MSc Software Engineering - University of Amsterdam - SURFsara

Stateful discovery of attack manifestations on networks and systems

Christos K. Tsigkanos

Master's Thesis

supervisors: Dr. Jurgen Vinju (CWI), Sedat Çapkin (SURFsara)

August 13, 2013

Detection of attack manifestations on networks and systems via correlation of log information, is crucial for security. Specification-based solutions have been proposed as a promising paradigm that counters the high variability of attacks.

In this study, a solution model using a form of Extended Finite State Automata is proposed, which is argued to be very effective as a fundamental and precise primitive in sequence analysis of log events. Attack behaviour specifications, are translated to runtime operations which show the viability of the approach.

After an initial evaluation, these augmented state machines accompanied with a proof-of-concept Domain Specific Language, offer a solid foundation for accurate conceptualization and classification of attack vectors; further applications with attack specification languages appear promising.

Contents

List of Figures

List of Tables

Listings

Acknowledgements I would like to express my deepest appreciation and gratitude to Dr. Jurgen Vinju for the guidance and engagement through the research process of this thesis; his scientific intuition exceptionally inspired and enriched my growth as a student and a researcher.

Furthermore I would like to thank Sedat Çapkin for his advanced knowledge and insights to the topic as well as for valuable suggestions along the way. My experience at SURFsara was indeed invaluable because it gave me the unique opportunity to practice ideas and disciplines learned through the academic environment, applied in a efficient industrial setting.

I am also indebted to my professors at the University of Amsterdam and CWI, among the top academic and research centers of the world, for providing me concrete foundations and food for thought via their high-level courses, discussions and scientific interactions on the epitome of Software Engineering.

My parents, Kanaris and Georgia, deserve special mention for their unconditional support and love. Without them this would not be possible.

To my parents

1. Introduction & Scope

This document describes the thesis project that was carried out by Christos Tsigkanos, under the supervision of Dr. Jurgen Vinju from Centrum Wiskunde and Informatica (CWI), and Sedat Çapkin from SURFsara B.V. The project entails intrusion detection by correlation from a multitude of log information in SURFsara operational networks, from the point-of-view of information security.

1.1. Document Structure

The document starts with outlining the context and description of the problem, a formulation of the underlying research questions and reasoning behind them. After a study of relevant scientific literature, what follows is an outline of the phases of the research approach used, namely Domain Analysis/Specification, Classification, Architecture/Design and subsequently Evaluation.

System Requirements, both functional and non-functional, grow throughout the Domain Analysis/Specification section, and a study of the proposed theoretical component follows at the Classification chapter. The rationale behind each design decision is intended to capture and convey the significant choices and solution patterns which have been made regarding the system. Then, a comprehensive overview of the proposed software solution is presented, using a number of different architectural views to depict different aspects of the system. This study concludes with the presentation of cases where the prototype system has been used, along with relevant analysis and discussion.

1.2. Problem Description and Context

SURFsara manages for science and research purposes a number of High Performance Computing systems. These systems have an open accessible character. While ensuring the safety of scientific research data on these systems must be guaranteed. Information security has a high priority within SURFsara; to ensure the safety of information and other valuable assets, systems are constantly monitored.

Intrusion detection is the process of monitoring the events occurring in a computer system or network and analyzing them for signs of possible incidents, which are violations or circumventions of computer security policies, user policies, or malicious attacks. Such incidents have many causes, such as malware as network worms or viruses, attackers gaining unauthorized access to systems from the Internet, or legitimate users who misuse their access or attempt to gain additional privileges for which they are not authorized.

For the purpose of intrusion prevention and detection, large amounts of log information are generated daily from the networks and systems. The security

group of SURFsara, as the major stakeholder, envisions a security system that identifies anomalies and misuses from correlations between different types of information, so attacks or security incidents are detected in time.

Such a system, is usually a software application that monitors network and system activities for malicious activities or policy violations, and produces reports to security officers. Its primary goal, is to identify possible incidents, log information about them, and report back analytics on attempts to circumvent security facilities in place.

Several solution schemes are proposed in scientific literature for the type of problem of intrusion detection such systems solve; these include machine learning or data mining techniques, statistical methods or expert systems, neural networks, or a combination of these. Another novel approach, is based on modeling through specification languages.

1.3. Detection Techniques

With the ever-growing attacks on network infrastructures, the need for techniques and advanced tools to detect, study and prevent attacks is increasing. The notion of intrusion detection, refers to a wide range of literature and techniques that counter malicious attacks [1]. Techniques generally are categorized as either misuse detection or anomaly detection.

In misuse detection approaches, abnormal system behaviour is defined; this stands against anomaly detection approaches, which utilize definitions of normal behaviours, aiming to classify outliers as abnormal. The main advantage of misuse detection is that it can very accurately detect known attacks, while its main drawback is the inability to detect previously unseen attacks. Anomaly detection, on the other hand, is capable of detecting novel attacks, but suffers from a very high false positive rate. This is because previously unseen, yet legitimate behaviours are regarded as anomalies.

Specification-based solutions have been proposed as a promising alternative that combine the strongest points of misuse and anomaly detection [28]. The main theme in these approaches, is that expert-defined specifications are used to characterize legitimate, or illegitimate program behaviors. This way deviations or instances of either, are very accurately detected, and thus the false positive rate, the major hurdle with anomaly detection, is reduced significantly.

The modeling of the problem domain lies in the ability of the security experts, who are constantly aware of the nature of potentially interesting situations. Enabling experts to actively define and detect intrusion behaviours pertinent to their specific use case, is a powerful concept: it is an intrusion detection system coupled with and supported by domain knowledge.

Interfacing between the domain expertise of security experts and the amount and variability of relevant data is a challenge where the concept of using a Domain Specific Language fits right in. Via this mechanism descriptions of

attack manifestations can be generated [4]; these are to be constantly checked against data available, and in turn provide back relevant insights.

This way, the definition of an anomaly or misuse is delegated to the security expert who uses the Domain Specific Language: the DSL should provide the underlying mechanisms and language constructs to support this. The DSL, as a mediator through the domain and the user, should outline all possible ways an interesting case could be defined. Thus, what does an outlier "looks like" is defined on a precise basis, depending on the specific use case. A formulation of research questions based on this follows.

1.4. Research Questions

Can expert knowledge about attack manifestations be captured in formal and executable specifications?

- What are the difficulties associated with information security oriented specification?
- How can specifications of attack manifestations can be efficiently translated to operations on data?

2. Related Work

Attack languages have been used for security requirements specification purposes and attempt to bridge the gap between software engineering and security engineering. The common objective, is that manually specified behavioral specifications are to be used as a basis to detect attacks, novel or otherwise, and several of these security specification-based languages have been proposed, i.e. AsmL, STATL, ASL and USTAT [10].

STATL [17] is an attack language that can describe different attack scenarios, where an attack is modeled as a sequence of steps that bring the system from an initial state to a compromised state. The Behavior Modelling Specification Language (BMSL), is a language designed for developing security-relevant behavior models at the system call level [16]. Specifications in BMSL consist of rules of the form of a tuple (pattern, action), where pattern can be an event sequence and action the the response to it. It can be used for both normal and abnormal behavior specification, by using negations of properties of normal or abnormal event histories. Validation in such language approaches is promising, and false alarm rates are considered generally low and quite comparable to misuse detection. Another result of note, is that generic specifications seem to be sufficient for detecting a majority of the attacks [16].

Other approaches such as the extensive specification theme apparent in the Audit Specification Language (ASL), include language support that allows specification authors to describe complex data structures [19]. This way, an

attack scenario can be generic enough and be able to handle different low-level formats (e.g., switching from IPV4 to IPV6 packet inspection). In [34], a pattern language and intrusion detection system is introduced, which uses a novel algorithm for correlating distributed event signatures.

Another matter, is that the underlying data required for security oriented specification, have to exhibit certain properties. In [2], an extensive study is done on the context of requirements of attack manifestations, however directed on different parts of the network stack. The framework proposed for log data extraction, proves valuable for its classification of attack manifestations with respect to data available. Case studies of attacks and their different data extraction requirements provide the foundation in finding classes of attacks with similar demands on log data [15].

A common denominator in many security specification languages, is a state transition concept in specifying behaviors. Using this, thorough models are built which represent a variety of situations and interactions that can happen in the relevant domain. Using stateful analysis, descriptions of profiles of behaviour which model activity in the domain are created and analyzed. The rationale behind the addition of state, is the fundamental notion that attacks happen as a sequence of steps, and thus should be analysed as such. This stateful sequence-based detection as relevant to networks security is the general fundamental theme for study [6, 7]. The trend of stateful detection is evident, as there exist also frameworks to translate between state-based languages, as indicated for AsmL and STATL in [18].

The primary hindrance of stateful security analysis methods is that they are very resource-intensive because of the complexity of the analysis and the overhead involved in performing state tracking for many simultaneous sessions and cases. Another serious problem is that stateful analysis techniques cannot detect attacks that violate general characteristics of the previously analyzed and understood behavior, such as novel denial of service attacks. Yet another disadvantage is that the implementation of the model used might conflict with the way the underlying events are represented across different systems, thus introducing development overhead in real-world scenarios.

However, the consideration for the state-transition concept is evident in scientific literature as an excellent match for modeling attacks that exhibit high complexity. It imposes a convienient structure for study and specification, and offers a solid foundation for runtime operations.

Lessons learned from the relevant literature concern several components of an attack detection system. As an initial step, characteristics of the event entities for attack manifestation specification must be defined, following the course in log data extraction such as in [2, 5]. Relations between these fields will provide insight for selecting the right set of entity features for classification.

At the core, regarding classification through sequence analysis, techniques help understand what properties time series of events in attack vectors should have, along with the theoretical foundation [28]. The specification approach followed in extensive frameworks and languages in intrusion detection, are valuable to enable complex attack description support [19]. Regarding the formal modeling of attack vectors through correlation of event streams there exists also research in [11].

Regarding evaluation of the proposed system, the use of case studies is the definitive way used by researchers in each context, using attack cases pertinent to each framework or system. The stateful analysis in languages and frameworks mentioned, i.e. in AsmL, STATL, ASL and USTAT is oriented in a wide part of the system-network stack, ranging from the system kernel level, to the network packet level, so use cases must be found which represent the right domain.

In the current study, an attempt is made to apply the lessons learned from similar approaches in literature, but utilizing both *system data* and *network meta-data*.

3. Research method

The project revolves around four main themes, which are described below. The interplay between them is key to the approach.

Domain Analysis and Specification
This research component aims in the description of the project goals, through a study of matters relevant to the domain. These include a study about the underlying data entities and their requirements, computational tradeoffs and specific domain constraints. In the form of design considerations, the discussion is directed towards the definition of the system's functional and non-functional requirements. Extrapolating on these, major concerns are how one can understand, accurately specify [3] and detect security-oriented events based on the current context, which is the subject of the next section.

Sequence Analysis and Classification
Strongly coupled with validation of the research implementation, this component entails observing occurrences of combinations of security oriented events in reality and building a model, classifying characteristics and enabling detection. Through classification, common denominators can be found in possible outliers, and the main analysis mechanism is developed.

Prototype Architecture and Design
Since possible answers to the *Research Questions* entail a pragmatic attempt at definition, a relevant prototype system must be designed, which addresses the analysis concerns, and based on the requirements set in the Domain Analysis section. This also includes the system architecture, after study of relevant literature. Design and implementation with the *Research Questions* in mind, is the shortest path to evaluation of the model and system.

Evaluation
After a formulation of appropriate key points, evaluation of the design, analysis mechanism and prototype system is attempted using a variety of case studies. In this section, typical domain problems are discussed, along with solutions using the proposed approach. Possible improvements over the current study, lessons learned and threats to validity are also included.

4. Domain Analysis and Specification

The goal of this section is to build system requirements throughout a study and discussion of underlying matters relevant to the current domain. The context diagram in Figure 1 identifies the entities interacting with the system, as well as the flow of data. In context, the operation is as follows: there is an influx of log events from various sources, which the system processes and stores in an external facility. Then, the security expert introduces attack specifications to the system, which use the information in the data store to produce result reports back to the operator.

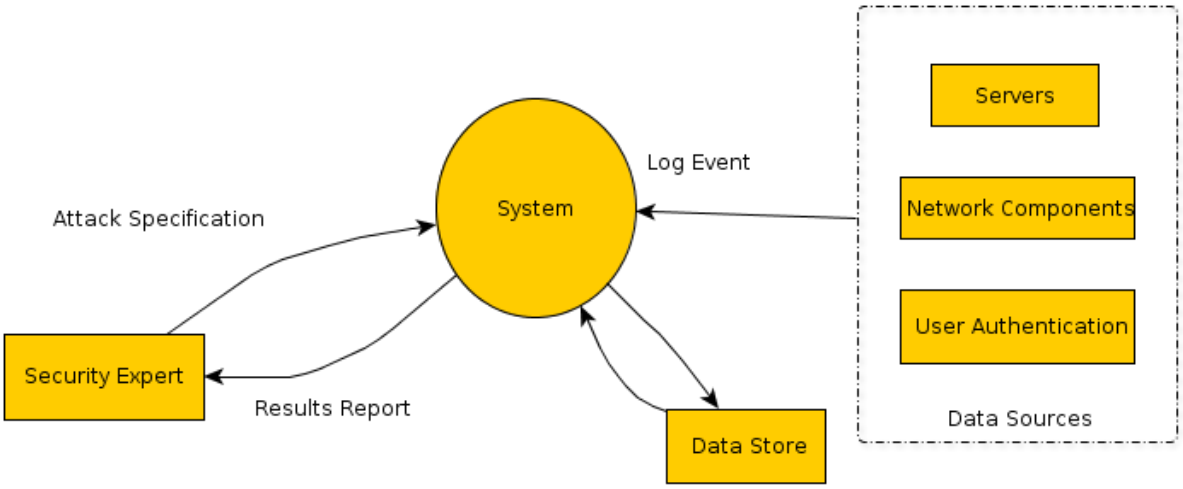

Figure 1: System Context Diagram

4.1. Data Format and Sources

The system aims at analysis of information contained in log files which are collected from a variety of systems and network components. These source systems, may use different log formats, and different components use different syntax for logging information. Thus the need for a common reference specification arises; a common denominator is necessary in order to abstract operations on the data. An accurate notion of 'log entity', allows for definitive specification whereas also possibly addressing data storage and operational concerns, which require uniform structures [2].

Moreover, an interesting observation is that every line in an arbitrary input log file is self-contained; all information about an event can be found there. Conversely, meta-data about the log file attributes or filesystem location are not important. This assumption a) does not enforce strict raw input constraints at the log event importing stage, such as relying on when or where

the log file came from, and b) has clear advantages over data storage distribution and parallelization of operations, as each line in a file can be parsed individually.

Available at the time of this study were mainly logs about authentication/syslog and netflow. However, generic syslog and kernel logging parsers are also included for future extension. Test data for the prototype corresponded to one month of systems and network usage, which was about 220 million events (or 180GB). An overview of the two primary log categories follows.

4.1.1. System/Authentication Data and Taxonomy

Syslog is the standard facility for data logging in UNIX systems. It presents an unified interface to various programs, gathering the relevant log information along with certain metadata, making it possible to store them centrally. Implementations are available for a variety of operating systems. In the current problem's case, log information can be from facilities which include authentication services, system facilities such as firewalls and applications such as web or mail servers.

Focus is given on the use and analysis of mainly *authentication* data, as pointed out by the stakeholders. Information related to authentication on SURFsara systems come from a variety of sources and relevant components, which are handling a large number of users and sessions at a time.

Syslog-related entities, sharing common characteristics, are parsed and the relevant fields are populated, if applicable. A detailed description of the eventual parsed fields can be found on Table 4.1.

Requirement 1: System and NetFlow log analysis
Requirement 2: Common log entity specification
Requirement 3: Flexibility in input mechanisms

Event Field	Event Description
action	An action associated with the entity, if applicable e.g. 'open', 'accept', 'close', 'fail'
component	The component which fired the event, if available, e.g. 'pam_access', 'pam_unix', 'pam_succeed_if', 'pam_keyinit'
date	Date and time in ISO 8601, up to a standard second resolution
pid	The PID of the unix process that generated the log event.
program	The program that generated the log event; versions of PAM, various authentication or login facilities, or system components such as firewalls, e.g. 'login', 'rpcbind', 'sshd-external', 'su', 'syslog-ng'
source	The source host of the log event, e.g. 'iokasti'
taxonomy	A general taxonomy of the syslog entries, e.g. 'access control'
type	Characterization of log event, if available, e.g. 'auth', 'session', 'account'
user	The user involved in the log event, if applicable, e.g. 'root'
source ip	External address related to the event, if applicable, for example in authentication attempts
body	The body of the log event

Table 4.1: Fields of syslog event entity

4.1.2. NetFlow Data and Taxonomy

NetFlow is a network protocol developed by Cisco Systems for collecting IP traffic information. It has become a wide industry standard for traffic monitoring and is supported on various routers, enabling collection of IP traffic meta-data: the network router, while conducting normal operations, gathers statistics about the current routed socket pairs.

The Flow abstraction
A "flow", is defined as the set of characteristics of a unidirectional sequence of IP packets between two networked endpoints. It is defined by some key fields: source and destination address and port, protocol type, type of service, as well as various routing information. The unidirectionality property in practice, means that for a TCP communication session two complementary flows will be observed, while for UDP only one.

The termination conditions of a flow are of note:

- Inactivity in transmission exceeds 15 seconds.
- The transport layer indicates that the connection is completed, and in case of TCP, FIN flags are observed. In this case the full acknowledgment handshaking is also included.
- For flows that remain continuously active, flow cache entries expire in a maximum of 30 minutes and new flow records are created.

The key concept of NetFlow, is that only meta-data and no paylod information about the connections are captured. This is an important difference from signature-based tools that do packet analysis. The NetFlow approach however, has some distinct advantages.

The notion of not considering packet payloads, greatly reduces the processing demands and makes the NetFlow protocol an excellent fit for busy, high-speed network environments. In addition, this abstraction characteristic makes it very useful in zero-day or novel attack detection, a use case where traditional signature-based methods would fail due to the high influx of data.

Since flow data is coming directly from the lower network level, NetFlow is capable of providing a unique perspective of the entire traffic profiles of a network at the core infrastructure level. Another advantage, is that higher data retention is possible, in contrast to traditional packet analysis setups. This allows for extensive later analysis and observation of network events. The flows in a typical setup are exported in near real-time via UDP to a collector host, which compresses and stores them.

NetFlow records are parsed as per the information contained. Note that on every flow record, all information is always present. Table 4.2 has a detailed description of the parsed fields available for analysis.

Event Field	**Event Description**
tos	Source type of service, e.g datagram's priority, low-delay, high-throughput, or highly-reliable (Layer 3)
dstas	Identifier of destination Autonomous System, used for BGP routing
dstip	Destination IP address of the flow (Layer 3)
srcip	Source IP address of the flow (Layer 3)
input	Input interface index used by the SNMP protocol
dstport	Destination port of the flow
srcport	Source port of the flow
first	Flow start timestamp, normalized to ISO 8601
last	Flow end timestamp, normalized to ISO 8601
flags	TCP flags of the flow, ORed, e.g. 000010 (SYN) + 000100 (RST) = 6 (FLAG)
srcas	Identifier of source Autonomous System, used for BGP routing
prot	Protocol of the flow packets (Layer 3), e.g. 'tcp', 'icmp'
packets	Number of packets observed in the flow
bytes	Number of bytes observed in the flow
output	Output router interface index, or zero if the packet was dropped.

Table 4.2: Fields of netflow event entity

4.2. Data Store

The storage of the log entities specified, is highly relevant to the design of the prototype system. It is evident from the nature of the data that log entities are self contained and loosely coupled, showing no strong relationships between them, as every log line is transformed to a log entity object populated with the available fields. This object-like structure is relevant to an evaluation of data storage options for the implementation stage.

A key factor is that initial data size is relatively unknown, since the operator can choose to import any amount of log data for analysis. So given this loose constraint requirement on the input data, strong assumptions are not effective: operations need to be able to scale as possible in a uniform fashion.

As for system operation, it is apparent that the large dataset remains unchanged throughout operations, so write-related actions are not much of a concern. Instead, the data storage solution must display high bulk insert performance, since the initial insertion of data is a demanding process. As for data access and aggregation operations, the use case at hand displays minimal data ordering needs; data are ordered in a time-series and the need for extensive indexing at the database layer is minimized.

Another quality that would prove beneficial by enabling rapid prototyping at the implementation phase, is direct mapping between objects in the database and programming constructs.

4.3. Data Quality

In the current context, due to the sheer amount of information, as well as the lack of fine-grained control over the initial raw input data, the issue of data quality arises even at the initial input stage. To overcome this, measures must be taken towards *data cleansing*; operations on the dataset highly depend on the expected data distribution. The expert might decide to enter data of a specific type or quantity, to test a specific scenario hypothesis, and so should be allowed to inspect the data entered for integrity. This theoretically translates to setting qualitative and quantitative standards on data present in the system: these can be estimated through learning techniques [28], the expert user can choose them through an interactive process or to ensure standards, can be hardcoded.

Another related matter also concerning baselines, is estimating thresholds. The problem of accurate threshold definition is a recurring theme in security analysis; the ability to distinguish normal from abnormal or malicious from legitimate, is highly sought. Thresholds highly depend on their context,

Requirement 4: Maintain limited dependance on input data size
Requirement 5: Realize high insert performance in data input

making statistical measures over the data set hard to define; the overall goal being to develop ways of determining baselines that depend on a relevant data subset. An event analysis mechanism that would provide auxiliary functions supporting this would be very beneficial. More on this subject can be found in Appendix A.

In the current problem's case however, the focus is given on data quality. The system must maintain flexibility in operation and be able to work with non uniform data distributions, since conceptually arbitrary data is coupled to the scenarios that the domain expert defines for analysis; in practice, the simplest case of this translates to aiding the expert to verify the dataset.

4.4. Online, Offline and tradeoffs in analysis

Another critical consideration, is the way the analysis is performed; this potentially affects also the core sequence analysis mechanism. Online or offline modes can be vastly different regarding implementation.

The stakeholders requested offline analysis, but with the option to expand the underlying core runtime engine in the future to support also online analysis. The reason online analysis is not initially addressed, is that as with every critical security matter, there are deployment considerations. Software that imposes changes in multiple points of critical infrastructure, such as core network routers or servers, must be thoroughly tested, security checked and augmented with resilience mechanisms before reaching production.

Another consideration, is that the amount of offline data, processing resources and capacity for analysis can not be predicted beforehand, but fluctuate depending on the current use case, environment and timespan the log entities represent and assumptions on the density of the data set can not be made.

These goals point to a solution pattern based on the notion of processing an event stream, as such an organization is compatible with both perspectives. The base principle, is the existence of an event feed, which whether in online or offline mode, is directed to scenario checking components. These statements represent requirements that constrain the solution space, and must be taken into account when evaluating possible solution schemes for the runtime analysis engine.

Requirement 6: Architecture must support online analysis

4.5. Challenges and Driving Quality Attributes

Several challenges can be identified in the solution space of the problem. These regard both the system as well as the accompanying specification language.

- Efficient computation; operations on the dataset available must be as quick as possible, and domain specific optimizations should allow for rapid processing,
- Runtime library interfacing; translating the attack scenarios to runtime computations, to account for the variation of the problem space is also a challenge.
- Data Storage; the choice of database greatly affects performance, and possible solutions must be evaluated.
- Language syntax; the new language must be accurately specified, whereas maintaining usability characteristics. The specification process must be aided by graphical reporting capabilities,

Additionally, the specification language should ultimately fullfil these goals:

- Simplicity; the language should provide just those features needed to represent attack scenarios, i.e. to not deviate from the domain.
- Expressiveness; the language should have a rigorously defined, implementation-independent syntax and semantics, so that the meaning of any attack scenario is unambiguous.
- Extensibility; it should be possible to extend the language in a well-defined, relatively simple way.

In the following sections, an attempt to address these points is made, regarding architecture, design and implementation of the prototype system.

5. Sequence Analysis and Classification

In this section, the foundational theoretical model and base mechanism for sequence analysis is presented, accompanied with a study of the implications of its use.

5.1. Attack Vectors and Sequence Detection

Misuse detection analysis in literature, can be characterised as stateless or stateful. Stateless analysis concerns the investigation of each event in the input stream in an independent fashion, while stateful analysis considers the relationships between events and is able to detect event sequencies that represent attack vectors. This analysis is considered more powerful and allows one to detect more complex attacks. However, drawbacks of stateful approaches include high CPU and memory costs and possibly a vast amount of complexity. An event stream assosiated with such stateful approaches, can be any continuous flow of information coming from system auditing facilities, application logs, or the various network appliances.

Taking into account the problem setting, it is observed that an attack vector can be broken up into a series of discrete event characteristics; actions on the network or a system, happen within the time domain and other constraints:

- In the case of the network level (netflow), events happen between two endpoints, i.e. a client and a server and are ordered on a time basis.
- In the case of the system level (syslog), available information are bound to the originating host or possibly other information, such as the initiating user, but always the time.

This corresponds to the definition of a security related attack vector: a malicious attacker, worm or virus, will take a series of steps, representing the path to a computer or network server in order to deliver a payload or malicious outcome [14]. This time series of steps is definitely represented distinctively in the underlying data set, since quantification is available at even the network flow level. It is evident that effective modelling of these discrete stages is the precursor of detection.

Thus an detection mechanism in this context must support efficient detection of a well defined, but sufficient abstractly described sequence of event characteristics. While the former concerns design and implementation, the latter is highly dependent in allowing accurate specification by enabling the domain expert. At the core of the problem, a solution can be reduced into effectively distinguishing malicious sequences of events from benevolent ones.

The approach, has to satisfy the following requirements:

- Flexible specification of the event characteristics for detection
- Extensibility in the sense of enabling detection of combinations of sequencies

- Clear syntax and easy conceptual understanding of the modeling constructs
- Compatibility with runtime concerns

Note that traditional database query mechanisms and languages are not fully applicable for this type of operations, as they do not satify these requirements. Moreover, in this case a) language constructs expressing sequential operations become quickly complex and cumbersome and b) there is an inherent incompatibility with an event-feed runtime architecture, which is highly sought in the current solution space.

5.2. Solution Candidates

Based on the assumptions that sequence detection includes and the requirements of the prototype system, possible solutions addressing the problem are evaluated.

Event processing is used in operational intelligence solutions to provide insight into business operations by running query analysis against live feeds and event data [20], and is heavily used in financial applications, where time-ordered data are frequent. The goal is to combine data from multiple sources, to infer analytics or patterns that identify meaningful opportunities or threats and respond to them as quickly as possible.

Implementations that address event processing include stream processing languages such as StreamSQL [21], Esper [22] or domain-specific integrations in products as in TIBCO StreamBase [25]. These approaches use time windows to process event streams and extend the SQL type system to allow manipulation such as stream relation and stream join operations. Others such as OpenPDC [23] or Kinetic Rule Language [24], aim at advanced rule-based action definition regarding event streams. Time-series databases also exist as a backend to support performant operations on time-ordered existing data. Architectures concerning event system processing are extensive, and typically concern large scale systems used in process monitoring or financial applications. Proprietary solutions dominate the field.

However, the fragmentation over different implementations regarding event processing is apparent. Each approach uses different semantics, offers different features and is targeted to specific event processing applications of mostly enteprise level. The absence of a unified framework and model makes solutions not compatible with one another, being tied up to specific languages, systems and architectures.

Requirement 7: Effective attack modeling
Requirement 8: Allow combination of scenario sequencies
Requirement 9: Provide facilities to aid understanding of scenario specification

Another solution candidate is Finite State Automata [27], a mathematical model of computation conceived as an abstract machine composed of states and transitions. The conceptual simplicity of a solution based on a form of Finite State Automata is argued that is an asset that extends to many advantages. An automaton is a primitive construct, which very naturally corresponds to the processing of discrete events. It exhibits high extensibility, as auxiliary operations and internal elements can be included in the model implementation seamlessly after deployment. Software design patterns also exist to enforce best practices on complex designs [26]. Another point is that this model is a highly transferable abstraction, allowing interoperability between implementations. Regarding runtime operations, the finite state automata model is compatible with every implementation language and setting, ranging from hardware implementations in embedded systems to high performance software systems. It has no dependancies on libraries or external systems. Distinction on the mode of operation between offline and online analysis mechanisms is not a concern, as the exact same model and design can be applied. Moreover, the memory footprint is minimal and the computation cost is equivalent to if-clauses. Also, in scientific literature on stateful analysis for security through specification languages, the use of various forms of finite state automata as a base to model attacks is common [6, 7, 18].

In the current problem setting, it fully fulfills the concerns previously set. In the next section, a solution using finite state automata is presented.

5.3. Extended Finite State Automata

Extended Finite State Automata (EFSA) are an efficient construct for modelling attack specifications. An EFSA, as outlined in [28], is similar to a Finite State Automaton, with some key augmentations revolving around a more complex set of transition trigger conditions, state variables and constructs to support higher level operations.

The model, should be able to be translated to operations on the dataset. Regarding the runtime behaviour characteristics, event streams representing sequences of attack vectors should traverse an automaton, beginning from the initial state. At each intermediate state, the state conditions are checked against the event parameters and possible various state variables present, depending on the specific operation. If the conditions are satisfied, a transition should be fired. Should the EFSA reach a final state, it is successful.

The base FSA model, does not allow the presence of variables in states. In the current domain such variables are needed to complement more advanced state conditions, and manipulation of them according to the transition logic enables computation dependent on the state's internal environment. Logical or arithmetic functional primitives present on the state which have the role of conditions, can manipulate these state variables and trigger transitions depending on the event stream.

In context, the model proposed is based on the assumptions that (a) an EFSA makes transitions that may have arguments, (b) it can use a finite set of state variables in which values can be stored, and (c) accepting states can have outward transitions. If trigger conditions are satisfied with respect to the present context, a transition is fired, bringing the automaton from the current to the next state, while potentially performing auxiliary operations.

Formally, an EFSA is defined to be a tuple $(\Sigma, Q, s, F, V, M, D, \delta)$, where:

- Σ is the alphabet of the EFSA. It is an alphabet of possible events,
- Q is a finite set of states of the EFSA,
- $s \in Q$ is the initial state of the EFSA,
- $F \subseteq Q$ is a set of accepting states, where for $f \in F$, f can have outward transitions,
- V is a finite tuple $(u_1, ..., u_n)$ of state conditions,
- M is a finite tuple $(m_1, ..., m_n)$ of state variables,
- $\delta : Q \times V \times \Sigma \times M \to Q \times M \times V$ is the transition relation.

Using this definition, a solution for specifying EFSA that model attack vectors is proposed. The reflection of this model to programming constructs can be found in Table 5.1.

In practice, the definition entails creating states Q, where each one is comprised of conditions V and state variables M. This combination of V and M, represents accurately the events that can incur transitions to this state. Beginning from a specific state designated as initial, control may flow through the automaton up to a state marked as accepting. A series of events that led from s to an $f \in F$, is marked as successful.

It is argued that this fundamental view can be a very compelling model for attack specifications; possible extensions are outlined in Appendix A.

EFSA	**Programming Construct**
$s \in \Sigma$	Event represented as a hash map
$q \in Q$	Instance of a State class.
s	Initial designated state where processing should start
$F \subseteq Q$	States designated as accepting
M	Variable of state information, represented as a hash map
V	Function that takes as argument the incoming event and the state hash map variable, returns *True* or *False*
δ	Instance of an EFSA class which handles the transitions between states

Table 5.1: Pairing of EFSA elements and programming constructs

6. Prototype Architecture and Design

In this section, an *in vitro* system is designed, to address the concerns set by the current study, using the theoretical model previously presented. After an overview of the system requirements derived from the Domain Analysis in Section 4, design and architecture are discussed, followed by implementation details of the prototype.

6.1. Prototype Requirements Overview

A set of requirements for the prototype are derived from the Description/Specification phase. These concern practical considerations or constraints of the proposed design.

Identifier	**Requirement Description**
R1.	Perform System and NetFlow log analysis
R2.	Introduce a common log entity specification
R3.	Have flexibility in input mechanisms
R4.	Maintain limited dependance on input data size
R5.	Realize high insert performance in data input
R6.	Architecturally support online analysis
R7.	Enable effective attack modeling
R8.	Allow combination of scenario sequencies
R9.	Provide facilities to aid understanding of scenario specification
R10.	Minimize disruption of network services, present topology and internal processes
R11.	Support required variability in order to change and/or easily introduce input mechanisms and log formats
R12.	Be flexible in data store deployment options
R13.	Expose an Application Programming Interface (API), for interoperability with future plug-in components
R14.	Exhibit realistic processing/response times

Table 6.1: Prototype Requirements Overview

A set of further requirements (R11 to R14) are derived from stakeholder concerns. Since the core analysis mechanism can be flexible, new input mechanisms and log formats should be easy to introduce. Regarding deployment options, due to the sensitive nature of security-oriented applications change is not easily embraced. A fully functional system must be thoroughly tested and verified before being ready for production-level deployment.

6.2. System Decomposition

The module decomposition view presents the modular division of the system, which is comprised of the Extraction, Analysis, Runtime and Application layers.

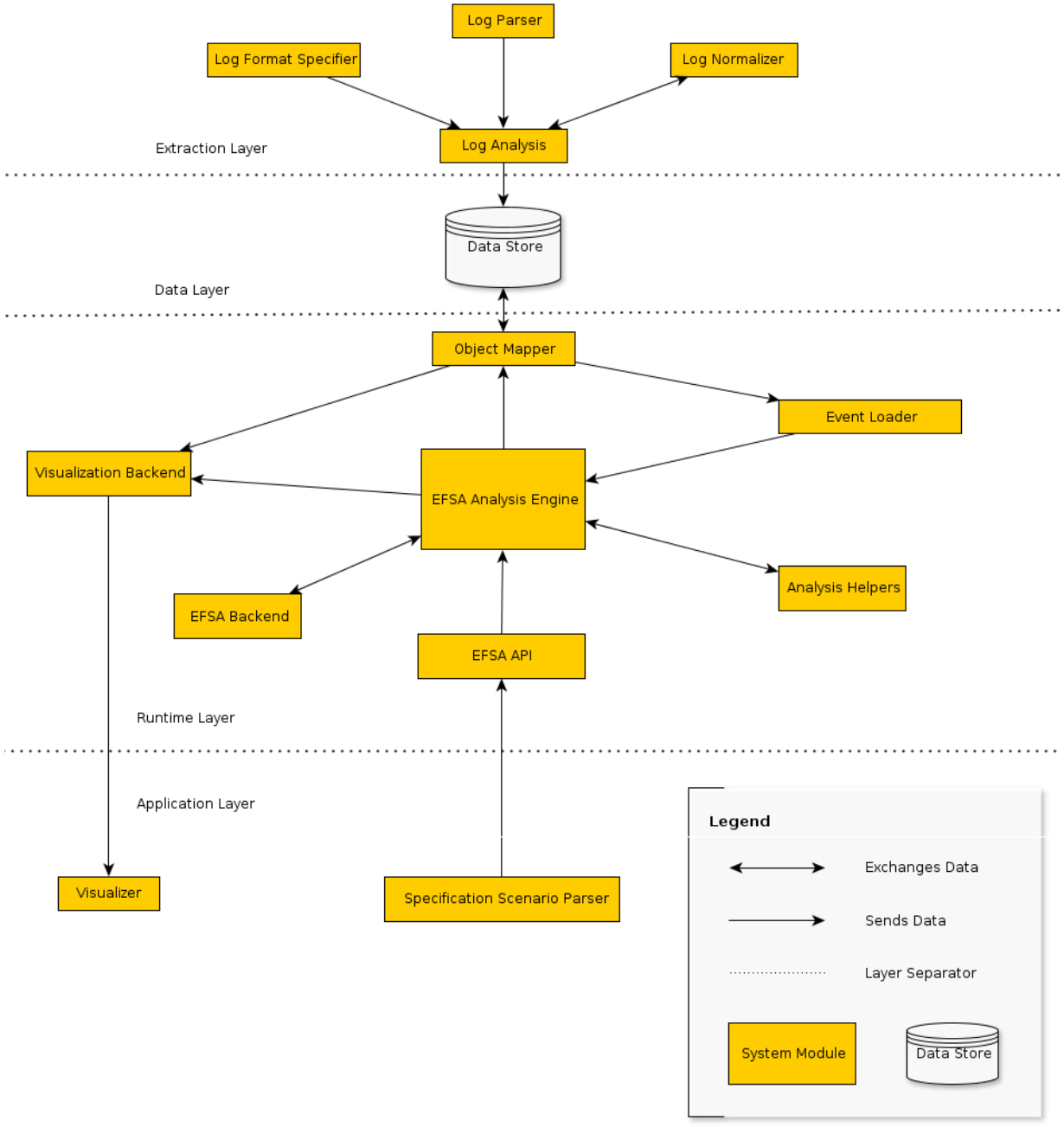

Figure 2: System Decomposition View

In the Extraction Layer, the central part is the Log Analysis module, which performs the necessary logic depending on the input it receives from the Log Parser, while the Log Normalizer and Format Specifier weigh in the extraction mechanism. In the Runtime Layer, everything revolves around the central EFSA Analysis engine, which making use of the EFSA backend and various helper functions, receives an event stream from the relevant submodule. The Application Layer, is responsible for parsing the user-inputted scenario specifications, and building the relevant EFSA, making use of the API that the Runtime Layer exposes. At a later stage, a visualization module displays the results back to the user.

6.3. Data Format, Storage and Extraction

The implementation of the data extraction mechanism, is centered around optimizing bulk insert performance. Initially, raw logs from various systems are placed by a script locally, in a non-uniform tree structure. The file formats contained range from text files to various compression formats used by each system. Based on assumptions set previously, every log line is parsed and the relevant fields are extracted, populating an 'object' entity with the detected fields.

At the parsing runtime level, a multi-process approach is used, in order to maximize the disk throughput. Concurrent processes traverse the log directory structure, parsing log files and using a database connection store the objects at the appropriate log category satisfying the requirements set on 6.1. This way, the overall process is disk I/O-bound, and horizontal scaling by adding more database shards is possible. When the data extraction process is finished a visualization facility allows the expert operator to inspect present data characteristics and distributions, to ensure qualities required for scenario specification.

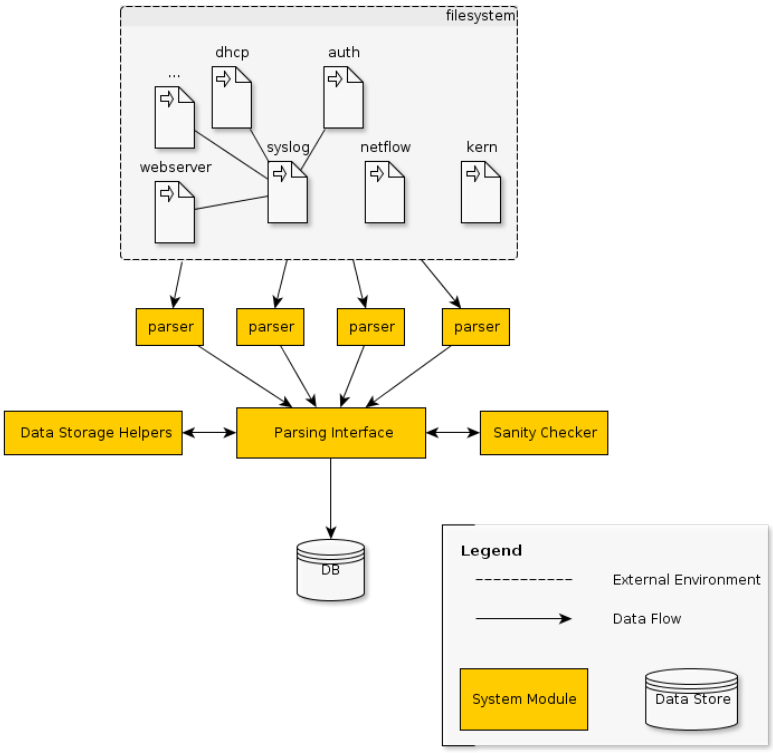

Figure 3: System Data Extraction flow diagram

6.4. Sequence Analysis and Runtime Architecture

Since the proposed Extended Finite State Automata approach is directly compatible with an event stream model, this is reflected on the runtime architecture. Given an EFSA, the goal is to investigate whether it can reach an accepting state, using a sequence of log events. This sequence, can be previously bound to and filtered by defined fields, to reduce the data needed for runtime computation. One should keep in mind however, that basically the existence of any satisfying subsequence is necessary for completion, as transitions are fired only on permitting events.

Design concerns raised regarding the sequence analysis operation include:

- The number of EFSA to be applied is unknown
- The size of the log event stream can not be determined beforehand

The first point is about the EFSA instances in memory, which is a space concern. The second point addresses time, since the system must exhibit realistic response times (R14). A tradeoff is attempted in those aspects, outlined in the next section.

The number of operations or their size is unknown at runtime, so measures must be taken to achieve a time-space tradeoff regarding memory and

disk storage. Regarding the actual EFSA representation, a flexible construct based on hashed values allows for minimization of the memory footprint. A range of helper functions model the behaviour of the theoretical foundation outlined in chapter 5.3. An attempt to counter the second concern with filtering on common event attributes is applied.

```
Data: Event Pool
Result: Satisfying Event Sequences
initialization;
attempt to filter on common attributes present on all states;
while events exist do
    get event from pool;
    foreach efsa do
        foreach next state do
            if accepting state then
                store run results;
            end
            if next state conditions satisfied then
                store event at state;
                transition;
            end
        end
    end
end
postprocessing;
```

Algorithm 1: Outline of simulated EFSA event stream processing

However, since the data size is significant, disk latency and datastore overhead are a serious hindrance, even for simple operations. To leverage against this, chunks of events size-dependent on available memory are sequentially loaded into an event pool, and the event stream abstraction is simulated. A possible optimization would invert the states in unreachable paths with respect to acceptance, thus progressively reducing the size of EFSA instances in memory. In the given context this proved not to be a concern and was not investigated further, as the bottleneck was the I/O operations of loading the event objects.

In every case, the results are stored back to the datastore, as the events that led to an accepting state of an EFSA. Extra code provided by the user can apply aggregation operations.

6.5. Scenario Specification: API and DSL endpoints

Building upon the EFSA definitions and system requirements, an Application Programming Interface is exposed which is responsible for defining specifica-

tions and handling the operations on data.

A scenario is programmatically represented as follows with respect to the definition in Section 5.3: a given EFSA is an instance of a class which handles generic initializations, and is responsible as well for graphically depicting the resulting model. Then, states (q) are defined: these are comprised of the state's information (M_i), which are checked if applicable against incoming event objects (Σ) using set operations. Alternatively, a custom state context-aware checker function included in the state handles this (V_i). Subsequently, transitions between states are declared, a state is characterized as initial (s) and others as accepting (F). Other facilities extend this runtime model by offering endpoints for user provided functions by firing events on state entry or in the case of a transition. Table 5.1 shows in context the pairing of this functionality with respect to the appropriate theoretical component.

A basic Domain Specific Language, is used as a user endpoint. It is intended merely as a proof-of-concept; functionality is reduced to simple scenario specification. Scenarios are parsed, and the relevant API calls form the resulting EFSA. A state declaration, is comprised of the parameters relevant to its context. These can refer to characteristics, the domain of values of the state, or a custom checking function to apply to incoming events. Subsequently, transitions are declared. The EFSA constructed by the user, is finally depicted in a diagram to aid understanding. What follows is the processing of the event stream that was specified (or a correlation of streams), and the results are presented back to the operator.

```
[su_escal]
stream=syslog
every=user
state idle {
    initial=True
}
state loginopen {
    action=open|accept
}
state escal {
    program=su
    action=open
    accepting=True
}
idle->loginopen
loginopen->escal
```

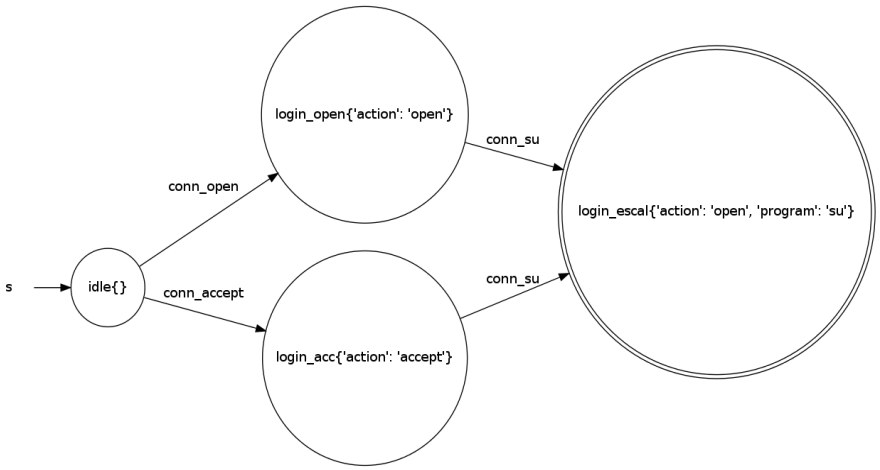

Figure 4: EFSA of a su priviledge escalation scenario

User authentication being a prime domain of security concerns, two scenarios are considered to showcase simple operations of the DSL. Due to the large number of users involved in the systems in question, close automatic monitoring of their login behaviours is highly sought. Ability to specify scenarios of this type where requested by the stakeholders. As a simplistic example, consider a hypothetical case where a user logs in and succeeds in priviledge escalation using the 'su' program, as a scenario depicted in Figure 4 along with the accompanying EFSA diagram. The action associated with a successful login, can be either 'open', or 'accept', so the declaration of an 'or' field is possible. The resulting EFSA is applied to every user available, and the successfully matched events are returned.

A common SSH authentication case of interest, is when an attacker tries an arbitrary number of user login attempts, ultimately being successful. A scenario of this type, can be useful for detecting a guest attack, by keeping track of the failed login attempts to an account, generating an alert if it crosses the maximum number of retries [8]. A variation point applied is that the failed attempts can be on a wide spectrum of different login hosts across the network. A characteristic in this case language-wise is the repetition of a state, introducing a 'loop'-like functionality in the EFSA model, as outlined in Listing 1.

Listing 1: EFSA of a brute force login scenario

```
[auth_attempts]
stream=syslog
every=user
state idle {
    initial=True
}
state loginaccept {
    action=open|accept
    accepting=True
}
state loginfail {
    source=SSH_SERVERS
    action=fail
}
idle->loginfail*3->loginaccept
```

As another example but making use of NetFlow data, consider a network worm's infection process. A computer worm is a standalone malware computer program that replicates itself in order to spread to other computers on the network. Unlike a computer virus, a worm does not need to attach itself to an existing program, but rather relies on security vulnerabilities on the target computers for spreading. To do this, it scans an arbitrary number of network hosts for possible infection, and if it succeeds it copies itself. However in this process uniform connection characteristics are observed. Such is the case for the *blaster* worm, where the manifestation of its propagation [33] can be easily modeled over the NetFlow protocol as a sequence as in Figure 5.

In a *blaster* infection, victim scanning occurs over TCP on port 135. Given a successful connection, exploit code is transfered to the infected host with a distinct flow size. Subsequently, worm code upload is initiated with a connection over 4444/TCP, and the actual upload happens over 69/UDP also with a distinct flow size [33].

Note that this is in contrast to the stateless, aggregation method described in [33]. In this case, propagation is modeled as a sequence of events using EFSA.

```
[blaster_classifier]
stream=netflow
state idle {
    initial=True
}
state victimscanrpc{
    prot=tcp
    dstport=135
    size<2017
}
state initwormcode{
    prot=tcp
    port=4444
}
state downwormcode{
    prot=udp
    port=69
    size=6592
    accepting=True
}
idle->victimscanrpc
victimscanrpc->initwormcode
initwormcode->downwormcode
```

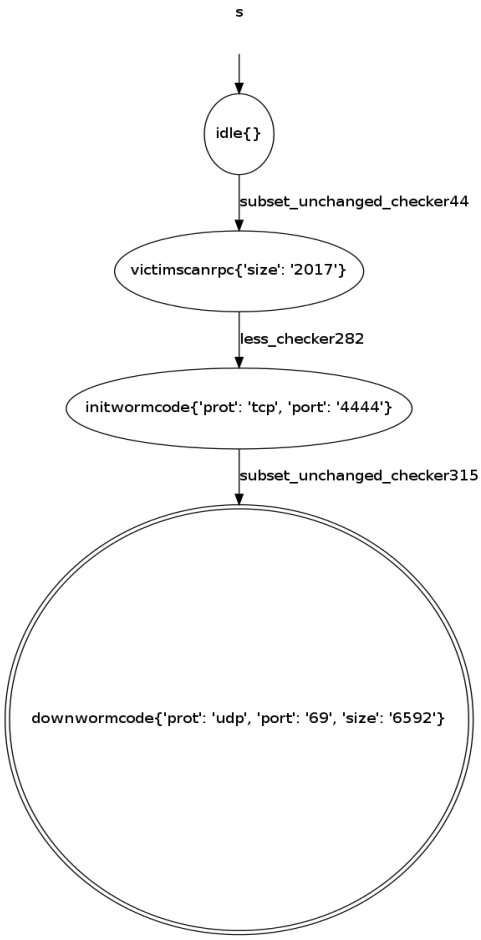

Figure 5: EFSA of a blaster worm infection scenario

Prototype Runtime

The prototype system was implemented using the *Python* programming language, using a single MongoDB database on a test machine as backend. Some runtime statistics can be found in Table 6.2.

Test machine: i5-2400 CPU @ 3.10GHz, 4GB RAM. Memory benchmarks by *Pympler*.

Operation	Remarks	Performance
NetFlow Logs Parsing, Index Building	15 days data 80GB, 108mil. objects	~6h
Authentication brute force login scenario run	30 days data, 255k objects	~40sec
NetFlow Connection Refused run	Size-intensive scenario	~2h
Memory footprint of a runtime EFSA instance	TCP connection refused EFSA, class size (states size)	1.73 KB (1.54 KB)

Table 6.2: Indicative prototype operations performance

7. Evaluation

Evaluation of the analysis mechanism as well as the prototype system is attempted with two case studies. These are set in different contexts, making use of different data and scenarios. Questions that are addressed pragmatically in this section revolve around topics such as:

- Is the EFSA analysis mechanism successful in detecting sequencies that use a correlation of events?
- Is the API interface adequate and diverse enough for real-world specification of scenarios?
- Is the DSL endpoint descriptive and expressive enough?
- Is the resulting output up to par for understanding?

7.1. Case Study: Slow Portscan Detection

Problem: Is it possible to detect slow portscanning in computer networks using Netflow data?

As a case study, a slow portscan scenario is considered, ideal for the simplistic underlying concepts, showcasing the sequence analysis mechanism on the large and dense NetFlow data set. The EFSA model used is the simple TCP connection refused in listing 2.

7.1.1. Context

Portscanning is considered a minor event by network security specialists, and this phenomenon is observed in every large network. Worms, malware or attackers perform automated scanning, searching for services and vulnerabilities, by means of attempting to connect to various ports on target machines. This process has the goal of information gathering about the network topology, possible vulnerabilities, or in the case of worms and malware, hosts that

can be further infected. It's significance is also correlated with recognizing precursors to more serious attacks.

Portscanning with respect to a specific host (thus vertical) is a generally understood problem [32], and many Intrusion Detection Systems monitor firewall log information and extract scanning patterns. However, by introducing two variation points to this simple scenario, detection is hindered.

Slow portscanning: the attacker introduces arbitrary long pauses between individual connection attempts, in order to evade the IDS rules which typically monitor connections closely together due to time-space limitations. Such scanning can theoretically take a lot of time to complete per scanned host, and while being more costly for the attacker, is very effective [30].

Horizontal portscanning: in order to search for a specific vulnerability or service, an attacker or worm/malware scans a wide range of addresses, on a specific port. This method is highly unlikely to be detected on a specific host, as the only element of information recorded per host, is a single unsuccessful connection attempt.

The presented approach uses the NetFlow data from the router level. The first variation point is countered by applying the solution to the data store available, thus limited by the date range of the input data already present. Regarding the horizontal problem component, by observing the same connection characteristics across different hosts and distinguishing between inner and outer counterparts, portscans can be detected. A more advanced approach, utilizing state variables counting elapsed time to detect SYN flooding attacks is presented in [17].

A foundational case where this type of horizontal port scanning is applicable, is a network worm's scanning process. Typically, hundreds or even thousands of probes with uniform characteristics to large blocks of IP addresses are observed, in short periods of time.

The fundamental case for modeling, is also the same TCP three-way handshake, as in a worm infection there exist three possible outcomes [29]. In the first one, the destination host is alive and the corresponding vulnerable service targeted is running. A TCP connection is established and possibly the worm manages to infect the target host. This case can be specified by the security expert as for the *blaster* [33] worm in Figure 5. In another case, the destination address the worm attempts to connect to does not respond to the SYN connection requests of the worm-infected source. In this case, a flow with only the SYN bit set is observed on the network, but may not show on the NetFlow records (Section 7.3). In the third case, the destination host is alive, but refuses a connection on the requested port and a flow with the RST/ACK bits is observed. Aggregation and visualization over size on NetFlow records of this third case are the primary methods of identifying worm propagation.

Listing 2: EFSA TCP Connection refused with custom checker functions on states

```
tcp = SaraEFSA('tcp_connection_refused')
# Common state variables
state_info.update({'client':client, 'dstport':dstport, 'server':server})
# States declaration
idle = Checker('idle', initial=True)
connect = Checker('connect', checker=conn_req, info=state_info)
accepted = Checker('server_accepted', checker=conn_acc, info=state_info)
refused = Checker('server_refused', checker=conn_ref, accepting=True, info=state_info)
established = Checker('client_established', checker=conn_est, info=state_info)
# Transitions declaration
idle['syn'] = connect
connect['syn'] = connect
connect['syn_ack'] = accepted
connect['rst_ack'] = refused
accepted['ack'] = established
# Initialization
tcp.initialize()
# A Custom checking function
def conn_est(obj, info):
    return info.get('attacker', None) == obj['srcip']
                    and info.get('flags', None) == 16
                    and info.get('server', None) == obj['dstip']
                    and info.get('dstport', None) == obj['dstport']
```

7.1.2. Analysis

Results expected from such an basic operation, are to be interpreted lightly. More advanced portscanning scenarios can be modelled as in [32]. The connections involved in such a scenario might as well be legitimate traffic from a misconfigured host. They can hint towards the construction of a list of *possibly* interesting or problematic hosts, and be used in conjunction with data from other IDS systems and methods with more confidence. However, a modeling instance of a worm or malware infection can be detected with this method by specifying specific connection characteristics. Example results in Listing 3 using aggregation operations on the resulting dataset, show the hosts involved where connections refused are more than a soft limit, the average time between attempts as well as the overall timespan.

As for the evaluation questions, this case uses only NetFlow data. The API interface can in detail express the characteristics of the satisfying event sequence, which in this simple case revolves around the addresses of the connection parties and the packet flags. However, the resulting output of the operation is definitely not adequate for human understanding; it requires further aggregation to make it useful as in Listing 3.

Listing 3: TCP Connection refused scenario aggregated results

```
198.51.5.201 -> 198.51.3.89 113 ports in 32969 - 60966 interval_avg: 1101.4375sec in 4 days
198.51.2.213 -> 198.51.6.156 415 ports in 32789 - 60879 interval_avg: 533.925121sec in 17 days
198.51.12.233 -> 198.51.6.182 927 ports in 32772 - 60963 interval_avg: 425.093953sec in 17 days
198.51.13.245 -> 198.51.6.217 1003 ports in 32792 - 60987 interval_avg: 89.3503sec in 2 days
198.51.5.201 -> 198.51.3.90 119 ports in 32792 - 60881 interval_avg: 1103.70339sec in 7 days
198.51.12.233 -> 198.51.6.84 120 ports in 32901 - 60312 interval_avg: 3874.521008sec in 7 days
198.51.13.245 -> 198.51.6.84 155 ports in 32952 - 60875 interval_avg: 614.38961sec in 7 days
198.51.29.138 -> 198.51.6.84 149 ports in 33066 - 60834 interval_avg: 1880.770271sec in 7 days
198.51.3.226 -> 198.51.6.84 144 ports in 32835 - 60969 interval_avg: 447.601399sec in 7 days
198.51.3.227 -> 198.51.6.84 161 ports in 32815 - 60846 interval_avg: 172.46875sec in 7 days
198.51.13.110 -> 198.51.6.123 127 ports in 33418 - 60780 interval_avg: 1677.396826sec in 7 days
198.51.13.245 -> 198.51.6.123 133 ports in 32884 - 60933 interval_avg: 2746.469697sec in 7 days
198.51.12.233 -> 198.51.6.236 250 ports in 33294 - 60980 interval_avg: 67.485944sec in 17 days
```

This scenario was targetted specifically to test the system with a simplistic EFSA model under heavy load due to the high number of accepted sequencies and the density of the NetFlow dataset. In Listing 3, sequencies of SYN packets followed by RST/ACK between two hosts are considered as refused connections. Verification of the data is performed by manually filtering the log entities in the dataset and comparing the outcome with the EFSA operation results.

7.2. Case Study: Remote Shell Web Application Exploit

Problem: Is it possible to model and specify attack scenarios on web applications through correlation of event data?

7.2.1. Context

Web applications today, can contain dangerous security flaws and are often the entry point in an otherwise secure network. No matter how strong firewall rulesets are, or how diligent the security patching process is, the web application layer poses many risks. Two of the most common as per the OWASP consensus [37], are injection and directory traversal attacks.

The injection technique involves including portions of code statements in an entry field in an attempt to get the application to pass a newly formed rogue command for execution. Directory traversal attacks consists of exploiting insufficient validated user-supplied input, so that the attacker gains access to file or binary that was not intended to be user-accessible.

Injection attacks involving SQL queries are encountered very often in production environments, and automated penetration testing tools assisting attackers also exist. Evidence of injection attacks are found in malformed request URLs in web server logs, and they have been studied thoroughly by the

scientific community [38]. Their detection is done with regular expressions that look for certain character patterns [37, 36]. Another case, where plenty of forensic information exists, are attempts involving malicious file execution through directory traversal attacks.

The flow of such an attack vector, is as follows [38]: In this scenario, an attacker discovers a vulnerability in a web application by performing repeated discovery attempts on the server. He then proceeds to exploit the poorly written application code through e.g. malicious code injection. At this point, the attacker has succeeded in acquiring access as the web server user. Then, the inserted exploit sets up a remote shell from the web server back to the attacker, on an usually high port. Once the attacker gets access to the server shell, the attacker can use the compromised host to scan the local network for other possible targets. He then attempts to escalate his privileges on the server using a local exploit, downloads and subsequently installs a rootkit.

7.2.2. Sequence Model

Using the scenario presented in [38], the modeling of the attack can be broken up to distinct events; the sequence of steps the attacker follows, are evident in the system logs. A model that encompasses all possible states and transitions must be implemented and checked against the data. Regarding the attack on the web server, several techniques based on regular expressions exist to detect e.g. injection attempts. Subsequently, the possible attempt of the locally compromised server to communicate with the attacker host will be evident in the NetFlow dataset, as a connection from an unusual port. Since the server can be considered compromised at this point, the attacker will possibly try to escalate the priviledges of the web server user, which in a simple scenario will be evident on the system logs. In the case that he fails, he uses the compromised server to scan for other vulnerabilities on the network, which is evident on NetFlow activity.

In Figure 7.2.2, states corresponding to an injection attempt and discovery phase as well as a traversal case is modeled. The programmatically corresponding scenario is trivial and is omitted; a satisfying example sequence can be found in Table 7.2.2. In a real world scenario however, multiple vulnerabilities would correspond to multiple states at the first level, leading to the infection server state.

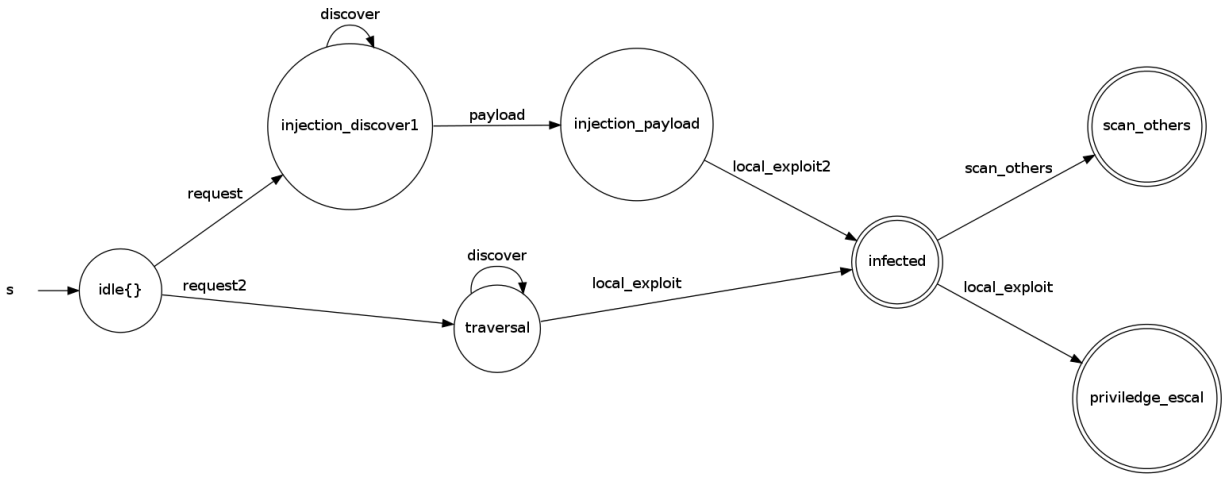

Figure 6: EFSA of a remote shell web application scenario

Event	Relevant Event Payload	Explanation
a	'body': /cgibin/af.cgi?_browser_out=|\necho;id;exit|	Example OS Shell injection attempt in request URL.
b	'body': /siteone/ index.php?page=http://explet32.org/CMD.gif?&cmd=cd%20/tmp; wget%20http://123.36.224.21/~fanta/dc.txt;perl%20 dc.txt%24384	Attacker coerces index.php to run CMD.gif which changes directory to /tmp, downloads the exploit dc.txt, and executes it, opening a high port on the server.
c	'srcport': 24384, 'dstip': '194.171.96.170', 'srcip': '194.171.96.171'	Flow from a high source web server port back to the attacker's IP.

Table 7.1: Example sequence of events leading to an accepting state of a web application exploit EFSA

7.2.3. Analysis

In this case, the scenario uses a combination of webserver and NetFlow events. In contrast to the previous one, in this use case a mere notification of a matching event sequence is adequate as a report to the operator; this is a discrepancy that is of note with regard to the reporting facilities of the prototype system and DSL.

Regarding the accuracy of the attack model, several log events will successfully match on the initial attack states of the EFSA, as automated attacks

where scripts randomly issue requests to servers are very common. However the automata will not traverse further, reducing the chance of the model generating a falsely positive result. As for the NetFlow rules involved, the problem of accurate threshold determination is also present. As expected, with a lower level of introduced abstraction, it is easier to correlate sequences that represent an attack and disambiguate them from harmless events. More on possible extensions of the EFSA model in this regard can be found in Appendix A.

7.3. Threats to validity

NetFlow Records Accuracy and Limitations
Netflow despite being a relatively widely used router protocol, its implementation might exhibit minor differences between routers of different vendors or versions. While Netflow records used in this study are considered homogenous in this regard, this must be taken into account as a general directive.

Another consideration is that flow records are a statistical measure, and may not be completely accurate even at the lower level. Whereas the conceptual flow model continues to apply, outliers can be missed. This phenomenon can happen for a variety of reasons related to data sampling or network conditions. As connection routing can change abruptly as the network rearranges itself, the fingerprint mirrored in the flow record can be arbitrary, and might reflect any condition of the underlying connections. Load balancing between base router configurations, or BGP route changes before flow expiration are a typical cause, as interfaces can change. Inactivity timeouts or slow responding hosts may cause packet retransmissions, which is not distinguished in the relevant flow fields; there exists a multitude of border cases which affect record accuracy. Situations like this happen often *ex vitro*, and thus must be taken into account when modeling scenarios.

NetFlow deals with and stores highly filtered traffic. For example, traffic may or may not be logged if it is dropped by a switch or router on another network component. In this study, only flows going from a server to a client are considered; by doing this, the problem with clients generating false positives by trying to contact services that do not exist is avoided. Another issue is the time resolution used in the operations on data. In this study, sensitivity up to a second was used, to simplify data intensive operations and indexing. However, this is not a realistic condition, and the millisecond resolution natively offered by routers is more accurate.

To sum up, the largest disadvantage of using NetFlow data for attack detection is that there is absolutely no access to the raw network traffic, but only to a restricted subset of meta-data. This by definition severely limits the attack detection potential of the dataset.

False Positive and False Negative Rates
Generally, the major hindrance with anomaly detection techniques is a high

false positive rate, where there is signaling to the system to produce an alarm when no attack has taken place. Conversely, failures to detect an actual attack are referred to as false negatives.

Specification based detection, aims to bridge the gap between the two via making use of domain expert knowledge, implemented via Domain Specific Languages. In this case, apart from the efficiency of the language constructs and general abilities, the rate of success lies in the specification. On one hand, the language should support the required variability to express accurately attack specifications; on the other hand, facilities in the underlying system should guide the expert through this process and the model itself should provide intuition in baseline determination. More on possible extensions of the EFSA model in this regard can be found in Appendix A.

Model Limitations

Regarding the EFSA model, a crucial assumption has been made regarding multiple satisfaction of neighbouring states. The case where conditions on more than one states are satisfied, is simplified to the transition to the first satisfying one. This was done to aid conceptual understanding and minimize model complexity, but in the real-world all path conditions must be taken into account.

Logging Constraints

The base assumption made on every study of security log analysis, is that log records actually exist. This is an important constraint, in the sense that an attacker can have the chance to remove revealing entries in logs. This is highly pertinent to system and authentication data, and must be taken into account when considering any scenario in this context.

7.4. Discussion

Regarding the validity of the model, the EFSA proves to be a very accurate archetype in the representation of the discrete states and possible connections between them.

However, more advanced language constructs in the Domain Specific Language component would allow more accurate representations of abstract data types. Describing characteristics in a more flexible way, would open up more possibilities for extended scenario specification. Advancements in this regard would lower the false positive and false negative rates altogether, by enabling more advanced specification of attributes of accepting event sequencies.

In the current implementation, lack of advanced features are leveraged by directly building models through the generic runtime API and supplying custom functions. However, expressiveness in the current state is highly hindered. Experience has shown great effectiveness in fully designed languages for security related purposes [16]. Affected are both syntax and semantics; special syntax would be optimized for the domain at hand, and allow for complex semantics definition and use. Secondary attributes such as ease of use would also be improved.

Rigorously defined and implementation agnostic language syntax support, optimized for patterns specification would allow reuse of scenarios. This could lead to an emergence of library-like abstractions that would radically affect the specification's semantic capacity.

Regarding the presentation of results of operations, automatic domain specific aggregation functionality leading to graphing capabilities would greatly benefit the end user. However, the event sequence notion must be preserved internally, in order to allow for interoperability and system extension. This is evident in the discrepancy between the two case studies presented: in the first one where the model is very simple but applied to numerous subsequences of client-server pairs, heavy aggregation operations are needed to group result primitives i.e. by host or port number into meaningful reports. In the second one, merely a notification of an accepting sequence is adequate for a qualitative result. Despite representing two quantitatively different types of operations, this disagreement in results semantics and size should be managed by the DSL backend as transparently as possible.

8. Conclusions

In this study, a solution pattern using Extended Finite State Automata for attack modeling was presented along with a prototype runtime and proof-of concept DSL endpoint. To aid in information security-oriented specification, an attempt is made to quantify classification of attack vectors by analysis in discrete steps.

This EFSA model, is argued to be very effective in classification as a fundamental and precise primitive in sequence analysis. Attack vectors in this case are composed of sequences of events, and functional specifications using states, actions and connections between them form a very able construct, reflecting expert knowledge. Difficulties associated with this process can be countered by a Domain Specific Language, which would aid and enable the domain expert.

Conversely, the runtime operations derived from such specifications, show the viability of the model in this context. It is compatible with both online and offline modes of analysis, and a plethora of applications correlating security-oriented information are possible. These augmented state machines, offer a solid foundation for accurate conceptualization and specification of attack vectors; using this as a base, further applications with learning properties in mind appear promising. A valuable byproduct, is that understanding of increasingly complex specifications is intuitive to the operator, and abstraction is possible to some extent.

The benefits of having a dedicated language for attack manifestation specification are evident; complex scenarios using a plethora of event sources can be modeled in an abstract yet concice way, a notion which encompasses a different viewpoint than signature based solutions. The high number of false positives which is the usual problem with intrusion detection techniques, is reduced in the case of security specification languages by actively leveraging domain specific knowledge. This approach appears highly promising, and further advancements in this field could utilize a wide plethora of system or network information. However, the problem of accurate threshold definition remains; future work on combinations of detection methods and advanced modeling using EFSA can shed light on this important matter.

A. EFSA as a Foundation & Future Work

The EFSA model along with the accompanying event stream concept, exhibits several advantages as a powerful construct for the study of network and security behaviour. It provides a solid foundation, and is also highly compatible with additional, more advanced analysis methods.

The practice of using a state transition model for attack detection is evident in scientific literature, and there exist also frameworks to translate between state-based languages, as indicated for AsmL and STATL in [18]. This increases interoperability between different implementations and allows for a common base for discussion and applications of security specification languages.

Moreover, the model presented in this study can be extended further in the application layer to exhibit more advanced capabilities, presented here as possibilities of future work.

A.1. Threshold Definition

A byproduct of the EFSA model defined in this study, is its ability to address also the problem of accurate threshhold definition, a general theme in security analysis. Especially in networking environments, irregularity in expected behaviour is the norm and the parameters of the problem increase significantly. In this case, the EFSA scheme can provide valuable analytics, by the *complete* modeling of interesting, possible scenarios. In such an instance, the designation of certain states as accepting is not that significant, but the flow of data by itself is, along with the effect it has on the automaton. Just by modeling every possible path the event stream might follow, the opportunity for the determination of a range of statistical attributes on the data arises; in this case, as an event possibly traverses the automaton, information about its path is recorded at every state. This can prove insightful in dynamically determining threshholds on the data, depending on specific scenarios modeled.

The per state information storage functionality, is implemented by supporting arbitrary state variables. To complement the case where every possible scenario is specified, features in the implementation of the EFSA model, allow checking for specific path traversal, given an event sequence. The statistical domain of values present in a state after the flow of an event stream through it, can be helpful to determine a threshhold with respect to the state conditions set within it. Note that usage of this functionality would potentially affect the space tradeoff mentioned in Section 6.4, that is the footprint of the EFSA in memory. In the NetSTAT attack specification language, local state variables are used to track elapsed time between events [17]. This concept can be expanded using the feature of local state information presented in 5.3, where for instance can be used for Distributed Denial of Service attacks,

in which aggregation of flow characteristics for behaviour modeling is highly beneficial.

A.2. Learning Properties

In a more advanced case, as an event flows through an EFSA instance, advanced logic present at a state can potentially affect the current domain of conditions. Essentially this consists of the addition of "state" property in states, which opens up a wide set of possibilities. Relevant computations on the state can be performed and the conditions present can be dynamically altered, allowing the system to adapt to the data flowing through it. The transition relation, can be an arbitrary processing function with access to the current state, dynamically defining new or changing the local domain V. This emergence of the system learning property can perform highly advanced correlations.

Both of these approaches are introduced in [28], where statistical properties of packet sequences are mapped into statistical properties associated with the transitions of the automaton; results are promising. This combination exhibits best the benefits of anomaly detection coupled with specification-based detection. Language support combining learning features with advanced specification development, would bring together both worlds, allowing efficient anomaly detection as well as accurate threshold definition.

References

[1] Varun Chandola, Arindam Banerjee, and Vipin Kumar, "Anomaly Detection: A Survey", ACM Computing Surveys, Vol. 41(3), Article 15, July 2009.

[2] Emilie Lundin Barse, Erland Jonsson: Extracting Attack Manifestations to Determine Log Data Requirements for Intrusion Detection. ACSAC 2004: 158-167 Ulf Larson, Emilie Lundin Barse, Erland Jonsson: METAL - A Tool for Extracting Attack Manifestations. DIMVA 2005: 85-102.

[3] Mathew Graves and Mohammad Zulkernine. 2006. Bridging the gap: software specification meets intrusion detector. In Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services (PST '06). ACM, New York, NY, USA.

[4] Mohammad Zulkernine,Mathews Graves,Muhammad Umair Ahmed Khan, Integrating software specifications into intrusion detection, International Journal of Information Security,Volume 6, Issue 5 , pp 345-357.

[5] GULP: A Unified Logging Architecture for Authentication Data, Matt Selsky and Daniel Medina, Columbia University, Pp. 1-5 of the Proceedings of LISA '05: Nineteenth Systems Administration Conference, 2005.

[6] K. Ilgun, R. A. Kemmerer, and P. A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach", IEEE Trans. on Software Eng., 21(3), March 1995.

[7] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors", Proc. of the IEEE Symposium on Security and Privacy, pp. 144-155, California, USA, 2001.

[8] Raihan, Mohammad Feroz, and Mohammad Zulkernine. "Detecting intrusions specified in a software specification language." Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International. Vol. 1. IEEE, 2005.

[9] G. Vigna, S.T. Eckmann, and R.A. Kemmerer, "Attack Languages", Proc. of the IEEE Information Survivability Workshop, Boston, MA, October 2000.

[10] K. Ilgun, USTAT: A Real-time Intrusion Detection System for UNIX., Master's Thesis, Dept. of Computer Science, University of California, Santa Barbara, July 1992.

[11] Cuppens, Frederic, et al. "Correlation in an intrusion detection process." Internet Security Communication Workshop (SECI'02). 2002.

[12] Abad, Cristina, et al. "Correlation between netflow system and network views for intrusion detection." Workshop on Information Assurance (WIA04). Phoenix, Arizona. 2004.

[13] Thonnard, Olivier, and Marc Dacier. "A framework for attack patterns' discovery in honeynet data", Digital investigation 5 (2008): S128-S139.

[14] Vigna, Giovanni, et al. "A stateful intrusion detection system for world-wide web servers." Computer Security Applications Conference, 2003. Proceedings. 19th Annual. IEEE, 2003.

[15] Abad, Cristina, et al. "Log correlation for intrusion detection: A proof of concept." Computer Security Applications Conference, 2003. Proceedings. 19th Annual. IEEE, 2003.

[16] Uppuluri, Prem, and R. Sekar. "Experiences with specification-based intrusion detection." Recent Advances in Intrusion Detection. Springer Berlin Heidelberg, 2001.

[17] Eckmann, S.T., Vigna, G., and Kemmerer, R.A., "STATL: An attack language for state based intrusion detection", Journal of Computer Security, vol. 10, no. 1/2, pp. 71-104, 2002.

[18] Q. Zhang and M. Zulkernine, "Applying AsmL Specification for Automatic Intrusion Detection", Proc. of the Workshop on Specification and Automated Processing of Security Requirements, pp. 221-233, Linz, Austria, September 2004.

[19] Vankamamidi, R., ASL: A specification language for intrusion detection and network monitoring, Master's Thesis, Dept. of Computer Science, SUNY at Stony Brook, NY 2001.

[20] Martin-Flatin, Jean Philippe, Gabriel Jakobson, and Lundy Lewis. "Event correlation in integrated management: Lessons learned and outlook." Journal of Network and Systems Management 15.4 (2007): 481-502.

[21] StreamSQL, http://www.sqlstream.com/.

[22] Esper, http://esper.codehaus.org/.

[23] OpenPDC, http://openpdc.codeplex.com/.

[24] Kinetic Rules Engine, https://github.com/kre/Kinetic-Rules-Engine.

[25] TIBCO StreamBase, http://www.streambase.com/.

[26] Sandro Pedrazzini, The Finite State Automata's Design Patterns. In Revised Papers from the Third International Workshop on Automata Implementation, Jean-Marc Champarnaud, Denis Maurel, and Djelloul Ziadi (Eds.). Springer-Verlag, London, UK, 213-219.

[27] Black, Paul E (12 May 2008). "Finite State Machine". Dictionary of Algorithms and Data Structures (U.S. National Institute of Standards and Technology).

[28] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. 2002. Specification-based anomaly detection: a new approach for detecting network intrusions. In Proceedings of the 9th ACM conference on Computer and communications security (CCS '02), Vijay Atluri (Ed.). ACM, New York, NY, USA.

[29] Detecting Worms and Abnormal Activities with NetFlow, Yiming Gong, Symantec Security, 2004.

[30] Bjarte Malmedal, Using Netflows for slow portscan detection, Master's Thesis, Institutt for informatikk og medieteknikk Hogskolen i Gjovik, 2005.

[31] Galtsev, Aleksey A., and Andrei M. Sukhov. "Network attack detection at flow level." Smart Spaces and Next Generation Wired/Wireless Networking. Springer Berlin Heidelberg, 2011. 326-334.

[32] Bhuyan, Monowar H., D. K. Bhattacharyya, and Jugal K. Kalita. "Surveying port scans and their detection methodologies." The Computer Journal 54.10 (2011): 1565-1581.

[33] Dubendorfer, Thomas, et al. "Flow-level traffic analysis of the blaster and sobig worm outbreaks in an internet backbone." Detection of Intrusions and Malware, and Vulnerability Assessment. Springer Berlin Heidelberg, 2005. 103-122.

[34] KrÃijgel, Christopher, Thomas Toth, and Clemens Kerer. "Decentralized event correlation for intrusion detection." Information Security and Cryptology-ICISC 2001. Springer Berlin Heidelberg, 2002. 114-131.

[35] Gregorio-de Souza, Ian, et al. "Detection of complex cyber attacks." Proc. of SPIE Vol. Vol. 6201. 2006.

[36] Detecting Attacks on Web Applications from Log Files, Roger Meyer, SANS InfoSec Institute, 2008.

[37] OWASP Top Ten Flaws in Web Application Security 2013, www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[38] Su, Zhendong, and Gary Wassermann. "The essence of command injection attacks in web applications." ACM SIGPLAN Notices. Vol. 41. No. 1. ACM, 2006.