDS classifications.

### 2.1.2 Intrusion Detection System Classifications

Intrusion detection system classifications are used to determine the scope and capability of an IDS. This section explores the two most common IDS classifications. The detail of the classifications are described as follows.
2.1. Intrusion Detection Systems: Architecture, Classifications, and Requirements

Classified by Source of Event

This classification identifies an IDS by the source of recorded events. A host-based 
**IDS (HIDS)** has its sensor reside on a host where the sensor monitors operating 
system audit data or application logs. Examples of such systems are: USTAT [42] 
and EMERALD eXpert [59]. Since the sensor is installed on a host, the sensor 
can extract a lot of information from the host. However, in an environment with 
a large number of hosts, installing and maintaining sensors on all hosts is difficult. 
Network-based IDSs were introduced to address this problem.

A network-based IDS (NIDS) uses a sensor which monitors network traffic. 
The sensor is connected to the network to be monitored. Examples of such systems 
are Snort [90] and Bro [77]. A NIDS addresses the intrinsic problems in HIDS 
where HIDS sensors must be installed on all hosts to be monitored whereas in 
NIDS only one sensor which monitors network traffic is required. However, there 
are three major limitations in NIDS. Firstly, a NIDS cannot monitor encrypted 
network traffic, e.g., HTTP over SSL, Secure Shell, and VPN traffic. Secondly, 
a NIDS cannot verify attack results (whether they succeed or fail) due to lack of 
information from the host under attack. Thirdly, a NIDS provides less information 
about a host compared to HIDS as NIDS can see only network traffic. These 
problems have led to the development of the hybrid IDS.

A hybrid IDS incorporate elements of both HIDS and NIDS, and thus 
addresses the limitations of the two types of IDS. Examples of such systems are the 
STAT Framework [109] and the Prelude IDS [108]. A hybrid IDS can have both 
host and network sensors and thus it detects both host-based and network-based 
attacks. However, the main obstacle for the development of hybrid IDS is the 
absence of standard representation of recorded events. Recorded events from hetero-

geneous sources have to be dealt with on an ad hoc basis. There exist some 
efforts to standardise event representation [2, 13, 18, 26], but such efforts represent 
only host-based events (cannot represent network-based events) or specific to one 
platform.

Classified by Intrusion Detection Approach

In general, there are two approaches to intrusion detection: anomaly-based detec-
tion and signature-based detection. One may argue that there is also a specification-
based approach. But we consider specification-based approach as a sub-type of the 
anomaly-based approach.
Chapter 2. Intrusion Detection Systems

The **anomaly-based** detection approach identifies events that *deviate* from normal behaviours as attacks. The approach generally involves statistical analysis. An anomaly-based intrusion detection approach commonly consists of two stages of operation: *learning* normal behaviour and *detecting* abnormal behaviour. During the learning period, profiles of normal system behaviour are built. The learning process has to be performed periodically to update the profiles. After the *learning* period, the approach switches into the *detection* stage where activities that exceed deviation thresholds are identified as attacks. The main advantage of the anomaly approach is the ability to detect novel attacks. However, there are two well-known limitations of the approach. Firstly, anomaly-based IDSs, in practice, tend to generate a high FP rate. The high FP rate is caused by flaws in the underlying assumption of the anomaly approach which defines events that deviate from normal behaviour as attacks. In practice, abnormal events are not necessarily attacks. Secondly, there are difficulties in training an anomaly-based IDS due to the fact that during the learning period, the behaviour in the system must be clean, i.e., no attack activities occur during training. Adversaries may insert attack events into a system during the learning period so that the IDS recognises attack events as normal behaviour, and thus such attacks cannot be detected. Sanitising the training data (removing attack events) is very difficult if not impossible. If the sanitising process is too strict, when the IDS operates in a real environment may result in a high FP rate. If the sanitising process is too relaxed, adversaries may introduce attack events into the system. Examples of systems that use the anomaly-based approach are Anomaly Detection of Web-Based Attacks [53] and PAYL [111].

We considered a specification-based approach as being a variation of the anomaly-based approach. The specification-based approach replaces the learning stage with the specification of system behaviour. Such a specification describes behaviour of a system when the system is in operation. Behaviours that deviate from the specification are considered malicious. The drawback of the specification-based approach is due to the fact that the specification of system behaviour is tedious to develop and time consuming. The specification must be thought through, i.e., cover all possible cases. An example of a specification-based IDS is [94].

The **signature-based (misure-based)** detection approach identifies attacks by reporting events that match the descriptions of well-known attacks (the so-called signatures). The approach builds on the principle that attacks leave some
detectable traces where such traces can be expressed using signatures. During attack detection, the approach comprises two steps: signature selection and evaluation. The signature selection process involves an IDS choosing a signature to compare to recorded events. The simplest selection mechanism, used by most systems, is to select signatures sequentially. There are also other mechanisms such as selecting signatures based on the properties of the signatures [52]. This mechanism reduces the number of signatures that must be evaluated against recorded events. After a signature has been selected, recorded events are evaluated against the signature. The primary limitation of signature-based IDS is the inability to detect novel attacks. Thus, to compensate for this limitation, signature-based IDSs require frequent update of the list of signatures. Examples of such systems are Snort [90], the STAT framework [109], EMERALD eXpert [59], and Bro [77]. The detail of some of these work are presented in Section 2.2.

We now identify IDS requirements in addition to accuracy and completeness. Also, existing IDS evaluation methodologies will be identified.

### 2.1.3 Intrusion Detection Systems: Requirements and Evaluation Methodologies

#### Requirements

Two ultimate goals of IDS research are to maximise **accuracy** and **completeness**. In addition to these two goals, there are several other requirements which should also be met by an IDS. In [11, 24, 77], several requirements of an IDS were identified. The detail of such requirements is described as follows.

An IDS must **run continually** with **minimal overhead**. The IDS should operate with little to no need for interaction with human operators. Also, the IDS should not overload the system on which the IDS is run. If the IDS requires a large amount of resources such as memory and processing power, the IDS itself may be the victim of a denial of service (DoS) attack. Also, an IDS should be **resistant to attacks** and **fault tolerant**. IDS designers and implementers should assume that the IDS itself will be attacked at some point. For instance, an adversary may obtain the source code of a particular open source IDS such as Snort or Prelude IDS. Analysing the source code, the adversary may gain knowledge about vulnerabilities in the IDS and thus the IDS can be compromised [48]. IDS implementers should be aware that an IDS may be the target of an attack, and thus the implementers
should provide mechanisms to recover after attacks (fault tolerance). The IDS should be capable of **handling a large volume of data**. This may not be directly related to resistance to attacks but if an IDS cannot handle large amount of data, the size of the data may cause an IDS to potentially be vulnerable to DoS attack.

In terms of installing and maintaining an IDS, the IDS must be **configurable**. An IDS should allow system operators to reconfigure the IDS to suit (such as ensuring the IDS complies with security policies) the environment in which the IDS is installed. In the case of a distributed IDS, the IDS must support dynamic re-configuration where it allows system operators to change the operation of IDS components with little impact on the detection.

**Evaluation**

Traditionally, IDS evaluation measures the accuracy and completeness of an IDS by running an IDS against a *labelled dataset*. A dataset is a set of *recorded events* collected from either a real working environment (or a so called *real dataset*) or an environment which is specifically implemented for IDS evaluation purposes (or the so called *synthetic dataset*). A labelled dataset is a dataset where all *recorded events* have been clearly identified as being either legitimate events or malicious events.

A *real dataset* allows an IDS being evaluated to be tested in an environment similar or close to the real environment. Despite this advantage, a *real dataset* has three main limitations [68]. Firstly, the environment is not controllable. Hence, the person evaluating an IDS may not be able to evaluate all requirements identified above. Secondly, the *real dataset* is difficult to label. Due to the fact that the environment is not controlled, it may be difficult to distinguish between legitimate events and malicious events. Thirdly, a *real dataset* has privacy issues. Since, the dataset is collected from a real working environment where the dataset may contain confidential information. This issue may be addressed by anonymising (removing sensitive information) the dataset. However, the anonymising process may also remove information that could be crucial for attack detection.

A *synthetic dataset* enables IDS evaluators to design an environment so that the dataset can be customised to include specific tests for several of the requirements of an IDS. Since the dataset is generated synthetically, it is easy to label the dataset. The limitation of a *synthetic dataset* is that the dataset may not accurately reflect
2.2 Multi-Step Attack Detection Techniques

There are two commonly used techniques for representing and detecting multi-step attacks: state-based and event-based. The state-based technique represents a multi-step attack as a sequence of states starting from the safe state and finishing at the compromised state. A state represents the status of the system or network being monitored or the progression of the attack. Each pair of states is connected by a transition where a transition is a single action that modifies the state. In the state-based technique, a multi-step attack is detected when the state reaches the compromised state.

The event-based technique detects a multi-step attack based on a sequence of events with no explicit regards for states. In the event-based technique, a multi-step attack is detected when all events that match the descriptions (a signature) of a multi-step attack are detected.

This section examines current intrusion detection techniques for detecting multi-step attacks. Signature-based IDSs using either the state-based technique or event-based technique are now reviewed.

2.2.1 State-based Technique

State Transition Analysis Technique (STAT) was introduced by Porras and Kemmerer [80]. The STAT concept builds on a state machine technique where a state represents the status of the system or network being monitored and a transition represents an event that has an impact on the state. An attack is a sequence of actions (transitions) that lead from an initial state to a compromised state. The first prototype, USTAT [42], monitors events on the Solaris operating system. USTAT implements three types of states (initial, intermediate, and terminal) and one type of transition, i.e., forward transition. USTAT provides attack detection at operating system (or host) level. NetSTAT was introduced in [110]. NetSTAT provides detection of network-based attacks.
An improved STAT system is proposed in [33]. The concept of transition has been redesigned. The new design of transitions has become the foundation for later work in STAT. There are three types of transitions: consuming, nonconsuming, and unwinding. A consuming transition represents a step that advances a state and makes the previous state invalid. A consuming transition is used in the case where an event can occur only once. For example, a particular file can be deleted only once. In particular, when a file is deleted, the consuming transition causes the state to advance from the “file exist” state to the “file deleted” state, and thus, the file exist state becomes invalid. A nonconsuming transition represents a step that creates a copy of the current state then advances the copied state. For a nonconsuming transition, the previous state is still valid. For example, a particular user logging into a system can be represented by a nonconsuming transition as a user can login to a system multiple times (given a system that supports multiple logins such as a UNIX-based operating system). In other words, if a particular user logs into a system, the same user can still log into the system using another login session. An unwinding transition represents a step that moves a state to the previous state. An example of an unwinding transition is when a user logs out of a particular session on a host. The state is reverted to a previous state, i.e., before a user logs into the host. This enhanced concept adds flexibility to the original STAT design but, at the same time, it adds more complication especially for writing a signature. Signature writers must have an intimate understanding of the transitions and carefully choose the correct types of transition when writing a signature.

STAT participated in the well-known DARPA intrusion detection evaluation [57, 61, 62]. The 1999 off-line DARPA evaluation [61] contains five types of attacks: DoS, probe, remote-to-local, user-to-root, and accessing confidential files without appropriate privileges. It consisted of hosts running four operating systems: Solaris, SunOS, Windows NT, and Linux. On the overall results [60], STAT generated less than 10 false alarms per day during the two weeks evaluation period. STAT detected all user-to-root attacks (only in the Solaris operating system) and accessing confidential files on the Solaris operating system. However, on other types of attacks, STAT detected only around 60% of attacks.

Attack graphs [78] represent multi-step attacks as states and transitions. An attack graph consists of a sequence of states which represent the progression of a multi-step attack. Each node in the attack graph represents the attack state, i.e.,
the progression of the attack or the effect of the attack on a system or network. An edge represents an activity (performed by an attacker) that modifies the state of the system or network being modelled. The final state of an attack graph represents the goal that is achieved by the attacker, e.g., the attacker gains an unauthorised access to a server.

Attack graphs can be employed to enhance the accuracy of an IDS [50]. Given an attack graph representing all possible attacks to achieve a particular goal, if a signature-based IDS contains signatures for all possible paths of the graph then such attacks will be detected. Hence, attack graphs are complementary to multi-step attack signature development.

State-based IDSs and attack graphs have in common that both build on the finite state machine concept. A STAT signature describes an attack in terms of states and transitions while attack graphs describe attacks at a more abstract level. Since attack graphs describe attacks at the abstract level, attack graphs can be used as a guide for developing STAT signatures.

2.2.2 Event-based Technique

Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) [81] is a distributed IDS framework built on rule-based expert system techniques. A collection of recorded events collected from event sources is considered to be a fact base (or database of facts) and attack signatures are considered to be rules. The language used to write rules is called Production-Based expert System Toolset (P-BEST) [59]. A signature comprises a list of event predicates and consequences which is conceptually represented as IF... THEN..... Attack detection is performed by comparing a set of signatures against a fact base. If there exists an event in the fact base that satisfies a rule, the results can be either, report an alert, derive a new fact, or delete an existing fact from the fact base. The rule-based expert system technique employed by EMERALD, in fact, resembles part of state-based techniques, i.e., a new fact (derived fact) can be considered to be new state (after a transition). However, a derived fact in the rule-based expert system technique is not limited to the state of a system being monitored, such as the number of bad login has reached a specified threshold where this number is counted by the EMERALD matching engine.

EMERALD participated in the DARPA intrusion detection evaluation [60]. EMERALD generated less than 10 false alarms per day during the two weeks
evaluation period. It detected all user-to-root attacks on the Solaris operating system. On the overall results, EMERALD detected more attacks than STAT.

**Correlated Attack Modelling (CAM)** was conducted at the SRI International [19]. CAM is a framework for detecting complex multi-step attacks using IDMEF-based alerts derived from multiple IDSs. CAM provides a language, the Correlated Attack Modelling Language (CAML) for expressing alert correlation steps. A multi-step attack description written in CAML is referred to as a *module*. A module specification comprises three sections: event, pre-condition, and post-condition. The event section specifies a set of events which will trigger a particular module. The events can be either IDS alerts or inferred events (events generated by another module). The pre-condition section specifies conditions of events specified in the event section. The post-condition section specifies the outcomes of the module, i.e., new events. The new events can be considered to be alerts or can be sent to another module. This feature allows CAML to specify a chain of events where the post-condition of the prior module must match the pre-condition of the following module. The event chain is necessary for specifying complex multi-step attacks where each step is considered to be one module. The implementation of CAM was built upon EMERALD and CAML was based on P-BEST. Modules written in CAML must be manually translated into signatures specified in the P-BEST language. After the translation, P-BEST signatures are used by EMERALD to detect attacks.

No evaluation results of CAM have been presented in the literature.

**Adaptable Real-time Misuse Detection System (ARMD)** [56] implements the event-based attack detection technique that in some part resembles the state-based technique. The attack detection mechanisms in ARMD are similar to the rule-based expert system in EMERALD except when an event matches a signature, the outputs are alerts whereas in EMERALD such outputs are either derived facts which are inserted into the *fact base* or alerts. The part that resembles the state-based technique is in the way a signature is specified. States in ARMD refer to the state of signature matching whereas in STAT, states refer to the status of the system or network being monitored. The signature specification language in ARMD is called the abstract misuse signatures (MuSigs). MuSigs require signature writers to specify the terminal state (or the so called *sink node*) of a signature which indicates that an attack is detected.

Evaluation results of ARMD are not available in the reviewed literature.
Coordinated Attacks Response and Detection System (CARDS) [113] is the successor of ARMD. CARDS employs the same attack detection mechanisms. The goal of CARDS is to provide a decentralised distributed IDS. CARDS provides a more flexible IDS architecture which overcomes the single-source characteristic of ARMD. In [74], authors presented a signature for Mitnick attack. However, evaluation results and performance of CARDS have not been presented.

2.2.3 Evolution of Multi-Step Attack Detection Techniques

Figure 2.2 shows the time line of the evolution of state-based and rule-based multi-step attack detection techniques described above based on the year of publication. The state-based technique used in STAT was first proposed in 1992 [80]. In 2000, the fundamental state-based technique employed by STAT has been redesigned. Three types of transitions have been introduced into the new design. In 2003, the architecture of STAT was extended from a standalone architecture (a single sensor) to a centralised architecture (multiple sensors with a management module) [109]. The architecture comprises a central management console and multiple types of STAT sensors.

Event-based techniques have been employed in EMERALD and ARMD and their successors. EMERALD employs a rule-based expert system technique where signatures are expressed in P-BEST [59]. In 2003, CAM was introduced as an alert correlation framework based on the EMERALD attack detection engine.

ARMD was introduced in as a standalone multi-step attack detection system. CARDS is the successor of ARMD employs the same multi-step attack detection mechanisms. However, CARDS emphasises the detection of distributed attacks (multi-step attacks that target multiple hosts).

In summary, state-based and event-based techniques have been employed in several systems. There is one major drawback with these two techniques which is the complexity in attack specification and detection. For example in state-based techniques, the complexity exists in multi-step signature specification as there are multiple types of states and transitions. Hence, signature writers must have an intimate understanding of signature instantiation and matching. Also, representing an attack using states of the system or network being monitored is counter-intuitive. For signature writers, a signature should describe the characteristics of attacks rather than the states of the system or network under attack. In event-based techniques such as the technique employed in EMERALD [59], the
syntax of EMERALD signature is difficult to read as it is combination of the native EMERALD language (P-BEST) and the C language.

We now investigate efforts to standardise event representation. Also, existing event representation models will be explored.

### 2.3 Event Representation and Abstraction

To detect multi-step attacks in an environment with heterogeneous components, an IDS must first recognise the syntax of the recorded events derived from heterogeneous sources. Currently, there is no standard representation of recorded events. The absence of a standard event representation causes IDSs to parse such events on an ad hoc basis. A standard event representation would provides several benefits to IDS research. Firstly, IDS development time can be shortened by using standard APIs for reading and parsing recorded events. IDS developers do not need to develop recorded event parsers in an ad hoc fashion. Secondly, interoperability between different components within an IDS or between IDSs can be achieved more easily.

In addition to standard event representation, abstract events are also important for representing events in a heterogeneous environment. Abstract events provide several benefits for writing signatures for a heterogeneous environment. Firstly, abstract events enable signature writers to write generic signatures. A generic signature is an attack signature that represents an attack regardless of platform or
implementation. For example, a read file (abstract event) that represents an event of a file being read on any platform. If read file is used in an attack signature, such a signature can will be triggered by a read activity on a particular file on any platform. Secondly, an abstract event reconciles recorded events derived from multiple sources. For instance, an HTTP request event (abstract event) represents recorded events derived from a web server and a network sensor. The two event sources provide information of a particular event from different perspectives.

Section 2.3.1 explores efforts to standardise event representation. The syntax of such representation will be described and limitations identified. Section 2.3.2 explores abstract event models in IDSs and computer forensic tools.

2.3.1 Canonical Event Representation

Despite several efforts [2, 13, 18, 26] to standardise event representation (including IDS alerts) derived from heterogeneous sources, none of these efforts have been widely adopted. Current IDSs still have to define an internal native representation of recorded events which are ad hoc and thus such a representation cannot be shared with other systems. The absence of a canonical event representation is one of the main issues when developing an IDS to operate in an environment with heterogeneous components. Current canonical event representation are now explored.

Bishop proposed syntax to standardise the representation of host events (henceforth called the Bishop’s format) [13]. The design of Bishop’s format focuses on two properties of event representation: extensibility and portability. For the extensibility, the format allows each log entry to have an arbitrary length. For the portability, the format adopts the printable ASCII representation. Such a representation addresses byte ordering, character representation, and floating-point issues.

A log entry in Bishop’s format comprises attribute names, attribute values, field separators, and special fields. The attribute names are defined by the implementation of the log sources. The attribute values are derived from the actual log entries. The field separators separate two attributes and special fields and attributes. The default field separator is ‘#’. The special fields indicate the properties of the log entry. For example, “#S#”, “#E#”, and “#I#” mark start and end of a log record and indicate that next field should be ignored respectively. An example of a Solaris BSM user login log presented in Bishop’s format (from [13])
is shown in Figure 2.3.

<table>
<thead>
<tr>
<th>Log Format</th>
<th>Sample Log Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bishop</td>
<td>#S#event=AUE_EXIT#date=09181991@113528 #usedtime=570000#logid=bishop#I#ruid=root#euid=root #egid=daemon#procid=1234#errno=0#reval=5#E#</td>
</tr>
</tbody>
</table>

Figure 2.3: A login entry in the Solaris BSM and Bishop’s format (from [13]).

Bishop’s format has not been widely implemented. To the best of our knowledge, there has been only one work [54] which employs Bishop’s format. There are several limitations in Bishop’s format. Firstly, the format is too generic. The format neither defines standard attribute names nor attribute types. Hence, there are semantics issues where attributes in one event source may have a different meaning in another. Secondly, the format can be used to represent only printable characters (ASCII format). The format cannot represent binary based log entries. Thus, Bishop’s format cannot represent captured network traffic events such as recorded events in PCAP format.

*Universal Logger Message (ULM)* [2] was proposed as an extension to UNIX syslog [112]. The UNIX syslog has a well know issue regarding the absence of uniform semantics of the syslog message body. Such an issue poses a difficulty for developing a robust syslog parser. The ULM aims to add semantics to syslog message to ease syslog parser development. The ULM defines a set of attribute names to be assigned to fields in syslog messages. However, some of the attribute names defined by the ULM could not be derived from syslog messages, e.g., logging level, state of the process, and ID of the physical connection of the log host. Hence, some attributes of logs in ULM require out-of-band information. An example of a syslog message and the ULM representation of the same message is shown in Figure 2.4 (from [2]). The log shows user ‘tuttle’ connected to host 10.0.2.1 from 10.1.3.5. The same log entry in ULM adds attribute names, log level, process id, and process name which are derived from the documentation of UNIX syslog and the knowledge of human operators.

There are some similarities between ULM and Bishop’s format. ULM has a narrower scope and can only be applied to the UNIX syslog. ULM defines a set of
### Log Format Sample Log Contents

<table>
<thead>
<tr>
<th>Format</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>syslog</td>
<td>Jan 20 00:05:03 itesec: 10.0.2.1 tuttle from space.foo-bar.com (10.1.3.5) 3456</td>
</tr>
<tr>
<td>ULM</td>
<td>DATE=19970120000503 HOST=itesec PROG=foo-gw LVL=debug PS=3456 DST.IP=10.0.2.1 DSTUSR=tuttle SRC.IP=10.1.3.5 SRC.FQDN=space.foo-bar.com</td>
</tr>
</tbody>
</table>

Figure 2.4: A log entry in UNIX syslog and ULM (from [2]).

uniform attribute names not defined in Bishop’s format. ULM has not been widely adopted because it is difficult, if not impossible, to define all possible attribute names. The UNIX syslog allows any application running on the UNIX operating system to record their events to syslog. Also, in many cases, it is impossible to retrieve out-of-bound information regarding a syslog message.

The Intrusion Detection Message Exchange Format (IDMEF) [26] was proposed as a standard format for IDS alerts. The IDMEF format aims to facilitate IDS alert correlation and information exchange. The IDMEF format is implemented using the Extensible Markup Language (XML). The IDMEF format can be produced by a few systems such as the Prelude IDS [108] and Snort (with an IDMEF plug-in) [79].

The main limitation of IDMEF is the amount of resources required to process (generate and analyse) an IDMEF format. Such a limitation is caused by XML. A document expressed in XML is generally large. In order to retrieve an attribute from an XML document, a program must traverse the XML document tree which in some cases can be a deep tree, and thus requires a lot of resources.

Event Correlation for Forensics (ECF) [18] defines a Canonical Format for events derived from heterogeneous sources. The Canonical Format defines eight attributes: event ID, time, subject, subject type, action, object, object type, and result. In addition to these attributes, the canonical format allows second level (log-specific) information to be incorporated into each log type but stored separately. The Canonical Format has been used to expressed recorded events collected from multiple sources that vary from physical logs (door logs), to operating system logs (the Microsoft Windows and UNIX operating system logs), and application logs (Apache web server web browser logs). The Canonical Format can express only system and application recorded events. The Canonical Format cannot represent network-based recorded events derived from captured network traffic stored
in the PCAP format.

Auto-ECF [1], the successor of the ECF, redefines the Canonical Format. The new canonical event representation defines four required attributes namely event ID, time, event type, and result, but allows for additional other fields also.

In summary, current event representation can represent only host-based recorded events. Hence, there is a need for a canonical event representation that is flexible, extensible, and can represent both host-based and network-based recorded events. The canonical event representation should be generic, in other words, it should represent events in general rather than specific to attack representation.

We now present existing abstract event models.

2.3.2 Event Abstraction

Abstract events enable attack signature writers to develop signatures for detecting attacks regardless of implementation or platform. Signatures using abstract events provides several benefits to IDSs operating in an environment with heterogeneous components. An abstract event model represents a collection of abstract events and concrete events and relationships amongst them.

Abstract events have been used in a few IDSs and computer forensic tools [1, 75]. Nevertheless, the scope of the abstract events in these work is limited. Also relationships between concrete events and abstract events are often hidden or hard-coded in the implementation of an IDS. In this section, existing abstract event models are explored.

ARMD introduces the concept of abstract views in [56]. Abstract views are high-level representation of host-based recorded events. A recorded event can be as low-level as a system call event or a higher level event, such as an operating system event. Abstract views focus only on misuse events rather than general system events, e.g., number of failed logins from particular ports and user behaviour out of office hours. The drawback of abstract views is their inflexibility due to the flat organisation of the model. Abstract views are inflexible because they are generated on an ad hoc basis, and their generation requires some adjustments or in some cases new implementation to fit with monitoring objectives. Also, the flat organisation limits the scope of the abstraction to either low-level or very high-level. In particular, there is only one level of abstraction (abstract view).

CARDS [75] introduces a hierarchical model for attack specification rather than for event abstraction. The model comprises system views, signatures, and
view definitions. The system view is the same as the abstract view in ARMD. A system view is derived from either a recorded event or multiple view definitions. Signatures define event patterns based on event abstractions provided by system views. A view definition provides an additional layer of abstraction which is derived from a signature that has been triggered. For example, TCPDOSAttacks (system view), shown in Figure 2.5, is derived from multiple view definitions: View Def. 1, View Def. 2, View Def. 3, and View Def. 5. View Def. 1, 2, and 3 are derived from three signatures which represent DOS attacks: Ping of Death attack, Land attack, and Teardrop attack, respectively. These signatures specify constraints on IPPacket (system view). View Def. 5 is derived from SYN flooding signature. The SYN flooding signature specify constraints on TCPPacket (system view). The TCPPacket is derived from View Def. 4 where View Def. 4 is derived from the TCPPacket signature. The TCPPacket signature specifies constraints on IPPacket. The main drawback of the event abstraction in CARDS is its complexity. Multiple levels of derivations (a system view → one or more view definitions→a signature→a system view→…, where → represents “derived from”) are complex. Also as shown in Figure 2.5, the term signature referred to both attacks (e.g., SYN flooding) and non attack events (TCP packet) which is counter intuitive.

Auto-ECF introduces an abstract event model. The model defines the one-to-many relationships between an abstract event and different types of recorded events. The advantage of the abstract event model in Auto-ECF is the ability to define composite events or Logical Event Patterns (LEPs). For instance, a Login Session is an LEP which is composed of a Login event followed by a Logout event. The Login event is an abstract event which can be either a SSH Login or a Win2K Login. The Logout event is an abstract event of SSH Logout and Win2K Logout. A major limitation of the abstract event model in Auto-ECF is the inability to represent events derived from captured network traffic.

Martignoni et. al. [64] have developed a hierarchical structure of events based on system calls. The structure is used to model malware behaviour. Nodes in the higher level are generated from aggregating or correlating nodes from the lower level. The limitation of this work is in the scope of the structure. The hierarchy is limited to the system calls produced by Qemu (a virtual machine) only. Although it provides an abstraction at the higher level, to define abstract events it requires an intimate understanding of low-level system calls and the abstract events are still quite low-level. For example, sync_tcp_client is an abstraction of tcp_sock,
Figure 2.5: TCP denial of service attack model in CARDS (from [75]).

bind, and connect system calls.

In summary, there is a need for a comprehensive abstract event model for specifying events in a heterogeneous environment. Existing abstract event models have been implemented as part of an IDS. Hence, such models are inflexible and are designed specifically for modelling attacks. Also, current abstract event models employ a relatively flat organisation, i.e., only one to two levels of abstraction. Such a flat organisation has a limited usage and thus does not provide the full potential of abstraction.

We now investigate time uncertainty in a heterogeneous environment.

### 2.4 Time Uncertainty

A timestamp is a value which indicates that an event occurred at a certain time. Typically, a timestamp is derived from an embedded digital clock on a computer. Timestamps are often the only means for correlating events from multiple sources as timestamps may be the only common attribute amongst different log formats [91]. Correct and reliable timestamps are crucial for multi-step attack detection
as timestamps are the key attributes used to determine the chronological order of attack steps.

Ideally, computer clocks should be synchronised to some common reference time such as the Coordinated Universal Time (UTC) [106] using a clock synchronisation protocol such as the Network Time Protocol (NTP) [69]. In reality, however, clocks are not always synchronised, even in a local network [91]. Thus, timestamps are often unreliable. In addition, in some cases, log entries derived from multiple sources may suffer from event lag where timestamps of the same event in different types of log are different [91]. For example, a log entry derived from a network sniffer and a log entry derived from a web server may contain different timestamps for the same HTTP traffic.

Time uncertainty has not been addressed well in IDS research. In fact, in multi-step attack detection research such as [4, 30, 75, 107, 113], clocks are assumed to be well synchronised, and thus time uncertainty has not been taken into account. Time uncertainty problems have been dealt with to some extent in digital forensics [16, 92].

Section 2.4.1 explores current clock synchronisation mechanisms. Section 2.4.2 investigates current work in computer forensics which look into clock skew and clock drift.

### 2.4.1 Clock Synchronisation Mechanisms

Computer clocks are well-known for their inaccuracy due to the inherent instability of the crystal oscillators used to implement these clocks [92]. The instability of the crystal oscillators is mainly caused by changes in ambient temperature fluctuations [69]. However, computer clocks can be synchronised periodically to external clocks with more accurate time. One such clock synchronisation method which provides the most accurate time is to directly connect a computer clock to an atomic clock and synchronise periodically. Atomic clocks are the most accurate clocks which tell time based on atomic oscillations\(^1\) [23]. However, this method is expensive and thus only used in situations where high clock precision is required. There are other methods to synchronise computer clocks, with less precision compared to atomic clocks but much cheaper, such as through a Global Positioning System (GPS),

---

\(^1\)Since 1967, the International System of Units has defined the second as “the second is the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom” [17].
shortwave radio frequencies, and computer networks.

The most common method to undertake computer clock synchronisation throughout computer networks is the Network Time Protocol (NTP) [69]. NTP services are provided by an hierarchy of servers connected to the Internet. The lower the server stratum number the more accurate is the clock. NTP applies sophisticated algorithms to calculate time and to compensate for issues such as transmission delay, the unreliability of the transmission medium, and unreachable servers.

### 2.4.2 Clock Skew and Clock Drift

Clock skew is the time difference between a particular clock in some device and a reference clock. Clock drift is the rate at which a clock gains or loses time compared to a reference time source. For example, if a clock on a computer gains 10 seconds per week compared to the clock on another computer then the clock drift would be 10 seconds per week. Clock drift, in some cases, may not be constant.

Buchholz and Tjaden conducted a large-scale, long-term study of clocks on web servers connected to the Internet [16]. The authors measured time from more than 8,000 web servers over a period of six months. Their study showed that the clocks of approximately half the servers being measured gave the same time as the clock on the authors’ machine (synchronised with their university’s time server). Almost one third of all servers had a clock skew which varied from around 10 seconds to more than one year.

Schatz et al. studied clocks on several hosts running the Microsoft Windows and the Linux operating system [92]. In their study, clocks on all hosts were synchronised using the Network Time Protocol (NTP) and the Simple Network Time Protocol (SNTP). Their work showed that even though clock synchronisation mechanisms are in place, a clock on one particular host drifted away from the reference clock (Windows domain controller machine). The cause of the anomalous drift could not be identified.

These two works demonstrate the difficulties of clock synchronisation in practice, both globally (on the Internet) and locally (in a small network like a LAN). There are other works in the computer forensics field such as Boyd and Foster who studied the impact of clock synchronisation in a criminal investigation case [15].

In current research into multi-step attack detection such as CARDS [75], attack detection operates based on an assumption that clocks on all computers being monitored are well synchronised. In fact, most, if not all, IDS research neglect to
take clock skew and clock drift into account and thus no solutions to clock skew and clock drift have been proposed.

We now identify challenges in multi-step specification and detection in a heterogeneous environment.

2.5 Research Challenges

This section identifies research challenges which aim to address limitations of the existing signature-based IDSs that can detect multi-step attacks. Also, limitations of existing canonical event representation and event abstraction will be identified.

2.5.1 Canonical Event Representation

Several canonical event representation have been proposed [2, 13, 18, 26], but none of them have been widely adopted. These representation have been neglected possibly due to their limited scope. These event representation are limited to one domain such as host events only or IDS alerts only. None of them can represent events derived from captured network traffic.

The research challenge in this area is to design and develop a canonical representation of events derived from heterogeneous sources. Such a representation should be flexible and extensible so that new event types can be added to the representation with ease.

2.5.2 Comprehensive Multi-Level Event Abstraction

Current abstract event models have several issues. Firstly, current abstract event models provide abstraction to limited types of events and are designed on an ad hoc basis. Some of the models, such as CARDS, have been designed to explicitly target attack representation, and thus the scope of the models are limited to only attack related events.

Secondly, current abstract event models use flat structures, i.e., only one or two levels of abstraction. An abstract event model should provide multiple levels of abstraction so that an appropriate level of abstraction can be used in signature specification. Auto-ECF defines only two levels of abstract event model, i.e., an abstract event level and a concrete event level. For instance, Login event (abstract event) represents an abstraction of a SSH Login and a Win2K Login. Since there
are only two levels of abstraction, the abstract event model in Auto-ECF cannot represent remote login and local login which represents abstract events of remote and local logins respectively without modifying the model.

Finally, existing abstract event models lack flexibility and scalability. Flexibility allows the event model to be reconfigured without affecting existing definitions of abstract events. Scalability refers to the ability to extend the model to incorporate new events without re-factoring existing events.

The research challenge here is to provide a comprehensive hierarchical abstract event model. Such a model should be capable of modelling event derived from heterogeneous sources. The model should provide abstraction to application, system, and network events. The model should also be flexible and extensible. New types of (both abstract and recorded) events should be able to be added to the model without affecting existing events.

### 2.5.3 Multi-Step Attack Specification and Detection Mechanisms

Current multi-step attack specification and detection mechanisms employed by existing systems are complex. In terms of multi-step attack specification, IDSs employing state-based techniques require signature writers to have an intimate understanding of signature instance initialisation, signature progression, types of states, and types of transitions. Choosing types of states and transitions is difficult in complex signature.

In terms of multi-step attack detection mechanisms, current techniques are difficult to trace. For example, EMERALD detects attacks by asserting facts in a fact base (recorded events) using the predefined rules (signatures). If an event triggers a rule, a new fact is derived. The new fact may be inserted into the fact base (dynamic update) which makes the system hard to predict and debug if an error has occurred.

The research challenge in this area is to investigate and develop an alternative attack specification and detection mechanism which is simpler and provides equal or better multi-step attack detection capabilities.
2.5.4 Treatment of Time Uncertainty

Computer clocks are well known for their unreliability. Such clocks are sensitive to the environmental factors such as temperature and electricity. Although clock synchronisation mechanisms, such as NTP [69] are available but they are often not implemented properly or regularly.

Reliable event timestamps are crucial for multi-step attack detection when recorded events are derived from multiple sources. Timestamps are one of the properties used for determining the chronological orders of events. If such timestamps are not reliable, multi-step attacks may not be detected. Time uncertainty issues have been neglected by most IDSs. For instance, CARDS defined the most comprehensive list of event pair relationships based on timestamps. Also, CARDS proposes an architecture for decentralised IDSs where sensors are installed on multiple hosts. In such an architecture, clock synchronisation is crucial for successful attack detection. However, in CARDS, clocks on all hosts are assumed to be perfectly synchronised [75].

Time uncertainty has been studied in computer forensics area. Schatz et al. [92] investigated clock drift in a small network comprises machines with the Windows operating system and the Linux operating system. The work explored several event correlation mechanisms based on time uncertainty.

The research challenge in this topic is to explore techniques to address clock skew and clock drift issues. Once the techniques have been identified, they will be incorporated into the multi-step attack detection mechanisms.

2.6 Summary

This chapter has investigated past and current research into IDS with emphasis on multi-step attack specification and detection in a heterogeneous environment. The ultimate goal of IDS research is to maximise detection accuracy and completeness. Other requirements in addition to the ultimate goal have been identified. IDS evaluation methodologies have been explored.

Two commonly used multi-step attack detection mechanisms, i.e., state-based and event-based, have been described. Past and current IDSs using the two techniques have been investigated and their drawbacks have been identified. The common drawback of these systems is the complexity inherent in their attack specification languages and attack detection mechanisms.
The chapter has reviewed past efforts to standardise event representation including a proposal for standard IDS alerts. Several limitations of such a representation have been identified. This chapter has also reviewed current abstract event models.

Time uncertainty is an important problem which has been neglected by most IDS research. Literature in computer forensics which try to address the problem has been explored.

Research challenges have been identified. The challenges are related to the event representation, event abstraction, multi-step attack specification and detection, and time uncertainty. These challenges are addressed by the work described in the following chapters.

The next chapter describes the two main contributions of this thesis. The first contribution is the canonical event representation and abstract event model, the so called Abstract Event System Architecture (AESA). The canonical event representation of AESA can represent recorded events derived from heterogeneous sources, including captured network traffic which cannot be represented by current representation reviewed in this chapter. The abstract event model provided by the AESA defines multiple abstraction levels which provide coverage of heterogeneous events, i.e., application, system, and network events. The second contribution described in the next chapter is the scenario detection engine based on unification. The engine provides a simple and intuitive attack specification and detection mechanism.
Chapter 3

Abstract Event Model, and Scenario Specification and Detection

Chapter 2 provided an overview of the intrusion detection systems field and explored key research in the signature-based IDS field. Research challenges in multi-step attack specification and detection in a heterogeneous environment have been identified. This chapter presents an abstract event system architecture for modelling and processing events in such an environment. This chapter also presents a scenario detection engine which builds on the unification algorithm to detect patterns of events that match signature specifications.

The proposed abstract event system architecture aims to address inherent problems with existing approaches to the representation of events derived from heterogeneous sources. These problems were identified in Chapter 2. Broadly, the design goals of our abstract event system architecture are twofold. Firstly, to provide a coherent and uniform representation of events derived from heterogeneous event sources, and secondly, to provide for the abstract representation of these events. To achieve these two goals, the abstract event system architecture presented in this chapter builds upon two concepts derived from the object-oriented design paradigm, object inheritance and abstraction.

The scenario detection engine described in this chapter builds on the unification algorithm. The unification algorithm provides a scenario matching mechanism
based on variable substitution where variables are substituted with their relevant values. In our scenario detection engine, the substituted values are instances of events.

The detail and design of the abstract event system architecture and the scenario detection engine using unification have been proposed and published in:


This chapter is organised as follows. Section 3.1 introduces the chapter with an example of a two-step scenario to demonstrate the need both for event abstraction and for a multi-step attack detection system. The scenario will be used throughout this chapter to demonstrate the applicability of the abstract event system architecture and the scenario detection engine. Section 3.2 discusses the fundamental concepts of the proposed abstract event system architecture and describes its overall system design. Section 3.3 describes the detail of our representation of heterogeneous sensor events and the details of an earlier approach and its shortcomings. Section 3.4 describes the abstract event model. Section 3.5 discusses two techniques to address time uncertainty. Section 3.6 presents the scenario detection engine using an adapted form of the unification algorithm and its application followed by an example which involves events from heterogeneous sources. Section 3.7 summarises the chapter.

### 3.1 Motivating Example: Failed Administrator Login Attempts

We present here an attack scenario that demonstrates the difficulties in representing heterogeneous events and detecting multi-step attacks. We assume a network
comprising four hosts with three different operating systems: *Linux host* and *Log host* use the Linux operating system, *Solaris host* uses the Solaris operating system, and *Windows host* uses the Microsoft Windows operating system. It is possible to log in to all hosts both locally and remotely. Local login can be done through a console, i.e., keyboard and monitor, connected to the host. The *Linux host* and *Solaris host* provide the Secure Shell (SSH) service for remote login. The *Windows host* provides the Windows Remote Desktop service for remote login. Log entries from all hosts are replicated and stored on a host called *Log host*. The *Log host* also runs a network traffic capturing program, i.e., tcpdump. The clocks on all hosts are synchronised to a reference clock.

Let us consider a scenario describing a failed administrator login attempt. The scenario comprises two steps as follows:

1. a user does a successful login, either *locally* or *remotely*, into a machine using an unprivileged account, and

2. from the machine, the user tries to *remotely* login into one of the other three machines (in the scenario network) using a system administrator account but the login fails.

This scenario is difficult to detect because of the variety of the platforms, in particular, the recorded events are stored in three different formats. Also, this scenario involves several login services both locally and remotely. To detect this scenario using a system that has no event abstraction, will require a large number of signatures to detect all possible cases.

All log entries on the *Linux host* are stored in the UNIX syslog format [112]. The *Solaris host* stores its log entries in a different format which extends the UNIX syslog format [101]. Activities on the Windows host are stored in the proprietary Windows event log format [12]. The captured network traffic is stored in the PCAP format [49].

The most commonly used method to monitor such activities is to use a signature-based IDS. However, there are several difficulties involved in both designing an acceptable signature for the scenario and detecting the scenario once the signature is defined. Firstly, there is the lack of a canonical representation of events derived from heterogeneous log sources. In this scenario, there are four different possible log formats: UNIX syslog, Solaris syslog, Windows event log, and PCAP. The formats of UNIX syslog and Solaris syslog are implemented slightly differently
with the Solaris syslog containing additional information regarding the level of a log message. The Windows event log implements a proprietary syntax which is not related to either of the UNIX based log formats. In order to use log entries from any or all of the hosts on the network along with captured network traffic efficiently, all the log formats must be transformed into canonical formats.

Secondly, the signatures must be specialised in a signature based approach. If we consider the signature to detect each attack step separately, Step 1 requires six signature variations, i.e., one signature per login service per operating system. Step 2 requires three signature variations based on the number of the operating systems, in particular one signature for each operating system. Thus, in order to detect all possible cases, 18 signatures must be specified or one signature with 18 alternatives. There may then be a non-trivial increase in the number of signatures as a new service or new host is installed into the network. For instance, consider the situation where a new host with the FreeBSD operating system which allows console login and SSH login, is added to the network (assumes that the syntax of FreeBSD syslog is different to Linux syslog). The number of signatures for this scenario will increase to 32 signatures (eight signatures for Step 1 and four signatures for Step 2). Thus, the large number of signatures can cause difficulties in signature management such as keeping all signatures up-to-date and adding new signatures for every new component.

The following sections discuss a novel abstract event system architecture and a scenario detection engine using unification. The abstract event system architecture addresses the heterogeneity of log syntax and provides abstraction of events. The scenario detection engine builds upon the unification algorithm which provides a systematic mechanism to detect multi-step attacks. The first contribution of this work, the abstract event system architecture, is now discussed.

3.2 The Abstract Event System Architecture

Monitoring activities in an environment which comprises diverse components (e.g., operating systems, software versions, hardware architectures) is a difficult task. The difficulties are mainly caused by the lack of a canonical representation of events derived from heterogeneous sources and lack of an appropriate abstract event model. In this section, a novel system architecture which addresses such difficulties will be discussed. Section 3.2.1 presents the fundamental concepts em-
ployed by the architecture. Section 3.2.2 describes the components of the architecture.

### 3.2.1 Fundamental Concepts

The operation of the Abstract Event System Architecture (AESA) consists of two transformation stages. The first stage transforms raw recorded events derived from heterogeneous sources into canonical sensor events. The second stage transforms sensor events into derived events.

The AESA builds upon inheritance and abstraction principles borrowed from the object oriented design paradigm. The AESA comprises two components: the Data Source Schema (DSS) and the Abstract Event Model (AEM). The DSS deals with the syntax of log entries and captured network packets (henceforth log entries and captured network packets are referred to as recorded events). It transforms recorded events derived from heterogeneous sources to produce canonical sensor events. The DSS employs inheritance in the case when one type of recorded events (child) is an extension of the another (parent) based on the syntax and number of attributes of the recorded events. The DSS includes multiple sensor event definitions where each sensor event definition comprises two components: a list of the attributes of each type of related recorded event and a linkage to a corresponding parent sensor event definition (if applicable).

The AEM models semantics of sensor events. The AEM represents multiple levels of abstract events and derived events appear as leaf nodes. The AEM employs abstraction where an abstract event is defined as a group of derived events that share a set of common attributes and semantics. An instance of a derived event represents an event occurring in the system or network being monitored. Such an instance is derived from one or multiple sensor events. The definitions of derived events and abstract events consist of a list of attributes that represent information available at the level of representation and a linkage to the corresponding parent event. The parent event in the AEM is always an abstract event.

The overview of the operation of the AESA is illustrated in Figure 3.1. As described above, the AESA has two transformation stages. The first stage transforms recorded events derived from heterogeneous sources into corresponding sensor events. Sensor events are canonical formats for the representation of recorded events with diverse syntax. For example, recorded events derived from captured network traffic stored in PCAP format on the one hand and Apache access logs on
the others are transformed into \texttt{ pcap\_traffic\_data} and \texttt{ apache\_combined\_log\_format} (sensor events) respectively. The details of sensor event definitions and the transformation of recorded events into sensor events are discussed in Section 3.3.

The second stage transformation converts sensor events into derived events. An instance of a derived event can be generated from the information of one or more instances of different sensor events. Allowing a derived event to be instantiated by multiple types of sensor events enables an instance of a derived event to represent multiple aspects of an activity. From the example shown in Figure 3.1, \texttt{ http\_exchange} is a derived event that represents an HTTP request regardless of the web server software. Details of \texttt{ http\_exchange} are derived from two types of sensor events: \texttt{ pcap\_traffic\_information} and \texttt{ apache\_common\_log\_format} (the parent of \texttt{ apache\_combined\_log\_format}). Abstract events exist to provide references to groups of derived events. From the same example, \texttt{ network\_protocol\_event} is an abstract event of \texttt{ http\_exchange} to which \texttt{ network\_protocol\_event} provides generic information about a network event (not specific to HTTP protocol events). When a scenario detection engine encounters an abstract event in a signature, the engine automatically infers derived events linked to the abstract event in order to match the signature. The system architecture of the AESA and its components are now described.

### 3.2.2 Components of the Abstract Event System Architecture

The configuration of the AESA is depicted in Figure 3.2. The AESA has six components: a DSS repository, DSS definitions, DSS parsers, an AEM repository, AEM definitions, and AEM parsers. The DSS repository is a database that stores instances of sensor events. The database contains multiple tables where each individual table stores instances of a single sensor event type. There is a specific requirement regarding the characteristic of the database, i.e., the database must be capable of storing inheritance information. The database must provide access to any stored event through direct referencing (accessing through the sensor event type) and indirect referencing (accessing through a parent sensor event type). In the case of indirect referencing, when the database receives a query from AEM parsers during the second transformation stage for a parent sensor event type, the database must return all instances of parent and child sensor events. This feature is common to object-oriented databases such as the DB4O [25] and some
relational databases such as the PostgreSQL database [83]. The DSS definitions are a list of all sensor event definitions. Each sensor event definition specifies a list of attributes which are available in each type of recorded event and a link to the parent sensor event definition (if applicable). The DSS parsers parse recorded events and generate instances of sensor events based on the sensor event definitions.

The AEM repository is a database that stores both derived events and abstract events. The AEM repository must have the same reference capabilities as the DSS repository. In particular, the database must support direct and indirect referencing. When the database receives a query from the scenario detection engine during the scenario detection process for an abstract event, all instances of derived events that are children of the abstract event being queried must be returned. The AEM definitions are a list of the definitions of derived events and abstract events. The derived event definition specifies a list of attributes associated with operations of the system or network being monitored and the linkage to the immediate corresponding abstract event. Each derived event must be associated with exactly one abstract event. The AEM parsers generate instances of derived events using information derived from sensor events. An instance of a derived event may be
derived from a single or multiple instances (that may be of heterogeneous type) of sensor events. Once the instances of derived events have been generated, they are stored in the AEM repository.

![Diagram of Transformation Stages](image)

**Figure 3.2: Components of the AESA.**

Let us explain operating procedures of the AESA using the motivating example. There are two transformation stages: from recorded events to sensor events and from sensor events to derived events. In the first stage, all recorded events from Linux host, Solaris host, Windows host, and Log host are transformed into instances of sensor events by the DSS parsers. Instances of these sensor events are stored in the DSS repository where the schema of corresponding sensor event types have been preloaded. Such schema are derived from the definitions of sensor events.

In the second stage, the AEM parsers generate instances of derived events using sensor events retrieved from the DSS repository. Once generated, these instances are then stored in the AEM repository where the schema of derived events and abstract events have been preloaded.

The linkage to an abstract event specified in the definition of a derived event is used by the the AEM repository (database) to return corresponding derived events when an abstract event is queried for during the scenario detection process. For example, when the authentication_event (abstract event) is queried for during the scenario detection process, all instances of derived events under the authentication_event branch are returned by the database. The structure of the AEM is discussed in detail in Section 3.4.

The outcomes of this configuration are the populated DSS and AEM repository. The AEM repository is later be used for scenario detection. The scenario detection engine that utilises the AEM repository is discussed in Section 3.6.3.
Detailed design of the DSS and the sensor event definitions are now discussed.

### 3.3 Sensor Events

Despite proposals [2, 13] for canonical log formats and standard IDS alert formats [26], they have not been widely adopted. There is a vast diversity in syntax used by different logs. For instance, the format of system logs on the UNIX-based operating system compared to the Windows operating system is completely different. Nonetheless, there is a need for a canonical representation of recorded events in order to enable the specification and deployment of system independent scenarios.

In our work we address this through the concept of sensor events. A sensor event is the canonical representation of information available in an entry of recorded events derived from heterogeneous sources. In other words, the sensor event provides a uniform coherent representation of recorded events regardless of their native syntax.

Two sensor event representations have been developed over the course of this work and will be discussed in this section. The Sensor Event Tree (SET) was developed in the early stages of this research. However, due to a number of weaknesses in the approach, the SET has been redesigned, and is thus now obsolete in the latest version of this work. The new design, namely the Data Source Schema (DSS), addresses the drawbacks in the original approach. The DSS provides a more flexible model which is better suited to the requirements of this work when compared to the SET.

Section 3.3.1 discusses the design of the DSS. Mechanisms to define sensor event definitions will be discussed. Section 3.3.2 discusses the details of the SET and its shortcomings.

#### 3.3.1 Data Source Schema

The Data Source Schema (DSS) includes a collection of the definitions of sensor events. The definition of a sensor event is determined based on the syntax of the sensor source or recorded event with one sensor event definition per recorded event type. The DSS employs the object inheritance concept where a child sensor event is an extension of its parent sensor event. Each definition of a sensor event comprises two parts: a list of attributes and a linkage to a corresponding parent.
sensor event (if applicable). The list of attributes contains information available in a recorded event. For instance, the sensor event definition corresponding to each log record in the generic UNIX syslog format [112] contains five attributes: timestamp, host name, process name, process id, and message body. These attributes are specified in the list of attributes of a sensor event namely unix_syslog. In addition to the attributes available in the format of each type of recorded event, there are two additional attributes common to all sensor events namely log_file_name and log_notes. These two attributes must be entered manually by human operators. The log_file_name represents the file name of recorded event from where a particular sensor event is derived from. The log_notes are textual details that can be entered by system operators. Such details are used to describe the configuration of the host (e.g., clock behaviour compared to other hosts) where the recorded events are retrieved from.

The linkage from a child sensor event to a parent sensor event is used in cases where the format of one recorded event type (child sensor event) is an extension of some other recorded event format (parent sensor event). The definition of the child sensor event has the same list of attributes as the parent sensor event with additional attributes. For example, the Solaris syslog format contains three additional attributes to the generic UNIX syslog format. The Solaris syslog contains the same set of attributes as the UNIX syslog format plus three more attributes: message ID, log facility, and log priority. Thus, the Solaris syslog sensor event is a child of the UNIX syslog sensor event.

Figure 3.3 shows a subset of the DSS. Each box represents the definition of a sensor event. The line connecting two boxes represents inheritance relationships where the box in the lower position, the child sensor event definition, is an extension of the box in the higher position, the parent sensor event definition. The child sensor event inherits attributes from its parent (inherited attributes are not shown in the figure). The unix_syslog node represents the attributes present in UNIX syslog messages. The solaris_syslog node represents attributes which are available in the syslog format of Solaris operating system. Since the Solaris syslog is an extension to the generic UNIX syslog, the solaris_syslog node is presented as child of the unix_syslog node. Another example of parent-child sensor event is the Apache web server log. The apache_common_log_format represents the attributes present in the NCSA’s Common Log Format (CLF) [63]. The apache_combined_log_format is an extension to the CLF where the values of the HTTP referrer and the user
agent information (browser signature) are added to the CLF format. The `windows_event_log` represents information provided by the Windows Event Log format. The `pcap_traffic_data` represents attributes of captured network traffic recorded by tcpdump.

![Diagram of log formats]

Figure 3.3: A subset of the DSS.

An instance of a sensor event represents an entry in the actual log or a network packet. All recorded events must be transformed into sensor events. Consider the example scenario discussed in Section 3.1, recorded events derived from Linux host, Solaris host, and Windows host can be transformed into instances of the `unix_syslog`, `solaris_syslog`, and `windows_event_log` respectively. The captured network traffic stored on the Log host can be transformed into instances of `pcap_traffic_data`. All instances of these sensor events are stored in the DSS repository where they are used by the AEM parsers at the later stage.

**Specifying the Sensor Event Definitions**

The definitions of sensor events are the results of manual log format analysis as there are no canonical log formats. Therefore, our log format analysis involves analysing the details of each log format from several definitions such as standards, documentation of the application, implementation of the application, etc. In some cases, where there is not enough information in the previously mentioned sources,
the analysis may have to be undertaken by study of the actual log entries to identify attributes.

At the current stage, predefined definitions of sensor events are classified broadly into four groups:

- operating system log: UNIX-based operating system logs and Microsoft Windows operating system logs;
- low-level system log (system calls): Linux system call logs;
- application log: Apache and IIS web server and Snort IDS;
- captured network traffic: PCAP captured traffic format.

A diagram showing the DSS with all predefined sensor event definitions is presented in Appendix A.

Prior to the development of the DSS, another structure, the Sensor Event Tree was developed to perform an analogous function in representing recorded events. However, this structure had shortcomings which made it a less than optimal solution. In the following sections, the structure and shortcomings of the now obsolete Sensor Event Tree are discussed.

### 3.3.2 Sensor Event Tree

In the early stages of this work, the sensor event model was represented by a tree structure, the so-called the Sensor Event Tree (SET). The diagram showing the full structure of the SET is presented in Appendix A. Several shortcomings of the SET were identified. The details of the shortcomings are discussed below. Hence, the SET is now obsolete. The SET was replaced by the DSS.

The SET is a collection of sensor event definitions and sensor event type definitions. The classification of sensor event definitions is based on the type of application such as SSH, Windows Remote Desktop, Apache web server, and IIS. However, this structure does not take into account the syntactic similarities between the logs which are an important part of the structure of the DSS. A sensor event type definition in the SET is an abstract representation of a group of sensor events, in other words, it represents the class of application. For instance, `web_server_log` is a sensor event type and is an abstract representation of the `apache_access_log` and `iis_log` event definitions. Also, the sensor event type definitions turned out
to constitute an unnecessary and unwanted complication by overlapping with the purpose of the AEM. After several experiments with different varieties of sensor sources, several shortcomings in the design of the SET were identified. We now present these shortcomings.

To validate the expressiveness of the SET, several experiments were conducted, which revealed the following shortcomings with the design of the SET:

- Several sensor event definitions overlapped with some derived events in the AEM. For instance, the ssh_login of the SET (represents an SSH log entry derived from UNIX syslog) overlapped with the ssh_authentication of the AEM.

- Some of the predefined sensor event types were not orthogonal and too abstract. For instance, the ssh_log and the windows_remote_desktop could be represented at the abstract level with either the network_based event or the remote_user_interface event.

- The predefined sensor event definitions could not represent generic system-wide log entries. Some log entries in generic log entries, such as UNIX syslog and Windows Event Log are discarded. For example, log entries of SSH login events are stored in the UNIX syslog. Using the SET parsers, only SSH login related log entries were extracted while other log entries were discarded.

Due to these shortcomings, which were a direct outcome of the tree structure of the SET, the SET was redesigned. In the revised design (the DSS) the tree structure was replaced by the flat structure where the concept of sensor event types in the SET has been removed. The flat structure addresses the issue regarding the non orthogonality of sensor event types and the definition of sensor event types being too abstract.

In the next section, the detailed design of the AEM and the abstraction provided by the AEM are discussed.

### 3.4 The Abstract Event Model

The Abstract Event Model (AEM) includes a collection of the definitions of derived events and abstract events. These definitions represent a coherent hierarchy of operations that occur in the system or network being monitored. This hierarchical structure in the AEM is created based on the abstraction concept.
Before we describe the details of the derived event and abstract event concepts, some definitions are required. The term ‘derived event’ refers to an instance of an actual event which is derived from instances of sensor events. The term ‘abstract event’ refers to a high-level representation of a group of derived events with common semantics. The definitions of derived events and abstract events each refer to a list of attributes associated with such events and a linkage between such events and their immediate parent events (abstract event). The details of derived events and abstract events are now discussed.

3.4.1 Derived Events

A derived event is a container which represents the attributes of an operation that has occurred in the application, network, or system being monitored. The attribute values of such an operation are derived from one or more instances of sensor event. Allowing a derived event to be extracted from multiple sensor events enables the derived event to represent such operations from multiple perspectives. For instance, the http_exchange is a derived event whose attributes can be extracted from one or more web server-related sensor events (e.g., apache_combined_log_format and iis_log_file_format) or captured network traffic sensor event (e.g., pcap_traffic_data). The web server log provides information about an HTTP request such as the request method, the request URI, the response from the server, client address, etc. The captured network traffic provides network related information such as the source address, the source port number, the destination address, the destination port number, network packet payloads, etc. To generate a derived event from two sensor event types, the generation requires some correlation between the two sensor event types. For instance, there is some overlap in the information between attributes of the two sensor event types, i.e., source address and destination address. Hence, these two attributes can be used to establish a link between two sensor event types. Note that, for now, we assume that the clocks on all sensor event sources are synchronised.

The definition of each derived event comprises three parts as follows:

- the list of attributes: The attributes that are associated with the event type being represented;

- the linkage to the abstract event: The name of the immediate parent abstract event;
• the list of sensor events: The list of sensor events whose information can be used to instantiate the derived event.

Figure 3.4 shows part of the specification of application-based and network-based derived events of the AEM. The snort_ids_alert represents an alert generated by the Snort IDS [90]. The attribute values of the snort_ids_alert are derived from: unix_syslog or snort_alert sensor events. The snort_alert here refers to the native alert outputs generated by the Snort IDS [100]. The tcp_exchange represents generic TCP connections. The attribute values of the tcp_exchange are derived from unix_syslog (Linux iptables) or pcap_traffic_data (tcpdump format). The http_exchange represents platform and software independent of an HTTP-related event. The information in http_exchange is derived from web server related sensor events such as apache_combined_log_format and iis_log_file_format and network related sensor events such as pcap_traffic_data. A diagram showing all predefined derived event definitions is presented in Appendix A.

Note that in some existing work, e.g., CARDS [75], network-based events are modelled follow the OSI reference model (OSI 7 layer model) [46]. For example, TCP events are modelled under IP events. However, in the AEM network-based events represent activities in the network being monitored rather than modelling network protocols. For example, the http_exchange derived events in our model are not located under the tcp_exchange derived events. Also, in practice, if one needs to specify HTTP activities in an attack signature, the specification of such activities should focus on the attributes of the HTTP activities, e.g., request URI and HTTP status code, rather than on TCP properties. Therefore, in our model, each network protocol event has its own branch.

3.4.2 Abstract Events

An abstract event is a high-level representation of a group of events (a group of derived events or a group of abstract events) with common semantics and containing common attributes. An abstract event is the product of applying the abstraction concept to derived events. Consider Figure 3.5 which shows a subset of the AEM. Each node represents the definition of an abstract event. The lines connecting nodes represent the parent-child relationships and the outcome is a single-root tree. With a single-root tree, any event (both derived events and abstract events) in the AEM can be represented with the highest form of abstraction, the event node which is the most generic representation of all events. Details of the
The `event` node is presented below. The `abstract events` immediately below the `event` node are the `operating_system_event` node, the `application_event` node, and the `network_protocol_event` node. The `operating_system_event` branch represents operating system-related events, e.g., authentication activities and operations on operating system processes. The `application_event` branch represents application-related activities, e.g., IDS alerts, events, and errors reported by an application. The `network_protocol_event` branch represents operations of network protocols. The level immediately below the `network_protocol_event` is classified by network protocol, such as `tftp_protocol` and `dhcp_protocol`.

The definition of the `event` node specifies six attributes which are common to all `derived events` and `abstract events` in the AEM, i.e., `event_time`, `host_id`, `subject`, `action`, `object`, and `log_source`.

Figure 3.4: Subset of application-based and network protocol-based `derived events`.
action, object, and log source. The event_time represents the timestamp as shown in the actual recorded events. The host_id refers to the identity of the host from which the recorded events were retrieved. The values of the host identification can be one or more DNS domain names or IP addresses. The event node can represent an abstract expression of an activity in the form of ‘subject action object’ where the subject represents a user or a machine, the action represents activities of the subject, and the object represents a target of operation such as a file or a machine. For instance, in a user login to a system, user identity is the subject, authentication is the action, and the system the user is logging to is the object. The log source attribute is a list of references to sensor events whose attribute values are used to generate derived events.

The definition of an abstract event comprises two parts: the list of attributes associated with the events (derived or abstract) being abstracted and the linkage to the immediate parent abstract event. The list of attributes contains only attributes which are common to the immediate child nodes. For instance, the definition of the operating_system_event shown in Figure 3.5 contains two attributes: process_id and process_name of which these attributes are common to the authentication_event and process_event. Also, the specification of the operating_system_event contains a linkage to the event node which is the abstract event of the operating_system_event.
We now discuss the use of the derived events and abstract events to model the motivating example.

### 3.4.3 Modelling Failed Administrator Login Attempts

Recall the failed administrator login scenario discussed in Section 3.1. The login services of three hosts, Linux host, the Solaris host, and Windows host, are being monitored. All hosts can be logged into locally, (i.e., at the console) or remotely, (i.e., through the SSH service on the Linux host and Solaris host and through the Windows Remote Desktop service on the Windows host).

To model the scenario using the AEM, two stages of event transformation are required. The first stage transforms login log entries collected from the Linux host, the Solaris host, and the Windows host and captured network traffic collected from the Log host into four types of sensor event; unix_syslog, solaris_syslog, windows_event_log, and pcap_traffic_data respectively. These sensor events are stored in the DSS repository. In the second transformation stage, these sensor events are transformed into four types of derived events based on the login services as follows:

- **unix_console_authentication** represents console login events on the UNIX-based operating systems, i.e., the Linux operating system and the Solaris operating system.

- **windows_console_authentication** represents console login events on the Windows operating system.

- **ssh_authentication** represents SSH login events on the UNIX-based operating systems, i.e., the Linux operating system and the Solaris operating system.

- **windows_remote_desktop_authentication** represents login events using the Windows Remote Desktop service on the Windows operating system.

These derived events can be abstracted using three abstract events (two levels of abstraction): local_authentication, remote_authentication, and authentication_event. Part of the AEM corresponding to this scenario is illustrated in Figure 3.6. The four login services on the three platforms are represented by the four derived events at the bottom of the figure. The attributes of unix_console_authentication and ssh_authentication are derived from instances of two types of sensor events: unix_syslog and solaris_syslog. The network related information, e.g., source and
destination IP addresses and ports, of the ssh\_authentication are derived from instances of pcap\_traffic\_data. The attributes of windows\_console\_authentication are derived from instances of windows\_event\_log (sensor events). The attributes of the windows\_remote\_desktop\_authentication are derived from instances of windows\_event\_log or pcap\_traffic\_data or both. The unix\_console\_authentication event and windows\_console\_authentication event have a common semantic, i.e., users authenticating themselves physically at a host, hence, the two events can be represented by an abstract event: local\_authentication. Similarly, the ssh\_authentication event and the windows\_remote\_desktop\_authentication event can be represented by an abstract event: remote\_authentication. Both abstract events, local\_authentication and remote\_authentication, can be abstracted further since both events have a common semantic, i.e., user authentication. Hence, a higher abstract event for local\_authentication and remote\_authentication is authentication\_event (abstract event) which is located at the top of the figure.

![Diagram of the authentication\_event branch of the AEM](image)

Figure 3.6: The authentication\_event branch of the AEM.

Using the abstract events discussed above, Step 1 of the scenario (a user logs into a system) can be specified with the authentication\_event. The authentication\_event enables the signature to represent a login event regardless of services or platforms. In Step 2 of the scenario, the user remotely logs into another system using an administrator account, the login activity being represented by the re-
mote\_authentication event. The remote\_authentication event represents remote login services, i.e., SSH and Windows Remote Desktop, regardless of the platform. By combining the specifications of the two steps, the scenario can be modelled with just one specification. However, without the AEM the detection of this scenario requires 18 signatures (the combination of six signatures to detect the first step and three signatures to detect the second step). The number of signatures will increase if a new platform (with new log syntax) is installed into the network being monitored as discussed in Section 3.1.

The AEM provides canonical formats of events derived from heterogeneous sources and abstract representations of events. The derived events of the AEM represent system and network operations regardless of the syntax of recorded events. The abstract events of the AEM provide platform independent representations of events. Such representations enable the generalisation of signatures. Generic signatures allow an IDS to detect attacks across multiple platforms. An example of such an attack is an SQL injection attack against web applications. These attacks work on any platform since the web applications can be run on multiple platforms using multiple types of web server software. Without abstract events, to detect such an attack requires one signature per platform per web server software. However, using the abstract events, the signature can be generalised so that to detect such an attack would require only one signature.

### 3.4.4 Discussion

The lack of a systematic methodology for the representation and abstraction of heterogeneous events to address the complexity of multi-step attack specification and detection in current IDSs are the two main reasons which drive the development of this work. Although there has been some work in the area of standard log or event representation (as discussed previously), no satisfactory and commonly accepted approach has emerged. The AESA addresses this by providing a canonical format for heterogeneous events. The AESA also provides abstract representations of events through the AEM where abstract event representations provide the foundation to specify platform-independent signatures.

The existing work that is most closely related to the AESA is the Event Correlation for Forensics (ECF) work [1, 18]. While the parsers of the ECF transform recorded events into a Canonical Format, abstraction is not applied to these canonical forms. Instead, scenarios are specified separately in an unstructured XML file.
In contrast, the AESA uses a two-stage transformation to identify both derived events and abstract events which represent multiple perspectives of an event. For example, an HTTP request can be represented from an application point of view (web server) and from a network point of view (captured network traffic).

This section has demonstrated the ability of derived events and abstract events to express a multi-step multi-platform scenario. In the following section, techniques to address clock skew and clock drift are presented.

## 3.5 Time Uncertainty

An environment that comprises multiple systems has a high possibility of having a time uncertainty issue. The time uncertainty issue has been neglected by most IDS research. Existing IDSs that involve multiple sensors such as [4, 30, 107, 113] operate either on an assumption that clocks on all machines being monitored are well synchronised or ignore the possibility of time uncertainty. In the alert correlation framework by Morin et. al. [73], they have drawn a conclusion that clock synchronisation is impossible. In their work, to address such an issue, they specify a constant which represents bounds on the difference between timestamps of two events. If the difference is less than the bound, the events match the condition specified in the alert correlation rules.

Although there are mechanisms, e.g., NTP [69], that allow a computer clock to synchronise with a trusted time source, such mechanisms have not always been implemented properly. Hence, in practice, perfect clock synchronisation is very difficult and the assumption that clocks are well synchronised is often invalid.

Time uncertainty of the timestamp associated with an event can arise from a variety of causes. The most common cause and the one we deal with in this thesis is that of unreliable clocks. Ideally, computer clocks should be synchronised with a reliable time source. However, in practice, computer clocks are often not synchronised or not regularly synchronised and thus do not provide reliable time. Unreliable clocks lead to timestamp uncertainty and correlating with unreliable timestamps is difficult.

In some cases, events derived from multiple sources may in addition suffer from event lag where the timestamps for the same event in different types of log are different. For example, a log entry derived from a network sniffer and a log entry derived from a web server may contain different timestamps for the same piece of
HTTP traffic.

Clock skew is the time difference between the clock on a host compared to a reference clock. If the clock skew is constant then the simplest and most intuitive technique to compensate for such a difference is to apply a value (either positive or negative) equal to the skew. Clock skew problems are addressed by the technique called *Constant Skew Compensation Technique* presented below.

Clock drift is the rate at which a clock gains or loses time compared to a reference time source. In other words, clock drift occurs when clock skew is not constant. In reality, clock drift is more likely to occur than a constant clock skew since computer clocks are sensitive to environmental factors. This section presents a technique, the *Linear Regression Technique*, to address problems caused by clock drift.

Section 3.5.1 identifies mechanisms to determine clock skew. Section 3.5.2 presents the constant skew compensation technique. The constant skew compensation technique addresses problems caused by constant clock skew. Section 3.5.3 presents the linear regression technique. The linear regression technique addresses problems caused by clock drift. Section 3.5.4 compares the two techniques and identifies applicability of each technique. It is noted that constant skew is actually a special case of constant clock drift where the drift rate is zero.

### 3.5.1 Determining Clock Skew

Under the assumption of zero event lag, clock skew can be determined by comparing timestamps of the *same event* collected from different physical hosts. For example, a firewall log entry of a particular HTTP request and a web server log entry that corresponds to the HTTP request may refer to the same event. In this case we would expect the source address from the two log sources to be the same and the destination address of the firewall log entry must be the address of the web server.

The clock skew identification process can be generalised to the following two steps.

1. *Identify a reference clock:* The reference clock should be that of a host with a wide variety of *recorded events* such as firewall or network sniffer.

2. *Identify the same event:* Examine the event records of the two hosts: the host identified in Step1 and another host (with clock skew). Identify a pair
of event records which refer to the same event based on semantics and attributes that are common to the two event record, e.g., source IP address and destination IP address. Clock skew is the difference between the timestamps of the same event derived from the two sources.

At this stage, the clock skew identification process must be performed manually by human operators as there is no standard mechanism which identifies the same event derived from heterogeneous sources. The process requires human expert knowledge and must be operated in an off-line fashion and on a case by case basis. Automation of this process may be the subject of future research.

We now describe our technique to compensate for clock skew.

### 3.5.2 Constant Skew Compensation

The constant skew compensation technique is simple and straightforward. The technique applies a value which is equal to the skew calculated by the procedures described above. The applied value is used to adjust timestamps of recorded events.

Having identified a reference clock and a host with clock skew. Under the assumption of constant clock skew, all that need to be done is to adjust the skewed timestamp by the “optimal” clock skew. This value may be estimated in various ways. This is done by selecting a few days worth of recorded events and finding clock skew values for multiple instances of the “same event” using the procedures described in Section 3.5.1. The “optimal” clock skew value could then be estimated by using the mean or median of the range of clock skew values or it could be estimated empirically by identifying the skew value which minimise false positives. But there is no algorithm is know for this, therefore, we conducted a series of experiments in which we apply incremental values of clock skew.

This technique must be operated in a manual fashion as it relies on the clock skew identification procedures described in Section 3.5.1. Also, the technique requires some knowledge about the data and attacks being analysed, i.e., labelled data, to determine an optimal time offset. The major limitation of this technique is the inability to use the technique in an environment where upper and lower bounds of clock skew is significant, in other words, where there is a significant clock drift.

We now describe the linear regression technique which enables us to model constant clock drift, to identify an estimated clock skew at a single point of time
and to predict clock skew in the future.

### 3.5.3 Clock Drift Modelling with Linear Regression

The linear regression technique models constant clock drift with a linear equation. In such a situation, the linear equation can be used to produce instantaneous clock skew values at any given point in time, rather than using the same constant skew value adjustment for all timestamps. Timestamps are compensated using the outputs from the linear equation.

We use the following least squares fitting equations [70] to model constant clock drift:

\[ y = mx + c \]  
\[ m = \frac{n \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i} \]  
\[ c = \bar{y} - m\bar{x} \]  
\[ \bar{y} = \frac{\sum_{i=1}^{n} y_i}{n} \]  
\[ \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \]

The equation for a straight line is shown in Equation 3.1. When the equation is applied to our case study, \( y \) is the length of clock skew (in seconds), \( x \) is the timestamp of log entries (in seconds since the UNIX epoch), \( m \) is the slope of the line, and \( c \) is the y-intercept. Using the least squares fitting technique, the slope is calculated using Equation 3.2 and the y-intercept is calculated using Equation 3.3. The values of \( \bar{y} \) and \( \bar{x} \) are the average of \( y \) and \( x \) which are calculated using Equation 3.4 and Equation 3.5 respectively.

The clock skew \( (y) \) for a given timestamp \( (x) \) is calculated by substituting slope, timestamp, and y-intercept in the Equation 3.1.
For example, Figure 3.7 shows a graph of clock skew values with almost constant drift. The X axis shows timestamps of recorded events. The Y axis shows clock skew values. Using visual inspection, it can be concluded that clock drift in this graph is more or less constant. Therefore, if a graph of clock skew values is similar to Figure 3.7 the linear regression technique can be applied.

![Graph of clock skew values with almost constant drift.](image)

**Figure 3.7:** Graph showing clock skew values with more or less constant drift.

The linear regression technique described in this section uses a single linear regression model. In case of non-continuous clock drift, multiple linear regression models may be used [70]. An example of such a case occurs when a machine is rebooted as shown in [92]. As an illustrative example, Figure 3.8 demonstrates clock skew values for a clock over a period of time. The X axis represents time period. The Y axis represents clock skew values which are results of the comparison between the clock being modelled and a reference clock. As the graph shows, the clock drifted at constant rate. The two drops in the graph are results of machine rebooting. In this particular example, three linear regression models must be used. The slope and y-intercept of the three models are derived from three lines, i.e., A, B, and C.
The main limitation of the linear regression technique is the difficulty associated with automatically deriving the regression equations because identifying the *same event* must be done manually (as we have discussed above regarding the difficulties to determine the skew).

### 3.5.4 Discussion

Time uncertainty has been largely neglected by IDS research that involves multiple event sources. Typically, IDSs operate based on the assumption that clocks on all sensors are perfectly synchronised [37, 72, 113]. However, in practice, perfect clock synchronisation is very difficult although there has been some work done in the computer forensics domain such as [92] and time uncertainty is still a continuing research area, including in the computer forensic area.

In this chapter, procedures to determine clock skew and clock drift have been presented. Two techniques to address time uncertainty problem (constant skew compensation technique and linear regression technique) are proposed. The constant skew compensation technique is used in the case where the clock skew is more or less constant. If the clock skew is not constant but there is a constant clock drift, the linear regression technique should be used. The linear equation
allows system operators to calculate clock skew values at any given point in time and to predict clock skew values in the future.

The linear regression technique provides clear advantages over the constant clock skew compensation technique. In general, the linear regression technique will give a more accurate instantaneous clock fit by which to adjust a skewed timestamp and thus should lead to fewer false positives. Attempting to fit random or scattered data may lead to contrary and anomalous results, hence the “In general” and “assuming a good fit”.

While automating the identification and application of clock skew and clock drift compensation is difficult, this research has identified a general method by which to do so. In particular, expert domain knowledge with regard to the particular log types involved can identify pairs of corresponding event records across two hosts which relate to the same event. Automating this process is arguably possible.

We next discuss on research into scenario specification and detection using unification.

3.6 Scenario Specification and Detection

This section discusses the unification-based scenario detection engine. There are two important definitions which need to be understood for the following discussions: scenario and scenario specification. A scenario (or attack scenario) refers to a group of events which occurs in the applications, networks, or systems being monitored. The description of a scenario is referred to as a scenario specification or a signature. These two terms are used interchangeably in this section. Scenario specifications are used by the scenario detection engine, presented in this chapter, to detect scenarios.

The use of unification for scenario detection was developed to provide an intuitive mechanism to multi-step attack (scenario) specification and detection. Attack detection techniques in existing systems are complex. The best known system is the State Transition Analysis Technique (STAT) framework [34] which employs a state-based attack detection technique. The technique employed by the STAT framework is complex in terms of signature development and signature detection. The STAT framework provides a signature language called STATL (STAT Language). A STAT signature comprises states and transitions where there are three
types of states and three types of transitions. Transitions have effect on signature instantiation and attack propagation. Thus, the signature writers must have an intimate understanding of the underlying matching mechanisms. A discussion regarding the complexity and limitations of current techniques are presented in Chapter 2.

The unification-based scenario detection engine proposed in this section addresses these complexity issues. An event or a set of events that signifies an attack is detected using variable substitution and logical expression evaluation. The signature language developed as part of the system provides a set of logical operators which are coherent and intuitive. Thus, signatures are much simpler and more intuitive to write compared to the signatures of systems using state-based techniques.

Section 3.6.1 provides the background on unification with a simple example. Section 3.6.2 discusses our adaptation of unification for scenario detection. Section 3.6.3 discusses the architecture of our scenario detection engine. Section 3.6.4 shows the application of our system to specify and detect the failed administrator login scenario. Section 3.6.5 discusses the benefits of the our unification-based scenario detection engine.

### 3.6.1 Unification Background

Unification was introduced by Robinson [89] as a formal mechanism which tries to make two expressions syntactically equivalent by substituting variables with sub-expressions from the other expression. The result returned by unification is either *succeed* or *fail*. If unification succeeds, it also produces a complete set of all variables and their corresponding values which is refer to as a *unifier*. If a variable can be substituted by multiple values, the collective set of the variable and all possible corresponding values are referred to as *unifiers*. Unification is usually the algorithm of choice in logic programming and the artificial intelligence field due to its ability to confirm or refute solutions to problems based on the given set of rules and a fact base.

In unification, an expression comprises two parts: one or more terms or variables and one or more operators. Consider an example of unification between expression $E_1$ against $E_2$ to $E_5$ in Table 3.1. Let $E_i$ be an expression where $i$ is 1 to 5, $\alpha$ and $\sigma$ be operators, $A, B, C$ be terms, and $x$ and $y$ be variables. From the table, unification between $E_1$ and $E_2$ succeeds and returns a unifier \{x → B\}. 
This unifier means $E_1$ and $E_2$ can be made syntactically equivalent if the variable $x$ is substituted with the term $B$. Unification between $E_1$ and $E_3$ succeeds and returns a unifier $\{x \mapsto C, y \mapsto A\}$. Unification between $E_1$ and $E_4$ fails because $\alpha$ and $\sigma$ are two different operators. Unification between $E_1$ and $E_5$ fails due to the fact that all occurrences of a variable must be substituted with the same term. The variable $x$ in $E_5$ is substituted with the term $A$ and the variable $x$ in $E_1$ is substituted with the different term $C$, therefore unification of $E_1$ and $E_5$ fails.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression</th>
<th>Result</th>
<th>Unifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1 : \alpha(A, x)$</td>
<td>$E_2 : \alpha(A, B)$</td>
<td>Succeed</td>
<td>${x \mapsto B}$</td>
</tr>
<tr>
<td>$E_1 : \alpha(A, x)$</td>
<td>$E_3 : \alpha(y, C)$</td>
<td>Succeed</td>
<td>${x \mapsto C, y \mapsto A}$</td>
</tr>
<tr>
<td>$E_1 : \alpha(A, x)$</td>
<td>$E_4 : \sigma(A, B)$</td>
<td>Fail</td>
<td>-</td>
</tr>
<tr>
<td>$E_1 : \alpha(A, x)$</td>
<td>$E_5 : \alpha(x, C)$</td>
<td>Fail</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.1: Examples of unification.

### 3.6.2 Unification in Scenario Detection

The scenario detection engine described in this work builds on the unification algorithm. A signature is a set of expressions which describe the characteristics of a scenario. The AEM repository with instances of derived events is equivalent to a fact base. Hence, when our scenario detection engine using unification is executed, it unifies signatures against the populated AEM repository. The scenario detection using unification in the proposed system is developed based on two rules as follows:

**Rule 1:** Free variables can represent either derived events or abstract events.

**Rule 2:** If a variable in an expression is constrained by a reference to an abstract event, the variable can be instantiated with any instances of derived events which are subclasses of the specified abstract event.

An expression in a signature comprises one or more of terms, variables, and operators. A term represents an event (either a derived event or an abstract event) or a constant (e.g., string, integer, or timestamp). A variable is a storage for derived events of interest. A variable in this work is immutable. In other words, once the variable has been instantiated, the value cannot be modified. Operators are functions which are applied to terms and variables. These operators yield a Boolean
value (either TRUE or FALSE). If a signature contains more than one expression, all expressions are joined with the Boolean AND operator. All expressions must return TRUE in order for a signature to match. The detailed discussion of these operators is given in Chapter 4.

Four categories of operators have been defined as follows:

### Identity Operators

- “==” implements an equality operator. The operator can be used in two ways: `variable == Event(event_name)` or `variable.attribute == constant`. If the term on the right is a derived event or an abstract event, the variable is instantiated with corresponding events using Rule 2. If the term on the right side is a constant (e.g., string and integer), the operator returns TRUE only if the value of the attribute is the same as constant.

- “!=” is the complement of “==”. The operator is applied to variables and terms using the following syntax: `variable.attribute != constant`. The operator returns TRUE if the value of the attribute of the variable and the constant are not equivalent. Note that this operator cannot be used when the right term is an event due to the fact that the complement of an event signifies that the variable must be all other events in the AEM repository. For example, `admin_login != Event(authentication_event)` reads `admin_login` (a variable) is not the event of type `authentication_event`. Thus, as far as we concerned, the complement of an event should not be used in expressions.

### String Operators

- `contains_pattern(variable.attribute, pattern, flags)` is a string pattern matching operator where `pattern` is specified using regular expressions. The operator returns TRUE if the attribute contains the given string pattern. The flags specify the properties of string comparison operations such as ignore case.

- `not_contains_pattern(variable.attribute, pattern, flags)` is a string pattern matching operator. The operator returns TRUE if the value of `variable.attribute` does not contain `pattern`.

- `length_greater_than(variable.attribute, length)` compares the (string) size of the value of `variable.attribute` to `length` where `length` can be either an integer or the
attribute of another variable. If *length* is the attribute of another variable, the length of *variable.attribute* and the length of the other attribute is compared. The operator returns TRUE only if the length of the value of *variable.attribute* is greater than the value of *length*.

- **length_less_than**(variable.attribute, length) compares the (string) size of the *variable.attribute* to *length*. The operator returns TRUE only if the length of the *variable.attribute* is shorter than the specified *length*.

**Set Operators**

- **one_of**(variable.attribute, [item1, item2, ...]) is a set operator which returns TRUE if the value of *variable.attribute* matches any item in the given list for the former operator.

- **not_one_of**(variable.attribute, [item1, item2, ...]) is a set operator which is the complement of *one_of*. The operator returns TRUE if the value of *variable.attribute* does not match any item in the item list.

- **one_of_patterns**(variable.attribute, [pattern1, pattern2, ...], flags) is similar to the *one_of* operator but the items in the list are string patterns written in regular expressions. This operator is a shorthand for *contains_pattern* with complex regular expressions, i.e., the regular expressions using multiple or (‘|’) signs.

**Time Operators (No Timestamp Compensation)**

The timestamp values of the two events (*variable1* and *variable2*) evaluated by time operators have a maximum granularity of one second. The time operator parameters *timeout*, *tmin*, *tmax*, and *time_interval* also have a maximum granularity of one second and have in the format: “D day HH:MM:SS”. *D* represents the number of days (this parameter is optional), *HH:MM:SS* refers to number of hours, minutes, and seconds respectively. For instance, “1 day 02:03:04” means 1 day, 2 hours, 3 minutes, and 4 seconds.

- **before**(variable1, variable2, timeout) is a chronological operator that returns TRUE if the timestamp of *variable1* is less than of *variable2*. The operator accepts an optional timeout value. If the timeout is given, the difference between the timestamps of *variable1* and *variable2* must be less than or
equal to \texttt{timeout}. In other words, \texttt{variable1.timestamp < variable2.timestamp} and 
\texttt{(variable2.timestamp - variable1.timestamp) \leq timeout}.

- \texttt{before\_between(variable1, variable2, tmin, tmax)} is a chronological operator that 
returns \textsc{true} if the timestamp of \texttt{variable1} is less than \texttt{variable2} and the 
difference between the timestamps of \texttt{variable1} and \texttt{variable2} is in the range of \texttt{tmin} and \texttt{tmax}. In other words, \texttt{variable1.timestamp < variable2.timestamp} and 
\texttt{(variable2.timestamp - variable1.timestamp) \geq tmin} and \texttt{(variable2.timestamp - variable1.timestamp) \leq tmax}.

- \texttt{before\_exact(variable1, variable2, time\_interval)} is a chronological operator that 
returns \textsc{true} only if the timestamp of \texttt{variable1} is less than \texttt{variable2} and the 
difference between the timestamps of \texttt{variable1} and \texttt{variable2} equals to \texttt{time\_interval}. In other words, \texttt{variable1.timestamp < variable2.timestamp} and 
\texttt{(variable2.timestamp - variable1.timestamp) = time\_interval}.

- \texttt{after(variable1, variable2, timeout)} is a chronological operator that returns \textsc{true} 
if the timestamp of \texttt{variable1} is greater than the timestamp of \texttt{variable2}. If the 
\texttt{timeout} is given, the difference between \texttt{variable1} and \texttt{variable2} must be less 
than \texttt{timeout}. In other words, \texttt{variable1.timestamp > variable2.timestamp} and 
\texttt{(variable1.timestamp - variable2.timestamp) \leq timeout}.

- \texttt{after\_between(variable1, variable2, tmin, tmax)} is a chronological operator that 
returns \textsc{true} only if the timestamp of \texttt{variable1} is greater than the timestamps of \texttt{variable1} and \texttt{variable2} is in the range of \texttt{tmin} and \texttt{tmax}. In other words, \texttt{variable1.timestamp > variable2.timestamp} and \texttt{(variable1.timestamp - variable2.timestamp) \geq tmin} and 
\texttt{(variable1.timestamp - variable2.timestamp) \leq tmax}.

- \texttt{after\_exact(variable1, variable2, time\_interval)} is a chronological operator that 
returns \textsc{true} only if the timestamp of \texttt{variable1} is greater than the timestamps of \texttt{variable1} and \texttt{variable2} equals to \texttt{time\_interval}. In other words, \texttt{variable1.timestamp > variable2.timestamp} and \texttt{(variable1.timestamp - variable2.timestamp) = time\_interval}.

The \texttt{timeout} parameter in time operators is used to limit the time horizon of event 
pairs to be matched. Without \texttt{timeout}s, the time operators match any events whose 
timestamps satisfy the conditions of the operators no matter how distant in time.
and this may result in unrelated events being matched and a high number of false positive. Nevertheless, time operators without timeouts have also been defined such as before(variable1, variable2). Signature writers must be cautious when using time operators without timeout since they may overload the scenario detection engine.

**Time Operators (Timestamp Compensation)**

The granularity of timestamp values and the parameters timeout, tmin, tmax, and time_interval is one second.

- **before_constant_skew(variable1, variable2, timeout)**, applies the constant skew compensation technique to the timestamps of variable1 and variable2. Clock skew compensation values (either positive or negative) are stored in the AEM repository (see Chapter 4 for more details). The operator returns TRUE only if the adjusted timestamp of variable1 is less than the adjusted timestamp of variable2. Each clock skew compensation value is specific to a host (one value per host). before_exact_constant_skew and before_between_constant_skew are also implemented where the functions of these two operators are the same as before_exact and before_between (described above), except the timestamps of the two parameters are adjusted before being evaluated.

- **after_constant_skew(variable1, variable2, timeout)** is a chronological operator that applies the constant skew compensation techniques. The operator returns TRUE only if the adjusted timestamp of variable1 is greater than the adjusted timestamp of variable2. Two variations have been implemented: after_exact_constant_skew and after_between_constant_skew. The functions of these two operators are the same as after_exact and after_between except that the timestamps of the two parameters are adjusted before being evaluated.

- **before_linear_regression(variable1, variable2, timeout)** applies the linear regression technique to the timestamps of variable1 and variable2. To calculate an adjusted timestamp, the variables in the linear regression \( y = mx + c \) must be substituted. The \( x \) variable in the linear regression equation is substituted by the timestamp and the slope and y-intercept which are retrieved from the AEM repository (these two values are calculated before the attack detection process start). The value of \( y \) is the clock skew value for a given timestamp \( x \). At run time, the timestamp \( x \) is converted to seconds since UNIX epoch and
the calculated skew $y$ is the skew in seconds. The compensated timestamp is the result of $y + x$. The compensated timestamp must be calculated for both \texttt{variable1} and \texttt{variable2}. The operator returns TRUE only if the compensated timestamp of \texttt{variable1} is less than the compensated timestamp of \texttt{variable2}. Note that the slope ($m$) and y-intercept ($c$) values specific to a host (different host have different values). Two variations have been implemented: \texttt{before\_exact\_linear\_regression} and \texttt{before\_between\_linear\_regression}. These two operators have the same functions as \texttt{before\_exact} and \texttt{before\_between} except the timestamps of \texttt{variable1} and \texttt{variable2} are adjusted using the linear regression equation before the pair of timestamps are compared.

- \texttt{after\_linear\_regression(variable1, variable2, timeout)} applies the linear regression technique to timestamps of \texttt{variable1} and \texttt{variable2}. The two timestamps are adjusted before the comparison. The operator returns TRUE only if the adjusted timestamp of \texttt{variable1} is greater than the adjusted timestamp of \texttt{variable2}. Two variations have been implemented: \texttt{after\_exact\_linear\_regression} and \texttt{after\_between\_linear\_regression}.

We now describe the architecture of the proposed scenario detection engine using unification.

### 3.6.3 The Scenario Detection Engine

As mentioned, the work of this thesis addresses problems relating to heterogeneous event correlation and evaluates the results of that work in the context of an off-line IDS. Thus, \textit{recorded events} from all systems or networks being monitored must be stored in a central storage. These recorded events are, then, parsed by the DSS and the AEM parsers and stored in the DSS and AEM repositories as discussed in Section 3.2.2.

The architecture of our IDS is shown in Figure 3.9. The IDS comprises four components: a populated AEM repository, a set of signatures, a scenario detection engine, and an alert reporting module. A signature is a set of logical expressions which describe the characteristics of attack scenarios to be detected. Signatures are used by the scenario detection engine where each signature is translated into an SQL statement based on unification concepts (one SQL statement per signature). A signature is translated into an SQL statement as the AEM repository is implemented using the PostgreSQL database. After several experiments and
several prototypes (see Chapter 4 for detail), the PostgreSQL database has been chosen to implement the AEM repository for performance reasons. A query based on the generated SQL statement is then sent to the AEM repository. The AEM repository responds to the query with zero or more result sets. If zero result set is returned, there is no event (or set of events) that match the signature in the AEM repository. Each result set represents an attack instance and includes several derived events (for a multi-step attack). A set of unifiers consists of pairs of variable and derived event. The set of unifiers is passed on to the alert reporting module which generates alert messages. Note that this work does not focus on alert message generation, the alert reporting module, at this stage, prints out all possible values of unifiers. If unification fails, nothing is returned from the scenario detection engine.

![Diagram](image)

**Figure 3.9:** Architecture of the scenario matching using unification.

Unification concepts are applied to scenario specification and detection. In the early stages of the development of the scenario detection engine, a unification engine was developed. However, the unification engine has a serious efficiency and capacity problems, i.e., the unification engine cannot handle large volumes of data. Therefore, for efficiency and robustness reasons, the PostgreSQL database
is used to perform the unification. Using SQL statement generated based on unification concepts, the PostgreSQL database produce the same result as our unification engine but the database returns results much more quickly even with large volumes of data.

Since the scenario detection using unification is performed by the PostgreSQL, a signature must be transformed into an SQL statement. As described in Section 3.6.2, to trigger a signature, all expressions (constraints) in the signature must be satisfied (return TRUE). Constraints on events are transformed into the SQL’s WHERE clauses in an SQL statement where all WHERE clauses are joined with an SQL’s AND operator.

We now present the detection mechanisms employed by the scenario detection engine using unification and an example of the proposed signature language which describes the failed administrator login scenario.

### 3.6.4 Scenario Specification and Example

Recall the example from Section 3.1 where a system administrator is monitoring a two-step scenario. In Step 1, a user logs into one of the three hosts: Linux host, Solaris host, or Windows host. In Step 2, the same user remotely logs into one of the three hosts using the administrator account. Log entries and captured network traffic from all hosts are stored on the Log host. All log entries are parsed by the DSS. Instances of sensor events and derived events are generated and stored in the DSS and the AEM repositories respectively.

Figure 3.10 shows the pseudo code of the signature of the scenario (see Chapter 4 for full syntax). The first four expressions specify the characteristics of Step 1 in the scenario. The first expression specifies that user_login is a variable. The second expression instantiates user_login with authentication_event. The authentication_event is the abstract event of local_authentication and remote_authentication (refer to Figure 3.6 for the authentication_event branch of the AEM). The third and fourth expressions specify two constraints on the authentication_event, namely the value of the user_credentials attribute must be neither ‘root’ nor ‘administrator’, and the value of the authentication_result attribute must be ‘success’. These four expressions are sufficient for detecting Step 1 of the scenario regardless of the login service or platform.

The next five expressions describe Step 2 of the scenario. The fifth expression specifies that the failed_admin is a variable. The expression in line six instanti-
ates the variable failed_admin with the abstract event, i.e., remote_authentication. The seventh and eighth expressions define two constraints on two attributes, the user_credentials must be either the string ‘root’ or ‘administrator’ and the authentication_result must be the string ‘fail’. The ninth and tenth expressions correlate the two steps. The ninth expression specifies that the origin of the remote_authentication must be the same host as the host in Step 1. The tenth expression specifies that the remote_authentication must occur after the authentication_event which instantiates the user_login in Step 1. Since, the expression does not include a timeout, this expression would match any remote_authentication event corresponding to the constraints and occurring after the authentication_event. However, a timeout can be specified. As discussed in Section 3.6.2, there are several variants of chronological operators where some operators receive a timeout.

```python
user_login = Variable()
user_login == Event(authentication_event)
not_one_of(user_login.user_credentials, ['root', 'administrator'])
contains_pattern(user_login.authentication_result, 'success')
failed_admin = Variable()
failed_admin == Event(remote_authentication)
one_of(failed_admin.user_credentials, ['root', 'administrator'])
contains_pattern(failed_admin.authentication_result, 'fail')
failed_admin.source_address == user_login.host_id
after(failed_admin, user_login)
```

Figure 3.10: Pseudo code for the signature of the failed administrator login scenario.

The signature representing the two-step failed administrator login example has been presented. As shown above, the signature is coherent and intuitive. Signature writers are required to only specify a set of constraints on abstract events or derived events. Our scenario detection engine returns instances of derived events that match all constraints to which such events constitute an attack. Since the signature uses abstract events provided by the AEM, it can be used to detect the scenario regardless of the platform using only one signature whereas in other systems such as the STAT [34] framework and EMERALD [81], signatures must be written specifically for each service on each platform.
3.6.5 Discussion

The unification-based IDS architecture addresses the need for the canonical representation and abstraction of events derived from heterogeneous sources. The system discussed in this chapter uses the canonical event representation and abstract event representation provided by the AESA. The scenario detection engine builds on the unification algorithm and provides an intuitive mechanism to detect multi-step scenarios.

The signature language and scenario detection engine described in this chapter builds on the unification algorithm. A signature comprises a list of logical expressions based on unification concepts. The unification algorithm provides mechanisms to confirm or refute these logical expressions through variable substitution. To match a signature, all logical expressions must be satisfied. Each logical expression represents a constraint on an event, and thus the scenario detection using the unification algorithm can be considered to be ‘event-based’.

Event-based techniques are simpler and more intuitive compared to some existing approaches such as state-based techniques. Signatures in event-based techniques describe the characteristics of attacks while in state-based techniques, signatures express the states of the systems under attack. For example, the STAT framework [34] employs a state-based technique. Signatures in the STAT framework are written in the STAT language (STATL). A signature is expressed as states of the system being monitored and transitions (events) that modify the states. There are three types of transition (consuming, non-consuming, and unwinding) where each type of transition has an impact on way the STAT detection engine (STAT engine) creates an instance of a signature and propagates a signature. The consuming transition causes the STAT engine to create a new instance of a signature and switch to a new state and delete the old instance. The non-consuming transition causes the STAT engine to create a copy of an instance of a signature and propagate the state while the old instance of signature still exist. The unwinding causes the STAT engine to switch to the previous state. With these multiple types of transitions and the way they affect the instance creation, the state-based detection technique can be complex in a multi-step attack detection. Also, expressing an attack based on the state of the system being monitored is counter-intuitive. For the scenario detection using the unification algorithm which is an event-based technique, attacks are expressed as logical expressions. The detection of malicious (attack) events is done based on a list of constraints.
Thus, the event-based technique is much simpler than the state-based technique because there is no need to worry about the types of states and transitions.

The event-based technique employed in this work is similar to rule-based expert system techniques where signatures are compared against a fact base (database of events). However, in the event-based technique, the fact base is constant whereas in the rule-based expert system technique, the fact base is updated dynamically depending on the signatures being triggered.

In order to avoid introducing yet-another-syntax, the signature specification described in this chapter uses the syntax of the Python language. This signature language (the so called *Python-based signature language*) is more user friendly compared to existing languages such as the C++ based language used in the STATL [33]. A signature written in the Python-based signature language is written in terms of signature goals, i.e., specify constraints of event that represent attacks. It comprises a list of logical expressions which describe the characteristics of a signature using terms, variables, and operators. The type of events that signify an attack is specified as a variable. The variable will be instantiated with instances of *derived events* that satisfy all logical expressions to which such *derived events* are attack events. On the contrary, in state-based systems, both the signature goals and detection mechanisms must be specified. For example, a STATL signature must specify the type of states (the signature goals) and type of transitions (how to detect the attack). Choosing the type of states and transitions in a complex scenario is a difficult task.

In summary, the unification-based approach provides the same attack detection capability as the state-based approach. In practice, however, the unification-based approach provides several benefits over the state-based approach. The concept of the unification-based approach is the foundation of our Python-based signature language. Such a concept is based on variable declaration and variable substitution which are simple and intuitive. A signature written in the Python-based signature language comprises of variable declarations and a set of predicates which represent attack characteristics. On the contrary, a signature in a state-based system comprises states (of the system or network being monitored) and transitions. As discussed previously, representing attack signatures with states and transitions is complex and requires an intimate understanding of the state machine concept. Also, state-based attack signatures are typically long and complex. A state-based attack signature must contain two types of statements: statements describing
states (from the safe state to the compromised state) and statements describing transitions or event predicates that modify the states. Hence, attack signatures in the proposed system are compact and easier to develop and read compared to signatures in the state-based approach system. In addition, the signature language and scenario detection engine presented in this chapter support signature composition which, to the best of our knowledge, is a novel feature. Signature composition allows signature writers to reuse existing signatures in a new signature. This feature is described in more detail in Chapter 4.

3.7 Summary

In this chapter, we have presented the canonical event representation and event abstraction framework, two techniques to address time uncertainty, and a unification-based scenario specification and detection technique.

The AESA provides canonical event representation and event abstraction as the foundation for scenario specification.

Time uncertainty problems have been neglected by IDS research. Two techniques to address time uncertainty have been presented, the constant skew compensation and a linear regression technique to model constant clock drift. Both techniques are incorporated into the scenario detection engine.

The unification-based scenario detection engine provides scenario detection through variable substitution where scenario specifications (containing variables) are unified against the AESA’s populated AEM repository. If the unification succeeds, a set of derived events is returned. These derived events represent attack events.

We have demonstrated the use of our overall system to detect a two-step failed administrator login scenario in a multi-platform environment. Using the abstraction provided by the AEM and the scenario detection engine using unification, such a scenario can be detected using only one signature whereas existing systems would require a large number of signatures (one signature per platform per login service).

In the next chapter, we discuss the implementation of our system. The Python-based signature language will be discussed in detail.
Chapter 4

The IDS Prototype

Chapter 3 presented the Abstract Event System Architecture (AESA) for modelling events and a unification-based scenario detection engine for detecting multi-step attacks. Two techniques to address time uncertainty have also been described. The Python-based signature language presented in Chapter 3 provides a coherent attack specification language. In this chapter the prototype of an IDS based on concepts described in Chapter 3 are discussed. The IDS prototype incorporates the AESA, two resolutions to time uncertainty, the Python-based signature language, and the scenario detection engine based on unification.

The components of the IDS prototype are illustrated in Figure 4.1. The IDS prototype is divided into two modules: the AESA module and the scenario detection module. Firstly, the AESA module includes the Data Source Schema (DSS) parsers which transform recorded events into instances of sensor events based on predefined definitions of sensor events and the Abstract Event Model (AEM) parsers which transform instances of sensor events into derived events based on the predefined definitions of derived events.

Secondly, the scenario detection module includes the scenario detection engine, a set of signatures, and the alert reporting module. The scenario detection engine transforms a signature into a database query (an SQL statement). The SQL statement is sent to the AEM repository to which the repository returns a result set (instances of derived events). If the result set is not empty, the scenario detection engine formats unifiers (pairs of variables and their corresponding derived events). These unifiers are used by the alert reporting module to report alerts to human
operators. Otherwise, an attack based on the signature being evaluated does not exist in the AEM repository.

This chapter is organised as follows. Section 4.1 discusses the implementation details of the IDS prototype. Section 4.2 discusses the Python-based signature language utilised by the IDS prototype. Section 4.3 discusses issues that have emerged during the development of the IDS prototype and solutions to these issues. Section 4.4 foreshadows the prototype evaluation methodology to be used in evaluating the prototype. The evaluation and its results are described in detail in Chapter 5. Section 4.5 summarises the chapter.

4.1 Implementation of the IDS Prototype

The IDS prototype was implemented using the Python programming language [86]. The Python language is an object-oriented language which provides extensibility and code re-usability. Also, programs written in Python have the capability
to run on multiple platforms, e.g., the UNIX operating system and the Microsoft Windows operating system. Further, the Python language supports infix operator overloading. Operator overloading is required for the implementation of the scenario specification as discussed in Section 4.2 below. Some details regarding the choice of the programming language are discussed in Section 4.3.

The prototype was implemented as an off-line IDS. Recorded events must be stored in a central location where such events are processed in batch mode. The user interface of the prototype is implemented as a Command Line Interface (CLI).

This section is organised as follows. Section 4.1.1 discusses the implementation of the AESA module. Section 4.1.2 discusses the implementation of the scenario detection module.

4.1.1 Implementation of the Abstract Event System Architecture

The components of the AESA module are shown in Figure 4.1 above. The details of the implementation of each component are as follows.

**DSS definitions** consist of a collection of *sensor event* definitions. DSS definitions comprise both a Python Application Programming Interface (API) and a database schema (henceforth these two components are referred to as the DSS API and the DSS database schema respectively). The DSS API is used by the parsers as templates for the generation of *sensor events*. The DSS database schema is used by the DSS repository to create a set of tables for storing instances of *sensor events*. There is one table for each *sensor event* type. The DSS database schema also specifies sensor event inheritance relationships using the INHERITS feature implemented in the PostgreSQL database [83] which is the database used as the DSS repository. For example, Figure 4.2 shows the database schema for the unix_syslog and solaris_syslog *sensor events* where solaris_syslog inherits all attributes of unix_syslog (see Chapter 3 for a detailed discussion of *sensor event* inheritance).

**DSS parsers** transform *recorded events* into canonical forms, i.e., *sensor events*. The parsers process text-based *recorded events* using pre-defined patterns written as regular expressions. These patterns are hard-coded into the parsers. The Python built-in regular expression library is used to perform the parsing. The syntax of the regular expressions implemented in the Python language is Perl Compatible Regular Expressions (PCRE) syntax. For non-text format *recorded events* such as captured network traffic stored in the PCAP [49], an API that
CREATE TABLE unix_syslog (
    dss_eventid    integer PRIMARY KEY DEFAULT nextval('dss_eid'),
    timestamp      timestamp NOT NULL,
    host_name      text,
    process_name   text,
    process_id     text,
    syslog_message text,
    log_file_name  text,
    log_notes      text);

CREATE TABLE solaris_syslog (
    dss_eventid    integer PRIMARY KEY DEFAULT nextval('dss_eid'),
    message_id     text,
    facility       text,
    priority       text
) INHERITS (unix_syslog);

Figure 4.2: Table inheritance example in the DSS database schema.

provides access to the format has been used. After the recorded events have been parsed, the DSS parsers generate instances of corresponding sensor events using the definitions provided by the DSS API. The DSS parsers incorporate the PyGreSQL library [6] which enables the parsers to connect to PostgreSQL from Python programs. The DSS repository is implemented using PostgreSQL.

The DSS repository was implemented using the PostgreSQL database. The PostgreSQL database implements table inheritance which is required by the AESA. The table inheritance feature allows database schema to express inheritance relationships between sensor events. The PostgreSQL database allows indirect referencing to a database. In particular, when the database receives a query from the AEM parsers during the generation of derived events, the database returns records (instances) of sensor events stored in the table being queried and any table which inherits from the table being queried.

AEM definitions include a collection of derived event definitions and abstract event definitions. The AEM definitions comprise both a Python API, the so called the AEM API, and a database schema, the so called AEM database schema. The AEM API is used by the AEM parsers in template fashion to generate instances of derived events. Also, the AEM API is used by the scenario detection engine when the engine performs scenario detection. The AEM database schema are used by the AEM repository to create tables for derived events and abstract events. The relationships between derived events and corresponding abstract events are specified using the INHERITS feature implemented in the PostgreSQL database.
AEM parsers incorporate the PyGreSQL library which enables the AEM parsers to connect to the DSS and AEM repositories. The AEM parsers read sensor events and parse them using string patterns defined in the PCRE syntax. The outcomes of the AEM parsers are instances of derived events which are stored in the AEM repository.

The AEM repository was implemented using the PostgreSQL database. The table inheritance feature of the PostgreSQL database is used to express the relationships between abstract events and derived events. By using the table inheritance, when the database receives a query for any abstract event type, the database returns instances of derived events that are associated with the queried abstract event type. The AEM repository is used by the scenario detection engine during the scenario detection process.

We now describe the implementation of the scenario detection engine using unification.

4.1.2 Implementation of the Scenario Detection Module

The components of the scenario detection module are shown in Figure 4.1. The details of the implementation of the scenario detection engine and the alert reporting module are as follows.

Scenario specifications (or signatures) are written in the Python-based signature language. This avoids introducing new syntax and minimises the learning period for signature writers, the language uses the syntax of the Python language. A signature is, in fact, a Python class which describes characteristics of an attack or a scenario. Each signature comprises a list of logical expressions. Details of the scenario specification language are given in Section 4.2.

The scenario detection engine includes a query generator which translates a signature into an SQL statement. The details of signature transformation are described in Section 4.2.2. After a signature has been transformed into an SQL statement, a database query based on the SQL statement is sent to the AEM repository. The repository responds with zero or more result set. In the former case, there are no events in the repository that match the signature and no attack is detected. Result sets are interpreted using the following rules:

1. One query can lead to several \((n)\) result sets: \(n\) attacks where \(n \geq 0\).

2. Each attack or result set includes several derived events.
3. The scenario detection engine returns to the alert module as follows:

   (a) \( n \) sets of unifiers (each set representing one attack).
   
   (b) Each set of unifiers consists of pairs of variable and \emph{derived event}.

The sets of unifiers are sent to the alert reporting module for alert generation.

The \textbf{alert reporting module} receives sets of unifiers from the scenario detection engine and alerts human operators. The alert messages are reported in the form of variable and \emph{derived event} pairs where all attribute values of the \emph{derived event} are displayed on a program console. Reporting alerts using a standard format such as the IDMEF, is not in the scope of this work.

We now describe the details of the scenario specification language.

\section*{4.2 Scenario Specification Language}

The syntax of the scenario specification language is the syntax of the Python language. In fact, a signature written in the scenario specification language is a Python class. Each signature is a \emph{list of logical expressions} which describe the characteristics of either a single-step or multi-step attack. A signature is matched only if all logical expressions return \texttt{TRUE}.

A pre-processor or syntax-oriented editor may be developed to make signature writing easier. Such a pre-processor would enable signature writers to write signatures without needing to be familiar with the Python syntax. However, this is not in the scope of this thesis. The development of these two components have been identified in the future work.

The syntax of the signature language is described in Section 4.2.1. When a signature is read into the scenario detection engine, the signature is translated into an SQL statement based on unification concepts. Section 4.2.2 describes the process of translating a signature into an SQL statement. Section 4.2.3 describes the signature composition feature of the signature language. Section 4.2.4 discusses the benefits of the Python-based signature language compared to other work.

\subsection*{4.2.1 Language Syntax}

Signatures written in our language must comply with the Python language syntax. The rules for constructing expressions and signatures are now described.
Expression Construction

An expression in our signature language is a logical expression which must return a Boolean result (True or False). Each expression comprises three types of components: variables, operators, and terms (events or constants). Expressions are written using the following syntax:

- Referencing a variable: Any reference to a variable must be written in the form `self.variable_name`. The `self` is specific to the Python language for specifying a class member. The keyword `self` in Python refers to the current object where the variable is defined. `variable_name` is the name of the variable.

- Defining a variable: Before a variable can be used in a signature, it has to be defined using the syntax: `self.variable_name = Variable('variable_name', self)`. The term `variable_name` is the name of variable and `Variable` is a Python class.

- Instantiating a variable with an event: A variable is instantiated using the syntax: `self.variable_name == Event(event_name)`. The term `self.variable_name` is a reference to the variable and `event_name` is a reference to the name of a derived event or an abstract event.

- Specifying constraints on the attribute of a variable using identity operators: An is written using in the form: `self.variable_name.attribute_name == constant`. This expression specifies that the value of the attribute must be equal to the constant. The constant can be a string, an integer, or a float number. To specify not equal, replace `==` with `!=`.

- Specifying constraints on the attribute of a variable using other operators: Other operators are implemented as functions which receive parameters of type variable, variable attribute, or constant depending on the definition of operators described in Chapter 3. The statement must be written using the syntax: `operator(parameter_1, ..., parameter_n)` where the `operator` is the name of the operator and the `parameter_1, ..., parameter_n` are the parameters of the operator.

We have defined the syntax for constructing expressions in our signature language. We now define the syntax for constructing signatures in our signature language.
Signature Construction

A signature written in the Python-based signature language is a Python class. Each signature comprises a list of logical expressions using the syntax defined above. Since a signature is a Python class, a signature requires a heading declaration using the Python syntax. The heading declaration consists of:

- The library importing statements: there are two mandatory libraries that must be imported into every signature: the AEM API and the library that defines the Scenario class definition and the logical expression operators. In case of a composite signature (described in Section 4.2.3), additional libraries must be imported using the import statement. The import statements are written in the form “from library_name import *” where the library_name refers to the name of the library, API or the name of other signatures.

- The class declaration: this statement specifies the name of the signature. The class declaration is written in the form “class signature_name(Scenario):” where the signature_name is the name of the signature and the Scenario is the super-class of all signature. All signatures must inherits from the Scenario class.

- The signature body: the body of a signature must be defined under a method called ‘__definition__’. The declaration of the method name is written in the form “def __definition__(self):”.

Using the failed administrator login attempt scenario from Section 3.1 as an example, the scenario comprises two steps:

- Step 1 a user logs into a host
- Step 2 the user remotely logs into another host using administrator credentials.

The signature of this scenario using the syntax described in this Section is shown in Figure 4.3. This signature is translated from the pseudo code shown in Figure 3.10 (see Chapter 3). The first three lines import the AEM library and the signature library and define the signature name as failed_admin_login_def. Line 5 defines a variable called user_login. In line 6, the user_login variable is instantiated with the abstract event authentication_event which refers to all login services on all hosts. Line 7 specifies a constraint on the user_credentials attribute of the
user_login variable. Since Step 1 of the scenario is to monitor non-privileged user logins into a system, the user_credentials must neither be the string ‘root’ nor ‘administrator’. Line 8 specifies that the user login must succeed. Line 9 defines a new variable called failed_admin. The failed_admin is instantiated with the abstract event remote_authentication. Line 11 specifies a constraint on the user_credentials attribute of the failed_admin variable. The constraint specifies that the values of the user_credentials must be an exact match of the string ‘root’ or ‘administrator’. Line 12 specifies that the authentication_result must contain the string ‘fail’. Line 13 correlates the attributes of the two variables. It specifies that the source_address attribute of the failed_admin variable must be the same as the host_id attribute of the user_login variable. In other words, the origin of the remote authentication must be the same host as the host in Step 1. Line 14 specifies the chronological order of the two variables where the failed_admin (the remote_login events) must occur after the user_login (the authentication_event events).

```python
from signature_lib import *
from aem_api import *
class failed_admin_login_def(Scenario):
    def __definition__(self):
        self.user_login = Variable('user_login', self)
        self.user_login == Event(authentication_event)
        not_one_of(self.user_login.user_credentials, ['root', 'administrator'])
        contains_pattern(self.user_login.authentication_result, 'success', 'ignorecase')
        self.failed_admin = Variable('failed_admin', self)
        self.failed_admin == Event(remote_authentication)
        one_of(self.failed_admin.user_credentials, ['root', 'administrator'])
        contains_pattern(self.failed_admin.authentication_result, 'fail', 'ignorecase')
        self.failed_admin.source_address == self.user_login.host_id
        after(self.failed_admin, self.user_login)
```

Figure 4.3: Signature for the failed administrator login scenario.

A signature for a two-step failed administrator login attempt scenario has been shown above. As described in Section 4.1.2, the signature has to be converted into an SQL statement by the scenario detection engine. In the next section, the signature translation process is described. The SQL statement derived from the signature shown in Figure 4.3 will be shown in the next section.
4.2.2 Signature Translation

The scenario detection engine translates a signature into an SQL statement. Figure 4.4 illustrates the logical operation of the scenario detection engine and the signature-to-SQL translation process. When the scenario detection engine is invoked, the query generator reads the list of attack signatures from the configuration file and creates an instance of a signature. The query generator translates logical expressions in the signature into an SQL statement (one SQL statement per signature). The translation rules are described below. A database query based on the generated SQL is sent to the database connector. The database connector, then, forms a database query based on the SQL statement. The database connector then sends the query to the AEM repository. The AEM repository responds to the query with result sets. The database connector interprets the result sets (see Section 4.1.2 for rules) and produces set of unifiers. These set of unifiers are then sent to the alert reporting module.

![Diagram](image)

Figure 4.4: Signature-to-SQL translation process.

Signatures are translated into SQL statements using the following rules:

- Expressions which instantiate variables in the signatures are translated into the SELECT and FROM clauses of the output SQL statement.

- Variable names are specified as table aliases. Event types are specified in the FROM clauses.

- Constraints of attributes of variables are specified in the WHERE clause of the SQL statement.
We now describe the implementation of operators in respect to the generation of the WHERE clause.

**Operators**

Five categories of operators have been defined in Chapter 3. When the query generator encounters an operator in a signature, the query generator generates part of a WHERE clause corresponding to the type of the operator. There is an exception, if an expression is in the form `variable == Event(event_name)` , the SELECT and FROM clauses are generated instead of the WHERE clause. Table 4.1 shows six of pre-defined operators and their corresponding SQL clauses (see Appendix B for the full list of operators and their corresponding SQL clauses). The left column shows signature operators and the right column shows corresponding SELECT, FROM, and WHERE clauses.

<table>
<thead>
<tr>
<th>Operators</th>
<th>SQL Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>variable == Event(event_name)</code></td>
<td><code>SELECT variable_name.aet_eventid AS variable FROM event_name variable</code></td>
</tr>
<tr>
<td><code>variable.attribute == constant</code></td>
<td><code>WHERE (variable.attribute = constant)</code></td>
</tr>
<tr>
<td><code>variable1.attribute1 == variable2.attribute2</code></td>
<td><code>WHERE (variable1.attribute1 = variable2.attribute2)</code></td>
</tr>
<tr>
<td><code>contains_pattern(variable.attribute, pattern, flags)</code></td>
<td><code>WHERE (variable.attribute ~Epattern)</code></td>
</tr>
<tr>
<td><code>one_of(variable.attribute, [item1, item2, ...])</code></td>
<td><code>WHERE ((variable.attribute = item1) OR (variable.attribute = item2) OR...)</code></td>
</tr>
<tr>
<td><code>before(variable1, variable2, timeout)</code></td>
<td><code>WHERE (variable1.event_time &lt; variable2.event_time) AND ((variable2.event_time - variable1.event_time) &lt;= interval 'timeout')</code></td>
</tr>
</tbody>
</table>

Table 4.1: Mapping between signature operators and SQL clauses.

Consider the example of the signature of failed administrator login attempts shown in Figure 4.3. The signature includes two variables namely, `user_login` and `failed_admin`. These two variables are instantiated with two abstract events: `authentication_event` and `remote_authentication` respectively. The SQL statement corresponding to the signature is shown in Figure 4.5. The references to the two
variables are converted into the SELECT and FROM clauses as shown on line 1 and 2. The constraints on the attributes of the two variables are translated into the WHERE clause as shown on lines 3 to 9 of Figure 4.5. For instance, line 4 in the SQL statement (Figure 4.5) is derived from the not_one_of operator specified on line 7 of Figure 4.3. A signature will be triggered only when all of its logical expressions return TRUE. Thus, all constraints in the WHERE clause are joined by the Boolean AND operator.

```
SELECT user_login.aet_eventid AS user_login, failed_admin.aet_eventid AS failed_admin
FROM authentication_event user_login, remote_authentication failed_admin
WHERE
  NOT ((user_login.user_credentials = 'root') OR (user_login.
                                   user_credentials = 'administrator')) AND
  (user_login.authentication_result ^= E'success') AND
  ((failed_admin.user_credentials = 'root') OR (failed_admin.
                                   user_credentials = 'administrator')) AND
  (failed_admin.authentication_result ^= E'fail') AND
  (failed_admin.source_address = user_login.host_id) AND
  (user_login.event_time < failed_admin.event_time)
```

Figure 4.5: Translated SQL statement for the failed administrator login signature.

### 4.2.3 Signature Composition

Signature composition is, to the best of our knowledge, a novel feature in signature-based IDS research. The signature composition feature allows signature writers to create a new signature by incorporating existing signatures and specifying additional constraints.

Since each signature in our system is a Python class, signature composition can be achieved by defining a new signature where sub-signatures are defined as a member of the class. To demonstrate the signature composition feature, the two-step failed administrator login attempt signature shown in Figure 4.3 is separated into two signatures as shown in Figure 4.6. The two signatures are specified in two separate files namely: `user_login.py` and `failed_admin_attempt.py`. The `user_login_def` signature represents Step 1 of the scenario where user credentials must be neither ‘root’ or ‘administrator’. The `failed_admin_attempt_def` signature represents Step 2 of the scenario.
4.2. Scenario Specification Language

```python
# user_login.py
from signature_lib import *
from aem_api import *
class user_login_def(Scenario):
    def __definition__(self):
        self.user_login = Variable('user_login', self)
        self.user_login += Event(authentication_event)
        not_one_of(self.user_login.user_credentials, ['root', 'administrator'])
        contains_pattern(self.user_login.authentication_result, 'success', 'ignorecase')

# failed_admin_attempt.py
from signature_lib import *
from aem_api import *
class failed_admin_attempt_def(Scenario):
    def __definition__(self):
        self.failed_admin = Variable('failed_admin', self)
        self.failed_admin += Event(remote_authentication)
        one_of(self.failed_admin.user_credentials, ['root', 'administrator'])
        contains_pattern(self.failed_admin.authentication_result, 'fail', 'ignorecase')
```

Figure 4.6: Two sub-signatures of the failed administrator login attempt scenario.

A composite signature of the failed administrator login scenario which incorporates the two sub-signatures is shown in Figure 4.7. The expression on line 7 invokes the signature which detects user login events. The signature is assigned to the variable called \texttt{step1}. The expression on line 8 invokes the signature which detects failed administrator login events. The signature is assigned to the variable called \texttt{step2}. The expressions on line 9 and 10 define the correlation of attributes of two events derived from two signatures.

The signature composition feature provides a convenient mechanism to signature writers to write new multi-step signatures. As shown in this section, a multi-step signature can incorporate existing signatures where existing signatures are specified as steps in a multi-step signature. This feature shortens the time required to write signatures.

The signature language described in this chapter has been implemented as a proof-of-concept. The approach to signature expression employed by the language is event-based where signatures specify attributes of events that signify attacks. Such an approach is simple and coherent compared to the state-based approach where a signature must specify types of states and transitions and how an attack
from signature_lib import *
from aem_api import *
from user_login import *
from failed_admin_attempt import *
class failed_admin_login(Scenario):
    def __definition__(self):
        self.step1 = user_login_def('step1')
        self.step2 = failed_admin_attempt_def('step2')
        self.step2.failed_admin.source_address == self.step1.
        user_login.host_id
        after(self.step2.failed_admin, self.step1.user_login)

Figure 4.7: Composite signature of the failed administrator login attempt scenario.

should be detected.

Note that when the query generator reads a composite signature, the generator generates only one SQL statement no matter how many sub-signatures are contained in one composite signature. Only one SQL statement is generated per composite signature because all logical expressions in all sub signatures must be satisfied (returned TRUE). Hence, one large complex SQL statement is generated for a composite signature.

We now discuss the advantages of our Python-based signature language over some well-known signature languages.

### 4.2.4 Discussion

Our signature language is easy to write and more intuitive compared to the other well-known signature languages: State Transition Analysis Technique Language (STATL) [33] and Production-Based Expert System Toolset (P-BEST) [59].

Consider an example of detecting failed user login attempts regardless of operating system or login services. In this section, a signature written in STATL and Python-based signature language will be compared.

A STATL signature for detecting this example is shown in Figure 4.8 (excerpted from [87]). The signature comprises three states namely s0, s1, and final, and three transitions namely trans1, trans2, and inactivity_expiry_in_s1. This signature can detect failed user login attempts but only for the SSH service. To detect failed user login attempts on other services and other operating systems, additional signatures are required.

Figure 4.9 shows two P-BEST rules (from six rules/signatures) for detecting
use syslog;
scenario syslog_failedpwd (int inactivity_timeout)
{
    string user;
    char *tempStr;
    timer tmr;
    global Messages *msg = new Messages();
    ...
    transition trans1 (s0->s1) nonconsuming
    {
        [SYSLOG_EVENT syslogevt[SSH sshevt]]:(sshevt.illegalUser && !
             msg->isPresentInHash (sshevt.user))
        {
            timer_start (tmr, inactivity_timeout, 0); // Start timer
            tempStr = ctime (&((syslogevt.time).tv_sec));
            tempStr[strlen (tempStr) - 1] = 0;
            log ("\n[\%s] : Illegal User:\%s From: \%s\n", tempStr, (sshevt .
              user).c_str (), (sshevt.rhost).c_str ());
            user = sshevt.user;
            msg->putInHash (user, ");
            ...
        }
    }
    transition trans2 (s1->final) consuming
    {
        [SYSLOG_EVENT syslogevt[SSH sshevt]]:(sshevt.failedPWD && msg
             ->isPresentInHash (sshevt.user) && (user == sshevt.user))
        {
            tempStr = ctime (&((syslogevt.time).tv_sec));
            tempStr[strlen (tempStr) - 1] = 0;
            log ("\n[\%s] : Attempt to break in by:\%s From: \%s\n", tempStr,
              (sshevt.user).c_str (), (sshevt.rhost).c_str ());
            ...
        }
    }
    transition inactivity_expiry_in_s1 (s1->final) consuming
    {[ [timer tmr]}
        {}
    }
    initial state s0 {}
    state s1 {}
    state final
    {
        msg->remove (user);
    }
}

Figure 4.8: STATL signature for detecting failed user login attempts (excerpted from [87]).
the example (excerpted from [59]). This particular P-BEST rule set can detect failed login attempts for login, telnet, rlogin, and rshd services. Rule A1 detects an invalid user name event. If such an event has been detected, rule A1 adds a new fact called bad_login to the fact base and increments the counter called current_bl_cntr. Rule A3 checks if the maximum number of bad logins has reached the value \( x \) (presumably a configured threshold). If the number of bad logins has reached the maximum threshold, a new fact called max_bl_reached is created and inserted into the fact base. The event source used by P-BEST rule set is the Solaris BSM. Hence, this rule set can be use only on the Solaris operating system.

Figure 4.10 shows the signature for detecting the scenario written in our Python-based signature language. This signature can detect failed user login attempts on any login services (both local and remote) and on any operating system platforms.

In summary, our Python-based signature language has several benefits over STATL and P-BEST as follows:

- intuitive and easy to develop: as demonstrated, the Python-based signature is shorter and much less complex compared to STATL and P-BEST.

- it specifies what to detect without specifying detection mechanism: A signature written in STATL requires signature writers to specify how attacks progress (types of transitions) which is counter-intuitive. On the contrary, a signature written in the Python-based signature language only requires signature writers to specify the characteristics of attacks (what is the attack like) without worrying about how an attack progresses which is much more intuitive and easier to write.

- it detects attacks regardless of implementation: the Python-based signature utilises the canonical event representation and event abstraction provided by the AESA module which enables a Python-based signature to generalise attack characteristics, hence, allowing a signature to express platform independent attacks.

In the next section, the problems that arose during the development of the IDS prototype will be discussed. Solutions to these problems are described.
rule[A1(*)]:
    [+e:bsm_event~A12]
    ![e.header_event_type == 'AUE_login ||
        e.header_event_type == 'AUE_telnet ||
        e.header_event_type == 'AUE_rlogin ||
        e.header_event_type == 'AUE_rshd
        ]
    ![e.return_result == 'INVALID_USER]
    [+cc: current_bl_cntr]
    [-max_bl_reached]

    =>
    [+bad_login |
        timestamp = e.header_time,
        user_name = "invalid username",
        command = e.header_command,
        ...
    ]
    ![cc| value +=1]
    ![$e:A12]

... rule[A3(*)]:
    [-max_bl_reached]
    [+cc: current_bl_cntr | value == 'x]
    [+ts:image~A3]

    =>
    ![printf("ALERT: Max Bad Logins \n")]
    [+max_bl_reached | value = 1]
    ![$ts:A3]
    ![EXpertReport('EXpertMessagePointerString, 1042,
        "description", 'pTypeString, "MAX LOGIN ALERT",
        "ruleName", 'pTypeString, "A3", "")]

...  

Figure 4.9: Subset of P-BEST rule set (two from six) for detecting failed user login attempts (excerpted from [59]).

from aem_api import *
from signature_lib import *
class user_login_def(Scenario):
    def __definition__(self):
        self.user_login = Variable('user_login', self)
        self.user_login = Event(authentication_event)
        contains_pattern(self.user_login.authentication_result, 'fail'
                        , 'ignorecase')

Figure 4.10: Python-based signature for detecting failed user login attempts.
4.3 Implementation Issues and Solutions

During the development of the IDS prototype, several issues were identified. This section discusses these issues and solutions that have been developed.

In the early stages of this research, the prototype of the AESA module was implemented using the Java language (henceforth called Prototype 1). The Java language was chosen because the language is an object oriented language which suits the representation of the DSS and AEM that employed the object inheritance and abstraction concepts. DSS and AEM parsers and DSS and AEM APIs had been developed. The details of the early prototype were published in [76].

The prototype produced satisfactory results. However, several problems emerged during development of the scenario detection. One major problem was a limitation of the Java language which is its inability to overload infix operators. The operator overloading feature is required by our signature language in order to provide a simple and intuitive syntax to signature writer. The signature language requires the overloading of two infix operators, i.e., “==” and “!=”.

In order to address the limitation, the signature language was developed using a more flexible language, the Python language. In order to reuse the DSS and AEM APIs, the AESA module was reimplemented using Jython [14] (the Python language interpreter implemented in the Java language). The scenario detection engine including the signature language was developed and run in Jython. Jython allows developers to develop programs using the Python syntax. Programs run under Jython have full access to Java APIs and any application developed in Java which includes the DSS and AEM APIs. The prototype using Jython is henceforth referred to as Prototype 2.

By using Jython, the prototype was able to overload the two infix operators (“==” and “!=”). Since, the prototype was run under Jython, the prototype still has access to the DSS and AEM APIs. Nonetheless, issues about efficiency and capacity of the prototype emerged during the experiments. Theses issues and solutions to the issues are now discussed.

The implementations of Prototype 1 and Prototype 2 both use DB4O [25] as DSS and AEM repositories. DB4O is an object-oriented database which enables developers to use DB4O as a persistent storage for objects. DB4O provides Java APIs which allow developers to store native Java objects into the database. Hence, using the provided APIs helps shorten the development of an interface between the two prototypes and the database.
Several experiments, including the experiments described in Chapter 5, have been conducted. Efficiency and capacity issues of both prototypes have been identified. During one of the experiments (the SOTM34 dataset discussed in Chapter 5) which involves a large set of recorded events (more than 260,000 recorded events), both prototypes failed to operate. Our investigation identified the causes of the issues which are the implementation of Jython and the limitations of DB4O.

These issues are addressed by using replacing Jython with CPython (the original Python interpreter written in the C language) and replacing DB4O with PostgreSQL.

Nevertheless, there were still efficiency issues. In the early stages of the development of the final prototype, a unification engine was implemented as part of the scenario detection engine. The unification engine queried the AEM repository for abstract and derived events. When the unification engine received instances of derived events from the AEM repository, the engine iterated through each instance of derived events and unified the instance against the logical expressions specified in the signature. The unification engine produced accurate results, i.e., identified attacks based on the signatures correctly. However, the unification engine could not handle large volumes of data. Therefore, for performance reasons, in the final implementation, the unification engine was replaced by the database due to the fact that the database performs the unification (the evaluation between an instance of a derived event and a list of logical expressions) much more efficiently than the prototype unification engine. Hence, in the final implementation, signatures are transformed into SQL statements where the generated SQL statements comply with the unification rules defined in Chapter 3. The evaluation results of the final prototype are discussed in Chapter 5.

### 4.4 Evaluation Methodologies

The most common methodology for evaluating an IDS is to operate the IDS against one or more labelled datasets and monitor the performance of the IDS based on a set of measures. A labelled dataset refers to a set of recorded events where legitimate events are clearly distinguished (labelled) from attack events.

Traditionally, IDS performance is identified by two measures: accuracy and completeness. The accuracy of an IDS refers to the ability to identify attacks with low false alarm levels (legitimate events identified as attacks). Such false
alarms are known as *False Positives (FPs)*. The lower the FP level the better
the performance of the IDS. The completeness of an IDS refers to the ability to
identify as many attacks as possible. The completeness is identified by the True
Positives (TPs). The higher the TP level the better the performance of the IDS.

Despite this tradition, the performance of an IDS identified by these measures is
subjective and indefinite. Since an IDS evaluation process operates the IDS against
a labelled dataset and monitors the FP and TP rate, these rates are relative to
the dataset. Hence, running the very same IDS against a different dataset or in
another environment, the FP and TP rate can be and often is completely different.
A detailed discussion regarding these two measures appears in Chapter 5.

Nonetheless, to follow the tradition, the accuracy and completeness of our IDS
prototype will be measured in these terms. In addition, the IDS prototype will be
evaluated as follows:

- The validity of event representation provided by the AESA. This measure
  examines the validity of *sensor events* and *derived events* based on *recorded
  events* in a dataset.

- Multi-step signature specification and detection. Specifying and detecting
  multi-step attacks is one of the goals of this research. This measure verifies
  the success of the outcomes of this research.

- Signature composition ability. This measure evaluates the novel feature pro-
  vided by the signature language described in Section 4.2.3.

- Ability to deal with time uncertainty. In an environment which comprises
  heterogeneous sensors, it is inevitable that there will exist time uncertainty.
  Time uncertainty issues have been largely neglected by IDS research. Dealing
  with such issues will provide benefits to IDS research as well as computer
  forensic research. This measure evaluates the ability of the IDS prototype
to detect attacks in an environment which suffers from clock skew and clock
drift.

Detailed discussion of IDS evaluation methodologies and of evaluation results of
the IDS prototype appears in Chapter 5.
4.5 Summary

This chapter describes the implementation details of the IDS prototype. Due to several issues, especially capacity issues, that emerged during the course of this research, three prototypes have been developed. The final prototype is the most robust of the three prototypes. The IDS prototype is divided into two modules: the AESA module and the scenario detection module. The AESA module includes a two-stage transformation of events: from recorded events to sensor events and from sensor events to derived events. The DSS and AEM repositories are implemented using the PostgreSQL database.

The scenario detection module includes a scenario detection engine which determines the success of the detection based on the result set returned by the AEM repository. The scenario detection engine translates a signature into an SQL statement (one SQL statement per signature) where the SQL statement is generated based on unification concepts. The SQL statement is sent to the AEM repository where the AEM repository responds with a result set. An attack is detected if the result set is not empty.

A Python-based signature language has been developed. A signature is a list of logical expressions where the signature alert will be triggered only if evaluation results of all expressions return TRUE. The advantages of the Python-based signature language over other signature languages have been identified. One of the outstanding advantages of the Python-based signature language is the ability to specify a signature that can detect attacks regardless of the implementation.

The Python-signature language also provides the signature composition feature which to the best of our knowledge is a novel feature. The signature composition feature allows signature writers to reuse existing signatures in a new signature where additional constraints can be specified.

IDS prototype evaluation methodologies have been foreshadowed. In addition to the accuracy and completeness of the IDS prototype, four other measures have been identified. Detailed discussion of IDS evaluation methodologies and of the evaluation results of the IDS prototype are discussed in Chapter 5.
Chapter 5

Evaluation

Chapter 4 described an IDS prototype based on the Abstract Event System Architecture (AESA) for event representation and abstraction and the scenario detection engine using unification. This chapter explores current IDS evaluation methodologies, it describes how the *IDS prototype* has been evaluated and presents an analysis of the evaluation results.

The most common IDS evaluation methodology \([10, 27, 61, 62, 68, 84]\) is to monitor the outputs of an IDS (alarms) when it is run against a dataset. The dataset refers to a collection of application and system events or captured network traffic (henceforth application and system events and captured network traffic are referred to as *recorded events*). The IDS prototype proposed in this thesis has been evaluated against two datasets: a dataset from the Scan of the Month (SOTM) project \([103]\) and a synthetic dataset. The analysis of the evaluation results will be presented.

Typically, IDS evaluation focuses on accuracy and completeness. However, we argue these two measures may by themselves be inconclusive and misleading. These measures are indicators of the performance of an IDS operating against a *particular dataset*. If the same IDS is tested against another dataset, the results will be different either better or worse. In this chapter, other characteristics will also be explored.

The remainder of the chapter is structured as follows. Section 5.1 explores past and present IDS evaluation methodologies, dataset classifications, and evaluation criteria. Section 5.2 investigates issues in and misconceptions about current IDS
evaluation methodologies. Issues and misconceptions will be discussed. Section 5.3 presents the evaluation methodology used for evaluating the IDS prototype. The configuration of the IDS prototype for evaluation will be described. Section 5.4 explores the datasets to be used in the evaluation. Section 5.5 discusses the analysis of the evaluation results. Benefits and limitations of the IDS prototype will be identified. Section 5.6 summarises the chapter.

5.1 Existing IDS Evaluation Methodologies and Evaluation Criteria

Traditionally, performance of an IDS is measured by accuracy and completeness. Accuracy refers to the ability to distinguish between legitimate events and attack events. Completeness refers to the ability to identify all instances of attacks. Such measures are often represented as the percentage of True Positive (TP) and percentage of False Positive (FP) respectively. The TP rate is the ratio between the number of attacks detected by an IDS to the number of all instances of attacks. The FP rate is the ratio between the number of false alarms (alarms are raised but there is no attack) to the number of events. The TP and FP rate of an IDS are determined by analysing the results of operating an IDS against a labelled dataset. A labelled dataset refers to the dataset whose legitimate events and malicious events have been clearly identified.

In addition to accuracy and completeness, there are other measures of IDS performance such as deployment issues, maintenance issues, and coverage. These measures will be discussed in this section.

This section investigates past and present IDS evaluation methodologies and evaluation criteria. Section 5.1.1 explores past IDS evaluation methodologies. Section 5.1.2 investigates classifications of datasets. Section 5.1.3 presents current IDS evaluation criteria.

5.1.1 IDS Evaluation Methodologies

The most common and most widely used IDS evaluation comprises three steps: create and label a dataset, run the IDS against the dataset, and analyse alerts generated by the IDS.

One of the first bodies of research into IDS evaluation is the system developed
by Puketza et al. [84, 85] at the University of California at Davis. The authors developed a software platform which provides a set of scripts that allow IDS evaluators to generate both legitimate activities and attacks. Only one signature-based NIDS (network-based IDS), the Network Security Monitor (NSM) [40], was evaluated. The evaluation focused on completeness (all known attacks were detected) and efficiency (load of the processor on the machine running NSM). The evaluation results were published in [85].

The most comprehensive IDS evaluations attempted to date, are the three IDS evaluations conducted by DARPA and MIT Lincoln Laboratories in 1998 [62], 1999 [60], and 2000 [57]. Six IDSs [36, 55, 71, 81, 95, 110] participated the 1998 evaluation. The analysis of the evaluation results is shown in [62].

In addition to the analysis of the evaluation results, the project has published the three datasets used during the evaluation [58]. All three datasets were generated using scripts which generate synthetic host and network background events and attacks. The novelty of these tests was the fact that multiple platforms were incorporated into the tests, i.e., several hosts running different operating systems, in particular, Solaris and Windows NT. Several IDSs were tested in the evaluations. These datasets are well known as the DARPA datasets. The datasets have been widely used amongst IDS researchers.

The DARPA IDS evaluation and the DARPA datasets have been criticised by McHugh [65, 66]. The criticisms are mainly directed at the mechanisms used to evaluate IDS, e.g., some details regarding the evaluation are not disclosed, the datasets are too synthetic, and the used of ROC to indicate IDS performance is misleading and biased towards systems with tunable parameters.

There have been several other evaluations of note such as research work conducted by IBM Zurich [28], France Telecom [29] and commercial evaluations [96, 97, 98] (see [29, 43, 67, 68] for summaries of existing evaluations). These evaluations, however, do not make their datasets publicly available and information about these tests is scarce.

We now investigate classes of datasets.

### 5.1.2 Dataset Classifications

Ideally ones would like to be able to test IDS accuracy in real-time against real event or network traffic data. The real data - if of sufficient volume - would presumably contain evidence of attacks as well as benign activity. One difficulty
with this approach is that the data is by definition unlabelled so that accuracy or success of the results produced by the IDS can be assessed only by a ‘third party’ oracle operating in parallel with the IDS test or operating post hoc on a record of the data.

A second major difficulty is that such a test, given it is real-time, is not reproducible.

For these two reasons, IDS testing has adopted a variety of testing techniques other than real-time testing with on-line event or traffic data. These techniques all necessarily rely to some degree on synthetic event or traffic data.

Such datasets can be classified into five classes (adapted from Mell et al. [68]) as follows.

Datasets with no background events. Datasets in this class contain only attack events. Due to its simplicity, this type of dataset is suitable for validating intrusion detection mechanisms employed by an IDS during the development phase. Such a dataset was used by [96].

Datasets with background events derived from a real environment (on-line real environment) and containing synthetic attacks. A dataset is generated by collecting recorded events from a production environment where attacks have been injected into the environment. The advantage of this type of dataset over the dataset of the no background event class is that the dataset is close to the ‘real world’ (not actually real world because the attacks are synthetic). This class of dataset has a few drawbacks. The main limitation of this approach is the fact that there are too many uncontrollable variables. For instance, there is no mechanism to guarantee that the environment being monitored will contain only synthetic attacks and no other attacks. Thus, it is difficult to label the dataset. There are also issues regarding the confidentiality and privacy of the information in the background events. Thus, datasets in this class cannot be made publicly available. An example of such a dataset is used in [98].

An adaptation of the previous class is the class of datasets with sanitised background events. This class of dataset addresses the confidentiality and privacy issues. Sensitive information, such as user name, password, IP address, is removed from the background events. The advantage of this approach is the dataset still represent real environment. The main drawback of this approach is the possibility that significant information that may be required to identify an attack may be removed during the sanitising process. Also, the sanitation process is error prone. In
particular, errors may be caused by sanitising scripts. For example, the SOTM34 dataset, used for evaluating the IDS prototype in this chapter, suffers from errors caused by sanitising scripts where the scripts corrupt the syntax of some recorded events. The details of these errors are discussed in Section 5.4.

The real dataset is an adaptation of the on-line real environment with synthetic attacks except that attacks are real attacks (not injected). In the real dataset, recorded events are collected from an on-line real environment without any modification. The dataset is analysed and labelled. IDS evaluations with real datasets have been widely adopted in the past few years, such as work presented in [43, 53, 111]. The advantage of the real dataset is the fact that background events and attacks are real and thus the evaluation results reflect the performance of an IDS in real working environment. However, there are some drawbacks in this class. Firstly, there is an issue similar to the dataset in the on-line real environment with synthetic attacks class where there is no guarantee that all attacks have been identified. Labelling such a dataset can be a very difficult task. Secondly, the characteristics of the dataset is not controllable such as types of attacks, number of attacks, etc. For instance, all attacks in the dataset may be of one type, e.g., port scanning, which is not harmful (in most cases). Finally, there are confidentiality and privacy issues involved. Thus, real datasets are not publicly accessible.

Datasets with synthetic background events. In this class, both background events and attacks are artificial. The main advantage of this class of datasets is full control of generated background events and attacks. The datasets in this class, however, may not represent the real world. However, such a class of datasets has been employed by many existing IDS evaluations including the DARPA evaluations, IDS evaluation research at IBM Zurich, and the evaluation platform developed by the University of California at Davis.

We now explore intrusion detection evaluation criteria.

5.1.3 Intrusion Detection Evaluation Criteria

Traditionally, IDS evaluations measure the ability of an IDS to detect attacks while generating few false alarms. However, there are other aspects of IDS evaluation which should be taken into account such as the ability to correlate events, ability to detect attack success, etc. This section investigates these aspects as well as accuracy and completeness.

Intrusion detection approaches are commonly classified as either anomaly-based
or signature-based. The anomaly-based approach detects attacks by identifying abnormal events as malicious. The signature-based approach detect attacks by compare recorded events against the event patterns of known attacks. If such events match the patterns, the events are identified as attacks.

Several work have identified IDS evaluation criteria [10, 24, 28, 29, 39, 38, 58, 61, 62, 68, 82, 85]. In this section, IDS evaluation criteria are divided into two categories: functional and non-functional characteristics.

**Functional Characteristics**

Functional IDS characteristics refer to *accuracy* and *completeness* of an IDS. These characteristics are commonly used as IDS performance indicators. Intrusion detection accuracy refers to the capability to distinguish between legitimate events and (malicious) attack events. If an IDS reports an alert but there is no attack, such an alert is referred to as a False Positive (FP) alert. Completeness refers to the capability to identify all instances of attacks. An alert that is generated when an attack has actually occurred is referred to as a True Positive (TP) alert.

Accuracy and completeness of an IDS are often represented with a Receiver Operating Characteristic (ROC) curve [10]. An ROC curve is plotted as a graph with two axes where the *X* axis represents percentage of FPs (ratio of FPs to to the number of events) and the *Y* axis represents percentage of TPs (ratio of TPs compared to the number of all instances of attacks). Ideally, an IDS ROC curve should be a dot on the top left corner which represents 100% TP and 0% FP. In other words, all instances of attacks are detected while no false alarm has been generated.

Using an ROC curve to represent IDS performance is most directly only applicable to parametric IDSs. A parametric IDS is an IDS that allows the system operator to adjust parameters so that the ratio between detection rates and false positive rates can be tuned [9]. Each point on the ROC curve represents the output for different values of parameter tuning. Hence, the ROC curve of a non-parametric IDS would be represented with a single point rather than a curve because the results would be the same every time.

For illustration purposes, Figure 5.1 shows an example of three ROC curves derived from three ‘toy’ IDSs. *IDS1* and *IDS2* are anomaly-based parametric IDSs and *IDS3* is a signature-based IDS. The detection performance of *IDS1* varies from 40% detection rate while it generates 10% false positives to 100% detection rate
while it generates 100% false positives, i.e., all events are identified as attacks. The detection performance of IDS2 varies from 70% detection rate while it generates 10% false positives to 100% detection rate and 100% false alarm rate. By comparing the two curves, it can be stated that IDS2 detection performance is better than IDS1 when both systems are evaluated with a particular dataset. The results vary if another dataset is used. IDS3 is a signature-based IDS (non-parametric ), hence, the evaluation result of IDS3 is represented as a point rather than a graph. The result of IDS3 evaluation can be interpreted as “a particular set of signatures” of IDS3 detects 90% of attacks while it generates 10% false positives.

![Example of three ROC Curves](image)

Figure 5.1: Example of three ROC Curves

On a related issue to completeness, there are is also one characteristic which should be taken into account. In particular, the ability to detect novel attacks. Ideally, an IDS should be able to detect a never-before-seen attack. Such a characteristic is commonly found in anomaly-based IDSs. The inability to detect novel attacks is one of the main limitations of signature-based IDSs.
Non-functional Characteristics

Non-functional IDS characteristics refer to other performance aspects of an IDS. These characteristics have not been discussed much in literature. The reason for these characteristics being neglected is possibly because there is still no mature evaluation methodology for IDS that is comprehensive, some of these features have not been carefully examined in a systematic context. Examples of non-functional characteristics are:

- Ability to correlate events: Ability to correlate events refers to a feature of an IDS which enable it to recognise relationships between events and to recognise a set of events as possibly one compound event. This characteristic also include the ability to correlate events derived from heterogeneous sources since IDSs are often operated in environments with diverse event sources.

- Deployment and maintenance issues: The deployment issues refer to difficulties that may occur when an IDS is deployed in a real environment. For example, some host-based IDSs may need access to low-level or kernel-level information which may not be available due to security reasons. The maintenance issues refer to the problems that may occur after an IDS has been deployed. For instance, an anomaly-based IDS may be very difficult to maintain due to the fact that it requires regular training where the training data must be free of attacks.

- Responsiveness, capacity, and resource consumption: The responsiveness refers to how quickly an IDS can identify an attack under normal circumstance and high load. The capacity refers to the amount of recorded events or the network throughput that an IDS can handle effectively. The resource consumption refers to the hardware resources (CPU, memory, and disk space) an IDS requires to operate. Ideally, an IDS should have a small footprint where the intrusion detection process would not slow down the host the IDS is running on.

We now discuss limitations of current IDS evaluation methodologies.
5.2 Remarks on Existing Evaluation Methodologies

Traditionally, IDS performance is identified by accuracy and completeness. However, we argue that these measures by themselves are inconclusive and misleading. These measures are specific to an environment or a dataset used to evaluate an IDS. Hence, an IDS will generate different results in a different environment or dataset.

Some issues in existing IDS evaluation methodologies have been previously identified. The most notable work is the critique on the DARPA IDS evaluation and DARPA datasets by McHugh [65, 66]. Several issues have been identified such as background activity generation, distribution of attacks, order of attacks are always the same, etc. While McHugh specifically dealt with the DARPA IDS evaluations, his criticisms can be generalised and applied to other evaluations as well.

This section investigates several limitations in existing IDS evaluation methodologies. Section 5.2.1 discusses issues related to ROC curve plotting and analysis. Section 5.2.2 explores issues in interpreting detection rates and false positive rates. Section 5.2.3 discusses issues in dataset generation.

5.2.1 Receiver Operating Characteristic Curve Issues

The main purpose of an IDS is to identify all possible malicious (attack) events correctly and to avoid reporting FPs. Hence, it seems sensible to measure IDS performance based on TP rates and false alarm rates. Traditionally, IDS evaluations focus on comparing performance of multiple IDSs using ROC curves. The IDS evaluation process comprises operating an IDS against a dataset, that has been thoroughly and carefully analysed, and collecting the alarms generated by the IDS. The number of alarms are represented as a point on a graph.

There are a few variations of definitions and scales of X and Y axes of an ROC graph (refer to Figure 5.1 for an example of a graph). For example in [10, 65, 66] the X axis represents the probability of an alarm being triggered given only background events (noise) and the Y axis represents the probability of an attack being detected given the whole dataset (attacks plus background events). In [99], the X axis represents the percentage of false alarms to the number of background events and the Y axis represents the percentage of detected attacks
to the number of all attacks.

Ideally, the optimum operating point for an IDS is the closest point to 100% detection and 0% false positives (the (0, 100) coordinate). However, in order to interpret the meaning of an ROC curve, several other factors must be taken into account such as cost of a false positive, the value of a TP, the probability of attacks and normal events [68]. Much of the literature overlooks these factors and does not discuss these factors when an ROC curve is presented.

For a non-parametric IDS, the ROC is represented with a single point.

In addition to the issues with ROC plotting there are also issues with ROC curve comparison and analysis. As discussed above, the main purpose of plotting ROC curves is to compare intrusion detection performance of IDSs. There are several approaches to interpret the meaning of ROC curves. For instance, Durst et al. [32] suggested that the greater the area under a curve the better the IDS detection performance is. However, Gu et al. [38] argued that it is not appropriate to use the area under an ROC curve because the area represents all operation points of an IDS. They suggest that the comparison should be based on the optimum operating point. There are also other approaches such as the work by Stolfo et al. where they incorporate cost-based analysis into ROC curves [99] and the work by Gaffney and Ulvila which incorporated the assessments of the effect of an attack to the environment with [35].

McHugh stated that using ROC curve to measure IDS performance is inappropriate because an ROC curve does not provide detailed information on why an IDS produces such a ROC curve [65]. Much of the published literature that uses an ROC curve to measure IDS performance does not provide parameter settings (types and values) that were used to plot the ROC curve. The lack of the parameter settings causes difficulties to determine factors that affect the shape of an ROC curve.

### 5.2.2 Detection Rates and False Positive Rated Issues

In relation to the above ROC curve issues, detection rates and false positive rates are relative to a dataset. Detection rates and positive rates are the results of running an IDS being evaluated against a labelled dataset. The results are, then, used to plot a ROC curve. Hence, detection rates and false positives rates are by themselves inconclusive general indicators for IDS performance.

In a recent publication by Ingham and Inoue [43], they have evaluated three
HTTP specific anomaly-based IDSs (each system provides multiple intrusion detection algorithms) [44, 45, 53, 111]. The evaluation results showed high false positive rates which conflicted with the results published in the original publication of the work [53, 111]. It is notable that although all IDSs in the evaluation employ anomaly-based intrusion detection approach, some of them failed to detect novel attacks, e.g., failed to detect variants of a worm. There are several factors which may contribute to the contradictory evaluation results. Firstly, different datasets were used. Hence, if a different dataset is used, different results will be produced. Secondly, since the implementation details of the IDSs being tested are not available publicly, Ingham and Inoue had to implement the two systems, i.e., [53, 111] based on the original publications. Therefore, the poor results may be caused by the implementation or as much as caused by the algorithms. Thirdly, the systems may not be deployed properly such as the training data may not be properly sanitised.

All in all, IDS performance cannot be solely identified by detection rates and false positive rates because the performance determined from those rates is strictly relative to a particular dataset. In other words, an IDS performing well (high detection rates and low false positive rates) on a dataset may perform poorly on a different dataset or in a real environment.

**False Positives Issues in Signature-based IDS**

Traditionally, false positives refer to false alarms generated by an IDS which identifies legitimate events as attacks. This definition is generalised to any intrusion detection approach. But we argue that this definition should be applied only to anomaly-based approaches and that they reflect the fact that not all abnormal events are attacks [67].

In signature-based intrusion detection approaches, we argue that the term ‘false positive’ is a misnomer as the so called ‘false positive’ is not caused by signature matching mechanism but rather by the following issues:

- **Bad data**: False positives may be caused by problems in *recorded events* such as unreliable timestamps, insufficient or incomplete *recorded events*, and the integrity of *recorded events* having been compromised.

- **Bad signature**: A signature should describe unique characteristics of an attack. If such a signature fails to do so, an imprecise signature is the cause
of a ‘false positive’. In addition, complexity of signature specification may contribute to ‘false positives’.

Partial solutions to issues regarding unreliable timestamps are the constant skew compensation and the linear regression technique described in Chapter 3. The application of the two techniques to a real dataset is presented in Section 5.5.6 and Section 5.5.7.

5.2.3 Dataset Generation Issues

Dataset generation is difficult and time consuming. As discussed in Section 5.1.2, datasets can be classified into five classes: no background events, on-line real environment with synthetic attacks, real dataset, sanitised background events, and synthetic background events. Each class has its own limitations and benefits. In this section, such limitations will be explored.

A dataset with synthetic background events is very costly and difficult to create [68]. By costly we refer to the fact that the environment for dataset generation must comprise the appropriate combination of hardware architecture, operating system, operating system version, libraries, compilers, software, and software version where the combination must be vulnerable to attacks that will be run against the environment [43]. Also, such a dataset should resemble the real world environment with some varieties of background events and attacks. Attacks should be practical, repeatable, and controllable (e.g., worm containment) [66]. Example of datasets of this class are the DARPA datasets. The claims that background events in the DARPA datasets are close to the real environment are not clearly proved [65, 66]. Regarding the variety of attack issues, the DARPA datasets contain only four web attacks [43].

In addition to the dataset generation issues discussed above, there are also issues regarding the required characteristics of a dataset in relation to the intrusion detection approach. For instance, some anomaly-based intrusion detection approaches require two set of datasets: training data and data to be analysed. The training data must contain only normal or benign events.

A dataset must also be comprised of the appropriate type and level of recorded events. For example, a network-based IDS typically requires captured network traffic as input. Thus, in this case, the dataset must contain captured network traffic.
In summary, there is currently no perfect set of IDS evaluation criteria [28]. Typically, IDS evaluation criteria are based on TP and FP rates. However, there are issues regarding these two measures as discussed above. IDS evaluations are costly and resource intensive. There have been several efforts to evaluate IDSs but only one of them, the DARPA IDS evaluation project, has been widely accepted. However, there are limitations in the DARPA IDS evaluation which were identified by McHugh. Dataset and system implementation availability are also issues in existing IDS evaluations. Publishing dataset and system implementation may be subject to privacy and contracts issues. But without access to dataset and implementation, the validity of results are not verifiable, thus, published results are often questionable.

5.3 Evaluation of the IDS Prototype

The IDS prototype described in Chapter 4 has been evaluated. This section describes the methodology, objectives, and configurations that were used.

This section is organised as follows. Section 5.3.1 describes the evaluation methodology that has been employed. The evaluation procedures will be described. Section 5.3.2 describes the configurations of the IDS prototype that were used for the evaluation. Section 5.3.3 identifies the objectives of the evaluation.

5.3.1 Evaluation Methodology

Our system evaluation employs the traditional IDS evaluation methodology where our system was operated against one or more datasets while some aspects of our system was monitored such as functional characteristics and ability to detect multi-step attacks. Our evaluation comprises three steps as follows:

1. Recorded event parsing and event generation: Firstly, all recorded events from two labelled datasets, the Scan of the Month (SOTM) 34 [103] and a synthetic dataset, are parsed using the DSS and AEM parsers. The parsers generate corresponding sensor events and derived events.

2. Attack Detection: execute the signature detection engine where the engine is equipped with a set of pre-defined signatures. Alarms were recorded for later analysis.
3. **Results Analysis**: the number and details of alarms are analysed and compared to the details of the datasets.

We now describe the configuration of our *IDS prototype* for use in the evaluation.

### 5.3.2 Configuration of the IDS Prototype for System Evaluation

The IDS prototype operates in *off-line* mode. The configuration of the *IDS prototype* is shown in Figure 5.2. All components of the prototype including the database are installed and run on one host. The operation of the prototype is divided into two phases: event transformation and scenario detection, henceforth called Phase 1 and Phase 2 respectively.

![Diagram of IDS Prototype Configuration](image)

**Figure 5.2: Configuration of the *IDS prototype*.**

In Phase 1, *sensor events* and *derived events* corresponding to *recorded events* from the datasets are generated. The operations in this phase are divided into
three steps: preparing the database, parsing recorded events, and generating derived events.

1. For the purpose of the evaluation and according to the requirements of our design (discussed in Chapter 4), the DSS and AEM repositories are implemented using the PostgreSQL database [83]. The DSS repository has been preloaded with 13 sensor event tables. The AEM repository has been preloaded with altogether 50 tables of derived event types and abstract events.

2. The DSS parsers transform recorded events from the two datasets into corresponding sensor events. The sensor events are stored in the DSS repository. The number of generated sensor events are presented in Section 5.5.1 below.

3. The AEM parsers transform sensor events into corresponding derived events. The generated derived events are, then, stored in the AEM repository. The number of generated derived events is presented in Section 5.5.1 below.

In Phase 2, the IDS prototype carries out attack detection. The scenario detection engine reads a signature (one at a time) from the list of signatures and generates an SQL statement corresponding to the signature. Details of SQL generation are described in Chapter 4. A database query based on the SQL statement is then made to the AEM repository. If the AEM repository returns instances of derived events, attacks are detected, otherwise the AEM repository does not contain the attack.

We now define the evaluation objectives.

5.3.3 Evaluation Objectives

Traditionally, the goal of an IDS evaluation is to determine detection rates and false positive rates. As a result, our evaluation results will include an analysis of detection rates and false positive rates to follow the tradition. The objectives of our system evaluation are to evaluate the following:

Validity of event processing: The objective of this test is to validate the canonical event representation and event abstraction provided by the AESA. The result of this test is the comparison between the number of recorded events present in the two datasets and the number of generated sensor events and derived events.
Platform independent attack representation: The objective of this test is to demonstrate that the IDS prototype can specify attacks using derived events and abstract events. The IDS prototype can specify and detect an attack regardless of implementation and software using only one signature.

Ability to specify multi-step attacks: The objective of this test is to verify that our scenario specification language can express multi-step scenarios. This test also verifies that our system can correlate events from heterogeneous sources. Because typically, in order to detect a multi-step attack scenario, a system must be able to handle events derived from multiple sources. The result of this test is the answer to whether our system able to specify and detect multi-step scenarios.

Signature composition ability: Signature composition allows signature writers to reuse existing signatures to write a new signature. The objective of this test is to determine whether our system has the ability to recognise composite signatures. The result of this test is the answer to whether our system can composite signatures.

Functional characteristics: The objective of this test is to determine accuracy and completeness of the IDS prototype using: no clock skew compensation, constant skew compensation, and linear regression. The IDS prototype is tested against the SOTM34 dataset. The results of this test are number of TPs and FPs.

We now discuss the details of the two datasets followed by the configuration of our test environment. Evaluation results based on these objectives are presented in Section 5.5 below.

5.4 The Datasets

To the best of our knowledge, there are very few datasets for IDS evaluations which are publicly available. One of the most cited and used datasets is the DARPA datasets. However, there are several drawbacks and limitations in the DARPA datasets, as we have already discussed in Section 5.2. Therefore, there is a need for a dataset which represents real data and is publicly available. In our evaluation, we used two datasets: a dataset from the Scan of the Month (SOTM)
project [103] (see below) and a synthetic dataset. The details of the two datasets are now discussed.

### 5.4.1 The Synthetic Dataset

We have created a synthetic dataset which consists of attacks against a web-based application and system administrator login events on two operating systems. The generation of the dataset emphasised the demonstration of the event abstraction feature provided by the AESA.

The attack against a web-based application is based on input validation errors in the PHP Bulletin Board (phpBB) [93]. phpBB is installed onto two web servers: the Apache web server on the Linux operating system and the Microsoft IIS on Microsoft Windows XP Professional. The two web servers and an attacker’s machine are setup using three virtual machines (VMWare). The Linux host (victim) is named ‘feisty’ with the IP address 192.168.0.92. The Windows XP host (victim) is named ‘xppro’ with the IP address 192.168.0.149. The attacker’s host has IP address 192.168.0.150. The vulnerable version of phpBB (version 2.0.6) was installed on both victim hosts. On feisty, phpBB was run on the Apache web server. On xppro, phpBB was run on the Microsoft Internet Information Service (IIS) server. Several attacks were launched from the attacker’s host against both victim hosts. Web server log entries were collected from both victim hosts where logs were recorded in the Apache combined log format and IIS log format.

The dataset also contains events of a system administrator successfully logged into two operating systems through different services. We have setup four hosts: three hosts running the Linux operating system and a host running Microsoft Windows XP Professional. One of the Linux hosts, namely ‘honey3’, provides the Secure Shell (SSH) service. We used SSH client from the other two Linux hosts to connect to SSH on the target host with ‘root’ account using two different authentication methods: password and public key authentication. On the Windows XP host, namely ‘forensics4’, we logged into the ‘administrator’ account at the console. Note that the format of Windows Event log contains multiple lines per record. Thus, for ease of processing, the Windows Event Log entries have been pre-processed combining multiple-lines into one line per log entry.

The number of recorded events in the dataset appear in Appendix C.
Instances of Attacks in the Synthetic Dataset

phpBB Attack

phpBB is a popular open source bulletin board system written in the PHP programming language [105]. Early versions of phpBB (from 1.0 to 2.0.7) are vulnerable to at least one input validation attack [93]. Multiple scripts in the phpBB failed to sanitise user input which lead to possible SQL injection attacks. The particular attack we used in our synthetic dataset was the attack against the ‘admin_smilies.php’ script where an SQL statement is inserted to retrieve unauthorised information from a back-end database. The attack was executed by sending a HTTP request to the vulnerable host which contains the pattern:

GET /admin/admin_smilies?mode=edit&id=select%20user_id,user_level,username,user_password%20from%20phpbb_users%20where%20user_id=1
HTTP/1.0

This attack returns the hashed password of a phpBB user whose ID is ‘1’ (administrator) from the database. An attacker can, later, crack the hashed password and retrieve the real password of the phpBB administrator. However, more harmful attacks could be used such as an SQL statement to delete a user account. There are 14 instances of the attack in our synthetic dataset where 13 of them were launched against feisty (Apache on Linux) and one of them was launched against xpro (IIS on Windows).

System Administrator Logging into a System

This scenario may not cause direct harmful damage to a target system. However, this scenario may be the first step of a multi-step scenario. The objective of this particular scenario is to demonstrate the application of event abstraction of the AESA. In particular, it demonstrates the ability to specify one abstract event which is automatically mapped to multiple derived events based on the relationships defined in the AEM described in Chapter 3.

The scenario comprises two separate events: using SSH to login remotely with the account ‘root’ on honey3 and logging in to a Windows XP machine on the console using ‘administrator’ account on forensics4. There were two instances of the SSH login to root account and two instances of Windows XP login to administrator account.

The total number of instances of both attacks is shown in Table 5.1.
<table>
<thead>
<tr>
<th>Attack Names</th>
<th>Number of instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB Attack</td>
<td>14</td>
</tr>
<tr>
<td>System Administrator Login</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.1: Number of attack instances in the synthetic dataset.

5.4.2 The Scan Of the Month Dataset

The scan of the month (SOTM) dataset 34 [21] from the Honeynet project [104] is suitable for our evaluation as it represents real data and real attacks. Also, the dataset contains background events (non attacks events) which can be used to test the ability to distinguish attacks from legitimate events (accuracy) of our IDS prototype. Another advantage of using the SOTM dataset is the fact that each dataset (challenge) has been analysed and validated by several participants. Thus, for our evaluation purposes, scenario definitions corresponding to the analysis provided by participants can be developed with ease.

The SOTM34 dataset was collected from a honeynet network. According to the official solution [22], the configuration of the honeynet network is shown in Figure 5.3. The honeynet network comprises five hosts namely: Bridge, Bastion, Combo, a fake remote syslog and one router. The real IP addresses of hosts in the honeynet were replaced with private non-routable addresses (11.11.79.0/24). The honeynet network was connected to the Internet through the router. Bridge is a Layer 2 filtering firewall implemented using Linux’s iptables. It recorded the information of incoming and outgoing traffic, e.g., source and destination addresses and some TCP attributes. Bastion is a Snort IDS running in passive mode. Neither Bridge nor Bastion had IP addresses. Combo connected to the same hub as Bastion and was configured with multiple IP addresses, as shown in Figure 5.3. There was a fake remote syslog in the network which did not receive any syslog messages from any host in the network, and thus has no significance in this dataset.

The details of software and software version installed on each host were provided in the solutions submitted by participants of the SOTM project [7, 51, 88]. Bridge used an unknown version of the Linux operating system. Combo ran Red-Hat 9 with the kernel version 2.4.20-8. Combo ran server software such as the Apache web server, Sendmail mail server, and SSH server. Combo also ran AWStats software which is an Apache web server monitoring tool. Readers are referred to the solution written by Richard et. al. [88] for a comprehensive analysis of soft-
Figure 5.3: Network configuration of the honeynet in the SOTM34.

ware and software versions installed on Combo. Bastion ran an unknown version of the Snort IDS and thus no detail on the signatures used by Snort were available.

The SOTM34 dataset comprises six types of logs: iptables firewall logs, Snort IDS alerts, Apache access logs, Apache error logs, Apache SSL error logs, and Linux syslog messages. The number of log records, log file names, and their logging duration appear in Appendix C. The iptables firewall logs are retrieved from Bridge. The Snort IDS alerts are retrieved from Bastion. The rest of the log entries are retrieved from Combo.

We now discuss two aspects of the SOTM34 dataset that must be addressed as follows: clock skew and drift issues and the data sanitising issues.

Clock Skew and Clock Drift Issues

Clocks on Bastion and Bridge were synchronised using NTP. However, due to a system misconfiguration, the clock on Combo was not synchronised and thus was approximately 4 hours 47 minutes slower than the clocks on Bridge and Bastion [22].
Using the mechanism to determine clock skew described in Chapter 3, we have chosen the clock on Bridge as the reference clock due to the fact that Bridge is the gateway to Combo. Thus, it is easy to correlate entries from Apache access log derived from Combo and iptables log entries derived from Bridge. Any HTTP requests destined to Combo would be recorded by iptables on Bridge. We manually correlated log entries from the Apache access log on Combo with the iptables log entries derived from Bridge. The time period where the entries of the Apache access log (from Combo) and the entries of iptables log (from Bridge) overlapped is between 25 February 2005 and 17 March 2005 (henceforth referred to as the sampling period). We correlated log entries from the two sources using the following mechanisms:

- we extracted log entries with unique IP addresses and their corresponding timestamps from the Apache access log. If there are multiple log entries from the same IP address, the entry with the earliest timestamp is used. This set of unique IP addresses is called the Apache Unique IP (AUIP) set.

- we extracted timestamps from the iptables log entries where the source IP address exists in the the AUIP set, the destination IP address is one of the IP addresses of Combo (the 11.11.0.0/16 subnet), and the destination port is 80. If there are multiple log entries with the same IP address, the entry with the earliest timestamp is used.

We identified 292 IP addresses where corresponding log entries exist in both the Apache access log and the iptables log. The smallest clock skew between log pairs (a log entry derived from the Apache access log and a log entry derived from the iptables log which represents the same event), is 4 hours 28 minutes and 22 seconds and the largest clock skew is 4 hours 49 minutes and 15 seconds. We have manually determined the drift rate of the clock on Combo based on the 292 log entries (based on the AUIP set). We found that the clock on Combo was drifting away from the clock on Bridge at a more or less constant rate of 5 seconds per day.

There were some difficulties during the correlation process between Combo and Bridge due to the following reasons. Firstly, the number of log entries in the Apache access log and the iptables log, during the sampling period, is not consistent. There are 2,406 log entries in the Apache access log while there are 3,183 corresponding log entries in the iptables log for the same period.
Secondly, there are several instances where the number of Apache access log entries from a particular client IP address is not equal to the number of corresponding iptables log entries. For instance, there are 19 log entries in the Apache access log whose client IP address is 220.170.88.36 but there are only 6 corresponding log entries in the iptables log.

Thirdly, there are several client IP addresses which exist in the Apache access log but do not exist in iptables log entries and vice versa.

The first two issues could possibly be caused by the implementation of the HTTP client (e.g., web browser). Web browsers may make several HTTP requests using the same TCP session (same source port). Such requests are recorded as one iptables log entry but the same set of requests are recorded as multiple Apache access log entries. The third issue may be caused by the limitation of hardware where some network packets were dropped during a high load.

Note that due to the inconsistencies between the Apache access log and iptables log, we can correlate only some parts of the log entries.

**Errors from Data Sanitising**

Several log entries in the iptables firewall log (Bridge) entries and Snort alerts (Bastion) are shown with wrong formatting. This is caused by the errors in the data sanitising script. For example, in the iptables firewall log entries, there are three records in the format “SRC=63.19.10.22 DST=11.11.79.90” whereas the correct format is “SRC=63.19.10.22 DST=11.11.79.90”. The original destination IP address was replaced by “.DST=11.11.79.90”. The records with wrong formatting have been replaced with the correct format manually before being used in the evaluation.

We now identify types of attacks in the SOTM34 dataset.

**Instances of Attacks in the Scan of the Month 34 Dataset**

According to the recorded events in the dataset, Combo was compromised several times. Attacks against Combo can be classified into three classes: policy violation, exploiting software vulnerability, and multi-step attack scenarios. The details of these attacks are as follows.
Policy Violation (PV)

The author of the SOTM34 dataset does not provide the security policy in use for the honeynet. However, it is stated that Combo was run as a honeypot with several services. Therefore, we can assume that Combo should not instantiate any connection to the Internet. Several malicious events occurred on Combo which could be considered to violate the policy that Combo not initiate Internet connections. Three types of policy violation activities have been identified as follows:

- **SSH login into a user account (PV1):** Since Combo is a honeypot machine, there should not be any user account nor should remote user login be allowed.

- **IRC events (PV2):** The honeynet was not running any IRC software. However, there are a number of IRC activities originate from Combo.

- **Outbound HTTP connections (PV3):** Combo is a honeypot, and thus it is unusual that Combo makes connections (as web client) to web servers on the Internet.

Exploiting Software Vulnerability (SV)

AWStats is a program written in the Perl programming language, which provides statistical information of a web server. The version of AWStats [31] installed on Combo is vulnerable to an input validation error which allows remote users to execute arbitrary commands on the vulnerable host [41]. Such arbitrary commands are executed with the privilege of the user who runs the web server. The vulnerability is caused by the failure to sanitise user inputs. In particular, users can insert arbitrary commands between the character ‘|’ or ‘%7c’ (%7c is the ASCII encoded of |) after the parameter ‘configdir’. In the Perl programming language, if a file name begins or ends with the ‘|’ character, the file name is interpreted as a command where the content of the file is piped to the command or the output of the command is piped to the file [3].

The success of the attack is determined by the returned HTTP status code in the Apache access log. The attack against AWStats succeeded 76 times. An attack against AWStats is Step 1 of the multi-step attacks (MSS1, MSS2, and MSS3) described below.
Multi-Step Attack Scenarios (MSS)

There were traces of four types of multi-step attack scenarios in the SOTM34 dataset. The details of the scenarios are based on our analysis and the solutions from [7, 22, 51, 88].

The analysis of MSS scenarios discussed below has taken clock skew and clock drift into account. In particular, timestamps of recorded events derived from Combo have been manually adjusted so that they would be consistent with recorded events derived from Bridge and Bastion.

For simplicity, timeouts are applied to each scenario in order to limit the time horizon of scenarios described in this section. The number of instances of MSS indicated in this section are relative to the timeouts. Timeouts are applied after timestamps have been adjusted.

There are four types of MSS in the SOTM 34 as follows.

MSS1: Exploit AWStats, download and run IRC bot

An adversary exploited the vulnerability in AWStats on Combo which allows the adversary to install and run an IRC bot on the victim host. The scenario comprises three steps as follows:

1. An attacker attacks AWStats on Combo in the manner described above (SV). Evidence of this step is present in the Apache access log entries.

2. The attacker uses Combo to downloads an IRC bot from the Internet. Traces of this step are present in the iptables firewall log entries. This step occurs within one minute of Step 1.

3. The IRC bot on Combo generates an IRC activity. This step was detected by the Snort IDS. This step occurs within 20 seconds of Step 2.

There are two recorded events that match the condition of Step 1, two recorded events that match the condition of Step 2, and 18 recorded events that match the condition of Step 3. Let reS.N represent a unique recorded event where S is the attack step 1, 2 or 3, and N be the number of event instance. For example, re1.1 represents the first recorded event that matches the condition of Step 1. The recorded events that match the condition of Step 3 can be divided into 2 groups: re3.1 – re3.9 and re3.10 – re3.18 corresponding to the two separate instances of
Steps 1 and 2. Using this definition, the number of instances of MSS1 can be considered to be either:

- two instances where Step 3 is a collection of recorded events. The first instance comprises: \(re1.1, re2.1, [re3.1, re3.2, ..., re3.9]\). The second instance comprises: \(re1.2, re2.2, [re3.10, re3.11, ..., re3.18]\).

- 18 instances where Step 3 of each attack instance is a single event. The attack instances are:
  
  \([re1.1, re2.1, re3.1], ..., [re1.1, re2.1, re3.9], [re1.2, re2.2, re3.10], ..., [re1.2, re2.2, re3.18]\).

In practice, the decision as to whether 2 or 18 attack instances occurred is a subjective one. For our evaluation, we considered the latter case as the recorded events that match the condition of Step 3 (\(re3.1 - re3.18\)) are different events.

**MSS2: Exploit AWStats, download and run a backdoor program**

This scenario is similar to the previous scenario but a backdoor program was downloaded instead of an IRC bot. The scenario comprises three steps as follows:

1. An attacker attacks AWStats on Combo in the manner described above (SV).
2. The attacker uses Combo to downloads a backdoor program from the Internet. This step occurs within one minute of Step 1.
3. The attacker connects to the backdoor on Combo. This step occurs within ten minutes of Step 2.

The backdoor was successfully installed and run on Combo. There is one instance of MSS2 in the dataset.

**MSS3: Exploit AWStats, download a backdoor program, connect to the backdoor, and make an outbound HTTP connection**

This scenario can be considered to be an extension to the MSS2 scenario. There is one additional step to the scenario where an adversary made multiple outbound HTTP connections. The scenario comprises the following steps:

1. The attacker attacks AWStats on Combo in the manner described above (SV).
2. The attacker uses *Combo* to downloads a backdoor program from the Internet. This step occurs within one minute of Step 1.

3. The attacker connects to the backdoor on *Combo* one day later. This step occurs within one day of Step 2.

4. *Combo* (used by the attacker) makes an outbound HTTP connection to a server on the Internet. This step occurs within one minute of Step 3.

There is one instance of MSS3 in the dataset.

**MSS4: Connect to the backdoor and make an outbound SSH connection**

The objective from an attacker point of view is to use a compromised machine as an attack launchpad so that the attack would be more difficult to trace back to the attacker. The scenario comprises the following steps:

1. The attacker connects to the backdoor on *Combo* which has been installed from either MSS2 or MSS3.

2. *Combo* (used by the attacker) makes an SSH connection to a server on the Internet. This step occurs within ten minutes of Step 1

In the dataset, there is one connection to the backdoor on *Combo* (Step1) and there are SSH connections to 13 hosts (195.115.0.1 – 195.115.0.13) that are originated from *Combo* (Step 2). Let $reS.N$ represent a unique recorded event where $S$ is the attack step: 1, or 2, and $N$ be the number of event instance. The number of instances of this scenario can be considered to be either:

- one instance of MSS4 where there is one instance of Step 1 and 13 instances of Step 2: $re1.1,[re2.1,\ldots,re2.13]$; or

- 13 instances of MSS4 where each instance comprises Step 1 followed by each of the 13 instances of Step 2: $[re1.1,re2.1],[re1.1,re2.1],\ldots,[re1.1,re2.13]$.

In our evaluation, we considered the latter to be the case as there are 13 different victim hosts. Ideally, we would like to identify MSS4 as one instance with an aggregated set of recorded events that match the condition of Step 2. The 13 SSH connections are specific to the SOTM 34 dataset. The number of SSH connections can be arbitrary.
Number of Attack Instances

The number of attack instances described above are shown in Table 5.2.

<table>
<thead>
<tr>
<th>Attack Names</th>
<th>Number of instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>PV1</td>
<td>145</td>
</tr>
<tr>
<td>PV2</td>
<td>156</td>
</tr>
<tr>
<td>PV3</td>
<td>16</td>
</tr>
<tr>
<td>SV</td>
<td>76</td>
</tr>
<tr>
<td>MSS1</td>
<td>18</td>
</tr>
<tr>
<td>MSS2</td>
<td>1</td>
</tr>
<tr>
<td>MSS3</td>
<td>1</td>
</tr>
<tr>
<td>MSS4</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.2: Number of attack instances in the SOTM34 dataset.

5.5 Evaluation Results

This section presents results of the evaluation of our *IDS prototype* against the synthetic and SOTM34 datasets. The results are presented based on the evaluation criteria identified in Section 5.3.3. Section 5.5.1 presents the results of validating the event processing using parsers provided by the AESA. Section 5.5.2 presents the ability to represent an attack regardless of the details of the target system or the particular implementation of the attack using *derived events* and *abstract events*. Section 5.5.3 presents the ability of the *IDS prototype* to specify multi-step attacks. An example of signature of a multi-step attack scenario from the SOTM34 dataset is given. Section 5.5.4 demonstrates the ability of our Python-based signature language to specify composite signatures. Section 5.5.5 presents the functional performance characteristics (detection performance) of the *IDS prototype*. The detection performance of the IDS prototype using the constant skew compensation technique and the linear regression technique is presented in Section 5.5.6 and Section 5.5.7 respectively.
5.5.1 Validity of Event Processing

Two datasets were used in our evaluations: the SOTM34 dataset and our synthetic dataset. *Recorded events* from both datasets was parsed by our DSS and AEM parsers. The results of *recorded event* parsing and DSS (*sensor events*) and AEM (*derived events*) generation are as follows.

**Sensor Events based on the Data Source Schema**

The number of sensor events derived from the synthetic dataset and SOTM34 dataset are shown in Table 5.3. Note that 16 syslog messages (eight in maillog, and eight in messages) were not parsed because their message bodies are in the form “last message repeated $n$ times” where $n$ is a number. Such messages do not record the name of the process name that generates these messages. Thus, these messages were discarded.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Entries</th>
<th>Sensor Events</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synthetic Dataset</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>access_log</td>
<td>232</td>
<td>apache_combined</td>
<td>232</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_log_format</td>
<td></td>
</tr>
<tr>
<td>in08021723.log</td>
<td>17</td>
<td>iis_log_file</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_format</td>
<td></td>
</tr>
<tr>
<td>honey3-syslog</td>
<td>2</td>
<td>unix_syslog</td>
<td>2</td>
</tr>
<tr>
<td>forensics4-logon</td>
<td>26</td>
<td>windows_event</td>
<td>26</td>
</tr>
<tr>
<td>-WinXP.csv</td>
<td></td>
<td>_log</td>
<td></td>
</tr>
<tr>
<td><strong>SOTM34 Dataset</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptablesyslog</td>
<td>179,752</td>
<td>unix_syslog</td>
<td>179,752</td>
</tr>
<tr>
<td>snortsyslog</td>
<td>69,039</td>
<td>unix_syslog</td>
<td>69,039</td>
</tr>
<tr>
<td>access_log*</td>
<td>3,554</td>
<td>apache_combined</td>
<td>3,554</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_log_format</td>
<td></td>
</tr>
<tr>
<td>error_log*</td>
<td>3,692</td>
<td>apache_error_log</td>
<td>3,692</td>
</tr>
<tr>
<td>ssl_error_log*</td>
<td>374</td>
<td>apache_error_log</td>
<td>374</td>
</tr>
<tr>
<td>messages*</td>
<td>1,166</td>
<td>unix_syslog</td>
<td>1,158</td>
</tr>
<tr>
<td>maillog*</td>
<td>1,172</td>
<td>unix_syslog</td>
<td>1,164</td>
</tr>
<tr>
<td>secure</td>
<td>1,587</td>
<td>unix_syslog</td>
<td>1,587</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>260,613</td>
<td><strong>Total</strong></td>
<td>260,597</td>
</tr>
</tbody>
</table>

Table 5.3: Number of *recorded events* and *sensor events* derived from the synthetic dataset and the SOTM34 dataset.
5.5. Evaluation Results

Derived events based on the Abstract Event Tree

The number of derived events derived from the SOTM34 dataset and the synthetic dataset are shown in Table 5.4. Nine types of derived events were generated: six types of SSH related derived events, HTTP exchange, TCP exchange, and Snort IDS alert.

<table>
<thead>
<tr>
<th>Derived events</th>
<th>Instances</th>
<th>Derived Events</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssh_authentication</td>
<td>1,551</td>
<td>http_exchange</td>
<td>3,803</td>
</tr>
<tr>
<td>sshIllegal_user</td>
<td>352</td>
<td>tcp_exchange</td>
<td>162,453</td>
</tr>
<tr>
<td>ssh_no_identification</td>
<td>366</td>
<td>snort_syslog</td>
<td>69,039</td>
</tr>
<tr>
<td>ssh_authentication</td>
<td></td>
<td>windows_console</td>
<td></td>
</tr>
<tr>
<td>_timeout</td>
<td>1</td>
<td>_authentication</td>
<td>2</td>
</tr>
<tr>
<td>ssh_scan</td>
<td>8</td>
<td>Total</td>
<td>237,575</td>
</tr>
</tbody>
</table>

Table 5.4: Number of derived events from the SOTM34 dataset and our synthetic datasets

We have verified that the sensor events and the derived events were generated correctly by comparing the their attribute values to the contents of recorded events in all datasets. We now present the ability to represent an attack regardless of the details of the target system.

5.5.2 Platform Independent Attack Representation using the AESA

The synthetic dataset contains traces of phpBB attacks against two servers: the Apache web server on the Linux operating system and the IIS server on the Windows operating system. The phpBB attack against two types of web servers can be represented using one derived event namely http_exchange. Figure 5.4(a) shows sensor events, derived events, and abstract events corresponding to the phpBB attack. Figure 5.4(b) shows the signature for the phpBB attack using http_exchange. The http_exchange event represents an HTTP request regardless of the web server software.
class phpbb_attack_def(Scenario):
    def __definition__(self):
        self.phpbb_attack_instance = Variable('phpbb_attack_instance', self)
        self.phpbb_attack_instance = Event(http_exchange)
        self.phpbb_attack_instance.http_status_code == '200'
        contains_pattern(self.phpbb_attack_instance.request_uri,
            'admin_smilies.*mode=edit.*select', 'ignorecase')

        (b) Signature for the phpBB attack.

Figure 5.4: Events and signature corresponding to the phpBB attack.

The synthetic dataset contains four instances of successful system administrator logins. In two instances of the scenario, a system administrator logs into the Linux operating system using SSH services. In the other two instances of the scenario, a system administrator logs into the Windows operating system using the console. Figure 5.5(a) shows sensor events, derived events, and abstract events related to the system administrator login scenario. Figure 5.5(b) shows the signature for the scenario using an abstract event namely, authentication_event. The authentication_event event represents all login events on both Linux and Windows
operating systems.

```
class admin_login_def(Scenario):
    def __definition__(self):
        self.admin_auth = Variable('admin_auth', self)
        self.admin_auth == Event(authentication_event)
        one_of_patterns(self.admin_auth.authentication_result, [
            'accept', 'success', 'ignorecase'])
        one_of_patterns(self.admin_auth.user_credentials, [
            'administrator', 'root', 'ignorecase'])

(b) Signature for the system administrator login scenario.
```

Figure 5.5: Events related to the system administrator login scenario.

The detection results of these two scenarios using the signatures shown in Figure 5.4(b) and Figure 5.5(b) are presented in Section 5.5.5. In the following section, we demonstrate the ability of our Python-based signature language to specify multi-step attacks.
5.5.3 Multi-step Attack Specification

Four multi-step attack signatures have been developed, namely, MSS1, MSS2, MSS3, and MSS4 (see Section 5.4.2 for details). As an example, the signature for MSS1 will be presented. Readers are referred to Appendix D for the other signatures.

The signature for detecting MSS1 written in our Python-based signature language is shown in Figure 5.6. The details of the signature are described as follows.

class MSS1_def(Scenario):
    # Step 1: AWStat attack
    self.awstat_attack_instance = Variable('awstat_attack_instance',
        self)
    self.awstat_attack_instance = Event(http_exchange)
    self.awstat_attack_instance.http_status_code = '200'
    one_of_patterns(self.awstat_attack_instance.request_uri, ['/cgi-bin/awstats.pl\?configdir=%7c.*', '/cgi-bin/awstats.pl\?configdir=.*\|.*'])

    # Step 2: Combo makes an outbound HTTP connection
    self.outbound_http = Variable('outbound_http', self)
    self.outbound_http = Event(tcp_exchange)
    one_of_patterns(self.outbound_http.source_address, '11.11.79.\d*', 'combo')
    self.outbound_http.destination_port = '80'
    after(self.outbound_http, self.awstat_attack_instance, '00:01:00')

    # Step 3: IRC events originating from Combo
    self.irc_events = Variable('irc_events', self)
    self.irc_events = Event(snort_ids_alert)
    one_of_patterns(self.irc_events.destination_port, '[6667, 8888]')
    contains_pattern(self.irc_events.alert_message, 'IRC')
    contains_pattern(self.irc_events.alert_classification, 'corporate privacy violation', 'ignorecase')
    self.irc_events.source_address = self.outbound_http.source_address
    after(self.irc_events, self.outbound_http, '00:00:20')

Figure 5.6: Signature for MSS1.

**Step 1 attack AWStats:** This attack step is represented by the variable called `awstat_attack_instance`. The variable is instantiated with `http_exchange` events with the following conditions. The HTTP status code is ‘200’. The URI of the HTTP request contains the string with the pattern:
Step 2 download an IRC bot: This attack step is represented by variable called outbound_http. The outbound_http is instantiated with tcp_exchange events with the following conditions. The TCP connection originates from either the host name ‘combo’ or the IP address ‘11.11.79.*’ (because Combo has multiple IP address aliases), the destination port is ‘80’ and the outbound_http occurs within one minute after Step 1.

Step 3 IRC activities: This attack step is represented by the variable called irc_events. The variable is instantiated with snort_ids_alert events with the following conditions. The Snort alert message must contains the string ‘IRC’ and the alert classification must contain string ‘corporate privacy violation’ (case insensitive). The network packet identified in the Snort alert must originate from the same host as identified in the outbound_http variable. The irc_events must occur after outbound_http within 20 seconds.

The timeout values in all of the after operators are used to limit the time horizon of event pairs to be matched. The timeout helps reduce false positives as there are a large number of events that satisfy the constraints specified in Step 2 and Step 3. Also, the timeout prevents the scenario detection engine to be overloaded as it limits the number of events the occur within the timeout range between Step 1 and Step 3.

Due to the clock skew and clock drift issues in the SOTM34 (see Section 5.4.2), the time operators used in this signature must be replaced by either after_constant_skew (implements the constant skew compensation technique) or after_linear_regression (implements the linear regression technique). The detection results using the two types of the time operators are described in Section 5.5.6 and Section 5.5.7 respectively.

In the next section, the ability to define composite signatures is demonstrated.

5.5.4 Signature Composition

Signature composition is, to the best of our knowledge, a novel feature of our system. Signature composition allows signature writers to incorporate and thus
reuse existing signatures as parts of larger scenarios and also allows signature writers to specify additional predicates to existing scenarios.

This section presents an example of how the previous scenario MSS1 can be specified in the form of a composite signature. Readers are referred to Appendix D for composite signatures of MSS2, MSS3, and MSS4.

The three steps of the signature of MSS1 (see Figure 5.6) can be rewritten into three single-step signatures. The contents of the three signatures are shown in Figure 5.7 (the libraries importing statements are not shown). The `awstat_attack.py` represents Step 1 of the MSS1 which is a successful attack against one of the AWStats vulnerabilities. The `outbound_http_connections.py` detects any outbound HTTP connection. This signature corresponds to Step 2 of MSS1. The `irc_events.py` represents alerts generated by the Snort IDS about IRC events.

```
# awstat_attack.py
class awstat_attack_def(Scenario):
    def __definition__(self):
        self.awstat_attack_instance = Variable('awstat_attack_instance', self)
        self.awstat_attack_instance == Event(http_exchange)
        self.awstat_attack_instance.http_status_code == '200'
        one_of_patterns(self.awstat_attack_instance.request_uri, ['/cgi-bin/awstats.pl\?configdir=*%7c.*', '/cgi-bin/awstats.pl\?configdir=*\|.*'])

# outbound_http_connections.py
class outbound_http_connections_def(Scenario):
    def __definition__(self):
        self.outbound_http = Variable('outbound_http', self)
        self.outbound_http == Event(tcp_exchange)
        self.outbound_http.destination_port == '80'

# irc_events.py
class irc_events_def(Scenario):
    def __definition__(self):
        self.irc_events = Variable('irc_events', self)
        self.irc_events == Event(snort_ids_alert)
        one_of_patterns(self.irc_events.destination_port, ['6667', '8888'])
        contains_pattern(self.irc_events.alert_message, 'IRC')
        contains_pattern(self.irc_events.alert_classification, 'corporate privacy violation', 'ignorecase')
```

Figure 5.7: Three signatures based on three steps of MSS1.

With our scenario composition ability, we can rewrite the MSS1 scenario by incorporating the three single-step signatures. The new MSS1 signature is shown
in Figure 5.8. AWStats attacks, outbound HTTP connections, and IRC events are defined as three variables namely step1, step2, and step3 respectively.

class MSS1_def(Scenario):
    def __definition__(self):
        self.step1 = awstat_attack_def('step1')
        self.step2 = outbound_http_connections_def('step2')
        one_of_patterns(self.step2.outbound_http.source_address,
                         ['11.11.79.\d+', 'combo'])
        after(self.step2.outbound_http, self.step1.
             awstat_attack_instance, '00:01:00')
        self.step3 = irc_events_def('step3')
        self.step3.irc_events.source_address == self.step2.
        outbound_http.source_address
        after(self.step3.irc_events, self.step2.outbound_http,
             '00:00:20')

Figure 5.8: Signature for MSS1 (composite).

The signature composition feature provides a convenient way to reuse existing signatures, i.e., incorporating existing signatures into a complex multi-step signature. A composite signature is more compact and easy to read. For example, a composite signature in Figure 5.8 is simpler and reasonably shorter but still retains the meaning compared to the non composite signatures in Figure 5.6. In addition, the signature composition feature provides a significantly shorter signature and much easier to read compared to multi-step attack signature in existing languages such as STATL and P-BEST. For example, in a scenario that comprises two steps: failed user login followed by machine reboot, the signature for such a scenario written in STATL and P-BEST would require the reiteration of the entire failed user login signature as shown in Figure 4.8 (for STATL) and Figure 4.9 (for P-BEST).

We now present the attack detection results of our IDS prototype with no clock skew compensation.

5.5.5 Attack detection with no Skew Compensation

We have developed a set of signatures for attacks present in the synthetic and SOTM34 datasets. All instances of single-step attacks (phpBB attack, system administrator login, PV1, PV2, PV3, and SV) have been successfully detected with no false alarms. The number of alerts, TPs, and FPs generated by our IDS prototype are presented in Table 5.5.
<table>
<thead>
<tr>
<th>Attack Names</th>
<th>Instances</th>
<th>Alarms</th>
<th>TP</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB attack</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>System Admin Login</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>PV1</td>
<td>145</td>
<td>145</td>
<td>145</td>
<td>0</td>
</tr>
<tr>
<td>PV2</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>0</td>
</tr>
<tr>
<td>PV3</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>SV</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>MSS1</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MSS2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MSS3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MSS4</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.5: Detection results (no skew compensation) for the synthetic and SOTM34 datasets.

MSS1, MSS2, and MSS3 have not been detected due to the clock skew issues (but this is addressed in the following sections). All three scenarios have a common pattern for detecting Step 1 and Step 2. To detect Step 1 (AWStats attack), a signature must specify constraints on \texttt{http\_exchange} events \textit{(derived events)}. All \texttt{http\_exchange} events in the SOTM34 are derived from \textit{Combo}. To detect Step 2 \textit{(Combo initiates HTTP connection to the Internet)}, a signature must specify constraints on \texttt{tcp\_exchange} events \textit{(derived events)}. All \texttt{tcp\_exchange} events in the SOTM34 are derived from \textit{Bridge}. As stated in Section 5.4.2, the clock on \textit{Combo} is significantly slower than clocks on \textit{Bridge} and \textit{Bastion}. Thus, all three scenarios have not been detected. Two techniques to address clock skew and clock drift as described in Chapter 3: constant skew compensation and linear regression. The evaluation results of applying the two techniques to these three multi-step attack scenarios are presented in Section 5.5.6 and Section 5.5.7.

All 13 instances of MSS4 have been detected with no FPs as MSS4 is not affected by clock skew. Ideally, we would like to report one instance of MSS4 where \textit{derived events} that match Step 2 are presented as a set of 13 SSH connections (see Section 5.4.2). Such an aggregation cannot be made in the current implementation. To address this limitation, the signature language and scenario detection engine must implement the expression “one or more”. Alternatively, the prototype may incorporate a post-processor which aggregates a group of alarms that have a
common event in an attack step. These two solutions are not in the scope of this work, and thus are identified as future work and are discussed in Chapter 6.

We now present the results of applying the constant skew compensation techniques to detect MSS1, MSS2, and MSS3.

### 5.5.6 Evaluation of Multi-Step Attack Detection using Constant Skew Compensation

The clock on *Combo* is slower than clocks on *Bridge* and *Bastion* (see Section 5.4.2 for more details). Clock skew affects only three scenarios namely, MSS1, MSS2, and MSS3. Single-step attacks and MSS4 are not affected by clock skew. Single-step attacks are not affected because traces of such attacks are derived from a single event source. MSS4 is not affected because traces of all attack steps in MSS4 are derived from the same host (*Bridge*). In order to use the constant skew compensation operators, an “optimal” clock skew must be identified. The “optimal” clock skew value can be estimated as a best fit using the mean or median of the range of clock skew values or it can be estimated by a series of experiments. In this section, we have chosen to conduct a series of experiments which increment the clock skew values from the smallest clock skew value to the largest clock skew value in order to find the clock skew value with maximum TP rate and minimum FP rate.

Using the methods described in 5.4.2 to identify clock skew of *Combo* while using the clock on *Bridge* as the reference clock, we identify a set of clock skew values (292 values), in order to make sure that the skew values cover entries which may have not been analysed, we have set the lower bound to be smaller than the smallest identified skew (4 hours 28 minutes and 22 seconds) and the upper bound larger than the largest identified skew (4 hours 49 minutes and 15 seconds).

A series of experiments have been conducted with increments of the clock skew compensation value starting from 4 hours 28 minutes 20 seconds to 5 hours. The constant skew technique is applied only to MSS1, MSS2, and MSS3, as *recorded events* corresponding to Step 1 of all three scenarios (AWStats attacks) are derived from *Combo* and recorded events corresponding to Step 2 are derived from either *Bridge*. All occurrences of the *after* operator in the signatures for all three scenarios have been replaced with *after_constant_skew*. Table 5.6 shows MSS1, MSS2, and MSS3 detection results after applying the clock skew compensation value. Each row represents a range of clock skew values with identical TP and FP. The clock
skew value ranges in the table represent groups of skew values which have common TP and FP results. The skew compensation value, where the TP and FP results are underlined, represents the optimal clock skew value for a particular scenario. For example, the clock skew compensation values between 4 hours 47 minutes 15 seconds and 4 hours 47 minutes 16 seconds is the optimal clock skew compensation value for MSS2.

All instances of MSS1 and MSS3 have been detected with no false alarms. Regarding MSS1, as described in Section 5.4.2, ideally we would like to report this attack as two instances where Step 3 for each instance is reported with a set of nine derived events. The signature of Step 3 should be expressed as “one or more”. The solutions to this problem are the same as identified in Section 5.5.5: implement “one or more” semantics or incorporate a post-processor which aggregates a group of alarms that have a common event in Step 1 and Step 2. Such solutions have been identified as future work and are discussed in Chapter 6.

The single instance of MSS2 has been detected with eight false positives. These false positives are caused by a lack of precision of the MSS2 signature. In this particular case, the signature for detecting Step 1 should specify a more precise pattern of the HTTP request string. Currently, such a pattern is defined as either of the following patterns:

"/cgi-bin/awstats.pl/\?configdir=.*%7c.*)" or

<table>
<thead>
<tr>
<th>Clock Skew Values</th>
<th>MSS1 (18 instances)</th>
<th>MSS2 (1 instance)</th>
<th>MSS3 (1 instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4h28m20s - 4h45m59s</td>
<td>0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h46m00s - 4h46m9s</td>
<td>0 9 0 4 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h46m10s - 4h46m11s</td>
<td>0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h46m12s - 4h46m13s</td>
<td>1 8 0 4 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h46m14s - 4h46m41s</td>
<td>18 0 0 8 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h46m42s - 4h47m12s</td>
<td>18 0 1 8 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h47m13s - 4h47m14s</td>
<td>1 8 1 4 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h47m15s - 4h47m16s</td>
<td>0 0 1 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h47m17s - 4h47m42s</td>
<td>0 9 1 4 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h47m43s - 4h48m17s</td>
<td>0 9 0 4 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4h48m18s - 5h00m00s</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Detection results after applying constant skew technique to the SOTM34 dataset.
Richard et. al. [88] has identified that as part of the AWStats attack, if the HTTP request string contains the string “wget www.shady.go.ro/a.tgz” the attack tries to download a backdoor program called “a.tgz” from http://www.shady.go.ro. Since the URL and the file name are arbitrary, the MSS4 signature does not contain such a string. Hence, the signature reports eight false positives. Nevertheless, the MSS2 signature can be written so that it will not generate any false positives by using either of the following patterns:

"/cgi-bin/awstats.pl\?configdir=.*\|.*"

The optimal clock skew value for all three scenarios combined together is between 4 hours 46 minutes and 42 seconds and 4 hours 47 minutes and 12 seconds where all instances of all scenarios have been identified. This range of optimal clock skew values is consistent with the analysis conducted by [51, 88]. The skew identified by the authors is 4 hours 47 minutes and 1 seconds which falls in the range of our optimal clock skew value range of 4 hours 46 minutes 42 seconds and 4 hours 47 minutes 12 seconds.

The limitation of this technique is the fact that it works effectively over a short period of time with little to no clock drift. This limitation can be overcome by using the linear regression technique discussed in the next section.

In addition, the technique is subjective and may not be practical. In order to identify the optimal clock skew value, the dataset must be labelled. The type and number of attacks must be known to a system operator. However, labelling such a dataset with clock skew and clock drift issues is a difficult task. In case of a non-labelled dataset, one might use the mean or median of clock skew ranges. For example, in our case study, the average size of the clock skew is 4 hours 47 minutes 39 seconds. But the average, in this case, is not the optimal clock value. As shown in Table 5.6, using this value, our system detects all instances of MSS2 and MSS3 but none of MSS1 instance was detected.

We now present the application of the linear regression technique to address time uncertainty caused by constant clock drift.
5.5.7 Evaluation of Multi-Step Attack Detection using Linear Regression

Using skew identification methods described in Section 5.4.2 to identify skew on Combo where the clock on Bridge is the reference clock, we retrieved a sampling period of 21 days or 292 values of clock skew data. The graph of these clock skew values is shown in Figure 5.9. The graph shows that the clock skew increases slightly everyday. We can conclude that the clock on Combo is drifting away from the reference clock at a more or less constant rate. Thus, the linear regression technique can be applied to this case.

\[ y = 6.046959 \times 10^{-5}x - 4.986802 \times 10^4 \]

Figure 5.9: Best fitting line using the least squares fitting technique.

Using the equations described in Chapter 3, we find the following line of best fit:

\[ y = 6.046959 \times 10^{-5}x - 4.986802 \times 10^4 \]  \hspace{1cm} (5.1)
from the Apache access log converted into seconds since the UNIX epoch. The timestamps are represented on the X axis. The clock skew values have been converted and are represented on the Y axis. From visual inspection, the line derived from the equation fits the data well, i.e., the line passes through the majority of the clock skew values. There are a few outliers to which we speculate that these outliers are due to the inconsistency between the log entries on combo and bridge (see Section 5.4.2 for more details). The number of these outliers are small, thus they are negligible. Clock skew for any given timestamp, where clock drift is taken into account, can be calculated using the Equation 5.1.

In addition, from the Equation 5.1, we can calculate clock drift rate per day. The slope represents the drift rate per second of the clock on combo. Hence, the drift rate of the clock on combo is 5.22 seconds per day. This drift rate is consistent with our observation discussed in Section 5.4.2.

The linear regression technique has been applied to MSS1, MSS2, and MSS3. In signatures of all occurrences of after_constant_skew operators are replaced with after_linear_regression. The detection results are shown in Table 5.7. All instances of MSS1, MSS2, and MSS3 are detected. The linear regression technique produces the same results as the results from constant skew compensation using the optimal value. Eight false positives generated by MSS2 are identical to the false positives in described in Section 5.5.6, and thus the same solution to address the false positives applies.

<table>
<thead>
<tr>
<th>MSS1 (18 instances)</th>
<th>MSS2 (1 instance)</th>
<th>MSS3 (1 instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>FP</td>
<td>TP</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.7: Detection results after applying the linear regression technique.

The false positives report by our system using the linear regression are identical to the false positives in the constant skew compensation. Thus, the same reasons described in Section 5.5.6 apply.

**Varying the Sampling Period**

The slope (clock drift) and y-intercept (clock skew value at the UNIX epoch) shown in Equation 5.1 are derived from the sampling period, between 25 February
2005 and 17 March 2005 (21 days), where log entries from Apache access log and iptables log are overlapped. Using the skew identification methods described in Section 5.4.2, we derived 292 clock skew values from *Bridge* and *Combo* during the sampling period. However, the long sampling period may not be ideal when applied to the real data. Therefore, to demonstrate the usefulness of the technique, the technique should produce similar results when a smaller sampling period is used.

We conducted experiments using smaller sampling periods. Five sampling periods, i.e., between one to five days starting from 25 February 2005 were tested. The detection rate (true positives and false positives) is shown in Table 5.8. Starting from four days sampling period, our system reports the same results as our system using 21 days sampling period. Hence, in this particular case study, four days sampling period is sufficient.

<table>
<thead>
<tr>
<th>No. Days</th>
<th>No. Values</th>
<th>MSS1 (18 inst.)</th>
<th>MSS2 (1 inst.)</th>
<th>MSS3 (1 inst.)</th>
<th>Slope</th>
<th>y-intercept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TP</td>
<td>FP</td>
<td>TP</td>
<td>FP</td>
<td>TP</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>63</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>76</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>292</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.8: Detection results, slope, and y-intercept derived from different size of sampling period.

In summary, the linear regression technique provides an automated mechanism to find the clock skew value at any given timestamp assuming constant clock drift. The technique can derive the linear equation using a small subset of data. The limitation of the technique is that identifying the *same event* in order to derive clock skew values can be a complex task. The future direction of this technique is to automate the skew estimation using a set of scripts assuming definitions of the *same event* for different services have been defined.

### 5.6 Summary

Traditionally, IDS performance is indicated by true positive and false positive rates.
5.6. Summary

We have evaluated our *IDS prototype* using one publicly available dataset, the SOTM34 dataset, and a synthetic dataset. The SOTM34 dataset comprises a variety of single-step attacks and multi-step attack scenarios. Our synthetic datasets were generated to demonstrate the event abstraction ability provided by our system.

We argue that IDS performance indication should not be limited to true positive and false positive rates because these rates by themselves are inconclusive. In addition, they are specific to a particular environment or a particular dataset.

As a result, the objectives of our evaluation have been to demonstrate: validity of the AESA, ability to specify platform independent signatures, ability to specify multi-step scenarios, signature composition, and functional performance characteristics. In addition, two resolutions to the problem of time uncertainty have been applied. Our *IDS prototype* has in the majority of cases produced satisfactory results. In particular, our prototype detected all instances of attacks (both single-step attack and multi-step attack scenario) while generating low false alarms.
Chapter 6

Conclusions and Future Work

This dissertation has investigated several difficult issues in multi-step attack specification and detection in intrusion detection systems (IDSs). These issues can be divided broadly into two areas.

Firstly, event representation and event abstraction are one area of investigation. The absence of a canonical representation of events derived from heterogeneous sources is one of the main obstacles hindering the development of multi-step attack detection. To detect a multi-step attack typically involves events derived from multiple sources. In the current state, log entries and captured network traffic (recorded events) are stored in the syntax that is native to systems or applications. Hence, existing IDSs tend to deal with the heterogeneity of syntax of recorded events on an ad hoc basis.

In addition, there is a need for a comprehensive multi-level abstract event model to facilitate signature specification in an environment with heterogeneous components. An abstract event is a high-level representation of a group of events that are related in some way. In terms of intrusion detection, abstract events allow signature writers to develop generic signatures, i.e., signatures that are not specific to a particular version of software of the system being monitored. Signatures using abstract events are very useful in an environment with heterogeneous components to avoid writing sensor specific signatures. Also, signatures that incorporate abstract events are flexible due to the fact that the same signatures can still be used even if new components have been introduced into the environment. Abstract events are crucial for specifying attack descriptions in an environment with heterogeneous sources.
components. In Chapter 3, the Abstract Event System Architecture (AESA) was proposed as the means to provide both canonical event representation and event abstraction for events derived from heterogeneous sources. The implementation details and evaluation results of the AESA prototype are described in Chapter 4 and Chapter 5 respectively.

Secondly, multi-step attacks are difficult to specify and detect partly due to the issues regarding event representation discussed above and partly due to the complexity of the detection mechanisms required. There exist a few multi-step attack (or scenario) detection systems such as the STAT framework [33] and EMERALD [81]. However, the multi-step attack specification and detection mechanisms employed by these systems are complex. For instance, the STAT framework represents an attack using states and transitions where states represent the state of the system being monitored and transitions represent events that modify the system states. There are three types of states and three types of transitions. Using the correct types of transitions and states when specifying a signature requires an intimate understanding of the matching mechanisms employed by the STAT framework. Also, modelling an attack based on the status of the system being monitored is counter-intuitive. Chapter 3 proposed a simple attack specification and detection mechanism using the unification algorithm. The prototype of the proposed concept was described in Chapter 4. Evaluation methodologies and evaluation results for the prototype were presented in Chapter 5.

Another related and important issue which must be addressed in a multi-step attack specification and detection system is that of time uncertainty. Time uncertainty problems have been largely neglected by IDS research. The timestamp is one of the most important attributes of an event since the timestamp is often used for determining the chronological order of events. However, computer clocks are well known for their unreliability. Computer clocks, in general, suffer from skew and drift. Chapter 3 introduced two techniques to address this issue; the clock skew compensation technique and clock drift modelling using linear regression.

The contributions of this research and future research directions are identified in the following sections. Section 6.1 summarises the design of the AESA and identifies future directions for heterogeneous event representation and event abstraction. Section 6.2 summarises the scenario detection engine built using the unification algorithm and identifies future directions for the development of the engine. Section 6.3 summarises the proposed solutions to address time uncertainty
and identifies future directions which might improve the proposed techniques. Section 6.4 describes the IDS framework proposed in this dissertation and identifies possible further development of the framework. Section 6.5 concludes the thesis.

6.1 Abstract Event System Architecture

The fundamental concepts of the AESA build on the inheritance and abstraction principles borrowed from the object oriented design. AESA comprises two components: the Data Source Schema (DSS) and the Abstract Event Model (AEM). The DSS provides canonical representations (the so called sensor events) of heterogeneous recorded events. Each sensor event represents an entry in a log file or captured network traffic. All recorded events are transformed into sensor events.

The AEM provides an abstract representation of sensor events. The AEM comprises derived events and abstract events. A derived event represents an event that occurred in the system or network being monitored. An instance of a derived event is generated from one or more instances of sensor events which may be of heterogeneous types. Allowing a derived event to be generated from multiple sensor events enables it to represent multiple perspectives of an event, e.g., from system and network perspectives. An abstract event represents a high-level view of a group of derived events or abstract events that share a common semantic.

The canonical event representation provided by the DSS addresses the issue of the diversity of heterogeneous event formats. The DSS provides sensor events which represent recorded events derived from heterogeneous sources.

The abstract event model provided by the AEM defines a comprehensive and multi-level model for representing events. At a high level, the predefined abstract events in the AEM cover operating system, application, and network events.

Since the AESA builds based on the object oriented design, it gains three benefits of object oriented design which are flexibility, extensibility, and reusability. If a new component is installed into the network being monitored, the AESA can be extended without re-factoring. The flexibility and extensibility of the AESA enables the architecture to be used in other applications such as modelling events for computer forensics purposes and network and system monitoring.
AESA Limitations and Future Work

Future work on the AESA should be directed towards improving the two-stage transformation. A recorded event must be transformed from recorded event to sensor event and from sensor event to derived event, so that the derived event can be used in signature specification. In the current implementation, the AESA operates in off-line mode. The AESA can be further developed to operate on event streams and thus be amendable to real-time operation.

Another future direction of the AESA is to implement multiple inheritance. Multiple inheritance here refers to the concept where a derived event can contribute to multiple abstract events. Such a principle will add flexibility to the AEM where a particular derived event can potentially contribute to two different abstract events. For instance, currently remote_authentication is an abstract event that may be mapped from two different derived events, namely, ssh_authentication and windows_remote_authentication. Figure 6.1 shows an example the AEM using multiple inheritance. The lines show the current implementation. A dotted line represents linkage from a derived event to additional abstract events. Using the multiple inheritance design, these two derived events can potentially contribute to two additional abstract events, namely, network_protocol_event and application_event. The multiple inheritance concept can be implemented easily. However, a fully qualified attribute name is required.

![Diagram of multiple inheritance](image.png)

**Figure 6.1:** Example of multiple inheritance.
Also, the AESA still requires performance improvement. The two-stage event transformation can still be improved. If the integration of AEM and DSS parsers has been addressed and the speed of the event transformation has been increased, the AESA can potentially be run in real-time.

6.2 Scenario Detection Engine using Unification

The scenario detection engine proposed in this thesis employs the unification algorithm. An attack signature comprises a list of logical expressions. The unification algorithm defines mechanisms to evaluate such expressions. A logical expression comprises two parts: one or more constants or variables and one or more operators. Logical expressions are used to specify constraints on events. In order to trigger an attack signature (attack detection), the evaluation results of all logical expressions in a signature must return TRUE. The unification algorithm evaluates logical expressions by substituting variables with appropriate values. In the context of this work, the variables are instances of derived events.

The unification-based scenario detection engine provides a simpler multi-step scenario detection technique compared to some existing techniques such as the state-based technique. In unification, the type of event of interest is specified as a unification variable and constraints on the event are specified as subsequent expressions.

Scenario Detection Engine Limitations and Future Work

In the current implementation, a variable can be instantiated by either derived events or abstract events. The future direction of the scenario detection engine should allow a variable to be instantiated by an attribute of a derived event or abstract event or by a constant. Allowing such a feature would improve the flexibility of the scenario detection engine.

In this dissertation, four pre-defined categories of operators have been defined: identity, string, set, and time. There is room for additional operators. Such as a time operator which specifies that two events are non-deterministic or occur simultaneously (in parallel). Another example is a counter operator which returns the frequency of the occurrence of a particular event. A counter operator may be used to implement the concept of event consumption. In particular, the event
consumption concept allows signature writers to specify that a consumed event can be matched only once.

6.3 Resolution of Time Uncertainty

Multi-step attacks often involve a sequence of events where such events occur on different hosts or in different systems. Hence, in order to detect multi-step attacks, an IDS must correlate events from multiple sources. There are only a few criteria which enable an IDS to correlate events from multiple sources: cause and effect, timestamps, and certain other properties. Event correlation based on timestamps is one of the most common techniques used as ordering is usually crucial to success of an attack, e.g., penetration then exploit, whereas the reverse order does not work. Also, timestamps are almost always available, i.e., all events have timestamps.

However, computer clocks are well known for their unreliability. Although there are several clock synchronisation mechanisms, they are often not used regularly or not properly implemented. Thus, asynchronous clocks lead to clock skew and clock drift issues. The mechanisms to determine clock skew and clock drift using event timestamps appear in Chapter 3. Thus, time uncertainty issues occur when correlating events from multiple hosts.

This thesis has explored two techniques to address time uncertainty issues. The first technique uses constant values to compensate for clock skew. This technique, as implemented in the prototype developed for this thesis, allows system operators to specify an individual clock skew compensation value for each host. The technique applies the specified constant values to timestamps of events based on the origin of the event. The new compensated timestamps are used by the scenario detection engine for evaluation.

The second technique uses a linear regression technique to model constant clock drift. This allows the estimation of the clock skew value between a clock and a reference clock at any given time.

Limitations of the Techniques and Future Work

The two techniques have been developed, integrated into the prototype and evaluated producing accurate results. However, the processes of identifying the optimal constant skew value (for the constant skew technique) and calculating the linear
regression equation (for the linear regression technique) must be performed manually. The difficulty in automating the process of determining the clock skew value or the linear regression equation lies in the fact that the process must first identify sets of the ‘same’ or ‘closely related’ events from two different sources. Such a process is difficult. Hence, pairing events from different sources must be done on an ad hoc basis. For instance, the process must determine that a particular HTTP request recorded by a web server corresponds to the related event recorded by a firewall. Future research into this specific aspect of the general event correlation problem is needed, what is needed is to automatically determine skew values and drift rates of the clock on a given host compared to a reference clock. The solution may be defining a set of rules which identify closely related events as recorded by different sources. Nevertheless, the rules have to be defined based on implementation and environment.

6.4 Intrusion Detection System Framework for Detecting Complex Scenarios

One of the outcomes of this thesis is an off-line IDS framework which combines the AESA, the scenario detection engine, the timestamp adjustment modules, and the Python-based signature language. The framework utilises the canonical event representations and event abstraction provided by the AESA. The descriptions of attack scenarios are written as a set of logical expressions, using the Python-based signature language defined in this thesis.

Each signature is a Python class that contains a set of logical expressions. The logical expressions are constructed from constants, variables, and pre-defined operators. The Python-based signature language used in this work is just one possible implementation of a signature language that utilises the AESA and the signature matching engine based on unification.

Signatures and the event repository are the two inputs into the scenario detection engine. A signature is triggered (attack detected) only if the evaluation of all logical expressions of the signature returns Boolean TRUE. If a signature is triggered, variables and instances of derived events that are associated with the signature are sent to the alert reporting module.

Our Python-based signature language is simpler and more intuitive compared to other well-known existing signature languages such as STATL [33] and P-BEST
[59]. Also, as demonstrated in Chapter 4, a signature written in our language is much shorter compared to signatures for the same attack written in STATL and P-BEST. In addition, a signature written in our language can express attack characteristics regardless of the details of the system being monitored by using abstract events provided by the AESA whereas the STATL and P-BEST signatures are specific to one operating system and specific software.

IDs Framework Limitations and Future Work

Alert reporting is not the main focus of this research. In its current state, alert messages contain derived events matching a signature. In the future, the format of these alert messages should comply with a standard such as IDMEF [26].

There are a few improvements that should be addressed in the future work on this framework. Firstly, there should be further development to enable the framework to operate on-line (real-time). Real-time detection brings the further advantage of potentially detecting a multi-step attack in mid stream and possibly preventing it. To achieve real-time detection, the two-stage event transformation must be developed further as discussed in Section 6.2.

Secondly, the Python-based signature language needs some refinement. Currently, an attack signature specified in the signature language is, in fact, a Python program. Thus, the syntax of the signature may not be intuitive to a non-Python programmer. In order to address this issue, a signature pre-processor may be implemented. Signatures may be written in a simpler and more non-programmer friendly way. Alternatively, a syntax-oriented signature editor or a graphical user interface (GUI) editor may be implemented.

Finally, a better signature selection mechanism may be implemented. Currently, during the operation of the framework, signatures to be evaluated are selected sequentially. Attack detection speed can be improved by implementing mechanisms which allow the framework to select signatures more effectively. An example might be the implementation of signature grouping where signatures are grouped by their characteristics such as host-based attack signatures and network-based attack signatures.
6.5 Conclusion

There are a number of issues which make multi-step attack specification and detection in an environment with heterogeneous components a difficult problem. In particular, there is the need to correlate events from heterogeneous sources where there are no standard representations of events and timestamp of events may be affected by the unreliability of computer clocks. In addition, the complexity of specifying and detecting multi-step attacks adds a further layer of difficulty to the process.

This research has investigated these issues. Solutions to the absence of canonical event representations, complex multi-step attack detection mechanisms, and time uncertainty have been described. Methods to provide canonical representation of heterogeneous events as well as abstraction of such events have been presented. The representations are used in the multi-step attack detection engine which builds on an adapted unification algorithm. Clock skew and clock drift compensation techniques have been incorporated into the multi-step attack detection engine allowing the engine to detect attacks even in an environment where computer clocks are not synchronised.

Some of the proposed solutions require further refinement as described in this chapter. If their limitations can be overcome, they will enable effective and efficient specification and detection of multi-step attacks in an environment with heterogeneous components.
Appendix A

Data Source Schema and Abstract Event Model

A.1 Data Source Schema

The Data Source Schema (DSS) is a collection of sensor events. 13 sensor events have been defined. Figure A.1 shows operating system and (low-level) systemcalls sensor events. Figure A.2 shows sensor events of application-based (web server and intrusion detection system) and captured network traffic.

<table>
<thead>
<tr>
<th>unix_syslog</th>
<th>windows_event_log</th>
<th>linux_strace</th>
<th>snare</th>
<th>syscalltrack</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>windows_date</td>
<td>timestamp</td>
<td>timestamp</td>
<td>timestamp</td>
</tr>
<tr>
<td>host_name</td>
<td>windows_time</td>
<td>process_id</td>
<td>process_id</td>
<td>process_id</td>
</tr>
<tr>
<td>process_name</td>
<td>event_source</td>
<td>process_name</td>
<td>process_name</td>
<td>process_name</td>
</tr>
<tr>
<td>process_id</td>
<td>event_type</td>
<td>return_result</td>
<td>return_result</td>
<td>return_result</td>
</tr>
<tr>
<td>syslog_message</td>
<td>category</td>
<td>user_id</td>
<td>user_id</td>
<td>user_id</td>
</tr>
<tr>
<td>log_file_name</td>
<td>event_type_id</td>
<td>rule_id</td>
<td>rule_id</td>
<td>rule_id</td>
</tr>
<tr>
<td>log_notes</td>
<td>full_user_name</td>
<td>host_name</td>
<td>host_name</td>
<td>host_name</td>
</tr>
<tr>
<td></td>
<td>messages</td>
<td>messages</td>
<td>messages</td>
<td>messages</td>
</tr>
<tr>
<td></td>
<td>timestamp</td>
<td>timestamp</td>
<td>timestamp</td>
<td>timestamp</td>
</tr>
<tr>
<td></td>
<td>log_file_name</td>
<td>log_file_name</td>
<td>log_file_name</td>
<td>log_file_name</td>
</tr>
<tr>
<td></td>
<td>log_notes</td>
<td>log_notes</td>
<td>log_notes</td>
<td>log_notes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>solaris_syslog</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>message_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>facility</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>priority</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1: Operating system and systemcalls sensor events.
Figure A.2: Application and network sensor events.

A.2 Abstract Event Model

The Abstract Event Model (AEM) is a collection of derived events and abstract events. At the top of the AEM is the event node (abstract event) which represents a generic event. The level immediate below the event node (level 1) represents three broad abstract events of the AEM: operating system event, application event, and network protocol event. Figure A.3 shows the event node and abstract events in level 1.

Figure A.3: Level 1 abstract events.
A.2.1 Operating System Event Branch

The operating system event branch models operating system related events. The branch is divided into two branches: authentication related events (shown in Figure A.4) and operating system process related events (shown in Figure A.5 and Figure A.6).

Figure A.4: Authentication event branch.
Figure A.5: Process operation event branch.

Figure A.6: File operation event branch.
A.2.2 Application Event Branch

The application event branch models intrusion detection system alert events, error events recorded by a server, and operations recorded by a server. Figure A.7 shows application events of a web server and a DHCP server. Figure A.8 shows SSH server events and Snort alerts.

Figure A.7: Web and DHCP server event branch.

Figure A.8: SSH server event branch and Snort derived event.
A.2.3 Network Protocol Branch

The network protocol branch models events based on network protocols. Figure A.9 shows TCP, UDP, and HTTP network exchange events. Figure A.10 shows TFTP protocol events.

![Figure A.9: TCP, UDP, and HTTP network event branch.](image)

![Figure A.10: TFTP network protocol event branch.](image)
Appendix B

Signature Operator and SQL Operator Mapping

This appendix presents the mapping between operators in our Python-based signature language and their corresponding SQL clauses. The operator mapping shown in this appendix is used during the translation of a signature to an SQL statement.
### B.1 Identity Operators

The identity operator test variables or variable attributes for equality. Table B.1 shows the mapping between identity operators and corresponding SQL clauses.

<table>
<thead>
<tr>
<th>Operators</th>
<th>SQL Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>self.variable_name == Event(event_name)</code></td>
<td><code>SELECT variable_name.aet_eventid AS variable_name FROM event_name</code></td>
</tr>
<tr>
<td><code>self.variable_name.attribute_name == constant</code></td>
<td><code>WHERE (variable_name.attribute_name = constant)</code></td>
</tr>
<tr>
<td><code>self.variable_name1.attribute_name1 == self.variable_name2.attribute_name2</code></td>
<td><code>WHERE (variable_name1.attribute_name1</code> <code>= variable_name2.attribute_name2)</code></td>
</tr>
<tr>
<td><code>self.variable_name.attribute_name != constant</code></td>
<td><code>WHERE (variable_name.attribute_name</code> <code>!= constant)</code></td>
</tr>
<tr>
<td><code>self.variable_name1.attribute_name1 != self.variable_name2.attribute_name2</code></td>
<td><code>WHERE (variable_name1.attribute_name1</code> <code>!= variable_name2.attribute_name2)</code></td>
</tr>
</tbody>
</table>

Table B.1: Identity operators and their corresponding SQL clauses.
B.2 String Operators

The string operators test an attribute of a variable for sub-string and string size. Table B.2 shows the mapping between string operators and corresponding clauses in an SQL statement.

<table>
<thead>
<tr>
<th>Operators</th>
<th>SQL Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains_pattern(variable.attribute, pattern)</td>
<td>WHERE (variable.attribute ~ Epattern)</td>
</tr>
<tr>
<td>contains_pattern(variable.attribute, pattern, 'ignorecase')</td>
<td>WHERE (variable.attribute ~* Epattern)</td>
</tr>
<tr>
<td>not_contains_pattern(variable.attribute, pattern, flags)</td>
<td>WHERE (NOT (variable.attribute ~ Epattern))</td>
</tr>
<tr>
<td>length_greater_than(variable.attribute, size)</td>
<td>WHERE (char_length(variable.attribute_name) &gt; size)</td>
</tr>
<tr>
<td>length_less_than(variable.attribute, size)</td>
<td>WHERE (char_length(variable.attribute_name) &lt; size)</td>
</tr>
</tbody>
</table>

Table B.2: String operators and their corresponding SQL clauses.
B.3 Set Operators

The set operators test variable attributes for a given list of items based on the operators. Table B.3 shows the mapping between set operators and corresponding clauses in an SQL statement.

<table>
<thead>
<tr>
<th>Operators</th>
<th>SQL Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>one_of(variable.attribute, [item1, item2, ...]</td>
<td>WHERE ((variable.attribute = item1) OR (variable.attribute = item2) OR ...))</td>
</tr>
<tr>
<td>not_one_of(variable.attribute, [item1, item2, ...]</td>
<td>WHERE (NOT ((variable.attribute = item1) OR (variable.attribute = item2) OR ...))</td>
</tr>
<tr>
<td>one_of_patterns(variable.attribute, [pattern1, pattern2, ...])</td>
<td>WHERE ((variable.attribute ~ Epattern1) OR (variable.attribute ~ Epattern2) OR ...)</td>
</tr>
</tbody>
</table>

Table B.3: Set operators and their corresponding SQL clauses.
### B.4 Time Operators (No Timestamp Compensation)

Table B.4 shows the mapping between the time operators and their corresponding SQL clauses.

<table>
<thead>
<tr>
<th>Operators</th>
<th>SQL Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>before(var1, var2, timeout)</code></td>
<td><code>WHERE ((var1.event_time &lt; var2.event_time) AND ((var2.event_time - var1.event_time) &lt;= interval'timeout'))</code></td>
</tr>
<tr>
<td><code>before_between(var1, var2, tmin, tmax)</code></td>
<td><code>WHERE ((var1.event_time &lt; var2.event_time) AND ((var2.event_time - var1.event_time) =&gt; interval'tmin')) AND ((var2.event_time - var1.event_time) &lt;= interval'tmax'))</code></td>
</tr>
<tr>
<td><code>before_exact(var1, var2, time_interval)</code></td>
<td><code>WHERE ((var1.event_time &lt; var2.event_time) AND ((var2.event_time - var1.event_time) = interval'time_interval'))</code></td>
</tr>
<tr>
<td><code>after(var1, var2, timeout)</code></td>
<td><code>WHERE ((var1.event_time &gt; var2.event_time) AND ((var2.event_time - var1.event_time) &lt;= interval'timeout'))</code></td>
</tr>
<tr>
<td><code>after_between(var1, var2, tmin, tmax)</code></td>
<td><code>WHERE ((var1.event_time &gt; var2.event_time) AND ((var2.event_time - var1.event_time) =&gt; interval'tmin')) AND ((var2.event_time - var1.event_time) &lt;= interval'tmax'))</code></td>
</tr>
<tr>
<td><code>after_exact(var1, var2, time_interval)</code></td>
<td><code>WHERE ((var1.event_time &gt; var2.event_time) AND ((var2.event_time - var1.event_time) = interval'time_interval'))</code></td>
</tr>
</tbody>
</table>

Table B.4: Time operators (no compensation) and their corresponding SQL clauses.
B.5 Time Operators (Timestamp Compensation)

The following sections present the time operators that implement the constant skew compensation and linear regression techniques.

B.5.1 Constant Skew Compensation Time Operators

Table B.5 shows the time operators that implement the constant skew compensation technique. For readability, two variables \(t1_{\text{adjusted}}\) and \(t2_{\text{adjusted}}\) are used in the SQL clauses below to show adjusted timestamps using the constant skew compensation technique. Let \(N\) be either 1 or 2, these variables represent:

\[ (\text{var}_N\_\text{event\_time} + (\text{SELECT clock\_skew FROM constant\_skew\_technique WHERE hostid = var}_N\_\text{host\_id})) \]

<table>
<thead>
<tr>
<th>Operators</th>
<th>SQL Clauses</th>
</tr>
</thead>
</table>
| before\_constant\_skew(var1, var2, timeout)    | \[ \text{WHERE } (t1_{\text{adjusted}} < t2_{\text{adjusted}}) \]<br>\text{AND } ((t2_{\text{adjusted}} - t1_{\text{adjusted}}) <= interval\,'timeout') \]
| before\_between\_constant\_skew(var1, var2, tmin, tmax) | \[ \text{WHERE } ((t1_{\text{adjusted}} < t2_{\text{adjusted}}) \text{ AND } ((t2_{\text{adjusted}} - t1_{\text{adjusted}}) >= interval\,'tmin') \text{ AND } ((t2_{\text{adjusted}} - t1_{\text{adjusted}}) <= interval\,'tmax')) \]
| before\_exact\_constant\_skew(var1, var2, time\_interval) | \[ \text{WHERE } ((t1_{\text{adjusted}} < t2_{\text{adjusted}}) \text{ AND } ((t2_{\text{adjusted}} - t1_{\text{adjusted}}) = interval\,'time\_interval')) \]
| after\_constant\_skew(var1, var2, timeout)     | \[ \text{WHERE } ((t1_{\text{adjusted}} > t2_{\text{adjusted}}) \text{ AND } ((t1_{\text{adjusted}} - t2_{\text{adjusted}}) <= interval\,'timeout')) \]
| after\_between\_constant\_skew(var1, var2, tmin, tmax) | \[ \text{WHERE } ((t1_{\text{adjusted}} > t2_{\text{adjusted}}) \text{ AND } ((t1_{\text{adjusted}} - t2_{\text{adjusted}}) >= interval\,'tmin') \text{ AND } ((t1_{\text{adjusted}} - t2_{\text{adjusted}}) <= interval\,'tmax')) \]
| after\_exact\_constant\_skew(var1, var2, time\_interval) | \[ \text{WHERE } ((t1_{\text{adjusted}} > t2_{\text{adjusted}}) \text{ AND } ((t1_{\text{adjusted}} - t2_{\text{adjusted}}) = interval\,'time\_interval')) \]

Table B.5: Time operators using constant compensation technique and their corresponding SQL clauses.
## B.5.2 Linear Regression Time Operators

Table B.6 shows the time operators that implement the linear regression technique. For readability, two variables are used in the SQL clauses below to show adjusted timestamps using the linear regression technique: \( t_1\text{\_adjusted} \) and \( t_2\text{\_adjusted} \). Let \( N \) be either 1 or 2, these two variables represent:

\[
\text{(SELECT (TIMESTAMP 'epoch' + (EXTRACT(EPOCH FROM \( \text{varN\_event\_time} \) \)) + (EXTRACT(EPOCH FROM \( \text{varN\_event\_time} \) \)) + (SELECT slope FROM linear\_regression\_equation WHERE host\_id = \( \text{varN\_host\_id} \)) \)) + INTERVAL '1 second') + (SELECT y\_intercept FROM linear\_regression\_equation WHERE host\_id = \( \text{varN\_host\_id} \))
\]

<table>
<thead>
<tr>
<th>Operators</th>
<th>SQLClauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>before_linear_regression(var1, var2, timeout)</td>
<td>WHERE ((t_1\text{_adjusted} &lt; t_2\text{_adjusted}) ) \AND ((t_2\text{_adjusted} - t_1\text{_adjusted}) &lt;= interval'\text{timeout}'))</td>
</tr>
<tr>
<td>before_between_linear_regression(var1, var2, tmin, tmax)</td>
<td>WHERE ((t_1\text{_adjusted} &lt; t_2\text{_adjusted}) ) \AND ((t_2\text{_adjusted} - t_1\text{_adjusted}) \Rightarrow interval't\text{min}')) \AND ((t_2\text{_adjusted} - t_1\text{_adjusted}) &lt;= interval't\text{max}'))</td>
</tr>
<tr>
<td>before_exact_linear_regression(var1, var2, time_interval)</td>
<td>WHERE ((t_1\text{_adjusted} &lt; t_2\text{_adjusted}) ) \AND ((t_2\text{_adjusted} - t_1\text{_adjusted}) = interval't\text{interval}'))</td>
</tr>
<tr>
<td>after_linear_regression(var1, var2, timeout)</td>
<td>WHERE ((t_1\text{_adjusted} &gt; t_2\text{_adjusted}) ) \AND ((t_1\text{_adjusted} - t_2\text{_adjusted}) &lt;= interval'\text{timeout}'))</td>
</tr>
<tr>
<td>after_between_linear_regression(var1, var2, tmin, tmax)</td>
<td>WHERE ((t_1\text{_adjusted} &gt; t_2\text{_adjusted}) ) \AND ((t_1\text{_adjusted} - t_2\text{_adjusted}) \Rightarrow interval't\text{min}')) \AND ((t_1\text{_adjusted} - t_2\text{_adjusted}) &lt;= interval't\text{max}'))</td>
</tr>
<tr>
<td>after_exact_linear_regression(var1, var2, time_interval)</td>
<td>WHERE ((t_1\text{_adjusted} &gt; t_2\text{_adjusted}) ) \AND ((t_1\text{_adjusted} - t_2\text{_adjusted}) = interval't\text{interval}'))</td>
</tr>
</tbody>
</table>

Table B.6: Time operators using linear regression technique and their corresponding SQL clauses.
Appendix B. Signature Operator and SQL Operator Mapping
Appendix C

Details of the Datasets

C.1 The Synthetic Dataset

The synthetic dataset are collected from four hosts namely feisty, xapro, honey3, and forensic4. The type of operating systems and number of log entries (recorded events) collected from each host are shown in Table C.1. The synthetic dataset contains four types of logs: Apache access log, Microsoft IIS log, UNIX syslog, and Windows Event Log. The Apache access log entries are stored in the Apache combined log format. The IIS log entries are stored in the IIS native format. The UNIX syslog log entries are stored in the generic UNIX syslog format. The Windows Event Log entries are present as comma separated values (CSV). Each log entry contains multiple lines. For the ease of processing, the Windows Event Log are pre-processed where the new line characters are replaced by “#” and thus each log entry is present as a single line.

The dataset contains two types of attacks: input validation attack against PHP Bulletin Board (phpBB) and successful system administrator logins on two operating systems. The phpBB attacks were launched against feisty (Apache web server) and xapro (Microsoft IIS). The system administrator login were tested on honey3 (the Linux operating system) and forensic4 (Microsoft Windows XP).
<table>
<thead>
<tr>
<th>Host Name</th>
<th>OS</th>
<th>Log Types</th>
<th>No. Entries</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>feisty</td>
<td>Linux</td>
<td>Apache access log</td>
<td>232</td>
<td>8:31:49 hrs</td>
</tr>
<tr>
<td>xppro</td>
<td>Windows XP</td>
<td>IIS log</td>
<td>17</td>
<td>3 mins 33 secs</td>
</tr>
<tr>
<td>honey3</td>
<td>Linux</td>
<td>UNIX syslog</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>forensics4</td>
<td>Windows XP</td>
<td>Windows Event Log</td>
<td>26</td>
<td>2 mins 4 secs</td>
</tr>
</tbody>
</table>

Table C.1: Number of recorded events and logging duration of the synthetic dataset.

### C.2 The Scan of the Month Dataset

The SOTM 34 dataset comprises six types of logs: iptables firewall logs, Snort IDS alerts, Apache access logs, Apache error logs, Apache SSL error logs, and Linux syslog messages. The number of recorded events, log file types, and logging duration are shown in Table C.2. The iptables firewall logs were collected from Bridge. The iptables firewall log contains information regarding incoming and outgoing network traffic, e.g., traffic direction, IP addresses, port numbers, and packet flags. The Snort IDS alerts were collected from Bastion. The Apache access logs, error logs, and SSL logs, and Linux syslog were collected from Combo. The Apache access logs recorded HTTP requests processed by the server. The Apache web server on Combo was configured as virtual web servers. With virtual web servers, Combo served as multiple web servers on one physical host. The Apache error logs and Apache SSL error logs contain error messages generated by the Apache web server. The Linux syslog messages were collected from Combo. Linux syslog messages contain system, security, and application related logs. The messages were generated by the syslog facility on Combo. There are three types of syslog messages: generic messages, mail-related and security related.

Logs in the dataset were recorded in three formats: UNIX syslog, Apache combined log format, and Apache error log format. The UNIX syslog format is used by the UNIX syslog facility to records system and application related messages. Each syslog message comprises a timestamp, the host name of the syslog facility, the name of the process that sends the message to the syslog facility, and the message body. All parts, except the message body, are generated by the syslog facility when a message is received and have the same format for all records.
<table>
<thead>
<tr>
<th>Host Name</th>
<th>Log Types</th>
<th>No. Entries</th>
<th>Logging Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bridge</td>
<td>iptables firewall log</td>
<td>179,752</td>
<td>34 days 11:46:24 hrs</td>
</tr>
<tr>
<td>Bastion</td>
<td>Snort alert</td>
<td>69,039</td>
<td>34 days 11:28:05 hrs</td>
</tr>
<tr>
<td>Combo</td>
<td>Apache access log</td>
<td>3,554</td>
<td>46 days 7:03:28 hrs</td>
</tr>
<tr>
<td>Combo</td>
<td>Apache error log</td>
<td>3,692</td>
<td>46 days 7:05:09 hrs</td>
</tr>
<tr>
<td>Combo</td>
<td>Apache SSL error log</td>
<td>374</td>
<td>44 days 20:28:25 hrs</td>
</tr>
<tr>
<td>Combo</td>
<td>Generic system log</td>
<td>1,166</td>
<td>46 days 8:57:14 hrs</td>
</tr>
<tr>
<td>Combo</td>
<td>Mail log</td>
<td>1,172</td>
<td>45 days 24:55:06 hrs</td>
</tr>
<tr>
<td>Combo</td>
<td>Security-related log</td>
<td>1,587</td>
<td>45 days 6:42:09 hrs</td>
</tr>
</tbody>
</table>

Table C.2: Types of log, number of log entries, and logging duration of the SOTM 34 dataset.

However, there is no restriction on the message body formatting, message syntax, or the details the message body must contain. Thus, the contents and formatting of the message body depends on the application developers. In the SOTM 34 dataset, the UNIX syslog format was used for iptables firewall logs, Snort IDS alerts, and Linux syslog.

The Linux syslog on Combo consists of 21 files: maillog (seven files), messages (seven files), and secure (seven files). The maillog contains brief information regarding incoming and outgoing emails. The messages contains kernel messages, operating system messages, and generic application logs. The secure contains SSH user authentication events and xinetd-based services starting and stopping messages.

The Apache combined log format [102] is used by the Apache web server to record HTTP requests processed by the web server. Each record comprises the IP address of the client, user name, authenticated user name, timestamp, request string sent by the client, HTTP status response code, bytes sent by the server, HTTP referrer, and browser information. The Apache web server on Combo operated as multiple virtual servers. The virtual servers were configured to record access logs to the same file. A new log file was created every time Apache restarts.

The Apache error log format record error messages generated by Apache or by program run by Apache such as CGI program. This format is used by the Apache error logs and Apache SSL error logs.
Appendix D

Attack Signatures used in the Evaluation

D.1 Signatures for Attacks in the Synthetic Dataset

phpBB Attack

```python
from signature_lib import *
from aem_api import *
class phpbb_attack_def(Scenario):
    def __definition__(self):
        self.phpbb_attack_instance = Variable('phpbb_attack_instance', self)
        self.phpbb_attack_instance = Event(http_exchange)
        self.phpbb_attack_instance.http_status_code == '200'
        contains_pattern(self.phpbb_attack_instance.request_uri, '
            admin_smilies.*mode=edit.*select', 'ignorecase')
```

System Administrator Login into a System

```python
from signature_lib import *
from aem_api import *
class admin_login_def(Scenario):
    def __definition__(self):
        self.admin_auth = Variable('admin_auth', self)
        self.admin_auth = Event(authentication_event)
        one_of_patterns(self.admin_auth.authentication_result, ['
            accept', 'success'], 'ignorecase')
```
one_of_patterns(self.admin_auth.user_credentials, ['administrator', 'root'], 'ignorecase')

D.2 Signatures for Attacks in the SOTM 34

PV1: SSH login into a user account

```python
from signature_lib import *
from aem_api import *

class ssh_successful_login_def(Scenario):
    def __definition__(self):
        self.succeed_ssh_login = Variable('succeed_ssh_login', self)
        self.succeed_ssh_login == Event(ssh_authentication)
        contains_pattern(self.succeed_ssh_login.authentication_result, 'accept', 'ignorecase')
        contains_pattern(self.succeed_ssh_login.authentication_method, 'password', 'ignorecase')
        one_of_patterns(self.succeed_ssh_login.destination_address, ['11.11.79.\d+', 'combo'])
```

PV2: IRC events

```python
from signature_lib import *
from aem_api import *

class irc_activities_def(Scenario):
    def __definition__(self):
        self.irc_events = Variable('irc_events', self)
        self.irc_events == Event(snort_ids_alert)
        one_of(self.irc_events.destination_port, ['6667', '8888'])
        contains_pattern(self.irc_events.alert_message, 'IRC')
        contains_pattern(self.irc_events.alert_classification, 'corporate privacy violation', 'ignorecase')
```

PV3: Outbound HTTP connections

```python
from signature_lib import *
from aem_api import *

class outbound_http_connections_def(Scenario):
    def __definition__(self):
        self.outbound_http = Variable('outbound_http', self)
        self.outbound_http == Event(tcp_exchange)
        one_of_patterns(self.outbound_http.source_address, ['11.11.79.\d+', 'combo'])
        self.outbound_http.destination_port == '80'
```
SV: Attacking a Vulnerability in AWStats

```python
from signature_lib import *
from aem_api import *
class awstat_attack_def(Scenario):
    def __definition__(self):
        self.awstat_attack_instance = Variable('awstat_attack_instance', self)
        self.awstat_attack_instance = Event(http_exchange)
        self.awstat_attack_instance.http_status_code = '200'
        one_of_patterns(self.awstat_attack_instance.request_uri, ['/cgi-bin/awstats.pl\?\?\?configdir=.*%7c.*', '/cgi-bin/awstats.pl\?\?\?configdir=.*\||\*'])
```

MSS1: Exploit AWStats, Download and Run an IRC bot

```python
from signature_lib import *
from aem_api import *
from awstat_attack import *
from outbound_http_connections import *
from irc_activities import *
class awstat_wget_irc_def(Scenario):
    def __definition__(self):
        self.step1 = awstat_attack_def('step1')
        self.step2 = outbound_http_connections_def('step2')
        after(self.step2.outbound_http, self.step1.
              awstat_attack_instance, '00:01:00')
        self.step3 = irc_activities_def('step3')
        self.step3.irc_events.source_address = self.step2.
              outbound_http.source_address
        after(self.step3.irc_events, self.step2.outbound_http,
              '00:00:20')
```

MSS2: Exploit AWStats, Download and Run a Backdoor Program

```python
from signature_lib import *
from aem_api import *
from awstat_attack import *
from outbound_http_connections import *
from rootshell_backdoor import *
class awstat_downloadBD_connectBD_def(Scenario):
    def __definition__(self):
        self.step1 = awstat_attack_def('step1')
        self.step2 = outbound_http_connections_def('step2')
```
after(self.step2.outbound_http, self.step1.
    avstat_attack_instance, '00:01:00')
self.step3 = rootshell_backdoor_def('step3')
after(self.step3.backdoor, self.step2.outbound_http, '00:10:00
')
self.step3.backdoor.destination_address == self.step2.
outbound_http.source_address

MSS3: Exploit AWStats, Download a Backdoor Program,
Connect to the Backdoor, and Make an Outbound HTTP
Connection
from signature_lib import *
from aem_api import *
from avstat_attack import *
from outbound_http_connections import *
from rootshell_backdoor import *
class awstat_BD_outbound_http_def(Scenario):
def __definition__(self):
    self.step1 = awstat_attack_def('step1')
    self.step2 = outbound_http_connections_def('step2')
after(self.step2.outbound_http, self.step1.
    avstat_attack_instance, '00:01:00')
self.step3 = rootshell_backdoor_def('step3')
after_between(self.step3.backdoor, self.step2.outbound_http,
    '1 day', '2 days')
one_of_patterns(self.step3.backdoor.destination_address,
    ['11\11\11\11\79\8d+', 'combo'])
self.step4 = outbound_http_connections_def('step4')
after(self.step4.outbound_http, self.step3.backdoor, '00:01:00
')

MSS4: Connect to the Backdoor and Make an Outbound
SSH connection
from signature_lib import *
from aem_api import *
from rootshell_backdoor import *
from outbound_ssh_connections import *
class connectBD_then_ssh_def(Scenario):
def __definition__(self):
    self.step1 = rootshell_backdoor_def('step1')
    self.step2 = outbound_ssh_connections_def('step2')
```python
self.step2.outbound_ssh.source_address == self.step1.backdoor.destination_address
after(self.step2.outbound_ssh, self.step1.backdoor, '00:10:00')
```
Appendix D. Attack Signatures used in the Evaluation
Bibliography


the 16th Annual Working Conference on Information Security (IFIP TC11),