Simple Event Correlator - Best Practices for Creating Scalable Configurations

Risto Vaarandi, Bernhards Blumbergs and Emin Çalışkan

This paper has been accepted for publication at the 2015 IEEE CogSIMA conference, and the final version of the paper is included in *Proceedings of IEEE CogSIMA 2015* (ISBN: 978-1-4799-8015-4)

Simple Event Correlator - Best Practices for Creating Scalable Configurations

Risto Vaarandi, Bernhards Blumbergs
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia

Emin Çalışkan
Cyber Security Institute
TÜBİTAK
Kocaeli, Turkey

***Abstract*—During the past two decades, event correlation has emerged as a prominent monitoring technique, and is essential for achieving better situational awareness. Since its introduction in 2001 by one of the authors of this paper, Simple Event Correlator (SEC) has become a widely used open source event correlation tool. During the last decade, a number of papers have been published that describe the use of SEC in various environments. However, recent SEC versions have introduced a number of novel features not discussed in existing works. This paper fills this gap and provides an up-to-date coverage of best practices for creating scalable SEC configurations.**

Keywords—Simple Event Correlator; event correlation; event processing; log file analysis

I. Introduction

During the past two decades, event correlation has become a prominent monitoring technique in many domains, including network fault monitoring, system administration, fraud detection, malicious insider and intrusion detection. Also, event correlation is one of the cornerstones for achieving better situational awareness. In order to address event analysis tasks in various domains, many commercial event correlation solutions have been created. Since its introduction in 2001 by one of the authors of this paper [1], Simple Event Correlator (SEC) has become a widely used open source alternative to commercial offerings. During the last decade, a number of papers have been published that describe the use of SEC in various environments, including academia [2], supercomputing centers [3–6], financial institutions [7, 8], telecom companies [8], and military [9]. SEC has been used for a wide range of purposes, including UNIX server log analysis [2], monitoring of supercomputer clusters [3–5], research experiments [6], correlation of large event volumes in centralized logging infrastructures [7], analysis of various security logs [9–11], IDS alarm classification [12], and network management [1, 8, 13]. However, many past papers have provided generic overviews of SEC deployments, and do not cover its advanced features in sufficient details. Moreover, its recent versions have introduced a number of new features that existing works have not discussed. The current paper fills this gap and provides an up-to-date coverage of best practices for scalable deployment of SEC. The remainder of this paper is organized as follows – section II discusses related work, section III presents recommendations for creating scalable SEC configurations, and section IV concludes the paper.

II. Related Work

Event correlation has received a lot of attention by many researchers, and most papers have adopted the following definition by Jakobson and Weissman [14] – event correlation is a conceptual interpretation procedure where new meaning is assigned to a set of events that happen within a predefined time interval. A number of approaches have been proposed for correlating events, including rule-based [14], graph-based [15], codebook-based [16], and Bayes network based [17] methods. In the industry, event correlation is implemented in most network management and SIEM frameworks, such as HP Openview, Tivoli, ArcSight, and AlienVault. In the open source domain, there are several log monitoring tools with some event correlation functionality – for example, Swatch [18] implements event counting and thresholding operations, while LogSurfer [19] supports pairwise event correlation. Furthermore, NxLog syslog server [20] directly borrows from SEC rule language and re-implements some SEC functionality in its core. ESPER [21] is a development toolkit which allows for augmenting Java and .NET applications with event correlation features. The first papers which provided detailed recommendations on deploying SEC were authored by Rouillard [2] and Vaarandi [10] a decade ago. The treatment by Vaarandi and Grimaila [11] is more recent, but does not address the creation of scalable configurations, and does not describe the new features of the current major release (introduced in 2012). In the following section, we will provide a detailed discussion of these topics.

III. Best Practices and Recommendations

From its inception, SEC was designed to be as lightweight as possible. For this reason, it was implemented as a UNIX tool which incorporates event matching and parsing, event processing, and output generation into a single program. SEC can be used interactively in UNIX shell pipelines, executed as a daemon (or several daemons), connected to other applications over FIFOs, pipes, and network sockets, etc. Other design considerations were platform independence and ease of installation – since SEC is written in Perl and requires no additions to a standard Perl installation, it runs on all modern UNIX and Linux platforms. SEC uses rule-based approach for event correlation, where rules are arranged into sequences (rulesets), with each ruleset stored in a separate text file. Input events can be received from regular files, FIFOs, and standard

input. Input events are typically matched with regular expressions, but for advanced matching and parsing custom Perl functions can be defined. SEC has been designed for real-time event processing only and incoming events are tagged with timestamps of reception. In order to achieve fast memory-based read-write data sharing between rules, event correlation operations, and other SEC entities, SEC has been implemented as a single-threaded tool. Nevertheless, it is straightforward to run many SEC instances with independent rulebases on the same host simultaneously.

A. Joining Rules Into Event Correlation Schemes

A number of web pages and papers provide examples of one SEC rule which correlates events independently. However, by using *contexts*, *synthetic events*, and other data sharing measures, several rules can be joined together into more powerful event correlation schemes. For example, the ruleset in Fig. 1 has been designed for processing Snort IDS syslog alarms, in order to detect repeated multifaceted attacks from the same host. The ruleset assumes the following alarm format:

Oct 25 11:36:06 mysensor snort[12341]: [1:16431:5] SQL generic sql with comments injection attempt - GET parameter [Classification: Web Application Attack] [Priority: 1] {TCP} 192.168.17.13:43148 -> 10.12.23.39:80

```
#
# The rules below are stored in /etc/sec/ids.sec
#

type=EventGroup
ptype=RegExp
pattern=snort\[\d+\]: \[(\d+:\d+):\d+\] .*\
 \{\w+\} ([\d.]+)(?::\d+)? -> [\d.]+(?::\d+)?
context=!IP_$2_ALARM_$1
count=alias ATTACKER_$2 IP_$2_ALARM_$1; \
  create TRIGGER_$1_$2 120 ( unalias IP_$2_ALARM_$1 )
init=create ATTACKER_$2
end=delete ATTACKER_$2
desc=attacking host $2
action=event Multifaceted attack from $2
thresh=10
window=120

type=SingleWithThreshold
ptype=RegExp
pattern=Multifaceted attack from ([\d.]+)
desc=multifaceted attacks from $1
action=pipe 'Continuous multifaceted attacks from $1' \
       /bin/mail root@example.com
thresh=5
window=1800
```

Fig. 1. Ruleset for processing Snort IDS alarms.

In order to start a SEC daemon for processing Snort IDS alarms that will be appended to */var/log/messages*, the following command line can be used:

/usr/bin/sec --conf /etc/sec/ids.sec --input /var/log/messages --detach

The first rule depicted in Fig. 1 will match an incoming Snort IDS syslog alarm with the regular expression which sets the *$1* match variable to alarm ID and *$2* match variable to attacker IP address. For example, if the above example Snort alarm is observed, match variables will be set as *$1=1:16431* and *$2=192.168.17.13*. The rule will then substitute match variables in the Boolean expression given with the *context* field, and the expression evaluates TRUE if the context *IP_192.168.17.13_ALARM_1:16431* does not exist. If that is the case, the rule will start an event correlation operation with the ID *<rulefile name, rule offset in rulefile, value of desc field>* which yields *</etc/sec/ids.sec, 0, attacking host 192.168.17.13>*. The operation expects 10 events within 120 seconds as defined with *thresh* and *window* fields of the rule. After the operation has been initialized, it first creates the context *ATTACKER_192.168.17.13* (according to the *init* field). After that, the operation sets up an alias name *IP_192.168.17.13_ALARM_1:16431* for this context as defined with the *count* field. The alias will exist for 120 seconds and will prevent the rule from matching further alarms with this particular combination of attacker IP and alarm ID. The alias lifetime is controlled by the trigger context *TRIGGER_1:16431_192.168.17.13* which will expire after 120 seconds and remove the alias. After creating the context and the alias, the operation sets its event counter to 1.

When further events appear that match the first rule, the operation ID is calculated, and if the operation with the given ID does not exist, it is initialized as described above (since the operation ID contains the attacker IP, there will be a separate event counting and thresholding operation for each attacker). However, if the operation exists, it will receive the matching event and increment its event counter, and also create an alias for attacker IP and alarm ID, in order to avoid counting further alarms of same type for the given attacker within 120 seconds.

If some operation has counted 10 alarms within the last 120 seconds, this indicates the use of different attack techniques from some malicious host within a short time frame. Therefore, the operation generates the synthetic event *Multifaceted attack from attackerIP* (as defined with the *action* field of the rule), and consumes further alarms silently until the end of the event correlation window. Before terminating, the operation will delete the context *ATTACKER_attackerIP* (according to rule's *end* field) which will also destroy all alias names associated with this context, in order to avoid interference with potential further operations for the same attacker IP. Note that alias lifetime triggers don't need removal, since they take no action for non-existing aliases, and potential future recreation of the trigger will destroy any previous instance. If the operation has seen less than 10 alarms for the attacker within the 120 second window, the operation slides the window forward and continues. If no events remain in the window after sliding, the operation terminates.

Synthetic events generated by operations started by the first rule in Fig. 1 are inserted into input buffer of SEC and treated similarly to regular input events from */var/log/messages*. Therefore, these events will match the second rule in Fig. 1 which will start a separate counting and thresholding operation for each attacker IP. If an operation observes 5 events within 1800 seconds for the given attacker, it sends an e-mail warning about repeated multifaceted attacks to *root@example.com*.

B. Advanced Event Matching with Perl Functions

Although regular expressions allow for flexible parsing of input events, they have some limitations. Firstly, apart from string recognition it is hard to implement other types of

matching, for example, arithmetic filters for numerical fields in input events. Secondly, regular expressions of different SEC rules work independently with no data sharing between them.

For instance, the ruleset in Fig. 1 assumes that the attacker IP is always found in the source IP field of the alarm. However, a number of attacks manifest themselves through specific victim responses to attackers. As a result, the destination IP address field reflects the attacker, for example:

Oct 25 14:19:03 mysensor snort[12341]: [1:2101201:11] GPL WEB_SERVER 403 Forbidden [Classification: Attempted Information Leak] [Priority: 2] {TCP} 10.12.23.39:80 -> 192.168.11.229:52466

Unfortunately, it is not straightforward to write a single regular expression for distinguishing external and home IP addresses in relevant alarm fields and setting match variables properly for all scenarios. In order to address complex event matching and parsing tasks, SEC allows for setting up custom Perl functions. Since user-defined code often benefits from external Perl modules, these can be loaded at SEC startup. Fig. 2 presents sample rules for improving the ruleset from Fig. 1.

```
type=Single
ptype=SubStr
pattern=SEC_STARTUP
context=SEC_INTERNAL_EVENT
desc=load Net::IP module and set $homenet
action=eval %ret (require Net::IP); \
  if %ret () else ( logonly Net::IP not found; \
                    eval %o exit(1) ); \
  lcall %ret -> \
  ( sub { $homenet = new Net::IP('10.12.23.32/29'); } )

type=EventGroup
ptype=PerlFunc
pattern=sub { if ($_[0] =~ \
/snort\[\d+\]: \[(\d+:\d+):\d+\] .*\
 \{\w+\} ([\d.]+)(?::\d+)? -> ([\d.]+)(?::\d+)?/) { \
    my $ip = new Net::IP($2); \
    if ($ip->Net::IP::overlaps($homenet) \
    == $Net::IP::IP_A_IN_B_OVERLAP) \
  { return ($1, $3); } else { return ($1, $2); } \
} return 0; }
context=!IP_$2_ALARM_$1
count=alias ATTACKER_$2 IP_$2_ALARM_$1; \
  create TRIGGER_$1_$2 120 ( unalias IP_$2_ALARM_$1 )
init=create ATTACKER_$2
end=delete ATTACKER_$2
desc=attacking host $2
action=event Multifaceted attack from $2
thresh=10
window=120
```

Fig. 2. Using a Perl function for matching and parsing Snort IDS alarms.

The first rule requires the presence of the *--intevents* option in SEC command line which forces the generation of special synthetic events at SEC startup, restarts, log rotations, and shutdown. In order to disambiguate these synthetic events from similarly looking regular input, SEC sets up a temporary context *SEC_INTERNAL_EVENT* which exists only during the processing of these events. The first rule matches the *SEC_STARTUP* event (generated at SEC startup) and loads the *Net::IP* Perl module. The rule also sets the Perl *$homenet* global variable to 10.12.23.32/29. If the module loading fails, the rule logs a relevant error message and terminates the SEC process by calling *exit(1)*. The second rule uses a Perl function for matching IDS alarms which receives the alarm message as its first parameter. The function matches each alarm with the regular expression from Fig. 1, but in addition to match variables *$1* and *$2*, match variable *$3* is set to the destination IP address. Then the *overlaps()* method from *Net::IP* module is used for checking if the source IP address belongs to the home network (represented by *$homenet* variable that was set from previous rule). If that's the case, the function returns alarm ID and destination IP, otherwise the function returns alarm ID and source IP. Outside the function, its return values are mapped to match variables *$1* and *$2*, and thus the *$2* variable always reflects the attacker IP in the rest of the rule definition.

Perl functions can not only be used as patterns for event matching and parsing, but also as additional filters in rule *context** fields. For example, the following rule fields match an SSH login failure syslog event if the connection originates from a privileged port on the client host (the port number of the client host is assigned to the *$1* match variable, and the variable is passed to a Perl function for verifying its value is smaller than 1024):

```
ptype=RegExp
pattern=Failed [\w.-]+ for \w+ from [\d.]+ port (\d+) ssh2
context=$1 -> ( sub { $_[0] < 1024 } )
```

In a similar way, many Perl functions can be defined for event matching and parsing which share global data structures (e.g., a hash table of malicious IP addresses). Since including longer functions in rule definitions might decrease rule readability, it is recommended to encapsulate such code into separate Perl modules and load them as depicted in Fig. 2.

C. Using Named Match Variables and Match Caching

When creating larger SEC rulebases with hundreds of rules, a number of rules might use identical regular expression or Perl function patterns. However, significant amount of CPU time could be spent for matching an event repeatedly with the same pattern. Moreover, the use of numeric match variables (e.g., *$1* and *$2*) assumes that the number of input event fields and their nature are known in advance, but this is not always the case. Finally, variable numbering can easily change if the pattern is modified, making rules harder to maintain. In order to address aforementioned issues, SEC supports named match variables and match caching as depicted by a ruleset in Fig. 3. This ruleset processes Linux *iptables* firewall syslog events which contain a number of fieldname-value pairs, for example:

Oct 26 11:05:22 fw1 kernel: iptables: IN=eth0 OUT= MAC=XXX SRC=192.168.94.12 DST=10.12.23.39 LEN=52 TOS=0x00 PREC=0x00 TTL=60 ID=61441 DF PROTO=TCP SPT=53125 DPT=23 WINDOW=49640 RES=0x00 SYN URGP=0

Depending on the nature of network traffic, *iptables* events can contain a variety of different fields, and writing one regular expression for all possible field combinations is intractable. On the other hand, the Perl function in the first rule takes advantage of iterative regular expression matching, in order to parse out each fieldname-value pair and store it into a Perl hash table. Since the function returns a reference to this hash table, named match variables *$+{name}* are created from all fieldname-value pairs in the table. For example, when the

above example event is matched, *$+{SRC}* and *$+{DST}* variables are set to 192.168.94.12 and 10.12.23.39, respectively, and *$+{SYN}* is set to 1 (default when fieldname does not have a value). Therefore, the naming scheme for match variables is dynamic and fully determined by input data. After the event has been matched, the result of parsing is stored in the pattern match cache under the entry *IPTABLES* (the match caching is configured with the *varmap* field of the rule). Note that the pattern match cache is cleared before processing each new input event, and thus all cache entries always reflect parsing results for the currently processed event. Also, each cache entry is implemented as a Perl hash table which can be accessed directly from rule *context** fields (see Fig. 3).

```
type=SingleWithThreshold
ptype=PerlFunc
pattern=sub { my(%var); my($line) = $_[0]; \
    if ($line !~ /kernel: iptables:/g) { return 0; } \
    while ($line =~ /\G\s*([A-Z]+)(?:=(\S*))?/g) { \
      $var{$1} = defined($2)?$2:1; \
    } return \%var; }
varmap=IPTABLES
continue=TakeNext
desc=too many blocked packets from IP $+{SRC}
action=logonly
thresh=100
window=120

type=SingleWithThreshold
ptype=Cached
pattern=IPTABLES
context=IPTABLES :> ( sub { exists($_[0]->{"SYN"}) && \
                            exists($_[0]->{"FIN"}) } )
desc=SYN-FIN flood attempt against IP $+{DST}
action=logonly
thresh=100
window=120
```

Fig. 3. Ruleset for processing Linux iptables firewall events.

Since the *continue* field of the first rule is set to TakeNext, all matching input events are passed to the following rule for further processing. In order to save CPU time, the second rule matches incoming *iptables* events by doing a quick lookup for the *IPTABLES* entry in the pattern match cache (as specified with *ptype=Cached* and *pattern=IPTABLES*). If this entry is found, the *:>* operator in the *context* field passes a reference to the entry into a Perl function which verifies the presence of *$+{SYN}* and *$+{FIN}* variables under the entry. If both variables exist, the rule matches an event, and the *$+{DST}* variable in the *desc* field is set from the *IPTABLES* entry.

Note that named match variables and match caching are also supported for regular expression patterns – for example, the regular expression *Connection closed from (?<ip>[\d.]+)* creates match variables *$1* and *$+{ip}* which are both set to an IP address, and these variables can be cached with the *varmap* statement.

D. Arranging rulesets hierarchically

Each SEC ruleset is stored in a separate text file, and rules from one file are applied to an input event in the order they have been defined in the file. Also, by default rulesets from different files are applied independently against each input event. However, if only few rulesets are relevant for most input events, the use of larger rulebases involves considerable performance penalty, since an input event will be potentially matched against many irrelevant rulesets.

SEC provides several options for addressing this problem. Firstly, if SEC has been started with the *--intcontexts* command line option, reception of any input event will trigger the creation of a temporary context that reflects the source of this event (e.g., *_FILE_EVENT_/var/log/messages*). After all rules have been applied against the input event, the context is deleted immediately. If some rules are designed to match events from specific sources only, such temporary contexts allow for preventing matching attempts for other sources. For example, the following rule fields match the regular expression with input events from */var/log/secure* only (square brackets around *_FILE_EVENT_/var/log/secure* force the check for the presence of this context *before* regular expression matching):

ptype=RegExp
pattern=Connection closed from (?<ip>[\d.]+)
context=[_FILE_EVENT_/var/log/secure]

Also, one user-defined context can be set for multiple sources. Prior to SEC-2.7.6, *_INTERNAL_EVENT* context was always used for all synthetic events, while with more recent SEC versions *cevent* and *cspawn* actions can be employed for generating synthetic events with custom contexts.

```
####################
# the content of /etc/sec/main.sec

type=Jump
context=[ _FILE_EVENT_/var/log/messages ]
ptype=PerlFunc
pattern=sub { my(%var); my($line) = $_[0]; \
    if ($line !~ /kernel: iptables:/g) { return 0; } \
    while ($line =~ /\G\s*([A-Z]+)(?:=(\S*))?/g) { \
      $var{$1} = defined($2)?$2:1; \
    } return \%var; }
varmap=IPTABLES
desc=parse and route iptables events
cfset=iptables-events

type=Jump
context=[ _FILE_EVENT_/var/log/secure ]
ptype=RegExp
pattern=sshd\[\d+\]:
desc=route sshd events from /var/log/secure
cfset=sshd-events

####################
# the content of /etc/sec/fw.sec

type=Options
procallin=no
joincfset=iptables-events

type=SingleWithThreshold
ptype=Cached
pattern=IPTABLES
desc=Too many blocked packets to IP $+{DST}
action=logonly
thresh=100
window=120

####################
# the content of /etc/sec/sshd.sec

type=Options
procallin=no
joincfset=sshd-events

…
```

Fig. 4. An example hierarchical ruleset.

Secondly, *Jump* rules can be used for submitting input events to specific rulesets for further processing, and rulesets can be configured to accept input from *Jump* rules only. Fig. 4 depicts an example for three rulesets which are arranged into two-level hierarchy.

From the three rulesets presented in Fig. 4, the ruleset from */etc/sec/main.sec* is applied for recognizing input events and submitting them to two other rulesets which are labeled as *iptables-events* and *sshd-events*. Since both rulesets contain an *Options* rule with the *procallin=no* statement, they will only accept input events from *Jump* rules. As a result, the ruleset in */etc/sec/fw.sec* is restricted to receive *iptables* syslog events from */var/log/messages* which have already been parsed by the *Jump* rule. Also, the ruleset in */etc/sec/sshd.sec* can only process SSH daemon syslog events from */var/log/secure*.

The above example illustrates that ruleset hierarchies can significantly reduce cost of event processing if many rules and rulesets are involved, especially if event parsing is accomplished in top levels of the hierarchy. In more general cases, rulesets can be arranged into graph-like structures which can introduce processing loops. Whenever SEC detects a loop during matching an event against rules, processing for the event is terminated.

IV. Performance Data and Conclusion

We have used best practices and recommendations from the previous section in a production environment for two years. One of our SEC instances is running on a Linux server and using a hierarchically arranged rulebase of 375 rules, in order to correlate syslog events from many production servers. According to recently collected performance data for 172 days, this SEC instance has processed 1,636,805,087 events during 14,881,059 seconds (109.9 events per second), and 1,331,412,766 events have been matched by rules. During event processing, the SEC instance has consumed 448,150 seconds of CPU time on a single core of an Intel Xeon X5650 processor (about 3% of available CPU time on one core). When we briefly experimented with disabling the hierarchical rulebase arrangement, the CPU load increased 4-5 times.

Although we have reviewed a number of powerful features of SEC for creating scalable configurations, many interesting topics have been left out from this paper due to space limitations. In particular, we haven't provided in-depth discussion on individual rule types, advanced use of contexts for aggregating and reporting event data, actions for working with sockets, clock-triggered event correlation schemes, and integration with other monitoring applications. In order to get a detailed insight into those issues, the interested reader is referred to the SEC official documentation and mailing list, but also to past papers [2, 10, 11].

Acknowledgment

The author of SEC expresses his gratitude to John P. Rouillard for many great ideas and creative discussions which have been crucial for developing SEC during the last 15 years. The authors also thank Mr. Kaido Raiend and Mr. Ain Rasva for supporting this work.

References

[1] Risto Vaarandi, "SEC – a Lightweight Event Correlation Tool," Proceedings of the 2002 IEEE Workshop on IP Operations and Management, pp. 111-115.

[2] John P. Rouillard, "Real-time Logfile Analysis Using the Simple Event Correlator (SEC)," Proceedings of the 2004 USENIX Large Installation System Administration Conference, pp. 133-149.

[3] Jeffrey Becklehimer, Cathy Willis, Josh Lothian, Don Maxwell, and David Vasil, "Real Time Health Monitoring of the Cray XT3/XT4 Using the Simple Event Correlator (SEC)," Proceedings of the 2007 Cray User Group Conference.

[4] Ross Miller, Jason Hill, David A. Dillow, Raghul Gunasekaran, Galen Shipman, and Don Maxwell, "Monitoring Tools for Large Scale Systems," Proceedings of the 2010 Cray User Group Conference.

[5] Jason J. Hill, Dustin B. Leverman, Scott M. Koch, and David A. Dillow "Determining the health of Lustre filesystems at scale," Proceedings of the 2011 Cray User Group Conference.

[6] Byung H. Park, Thomas J. Naughton, Pratul Agarwal, David E. Bernholdt, Al Geist, and Jennifer L. Tippens, "Realization of User Level Fault Tolerant Policy Management through a Holistic Approach for Fault Correlation," Proceedings of the 2011 IEEE International Symposium on Policies for Distributed Systems and Networks, pp. 17-24.

[7] David Lang, "Building a 100K log/sec logging infrastructure," Proceedings of the 2012 USENIX Large Installation System Administration Conference, pp. 203-213.

[8] Risto Vaarandi, "Tools and Techniques for Event Log Analysis," PhD Thesis, Tallinn University of Technology, 2005.

[9] Michael R. Grimaila, Justin Myers, Robert F. Mills, and Gilbert L. Peterson, "Design and Analysis of a Dynamically Configured Log-based Distributed Event Detection Methodology," The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology 01/2012; 9(3), pp. 219-241, 2012.

[10] Risto Vaarandi, "Simple Event Correlator for real-time security log monitoring," Hakin9 Magazine 1/2006 (6), pp. 28-39, 2006.

[11] Risto Vaarandi and Michael R. Grimaila, "Security Event Processing with Simple Event Correlator," Information Systems Security Association (ISSA) Journal 10(8), pp. 30-37, 2012

[12] Risto Vaarandi and Karlis Podins, "Network IDS Alert Classification with Frequent Itemset Mining and Data Clustering," Proceedings of the 2010 IEEE Conference on Network and Service Management, pp. 451-456.

[13] Risto Vaarandi, "Platform Independent Event Correlation Tool for Network Management," Proceedings of the 2002 IEEE/IFIP Network Operations and Management Symposium, pp. 907-909.

[14] Gabriel Jakobson and Mark Weissman, "Real-time telecommunication network management: Extending event correlation with temporal constraints," Proceedings of the 1995 IEEE International Symposium on Integrated Network Management, pp. 290-301.

[15] Boris Gruschke, "Integrated Event Management: Event Correlation using Dependency Graphs," Proceedings of the 1998 IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, pp. 130-141.

[16] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie, "High speed and robust event correlation," IEEE Communications Magazine 34(5), pp. 82-90, 1996

[17] M. Steinder and A. S. Sethi, "End-to-end Service Failure Diagnosis Using Belief Networks," Proceedings of the 2002 IEEE/IFIP Network Operations and Management Symposium, pp. 375-390.

[18] Stephen E. Hansen and E. Todd Atkins, "Automated System Monitoring and Notification With Swatch," Proceedings of the 1993 USENIX Large Installation System Administration Conference, pp. 145-152.

[19] http://www.crypt.gen.nz/logsurfer/

[20] http://nxlog-ce.sourceforge.net/

[21] http://esper.codehaus.org/