Decentralized Event Correlation for Intrusion Detection

Christopher Krügel, Thomas Toth, and Clemens Kerer

Distributed Systems Group
Technical University Vienna
Argentinierstrasse 8, A-1040 Vienna, Austria
{chris, ttoth, C.Kerer}@infosys.tuwien.ac.at

Abstract. Evidence of attacks against a network and its resources is often scattered over several hosts. Intrusion detection systems (IDS) which attempt to detect such attacks therefore have to collect and correlate information from different sources. We propose a completely decentralized system to solve the task of event correlation and information fusing of data gathered from multiple points within the network.
Our system models an intrusion as a pattern of events that can occur at different hosts and consists of collaborating sensors deployed at various locations throughout the protected network installation.
We present a specification language to define intrusions as distributed patterns and a mechanism to specify their simple building blocks. The peer-to-peer algorithm to detect these pattern and its prototype implementation, called *Quicksand*, are described. Problems and their solutions involved in the management of such a system are discussed.

Keywords: Intrusion Detection, Event Correlation, Network Security

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, sophisticated attackers do not concentrate on a single host alone but try to disguise their actions by distributing them over several machines. Each single action 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*).

As Zamboni [1] pointed out, most existing IDSs perform their data processing centrally, despite their distributed data collection [5, 12]. This causes limitations in their scalability, configurability 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 cause data loss.

To circumvent such shortcomings, hierarchical designs have been introduced. Systems like Emerald [10], GrIDS [13], AAFID [2, 1] or NetSTAT [14] 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 [10, 9] uses a publish/subscriber 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. Top level nodes still limit scalability and their failure can cut off large parts of the IDS (the whole subtree below).

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. Sensors are deployed at every (possible) host of the protected network and at potentially interesting network spots (e.g. router, switch). These sensors collaborate and exchange information in a peer-to-peer fashion without a centralized coordinator.

Only a few systems have already attempted to use a similar approach. The best known is CSM (Cooperating Security Managers) [16], 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 status is unclear.

The following paper describes Quicksand, the prototype implementation of our decentralized intrusion detection approach. First, we define a specification language to define basic events and distributed patterns, which are composed of such events. Then, we discuss Quicksand's system design and management as well as the underlying algorithm to disseminate information between peers. Finally, the performance of our prototype is evaluated and compared to intrusion detection systems that attempt to correlate data in a hierarchical or centralized fashion.

2 System Overview

We define an intrusion as a pattern of basic events that 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 can either stem from a local misuse or an anomaly incident. In addition to our sensors, we plan to integrate a number of existing third-party systems to perform the local detection and import their event data as input for our correlation algorithm.

By relating events from multiple nodes, one can detect a number of attacks that would remain unnoticed by only focusing on local events. One type of attacks can be found by checking for suspicious signatures of network connections between hosts as described in GrIDS [13]. This includes worms spreading in a network as well as telnet chains (a number of consecutive telnet logins by an intruder to hide his tracks). The opening of a connection between two computers looks harmless when monitored locally 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 notice 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 the information about the scan and its source to the webserver. When the webserver later receives abnormal traffic exactly from that source an attack is assumed. Sensors on different hosts can usefully exchange information to enhance their detection rate. The aim of this information flow is that each local node 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.

Quicksand can be used to describe and detect situations where a sequence of events (like port scans and then abnormal packets) occurs on multiple hosts. This allows to modify the reaction of sensors at nodes that get aware of emerging, hostile patterns. Quicksand implements a flexible reaction mechanism that allows to specify response functions for each step of an intrusion pattern. In contrast to traditional systems which usually only react (e.g. generate warnings or harden the firewall) after an incident has been detected, our mechanism allows fine-grain control over potential counter activities while threatening situations emerge.

The detection process finds patterns by sending messages between sensors running on nodes where interesting events occur. Similar to the approach presented in [15], the installation is covered by a *web* of sensors that need to be managed. As novel intrusions are discovered, signature databases of all sensors need to be consistently updated. Quicksand uses a central management station that stores the current configurations of all installed ID probes. We have implemented a mechanism that allows to transfer event and pattern descriptions as well as response functions over a (SSL encrypted) connection to sensors. A protected management channel is utilized to load and unload the building blocks and perform reconfiguration on-the-fly.

In addition, the detection of distributed patterns demands the knowledge of the correct temporal order of events. We have designed two different solutions to this well-known issue in distributed systems that trade additional overhead for improved accuracy.

3 Pattern Specification

The design of our pattern specification language is guided by two conflicting goals. The first goal states that the language 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 amount of data that needs to be exchanged to explode. In the worst case each host has to send all its data to every other node. This conflicts with the second goal, which demands that the amount of transferred data between hosts should be minimal. Therefore we have to impose limitations on the expressiveness of our pattern language.

An event is the smallest entity of a pattern and defined as the occurrence of something that might be part of an intrusion (e.g. an IP packet from a certain source address, the invocation of a certain process). In addition, we introduce the notion of *artificial events*. An artificial event is not related to an actual activity in the environment but is created during a response by Quicksand itself. It is piped back into the sensor's event input queue and can be utilized to satisfy constraints of different (or the same) pattern. Artificial events are a useful mechanism to exchange information between attack scenarios or to model timeouts. The output of a certain attack pattern can be used as input for another pattern to build hierarchical structures or to implement scenarios that count the number of times a certain basic pattern has occurred. A timeout can be implemented by starting a timer in a response that creates an artificial event when it expires.

Basic event objects consist of a list of attribute-value pairs that are filled by the sensor from the actual observed event instance. We define these objects as annotated `C structures` where each attribute is a member that can be a basic `C type` or a reference (pointer) to another structure. This allows to assemble more complex event descriptions as compounds of simpler objects. The annotation is used to specify the attribute that determines the target of send events (which are explained below).

A sensor that scans its input for the occurrence of a certain event needs to be compiled against this event's corresponding `C struct`. Therefore, we cannot add new basic types to a certain sensor while the system is running. This is no real limitation, as the basic events themselves usually do not change frequently. It is more important to be able to update and modify pattern description which represent the actual attacks, an operation supported by Quicksand.

A pattern describes activities on individual hosts as well as interaction 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. A connection (context) between event sequences on different hosts is established by *send events*.

Definition: *A pattern* 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 S_A *at host* A *is linked to a set of events* S_B *at host* B*, iff* S_A *contains a send event to host* S_B*. Any event that refers to a remote host (e.g. a packet sent to a host, the reception of a packet from a host) might be used as send event.*

The detection of a port scan that looks for an open port 80 can be used as a send event when the webserver (the referred host in this example) is known to the host that detects the scan. It is only required that the *target* B of the send event can be determined locally at A (usually by using the event data). The first event of S_B has to be the next event to occur after the send event in S_A. It is required that the send event is the last event in S_A. As mentioned above, basic events are annotated to specify the attribute which is used to derive the target.

Definition: *Pattern* 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 only allow 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 still 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 several hosts end at the root set.

3.1 Attack Specification Language

This section 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 used 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 the send events.

Each event can optionally be defined more precisely by constraints on its 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. The argument of `range` is a pair of values specifying the upper and lower bound of a valid range of values while the argument of `in` is a list that enumerates all possible values.

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 value in constraint definitions. The scope of variables is global and its type is inherited from the defining attribute.

For each event, an optional response function can be specified. This function is invoked whenever the corresponding event description is fulfilled and it can take the values of already bound variables or constants as arguments. A response function can be used to generate alerts for the system administrator or to perform active counter measures against an intruder (e.g. hardening of firewall). As these functions are invoked locally at the node where the corresponding event description has been detected, our system possess no central response component that can be taken out easily. In addition, response functions are used to create artificial event objects that are fed back into the detection process. This allows information exchange between different scenarios or the elegant implementation of timeouts.

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 [ response ]
assignment: '$'variable-id '='
             ( attribute-id | constant )
relation   : attribute-id operator
             ['('] {value ',' }* value [')']
response   : '<' function-id'(' arg-list ')' '>'
value      : constant | '$'variable-id
arg-list   : { arg-id ',' }* arg-id | e
```

The following example shows a classical telnet chain scenario.

```
ATTACK "Telnet Chain" [ Node1, Node2 ]
Node1 {
 send(Node2): tcp_connect [DstPort = 23;]
}
Node2 {
 tcp_connect [DstPort = 23;] <report('telnet-chain');>
}
```

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 as it 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 as specified in the annotation of a `tcp-connect` basic event. Whenever a telnet-chain has been detected at the root node, a report function is invoked.

4 System Design

4.1 Architecture

Our system consists of a set of hosts which are connected by a network. Each node runs a Quicksand sensor which is made up of several local intrusion detection units and an event correlation component.

Local Intrusion Detection Unit The local intrusion detection components are responsible for gathering audit data. They can be implemented as host based systems which collect data from the operating system and user processes or as network based systems that monitor packets via the machine's interface cards. Local ID units can apply misuse or anomaly based methods to extract interesting information (i.e. events) from the local data stream.

For our first prototype, we have built a network based sensor that operates similar to Snort [11] and an anomaly sensor that scans HTTP and DNS traffic [7] for suspicious (i.e. abnormal) content. Events which are relevant for distributed attacks and therefore specified as interesting basic events in ASL are passed to the event correlation unit.

Event Correlation Unit The event correlation unit receives a pre-filtered stream of events from its local intrusion detection unit as well as messages containing relevant data from other nodes and executes the distributed misuse detection algorithm (which is detailed in Section 4.2).

In order to be able to integrate third-party probes, the interface between the local and the event correlation unit has to be designed carefully. Information needs to be passed in both directions as the local sensors have to send interesting events to the correlator and receive configuration and setup information.

We have defined generic events as well as configuration directives based on the intrusion detection message exchange format (idmef) [4] defined by the IETF intrusion detection working group (idwg). When the local system does not support that format, specific plug-in modules are inserted to translate between each system's dialect and the standard format. Instead of creating our own message format we attempt to rely upon existing standards.

The correlation unit is also responsible to react on detected security violations. The ASL definition of an attack contains a description of countermeasures which should be invoked. A possible active response is the hardening of firewalls or the interruption of open network connections. Such activities have to be performed fast and reliable, therefore no central entity is involved. In addition to that, a passive response can be executed by transmitting an alarm message to a dedicated node running a control unit.

Control Unit At least one host needs an installed control unit. This central module is utilized by the system administer to configure the system. Its task includes the processing of attack specifications written in ASL and the configuration of the local intrusion detection and event correlation units that are distributed over the protected network.

Pattern specifications that are written in our Attack Specification Language need to be translated into data structures and functions suitable for our detection process. This is done by an ASL parser that compiles attack patterns into `C` source files which are then compiled into object files. Response functions are specified as `C` functions and compiled into separate object files.

Object files are merged into shared libraries which may be shipped to sensors over a secure connection (SSL). These libraries are automatically installed and can thereupon be utilized by the sensor. The control unit is used to remotely load and unload attack patterns. Whenever a certain pattern is loaded into the correlation engine a dedicated thread is started to execute it. Instead of using a generic interpreter to process pattern specifications, the detection algorithm is directly run as compiled code. To remain flexible and to be able to integrate new patterns on-the-fly (i.e. without changing the code of the event correlation unit), the algorithm's code is divided into a pattern dependent and a pattern independent part. The independent part is implemented as shared code in the correlation unit. Similar to an operating system, this part provides basic services that can be accessed by the threads running the attack scenario dependent code. The pattern related part is stored in the libraries produced by the ASL parser and the response libraries. These libraries are dynamically loaded into the sensors and execute the pattern dependent parts.

The control unit uses a local database to store information about the libraries that have already been installed at each host. This database holds a list of symbols exported by each object file as well as a list of installed object files at each host. That information is used to prevent the installation of libraries that depend on response functions in different object files which have not been installed yet. Currently, we only issue a warning message that enumerates the symbols that cannot be resolved. The problem has to be handled by the system administrator who has to perform the necessary installations manually. In the future, we plan an automatic support to perform this update automatically.

4.2 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 fulfill the temporal and content based constraints of a scenario an alert is raised. We are aware of the fact that a distributed system design might result in tremendous message overhead. This potential danger is addressed at the level of pattern specification (as only tree-shaped patterns are allowed) as well as in the pattern detection algorithm.

Pattern Graph 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. 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 to the first node of the host sequence to which the send event points is inserted. Additionaly, nodes are assigned unique identification numbers (ID).

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 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. Whenever an event matches an event description, a message is generated and the appropriated values copied into it. Then the message is forwarded to the node corresponding to the event description for further processing by the detection algorithm.

Each message is written as a triple <ID, timestamp, list of (attribute,value)>. The ID of the message is set to the identification of the node where the message is sent to (i.e. the node whose event description has matched). 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.

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.

Temporal constraints between events are introduced by send events. The first event in the host sequence at the target of each send event has to occur after the send event itself. In addition, the events of a host sequence have to occur in the same order as they are defined in the ASL description.

An event description that relates an event attribute to a constant value creates a *static constraint* while the use of variables that relate attributes of different events introduce *dynamic constraints*. It is obvious that it cannot always be immediately determined whether an event satisfies its dynamic constraints while all static constraints of a certain event can be resolved immediately. Actual event instances that fulfill the static constraints of a certain event description are transformed into messages which are forwarded to the actual detection process. Its task is to resolve all dynamic and temporal constraints.

Detection Process The idea is that each node of the pattern graph 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 (containing the relevant event instance data) 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 they 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 complete pattern. Whenever events at the root node fulfill the constraints there, the pattern has been detected.

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 making local decisions without a central coordination point. Different host sequences may (and usually do) occur at different hosts. 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 node constraints consist of

1. the set of temporal constraints between the event that is associated with the node and all events that are associated with the node's predecessors in the pattern graph **and**
2. all dynamic constraints that can be resolved at this node. Dynamic constraints (i.e. variable definition at one node and its use at another one) are 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.

Whenever messages fulfill the constraints of a certain node, it is guaranteed that events have occurred which satisfy all constraints (static, dynamic and temporal) of the subtree above this node. An example of a pattern graph with dynamic constraints is shown in Figure 1.

We define the *event pool* for each node as the place that stores messages that can potentially be used to satisfy the node constraints. Message instances that are not needed to satisfy the node constraints of this node (but of nodes closer to the root node) are stored in the *bypass pool*.

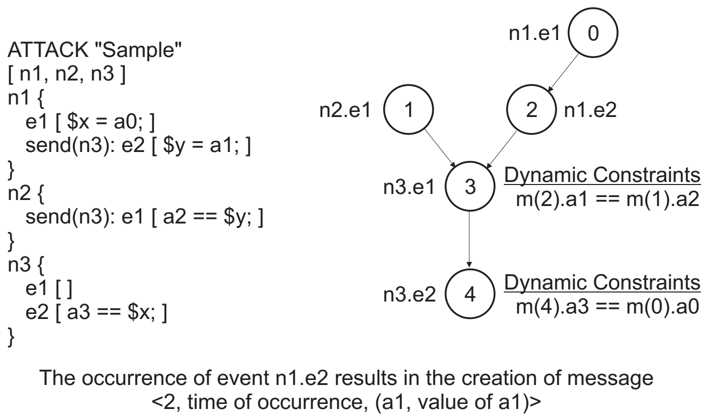

Fig. 1. Pattern Graph and Node Constraints

Detection Algorithm Having determined the node constraints and the event and bypass pools for each node the algorithm to actually move event objects between nodes can be explained. Each arriving message is checked to determine whether it can be used to satisfy node constraints (i.e. belongs to the event pool). If this is the case, the algorithm attempts to find a tuple of messages of *different* type (i.e. all referring to different event description with a different ID) that match *all* the node constraints. The tuple has to include one actual message for each type of the event pool and the new message has to be part of the tuple as well.

Consider a potential tuple for node 3 in Figure 1 with its event pool $\{m1, m2\}$. The event pool consists of messages with IDs 1 and 2 as messages of these types are used in the node's single dynamic constraint. In addition, both messages are used to resolve the temporal constraints between messages of node 3 and its predecessors node 1 and node 2. Therefore the tuple must consist of a message with ID 1 and a message with ID 2. When such a tuple (or tuples, when more than one set of event match the node constraints) can be identified, the detection process has found a number of events 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. As messages in the event pool might be needed later to satisfy the node constraints together with newly arriving instances, the original ones remain in the pool and only copies are forwarded. Whenever a matching tuple is found all messages in the bypass pool are automatically moved over the outgoing link as well, as they are needed at nodes closer to the root. In order to prevent the system from being flooded by duplicate messages each event pool entry is only copied and forwarded to the next node once.

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 and moving of pool entries must be handled differently.

The send node has to keep track which pool entries have already been copied to the destinations of the send events for each different destination.

We use timers to remove elements from the event pools after a certain, configurable timespan because elements cannot be kept infinitely long as memory is a limited resource. 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.

The following example in Figure 2 shows the step-by-step operation of the detection process to detect the distributed pattern which is described by the scenario in Figure 1.

Dotted arrows indicate the copying of event messages to the next neighbor. Associated with each node are two sets enclosed in brackets. The first holds the node's current event pool entries, the second its bypass pool.

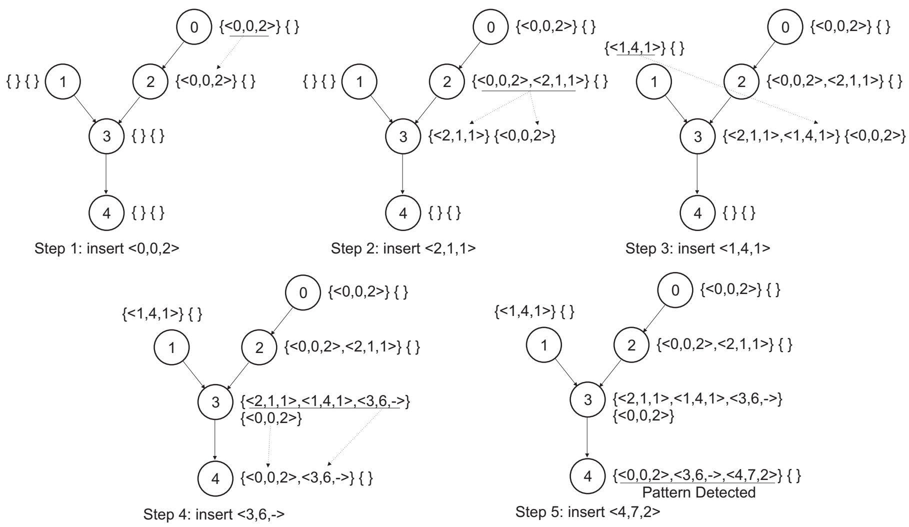

Fig. 2. Pattern Detection

4.3 Time Issues

The use of patterns that can specify temporal relationships between events occurring at different hosts introduces the following challange. The relative order of distributed events needs to be known locally at the node that has to determine whether certain events satisfy the time constraints of a distributed pattern.

In case of centrally collected events, order is usually established by timestamping arriving messages from remote sensors. A system without a central unit can solve this task in two different ways.

One variant uses physical timestamps to mark local events (and the resulting messages) and a synchronization mechanism to keep all physical clocks within acceptable accuracy bounds. This allows the receiver of a message to establish a relative order between remote and local events by comparing the messages' timestamps. Unfortunately, this model is very difficult to implement in large, heterogenous network environments as it requires external means of clock synchronization.

The other variant utilizes logical clocks. Lamport's happened-before relation [8] defines the notion of *causal relationship* to describe sequential events. Adapted to our system, two events are sequential when one happens-before the other in Lamport's sense. Logical clocks are used by having each host marking every sent packet with a logical timestamp. The sender can then refer to this logical timestamp when it detects that a certain packet has contained the send event of a certain pattern. When the receiver later gets the message indicating that a certain received packet has carried a send event it is easy to determine which local events have occurred before and after the reception of that packet (with a known logical time). We have done a simple prototype implementation of such a timeservice for Quicksand which is presented in detail in [6] but it requires a modification of the TCP/IP stack of the underlying operating system and has only been implemented for Linux.

For our tests (that involved different operating systems) a simpler approach has been chosen. We assume a constant delay between the actual send event and the message containing the information that this send event has accurred. At the receiving node, the processing of local messages is simply delayed for that constant timespan to get a correct temporal order between remote and local events. We are aware of the fact that this might introduce incorrect temporal dependencies. When messages arrive too early, our system might report intrusions that have not taken place (false positives) while we would miss intrusions (false negatives) when message take longer than expected. By using a reasonable long delay, false negatives can be practically prevented (especially in local networks). In addition, we have never experienced any false positive as the incidentally occurence of events specified by a certain pattern within this short timespan is very unlikey.

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 an IDS part (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) while the other measures the maximum network traffic at a single node (peak).

We compare Quicksand (representing a completely decentralized system) 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.

5.1 Theoretical Results

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 Δ. The interval Δ also specifies when messages representing events '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 τ. 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 = $\lfloor \log_m((m-1)*n) \rfloor$ 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	$\frac{n-1}{m}$
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 Δ.

As explained in Section 4.2, 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 Δ. The average number of send events at a single node during Δ is indicated as ω. 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 δ 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. υ 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*υ 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

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

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

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

Although the formula shows the potential for an exponential explosion of the needed network traffic, the next section will show that δ is usually very small for our area of application (i.e. less than 2 in almost all real cases). 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 υ 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	$e * n * \sum_{i=1}^{i<l} c^i$	$e * n * c^{l-1}$

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 comes from the 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.

5.2 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. Therefore, we had to restrict ourselves to our department's network and installed Quicksand on our webserver, 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 we would expect from the theoretical considerations.

We use our network based sensor (which is similar to Snort [11]) to collect data from the network and have an EDL (Event Definition Language) definition for basic packets (i.e. TCP, UDP, IP and Ethernet). 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	Average	Maximum	Minimum
Pattern depth (δ)	1.19	2.00	1.00
Events with variables (υ)	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 (Δ) of 24 hours.

Property	Average	Maximum	Minimum
Events per Δ	2340	3818	1732
Send event targets for single node (during Δ)	1.62	5	0
Total traffic (in messages)	3922	7536	3159
Peak traffic (in messages)	1011	2722	744

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. The unexpected high peak traffic values resulted from many scans for port 80 that the firewall reported to the webserver. In our setup, a high fraction of the messages concentrated on a few machines (webserver, 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 Quicksand, an intrusion detection system that implements 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.

The system design and implementation details have been described and a pattern language introduced. This specification language has to be restricted in order to prevent a message explosion in our peer-to-peer architecture. The consequential decentralized algorithm to find events that satisfy such patterns 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. IETF Intrusion Detection Working Group. Intrusion Detection Message Exchange Format. http://www.ietf.org/html.charters/idwg-charter.html.
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. Christopher Krügel and Thomas Toth. An efficient, IP based solution to the 'Logical Timestamp Wrapping' problem. In *6th International Conference on Telecommunications*, 2001.

7. Christopher Krügel, Thomas Toth, and Engin Kirda. Service Specific Anomaly Detection for Intrusion Detection. In *ACM Symposium on Applied Computing (to appear)*, 2002.
8. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, 21(7):558–65, 1978.
9. 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.
10. 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.
11. Martin Roesch. Snort - lightweight intrusion detection for networks. In *USENIX Lisa 99*, 1999.
12. 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, October 1991.
13. 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.
14. G. Vigna and R. Kemmerer. NetSTAT: A network-based intrusion detection system. In *Proceedings of the 14th Annual Computer Security Applications Conference*, December 1998.
15. Giovanni Vigna, Richard A. Kemmerer, and Per Blix. Designing a Web of highly-configurable Intrusion Detection Sensors. In *Recent Advances in Intrusion Detection*. Springer Lecture Notes in Computer Science, 2001.
16. 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.