TeStID: A High Performance Temporal Intrusion Detection System

Abdulbasit Ahmed, Alexei Lisitsa, and Clare Dixon
Department of Computer Science
University of Liverpool
Liverpool, United Kingdom
{Aahmad, Lisitsa, CLDixon}@liverpool.ac.uk

***Abstract*—Network intrusion detection systems are faced with the challenge of keeping pace with the increasingly high volume network environments. Also, the increase in the number of attacks and their complexities increase the processing and the other resources required to run intrusion detection systems. In this paper, a novel intrusion detection system is developed (TeStID). *TeStID* combines the use of high-level temporal logic based language for specification of attacks and stream data processing for actual detection. The experimental results show that this combination efficiently make use of the existing testing machine resources to successfully achieve higher coverage rate in intensive network traffic compared with *Snort* and *Bro*. Additionally, the solution provides a concise and unambiguous way to formally represent attack signatures and it is extensible and scalable.**

***Keywords*-network intrusion detection system; temporal logic; parallel stream processing; runtime verification**

I. Introduction

Intrusion Detection Systems (IDS) detect intruders' actions that threaten the confidentiality, availability, and integrity of resources [1]. Network Intrusion Detection Systems (NIDS) reside on the network, and are designed to monitor network traffic. An *NIDS* examines the traffic packet by packet in real time, or close to real time, to attempt to detect intrusion patterns [2].

The increasing network throughput challenges the current *NIDS* running on customary hardware to monitor the network traffic without dropping packets. Consequently, many attacks are not detected by the current *NIDS* [3], [4]. Some vendors provide a highly specialized and configured hardware to prevent the drop of packets [5], [6].

The techniques developed to solve the problem of IDS reliability due to packets loss in high-speed networks can be grouped into the following:

- Data reduction techniques (i.e., data based approach) [7].
- Load balancing, splitting, or parallel processing of traffic (i.e., distributed/parallel execution based approach) [3].
- Efficient algorithms for pattern matching (i.e., algorithm based) [8].
- Hardware based approach such as using graphics processing units [9] or field-programmable gate array (FPGA) devices [10].

Some works use combinations of the above techniques as in [8] where the algorithm is hardware implementable. The research area is still active and there is no solution that can keep up with the increase in bandwidth.

Another issue with current IDS is that the attack signatures are often specified in rather low-level languages and, especially, in the case of multiple packets attacks, may become cumbersome and error-prone, difficult to write, analyse and maintain. For instance, *Bro* [11] has special scripting language for detecting multiple packet attacks.

In this paper, we present a new Temporal Stream Intrusion Detection System (TeStID), the design of which addresses both the issue of efficiency and reliability in high-speed networks and the issue of the high-level and unambiguous specifications. In *TeStID* a multiple packet attack is represented with a formula, whereas in *Bro* one page of code or more might be needed to represent the same attack. The system uses Temporal Logic (TL) [12] for the attack signature specifications and available Stream Data Processing (SDP) [13]–[15] technology for the actual detection. *TL* is the extension of classical logic with operators that deal with time which allow us to formally specify temporal events (i.e., network packets) as they traverse the network. The *SDP* is a database technology applied to streams of data which are designed with processing capabilities suitable for data intensive applications.

We use Many Sorted First Order Metric Temporal Logic (MSFOMTL), which was defined in [16] to represent data packets arriving over time and attack patterns. This allows us to state, for example, "a packet arrives between 6 and 9 seconds".

In [16], we described the architecture of *TeStID* and provided preliminary experimental results using the DARPA IDS Evaluation Data [17]. The experiments and case studies focussed on the detection of multiple packet attacks.

In this paper, we extend our work and provide further detailed experimental analysis considering both single and multiple packet attacks, allowing single packet attacks with payload and experimenting with the high performance features available in stream data processing. Additionally, we compare the performance of TeStID with two well known open source NIDSs: *Snort* [18] and *Bro* [11].

The rest of the paper is organized as follows. Section

II presents an overview of the proposed system and its design. Section III presents the experiment setup environment. Section IV presents the experimental results with and without using the high performance features of stream data processing. Related work is presented in Section V. Finally, Section VI concludes this paper and discusses future work.

II. TeStID Overview and Design

In *TeStID* attack signatures are formally represented using temporal logic. The signature based intrusion detection problem is reduced to the problem of checking whether a temporal formula ϕ representing an attack pattern s true in a temporal model M representing a linear sequence of all received (observed) network packets, that is $M \models \phi$?

In our model, we are dealing with finite initial segments of potentially infinite sequences. This and the fact that properties considered are time bounded makes the decidability of model checking trivial.

In the TeStID system the TL formulae specifying attacks are automatically translated into stream SQL (SSQL) language constructors. We use StreamBase [15] as the stream data base engine and it uses *SSQL* as the stream query language. Consequently, this *SSQL* code is executed to detect temporal patterns specified in the original formula in the incoming events.

The syntax of temporal logic *MSFOMTL* used in the system is as follows. An atomic formula has the form $P(te_1, te_2, \ldots, te_n)$ where P is a predicate and for $i = 1, \ldots, n, te_i$ is a term. The atomic formulae represent the information about individual packets and terms correspond to the fields within a packet.

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

$$\mathcal{L} := P \mid \varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Diamond_{[t_1,t_2]} \mid \Box_{[t_1,t_2]} \mid \blacklozenge_{[t_1,t_2]}\varphi \mid \blacksquare_{[t_1,t_2]}\varphi \mid (\forall x)\varphi \mid (\exists x)\varphi \tag{1}$$

In addition to the usual connectives of propositional logic these formulae include bounded temporal operators "always in the time between t_1 and t_2" $\Box_{[t_1,t_2]}$, "eventually in the future in a time between t_1 and t_2" $\Diamond_{[t_1,t_2]}$, "sometime in the past in a time between t_1 and t_2 before now" $\blacklozenge_{[t_1,t_2]}$, and "always in the past in all time between t_1 and t_2" $\blacksquare_{[t_1,t_2]}$. The incoming events form the temporal models, $\mathcal{M} = \langle \mathcal{T}, <, \mathrm{I} \rangle$ where:

- $\mathcal{T} = \{\tau_0, \tau_1, \ldots\} \subset \mathbb{R}^+$, where $\mathbb{R}^+$ is a non-empty set of positive real numbers and $\mathcal{T}$ is the set of all arrival moments.
- $<$ is a linear order on $\mathcal{T}$.
- I is an interpretation which maps $\mathcal{T}$ into the set of all possible packets $[\![\mathcal{P}]\!]$: $\mathrm{I} : \mathcal{T} \rightarrow [\![\mathcal{P}]\!]$

So, $\mathrm{I}(\tau_i)$ represents a packet arriving at a moment $\tau_i \in \mathcal{T}$.

III. The Experiment Setup

The setup of the test environment closely resembles the actual deployment of NIDS. In a typical NIDS deployment, the network sensor device receives a copy of all the traffic that traverse the network. The testing environment is setup as follows:

- *TeStID* is installed on an INTEL® Core$^{\text{TM}}$ i5 2.26 GHz machine with 4 GB of memory and a Gigabit Network interface that is capable of running in promiscuous mode (i.e., listening to all network traffics). *TeStID* was developed with *StreamBase* developer version 7.1.
- Another computer (INTEL® Core$^{\text{TM}}$2 Quad Processor Q6600 and 2 GB memory) with a Gigabit Network interface is used to replay the data.
- TCPREPLAY [19] was used to replay the trace files. TCPREPLAY is a tool that replays TCP dump files [20] at specified speeds onto the network.
- A switch to connect the two PCs or simply crossover network cable.
- A custom data file, which was prepared using a DARPA dump test file and some test files from the free license version of Traffic IQ Professional$^{\text{TM}}$ [21].
- *Snort* version 2.9.1, and *Bro* version 1.5.1.

The data file has 3,014,600 packets. It has 50 different single packet with payload attacks which are distributed over 792 packets. This means 792 total instances of the 50 attacks.

During the experiments, the send and receive buffers were increased to ensure that all packets are sent successfully and are received with no loss of packets. The reading buffer parameters rmem_max and rmem_default increased significantly to 110 MB. Also, the writing buffer buffer parameters wmem_max and wmem_default increased to the same value. Using these values, TCPREPLAY successfully replayed the dump files at top speed multiple, that is, the maximum that can be send on the designated hardware. On the receiving end, TCPDUMP successfully captured all the packets. This was important tuning step as *Snort*, *Bro*, and *TeStID* use TCPDUMP to sniff packets.

IV. The Experimental Work

The experiments were run on *Snort*, *Bro*, and *TeStID* using the test data file. These attacks were coded in *Snort* and *Bro* according to syntax specified in their documentations [11], [18]. For *TeStID* these attacks are written in a file using *MSFOMTL* syntax. Then the translator is run to translate these formulae into *SSQL* code. A typical example, is *Snort* attack id 255 "DNS Zone Transfer TCP". This attack is identified when the destination port (x_4) is 53 and the payload (x_{12}) contains a string that is identified by the regular expression `".{14}.*\u0000\u0000\u00fc.*"` which means the string must contains any 14 characters, possibly followed by an additional character, followed by two null characters, and "ü". Formally, it is represented in *MSFOMTL* as follows:

$$
\begin{gathered}
(\exists x_4, x_{12})((\exists y_1, y_2, y_3, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}) \\
P(y_1, y_2, y_3, x_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}, x_{12}) \wedge (x_4 = 53) \\
\wedge (x_{12} = f(\texttt{".\{14\}.*\textbackslash u0000\textbackslash u0000\textbackslash u00fc.*"})))
\end{gathered} \tag{2}
$$

The meaning of the terms y_1,y_2,y_3,y_5,y_6,y_7,y_8,y_9,y_{10},y_{11} are source IP address, source port, destination IP address, sequence number, acknowledgement number, ack flags, syn flag, reset flag, push, and urgent flag, respectively. These are free terms and can take any valid value from its corresponding sort domain.

When translating the above formula into *SSQL* we obtain the following code:

```
CREATE STREAM out_Input;
APPLY JAVA "TCP_W_Payload" AS Input (
 schema0 = "<?xml version=\"1.0\" encoding=
 \"UTF-8\"?>\n<schema name=\"schema:Input\">
\n
<field description=\"\" name=\"x1\"
type=\"string\"/>\n
.
.
<field description=\"\" name=\"x12\"
type=\"string\"
/>\n</schema>\n"
)
INTO out_Input;

CREATE OUTPUT STREAM Output;
SELECT * FROM out__Input
WHERE (out_Input.x4 = 53)
and (regexmatch(".{14}.*\u0000\u0000\u00fc.*"
     , x12))
INTO Output;
```

Each of the 50 attacks produce similar code to the above. There are two main parts in each code: the input adapter call (APPLY JAVA) and the select statement (from here onward we refer to it as the filter code).

A. The Experiments Results: multiple packet attacks

The results of experiments on multiple packet attacks using *TeStID* were presented previously in [16]. The results obtained is reproduced in Table I. The first column contains the attack names, the second column contains the DARPA [22] data files used, the third column contains the number of attacks detected in normal replay, the fourth column contains the number of attacks detected in 1350 speed multiple of normal recorded speed (the highest speed achieved without packet loss), and the last column is the actual number of attacks in the data. The bandwidth achieved is 524 Mbits/Sec and the peak number of packets was about 250,000 packets per second. In *Snort* the specification of multiple packet attacks is not possible but in *Bro* it is feasible. These attacks need to be written in *Bro* scripting language. We ran the script for the Syn Flood or Neptune which comes with *Bro* default installation. *Bro* achieves about 300 Mbits/Sec and about 76,000 packets/Sec using the same test data file in Table I and the same testing environment. *Bro* crashed within seconds at the speed multiple of 1350 after consuming the available resources on the testing machine.

Table I. Results of Multiple Packet Attacks

Attack Name	Data Set	Packets Replayed	Normal Replay	1350X Normal	Actual No. of Attacks
DoSNuke	01/04/99 in	2,356,503	1	1	1
	05/04/99 in	2,291,319	2	2	2
	06/04/99 in	3,404,824	1	1	1
Neptune	05/04/99 in	2,291,319	1	1	1
	06/04/99 out	2,558,481	3	3	3
	09/04/99 in	3,393,918	1	1	1
TCPReset	06/04/99 in	3,404,824	2	2	2
	07/04/99 in	2,087,942	1	1	1
	09/04/99 in	3,393,918	1	1	1
ResetScan	08/04/99 in	3,201,381	2	2	2

In this paper, the experiments on *TeStID* are further extended to explore its capabilities to detect single packet attacks with payload. The experiments on these attacks can be grouped into two sets. In the first set, no high performance features are used. In the second set the high performance features are used. The following sections give more detail and the results of each set.

B. The Experiments Results: single packet attacks

Here, we present the results of the experiments with the detection of single packet attacks with payload. In this type of attacks the packets are inspected deeply and not only the headers as in the multiple packet attacks. Three different variants of the execution of *SSQL* code were considered.

In *StreamBase* a basic execution unit running on the server is called a container. The *SSQL* codes run inside this container which has a name and a set of associated handling processes that are created by the stream server. In this set of experiments there are three possible implementations to run single packet attack detection on the *StreamBase* server without using the high performance features.

- In the first variant of implementation, each translated specification of an attack is written as an independent program that runs in its own container.
- In the second implementation each translated specification of an attack is an independent program but all programs are using the same container.
- In the third implementation all the translated specifications of the attacks are written as one program and run in one container sharing the input adapter.

The results obtained for this set of experiments are shown in Table II. Each experiment is an average of three runs and before each run the machine was restarted. The first column shows the tested implementation. Columns two through four contain the results of running each system while using multiple replay speeds of 2, 4, and 8 respectively, to send the packets (blank if no test is done).

Table II. results without using high performance features

Implementation	X 2	X 4	X 8
50 independent programs files	-6	-139	-384
50 programs in one file	0	-116	-323
50 programs sharing input adapter	0	-6	-74
BRO v1.5.1	0	0	-1
SNORT v2.9.1	-119	-125	

In the first row, 50 different program files are run where each program corresponds to one attack. This implementation gives the worst coverage rate as it misses six attacks in the speed multiple of two (x 2 column). The result makes sense as each program needs to capture the packets with the input adapter code which in turns calls a JAVA code that interfaces with the network interface and then the rest of the program processes the packets (tuples) in sequential fashion. This scenario is the same for all the 50 programs which means each program will have its own space (parallel region or container in *StreamBase* terminology). The *StreamBase* engine suffers from the overhead of having to interact with all these regions.

The second row shows the results of running the code for all 50 attacks in one container (i.e., the attacks are all written in one file). This implementation gives slightly better results as it misses 116 attacks in the speed multiple of four. With this implementation the *StreamBase* engine achieves more efficient processing with the decrease of the inter process handling overhead.

In the third row, all the 50 filter codes are in one file but only one input adapter code is used. All the filtering codes read from the same input adapter. This means less resources from the system are used. In *StreamBase* when a tuple (packet) arrives it must be processed till completion before the next tuple arrives. This means the input adapter feeds the tuple to the first coded filter and then coded map of the first attack. Then the codes of the second and so on. If another tuple (packet) arrived, it will be retained in a buffer until the first processing is finished. This type of implementation was the best without the use of concurrency and multiplicity options as it misses only 6 attacks at the multiple speed of four. Unfortunately, this is worse than what *Bro* achieved in the fourth column. *Bro* misses only one attack in the speed multiple of eight. The *Snort* result is shown in the fifth column and it misses 119 attacks in the speed multiple of two. It was worse in speed multiple of four as it missed 125. The blank in the table for speed multiple of eight for *Snort* means that test was not done as we thought it is not necessary.

These results gave the motivation to investigate the high performance features of *StreamBase* and thus a second set of experiments were carried out as described in the next section.

V. Investigating High Performance Features

In general, stream data processing engines have high performance features. *StreamBase* has concurrency and multiplicity options that can be used to allow us to achieve higher performance. In this set of experiments we use these features. First of all we explain the following related definitions to the high performance features used:

- Concurrency means part of the code runs in its own thread.
- Multiplicity refer to the number of instances of the code.
- Dispatch style is related to the multiplicity. The dispatch style specifies how each instance receives data tuples: in round robin, broadcast, or based on a data value. In broadcast each instance will receive a copy of the incoming tuple. In round robin, the first tuple goes to the first instance and the second goes to the second and so on. Based on value is by checking the value against a test condition and then dispatching to the designated instance for that value.

In *StreamBase* the concurrency, the multiplicity, or both can be set. According to *StreamBase* manual, these options can be used for portions of the application if the code portion is long-running or compute-intensive, can run without data dependencies on the rest of the application, and it would not cause the containing module to be waiting or blocked.

In this set of experiment, each *SSQL* code for detecting an attack in the previous section contains two components: the input adapter and filter code. For the input code, we can use the concurrency option but not the multiplicity as it has no input stream. For the filter code part we can use both options. This means that there are eight possible implementations as can be seen in Figure 1. Table III shows

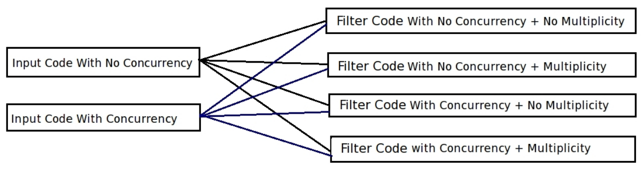
Input Code With No Concurrency
Input Code With Concurrency
Filter Code With No Concurrency + No Multiplicity
Filter Code With No Concurrency + Multiplicity
Filter Code With Concurrency + No Multiplicity
Filter Code with Concurrency + Multiplicity

Figure 1. All Possible Implementations Using Concurrency/Multiplicity

the results of the experiments of this category. Notice that the experiments start at speed multiple of 8 as we try to achieve better result compare to *Bro* in Table II. Table III (CC) denotes “Concurrency”. (NC) denotes “No Concurrency”, (1) denotes single instance or no multiplicity, and (n) denotes n multiplicity where n is an integer such that $n > 1$. For instance, the first row shows the results of running non concurrent input code (NC) and non concurrent (NC) 50 filter codes of (1) instance each (i.e., no multiplicity). The following are observations on these results:

Table III. Experiments Results Using Concurrency and Multiplicity

	Implementation	X8	X16	X24	X48
1	NC Input Code + 50 NC Filter Code (1)	-79			
2	NC Input Code + 50 NC Filter Code (2)	-67			
3	NC Input Code + 50 CC Filter Code (1)	-293			
4	NC Input Code + 50 CC Filter Code (2)	-299			
5	CC Input Code + 50 CC Filter Code (1)	-126			
6	CC Input Code + 50 CC Filter Code (2)	-128			
7	CC Input Code + 50 NC Filter Code (1)	0	0	0	-5
8	CC Input Code + 50 NC Filter Code (2)	0	0	0	-2
9	CC Input Code + 50 NC Filter Code (3)	0	0	0	-1
10	CC Input Code + 50 NC Filter Code (5)	0	0	0	-2
11	CC Input Code + 50 NC Filter Code (10)	0	0	0	-5

- The first row result is almost the same result obtained previously with no concurrency and no multiplicity (the third row in Table II). The difference is that we implement the attack code as a module and we did not use the concurrency or the multiplicity.
- Using 50 concurrency for the filter code give the worst result (row 3-6).
- Using input code with concurrency and no concurrency for the filter code gives better results in general (rows 7-11).
- The reason for the bad performance results when using 50 CC (row 3-6) compared with using 50 NC for the filter code (row 7-11) is that the system running in the CC implementation maintains many thread-switches per attack vs. no switch per attack with NC implementations.
- Row number 9 has the result of best performance where three non concurrent instances of filter codes are used. This is consistent with *StreamBase* rule of thumb that is for best performance the number of instances should be equal to the number of cores on the machine or less. So, we have three instances in addition to the input code running on four cores on the testing machine.
- In rows 7-8 less than three number of instances used and in rows 10-11 more than three instances used. Increasing or decreasing the number of instances from three cause the performance to degrade.

This set of experiments using the concurrency and multiplicity features of *StreamBase* enable us to achieve a result that exceeded the results of *Snort* and *Bro* which are presented in Table II. Furthermore, the testing machine has four cores, but the solution can take advantage of using more cores. *Snort* doe

VI. Related Work

Temporal logic is used in the network based IDS *MONID* [23] and *ORCHIDS* [24]. *MONID* a prototype tool based on *Eagle* [25]. *Eagle* is a runtime verification or runtime monitoring system that uses finite traces. Simply, it monitors the execution of a program and checks its conformity with a requirements specification, often written in a temporal logic or as a state machine. Naldurg, Sen, and Thati [23] propose the use of *Eagle* in online intrusion detection systems. In *MONID*, *TL* is used to represent a safety formula ϕ (specification of the absence of an attack) and the system continuously evaluates ϕ against a model M representing a finite sequence of events. Whenever ϕ is violated (i.e., $M \not\models \phi$) an intrusion alarm is raised.

ORCHIDS [24] is a misuse intrusion detection tool, capable of analyzing and correlating temporal events in real time. *ORCHIDS* uses an online model checking approach. *ORCHIDS* uses temporal logic to define attacks that are complex, correlated sequences of events, which are usually individually benign. The attack signatures are represented or described in the system as automata. The *ORCHIDS* online algorithm matches these formulae against the logs and returns enumerated matches [24]. Similar to *MONID*, *ORCHIDS* reads events from many sources that have different formats (networks and system logs) and this makes representing attack signatures difficult in standard methods. As far as we know from published paper [24], the tests for *ORCHIDS*, mainly concentrated on the proof of concept and not performance.

The main differences between the system we described in this paper as compared with *MONID* and *ORCHIDS* is that the model checking problem ($M \models \phi$) is reduced to the stream query evaluation, which is subsequently executed by high-performance *SDP* engine. Also, 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. We could not test *MONID* and *ORCHIDS* in the same testing environment we used in our experiments, as their implementations were not available.

Snort [18] and *Bro* [11] are the oldest and most popular open source NIDSs known today. According to the *Snort* organization site, over 4 million downloads and more than

400,000 registered users show the wide popularity of *Snort* as a deployed IDS solution. Specifying multiple packet attacks is not possible in *Snort*, but it is possible in *Bro*. *Bro* is considered highly stateful (i.e., it keeps track of each established session states) and was developed primarily as a research platform for intrusion detection and traffic analysis. It has no subscription service where users can download new attack signatures like what the *Snort* community provides.

Data stream processing was suggested to be used in intrusion detection by the *StreamBase* Event Processing Platform™ software development team. An example of using *SB* in intrusion detection was given in the demo package in which a statistical method was used to predict anomaly behaviour of network traffic.

GIGASCOPE [26] is a proprietary data stream management system (DSMS) and was developed by AT&T and it is currently used in many AT&T network sites. *GIGASCOPE* is special purpose DSMS engine for detailed network applications. Johnson, Muthukrishnan, Spatscheck, and Srivastava [27] from AT&T research lab argued that *GIGASCOPE* can serve as the foundation of the next NIDSs because of the functionality and performance. They presented some written examples to detect Denial of Service attacks.

VII. Conclusion and Future Work

The combine use of temporal logic and stream data processing as proposed in this paper is a promising solution toward network intrusion detection system in high-volume network environments. The use of *TL* gives the system the advantage of providing a concise and unambiguous way to represent attacks. Also, using *TL* abstracts the user away from the technical requirements details. In addition, the system is extensible as it is easy to add new attacks and recompile the system to include these.

Using stream data processing gives the system the advantage of having scalable performance. As the results of our experiments suggest the system outperforms both *Snort* and *Bro* systems in high-speed network environments. The system benefits from adding additional CPUs to meet a larger volume of data. To show the feasibility and benefits of using stream processing, the *SB* development version was adequate, even though that it has less capabilities in terms of execution speed and utilizing machine resources than the server version [15].

In the future work, we are planning to extend *TeStID* to be used in *protocol anomaly based* network intrusion detection. *MSFOMTL* will be used to represent parts of the TCP protocol normal specification and any deviation from this specification will be reported at the runtime.

Acknowledgment

The first author would like to thank the Saudi Ministry of Higher Education for partially funding this research. Also, we would like to thank the Stream Base Inc. for giving us the necessary license to use their products in our work.

References

[1] W. Stallings, Network Security Essentials: Applications and Standards. Upper Saddle River, NJ: Prentice Hall, 2000.

[2] K. Scarfore and P. Mell, "Guide to intrusion detection and prevention systems (IDPS)," National Institute of Standards and Technology (NIST), Special Publication 800-94, Feb. 2007.

[3] H. Lai, S. Cai, J. Xi, and H. Li, "A parallel intrusion detection system for high-speed networks," ACNS 2004. LNCS, vol. 3089, 2004, pp. 439–451.

[4] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Operational experiences with high-volume network intrusion detection," in Proceedings of the SIGSAC: 11th ACM Conference on Computer and Communications Security (CSS'04), 2004, pp. 2–11.

[5] Endace Ltd., "EndaceAccess™," http://www.endace.com/, 2012, Accessed on 02 January 2013.

[6] Hewlett-Packard Development Company, L.P., "Tipping Point Digital Vaccine Services," http://h17007.www1.hp.com/docs/security/400931-004_DigitalVaccine.pdf, 2012, Accessed on 02 January 2013.

[7] K.-Y. Lam, L. Hui, and S.-L. Chung, "A data reduction method for intrusion detection," J. Syst. Softw., vol. 33, no. 1, Apr. 1996, pp. 101–108.

[8] S. Dharmapurikar, J. Lockwood, and M. Ieee, "Fast and scalable pattern matching for network intrusion detection systems," IEEE Journal on Selected Areas in Communications, vol. 24, no. 10, Oct. 2006.

[9] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection, ser. RAID '08, 2008, pp. 116-134.

[10] D.-H. Kang, B.-K. Kim, J.-T. Oh, T.-Y. Nam, and J.-S. Jang, in Agent Computing and Multi-Agent Systems, ser. Lecture Notes in Computer Science, Z.-Z. Shi and R. Sadananda, Eds., 2006, vol. 4088.

[11] Lawrence Berkeley National Laboratory, "Bro Intrusion Detection System," http://www.bro-ids.org/, 2011, Accessed on 02 January 2013.

[12] M. Fisher, An Introduction to Practical Formal Methods Using Temporal Logic. Wiley, 2011. [Online]. Available: http://books.google.co.uk/books?id=zl6OLZv7d1kC

[13] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "STREAM: The Stanford stream data manager," in SIGMOD Conference, 2003, p. 665.

[14] EsperTech Inc., "Esper - event stream and complex event processing for java," http://esper.codehaus.org/esper-3.3.0/doc/reference/en/html_single/index.html, 2009, Accessed on 02 January 2013.

[15] StreamBase Systems, "StreamBase Server," http://www.streambase.com/products-StreamBaseServer.htm, 2012, Accessed on 02 January 2013.

[16] A. Ahmed, A. Lisitsa, and C. Dixon, "A misuse-based network intrusion detection system using temporal logic and stream processing," in Proceedings of the 5th International Conference on Network and System Security, P. Samarati, S. Foresti, J. Hu, and G. Livraga, Eds. Milan, Italy: IEEE, 2011, pp. 1-8.

[17] R. Cunningham, R. Lippmann, J. Fried, S. Garfinkel, R. Kendall, S. Webster, D. Wyschogrod, and M. Zissman, "Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation," Defense Advanced Research Projects Agency, Department of US Defense, Technical report, 1998.

[18] Sourcefire, "SNORT," http://www.snort.org/, 2010, Accessed on 02 January 2013.

[19] TRAC, "Welcome to TCPREPLAY," http://tcpreplay.synfin.net/, 2010, Accessed on 02 January 2013.

[20] The TCPDUMP Group, "TCPDUMP and LIBPCAP," http://www.tcpdump.org/, 2010, Accessed on 02 January 2013.

[21] IDAPPCOM Ltd., "Traffic IQ Library," http://www.idappcom.com/index.php, 2012, Accessed on 02 January 2013.

[22] MIT Lincoln Laboratory, "DARPA Intrusion Detection Data Sets," http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html, 1999, Accessed on 02 January 2013.

[23] P. Naldurg, K. Sen, and P. Thati, "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.

[24] J. Olivain and J. Goubault-Larrecq, "The ORCHIDS intrusion detection tool," in Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05), 2005.

[25] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in Proceedings of the VMCAI'04, 5th International Conference on Verification, Model Checking and Abstract Interpretation, Venice, Italy, 2004, pp. 44-57.

[26] C. Cranor, T. Johnson, and O. Spataschek, "Gigascope: A stream database for network applications," in SIGMOD, 2003, pp. 647-651.

[27] T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava, "Streams, security and scalability," in Proceedings of the 19th annual IFIP WG 11.3 working conference on Data and Applications Security, ser. DBSec'05, 2005, pp. 1-15.