GrIDS–A GRAPH BASED INTRUSION DETECTION SYSTEM FOR LARGE NETWORKS[*]

S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger,
J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, D. Zerkle

Department of Computer Science,
University of California, Davis,
Davis, CA 95616

email: <*lastname*>*@cs.ucdavis.edu*

Abstract

There is widespread concern that large-scale malicious attacks on computer networks could cause serious disruption to network services. We present the design of GrIDS (Graph-Based Intrusion Detection System). GrIDS collects data about activity on computers and network traffic between them. It aggregates this information into activity graphs which reveal the causal structure of network activity. This allows large-scale automated or co-ordinated attacks to be detected in near real-time. In addition, GrIDS allows network administrators to state policies specifying which users may use particular services of individual hosts or groups of hosts. By analyzing the characteristics of the activity graphs, GrIDS detects and reports violations of the stated policy. GrIDS uses a hierarchical reduction scheme for the graph construction, which allows it to scale to large networks. An early prototype of GrIDS has successfully detected a worm attack.

Keywords: Intrusion detection, networks, information warfare, computer security, graphs.

1 Introduction

The Internet is increasingly important as the vehicle for global electronic commerce. Many organizations also use Internet TCP/IP protocols to build intra-networks (intranets) to share and disseminate internal information. A large scale attack on these networks can cripple important world-wide Internet operations. The Internet Worm of 1988 caused the Internet to be unavailable for about five days [1]. Seven years later, there is no system to detect or analyze such a problem on an Internet-wide scale. The development of a secure infrastructure to defend the Internet and other networks is a major challenge.

In this paper, we present the design of the *Graph-based Intrusion Detection System* (GrIDS). GrIDS' design goal is to analyze network activity on TCP/IP networks with up to several thousand hosts. Its primary function is to detect and analyze large-scale attacks, although it also has the capability of detecting intrusions on individual hosts. GrIDS aggregates network activity of interest into *activity graphs*, which are evaluated and possibly reported to a system security officer (SSO). The hierarchical architecture of GrIDS allows it to scale to large networks.

GrIDS is being designed and built by the authors using formal consensus decision-making and a well-documented software process. We have completed the GrIDS design and have almost finished building a prototype.

This paper is organized as follows. Section 1.1 briefly describes related work on intrusion detection systems and motivates the need for GrIDS. Section 1.2 discusses classes of attacks that we expect to detect. In Section 2.1, the simple GrIDS detection algorithm is described, followed by a more detailed

[*]The work reported here is supported by DARPA under contract DOD/DABT 63-93-C-0045.

discussion in Section 2.3. Section 2.4 has a treatment of the hierarchical approach to scalability and Section 2.5 discusses how the hierarchy is managed. Section 2.6 outlines the policy language. Section 2.7 covers some limitations of GrIDS. Finally, Section 3 presents conclusions and discusses future work.

1.1 Previous Work

The field of intrusion detection began with a report by Anderson [2] followed by Denning's well-known paper that became the foundation for IDES [3]. A recent review of the field is available [4] that gives more detail than we can provide here.

Early systems were designed to detect attacks upon a single host (e.g., IDES (later NIDES) [5, 6] and MIDAS [7]). Although they could collect reports on a single local area network (LAN), these systems did not aggregate information on a wider scale.

Later systems considered the role of networks. The Network Security Monitor (now called Network Intrusion Detector or NID) looked for evidence of intrusions by passively monitoring a single LAN [8]. NADIR [9] and DIDS [10] collect and aggregate audit data from a number of hosts to detect co-ordinated attacks against a set of hosts. However, in all cases, there is no real analysis of patterns of network activity; aggregation is used only to track users that employ several account names as they move around in the network.

NADIR and DIDS use distributed audit trail collection and centralized analysis. Centralized analysis severely limits the scalability of the detection algorithms. In internetworks of multiple administrative domains, different domains may be unwilling to share all activity information with others. Also, sufficient processing and communications resources to analyze activity in very large internetworks is unlikely to be available.

GrIDS moves beyond these limitations by using a hierarchical aggregation scheme in order to scale to larger networks.

1.2 Network Attacks

This section briefly discusses some large-scale attacks that GrIDS aims to detect; it indicates how GrIDS distinguishes malicious activities from normal behavior.

A *sweep* occurs when a single host systematically contacts many others in succession. *Doorknob rattling* is a sweep that checks for vulnerable hosts, (*e.g.* hosts that employ weak or default passwords on user accounts). There are legitimate reasons for sweep activity (*e.g.* polling of network resources such as SNMP, centralized backups, audit sweeps by security administrators). However, legitimate sweeps tend to be highly circumscribed and regular—the source host, services used, hosts contacted, and time of day are known. Thus, they can be differentiated from malicious sweeps.

Coordinated attacks are multi-step exploitations using parallel sessions where the distribution of steps between sessions is designed to obscure the unified nature of the attack or to allow the attack to proceed more quickly (*e.g.* several simultaneous sweep attacks from multiple sources). The combined nature of the distributed attack is only apparent if the attack is traced back to the same source, or if features of the attacks are similar. To detect such coordinated activity, an IDS must correlate sessions across several hosts and possibly across several distributed detectors.

Seely [11] defines a *worm* as "a program that propagates itself across a network using resources on one machine to attack other machines." The best known worm attack is the Internet worm of 1988 which infected thousands of hosts throughout the Internet, rendering many of them unusable. Worms are evidenced by a large amount of traffic forming a tree-like pattern and by similar activity occurring amongst affected hosts. Intrusion detection systems may detect a worm by analyzing the pattern of spread.

2 GrIDS–Graph-Based Intrusion Detection System

We now explain the nature and operation of the GrIDS system. Firstly, we present a simple example to illustrate the main concept. Next, we discuss the architecture and components that make up the distributed system. Then we give a more detailed description of how these components operate to detect intrusions. For a complete account, refer to [12].

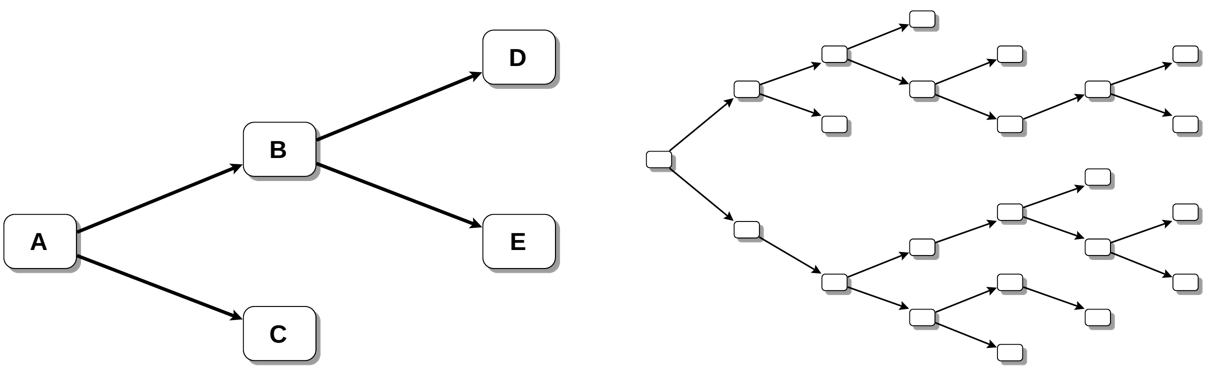

Figure 1: The beginning of a worm graph, and the graph after the worm has spread.

2.1 Detecting a Worm

GrIDS constructs *activity graphs* which represent hosts and activity in a network. Let us take the tracking of a worm as a simple example of building such an activity graph. In Figure 1, the worm begins on host *A*, then initiates connections to hosts *B* and *C* which causes them to be infected. The two connections are reported to GrIDS, which creates a new graph representing this activity and records when it occurred. The two connections are placed in the same graph because they are assumed to be related. In this case, this is because they overlap in the network topology and occur closely together in time.

If enough time passes without further activity from hosts *A*, *B*, or *C*, then the graph will be discarded. However, if the worm spreads quickly to hosts *D* and *E*, as in the figure, then this new activity is added to the graph and the graph's time stamp is updated. Eventually, the worm's spread is represented as a larger graph, as shown on the right of Figure 1.

Thus when a worm infects a network protected by GrIDS, the network activity associated with its propagation causes GrIDS to build a tree-like graph. A detection heuristic can recognize this tree-like graph as a potential worm. This evaluation might count the number of nodes and branches in the graph. Recognition (detection) occurs when the counts exceed a user-specified threshold, thus causing GrIDS to report a worm.

In the previous example, all connections were incorporated into the graph regardless of connection type. GrIDS can use other information to relate network activities, such as destination port numbers, or the type of operating systems. In fact, arbitrary information can be utilized since GrIDS can import user-supplied correlation functions into its graph building algorithm.

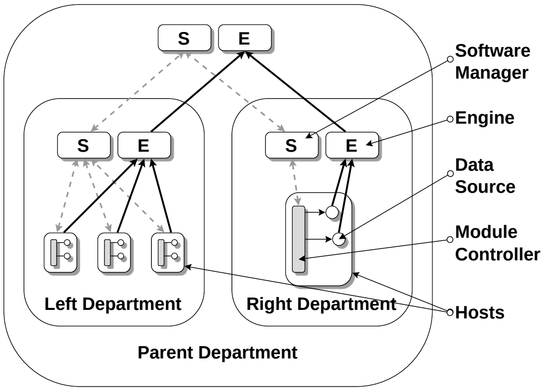

Figure 2: Overall architecture of the system.

Similarly, sweeps and other patterns of misuse produce graphs of a characteristic shape, and GrIDS may be programmed to detect and report them.

To verify our design concept, a basic implementation of this algorithm (which we christened *Early Bird*) was built. While it would be premature to quantitatively evaluate this version, the code was tested for several weeks on our LAN with tcp-wrapper [13] data as input. It was not difficult to tune the software to detect a worm or sweep attack within seconds but produce only one or two false alarms per day from normal user traffic.

2.2 Architecture

Figure 2 depicts a simple hierarchy with three departments: *Left* has three hosts, *Right* has one host, and *Parent* contains *Left* and *Right*.

All GrIDS software is in the form of modules with a standardized interface. The modules are started, stopped, and controlled by a module controller process located on each host.

Each department has two special modules: the software manager and the graph engine. The software manager is responsible for managing the state of the hierarchy and the distributed modules. The hierarchy is re-arranged dynamically by drag-and-drop in a user interface, and starting and stopping particular modules is similarly automated.

GrIDS data sources are modules that monitor activity on hosts and networks and send reports of detected activity to the engine. The activity is reported in the form of a node or an edge for possible inclusion in an activity graph.

Data sources that are part of GrIDS include network sniffers and point IDSs (intrusion detection systems that work on a single host or LAN). However, GrIDS provides an extensible mechanism such that other security tools can be incorporated as data sources without significant change to the tool or GrIDS.

The graph engine takes input from data source modules. The engine builds graphs, and then passes summaries of those graphs up to the engine for its parent department. The parent engine, in turn, builds graphs which have a coarser resolution.

In addition to the components shown, there are user interface modules for allowing human interaction with the system, management functions, and display of alerts. There is also a central organizational hierarchy server which has a global view of the topology of the hierarchy, and is responsible for ensuring that changes to the hierarchy happen in a consistent manner.

2.3 Graph Building

This section discusses the GrIDS *engine*, which collects reports from the data sources and builds them into graphs.

Graphs consist of nodes and directed edges. A single graph represents a causally connected set of events on the network. Nodes represent hosts or departments, and edges represent network traffic between them. Nodes and edges are annotated with attributes that hold supplementary information. In addition, a graph has *global attributes* which maintain state information about the graph as a whole.

Because GrIDS searches for numerous types of network abuse, different kinds of graph are needed. Graphs are constructed in a flexible way; users write *rule sets* which specify how graphs are built from reports. A single graph containing all network activity is too awkward to analyze effectively, so GrIDS allows multiple rule sets. For each rule set it maintains a *graph space* which contains a number of connected graphs. A rule set is an executable specification of one kind of graph; it determines whether an incoming report will be incorporated into existing graphs, and what the results will be. It also specifies when the engine will consider a graph as suspicious and what actions to take if it is. Rule sets operate independently from one another.

Each new report is presented to each rule set in the form of a partial graph. If the report satisfies the rule set's *preconditions*, the engine considers adding the report to the graphs in that rule set's graph space.

A rule set specifies *combining rules* (for nodes and for edges), to determine if an incoming graph should be combined with an existing overlapping graph, and how that should occur. Disjoint graphs cannot be combined. If a combining condition is satisfied on at least one node or edge, then the incoming graph is combined with that existing graph, and the graph's attributes are recomputed. Finally, if no graph combining occurs, but the incoming report did pass the preconditions, then it forms a new graph in the graph space.

2.3.1 An Example Rule Set

Rule sets serve several purposes: to decide if two graphs should combine, to compute the attributes of the combined graph, and to decide what actions to take, if any. Computing the edges and nodes in the combined graph is a straightforward matter which the engine does automatically. However, since it does not know the semantics of user-defined attributes, the rule set must specify how to combine them

A rule set consists of several sections:

- A name
- Initializations
- Preconditions
- Graph combining rules
- Assessment and actions

The following example rule set detects worms by aggregating adjacent connections into the same graph if they occur close together in time. It also includes any node reports which have an *alert* attribute, if they fall in the appropriate time frame. Some portions of the rule set which give low level detail have been omitted for clarity.

Throughout the following rules, `new` refers to attributes appearing on the incoming report, `cur` refers to attributes appearing on an existing graph for this rule set, and `res` refers to attributes being computed for the resulting graph. The `{ ... }` syntax denotes a set constructor.

```
ruleset worm_detector;

timeout 30;

report global rules {
  res.global.alerts = {};
  res.global.time = 0;
}

node precondition defined(new.node.time)
               && defined(new.node.alert);
edge precondition defined(new.edge.time);
```

The `report global rules` initialize the graph space.

Node and edge preconditions filter the reports that are not pertinent to the kind of abuse that this rule set is trying to detect. For each node and edge in the incoming graph, the appropriate kind of precondition is evaluated.

This node precondition requires an incoming node to have a *time* attribute and an *alert* attribute. Similarly, incoming edges (in reports) are accepted if they possess a *time* attribute.

Node rules may access both sets of global attributes and the attributes on the local node being considered. The sample adds any *alert* attributes on the current node to the global *alerts* attribute, initializes the local *alerts* attribute and the *time* attribute. Similarly, the edge rules combine alerts.

```
report node rules {
  res.global.alerts =
    {res.global.alerts, new.node.alert};
  res.node.alerts = {new.node.alert};
  res.global.time =
    max({res.global.time, new.node.time});
  res.node.time = new.node.time;
}

report edge rules {
  res.global.alerts =
      {res.global.alerts, new.edge.alert};
  res.edge.alerts =
      {res.edge.alerts, new.edge.alert};
  res.global.time =
      max({res.global.time, new.edge.time,
         new.source.time, new.dest.time});
  res.edge.time = max({new.edge.time,
         new.source.time, new.dest.time});
}
```

The next three sections of the rule set specify whether to coalesce two graphs, and compute attributes on the coalesced graph. (Disjoint sub-graph global attributes are re-computed on those nodes and edges within the intersection of two graphs.)

First we specify how the global attributes of two disjoint sub-graphs are combined by the engine. This initial combination can be modified by subsequent local rules. The `combine global` section updates the global *alerts* attribute for a graph to be the union of the existing *alerts* attributes of for the graphs under combination:

```
combine global rules {
  res.global.alerts =
       {new.global.alerts,
        cur.global.alerts};
}
```

The attribute *combine* determines whether the graphs should be combined. If the *combine* attribute evaluates to true on *any* overlapping node or edge in the sub-graphs, then the graphs are coalesced. In the example below, the sub-graphs are combined if at least one of the shared nodes has a non-empty *alerts* attributes, and if the nodes' time attributes are within thirty seconds.

If the sub-graphs are combined, the remaining node rules specify how attributes at nodes combine. In this case, the *alerts* attribute at a node in the final graph is the union of the *alerts* attributes for the constituent nodes, and the *time* attribute is the latest of the *time* attributes on the constituent nodes.

The edge rules are similar.

```
combine node rules {
  res.node.combine =
    !empty({new.node.alerts,
            cur.node.alerts})
    && abs(cur.node.time -
           new.node.time) < 30;
  res.node.alerts = {cur.node.alerts,
                     new.node.alerts};
  res.node.time =
    max({cur.node.time, new.node.time});
}

combine edge rules {
  res.edge.combine =
     abs (cur.edge.time - new.edge.time)
         < 30;
  res.edge.alerts =
     {cur.edge.alerts, new.edge.alerts};
  res.edge.time =
     max({cur.edge.time, new.edge.time});
}
```

Finally, the assessment rules evaluate the resulting graph and take appropriate actions. The actions on the right hand side are built-in functions, user defined functions, or updates to global attributes.

```
assessments {
  (!empty(res.global.alerts))  ||
   (res.global.nnodes >= 8)  ||
   (res.global.nedges >= 13)  ==>
              alert(), report-graph();
(3 < res.global.nnodes < 8) ||
(5 < res.global.nedges < 13) ==>
              report-graph();
}
```

Note that several attributes referred to above were neither declared nor computed by the earlier rules. These are *automatically computed* attributes; their values can be read by the rules, but not written:

- `global.ruleset` – the name of the rule set.
- `global.nnodes` – the number of nodes in a graph.
- `global.nedges` – the number of edges in a graph.
- `node.name` – the name of this particular node.
- `edge.source` – a list of the domains associated with the source of this edge that are within this engine's domain, starting with the domain for the source within this engine's domain and ending with the host.
- `edge.dest` – same as `source` except pertaining to the destination side of the edge.

2.4 Aggregation

GrIDS models an organization as a hierarchy of departments and hosts. Each department in the hierarchy has an engine of its own, which builds and evaluates graphs of activity within that department. However, activity which crosses departmental boundaries is passed up to higher levels in the hierarchy for further analysis.

As graphs propagate upward, entire departments may be represented as a single vertex, rather than a vertex per host, in a *reduced graph*. For example, the graph in Figure 3 represents an activity that involves hosts of three departments. Each department sees only the activity within its boundaries; these do not appear suspicious. The whole graph is not visible from any of the lower departments.

The higher level department does not have access to the full graph on the left either. At this level in the departmental hierarchy, the reduced graph (shown on the right) is seen. Because some information has been lost in the reducing of the subgraphs, this graph's topology is not suspicious either. However, attributes of the individual subgraphs are passed up forming attributes on the nodes in the aggregated graph. This allows the higher level module to draw stronger conclusions about the graph.

For example, each sub-department can pass up the size of the subgraph it sees, the branching factor of the graph, and the entrance and exit points of the graph into and out of this department. Thus, GrIDS can deduce that the total graph seen at the higher level has ten hosts in it. Similarly, an approximation of the branching factor and the depth of the graph can be computed.

Intractably large graphs never appear at any level. At lower levels, only sections of the graph are seen. At higher levels, only summary information about lower graphs is seen. Using this approach of aggregating graphs, GrIDS infers and reduces the data that must be analyzed at the higher levels of the hierarchy. It is this that makes GrIDS a scalable design.

The hierarchy which handles aggregation of graphs is also used to manage rule sets. A rule set is inherited by all the descendants of that node.

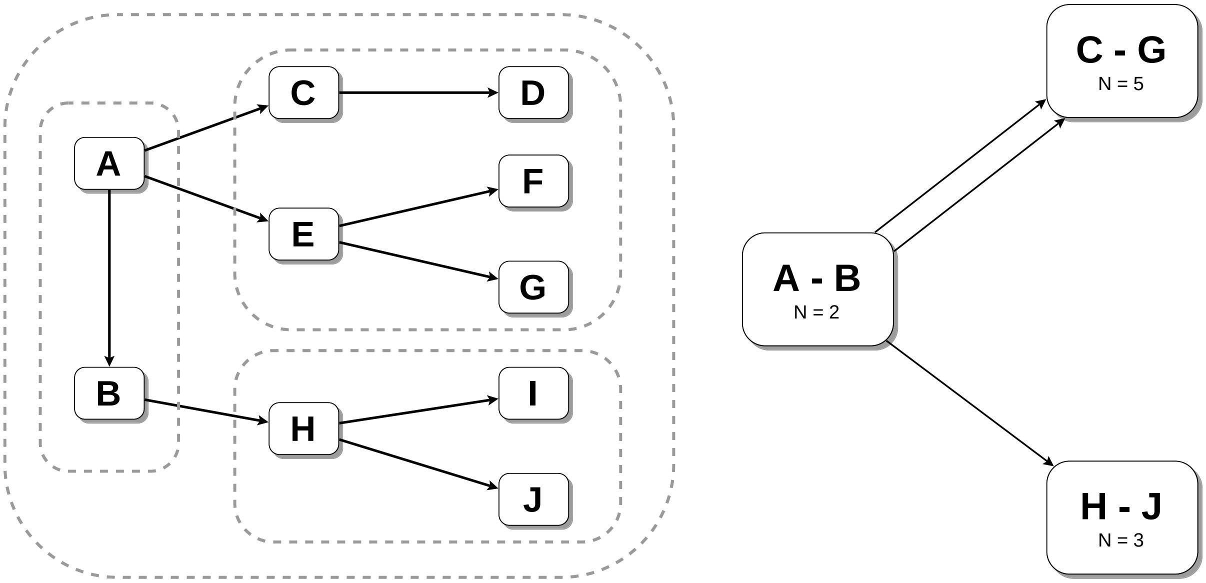

Figure 3: A graph amongst several departments (left), and the corresponding reduced graph. The dashed lines are departmental boundaries.

2.5 Managing the Hierarchy

Since organizations change, the hierarchical structure of departments and hosts must permit changes, but only by authorized users. This section describes how the hierarchy can be changed in a consistent manner.

An access control system controls who is able to view and manage the hierarchy. Each node (host or dept) in the hierarchy maintains an access control list (ACL) that specifies who may manage that node or any node in the subtree rooted there.

Users manage the hierarchy through *views* of subsets of the hierarchy which show the topology of the departments and hosts involved, and making *transactions* which change the hierarchy. Transactions include moving a department, adding a new host, changing the location of the graph engine for a department, etc. The challenge is to ensure that these transactions occur atomically and that the hierarchy is always left in a consistent state afterwards.

Several modules are involved in implementing the hierarchy. Each department has a software manager which is responsible for monitoring the hosts and modules in its department, tracking which hosts are currently functioning, and maintaining department-wide states such as the access control list. In addition, each host has a module controller responsible for the GrIDS software running on that particular host. There are multiple user interfaces which have various views of parts of the hierarchy. All of these must be kept consistent.

Software managers and module controllers only know the *local* topology, i.e. their immediate parents and children.

A centralized *organizational hierarchy server* (OHS) maintains a complete global picture of the entire hierarchy. User interfaces maintain copies of as much of the hierarchy topology as their users presently wish (and are authorized) to manage.

Local knowledge simplifies efficient implementation of atomicity and consistency; locking, *etc.* can be centralized at the OHS. The use of a centralized system has some potential to limit scalability. Clearly, a single OHS will not work for the entire Internet. However, the OHS is only involved in *changes* to the topology of GrIDS, not in its routine operation. Hence, this limitation is not pressing.

We now outline how a transaction on the scenario depicted in Figure 4 would be carried out. Full details can be obtained from [12].

In the following, we use the notation S_C to refer to the software manager at C, M_C to refer to the module controller on the machine on which S_C is running, and similarly for the other departments. The organizational hierarchy server is O, and the interface is I.

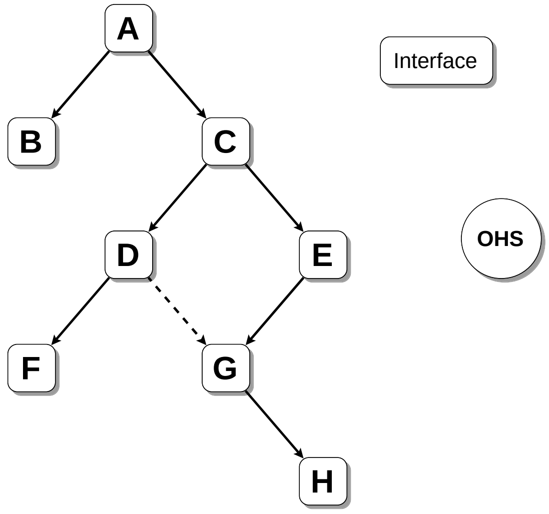

Figure 4: An example hierarchy. Department G is about to be moved from under department E to under department D. An interface and the organizational hierarchy server are also shown.

When a user starts up an instance of the user interface, she is prompted for a user identifier, a password, and a department in the hierarchy which she wishes to administer (in this case C).

I requests a copy of the hierarchy below C from O. O contacts S_C to verify that the user is authorized to access C. Then O replies to I with a copy of the hierarchy rooted at C. O maintains a list of interfaces that have copies of the hierarchy. I displays the subtree on the user's screen. The copy is marked with a *version number* which is used in subsequent transactions to detect stale copies.

Suppose that, having inspected the hierarchy, the user decides to move department G (and by implication, its descendants) under D instead of E (illustrated by the dashed line in figure 4. The first step is to send a message to O. This message describes the planned action and supplies the hierarchy version number on which the planned action was based. O first determines if I's planned action is consistent with existing locks in the hierarchy and based on an up-to-date view. If so, it locks the appropriate part of the hierarchy and contacts S_D and S_E to verify that the user has the necessary permissions. If she does not, the lock is released. Assuming that the action appears feasible, O gives permission for I to go ahead.

Now I contacts S_E (the software manager for the parent of department G) and informs it that G is to be moved. S_E sends messages to S_G warning of the impending change. Then S_E sends messages to M_G to alter the destination of S_G and A_G's messages and the location of M_G's parent. Since modules only have local knowledge, only S_G, A_G and M_G need to be updated. If these transactions have succeeded, S_E updates its own data structures and acknowledges completion to I.

Next I informs S_D of the move. S_D then informs S_G of the new information it needs to be a child of D (*e.g.*, the access control list inherited from S_D). Upon completion, I reports back to O. O may then remove the locks on the hierarchy. Finally, O advises the interfaces that have invalid copies of the hierarchy. The OHS makes a best effort to inform the interfaces but does not block if interfaces are busy or no longer exist as this could prevent subsequent OHS transactions from proceeding. If any interface is not updated, the use of version numbers ensures that any transactions using stale hierarchy data are detected.

2.6 Policy

GrIDS includes a policy language to express acceptable and unacceptable behavior on the network. A network is a collection of users, hosts and departments. These entities communicate via pair-wise network connections which are labelled with the application protocol employed (e.g., TELNET, NFS, HTTP). Thus, a connection originates from a user, host or department and terminates in another user, host and/or department.

Policies are compiled into rule sets which build graphs and evaluate them for policy violations. This saves the user from having to write rule sets manually. In general, rule sets are more complicated to specify correctly than are policies. The present version of GrIDS only allows for policies stated with respect to a single graph edge (network connection).

The authorization model employed is similar to an access control model. The user specifies whether a connection is permitted or prohibited. Thus a rule regarding a certain type of connection consists of a tuple (*action*, *time*, *source*, *destination*, *protocol*, *stage*, *status*, ...) where *action* is allow or deny, *time* qualifies the rule with respect to a clock or time interval, *source*, and *destination* describe the connection endpoints and *protocol* describes the connection type. A connection progresses through several stages (e.g. start, login, authentication, stop, etc.), and the *stage* and *status* attribute further characterizes the connection.

As an example, consider the policy

> No student in the Computer Science Department is to read or write to the grade server hosted in Administration; faculty are permitted to submit grades and to read grades; teaching assistants are permitted to read grades; the department chair is permitted to change grades.

To check this policy with GrIDS, the policy compiler generates rule sets for three domains: Computer Science, Administration, and the department that constitutes the least upper bound of these two domains. The policy writer merely specifies the tuple that identifies which connections between these domains are allowed or disallowed.

Even though this policy mechanism is very simple, it allows considerably more flexibility than is possible with the main tool currently used for expressing network access policies: firewalls.

2.7 Limitations

GrIDS tackles some of the hard issues which need to be faced for an intrusion detection system to operate on a large network. A lot of our effort has gone into making the aggregation mechanism scalable, and allowing the system to be dynamically configurable so that it is still manageable when deployed on a large scale.

The current version of GrIDS is intended as a proof of concept for our approach to scalability and aggregation; as such, it has limitations. Before GrIDS can be considered for deployment in production environments, additional safeguards must be taken to ensure the integrity of communications between GrIDS modules, and to prevent an attacker from replacing parts of GrIDS with malicious software of her own. The prototype will not be resistant to denial of service attacks, disruptions of the network time protocol, or faults in the networks or computers on which it runs.

GrIDS is designed to detect large-scale attacks or violations of an explicit policy. A widespread attack that progresses slowly might not be diagnosed by our aggregation mechanism. However, suspicious activity associated with the attack could be detected since point IDSs can be installed on GrIDS to detect intrusions that involve only one or a few sites.

3 Conclusions

We have presented the design of GrIDS. We have argued that GrIDS is helpful in detecting automated and spreading attacks on networks. GrIDS presents network activities to humans as highly comprehensible graphs. In addition, the GrIDS policy mechanisms allows organizations much greater control over the use of their networks than is possible, for example, with firewalls alone. GrIDS does this in a manner that is scalable and requires modest resources. GrIDS itself is manageable.

There is a great deal of further work to be done on GrIDS. The initial design is complete, and a prototype implementation is almost finished. We will proceed to evaluate the prototype and publish those results. Beyond that, robustness against random faults and attacks on GrIDS itself is the next priority. We also plan to further refine the policy language implemented by GrIDS.

Many important networks are vulnerable to widespread attack. We hope that GrIDS is a helpful step toward defending against such attacks.

Acknowledgements

We are grateful to DARPA for funding this research and to our technical monitor there, Teresa Lunt, for helpful discussion of this design.

References

[1] M. Eichin and J. Rochis. With microscope and tweezers: An analysis of the Internet worm of November 1988. *IEEE Symposium on Research in Security and Privacy*, 1989.

[2] James P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, 1980.

[3] Dorothy E. Denning. An intrusion detection model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 118–131, 1986.

[4] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, 8:26–41, May-June 1994.

[5] T. Lunt *et al.* IDES: The enhanced prototype. Technical report, SRI International, Computer Science Lab, October 1988.

[6] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion detection expert system (NIDES). Technical Report SRI-CSL-95-07, SRI International, Computer Science Lab, May 1995.

[7] M. Sebring *et al.* Expert systems in intrusion detection: A case study. *Proceedings of the 11th National Computer Security Conference*, 1988.

[8] L. T. Heberlein *et al.* A network security monitor. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1990.

[9] K. Jackson, D. DuBois, and C. Stallings. An expert system application for network intrusion detection. *Proceedings of the 14th Department of Energy Computer Security Group Conference*, 1991.

[10] S. Snapp *et al.* DIDS – motivation, architecture and an early prototype. *Proceedings of COMPCON*, 1991.

[11] D. Seely. A tour of the worm. *IEEE Trans. on Soft. Eng.*, November 1991.

[12] Computer Security Research Group. *The Design of GrIDS: A Graph-Based Intrusion Detection System.* Technical report, UC Davis Department of Computer Science, Davis, California, in preparation.

[13] Steven McCanne, B. Jacobsen, and Craig Leres. Tcpdump. ftp://ftp.ee.lbl.gov.