# Distributed Pattern Detection for Intrusion Detection

Christopher Krügel
Distributed Systems Group
Technical University Vienna
chris@infosys.tuwien.ac.at

Thomas Toth
Distributed Systems Group
Technical University Vienna
ttoth@infosys.tuwien.ac.at

## Abstract

*Evidence of attacks against a network and its resources is often scattered over several hosts. Intrusion detection systems therefore have to collect and correlate information from different sources. For this purpose, distributed data is forwarded to dedicated hosts where it is further processed. Such a design renders the whole ID system vulnerable to attacks against these special nodes. As networks and traffic grow, they also become performance bottlenecks. We propose a completely decentralized approach that models an intrusion as a pattern of events that occur at different hosts and which does not rely on dedicated correlation entities to detect them. We present a specification language for these patterns and a distributed algorithm to find events that satisfy them. The theoretical properties of our solution are reviewed and experimental data is provided.*

## 1 Introduction

Intrusion detection systems (IDS) are network security tools that process local audit data or monitor network traffic to search for specific patterns (misuse based) or certain deviations from expected behavior (anomaly based) which indicate malicious activities against the protected network. The traces of a simple attack are often visible in a single log-file or can be monitored at a single network interface card. However, advanced hackers[1] do not concentrate on a single host alone but try to disguise their actions by distributing them over several machines. Each single activity considered for itself looks innocent but in their entirety these actions constitute an attack. This makes it necessary to collect and relate audit data from different sources (a process called *event correlation*).

[1]The term hacker is used to describe a persons with the malicious intend to gain unauthorized access to network resources.

As Zamboni [1] pointed out, most existing IDSs perform their data processing centrally, despite their distributed data collection [5, 10]. This causes limitations in their scalability, ease of configuration and fault tolerance. The failure of the central unit completely deactivates the correlation process and effectively blinds the IDS. The processing capacity of this node also limits the number of events it can handle in a certain amount of time. When too many sensors are forwarding their messages to the central host the resulting backlog increases the reaction time of the system or might even cause data loss.

To circumvent such shortcomings, hierarchical designs have been introduced. Systems like Emerald [8], GrIDS [11], AAFID [2, 1] or NetSTAT [13] have a layered structure where data is locally preprocessed and filtered. Only events that might be part of a distributed attack scenario are forwarded to a higher level entity. Emerald [8, 7] uses a publish/subscribe system to disseminate relevant data between nodes. Nevertheless, these systems use dedicated nodes that act as central points for collecting data from remote sensors.

Although hierarchical structures and filtering at low levels allow better scalability, the systems are still vulnerable to faults and overloading of nodes that are close to the root of the hierarchy.

We attack the inherent problems of centralized, dedicated nodes by proposing a completely decentralized approach where the detection of an intrusion is restricted to those nodes where parts of the attack are directly observable. Only a few systems have already attempted to use a similar approach. The best known is CSM (Cooperating Security Managers) [14], a design which distinguishes between a local ID component and an information forwarding unit at each node. The forwarding unit allows to exchange information between nodes along the login chain of users. While this system was the first to show the possibility of distributed cooperation in principal, its applicability is limited by the fact that cooperation is only done along user's login chains. Another system called Micael [3] proposes mobile agents to accomplish the task of distributed event correlation without a central entity but its current state is unclear.

We describe an algorithm that is capable of detecting patterns of events that occur at multiple nodes of a network in a completely decentralized fashion. The specification language to define distributed patterns is explained as well. We evaluate our design and compare the results to traditional intrusion detection systems that attempt to correlate data in a hierarchical or centralized fashion.

## 2 System overview

An intrusion is defined as a pattern of basic events that can occur at multiple hosts. A basic event is characterized as the occurrence of something of interest that could be the sign of an intrusion (e.g. the receipt of a certain IP packet, a failed authentication or a password file access). Such events could either stem from a local misuse or an anomaly incident. We intend to integrate third-party systems to perform the local detection and feed their event data into our correlation algorithm (currently, Snort [9] is used to obtain basic events).

By relating events from multiple nodes, one can detect a number of attacks that would remain unnoticed by only focusing on local activity. One type of attacks can be found by checking for suspicious signatures of network connections between hosts as described in GrIDS [11]. This includes worms spreading in a network and telnet chains (a number of consecutive telnet logins by an intruder to hide his tracks). A connection that is established between two computers looks harmless when monitored locally at one machine but when the whole network is considered, suspicious chain or tree patterns might emerge.

Relating events between different hosts might also increase the chance for anomaly detection sensors to catch an attack. Consider a port scan from a certain machine that is scanning each host on a subnet for an open port 80 and 8080. Whenever the firewall (or port scan detector) monitors such an activity, it forwards this information (including the source of the scan) to the web server. When the web server itself later receives abnormal traffic exactly from that source an attack is assumed. The aim of this information flow is that each local node (or sensor) gets a better understanding of what happens inside the whole network. The combination of local information from different places leverages the understanding of the network traffic each participating node monitors.

The detection capability of anomaly detection sensors can usually be raised by lowering the threshold that separates normal from suspicious behavior. In case an anomaly sensor gets additional information indicating that a certain host already has taken offensive actions against different other hosts or the whole network, it can selectively become more sensitive when analyzing traffic from that machine. We want each host to be aware of activities that manifest tehmselves at different parts of the network and influence the network traffic. Following [4] our nodes should develop an understanding of the complex activities inside the network, an idea we call *Network Awareness*.

Our distributed patterns and the detection algorithm can describe and detect situations where a sequence of events occurs on multiple hosts (like port scans followed by abnormal packets). This is used to modify the reaction of sensors at nodes that get aware of emerging, hostile patterns.

The decentralized pattern detection process finds distributed patterns by sending messages between nodes where interesting events occur. Therefore, each node of the protected network has to run a process that executes the distributed pattern detection algorithm. The detailed description of the layout of such events and patterns as well as the detection algorithm forms the core of the paper. The renunciation of dedicated central components and the effort of designing a fully distributed system is rewarded by good scalability and fault tolerance properties of our IDS. When a single node in our system fails (or is compromised), it stops its local detection and ceases to forward pattern information. This prevents the detection of pattern instances where events occur at the compromised host, but the rest of the system remains intact. In addition, messages are not sent to designated nodes but exchanged between equal peers. This allows to distribute the complete message traffic over the network without some predefined central bottlenecks. We are aware of the fact that a distributed system design might result in tremendous message overhead. This potential danger is addressed at the levels of pattern specification and detection.

## 3 Pattern specification

The design of our pattern specification language is guided by two conflicting goals. The first goal demands a language that should be as expressive as possible. It would be desirable to allow the description of complex relationships between events on different hosts using regular or tree grammars. As our system relies on peer-to-peer message passing between hosts without a central coordination point, arbitrary complex patterns might cause the data that needs to be exchanged to explode. In the worst case each node has to send all its data to every other node. This conflicts with the second goal, which demands that the amount of data that has to be transferred between hosts should be as small as possible. Therefore we have to impose limitations on the expressiveness of our pattern language.

As stated above, an event is the smallest entity of a pattern and defined as the occurrence of something that might be part of an intrusion. We have designed a simple language (called Event Definition Language - EDL) that allows to specify an event as a set of attributes with their types (e.g. string, integer). In order to provide the system with EDL data, we have to install sensors that watch for the occurrence of interesting events and transform them into EDL objects by setting the given attributes to the actual values derived from the observed event instance.

A pattern describes activities on individual hosts as well as interactions between machines. The basic building block of a pattern is a sequence of basic events that happens locally on one machine (called *host sequence*). One can specify a list of events at a local host by enumerating them and imposing certain constraints on their attributes. We distinguish between constraints which relate single event attributes with constant values and constraints which relate different attributes of events using variables. One can use the standard logical operators for both types and an extended set of operators (including `in` and `range`) to relate attributes with constant. A connection (context) between event sequences on different hosts is established by *send events*.

**Definition:** *A pattern* $\mathsf{P}$*, relating events that occur at* $n$ *distinct hosts, consists of* $n$ *sequences of events, one for each node (an event sequence at a single node is called* host sequence*).*
*A set of events* $\mathsf{S}_A$ *at host* $\mathsf{A}$ *is linked to a set of events* $\mathsf{S}_B$ *at host* $\mathsf{B}$*, iff* $\mathsf{S}_A$ *contains a send event to host* $\mathsf{S}_B$*.*
*Any event that refers to a remote host (e.g. the sending of a packet to a host, the reception of a packet from a host) might be used as a send event.*

The detection of a port scan that attempts to find open HTTP ports can be used as a send event when the address of the web server (the referred host in this example) is known to the host that detects the scan. It is only required that the target $\mathsf{B}$ of the send event can be determined locally at $\mathsf{S}_A$ (from the event data). The first event of $\mathsf{S}_B$ has to be the next event to occur after the send event in $\mathsf{S}_A$. It is required that the send event is the last event in $\mathsf{S}_A$.

**Definition:** *Pattern* $\mathsf{P}$ *is valid, iff the following properties hold.*

1. *Each set of events is at least linked to one other set.*
2. *Every set except one (called the* root set*) contains exactly one send event as the last event of the host sequence. The root set contains no send event.*
3. *The connection graph contains no cycles. The connection graph is built by considering each event set as a vertex and each link between two sets as an edge between the corresponding vertices.*

These definitions allow only tree-like pattern structures (i.e. the connection graph is a tree), where the node with the root set is the root of the tree. Although this restriction seems limiting at first glance, most desirable situations can be described. Usually, activity at a target host depends on events that have occurred earlier at several other hosts. This situation can be easily described by our tree patterns where connection links from those hosts end at the root set.

The case where events on two different nodes both depend on the occurrence of a single event at a third node cannot be directly expressed in our pattern language (as the root set would contain two send events). Nevertheless, a centralized application might split the original, illegal pattern into subpatterns (each representing a legal tree like structure) and relate the results itself.

### 3.1 Attack Specification Language

This section briefly describes the syntax and semantics of our pattern description language (called Attack Specification Language - ASL).

A pattern definition is written as follows

```
ATTACK "Scenario Name" [ nodes ] pattern
```

The *nodes* section is used to assign an identifier to each node that is later referred in the pattern definition.

The *pattern* section specifies the pattern. It consists of a list of event sets, one for each node that appears in the node section. The event set is a list of identifiers, each describing an event. A predefined label called `send` is used to identify the target node of send events.

Each event can optionally be defined more precisely by constraints on the event's attribute values. These attribute values can be related to constant values or to variables by a number of operators (`=`, `!=`, `<`, `>`, `>=` and `<=`) or to constant values by a `range` or an `in` operator as defined below.

x range $(x_0, x_1) \leftrightarrow x_0 \leq x \leq x_1$
x in $(x_0, x_1, ..., x_n) \leftrightarrow \exists$i (0 $\leq$ i $\leq$ n) and x = $x_i$

A variable is defined the first time it is used. One must assign a value (bind an attribute value) to each defined variable exactly once while it may be used arbitrarily often as a right argument in constraint definitions. The scope of variables is global and its type is inherited from the defining attribute.

With these explanations, we introduce the (incomplete) syntax (in BNF) of the pattern section (all identifiers represent strings).

```
pattern      : {event set}+
event set    : node-id '{' {event}+ '}'
event        : ['send('target-id'):'] event-id
               '[' {constraint ';' }* ']'
constraint   : assignment | [label] relation
assignment   : '$'variable-id '='
               ( attribute | constant )
relation     : attribute operator
               ['('] {value ',' }* value [')']
value        : constant | '$'variable-id
attribute    : event-attribute-id
operator     : '='|'!='|'<'|'>'|'>='|'<='|
               'in'|'range'
```

The following example shows a classic telnet chain scenario.

```
ATTACK "Telnet Chain" [ Node1, Node2 ]
Node1 {
  send(Node2): tcp_connect [ DstPort == 23; ]
}
Node2 {
  tcp_connect [ DstPort == 23; ]
}
```

It describes a connection from `Node1` to port 23 at `Node2` and from there to port 23 of another remote machine. `Node2` describes the root set (i.e. has no outgoing send event). The target of the send event can easily be extracted from the `tcp_connect` event as the destination IP address. This fact is specified in its EDL definition.

# 4 Pattern detection

The purpose of the pattern detection process is to identify actual events that satisfy an attack scenario (written in ASL). When a set of events fulfills the temporal and content based constraints of a scenario an alert is raised. Notice that instead of simply sending a message to a central system administration console (that yields again a single point of failure), more sophisticated responses can be implemented. The node itself can issue commands to reconfigure the firewall or to terminate offending network connections, thereby elimination the single point of failure introduced by the central console of a human operator.

## 4.1 Basic data structures

In order to be able to process an attack description, it has to be translated from ASL into a data structure suitable for our system.

**4.1.1 Pattern graph:** This is done by transforming a scenario into an acyclic, directed graph (called *pattern graph*). An attack scenario describes sequences of events located at different hosts that are connected by send events. Each single event specified by an ASL scenario is represented as a node of the resulting graph. The nodes of each host sequence are connected by directed edges. An edge leads from a node representing a certain event to the node which represents the immediate successor of that event in the ASL pattern description. Send events require a little different treatment as they are the last event in their host sequence and therefore do not have an immediate successor. In this case a directed edge leads to the first node of the host sequence where the send event points to.

The resulting graph shows a tree shape and all paths through the graph end at the last event of the root set's sequence (called *root node*). Each node receives a unique identification number that consists of a part that identifies the attack scenario itself and a part that identifies each node within the scenario. The following example (see Figure 1 below) shows the result of such a transformation, which is actually straightforward as ASL only allows tree shaped patterns. The attack scenario describes a pattern of a potential attack against a web server by a hypothetical variant of the `CodeRed` worm. Our variant does not only scan for an open port 80 but also attempts to retrieve the type of operating system the web server runs by asking the DNS server for the web server's HINFO (hardware and OS info entry) entry. This allows to target Microsoft machines more accurately. Whenever a port scan detector notices a scan against port 80 from a certain IP address and the DNS server gets HINFO queries from the same address and finally the web server receives an HTTP request from that machine, an alarm is raised as such behavior is presumably suspect.

**4.1.2 Messages:** The detection algorithm does not deal with events itself, instead it operates on messages. A message is a compact, more suitable representation of an event. Most attack descriptions rely only on a small subset of the event's attributes for correlation (e.g. only IP addresses instead of the complete IP

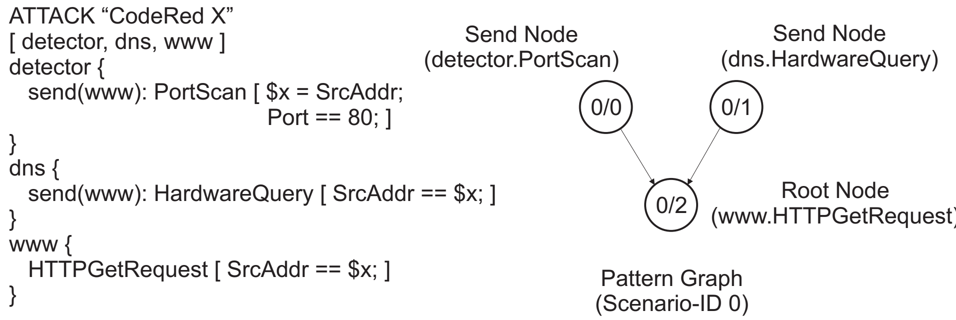



Figure 1: Pattern Graph Transformation

header). In ASL, only attributes that are assigned or compared to variables are of interest to the further detection process. Therefore there is no need to operate on the complete event objects.

It is easy to see that a single event can match the description of multiple event patterns in an attack scenario. Thus, if more than one description is matched several message instances (one for each matching pattern) are created. Whenever a message is created all relevant attributes (i.e. the attributes that are assigned or compared to variables in the ASL description) are copied into it. Then it is forwarded to the node representing the matching event description for further processing.

Each message can be written as a triple <id, timestamp, list of (attribute,value)>. The id of the message is set to the identification of the node. The timestamp denotes the occurrence of the original event and the attribute/value list holds the values of the relevant event attributes (which have been copied from the original event attributes). The id of a message defines its type. Different actual message instances with identical ids are considered to be of the same message type.

It is possible that messages of different types receive different attributes from a single event - depending on which ones are actually used in the attack description. In addition, the attribute/value list can be empty when the corresponding ASL event pattern does not reference any variables at all. In Figure 1 a port scan event that targets port 80 from IP address 128.131.0.1 would cause the creation of the message instance <0/0, time_of_occurrence, (SourceAddr, 128.131.0.1)> of type 0/0.

## 4.2 Constraints

An attack description in ASL imposes a number of different constraints on the events that must be taken into account by the detection algorithm. The set of constraints can be divided into a *static*, a *dynamic* and a *temporal* subset.

**4.2.1 Temporal constraints:** The paths through the pattern graph reflect the temporal relationships of events. Event A has to happen before event B if and only if B is on the path which leads from A to the root node. The events of a host sequence have to occur in the same order as they are defined in the ASL description. When a host sequence is linked to another host sequence by a send event, all events of the destination sequence have to occur after the send event in the source sequence.

**4.2.2 Static constraints:** An event pattern that relates an event attribute to a constant value imposes a static constraint onto events (e.g. the equality relation between the `Port` attribute and the value `80` in the `PortScan` event in Figure 1). Static constraints are easy to evaluate immediately as soon as a new event of the appropriate type has been received. When an event satisfies all static constraints of a certain node (respectively its corresponding event pattern) a new message instance is created and forwarded to that node. Static constraints are used to decide which messages need to be created from a certain event but are not used later during the actual detection process.

**4.2.3 Dynamic constraints:** A dynamic constraint is introduced by the use of variables in an attack description. The definition of a variable in an event pattern and the subsequent use of this variable in other event patterns introduces relationships between attributes of different events. Although it is possible to define and use the same variable within a single event pattern such a variant can be trivially handled by the more general approach.

The definition of a variable by a certain event attribute and its subsequent use as an operand in a relation with another attribute creates a direct relation between these two attributes. In Figure 1 above, the definition of variable `x` as the the value of attribute `SrcAddr` in the `PortScan` event description and its use in the equality operations with the attributes of the `HardwareQuery` and `HTTPGetRequest` events create the following two dynamic constraints.

```
[PortScan.SrcAddr == HardwareQuery.SrcAddr ]
[PortScan.SrcAddr == HTTPGetRequest.SrcAddr]
```

Attributes that define or are related to variables are copied into messages. Therefore, it is possible to express the relationship between event attributes as (dynamic) constraints on the values of their corresponding message types. It is obvious that it cannot always be immediately determined whether an event satisfies its dynamic constraints, hence events that satisfy all static constraints of a certain event pattern cause a message to be created and passed to the appropriate node (the one which is associated with the pattern). It is the task of the actual detection process to resolve all dynamic and temporal constraints.

### 4.3 Detection process

The basic detection process can be explained as follows. We have already mentioned that events cause messages to be forwarded to their corresponding nodes (to the nodes that are associated with patterns matched by the event). The messages may then be moved along the directed edge of the graph to other nodes according to certain rules. The idea is that each node can be considered as the root of a subtree of the complete tree pattern. There are **node constraints** assigned to each node of the graph such that if there are messages which satisfy the node constraints, there are events that fulfill the dynamic and temporal constraints of the complete subtree above that node. Whenever the node constraints of a node are satisfied certain messages may be moved one step closer to the root node, hence they are pushed over the node's outgoing edge to its neighbor node below (as we have a tree shaped graph, there is at most one outgoing edge for each node). Then these messages are processed at the destination node. This allows to successively satisfy subtrees of the complete pattern and move messages closer to the root node of the pattern graph. Whenever messages at the root node fulfill the constraints there, the pattern has been detected (i.e. there exist events that fulfill all constraints of the attack scenario).

The advantage of this approach is the fact that only local information is necessary to decide which messages should be forwarded. This allows to actually distribute nodes of the pattern graph over several hosts and have each node make local decisions without a central coordination point. Different host sequences may potentially occur at different hosts.

**4.3.1 Node constraints:** The node constraints have to make sure that all events described by the subtree pattern have occurred, that their temporal order is correct and that all dynamic constraints (which can be resolved up to this point) are met. The messages that are important for a certain node to satisfy its node constraints belong to one of the following three groups.

- Messages that are created from events that match the event description of the node itself (i.e. that have the same id as the node). It is obvious that in order to satisfy a pattern, one event for each node of that pattern is needed. In order to fulfill a subpattern originating at a node it is necessary to receive one message created from an event that matches the local event description itself (such a message is called a *local message* for that node).
- Messages that are created from events that match the event description of the node's immediate predecessors. Usually each node has only one predecessor but this number can vary for the first node of each host sequence. These nodes may have more than one predecessor or non at all.
- Messages whose value(s) are used in at least one dynamic constraint at that node.

The node constraints consist of

1. the set of temporal constraints between the local message and the predecessor nodes' local messages **and**
2. all dynamic constraints that can be resolved at this node

The set of the temporal constraints between the local message and its predecessor messages guarantees that events described by the local node and by all its immediate predecessors have occurred. As messages from predecessor nodes may only be forwarded by them when the events at their predecessor nodes have occurred as well, it is assured that all events specified by a subtree pattern have taken place in the correct temporal order. The node constraints have to be modified for nodes without predecessors. For those it is only necessary that a local message exists.

A dynamic constraint between attributes of two events can be resolved as soon as both operands are available. When messages representing the two events are on-hand, their relation can be evaluated and one can determine whether the dynamic constraint is satisfied or not. Therefore, every dynamic constraint (i.e. a variable definition at one node and its use at another one) is inserted into the pattern graph at the earliest node possible. The earliest possible node is determined by finding the first common node in the paths from each of the constraint operands to the pattern graph's root node. When one node is on the path of the other one, the constraint is inserted directly there, otherwise it is inserted at the node where both paths merge. A pattern graph with dynamic constraints is shown in the lower half of Figure 2 (the table will be explained later).

A problem arises when transitive relations are introduced by relating a single message to several other messages. The attributes of events that are independent at first glance become linked by being related to a common, third event. In such a case, it is not enough to insert the constraints at the earliest possible node.

Consider the pattern graph in Figure 3 and suppose that the messages <0, t1, (a1,0)>, <0, t2, (a1,1)>, <1, t3, (a2,0)>, <1, t4, (a2,1)> and <3, t5, (a3,0)> are received in that order. The first four messages (the first two from node 0, the next two from node 1) are eventually passed to node 3 as the value of the first and the third message (which is 0) as well as the value of the second and fourth message (which is 1) are equal (dynamic constraint evaluated at node 2). As the attributes of messages with id 1 and 2 are not compared again at node 3 the value of the final (fifth) message is equal to the value of the first message and smaller than the value of the forth one. This results in an illegal report of a successful match.

To prevent this problem, all dynamic constraints that are connected by having common messages as their operands are combined in a subset of the scenario constraints called a *cluster*. When a dynamic constraint operates on messages that are used in no other dynamic constraints the message itself becomes a cluster. In Figure 3 all three dynamic constraints are part of a single cluster.

In addition to the insertion of each constraint at the earliest possible node, all constraints of a cluster are additionally inserted at the cluster root node (but obviously no duplicate constraints are inserted). Similar to the situation with a single constraint, the cluster root node is the first common node of all the paths that lead from each operand of every cluster constraint to the root node of the pattern graph. With these additional constraints the messages shown above do not result in a false detect.

#### 4.3.2 Message and bypass pool:

Each node has a *message pool* and a *bypass pool*. The message pool is a place that stores message instances that can potentially be used to satisfy the local node constraints. The bypass pool holds message instances that can potentially satisfy node constraints of nodes that are closer to the root of the pattern graph (but which are not used for the current node constraints). Messages in the bypass pool are forwarded as soon as their temporal constraints are met.

After the node constraints have been determined, it is easy to calculate the types of the messages for the message and the bypass pool. Obviously, the message pool for each node consists of all message types that are used in at least one of its node constraints.

The message types needed for the bypass pools are determined next. For each message type, every node on the path between the first and the last use of messages of that type is examined. When the message type is not contained in the message pool of a node on that path, it is added to the bypass pool there. This assures that messages which are needed to determine node constraints at nodes closer to the root are correctly forwarded there.

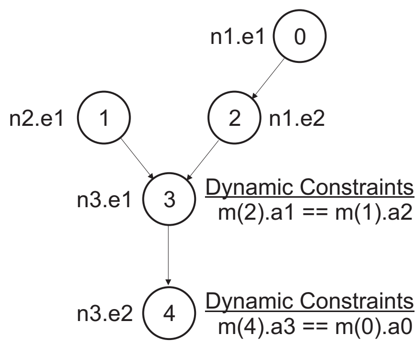
The table in the upper half of Figure 2 shows the node constraints and the types of messages that must be inserted into the message and bypass pools for the given pattern graph.

#### 4.3.3 Detection algorithm:

Having determined the node constraints for each node (which make sure that the subtree pattern above this node is satisfied) as well as the message and bypass pools, the algorithm to actually move messages between nodes can be explained. The id of a newly arrived message is checked to determine whether it should go to the message or to the bypass pool. When it belongs to neither group, it is simply discarded. This prevents messages that are not needed anymore from being moved further toward the root node. When the message belongs to the bypass pool it is put there and no immediate further actions are necessary, otherwise it is added to the message pool. Whenever a new message is inserted into the message pool the node constraints are checked. The algorithm attempts to find a tuple of messages of *different* type (i.e. all with distinct identifications) that match *all* the node constraints. The tuple has to include one actual message instance of each message type (i.e. message id) of the message pool and the new message has to be part of the tuple as well. Consider a potential tuple for node 3 in Figure 2 with its message pool {m1, m2, m3}. It must consist of a message instance with ids 1, 2 and 3. When such a

| Nodes | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Node Constraints | ∃m(0) | ∃m(1) | m(2).time > m(0).time | m(3).time > m(1).time<br>m(3).time > m(2).time | m(4).time > m(3).time |
| Dynamic Cons. | | | | m(2).a1 == m(1).a2 | m(4).a3 == m(0).a0 |
| Message Pool<br>Bypass Pool | {m(0)} | {m(1)} | {m(0), m(2)} | {m(1), m(2), m(3)}<br>{m(0)} | {m(0), m(3), m(4)} |

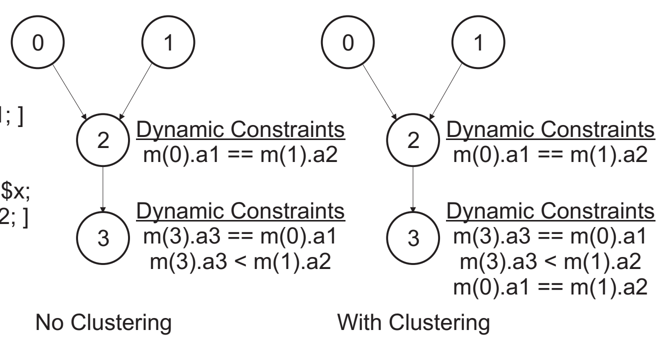
```
ATTACK "Sample"
[ n1, n2, n3 ]
n1 {
   e1 [ $x = a0; ]
   send(n3): e2 [ $y = a1; ]
}
n2 {
   send(n3): e1 [ a2 == $y; ]
}
n3 {
   e1 [ ]
   e2 [ a3 == $x; ]
}
```




The occurrence of event n1.e2 results in the creation of message
<2, time of occurrence, (a1, value of a1)>

Figure 2: Complete Pattern Graph

```
ATTACK "Cluster"
[ n1, n2, n3 ]
n1 {
   send(n3): e1 [ $x = a1; ]
}
n2 {
   send(n3):  e2 [ a2 == $x;
                   $y = a2; ]
}
n3 {
   e3 [ ]
   e4 [ a3 == $x;
        a3 < $y; ]
}
```




m(x).y indicates the value of attribute y of a message with id x

Figure 3: Constraint Clustering

tuple (or tuples, when more than one set of messages match the node constraints) can be identified, the detection process has found a number of messages that match the subtree pattern starting at the local node. The tuple's messages have to be moved over the outgoing link to the next node. Because messages in the message pool might be needed later to satisfy the node constraints together with newly arriving messages, the original messages remain in the pool and only copies are forwarded. In order to prevent the system from being flooded by duplicate messages, each message pool entry is only copied and forwarded to the next node once. For each tuple that matches the local constraints the bypass pool is inspected. The temporal constraints between each message in the bypass pool and all messages of the matching tuple are checked. When a bypass pool message satisfies all temporal constraints between itself and each tuple message, it is removed from the pool and moved to the next node. This is needed to make sure that only messages which do not violate any temporal constraints are passed on.

The situation is slightly different for send nodes. As a send node can have different next neighbor nodes at different hosts (depending on the target of the send event) the copying of message pool entries and the deletion of bypass pool elements must be handled differently. The send node has to keep track which message pool entries have already been copied and which bypass pool elements have already been removed and forwarded to the destinations of the send events for each different destination. This implies that bypass pool elements can never be deleted because they might have to be sent to a completely new destination host. It is obvious that elements cannot be kept infinitely long because memory is a limited resource. We use timers to remove elements from the message and the bypass pools after a certain, configurable time span. This means that patterns which evolve over a long time might remain undetected. Note that this is not a limitation of our approach but a problem that affects all systems that operate online and have to keep state. Such systems need a policy that decides which events to delete when the available memory is exhausted.

The following example in Figure 4 shows a step-by-step detection of the distributed pattern which is described by the scenario in Figure 2. The node constraints of Figure 2 are used and each tuple is underlined in the figure. Dotted arrows indicate the copying of messages to the next neighbor. Associated with each node are two sets enclosed in brackets. The first holds the node's current message pool entries, the second its bypass pool elements.

## 5 System evaluation

The aim of this section is to show that the proposed detection process operates as efficient as current solutions while providing superior fault tolerance and scalability properties. This makes it necessary to define the evaluation criteria that we use to measure these properties.

We measure *fault tolerance* as the percentage of nodes of the complete network which have their events correlated after a single machine running parts of the IDS (sensor or correlator) fails or is taken out. This indicates the percentage of distributed patterns that can still be detected. When a node failure partitions the set of hosts into several subsets where events are still related within each of these subsets, the highest percentage among all of them is chosen. When a correlator that is responsible only for a subset of all nodes fails, the remaining system may still perform event correlation on a reduced set of hosts. The fault tolerance measures exactly that fraction of nodes.

The *scalability* of distributed intrusion detection systems is characterized by two values. One indicates the total network traffic between all nodes (total traffic) while the other measures the maximum network traffic at a single node (peak traffic).

We compare our completely decentralized system (distributed approach) to a design that deploys sensors at every host and centrally collects their data (centralized approach) and to one that introduces several layers of processing nodes (hierarchical approach) on top of the sensors which forward data that might be part of a larger attack scenario to upper level sensors. An example of a centralized system is NSTAT [6], while Emerald [8] or AAFID [2, 1] follow a hierarchical approach.

For our theoretical discussion, we assume a network with n hosts and the occurrence of n*e interesting events during a time interval of length $\Delta$. The interval $\Delta$ also specifies when messages 'time out' and are removed from the detection process. While the number of events in the whole system is assumed to be proportional to the number of nodes, the number of events at each single host may not exceed a certain threshold $\tau$. This is reasonable as it allows a certain variance of the distribution of events within the system (i.e. modelling local hotspots like WWW or NFS servers in very large networks) without allowing a single node from having to deal with arbitrary many events as the number of nodes grows larger.

The coverage of a network (in %) after a single node failure is given below for the different systems. We assume that the hierarchical system uses l =

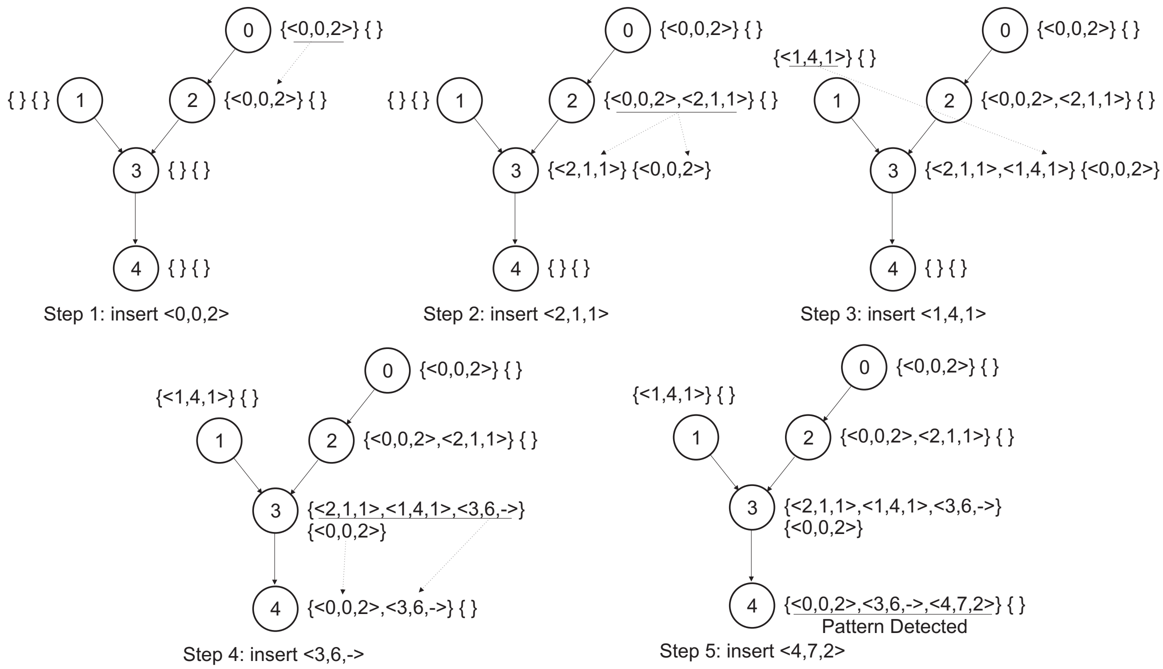



Figure 4: Sample Pattern Detection

$\lfloor \log_m((m-1)*n) \rfloor$ layers with $m^k$ nodes (k ... 0 to l-1) in each layer, where m specifies the number of children for each node.

| System | Type of Node | Coverage |
|---|---|---|
| Centralized | Sensor | $\frac{n-1}{n}$ |
| | Correlator | 0 |
| Hierarchical | Sensor | $\frac{n-1}{n}$ |
| | Correlator | |
| | (at layer k) | $n - \frac{m^{l-k}-1}{m-1}$ |
| | Root (layer 0) | $\frac{n-1}{m}$ |
| Decentralized | Node | $\frac{n-1}{n}$ |

The loss of a node at layer k in the hierarchical model stops correlation of the complete subtree with $\frac{m^{l-k}-1}{m-1}$ nodes. When the root is lost, each subtree with $\frac{n-1}{m}$ nodes can still do correlation. Not surprisingly, this shows that centralized and hierarchical system are more vulnerable especially to the loss of important nodes (i.e. nodes in top layers positions or the root) than our completely distributed design. We do correlation only at nodes where the relevant events are actually observable, therefore a loss of some hosts cannot influence the detection capability of the remaining system.

The theoretical scalability values of our system depend on assumptions about the used patterns and the number of send events to different targets during the time interval $\Delta$.

As explained in Section 4.3.3, all messages at the send node have to be copied to each new target of a send event. This results in message traffic which is proportional to the number of send events with different targets during $\Delta$. The average number of send events at a single node during $\Delta$ is indicated as $\omega$. Depending on the used patterns, different amounts of messages have to be copied over send event links. In the optimal case, only one message instance representing the send event itself has to be transmitted. When the attack scenario contains dynamic constraints between events that are separated by one or more send event link, additional messages have to be moved to the target host as well. The situation worsens when a message has to move over several consecutive send links as it gets copied to each target at every step (yielding potential exponential growth of the number of messages). Therefore the depth $\delta$ of a pattern (defined as the maximal number of consecutive send links a message has to traverse) is an important factor to determine scalability of our system. Usually, not all event patterns define or use variables and messages created from those events do not need to be forwarded. $\upsilon$ denotes the fraction of pattern descriptions of an attack scenario that actually do define or use variables and result in messages that might need to be transferred over the net. When e events occur at a single node, only e*$v$ of them might need to be sent over send links.

As each message only contains part of the data of the complete event object we save bandwidth in comparison to systems that have to send the whole event itself (because they do not know which information is important at the higher levels). The ratio between the event object size and the message size (including id and timestamp) is written as $r$.

The explanation (and notation) given above allows to formulate the estimate of the total network traffic as

$$\frac{n*(e*v)*\omega^{\delta}}{r}$$

As each node equally participates in the detection process, the peak traffic is equal to the traffic at a single node which results in

$$\frac{n*(e*v)*\omega^{\delta}}{n*r} = \frac{(e*v)*\omega^{\delta}}{r}$$

Although the formula shows the potential for an exponential explosion of the needed network traffic, the next section will show that $\delta$ which only depends on the used patterns is usually very small for our area of application (e.g. none of our attack scenarios had a $\delta$ greater than 2). It is interesting to notice that the peak traffic does not depend on the total number of nodes in the system which indicates good scalability properties. Additionally, the factors $v$ and r help to keep the total bandwidth utilization reasonably low.

The following table shows the total and peak traffic values for a centralized and a hierarchical solution. We assume that each hierarchy layer is capable of reducing the events it forwards to a higher level node by a constant factor c.

| System | Total Traffic | Peak Traffic |
|---|---|---|
| Centralized | $e*n$ | $e*n$ |
| Hierarchical | $e*n*\sum_{i=1}^{i<l} c^i$ | $e*n*c^{l-1}$ |
| Decentralized | $\frac{n*(e*v)*\omega^{\delta}}{r}$ | $\frac{(e*v)*\omega^{\delta}}{r}$ |

The total and peak traffic values for the centralized solution reflect the fact that all event data is sent to a single location. The traffic in the hierarchical system is created by data that is forwarded by nodes to their higher layer parents. As a fraction (determined by c) of the event data is forwarded over several levels the total traffic consists of the sum of the traffic volumes between each layer. The peak traffic occurs at the root node (i = l-1). Although it is significantly smaller than in the centralized case, it still depends on the number of nodes in the system.

In both the decentralized and the hierarchical system, the total traffic volume increases when compared to a centralized design. Nevertheless, the peak traffic indicates that they scale much better than a centralized one.

### 5.1 Experimental results

Our system is designed to provide a scalable solution for enterprise sized networks. Unfortunately, we do not have the resources to perform scalability tests on such a scale. A simple simulation does not seem reasonable because it would, based on our assumptions, exactly deliver the results we have derived theoretically. Therefore, we decided to perform an actual experiment but had to restrict ourselves to our department's network. We ran processes executing our detection algorithm on the web server, the DNS server, our firewall and 6 additional hosts. These machines are running Linux 2.2.14 and SunOS 5.5.1 on different Pentium II, Athlon and Sparc hosts. The idea is to gather experimental data that can be compared to values that we would expect from the theoretical considerations.

We use `tcpdump` [12] and Snort [9] to collect data from the network and have an EDL (Event Definition Language) definition for common network packets (i.e. TCP, UDP, IP and Ethernet) as well as Snort alerts. These are the basic building blocks for attack scenarios written in ASL (Attack Specification Language). We use 16 different distributed patterns that aim to detect distributed signatures and anomalies (as explained in Section 1) with the following properties.

| Property | Avg | Max | Min |
|---|---|---|---|
| Pattern depth ($\delta$) | 1.19 | 2.00 | 1.00 |
| Fraction of events with variables ($v$) | 0.83 | 1.00 | 0.50 |

The given numbers are based on a week of real data collected in our network during which we processed 16374 events. We used a time interval ($\Delta$) of 24 hours.

| Property | Avg | Max | Min |
|---|---|---|---|
| Events per $\Delta$ | 2340 | 3818 | 1732 |
| Send event targets for single node (during $\Delta$) | 1.62 | 5 | 0 |
| Total traffic (in messages) | 3922 | 7536 | 3159 |
| Peak traffic (in messages) | 1011 | 2722 | 744 |

An analysis of the traffic showed that our system has been exposed to a number of real intrusion attempts where one has even been successful (exploited a hole in our ftp-server). This supports our assumption that we used real traffic including actual attacks for our evaluation. As expected our used patterns did not result in a message explosion and the total number of messages never exceeded twice the number of actual events. When one also considers that only relevant attributes (mainly 2 or 4 bytes) instead of the whole event have to be transmitted the used bandwidth can be compared with a centralized system. The unexpected high peak traffic values resulted from many scans for port 80 that the firewall reported to the web server. In our setup, a high fraction of the messages concentrated on a few machines (web server, DNS server) while regular nodes transmitted fewer messages. However, an increase of nodes in our local network would not raise the load at these machines significantly (as the port scan messages were caused by machines on the Internet anyway) while producing more total traffic inside the network. In such a case (as with large intranets) we expect that the ratio between the messages at these servers and messages at regular nodes decreases.

## 6 Conclusion

We present a distributed pattern detection scheme to relate events that occur at different hosts. This can be used to detect distributed signatures (like telnet chains) and anomalies to support our concept of network awareness. In order to prevent a message explosion the pattern specification language has to be restricted. The consequential decentralized algorithm to find events that satisfy such patterns has been implemented and exhibits good scalability and fault tolerance properties.

## References


[1] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An Architecture for Intrusion Detection using Autonomous Agents. In *14th IEEE Computer Security Applications Conference*, December 1998.

[2] Marc Crosbie and Eugene Spafford. Defending a computer system using autonomous agents. In *Proceedings of the 18th National Information Systems Security Conference*, October 1995.

[3] Jose Duarte de Queiroz, Luiz Fernando Rust da Costa Carmo, and Luci Pirmez. Micael: An autonomous mobile agent system to protect new generation networked applications. In *2nd Annual Workshop on Recent Advances in Intrusion Detection*, September 1999.

[4] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[5] Judith Hochberg, Kathleen Jackson, Cathy Stallins, J. F. McClary, David DuBois, and Josephine Ford. NADIR: An automated system for detecting network intrusion and misuse. *Computer and Security*, 12(3):235–248, May 1993.

[6] Richard A. Kemmerer. NSTAT: A model-based realtime Network Intrusion Detection System. Technical Report TRCS97-18, Computer Science Dep., University of California Santa Barbara, November 1997.

[7] Peter G. Neumann and Phillip A. Porras. Experience with EMERALD to date. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, California, USA, April 1999.

[8] Phillip A. Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 20th NIS Security Conference*, October 1997.

[9] Martin Roesch. Snort - lightweight intrusion detection for networks. In *USENIX Lisa 99*, 1999.

[10] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection system) - Motivation, Architecture and an early Prototype. In *14th National Security Conference*, pages 167–176, 1991.

[11] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS - A Graph Based Intrusion Detection System for large networks. In *Proceedings of the 20th National Information Systems Security Conference*, volume 1, pages 361–370, October 1996.

[12] tcpdump/libpcap. http://www.tcpdump.org.

[13] G. Vigna and R. Kemmerer. NetSTAT: A network-based intrusion detection system. In *Proceedings of the 14th Annual Computer Security Applications Conference*, December 1998.

[14] Gregory B. White, Eric A. Fisch, and Udo W. Pooch. Cooperating Security Managers: A peer-based intrusion detection system. *IEEE Network*, pages 20–23, January/February 1996.