Temporal Data Streams for Anomaly Intrusion Detection (Extended Version)

Abdulbasit Ahmed[2], Alexei Lisitsa[1], Clare Dixon[1]

[1] Department of Computer Science, University of Liverpool, Liverpool, UK
[2] College of Computing and Informatics, Saudi Electronic University, Riyadh, Saudi Arabia

Abstract. Intrusion detection systems (IDS) aim to protect computer systems against attacks. The detection methods employed in anomaly-based IDS are based, in particular, on monitoring networks for patterns of activity that differ from normal behaviour. Issues to be addressed with anomaly-based systems include deciding and representing what constitutes normal behaviour as well as being able to detect deviations from this efficiently in high speed networks. Here we describe an approach to anomaly-based intrusion detection utilising temporal logic and stream data processing. Temporal logic is used to specify the normality conditions which, after translation into data stream queries, are efficiently executed on streams of network packets. The proposed approach allows the concise representation of patterns of normal behaviour, possibly involving multiple steps, as well as being able to detect their violations over a high volume of data in high speed networks.

1 Introduction

In order to meet today's cyber security challenges a key requirement is the efficient processing of high volume, transient data coming from high-speed network traffic. According to an estimation provided by Endace [12], for the detection of the Conflicker worm "on a 50% loaded 1Gb/s link, a packet loss of 0.00002% could cause a system to miss the critical packet that triggers the alert." A second challenge relates to the difficulty of specifying cyber defence tasks in a concise and unambiguous way, which is transparent to the human user, given the ever increasing number and complexity of the cyber threats. Furthermore, the efficiency of the detection, prevention and response to security threats requires mechanisms for the reduction, abstraction, aggregation and comprehension of a vast volume of short lived, transient data.

Both of these challenges appear in the context of Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), which are designed to monitor networks or systems to detect and prevent against malicious entities and events. Whilst widely used IDS such as *SNORT*[3] (the de facto standard for open source IDSs) and Bro[4] have been successfully used to detect attacks, the advent of high

[3] www.snort.org
[4] www.bro-ids.org

speed/high volume networks entails the need for robust, scalable and efficient solutions. Additionally, to describe attack patterns or patterns of normal behaviour we often need the ability to describe not only the contents of network packets but also a particular sequence of packets or events in a suitable language. Many current IDS are not designed for the detection of temporal, multiple step attacks over flows of data, involve complex, low level languages to specify attacks and drop packets in high speed and volume network traffic.

In summary, we believe that Intrusion Detection and Intrusion Prevention, as well as many other cyber security related tasks require: the efficient processing of transient data (stream data) to cope with the high-volume and the high-speed network traffic; the ability to specify multiple-step patterns; and a high-level specification language to specify the attacks, policies, normal behaviour, etc in an unambiguous and concise way.

Intrusion detection methods and techniques fall into two broad categories - one consists of the methods using *explicit* knowledge of threats, that is *signature-based* intrusion detection (sometimes also referred to as *knowledge-based*, or *misuse* detection); and the other relies upon the detection of anomalies, or deviation from normal behaviour (*anomaly-based detection*). The second category can be further split into the methods *learning* normal behaviour at the operation time using statistical and machine learning techniques [21, 13, 18], and the methods using pre-defined normal behaviour [11], e.g. based on the specification of the network protocols.

In this paper we focus on the latter, that is specification-based protocol anomaly detection. Issues involve deciding what normal behaviour is and how to represent this as well as how to detect abnormal behaviour. We address both challenges above, that is efficiency in high-throughput traffic and the high-level, concise and transparent specification of IDS tasks.

We utilize and expand our generic concept of Temporal Stream Processing for Cyber Security, proposed in the previous work [2, 3], which is based on the following principles:

1. temporal logic is used for the specification of data stream queries/data stream transformers; and
2. temporal specifications are translated to standard data stream queries languages and translations are used for the execution using available high performance data stream engines.

In previous work [2, 3] temporal stream processing was proposed for *signature-based* IDS where temporal logic is used to specify known attack signatures, which are translated into stream queries and then executed using a stream processing engine. Experimentation with the resulting system showed efficient detection in very high-speed networks. It outperformed *SNORT* in terms of the packet loss when working in the high-speed networks [3]. Furthermore, the system demonstrated the convenience of the high-level specification language especially for specifying multi-step attack signatures.

In this paper we extend the approach of [2, 3] and apply it to the *specification-based protocol anomaly detection.* Here, unlike our previous work, temporal logic is used to specify not the attack signatures, but rather the patterns of the expected normal behaviour. Then similarly to [2, 3] temporal specifications are translated into data stream queries which are executed by the efficient data stream processing engine. This paper should be viewed as a proof of concept for demonstrating these ideas. Further work is necessary to fully explore, analyse and demonstrate its potential. The contributions of the paper are:

– the application of Temporal Stream Processing for Cyber Security to anomaly based intrusion detection via the specification of normal behaviour;
– a discussion of what we mean by normal behaviour here and its representation using temporal logic;
– a demonstration of the approach using examples of both single step and multi-step normal behaviour.

The rest of this paper is organised as follows. Section 2 explains our approach to the specification of normal behaviour. In Section 3 we provide the syntax and semantics of the temporal logic we use to specify normal behaviour. Section 4 gives details about Stream Data Processing and some of the facilities it provides. In Section 5 we describe the TeStID System for network IDS. In Section 6 we show how parts of normal behaviour may be specified using temporal logic and Section 7 shows how this can be mapped into a corresponding stream query. In Section 8 we provide results of running these queries and we provide details of related work in Section 9. Conclusions and further work are described in Section 10.

2 Representing Normal Behaviour

In this paper we aim to model normal behaviour by capturing the requirements on the properties of individual network packets and of sequences of packets using temporal logic. Although the proposed approach is very flexible and is applicable for modelling the behaviour of arbitrary networks, including the simultaneous use of various protocols, for the purpose of this paper we restrict ourselves to the case of TCP/IP behaviour only.

Where the “normality” requirements come from is a delicate question. One may first try to reduce requirements on normality to the correctness of protocols and their implementations. For example, the relevant Request for Comments (RFC) can be consulted to identify expected or normal behaviour. An RFC describes the syntax, semantics, and the function of a protocol, separate from its implementation. However, as the authors of the most comprehensive formal analysis of TCP/IP and Sockets API [6] have noticed that the development of TCP/IP protocols was originally “focused on “rough consensus and running code” augmented by prose RFC specifications that do not precisely define what it means for an implementation to be correct.” This imprecision of semi-formal RFC specifications leads to the inevitable variability of behaviour of different implementations of the TCP/IP stack in use which are commonly accepted as correct or compliant with the RFCs. Furthermore, there may be implementations which are only partially compliant with all the required RFCs. Finally, the normal behaviour of TCP/IP may reflect the particular context of use, that is a particular setup of the network, the use of cyber defence tools (firewalls, IDS, IPS) etc.

Attacks relating to normal behaviour may be due to a vulnerability in the RFC itself, or due to the implementation of the RFC. There may be different implementations for the same protocol (e.g. for different operating systems) so attacks may affect some platforms which do not comply with the RFC but not others.

The Land attack simulates a TCP connection, but uses the victim's own IP address as the source address. The victim computer then attempts to contact itself in order to respond to the simulated connection request. If the target systems are not compliant with RFC 2267 [20], then they may crash or lose services for some time. When this attack first appeared in 1997 it was due to a vulnerability in the original TCP RFCs. The attack succeeded on some platforms because of the weakness in the specification that was not addressed by the implementers of the affected platforms. In 1998, RFC 2267 was released and the attack affected only the systems that did not implement the improved specification. The attack resurfaced again in 2005 on Windows 2003 and Windows XP SP2 [26] as these operating systems were not compliant with RFC 2267.

One way to capture all variants of normality in a uniform way is to use methods *learning* normal behaviour at operation time. Here techniques such as statistical methods are used which keep averages of particular values and detects whether thresholds are exceeded based on standard deviations [21, 13]. Other approaches propose the use of machine learning and data mining techniques such as clustering and classification [18]. This is a well-established approach in anomaly based intrusion detection in which detection of the deviation from learned normal behaviour leads to an alarm. While this allows the detection of unknown attacks and threats, it almost inevitably leads to a non-negligible amount of false positives and false negatives.

We focus here on another possibility, that is to use explicit specifications of the normal behaviour. We acknowledge the inherent incompleteness of normality conditions and do not aim to address the completeness issue. Rather, we take a pragmatic approach and develop the principles for flexible specification and efficient execution of normality condition checking. The flexibility covers all variants of normality conditions discussed above:

1. derived from the specifications of the protocols;
2. derived from specifics of various implementations; and
3. derived from different contexts of use.

The protocol anomaly detection based on the explicit specifications of the normal behaviour was first proposed in a [11] and since then has been developed e.g. in [23, 24] and implemented in such systems as the CISCO IPS Intelligent Detection technology[5] and the Tipping Point intrusion prevention system[6].

As a basis for the specification of normal behaviour, we use expected network communications behaviour as described in the RFC [11, 15]. We use, in particular, the TCP state transition diagram (see for example [27]) to derive the normality conditions. It ascribes the transitions between the states of the TCP protocol accompanied by *send* or *receive* action labels.

We assume that a network based IDS does not have access to the internal states of TCP stacks at the end points of communication and the only information available to IDS is that from the packets observed on the wire (*wire interface only* in terms of [6]). Because of that we consider the following general form of normality conditions derived from the specifications.

Principle 1 (normality of the specification compliant behaviour) *The sequence of packets observed on the wire is normal iff it is consistent with TCP state transitions diagrams at both ends of the connections.*

We express the condition above, as well as *all other normality conditions* in terms of *expected* behaviour: if the sequence of packets $l_1, \dots l_n$ related to the same connection and satisfying some side conditions has been observed on the wire then for the execution being normal we expect to observe the next one of a few possible packets with related side conditions. For a moment we can symbolically denote the requirements of the above type by

$$(l_1, \dots, l_n)_S \to \bigvee_i l^i_{S_i}.$$

Side conditions S and S_i here are understood very broadly and may be both on the timing constraints between packets and on the content of the packets. Notice that the compliance, if any, of the expected behaviour rules with the specification, is not subject to dynamic, on-line/runtime testing or analysis. If, indeed, the expected behaviour rule expresses normality with respect to TCP specification, as stated in Principle 1, then it has to be checked as such statically before deployment.

For protocol anomaly, only parts of the specifications are used. The use of full TCP/IP specifications would be problematic for two reasons:

1. there still would not be 100% precision due to the inherent imprecision and ambiguity of TCP/IP specifications; and
2. even with most detailed existing specifications, such as in [6] it would be too computationally expensive (compare with [6] where the compliance of TCP traces was checked off-line).

In the proposed approach, the parts of the specification can be selected due to their frequent use (e.g. session establishments and fragmentation/defragmentation

[5] www.cisco.com/c/en/us/products/collateral/security/ips-manager-express/
[6] www8.hp.com/us/en/software- solutions/software.html?compURI=1343617

of packets) or due to their critical nature (e.g secure data transfer and authentication). Furthermore, the conditions might not necessarily be derived from specifications, but rather chosen by a user to reflect the specifics of the normal behaviour of a particular implementation, or to reflect the particular context of use.

In the next section we present the precise syntax of the temporal logic capable to express the necessary details of such requirements.

3 Temporal Logic

Temporal logic is the extension of classical logic with operators that deal with time which allows us to formally specify temporal events. For example we can state that a formula φ will hold now or at some point in the future using the formula $\Diamond\varphi$. Temporal logics have been developed for real-time systems, for example Metric Temporal Logic (MTL) [16] where temporal operators are decorated with time intervals to specify timing constraints. For example, $\Diamond_{[0,5]}\varphi$ means eventually φ will hold within 0 to 5 seconds from now. This is useful in the specification of normal behaviour as we often need to state a sequence of events within some time interval. We use Many Sorted First Order Metric Temporal Logic (*MS-FOMTL*) [16] to represent patterns of *normal behaviour* relating to data packets arriving over time. This allows us to state expectations relating to the content of packets, the order they should arrive, relative timing constraints, etc.

The underlying temporal model, $\mathcal{M}$, represents the sequence of packet arrivals and we use temporal logic formulae, φ, to model normal behaviour. In anomaly IDS we want to provide an alert when the underlying temporal model does not satisfy the normal behaviour (i.e. $\mathcal{M} \not\models \varphi$ or equivalently $\mathcal{M} \models \neg\varphi$).

3.1 Representation of Packets and Models

The incoming network stream packets form the temporal model $\mathcal{M}$. Packets are captured in order by arrival time τ. Each captured packet belongs to a particular network communication protocol (TCP, UDP, ICMP, etc). In this paper we consider TCP as a case study. The set of all possible packets is denoted by $[\![\mathcal{P}]\!]$.

Rather than explicitly representing each field in a TCP packet we focus on twelve fields that we have used to represent attacks in signature-based detection [3]. This helps with the ease of presentation, demonstrates the usefulness of the approach, and helps with avoiding errors in the specification of normal behaviour. However, if necessary, we could incorporate additional fields.

We will use P to represent a predicate of the TCP protocol type. P has an arity of 12, $s_1 \times ... \times s_{12}$, where s_i $(1 \leq i \leq 12)$ is the sort of a particular predicate argument (i.e sender address, receiver address, sender ports ... etc.). The specification of a TCP predicate is:

$$P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12})$$

where:

- x_1 : is a string variable representing the sender IP address;
- x_2: is an integer variable representing the sender port;
- x_3: is a string variable representing the receiver IP address;
- x_4: is an integer variable representing the receiver port;
- x_5: is an integer variable representing the sequence number;
- x_6: is an integer variable representing the acknowledgment number;
- x_7: is a boolean variable representing the *ack* flag;
- x_8: is a boolean variable representing the *syn* flag;
- x_9: is a boolean variable representing the *rst* flag;
- x_{10}: is a boolean variable representing the *push* flag;
- x_{11}: is a boolean variable representing the *urg* flag.
- x_{12}: is a string representing the payload or data.

The packets arrive at some point in time. We consider the arrival of these packets as instantaneous, arbitrary occurring events. Two packets can not arrive at the same time, one must be before the other. The model of time consists of a set of arrival points $\mathcal{T} \subset \mathbb{R}^+$ (where $\mathbb{R}^+$ is the set of non-negative real numbers) and we require $\mathcal{T}$ to be discrete: for any interval $[a, b]$, the set $[a, b] \cap \mathcal{T}$ is finite. The model is represented as $\mathcal{M} = \langle \mathcal{T}, <, \mathrm{I}, I_s \rangle$ where:

- $<$ is a linear order on $\mathcal{T}$;
- $\mathcal{T} = \{\tau_0, \tau_1, \ldots\} \subset \mathbb{R}^+$, where $\mathbb{R}^+$ is a non-empty set of positive real numbers and $\mathcal{T}$ is the set of all arrival moments;
- I is an interpretation which maps $\mathcal{T}$ into $[\![\mathcal{P}]\!]$, $\mathrm{I} : \mathcal{T} \to [\![\mathcal{P}]\!]$ so, $\mathrm{I}(\tau_i)$ represents a packet arriving at a moment $\tau_i \in \mathcal{T}$; and
- I_s is the interpretation of sort s over the domain $\mathcal{D}_s$.

3.2 MSFOMTL Syntax

The syntax of *MSFOMTL* (Many Sorted First Order Logic) is based on [17, 16, 4]. Here we restrict our presentation of syntax and semantics to the symbols required in this paper. Logical symbols are the quantifier $\exists$, the logical connectives $\wedge$ ("and"), $\vee$ ("or") $\rightarrow$ (implies) and $\neg$ ("not"), the logical binary predicate symbols $=, \neq, <, \leq, >$, and $\geq$, the bounded future temporal operators $\Diamond_{[t_1,t_2]}$ ("eventually"), and $\Box_{[t_1,t_2]}$ ("always"). The subscripts $[t_1, t_2]$ in the operators refer to their scope (between the moments t_1 and t_2 from now).

In many sorted logic, the arguments of predicate and function symbols may have different sorts s and every sort $s \in S$ where S is a finite set of sorts. The non-logical symbols of *MSFOMTL* consist of the finite disjoint sets of predicate symbols, function symbols, constants, and variables. The alphabet of the language of *MSFOMTL*, $\mathcal{L}$, consists of the union of all the non-logical symbols.

Terms The set of terms in $\mathcal{L}$ of sort s is the smallest set of expressions with the following properties:

- Each constant symbol c of sort s is a term where $s \in S$.

- Each variable v of sort s is a term where $s \in S$.
- If f is a function symbol of n-arity $s_1 \times ... \times s_n \rightarrow s$ and te_i is a term of sort s_i, then $f(te_1, ..., te_n)$ is a term of sort s.
- If te_1 and te_2 are numeric terms of sort s, then $te_1 + te_2$, $te_1 - te_2$, $te_1 \times te_2$ and $te_1 \div te_2$ are terms of sort s, where $s \in S$.

Formulae The formulae of *MSFOMTL* are defined as follows:

- If $te_1 \times ... \times te_n$, where each te_i is a term of sort s_i, and P is a predicate symbol with n-arity $s_1 \times ... \times s_n$, then $P(te_1, ..., te_n)$ is an atomic formula.
- If te_1 and te_2 are terms of the same sort s then $te_1 = te_2$, $te_1 \neq te_2$, $te_1 > te_2$, $te_1 < te_2$, $te_1 \leq te_2$ and $te_1 \geq te_2$, are atomic formulae.
- Every atomic formula is a formula.
- If φ is a formula then $\neg\varphi$ is a formula.
- If φ is a formula then $\Diamond_{[t_1,t_2]}\varphi$ and $\Box_{[t_1,t_2]}\varphi$ are formulae.
- If φ and ψ are formulae then $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\varphi \rightarrow \psi$, are formulae.
- If φ is a formula and x is a variable then $(\exists x)\varphi$ is a formulae.

3.3 MSFOMTL Semantics

In *MSFOMTL* the arguments of functions and predicates have different sorts and each sort s ranges over a domain $\mathcal{D}_s$. We denote I_s as the interpretation over the domain $\mathcal{D}_s$. The following mapping can be defined:

- Each variable v in $\mathcal{L}$ of sort s is evaluated as v^{I_s} which is an element in $\mathcal{D}_s$.
- Each constant symbol c in $\mathcal{L}$ of sort s is evaluated as c^{I_s} which is an element in $\mathcal{D}_s$.
- Each functional symbol f of arity n and of sort $s_1 \times ... \times s_n \rightarrow s$ is mapped to a function $\mathcal{D}_{s_1} \times ... \times \mathcal{D}_{s_n} \rightarrow \mathcal{D}_s$. The evaluation of a function $f(te_1, ..., te_n)^{I_s} = f^{I_s}(te_1^{I_{s_1}}, ..., te_n^{I_{s_n}})$, where te_i is a term of sort s_i.
- Each predicate symbol P with n-arity $s_1 \times ... \times s_n$ is mapped to a predicate P^{I_s} which is a subset of $\mathcal{D}_{s_1} \times ... \times \mathcal{D}_{s_n}$. A predicate is evaluated to be true when $(te_1^{I_{s_1}}, ..., te_n^{I_{s_n}}) \in \mathrm{P}^{I_s}$, where te_i is a term of sort s_i.

A temporal formula φ holds at $\mathcal{M} = \langle \mathcal{T}, <, \mathrm{I}, I_s \rangle$ at an arrival time $\tau_i \in \mathbb{R}$, that is, $\mathcal{M}, \tau_i \models \varphi$ is defined recursively as follows:

$$
\begin{array}{ll}
\mathcal{M}, \tau_i \models P(te_1, ..., te_n) & \text{iff } P^{I_s}(te_1^{I_{s_1}}, ..., te_n^{I_{s_n}}) = \mathrm{I}(\tau_i) \\
\mathcal{M}, \tau_i \models \neg\varphi & \text{iff } \mathcal{M}, \tau_i \not\models \varphi \\
\mathcal{M}, \tau_i \models \varphi_1 \wedge \varphi_2 & \text{iff } \mathcal{M}, \tau_i \models \varphi_1 \text{ and } \mathcal{M}, \tau_i \models \varphi_2 \\
\mathcal{M}, \tau_i \models \varphi_1 \vee \varphi_2 & \text{iff } \mathcal{M}, \tau_i \models \varphi_1 \text{ or } \mathcal{M}, \tau_i \models \varphi_2 \\
\mathcal{M}, \tau_i \models \varphi_1 \rightarrow \varphi_2 & \text{iff } \mathcal{M}, \tau_i \models \neg\varphi_1 \text{ or } \mathcal{M}, \tau_i \models \varphi_2 \\
\mathcal{M}, \tau_i \models \Diamond_{[t_1,t_2]}\varphi & \text{iff for some } \tau^{'} \\
 & \quad (\tau_i + t_1 \leq \tau^{'} \leq \tau_i + t_2) \\
 & \quad \text{s.t. } \mathcal{M}, \tau^{'} \models \varphi \\
M, \tau_i \models \Box_{[t_1,t_2]}\varphi & \text{iff for all } \tau^{'} \\
 & \quad (\tau_i + t_1 \leq \tau^{'} \leq \tau_i + t_2) \\
 & \quad \mathcal{M}, \tau^{'} \models \varphi \\
\mathcal{M}, \tau_i \models (\exists x)\varphi & \text{iff for some } I_s\ x^{I_s} = a \text{ and} \\
 & \quad \mathcal{M}, \tau_i \models \varphi[x/a]
\end{array}
$$

Finally, the arithmetic functions (e.g, $+, -, \times, \div$) for numeral terms in $\mathcal{L}$ are the standard binary operations for arithmetic functions.

4 Stream Data Processing

Stream Data Processing [7, 5] is concerned with handling and processing flows of data. Data Stream Management Systems (DSMS) are designed to handle large volumes of data arriving in rapid, time-varying, continuous streams. They can handle queries that are issued once and then continuously evaluated over the data (continuous queries). For example, "raise an alarm when a packet is detected where the source and the destination address are the same and the *syn* flag is set". Another useful feature is *sliding window* query processing relative to an ordered field e.g time or tuple count. This focuses on a finite history relative to the current time where new data is added to the history and older data is removed. These features are well suited for applications like network monitoring, network traffic analysis, and intrusion detection. DSMS often have operators to partition streams into substreams based on the data in each record or to recombine substreams with the same data structure. Other useful features are aggregation, pattern matching, merging, and mapping the stream based on values in the data.

4.1 StreamBase Stream SQL

In this work we use StreamBase[7] a commercial DSMS. The StreamBase Complex Event Processing (CEP) platform allows us to build a system that can analyse and act on real time data. StreamBase has rich Stream Data Processing (SDP)

[7] www.streambase.com

functionalities that enable us to translate from *MSFOMTL* to its stream SQL. It has high performance and scalability features such as parallelism and multi-threading.

StreamSQL (SSQL) is a query language that extends SQL with the ability to process continuous data streams. Queries can be constructed using SSQL or graphical event flow tools. SSQL language has the following processing capabilities:

– Non temporal operators: These operators do not have a time window and act continuously on the stream (as the event arrives) allowing operations such as filtering or merging streams, creating sequences, timers, or tables (in memory or external storage), mapping values to the stream (e.g adding time stamps), correlating multiple streams, etc.
– Temporal operators: These operators have windowing constructs allowing us to query temporal events over a specified time window. The query is evaluated continuously within the sliding window. For example the *pattern match* and *aggregate* operators both have a time window. The *pattern match* operator accepts inputs and matches specified temporal patterns in the query with these inputs. The *aggregate* operator performs aggregations and computations on real time streams or stored tables. The time window in the *aggregate* operator specifies the time window for the aggregation and how to advance following the time window expiration.
– Extensibility: The StreamSQL operator set is extensible, developers can add functions, operators, adapters (input or output operators).

The SSQL language has many operators. Some of these operators are data definition language (DDL) operators, and some of them are data manipulation language (DML) operators. To query data streams, the *select* statement is used (from DML). The semantics of a query depends on the clauses that are used in the *select* statement for example the *filter*, the *pattern* and the *aggregate* operators. The syntax and semantics for these operators are as follows:

– The syntax of the *filter* operator is:

```
SELECT target_list_entry
               [, target_list_entry...]
  FROM event_source [...] |
  [WHERE predicate]
  [INTO stream_identifier]
```

where:
  • `target_list_entry`: are field identifiers;
  • `event_source`: is the source of input such as stream or table;
  • `predicate`: are conditions on the select fields that limit the returned set by the select statement;
  • `stream_identifier`: is a unique stream identifier which can be either the final output stream or a stream that can be used by other SSQL components.

In stream processing a query is evaluated continuously. So, at any moment in time a running query might be answered. The query is answered if it returns some results and this means the query valuation is true, otherwise it is false. The filter operator is used to query the incoming events and if there is a tuple that satisfies the restrictions specified on particular elements of the tuple at a moment of time τ, then that tuple is returned (output). This means the query valuation at τ is true. If no tuple is returned, then the query valuation at τ is false.

– The syntax of the *pattern* operator is:

```
SELECT target_list_entry
           [, target_list_entry...]
 FROM event_source [...] |
 FROM PATTERN template
           [pattern_operator template ...]
 WITHIN (interval TIME)
 [WHERE predicate]
 [INTO stream_identifier]
```

where:

- `target_list_entry`, `event_source`, `predicate` and `stream_identifier`: are as previously
- `interval TIME`: is a timeout in seconds;
- `template`: is a stream identifier;
- `pattern operator`: is a logical operator that relates a pair of templates (`NOT streamA`, `streamA AND streamB`, `streamA THEN streamB`, `streamA OR streamB`);

The *pattern* operator queries the event sources and only returns a true valuation if a query on the first template returns a tuple at τ that satisfies the conditions in the *where* clause. Then within the specified time interval (i.e. within $\tau +$ `interval TIME`) the second query relating to the second template returns a tuple that satisfies the conditions on the *where* clause and the logical pattern operator between the two templates. No tuples are returned if the pattern operator fails to match any pattern and this means the valuation of the pattern operator is false.

More information about the SSQL query operators mentioned here and other constructors can be found at the Streambase web site.

5 The TeStID System

TeStID (Temporal Stream Intrusion Detection) is a network based intrusion detection developed by the authors based on using temporal logic to specify attack patterns and using SDP as the attack detection engine. By combining the expressiveness, conciseness, clear semantics, and ability to represent temporal patterns of Temporal Logic with the stream processing technology we have developed an online system for network based intrusion detection that can handle high volume network traffic. In previous work [2, 3] we have described the use of TeStID for signature-based network IDS. It allowed the specification and detection of both single and multiple step attacks and outperformed both *SNORT* and Bro over high-speed networks.

Here we show how TeStID can be applied to anomaly network IDS. The advantages of the system are as follows:

– It provides a concise and unambiguous way to formally write the specification of normal behaviour.
– It allows the specification and detection of multiple step attacks.
– It is suitable for high volume/speed networks because of the underlying SDP engine.
– It is extensible in that additional or updated descriptions of normal behaviour can be added that can be automatically translated into SSQL queries.

5.1 TeStID System Architecture

The system architecture of TeStID is shown in Figure 1. It consists of the fol-

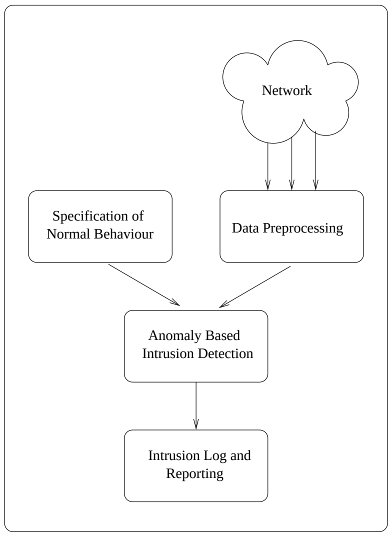
Network
Specification of Normal Behaviour
Data Preprocessing
Anomaly Based Intrusion Detection
Intrusion Log and Reporting

Fig. 1. TeStID System Architecture for Anomaly network IDS

lowing components:

– Data preprocessor: This component captures or sniffs the data that traverses the network. When it captures the data, it can perform preprocessing or filtering of the data as required by the module that uses it. For example, the module for detecting attacks against TCP/IP will need to process all the

TCP packets, the data preprocessing will filter out all the TCP traffic and provide it as a stream to the module. More filtering is possible for example filtering by source port or destination port.
- Specification of Normal Behaviour: This is written using *MSFOMTL* (see Section 3). These formulae are stored in a file which is read and parsed by the translator.
- The anomaly based intrusion detection: After parsing the *MSFOMTL* formulae representing normal behaviour the translator will translate this into equivalent Stream SQL queries which can be run and any traffic not matching the normal behaviour will be reported.
- Intrusion log and reporting: Provide reporting and logging of the attacks.

6 Protocol Anomaly Specifications

We use the *MSFOMTL* to represent parts of the TCP protocol normal specification ϕ and detect any deviation from this specification in the temporal logic models $\mathcal{M}$ (incoming events) (i.e. the protocol specification ϕ is not satisfied in $\mathcal{M}$ ($\mathcal{M} \not\models \phi$ or equivalently $\mathcal{M} \models \neg\phi$). The protocol specifies requirements on packets e.g. conditions or restrictions on the value of some fields and the order in which they arrive.

We consider two main types of specification of normal behaviour. The first involves the expected formation of a single packet. The second covers multiple step protocol specifications where we model normal behaviour of a fragment of the protocol.

6.1 Single Step Anomaly Specification

In the specifications of protocols, there are often restrictions on the fields e.g. upper and/or lower bound values on that field, the field must be within a certain range, or is based on another field value using logical comparison and/or arithmetic operation (e.g $x_6 \geq x_5 + 1$). We can represent this type of anomaly as follows:

$$\varphi \rightarrow \psi \tag{1}$$

where:

- φ is first order predicate (representing a packet); and
- ψ an atomic formula or Boolean combination of atomic formulae.

In summary, for single step anomaly, a formula represents normal behaviour (within some packet). When this formula is *not satisfied*, an alarm is raised. Thus, we aim to detect $\neg(\varphi \rightarrow \psi)$ or equivalently $(\varphi \wedge \neg\psi)$.

Example 1. RFC 6335 [8] states that the port range is from 0-65535. Also, it states that the lowest and highest bounds are usually reserved, i.e. ports 0 and 65535 are officially reserved by IANA [25]. So for normal specification requirements we could make sure that there are no packets with a source port (x_2) or a destination port (x_4) with a value less than or equal 0 and greater or equal 65535. The normal specification can be represented as:

$$\begin{array}{l}((\forall x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \\ P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \rightarrow \\ (((x_2 > 0) \wedge (x_2 < 65535)) \wedge ((x_4 > 0) \wedge (x_4 < 65535)))).\end{array}$$

The negated form is as follows:

$$\begin{array}{l}((\exists x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \\ P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \wedge \\ (((x_2 \leq 0) \vee (x_2 \geq 65535)) \vee ((x_4 \leq 0) \vee (x_4 \geq 65535))))\end{array}$$

Example 2. As another example for single step anomaly, we use the RFC 793 [14] for TCP/IP specification. In this specification, the reliability of TCP/IP is described. One of the basic requirements for this reliability (not losing data and the ability to recover) is assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment from the receiving TCP [14]. This means if the *ack* flag is set (x_7) the acknowledgment number (x_6) must be greater than 0. This is represented as:

$$\begin{array}{l}(\forall x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \\ (P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \\ \quad \rightarrow (x_7 = 1 \rightarrow x_6 > 0))\end{array}$$

and the negation of the above formula as follows:

$$\begin{array}{l}(\exists x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \\ (P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \wedge \\ ((x_7 = 1) \wedge (x_6 \leq 0)))\end{array}$$

6.2 Multiple Step Anomalies

The syntactical subset of MSFOMTL corresponding to multiple step anomalies allows us to specify expected behaviour rules formulated semi-formally in Section 2.

The syntactical form of multiple step anomalies is the following:

$$\begin{array}{r}\Box(P_1 \rightarrow \\ \Box_{[t_{s_1}, t_{e_1}]}(P_2 \rightarrow \\ \dots \\ \Box_{[t_{s_{(n-1)}}, t_{e_{(n-1)}}]}(P_n \rightarrow \bigvee_{i=1\dots k} \Diamond_{[t_{s_n^i}, t_{e_n^i}]} P_{n+1}^i)))\end{array} \tag{2}$$

(where P_i denotes a first order predicate) and the negation of the above formulae is

$$\begin{array}{rl} \Diamond(P_1 & \wedge \\ \Diamond_{[t_{s_1}, t_{e_1}]}(P_2 & \wedge \\ & \dots \\ \Diamond_{[t_{s_{(n-1)}}, t_{e_{(n-1)}}]}(P_n \wedge_{i=1\dots k} \neg \Diamond_{[t_{s_n^i}, t_{e_n^i}]} P_{n+1}^i))) \end{array} \tag{3}$$

The negated formula will be satisfied only when $P_1, \dots, P_n$ appear in the correct order within the timing constraints but P_{n+1} cannot be satisfied within its timing constraints.

Example 3. In this example we use the TCP simultaneous connection synchronisation specification as described in RFC 793 [14]. There are two ways for establishing connections in TCP described in the RFC, the three-way handshake and the simultaneous connection synchronisation. In the simultaneous connection specification the normal observable steps between clients A and B for the process in the RFC are as follows.

- Client A sends a packet to B with source IP (x_1), source port (x_2), destination IP (x_3), destination port (x_4), initial sequence number $= S_A$, and sets the *syn* flag.
- Client B just after the time that A is sending the above request, sends a packet to A with source IP (x_3), source port (x_4), destination IP (x_1), destination port (x_2), initial sequence number $= S_B$, and sets the *syn* flag.
- Client A receives the synchronisation request from B and responds with a packet that has the acknowledge number $= S_B + 1$ and both the *syn* and *ack* flags set.
- Client B receives the synchronisation request from A and responds with a packet that has the acknowledge number $= S_A + 1$ and both the *syn* and *ack* flags set.

To represent the above specification, we use the syntax form (2) as follows:

$$\begin{array}{l} \Box((\forall x_1, x_2, x_3, x_4, S_A, S_B) \\ ((\exists y_6, y_7, y_9, y_{10}, y_{11}, y_{12}) \\ \quad P(x_1, x_2, x_3, x_4, S_A, y_6, y_7, 1, y_9, y_{10}, y_{11}, y_{12}) \rightarrow \\ \Box_{[0,1]}((\exists w_6, w_7, w_9, w_{10}, w_{11}, w_{12}) \\ \quad P(x_3, x_4, x_1, x_2, S_B, w_6, w_7, 1, w_9, w_{10}, w_{11}, w_{12}) \rightarrow \\ \Box_{[0,1]}((\exists z_9, z_{10}, z_{11}, z_{12}) \\ \quad P(x_1, x_2, x_3, x_4, S_A, S_B + 1, 1, 1, z_9, z_{10}, z_{11}, z_{12}) \rightarrow \\ \Diamond_{[0,1]}(\exists k_9, k_{10}, k_{11}, k_{12}) \\ \quad P(x_3, x_4, x_1, x_2, S_B, S_A + 1, 1, 1, k_9, k_{10}, k_{11}, k_{12}))))) \end{array}$$

This is negated as follows (3):

$$
\begin{array}{l}
((\exists x_1, x_2, x_3, x_4, S_A, S_B) \\
\Diamond_{[0,1]}((\exists y_6, y_7, y_9, y_{10}, y_{11}, y_{12}) \\
\quad P(x_1, x_2, x_3, x_4, S_A, y_6, y_7, 1, y_9, y_{10}, y_{11}, y_{12}) \wedge \\
\Diamond_{[0,1]}((\exists w_6, w_7, w_9, w_{10}, w_{11}, w_{12}) \\
\quad P(x_3, x_4, x_1, x_2, S_B, w_6, w_7, 1, w_9, w_{10}, w_{11}, w_{12}) \wedge \\
\Diamond_{[0,1]}((\exists z_9, z_{10}, z_{11}, z_{12}) \\
\quad P(x_1, x_2, x_3, x_4, S_A, S_B + 1, 1, 1, z_9, z_{10}, z_{11}, z_{12}) \wedge \\
\neg\Diamond_{[0,1]}(\exists k_9, k_{10}, k_{11}, k_{12}) \\
\quad P(x_3, x_4, x_1, x_2, S_B, S_A + 1, 1, 1, k_9, k_{10}, k_{11}, k_{12})))))
\end{array}
$$

Next we show how this is translated into Stream SQL queries.

7 Formula Mapping

We show how to translate the two main forms for abnormal behaviour, from the negation of normal behaviour defined in Section 6, into SSQL. Single step anomalies are translated using the *filter* operator which filters relevant packets according to particular conditions. Multiple step anomalies use the *pattern* operator which allows the specification of a sequence of packets with timing constraints and particular conditions between them. Anomalies relating to a count of particular packets received within some time frame could be translated using the *aggregate* operator.

7.1 Single Step Anomalies

These are abnormal behaviour represented by conditions within one packet of the form

$$\varphi \wedge \psi$$

where:

- φ is first order predicate (representing a packet).
- ψ is Boolean combination of atomic formulae.

Some pre-precessing takes place to read packets from the network and identify the relevant fields and to set up the relevant input (`inputstream`) and output (`outputstream`) streams. The second stage is the mapping process. For the mapping, we define the mapping function (M1) which maps a subset of *MSFOMTL* (Δ) into a subset of SSQL (Θ):

$$\text{M1} : \Delta \longrightarrow \Theta.$$

M1($\varphi \wedge \psi$) $\mapsto$

```
SELECT * FROM inputstream
WHERE M1'(ψ)
INTO outputstream;
```

and
M1'($\psi_1 \wedge \psi_2$) $\mapsto$ M1'(φ) and M1'(ψ)
M1'($\psi_1 \vee \psi_2$) $\mapsto$ M1'(φ) or M1'(ψ)
M1'($te_1 \boxdot te_2$) $\mapsto$ M1'(te_1) $\boxdot$ M1'(te_2)
M1'(c_i) $\mapsto c_i$
M1'(x_i) $\mapsto xi$
where parenthesis are preserved and in the above te_1, te_2 are terms, c_i is a constant, x_i is a variable, and

$$\boxdot \in \{=, <>, >, <, >=, <=, +, -, *, / \}.$$

Example 4. We show how to translate Example 1 into SSQL. We take the negated formula specification, repeated below and translate it into SSQL.

$$((\exists x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12})$$
$$P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \wedge$$
$$(((x_2 \leq 0) \vee (x_2 \geq 65535)) \vee ((x_4 \leq 0) \vee (x_4 \geq 65535))))$$

The mapping to SSQL using the mapping function M1 will be :
M1(ϕ) =

```
SELECT * FROM inputstream
WHERE (((x2 ≤ 0) or (x2 ≥ 65535)) or
    ((x4 ≤ 0) or (x2 ≥ 65535)))
INTO outputstream;
```

7.2 Multiple Step Anomalies

Here an abnormal sequence of packets are represented

$$\varphi_1 \wedge \Diamond_{[t_1,t_2]}\psi \quad \text{or} \quad \varphi_1 \wedge \neg\Diamond_{[t_1,t_2]}\varphi_2$$

where:

- φ_1 and φ_2 are first order predicates (representing packets);
- ψ is the same form as the above.

Similar pre-processing takes place as in the single packet case. An additional preprocessing step will split the input into two streams, one relating to each conjunct, by filtering the main stream by the contents of the constant values of each predicate. The code template for this is as follows.

```
CREATE STREAM Filter1;
    // creates input stream for first conjunct
CREATE STREAM Filter2;
    // creates input stream for second conjunct
SELECT * FROM inputstream
WHERE xi = cj [and ...]
    // condition(s) on constant value(s)
    // in the first conjunct
INTO Filter1
WHERE xi = ck [and ...]
    // condition(s) on constant value(s)
    // in the second conjunct
INTO Filter2;
```

For the mapping, we define the mapping function (M2) that maps a subset of *MSFOMTL* ($\mathit{\Delta}$) into a subset of SSQL ($\mathit{\Theta}$):

$$\mathrm{M2} : \mathit{\Delta} \longrightarrow \mathit{\Theta}$$

The mapping process to the pattern constructor deals with at least two predicates. We will assume that start of any timing constraints (below denoted t_1) is 0. We can define the basic elements of the mapping function M2 as follows:

$\mathrm{M2}(\varphi \wedge [\neg]\Diamond_{[t_1,t_2]}\psi) \mapsto$

```
  SELECT input1.x1,...,input1.xn,  // input1
         input2.x1,...,input2.xn,  // input2
  FROM PATTERN (Filter1 as input1 THEN
               [NOT] Filter2 as input2)
```

  WITHIN $\mathrm{M2}([t1, t2])$ TIME
  WHERE $\mathrm{M2}(\varphi, \psi)$
  INTO output;

where if the second conjunct is negated then NOT is present in the fourth line above and

$\mathrm{M2}([t_1, t_2]) \mapsto t_2 - t_1$
$\mathrm{M2}(\varphi, \psi) \mapsto \mathrm{conj}(\{$input1.xi = input2.yj $|$
for all x_i in φ, for all y_j in ψ, such that $x_i = y_j\})$

where

$$\mathrm{conj}(A) = a_1 \texttt{ and } a_2 \texttt{ and } \ldots \texttt{ and } a_n \qquad \text{for all } a_i \in A.$$

A justification of the correctness of the translation is provided in [1]. Essentially conditions on packets are translated using WHERE syntax, and timing constraints on subsequent packets using the PATTERN and WITHIN...TIME syntax.

8 Sample Results

Rather than testing on a real high-speed network, we make use of the US Defence Advanced Research Projects Agency (DARPA) publicly available IDS evaluation data sets [10, 9]. This means we know the types of attack in the data, it allows us to easily repeat experiments and experiment with speeding up the replay of the data. We demonstrate results for one of each type of attack.

– *Example 1* The translated code for Example 1 was run against the test DARPA data files. Two anomalies were raised where the TCP port of 0 was used. They also belong to an already known signature-based attack in *SNORT*, with *SNORT* identification sid-524[8]. This attack uses TCP port 0 for scanning target machines. Port 0 is outside range of normal specification and this is why it was caught in our experiments.
– *Example 3* The formula in this example specifies expected behaviour which means that it will raise an alert only when the fourth packet (or the fourth step) of the simultaneous handshake is missing. This formula was translated into SSQL code and tested using the data file mentioned above. No alert or alarm was raised. The data file has a total of 88458 TCP connection records, but the simultaneous way of TCP connection did not occur.
– *Example 3 (cont.)* Note when we implemented a version of Example 3 with weaker requirements, meaning that an alert will be raised whenever a step (2, 3, or 4) is not completed. The result was a total of 88458 alerts. This is because all the connections in the test data file used the three way handshake of TCP connection rather than the simultaneous way. The three way TCP handshake and the simultaneous TCP handshake share the first step but they differ in the second step. This illustrates that this would not be a good representation of normal behaviour and shows how it is important to specify normal behaviour carefully to avoid such issues.

9 Related Work

Using temporal logic as a specification language for the Intrusion Detection Systems is not new. It has been used at least in such IDS as *MONID* [19] and *ORCHIDS* [22]. In *MONID*, temporal logic is used to represent a safety formula ϕ (specification of the absence of an attack) and the system continuously evaluates ϕ against a model $\mathcal{M}$ representing a finite sequence of events. Whenever ϕ is violated (i.e. $\mathcal{M} \not\models \phi$) an intrusion alarm is raised. In that sense it is close to our use of temporal logic for the protocol anomaly detection we presented in this paper.

ORCHIDS [22] is a signature based intrusion detection tool which uses temporal logic to define attacks that are complex, correlated sequences of events.

The main differences between the system we described in this paper as compared with *MONID* and *ORCHIDS* (apart from some variations in the temporal logic language) is that here checking for abnormal behaviour ϕ in some model $\mathcal{M}$ ($\mathcal{M} \models \phi$) is reduced to the stream query evaluation (instead of using custom execution mechanisms), which is subsequently executed by high-performance *SDP* engine. In the proposed system, the way of using temporal logic takes advantage of its expressiveness and conciseness to allow the user to express attack signatures transparently and independently from the underlying technical implementations.

[8] www.snort.org/search/sid/524

SNORT is an open source network intrusion detection and prevention system. It performs packet level traffic monitoring and analysis. Packets are examined for matching attack signatures. *SNORT* includes a library of attack signatures known as *SNORT* rules and an attack specification language allowing users to add their own rules. Bro is a network monitoring system that can be used for network IDS. It has its own scripting language that allows the specification of custom written attacks and can be used to detect of multiple step attacks.

TeStID [2, 3] was originally developed for signature-based intrusion detection. It was shown to detect attacks efficiently in very high-speed networks, outperforming *SNORT* and Bro, whilst also having an elegant and concise representation of attacks, in particular of multi-step attacks, via temporal logic.

10 Conclusions and Future Work

We have proposed the concept of Temporal Stream Processing for Cyber Security here using Temporal Logic to specify (parts of) the normal behaviour and their translation into a stream data query and execution via a data stream engine. In this paper we focused on anomaly based IDS and gave examples of partial specifications of TCP. We showed their translation into Stream SQL queries and gave details of their execution on a data set.

This is only a starting point showing a proof of concept. Further work must be carried out with the formalisation a larger portion of TCP and experimenting further with TeStID using this. This will allow better experimentation with the performance of the system in high speed networks and comparison with other tool similar to what we have considered for signature-based IDS [3].

Other protocols could also be formalised. Rather than considering protocol anomaly we can investigate *behaviour anomaly* at higher levels of abstraction than considered at protocol level where "normal" means either following some specifications/conventions, or just the users choice. Additionally we can use the approach to traffic transformation by normalising network traffic to deal with attacks stemming from out of order packets or fragmentation on one hand or the need to aggregate/abstract on the other.

In previous work we applied this approach to signature based IDS [2, 3]. We believe this approach can be applied to other cyber security tasks and will form the basis for a new generation of cyber security tools, including those for IDS and IPS, with the novelty in combining the simplicity, conciseness and clarity of temporal logic with the high speed and volume data stream engines.

Acknowledgements

The first author was supported by the Saudi Ministry of Higher Education. Support was received by Stream Base Inc. in providing the necessary license to use their products in this work.

References

1. Ahmed, A.: Online Network Intrusion Detection Using Temporal Logic and Data Stream Processing. Ph.D. thesis, Department of Computer Science, University of Liverpool (2013)
2. Ahmed, A., Lisitsa, A., C.Dixon: A Misuse-Based Network Intrusion Detection System Using Temporal Logic and Stream Processing. In: Proc. of the 5th International Conference on Network and System Security. pp. 1–8. IEEE (2011)
3. Ahmed, A., Lisitsa, A., C.Dixon: TeStID: A High Performance Temporal Intrusion Detection System. In: Proc. of The Eighth International Conference on Internet Monitoring and Protection (ICIMP). IARIA XPS Press (2013)
4. Alur, R., Henzinger, T.: Logics and models of real time: A survey. In: Proceedings of the 1991 REX Workshop on Real Time: Theory in Practice. LNCS, vol. 600, pp. 74–106. Springer (1992)
5. Andrade, H., Gedik, B., Turaga, D.: Fundamentals of Stream Processing Application Design, Systems, and Analytics. Cambridge University Press (2013)
6. Bishop, S., Fairbairn, M., Norrish, M., Ridge, T., Sewell, P., Smith, M., Wansbrough, K.: Engineering with Logic: Rigorous Specification and Validation for TCP/IP and the Sockets API. http://www.cl.cam.ac.uk/~pes20/Netsem/paper3.pdf, Accessed April 2015
7. Chakravarthy, S., Jiang, Q.: Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing. Springer (2009)
8. Cotton, M., Touch, L.E.J., Westerlund, M., Cheshire, S.: Request for Comments: 6335. https://tools.ietf.org/html/rfc6335 (2011), Accessed April 2015
9. Cunningham, R., Lippmann, R., Fried, J., Garfinkel, S., Kendall, R., Webster, S., Wyschogrod, D., Zissman, M.: Evaluating Intrusion Detection Systems without attacking your Friends: The 1998 DARPA Intrusion Detection Evaluation. Technical report, Defense Advanced Research Projects Agency, Department of US Defense (1998)
10. DARPA Intrusion Detection Evaluation. http://www.ll.mit.edu/mission/communications /ist/corpora/ideval/ (2012)
11. Das, K.: Protocol Anomaly Detection for Network-based Intrusion Detection. SANS Institute InfoSec Reading Room http://www.sans.org/reading_room/whitepapers/ detection/protocol_anomaly_detection_for_networkbased_intrusion_detection_349 (2002), Accessed April 2015
12. Endace: Cyber security: What every organization needs to know about network monitoring and recording. Printed materials distributed by Endace Ltd at Cyber Security Meeting, 11.11.2010, London (2010)
13. Haggerty, J., Shi, Q., Merabti, M.: Statistical signatures for early detection of flooding denial-of-service attacks. In: Sasaki, R., Qing, S., Okamoto, E., Yoshiura, H. (eds.) Security and Privacy in the Age of Ubiquitous Computing, IFIP Advances in Information and Communication Technology, vol. 181, pp. 327–341. Springer US (2005), http://dx.doi.org/10.1007/0-387-25660-1_22
14. Transmission Control Protocol Specification. https://tools.ietf.org/html/rfc793 (1981), Accessed April 2015
15. Kang, D.H., Kim, B.K., Oh, J.T., Nam, T.Y., Jang, J.S.: FPGA based intrusion detection system against unknown and known attacks. In: Shi, Z., Sadananda, R. (eds.) Agent Computing and Multi-Agent Systems, LNCS, vol. 4088, pp. 801–806. Springer (2006), http://dx.doi.org/10.1007/11802372_97

16. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2(4), 255–299 (1990)
17. Manzano, M.: Introduction to many-sorted logic. John Wiley & Sons, Inc., New York, NY, USA (1993)
18. Miller, Z., Deitrick, W., Hu, W.: Anomalous network packet detection using data stream mining. J. Information Security 2(4), 158–168 (2011)
19. Naldurg, P., Sen, K., Thati, P.: A temporal logic based framework for intrusion detection. In: Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems. Madrid, Spain (2004)
20. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. http://www.ietf.org/rfc/rfc2267.txt (1998), Accessed April 2015
21. Neumann, P., Porras, P.: Experience with EMERALD to date. In: Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring - Volume 1. pp. 8–8. ID'99, USENIX Association, Berkeley, CA, USA (1999), http://dl.acm.org/citation.cfm?id=1267880.1267888
22. Olivain, J., Goubault-Larrecq, J.: The ORCHIDS intrusion detection tool. In: Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05) (2005)
23. Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., Zhou, S.: Specification-based anomaly detection: A new approach for detecting network intrusions. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 265–274. CCS '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/586110.586146
24. Stakhanova, N., Basu, S., Wong, J.: On the symbiosis of specification-based and anomaly-based detection. Computers and Security 29(2), 253–268 (2010), http://dblp.uni-trier.de/db/journals/compsec/compsec29.html#StakhanovaBW10
25. Touch, J., Lear, E., Mankin, A., Kojo, M., Ono, K., Stiemerling, M., Eggert, L., Melnikov, A., Eddy, W., Zimmermann, A., Kohler, E., Mankin, A.: Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt (2015), Accessed April 2015
26. Windows Server 2003 and XP SP2 LAND attack vulnerability. http://www.securityfocus.com/archive/1/392354 (2005), Accessed April 2015
27. Wright, G., Stevens, W.: TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley (1995)