In: Advances in Security Information Management: Perceptions and Outcomes
Editor: Suarez-Tangil, G./ Palomar, E., pp. 1-18

Chapter Proposal

COMPLEX EVENT PROCESSING BASED SIEM

Vincenzo Gulisano and Ricardo Jiménez Peris and Marta Patiño Martnez and Claudio Soriente and Valerio Vianello[*]
Universidad Politécnica de Madrid

Abstract

Correlation engines are a key component of modern SIEMs. They employ user-defined rules to process input alerts and identify the minimal set of meaningful data that should be provided to the final user. As IT systems grow in size and complexity, the amount of alerts generated by probes is constantly increasing and centralized SIEMs (i.e., SIEMs with a single-node correlation engine) start to show their processing limits.

In this chapter we present a novel parallel correlation engine to be embedded in next generation SIEMs. The engine is based on Complex Event Processing and on a novel parallelization technique that allows to deploy the engine on an arbitrary number of nodes in a shared-nothing cluster. At the same time, the parallel execution preserves semantic transparency, i.e., its output is identical to the one of an ideal centralized execution. Our engine scales with the number of processed alerts per second and allows to reach beyond the processing limits of a centralized correlation engine.

Keywords: Correlation engine, scalability, complex event processing.

1. Introduction

Security Information Management Systems (SIEMs) represent an effective tool to monitor complex systems and help security IT staff to identify and react to security threats. SIEMs

[*] E-mail address: {vgulisano,rjimenez,mpatino,csoriente,vvianello}@fi.upm.es

provide reliable information abstraction and correlation so that the amount of information delivered to the final user is free of background noise and identifying real threats as well as their sources becomes easier. The heart of a SIEM is its correlation engine that employs user-defined rules to process input alerts and identify the minimal set of meaningful data that should be provided to the final user.

As IT systems keep growing and become more complex, current SIEMs [Ali10, Pre10] start to show their limits. They rely on a centralized correlation engine that can not cope with massive amount of information produced by large and complex infrastructures and are forced either to shed the load or to queue alerts and delay their processing. This is not an option in many scenarios where no information can be discarded and real-time processing of alerts and timely discovery of threats is a must.

Motivation. Complex Event Processing (CEP) [ACc$^+$03, AAB$^+$05, CDTW00, CCD$^+$03, SHCF03] represents a promising tool to improve current SIEMs. They can process large amounts of information in real time and provide information abstraction and correlation, similarly to SIEM correlation engines. Within the context of CEPs, an alert is called an *event*, a directive is a *query* and the rules of a directive are referred to as *operators*.

Some CEPs offer distributed processing [SHCF03, AAB$^+$05] and, at a first glance, might seem as a perfect match to overcome the single-site execution limitations of current SIEMs. However, the scalability achieved by previous CEP proposals is still limited due to single node bottlenecks (e.g. [AAB$^+$05]) or high distribution overhead (e.g. [SHCF03]). In particular, current distributed CEPs try to overcome the processing limit of a centralized execution either with inter-query parallelism or intra-query parallelism via inter-operator parallelism [CBB$^+$03].

The former allows to run different queries on different nodes. Intra-query parallelism through inter-operator parallelism, deploys different operators on different nodes so that the per-event load at each node decreases. However, both options still require the whole data flow to pass through a single node, that is, no real parallelism is achieved.

Providing real parallelism (i.e., intra-operator parallelism) in CEPs is a challenging task. A naïve way to achieve it, also known as *intra-partition* parallelism, dictates to replicate an operator on several nodes and have each node processing a fraction of the input. Partitioning the input data over several processing unit is an ad-hoc (hence, error-prone) process. This strategy allows to overcome the processing capacity of a centralized system because no node processes the whole input data; however, intra-partition parallelism is a poor match for SIEMs, where correlation of all the available information is key to threat discovery. Indeed, with intra-partition parallelism, events that once correlated might reveal a threat, could be processed by different nodes and the threat might remain hidden.

Contributions. This chapter proposes a novel approach to alert correlation in SIEM systems. We propose to use CEP systems as the underlying correlation engine of a SIEM, in order to improve their scalability and applicability in scenarios where massive amount of alerts must be processed with strict timing requirements.

We introduce a novel parameterizations technique for CEP systems that achieves intra-operator parallelism [GJPPMV10]. The proposed technique allows to deploy any query operator on an arbitrary number of nodes in a shared-nothing environment.

The logical input stream of alerts is partitioned in many physical streams that flow in parallel through the system so that no node is required to process the whole input. The parallel system enables throughput rates far beyond the ones allowed by a single processing node while guaranteeing the correctness of a centralized execution. The result is a highly-scalable CEP that can process massive amount of input data on-the-fly and that is a perfect candidate to enhance state of the art SIEM correlation engines.

Organization. Section 2. provides an overview of Complex Event Processing introducing basic terminology and operators. Section 3. introduces our novel parallelization technique that allows the correlation engine to scale with the input load while guaranteeing that no threats are missed. Section 4. provides a concrete example of how to convert a SIEM directive in a CEP query while Section 5. provides some concluding remarks.

2. Complex Event Processing

Complex Event Processing (CEP) finds patterns of interest over streams of data. A data stream S is an infinite sequence of events $e_0, e_1, e_2, \ldots$. All events of a stream share a "schema" that defines the number and type of their attributes; attribute A_i of event e is referred to as $e.A_i$. We assume events are timestamped at data sources that are equipped with well-synchronized clocks managed through, e.g., NTP [Mil03]. In case data sources clock synchronization is not feasible, we assume events are timestamped at the entry point of the CEP system. In either case, we refer to the timestamp of event e as $e.ts$, that is, timestamp is an additional attribute of all schemas.

Table 1 provides a sample schema of a security event; it lists relevant attributes that will be used in the examples throughout the chapter. Pair $Plugin\_id$ and $Plugin\_sid$, identify the sensor that generated the event (e.g, SNORT) and the event type (e.g., FINGER probe 0 attempt), respectively. Src_IP, Src_Port and Dst_IP, Dst_Port specify the two endpoints that were interested by the event. ts is the timestamp generated when the event was recorded.

Table 1. Sample security event schema.

$Plugin\_id$	Id of the device that generated the event
$Plugin\_sid$	Id of the type of event
Src_IP	Source IP address
Src_Port	Source port number
Dst_IP	Destination IP address
Dst_Port	Destination port number
ts	Timestamp

CEP allow users to specify pattern of interests over incoming data through *continuous queries*. The latter is "continuosly" executed over the streaming data, so that each time a pattern of interest is identified, the result is presented to the user.

A continuous query is defined over one or more input streams and can have multiple output streams. Queries are modeled as an acyclic direct graph made of "boxes and arrows".

Each box (operator) represents a computational unit that performs an operation over events input through its incoming arrow(s) and outputs resulting events over its outgoing arrow(s). Each arrow (stream) represents a data flow. An arrow between two boxes means that the second box consumes events produced by the first one.

Typical query operators of CEP systems are similar to relational algebra operators. They are classified depending on whether they keep state information across input events. Stateless operators (e.g., Map, Union and Filter) do not keep any state across events and perform one-by-one computation; that is, each incoming event is processed and an output event is produced, if any.

Stateful operators (e.g., Aggregate and Join) perform operations on multiple input events; that is, each incoming event updates the state information of the operator and contributes to the output event, if any. Because of the infinite nature of data streams, stateful operators keep state information only for the most recent incoming events. This technique is referred to as *windowing*. Windows can be defined over a period of time (e.g., events received in the last hour) or over the number of received events (e.g., last 100 events).

In the following we introduce the basic processing tasks of a SIEM correlation engine and map them to a CEP operator, providing examples of both its syntax and semantics. Readers interested in CEP are referred to [GGR12].

2.1. Transforming Events

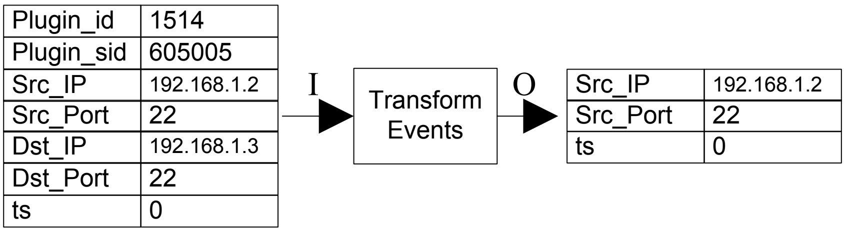
Raw information conveyed by incoming events might need further processing before being used for correlation. For example, in order to reduce the per-event processing overhead and allow higher throughput, events attributes that are not required for correlation, should be removed as soon as the event reaches the engine. Other transformations include, timezone translation of the event timestamp, conversion among metrics (e.g., Celsius degrees to Fahrenheit degrees), etc.

Event transformation in CEP systems is performed by the Map operator. The latter is a generalized projection operator defined as

$$M\{A'_1 \leftarrow f_1(e_{in}), \ldots, A'_n \leftarrow f_n(e_{in})\}(I, O)$$

with I and O that denote the input and output stream, respectively. e_{in} is a generic input event, $\{A'_1, \ldots, A'_n\}$ is the schema of the output stream and $\{f'_1, \ldots, f'_n\}$ is a set of user-defined functions. Each input event is transformed according to the set $\{f'_1, \ldots, f'_n\}$ and the resulting event is propagated over the output stream. The output stream schema might differ from the input one, but the output event preserves the timestamp of the input one. Figure 1 shows a sample Map operator that extracts source IP address and port from each incoming event. The latter event complies with the schema of Table 1 while output events have attributes Src_IP, Src_Port, ts. The syntax of the operator in Fig. 1 is

$$M\{A'_1 = e_{in}.Src\_IP, A'_2 = e_{in}.Src\_Port, A'_3 = e_{in}.ts\}(I, O)$$



Plugin_id	1514
Plugin_sid	605005
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

Src_IP	192.168.1.2
Src_Port	22
ts	0

Figure 1. Extracting attributes from incoming events.

2.2. Filtering Events

One of the main tasks of the correlation engine is to remove background noise from the incoming streams of alerts. A basic technique to reduce the load of alerts of interest is to look into each event and choose whether it is relevant or not, based on its content. For example, if the engine is monitoring a particular subnetwork, events related to nodes outside of the monitored set, can be safely discarded.

The CEP operator that is used either to discard events or to route events to different output streams, depending on the event content is the Filter operator. It is similar to a case statement and it is used to route input events over multiple output streams. It is defined as

$$F\{P_1, \ldots, P_m\}(I, O_1, \ldots, O_m[, O_{m+1}])$$

where I is the input stream, $O_1 \ldots, O_m, O_{m+1}$ is an ordered set of output streams and $P_1, \ldots, P_m$ is an ordered set of predicates.

The number of predicates equals the number of output streams and each input event is forwarded over the output stream associated to the first predicate that the event satisfies. That is, e_{in} is forwarded over O_j where $j = \min_{1 \leq i \leq m}\{i \mid P_i(e_{in}) = TRUE\}$.

Events that satisfy none of the predicates are output on stream O_{m+1}, or discarded if output $m + 1$ has not been defined.

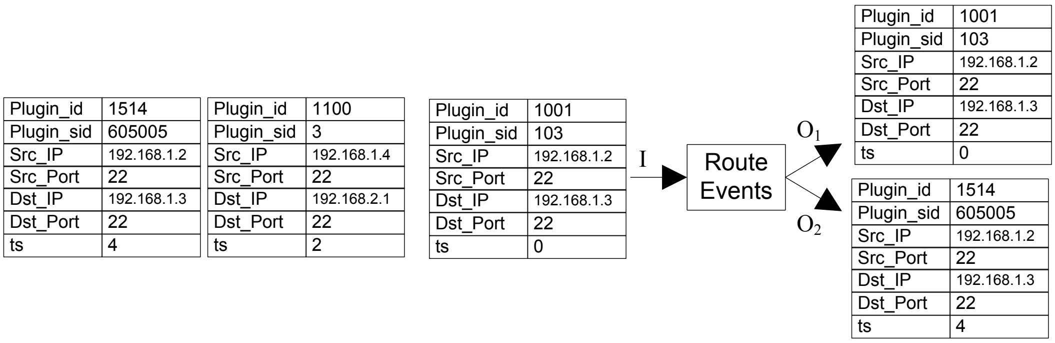
Predicates are defined over the attributes of the input event and include comparisons between two attributes (e.g.,$e_{in}.Src\_Port = e_{in}.Dst\_Port$)or comparisons between an attribute and a constant(e.g., $e_in.Dst\_Port = 22$).

Figure 2 shows a sample Filter operator that routes events based on their $Plugin\_ID$ attribute. SNORT events ($Plugin\_ID = 1001$) are routed over output O_1, while CISCO Pix events ($Plugin\_ID = 1514$) are forwarded over output stream O_2; as no additional output stream has been specified, events with $Plugin\_ID \notin \{1001, 1514\}$ are discarded. The syntax of the operator in Fig. 2 is

$$F\{e_{in}.Plugin\_id = 1001, e_{in}.Plugin\_id = 1514\}(I, O_1, O_2)$$

2.3. Merging events

SIEM correlation engines collect event from a multitude of sources, so that events need to be merged in a unique stream before further processing. In CEP systems, the Union operator merges two or more input streams with the same schema into a single output stream. Input



Plugin_id	1514
Plugin_sid	605005
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	4

Plugin_id	1100
Plugin_sid	3
Src_IP	192.168.1.4
Src_Port	22
Dst_IP	192.168.2.1
Dst_Port	22
ts	2

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

Plugin_id	1514
Plugin_sid	605005
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	4

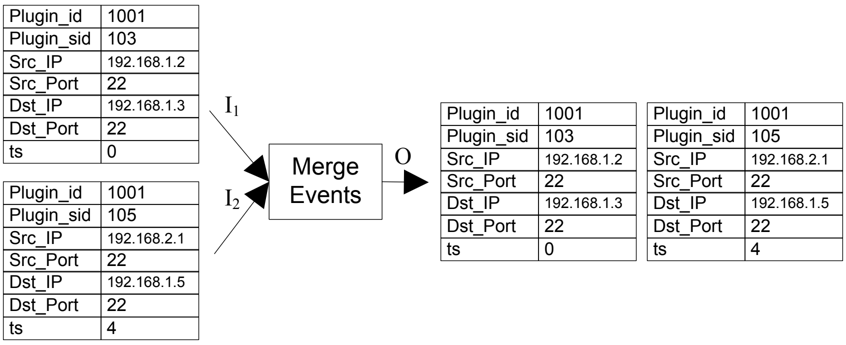
Figure 2. Route SNORT and CISCO Pix events, discard others.

events are propagated over the output stream in FIFO order. The Union operator is defined as

$$U\{\}(I_1, \ldots, I_n, O)$$

where $I_1, \ldots, I_n$ is a set of input streams and O is the only output stream; all streams share the same schema. Figure 3 shows a sample Union operator that merges streams from two SNORT sensors into a single output stream, propagated for further processing. The operator syntax is

$$U\{\}(I_1, I_2, O)$$



Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

Plugin_id	1001
Plugin_sid	105
Src_IP	192.168.2.1
Src_Port	22
Dst_IP	192.168.1.5
Dst_Port	22
ts	4

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

Plugin_id	1001
Plugin_sid	105
Src_IP	192.168.2.1
Src_Port	22
Dst_IP	192.168.1.5
Dst_Port	22
ts	4

Figure 3. Merge SNORT events from two different sensors.

2.4. Aggregating events

In some scenarios, only little information can be inferred from single events, while a greater knowledge can be obtained aggregating multiple events together. For example, each new connection to a given node might be of little interest, while the number of connections over the last, e.g., minute, hour, day, might provide a greater amount of information.

The Aggregate operator is used in CEP systems to provide information related to group of incoming events. The operator computes an aggregate function (e.g., average, count, etc.) over a window of its input stream. It is defined as

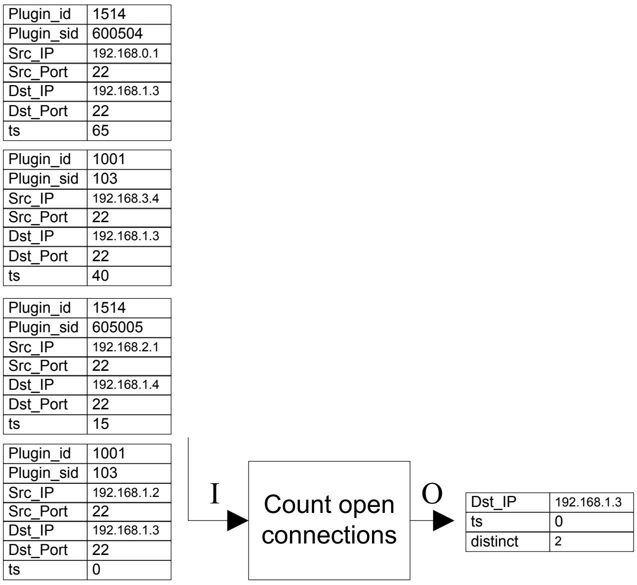
$$\begin{gathered} Ag\{Wtype, Size, Advance, A'_1 \leftarrow f_1(W), \ldots, A'_n \leftarrow f_n(W), \\ [Group - by = (A_{i_1}, \ldots, A_{i_m})]\}(I, O) \end{gathered}$$

Events over input stream I are stored in the current window W until it becomes full. $Wtype$ specifies the window type that can be either time-based ($Wtype = time$) or event-based ($Wtype = numEvents$). If the window is time-based, it is considered full if the time distance between the incoming event and the earliest event in the window exceeds the window $Size$. In case of event-based windows, a window is full if it contains $Size$ events.

Once a window is full, an output event is produced. Output events are propagated over stream O and have timestamp equal to the timestamp of the earliest event in the current window. The output event schema is $\{A'_1, \ldots, A'_n\}$ and $\{f_1, \ldots, f_n\}$ is a set of user-defined functions (e.g., sum, count, average, etc.) computed over all events in the window.

After an output event has been propagated, the window is updated (or "slid" forward) and stale events are discarded according to parameter $Advance$. If $Wtype = time$ and e_{in} is the current input event, a event e in the current window is discarded if $e_{in}.ts - e.ts > Size$. If $Wtype = numEvents$, the earliest $Advance$ events are discarded from the current window.

Finally, the parameter Group-by is optional and is used to define equivalence classes over the input stream. In particular, assume Group-by$= A_i$, where A_i is an attribute of the input schema. Then, the Aggregate operator handles separate windows for each possible value of A_i.



Plugin_id	1514
Plugin_sid	600504
Src_IP	192.168.0.1
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	65

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.3.4
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	40

Plugin_id	1514
Plugin_sid	605005
Src_IP	192.168.2.1
Src_Port	22
Dst_IP	192.168.1.4
Dst_Port	22
ts	15

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

Dst_IP	192.168.1.3
ts	0
distinct	2

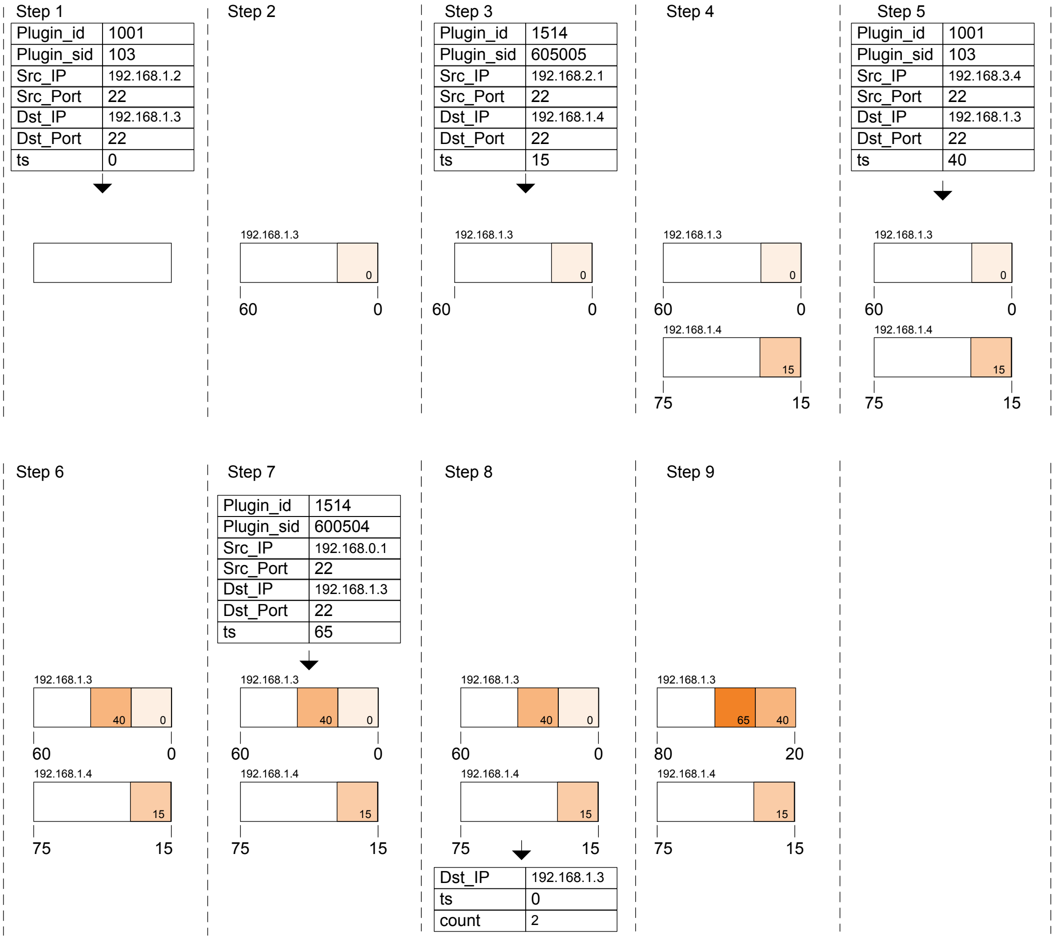
Figure 4. Count open connections for each server on a per-minute basis.

To further clarify the semantic of operators based on windows, we refer to the Aggregate operator of Figure 4. The operator counts the number of open connections for each server (i.e., Group-by= Dst_IP) during the last minute (i.e, $Size = 60$), with an advance of twenty seconds (i.e., $Advance = 20$). It is defined as

$Ag\{time, 60, 20, A'_1 = count(), Group - by = e_{in}.Dst\_IP\}(I, O)$

Figure 4 shows the sequence of input events. Three events refer to a connection to IP address 192.168.1.3 (Dst_IP) and their timestamps are 0, 40 and 65, respectively. One event refers to a connection to IP address 192.168.1.4 and is generated at time 15.

Figures 5 shows the window evolution as the above events are received by the operator. In the following, we provide detailed steps of the operator behavior as events are received.



Step 1

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.1.2
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

Step 3

Plugin_id	1514
Plugin_sid	605005
Src_IP	192.168.2.1
Src_Port	22
Dst_IP	192.168.1.4
Dst_Port	22
ts	15

Step 5

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.3.4
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	40

Step 7

Plugin_id	1514
Plugin_sid	600504
Src_IP	192.168.0.1
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	65

Step 8

Dst_IP	192.168.1.3
ts	0
count	2

Figure 5. Aggregate Sample Time-Based Window Evolution

- **Step** 1. Event relative to IP address 192.168.1.3 (Dst_IP) with timestamp 0 is received.
- **Step** 2. As this is the first event relative to IP 192.168.1.3, a new window is created and the event is stored (recall that Group-by= Dst_IP).

- **Step** 3. Event relative to IP address 192.168.1.4 with timestamp 15 is received.

- **Step** 4. As for the previous event, there is no window for events relative to IP 192.168.1.4, so a new window is created and the event is stored.

- **Step** 5. Event relative to IP address 192.168.1.3 with timestamp 40 is received.

- **Step** 6. A window for IP address 192.168.1.3 is already defined. No event currently stored in the window is discarded as the timestamp difference between the input event (40) and the earliest event in the window (0) is smaller than the window size (i.e., 60 seconds). The incoming event is stored in the window.

- **Step** 7. Event relative to IP address 192.168.1.3 with timestamp 65 is received.

- **Step** 8. The window relative to IP address 192.168.1.3 must produce an output as the timestamp difference between the incoming event (65) and the earliest event in the window (0) is greater than the window size. The output event is produced considering the events currently stored in the window (the ones with timestamp 0 and 40).

- **Step** 9. After the output is produced, the window is slid forward of $Advance$ time units until the timestamp of the incoming event falls within the current window. Finally, the incoming event is stored. In the example, the event with timestamp 0 is discarded; in general, any event with timestamp between 0 and 20 would be removed.

2.5. Correlating Events

The ultimate goal of the SIEM engine is to correlate events coming from different input streams and extract information of interest. For example, a successful brute-force attack can be detected correlating a number of denied login attempts followed by one successful login event.

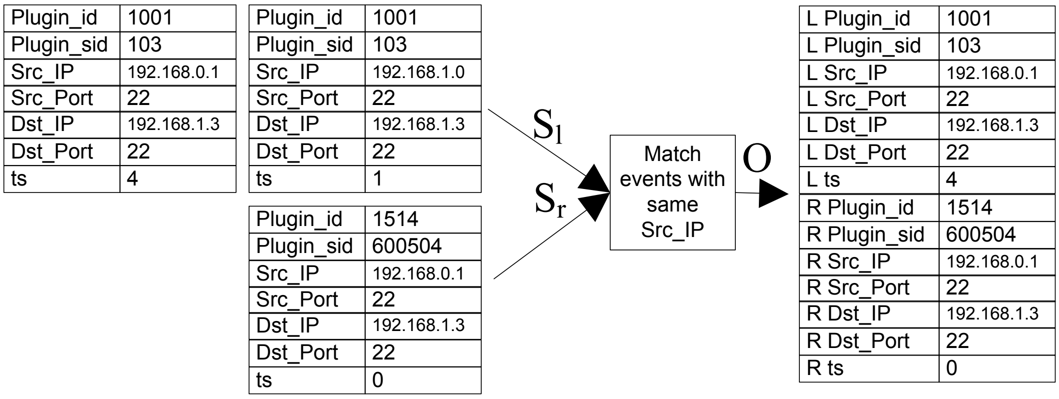
The Join operator can be used in CEP systems to correlate events from different sources. It is a binary operator defined as

$$J\{P, Wtype, Size\}(S_l, S_r, O)$$

S_l, S_r are two input streams referred to as *left* and *right*, respectively, while O denotes the output stream. P is a predicate over pairs of events (one from each input stream), that is, the Join predicate include comparisons between attributes of the same event or between attributes of different events. Parameters $Wtype$ and $Size$ are windows parameters similar to the ones in the Aggregate operator.

The Join operator keeps two separate windows, W_l, W_r, for each input stream. Events arriving on the left (resp. right) stream are stored in the left (resp. right) window and used to update (i.e., slide forward) the right (resp. left) window. If $Wtype = time$, upon arrival of event $e_{in} \in S_l$, window W_r is updated removing all events e such that $e_{in}.ts - e.ts \geq Size$. If $Wtype = NumEvents$, upon arrival of event $e_{in} \in S_l$, window W_r, if full, is updated removing the earliest event.

After window update, for each $e \in W_r$, the concatenation of events e_{in} and e is produced as a single output event if $P(e_{in}, e) = TRUE$.



Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.0.1
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	4

Plugin_id	1001
Plugin_sid	103
Src_IP	192.168.1.0
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	1

Plugin_id	1514
Plugin_sid	600504
Src_IP	192.168.0.1
Src_Port	22
Dst_IP	192.168.1.3
Dst_Port	22
ts	0

L Plugin_id	1001
L Plugin_sid	103
L Src_IP	192.168.0.1
L Src_Port	22
L Dst_IP	192.168.1.3
L Dst_Port	22
L ts	4
R Plugin_id	1514
R Plugin_sid	600504
R Src_IP	192.168.0.1
R Src_Port	22
R Dst_IP	192.168.1.3
R Dst_Port	22
R ts	0

Figure 6. Match SNORT and CISCO Pix events with the same Src_IP.

Window update, predicate evaluation and output propagation for input events over the right stream are performed in a similar fashion.

Figure 6 shows a join operator that receives events from SNORT and CISCO Pix sensors. Events coming from the two streams are matched any time they refer to the same source IP address (Src_IP) and their distance in time is less than 10 seconds. The syntax of the Join operator is

$$J\{P = (left.e_{in}.Src\_IP = right.e_{in}.Src\_IP), time, 10\}(S_l, S_r, O)$$

3. Parallel Complex Event Processing

The ultimate goal of a SIEM is to detect **all** threats. As SIEMs are used to monitor sensitive infrastructure, all attempted or successful attacks must be discovered and any piece of information must be carefully analyzed.

Our goal in architecting a parallel correlation engine is to make sure that the parallel system identifies all threats, just as a centralized one. At the same time, we must minimize distribution overhead, in order to make parallelization cost-effective. If a parallel system provides the same results of its centralized counterpart, the former is said to be *semantically transparent*. In the following, we provide details on how to parallelize each of the operators presented in Section 2..

Semantic transparency for stateless operators is straightforward. Assume a stateless operator, e.g., a Filter, is parallelized and distributed over multiple nodes; that is, the same operator is deployed on multiple nodes and each instance processes a fraction of the input events. As stateless operators process events one-by-one, no matter which event will be processed by which instance of the parallel operator, the final outcome will match the one of a centralized execution.

However, semantic transparency becomes challenging with stateful operators. As a stateful operator keeps state across input events, we must guarantee that all events that are aggregated/correlated in a centralized execution, are processed by the same instance of the parallel operator. As an example, let us consider an Aggregate operator deployed over

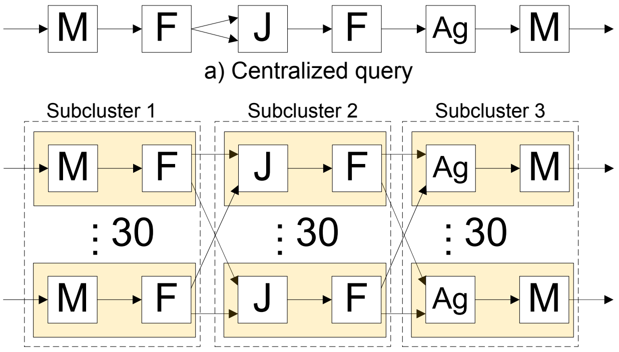


Figure 7. Query Parallelization Strategy

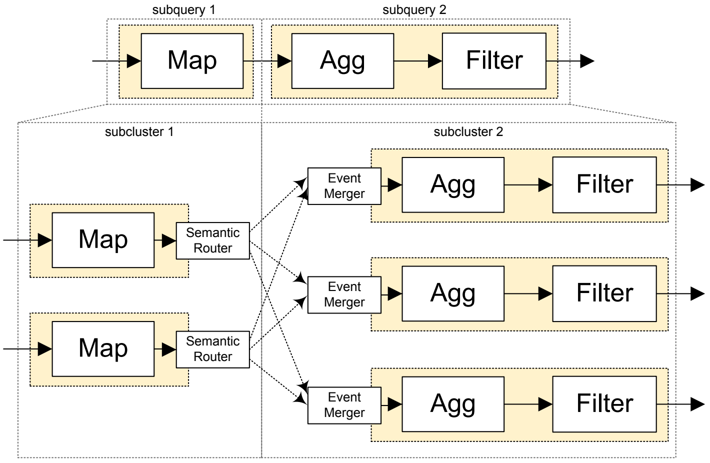


Figure 8. Subquery parallelization

multiple nodes that is used to compute the number of open connections for each server during the last hour. We must ensure that all the events related to open connections for a particular server are processed by the same Aggregate instance in order to provide the correct results, that is, the semantic provided by the distributed operator is equal to the semantic of the centralized one.

This section provides details of the proposed parallelization technique and the rationale behind our design.

3.1. Query Parallelization Strategy

Our query parallelization strategy aims at reducing the communication overhead among nodes. The rationale is that, in order to guarantee semantic transparency, communication is required only before stateful operators. Therefore, we define the parallelization unit (called *subquery*) according to stateful operators of a query. Each query is split into as many subqueries as stateful operators, plus an additional one, if the query starts with stateless operators. A subquery consists of a stateful operator followed by all the stateless operators connected to its output, until the next stateful operator or the end of query. If the query starts with stateless operators, the first subquery consists of all stateless operators before the first stateful one.

As the query of Fig. 7.a has two stateful operators plus a stateless prefix, Fig. 7.b shows three subqueries. The first one contains the prefix of stateless operators, and the next two, one stateful operator with the following stateless operators. Assuming a cluster (i.e., a set) of 90 nodes, each subquery is deployed on a subcluster of 30 nodes. Data flows from one subcluster to the next one, until the output of the system. All instances of a subcluster run the same subquery, called *local subquery*, for a fraction of the input data stream, and produce a fraction of the output data stream.

3.2. Semantic transparency

Once the query has been partitioned, semantic transparency requires that input events are carefully distributed to each subquery containing a stateful operator. That is, the output of a subquery feeding its downstream (i.e., following) counterpart must be carefully arranged so that, if the downstream subquery has a stateful operator, events that must be aggregated/correlated together are received by the same instance of the stateful operator.

In this section we show how to parallelize a query and achieve semantic transparency by means of the sample query in Fig. 8. The latter provides the IP address of the servers that managed a number of connections above a given threshold over the last hour, every ten minutes. It receives a stream of events with the schema of Table 1. The Map operator is used to extract relevant fields from incoming events. The Aggregate has windows defined over time with $Size = 3600$ seconds and $Advance = 600$; it groups connections by server (i.e., Group-by= Dst_IP) and counts the number of relevant events per server. Finally, events output by the Aggregate that carry a number of connections greater than a given threshold are forwarded to output by the Filter operator.

As the query has one stateful operator and a stateless prefix, it is split into two subqueries. Subquery1 consists of the Map operator while Subquery2 has the Aggregate and the Filter operators. They are allocated to subcluster 1 and 2, respectively.

To guarantee effective event distribution from one subcluster to the downstream one, output events of each subcluster are assigned to *buckets*. Event-buckets assignment is based on the fields of the event. Given B distinct buckets[1] and event $e = \{A_1, A_2, ..., A_n\}$, its corresponding bucket b is computed by hashing one or more of the event fields[2], e.g., A_i, A_j, modulus B, e.g., $b = Hash(A_i, A_j) \mod B$. All events belonging to a given bucket will be forwarded to and processed by the same node of the downstream subcluster.

In order to distribute the buckets across N downstream nodes, each subcluster employs a *bucket-node map* (BNM). The BNM associates each bucket with a node of the downstream subcluster, so that $BNM[b]$ provides the downstream instance that must receive events belonging to bucket b. In Figure 8, the number of nodes in subcluster 2 is $N = 3$ while B can be set arbitrarily, as long as $B \geq N$.

Event-buckets assignment and BNMs are endorsed by special operators, called *Semantic Router (SR)*. They are placed on the outgoing edge of a subcluster and are used to distribute the output events of the local subquery to the nodes of the downstream subcluster.

[1] B is a user-defined parameter.

[2] As explained later, the fields used to compute the hash depend on the semantic of the downstream operator.

As discussed above, semantic transparency requires events that must be aggregated/correlated together to be processed by the same node. However, it is also required that event processing happens in the same order as in a centralized execution. For this reason, we place another special operator, called *Event Merger (EM)*, on the incoming edge of each subcluster. EMs take multiple timestamp-ordered input streams from upstream (i.e., preceding) SRs and feed the local subquery with a single timestamp-ordered merged stream.

The bottom part of Figure 8 shows how the original query is split in subqueries, augmented with SRs and EMs and deployed in a cluster of nodes.

In the following we detail the parallelization logic encapsulated in SRs and EMs for each of the stateful operator we consider.

3.2.1. Semantic Routers

Semantic Routers are in charge of distributing events from one local subquery to all its downstream peers. To guarantee that events that must be aggregated/joined together are indeed received by the same node, SRs located upstream of a stateful subquery must be enriched with *semantic awareness*, that is, they must be aware of the semantics of the downstream stateful operator.

Join operator Assume a stateful subquery contains a Join operator. In order to guarantee correct distribution of the events, upstream SRs partition their input stream into B buckets and use the BNM to route events to the N nodes where the Join is deployed. The attribute specified in the equality clause of the Join predicate is used at upstream SRs to determine the bucket of an event. That is, let A_i be the attribute of the equality clause, then for each event e, $BNM[Hash(e.A_i) \mod B]$ determines the recipient node to which e should be sent. If the Join predicate contains multiple equality clauses, the hash is computed over all the attributes of those clauses. Therefore, events with the same values of the attributes defined in the equality clause will be sent to the same instance and matched together.

Aggregate operator If the stateful operator is an Aggregate, its parallelization requires that all events sharing the same values of the attributes specified in the Group-by parameter should be processed by the same instance. In the example of Fig. 8 the Aggregate groups events by their destination IP address. Hence, data partitioning by upstream SRs is performed in a similar way to the Join operator. That is, the set of attributes specified in the Group-by parameter are hashed and the BNM is used to partition the stream across the N instances of the downstream subcluster.

3.2.2. Event Mergers

Just like SRs, Event Mergers (EMs) are key to guarantee semantic transparency. EMs merge multiple timestamp-ordered input streams coming from upstream SRs and feeds the local subquery with a single timestamp-ordered input stream. As a result, the local subquery will produce a timestamp ordered output stream. To guarantee that the output stream is timestamp-ordered, input streams must be carefully merged. In particular, an EM forwards the event with the earliest timestamp, any time it has received at least one event from each

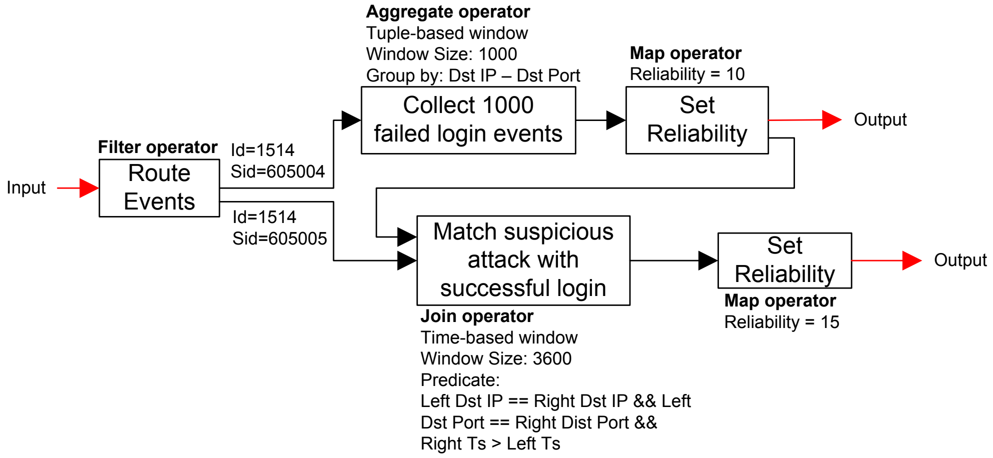


Figure 9. Detect suspicious and successful brute-force attacks

input stream. To avoid blocking of the EM, upstream SRs might send dummy events for each output stream that has been idle for the last d time units. Dummy events are discarded by EMs and only used to unblock the processing of other input streams.

4. Transforming SIEM Directives into CEP Queries

This section presents a sample directive taken from OSSIM SIEM [Ali10] and shows how it can be expressed as a CEP query and parallelized through the technique of Section 3..

The directive is designed to detect a brute-force attack against a CISCO Firewall and, eventually, its successful result. A first alarm should be raised if an unusual number of "denied login" events for a particular connection is generated (e.g., 1000 denied logins). After the first alarm has been triggered, a more severe alarm should be raised if a "permitted login" event is received for the same connection (i.e., the attacker achieved to login to the firewall).

Both denied login and permitted login events are sent from a CISCO Pix sensor ($Plugin\_id$ 1514). Denied login event is identified by $Plugin\_sid$ 605004 while permitted login event by $Plugin\_sid$ 605005. The first alarm sets reliability[3] to 10 while the second alarm sets it to 15.

The query in Figure 9 shows the implementation of the described directive in a CEP system.

The query defines one input and two output streams. Input stream schema is compliant with the one in Table 1 while the output streams share the same schema with attributes Src_ip, Dst_ip and $Reliability$.

Attributes Src_ip and Dst_ip refer to the source and destination IP addresses related to the login attempt.

An initial Filter operator is used to forward only events of interest, i.e., $Plugin\_id =$

[3] Reliability is a standard parameter of SIEM systems that defines the trustworthiness of the alarm; that is, only alarms with reliability greater than a given threshold are reported to the user.

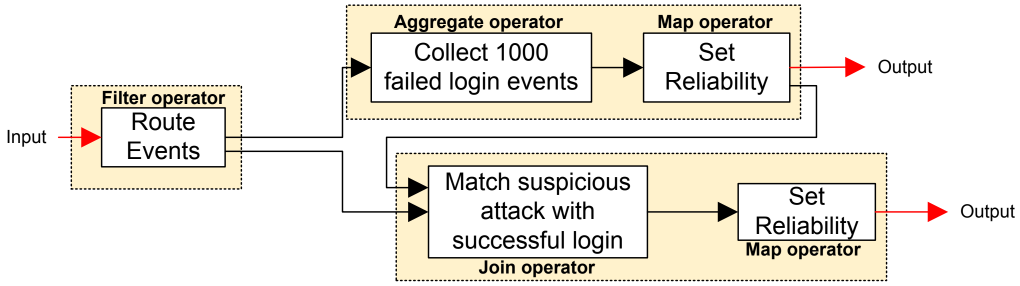


Figure 10. Query Partitioning

1514 and $Plugin\_sid \in \{605004, 605005\}$; all other events are discarded. Events related to a denied login ($Plugin\_id = 1514$ and $Plugin\_sid = 605004$) are processed by an Aggregate operator.

The Aggregate $Wtype$ is set to $numEvents$ and has $Size = 1000$ and $Advance = 1$. Group-by is set to Dst_IP, Dst_Port, that is, the Aggregate keeps separate windows for each port on each destination host. Any time 1000 denied login events are received for the same pair Dst_IP, Dst_Port, an output event is forwarded to the following Map operator. The output event carries information about the attack target (i.e., Dst_IP, Dst_Port) and the timestamp of the earliest denied login event.

The Map operator takes the input event, adds attribute $Reliability = 10$ and forwards the event over the output stream. The events produced by the Map operator are sent to the query output and also forwarded to the Join operator for further correlation.

The Join operator is used to correlate suspicious brute-force attempt alarms coming from the Map operator with successful login events forwarded by the Filter.

The Join operator predicate P matches two incoming events if they refer to the same destination (i.e., if they share the same Dst_IP, Dst_Port attributes) and if the successful login event timestamp is greater than the timestamp of the brute-force attempt alarm (i.e., the successful login happens after the brute-force attack). Windows size is equal to 3600 seconds. This implies that two matching events cannot be distant in time more than one hour.

Events produced by the Join operator are forwarded to another Map operator that sets $Reliability = 15$. Produced events are forwarded to the query output.

Figure 10 shows how the query in Fig. 9 is partitioned according to the technique described in Section 3.1..

An initial subquery is allocated for the stateless prefix of the original query (i.e., the Filter operator). Another subquery contains the first stateful operator and its stateless suffix, until the next stateful operator (i.e., the Aggregate operator and Map1 operator). A final subquery is allocated for the Join operator and Map2 operator, that is, the last stateful operator and its stateless suffix.

Once the original query has been partitioned into subqueries, a Semantic Router operator and an Event Merger operator are added for each edge connecting two distinct subqueries. Figure 12 shows the three subqueries (SQ1, SQ2 and SQ3) augmented with Sematic Router and Event Merger operators.

The Filter operator will have one SR for each outgoing edge, SR_{F1} and SR_{F2}.

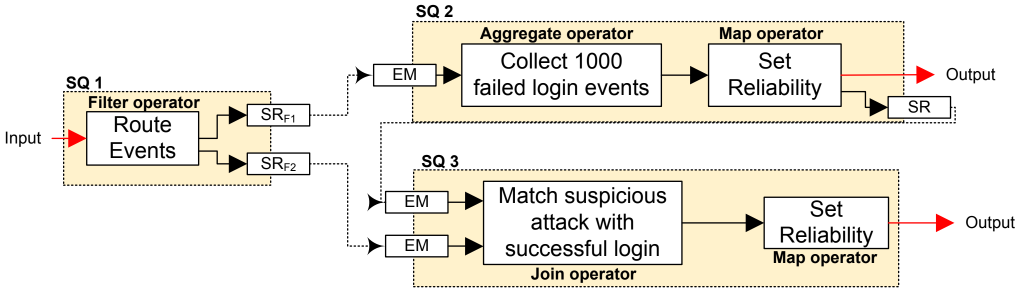


Figure 11. Subqueries with Semantic Routers and Event Mergers

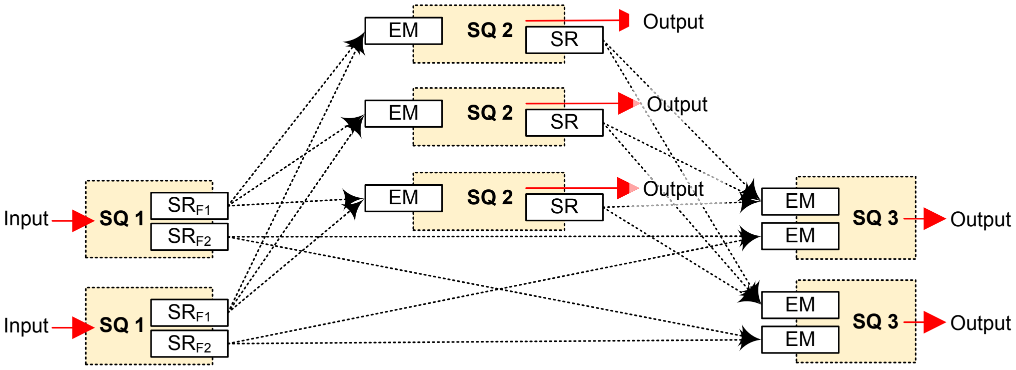


Figure 12. Subquery deployment

The former will forward events with $Plugin\_id = 1514$ and $Plugin\_sid = 605004$ while SR_{F2} will handle events with $Plugin\_id = 1514$ and $Plugin\_sid = 605005$. The Aggregate operator specifies Group-by$=$ Dst_IP, Dst_Port while the Join operator has the same attribute pair in the equality clause of its predicate. Hence, for each incoming event e_{in}, each SR on the outgoing edge of the first subquery will compute $Hash(e_{in}.Dst\_IP, e_{in}.Dst\_Port) \mod B$ to compute the bucket of the event and then it will look up the BNM to identify the designated recipient among the instances of the downstream subquery.

The subquery containing the Aggregate operator is enriched with an Event Merger on the incoming edge and one Semantic Router to forward output events to the next subquery. Here as well, SR will use attributes of Dst_IP, Dst_Port of each incoming event to compute the bucket of the input event and then the designated receiver among the instances of the downstream subquery.

The subcluster containing the Join operator is enriched with two EM operators, one for each input stream.

Finally, Figure 12 shows subqueries SQ1, SQ2, SQ3, allocated to subclusters of 2, 3 and 2 nodes, respectively.

5. Conclusion

In this chapter we propose to use Complex Event Processing systems as the correlation engine of next generation SIEMs. We argue that current correlation engines relying on a centralized execution are not suitable for complex scenarios where high-throughput data must be processed with tight timing constraints. We show how Complex Event Processing can be used to express SIEMs directives and provide a novel parallelization technique that allows to parallelize the execution and deploy the engine over an arbitrary number of nodes in a shared-nothing cluster. The parallel system can handle input load far behind the ones afforded by single-node correlation engines. At the same time, our parallelization technique guarantees semantic transparency, i.e., it guarantees that the parallel execution provides the same output of an ideal centralized execution. The result is a highly-scalable correlation engine that can be easily embedded in next generation SIEMs to process massive amount of input data and make sure that no threats are "missed".

References

[AAB$^+$05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR 2005*, pages 277–289, 2005.

[ACc$^+$03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[Ali10] AlienValult. AlienVault Unified SIEM. http://www.alienvault.com/, 2010.

[CBB$^+$03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[CCD$^+$03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 379–390, 2000.

[GGR12] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2012.

[GJPPMV10] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Patrick Valduriez. Streamcloud: A large scale data streaming system. In *International Conference on Distributed Computing Systems (ICDCS'10)*, pages 126–137, 2010.

[Mil03] David L. Mills. A brief history of ntp time: memoirs of an internet timekeeper. *Computer Communication Review*, 33(2):9–21, 2003.

[Pre10] PreludeIDS Technologies. Prelude PRO 1.0. http://www.prelude-technologies.com/, 2010.

[SHCF03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *19th International Conference on Data Engineering (ICDE'03)*, pages 25–36, 2003.