A Method to Detect Intrusive Activity in a Networked Environment[1]

L.T. Heberlein, K.N. Levitt, B. Mukherjee

Computer Security Laboratory
Division of Computer Science
University of California
Davis, Ca. 95616

ABSTRACT

Intrusive activity is occurring on our computer systems, and the need for intrusion detection has been demonstrated. This paper discusses some of the benefits and drawbacks of trying to detect the intrusive activity by analyzing network traffic. A general solution, based on detecting and analyzing abstract objects, is formulated. Finally, results from applying the solution are presented.

1. Introduction

Computers are the targets of attacks [3]. Reports appear in the media almost weekly about outsiders breaking into computers, employees misusing computers, and rogue viruses and worms penetrating computer systems. Incidents such as the internet worm of 1988 [3], the Wank worm [3], and the Netherland hackers have gained international recognition, and they serve to emphasize the vulnerability of computer systems around the world.

These reported incidents are cases of intrusive activity in our computer systems. Intrusive activity can be defined as any attempt which, if successful, will result in one of the following:

- disclosure of information against the wishes of the owner of the information
- modification of information against the wishes of the owner of the information
- denial of the use of services by legitimate users of the system
- use of resources against the wishes of the system's owner (e.g. disk or CPU)

The first three bulleted items are discussed in [4]. The last bulleted item, the stealing of resources, covers actual observed activity which did not fit easily into the three previous categories. For example, using our network security monitor (NSM) [8], we have observed an intruder use a system to crack password files. The intruder was not interested in either looking at existing information on the system, modifying information on the system, or denying resources to legitimate user. The intruder simply used the CPU, when it was idle, to crack passwords.

Authentication and access control mechanisms are designed to guard against intrusive activity; however, these mechanisms have not been wholly successful. Failure of these mechanisms is due in part to the ease by which passwords can be compromised, failure by system administrators and users to properly use the access control mechanisms, poor operating system designs, and flawed operating system implementations (i.e., bugs).

The failures of authentication and access control mechanisms are compounded by the decentralization of computer systems and the increased access to a computer system by computer networks. The decentralization of computer systems is the movement away from a single mainframe computer to multiple workstations and personal computers. The movement is fueled by the increasing power and decreasing costs of workstations and personal computers. The result of decentralization is a type of computer system which is administered by people, usually the user community, with little or no formal training in system administration or computer security. This in turn results in a greater chance for poorly configured authentication and access control mechanisms.

Connecting a computer to a network also increases the chances of intrusive activity occurring on that computer since this process increases the number of people who can potentially access it. Connecting a computer to a network provides a path to that computer for every user with access to the network. If the

[1] This work is supported in part by Lawrence Livermore National Laboratory

network is part of the internet, essentially everyone with access to a telephone has a path to that computer.

With the realization that current authentication and access control mechanisms have not provided adequate security against intrusive behavior, institutions which use computers and computer networks have become interested in detecting the intrusive activity which is occurring. If an intrusion can be detected, an institution can at least know from where intrusive activity is coming, how the activity is being perpetrated (and therefore, hopefully how to stop it), and what data have been compromised.

In the summer of 1988, University of California at Davis and Lawrence Livermore National Laboratory began an effort to detect intrusive activity on a network of heterogeneous computer systems. A brief overview of this effort is presented in section two. Sections three and four present the mechanisms by which our monitor detects intrusive activity. And section five presents some of the results of our efforts as well as directions for future research.

2. Network Monitor

Intrusion detection systems examine available sources of information about the various operations in a computing system to determine if intrusive activity is occurring. The main source of information for most intrusion detection system is the audit trails generated by the operating system. Although the audit-trail-based analysis has provided a measure of success, a number of limitations exist with this method. First, audit trails traditionally do not provide much of the information necessary to perform security analysis. This is due in part to the historical purpose of audit trail collection - the billing of customers. Second, audit trails tend to be system specific. Each operating system provides a different set of information in a different format. An intrusion detection system designed to work on the Multics operating system's audit trails would need a great deal of restructuring to operate on another operating system's audit trails. Third, the collection of audit trails is expensive in terms of CPU usage and storage utilization. Many organizations, even those working in the field of computer security, turn off auditing on their machines to avoid the resource penalty. Fourth, the audit trails themselves can be the target of an intruder. Intruders have been known to turn off auditing on machines in order to hide their tracks. Fifth, and last, the delay in the actual recording and analysis of the audit information can allow an intruder to do damage and exit the machine before the intrusion is noticed [17]. So, although there exists a strong desire for immediate notification of intrusive activity, audit mechanisms can introduce a delay factor.

By exploiting the broadcast property of a local area network (LAN) and network protocol standards, the analysis of network traffic can solve a number of the drawbacks associated with audit-trail-based analysis. First, network standards exist by which a variety of hosts can communicate. An intrusion detection system based on network traffic can therefore simultaneously monitor a number of hosts consisting of different hardware and operating system platforms. Second, the collection of network traffic does not create any performance degradation on the machines being monitored, so network monitoring is more attractive to a user community which places importance in the performance and responsiveness of their machines. Third, since a network monitor can be logically isolated from the computing environment, its analysis cannot be compromised by an intruder. Typically, the intruder has absolutely no way of knowing that the network is being monitored. And fourth, since a network monitor draws its information directly from the network, no delay occurs from the instant an intrusion occurs until the instant the evidence is available. Instead, intrusive activity can be observed as it occurs.

The original work on this type of network monitoring was based on simple traffic analysis: modelling the flow of data among the different machines [9,10]. In [9,10], network traffic is modelled with a concept called a data path. A data path is a method by which one machine can communicate with a second machine. A data path is defined by the three-tuple <src_host, dst_host, network_service>. If the traffic flow shifted (e.g., a new data path is observed) at any point, this information would be reported as a possible intrusion. For example, a particular host initiating a login to a host to which it has never logged into before would be considered suspicious. This work was based on Denning's hypothesis that intrusive activity would manifest itself as anomalous behavior [2].

Although this method showed early promise, a major drawback quickly became apparent: the information available from simple network packet analysis was at a level much too low to detect subtle intrusive activity. For example, an intrusion over a commonly used data path would not be detected. Unfortunately, this is often the case when the intrusion is being perpetrated by an insider.

To provide for a more effective intrusion detection system, our monitor needed the capability to detect and analyze higher-level objects which are not directly observed (i.e., individual network connections and hosts). Also, to perform the analysis information about each object-attributes for the object-needed to be known.

The logical architecture of our system is shown in figure 1, and the components which provide for the additional complexity of analysis, viz. object detector and object analyzer, are shown in the dashed box. The functionality provided by these components have greatly enhanced our efforts to detect intrusive activity. Results from actual use of our monitor can be found in [7,8]. We have attempted to both generalize and formalize the methods by which our monitor detects and analyzes objects, and this work is presented in sections three and four.

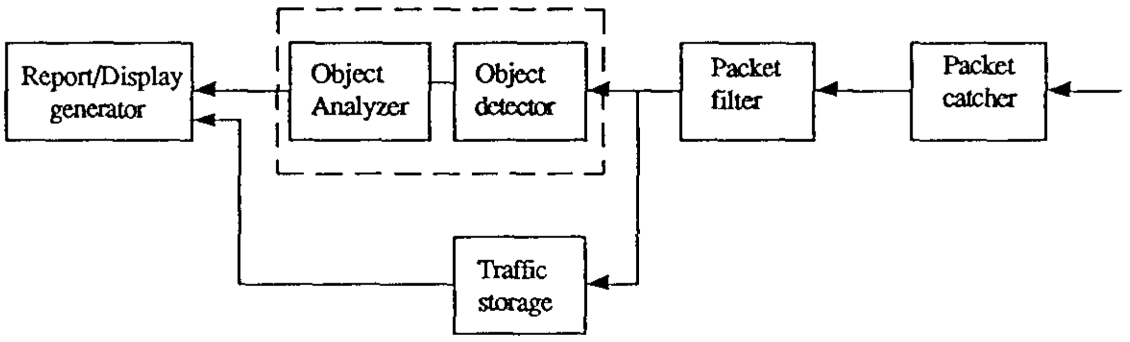


Figure 1

3. System Description Language

The problem of detecting intrusive activity in a heterogeneous network of computers through the observation of network packets can be generalized to the detection of a behavior in a complex system (e.g., networked system) from the analysis of low level information (e.g., network packets). The complex system is composed of a variety of components (i.e. hosts, connection, and packets) each of which in turn may be composed of other components, but only the simplest of components, the lowest levels of information, are directly visible to a monitor. Unfortunately, to detect the behavior of interest (i.e. intrusive activity), the complex components which are not directly observed, as well as the low level components, must be examined for the manifestation of the behavior.

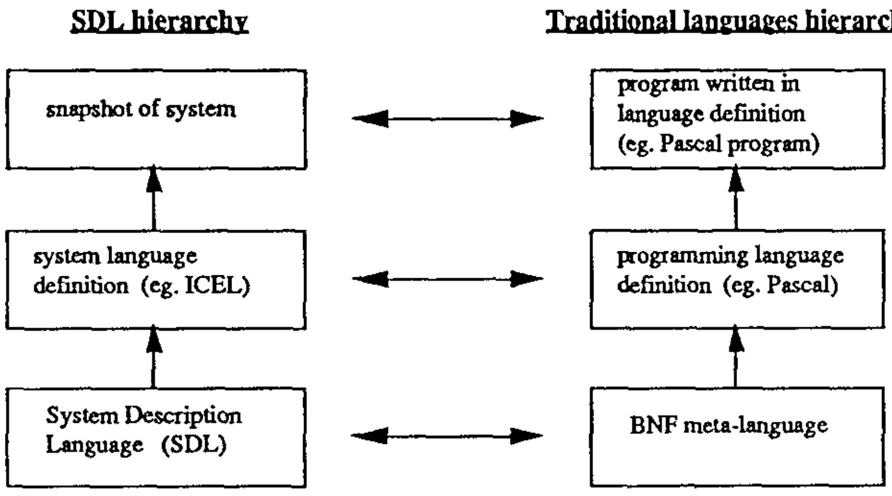

SDL hierarchy
Traditional languages hierarchy
snapshot of system
program written in language definition (eg. Pascal program)
system language definition (eg. ICEL)
programming language definition (eg. Pascal)
System Description Language (SDL)
BNF meta-language

Figure 2

To provide for a formal mechanism to infer the complex components of a system, we have defined a meta-language, called the system description language (SDL), to describe the relationships among components of a system. The description of a system with this language is called its system language definition. As the low level information is observed, the system language definition is used to infer the existence of the complex objects and the relationships between them. A snapshot of all the low level objects and the inferred complex objects and their relationships to one another represent a model of the actual system at a particular time instant. It is this model which will be examined for the manifestation of the behavior of interest.

The SDL, the system language definition, and the snapshot of an actual system have a direct resemblance to the definition of a traditional programming language. The SDL provides a functionality similar to that of the BNF meta-language. The system language definition is similar to a traditional program language definition (e.g., Pascal). And the snapshot of a system is similar to a program defined by a traditional programming language. This relationship is shown in figure 2.

The system description language is the focus of this section. Section 3.1 introduces the issues which must be addressed by the system description language. Section 3.2 presents a review of attribute grammars, the ancestor of the system description language. And section 3.3 discusses the actual system description language.

3.1 Issues to be Addresses by the SDL

To design a meta-language which can be used to describe and model complex systems from the observation of low level information, a number of issues must be addressed. First, how are the low level, simple components of the system detected, and how are the attributes of each low level object determined? We have chosen to not address this issue in this paper, and it is not part of the language definition. The low level components are detected, and their associated attributes are determined by a preprocessor. This is not unlike the design of conventional programming languages which assume the presence of a lexical analyzer to detect tokens, and, if necessary, determine their attributes.

The second issue is the identification and representation of components of the system which are not observed directly. In fact, a complex object which does not have a real world counterpart may be desired. For example, our model for the computer network environment includes an object called a "service-set." The service-set object does not exist in the actual system, but its presence is helpful in analyzing other components such as network connections. The system description language must provide a mechanism for inferring the existence of these unobserved, perhaps nonexistent, objects. Furthermore, the language must provide mechanisms to determine enough information about these abstract objects so they can be analyzed for the behavior of interest.

The third issue is concerned with the transitory nature of many of the objects in a system. Systems such as a heterogeneous network have a number of components which exist for a time, and then disappear. For example, network connections are created and destroyed continuously. The system description language must be able to handle the creation and destruction of components, and the system description language must provide information to determine when a component should be created or destroyed. Thus the model of an actual system, as determined by a system language definition, can change over time.

In summary, the system description language assumes that the low level, simple components and their attributes are provided to it. From these simple components, the systems description language must provide a mechanism to infer the existence of, and the relationships between, complex objects. The system description language must provide mechanisms to determine enough information about the complex objects to analyze the objects for the presence of the behavior of interest. Finally, the system description language must provide a means both to determine when a component to the system is created or destroyed and to modify the model of the system due to the creation or destruction of a component.

3.2 Attribute Grammars

The system description language which satisfies the above requirements is built upon the concept of attribute grammars. A quick introduction to attribute grammars is provided below. Readers already familiar with this subject may want to skip to section 3.3.

An attribute grammar describes both the strings accepted by a language (e.g., the syntax of the language) and a method to determine the "meaning" of those strings (e.g., the semantics of the language). An attribute grammar consists of a context-free grammar, a set of attributes for each symbol in the grammar, and a set of functions defined within the scope of a production rule in the grammar to determine the values for the attributes of each symbol in that production [1]. The following example of an attribute grammar for the definition and interpretation of binary numbers[2] will be used to clarify the relationships between these three

[2] This example is taken from [12].

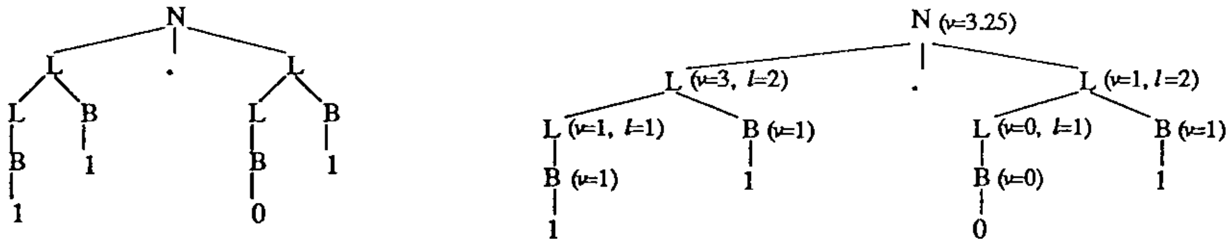
A	B	
$N \rightarrow L.L$	$N \rightarrow L_1.L_2$	$v(N) = v(L_1) + v(L_2)/2^{l(L_2)}$
$N \rightarrow L$	$N \rightarrow L$	$v(N) = v(L)$
$L \rightarrow LB$	$L_1 \rightarrow L_2B$	$v(L_1) = 2v(L_2) + v(B)$, $l(L_1) = l(L_2)+1$
$L \rightarrow B$	$L \rightarrow B$	$v(L) = v(B)$, $l(L) = 1$
$B \rightarrow 1$	$B \rightarrow 1$	$v(B) = 1$
$B \rightarrow 0$	$B \rightarrow 0$	$v(B) = 0$

Figure 3

components of an attribute grammar.

The context-free grammar for our language of binary numerals is defined by G = (V,N,P,S) where V is the set of symbols, N is the set of nonterminal symbols, P is the set of production rules, and S, an element of N, is the start symbol. The set of terminal symbols, a subset of V, is {1,0,.}. These are the ASCII characters one, zero, and period. The set of nonterminal symbols, N, is {B,L,N}. They represent the abstract objects bit, list of bits, and number. The start symbol for our attribute grammar for binary numbers is N, the abstract number. The set of production rules relating these symbols and providing the definition of acceptable strings is given in figure 3A.

By this context free grammar, we can see that the string 11.01 is an acceptable binary number. The parse tree for this string is given in figure 4A.



A B

Figure 4

The context-free grammar can be used to build a parse tree of a string and determine whether the string is valid in the language; however, the context-free grammar cannot be used to determine the meaning of the string. The addition of attributes and attribute functions are necessary to determine the meaning of the string.

The set of attributes, A, for each nonterminal are given as follows: A(B) = {*v*}, A(L) = {*v*,*l*}, and A(N) = {*v*}. The attribute *v* is the value of a symbol, and the attribute *l* is the length of a symbol.

The set of functions defined within the scope of each production rule is given in figure 3B.

By using the attributes for each symbol and the attribute functions, we can now assign meaning to each symbol in the parse tree (see figure 4B). For our language of binary numbers, the most important meaning is that of the start symbol N. Our string 11.01 now has the meaning of 3.25.

3.3 System Description Language

This section introduces the system description language, an extension of attribute grammars. This system description language provides a structure by which a system's components and relationships between components can be described. The description, or system language definition, of a system can be used to both infer the existence of complex objects (e.g., determine the syntactic structure of the system) and assign "meaning" to these objects (e.g., the semantic information about the system). The meaning of an object, the values of its attributes, will be used to determine if the behavior of interest is present in any of the components of the system.

Similar to an attribute grammar, a system language definition written in the SDL consists of a structural grammar, a set of attributes for each object, or symbol, in the structural grammar, and a set of functions defined within the scope of a production rule of the structural grammar which determine the attribute values for each object in that production.

3.3.1 Objects

Objects are the components of the system which will be modelled. These objects may or may not have real world counterparts. Two varieties of objects exist: basic objects and complex objects. Basic objects are the low-level components which are directly observed. These are similar to terminal symbols in traditional programming languages. Complex objects, on the other hand, are not observed and must be inferred from the observation of basic objects. Complex objects are similar to non-terminal symbols in traditional programming languages. These two objects are discussed further below.

3.3.1.1 Basic Objects

Basic objects are simple, indivisible components of the complex system being modelled; they are detected and their attributes determined by a preprocessor. This preprocessor performs the job of a lexical analyzer in traditional programming languages. Basic objects are treated as events; they only exist for the moment at which they are observed. For example, in the networked system, packets are basic objects. Basic objects for other systems may be an audit record from an operating system, a message to a spacecraft component, or a sampled data point from some measuring instrument.

A basic object type is defined by a name and a list of attributes. The name format for our system is the same as the standard C identifier. Attributes will be discussed in section 3.3.2. An example basic object representing a possible network packet is:

basic: packet { *attribute list* }

The keyword **basic** states that the following object type is a basic object, and the object type's name is packet. Attributes for this object will be discussed later in section 3.3.3.

3.3.1.2 Complex Objects

As mentioned previously, complex objects are components of a system which are not directly observed by the monitor, so they must be inferred from the observation of the basic objects. A complex object is composed of basic objects and/or other complex objects. For example, a complex object type called process may be defined for an audit-trail-based monitor. Although processes are not directly observed by the monitor, information about them can be inferred from the audit records. Therefore, in our model, processes are composed of audit records. Compositions will be discussed further in section 3.3.3.

A major difference between complex objects and basic objects is that complex objects have persistence. Whereas basic objects are treated as events, complex objects are treated as persistent elements which are created and possibly destroyed. The creation of a complex object occurs as soon as it can be inferred. The destruction of an object is considerably more difficult and depends on both the definition of the complex object and the existence of objects which compose the complex object. The two rules which govern the possible destruction of an object are described below.

First, if any object **A** exists and is part of an object **B**'s composition, then object **B** should continue to exist. Second, if the last object which is part of object **B**'s composition is destroyed, then object **B** will be destroyed after a specified time delay, Δt, unless another object which is part of **B**'s composition is created or observed. This specified Δt is the value of a function associated with the object, and it may depend on the object's attributes.

Complex objects can be composed of only basic objects, only complex objects, or a combination of basic and complex objects. Complex object types are defined in my system by one of the following forms depending on their composition:

complex type i: *name* { *attribute list* }

where i varies from 1 to 3 depending on the makeup of the composition objects.

3.3.2 Attributes

As mentioned previously, each object has a set of attributes associated with it. These attributes provide a "meaning" to each object. It is the attributes which will be used to determine if the object is associated with a particular behavior. These attributes are also used, along with the production rules described in section 3.3.3, to determine if an object **A** is part of object **B**'s composition.

Each attribute consists of a name and a type. The name is used to reference the value, and the type determines the value type which can be assigned or retrieved from the attribute. For example, "int value" would describe an attribute of type "int" which is referenced by the name "value." Attribute types may be complex structures defined in the same format as complex types are described in the C language [11].

Many of the attribute values of an object will be assigned by the monitor. For example, when the existence of a new host is inferred, a host object is created and its internet address is immediately assigned by the monitor. The values of other attributes, however, are determined by attribute functions. Attribute functions, described in section 3.3.4, take as input attribute values associated with the object and possibly attribute values of other objects associated with it by the production rules (see section 3.3.3).

A complex object type to represent a stream (a unidirectional flow of data from one process to another process) composed of packets can now be defined as follows:

```
complex type 1:     stream{
                              inet_addr     src_addr
                              inet_addr     dst_addr
                              int           src_port
                              int           dst_port
                              int           creation_time
                              int           num_of_packets
                              int           num_of_bytes
                    }
```

This simple definition of a process has a simple identifier, *stream*, addresses for the source and destination hosts, source and destination ports to specify the processes on the two machines, a time of creation, the number of packets exchanged between the two processes, and the number of bytes in all the packets exchanged.

The set of attributes for an object O can be defined as $A(O) = \{a_1, a_2, ..., a_n\}$. For example, A(process) = {src_addr, dst_addr, src_port, dst_port, creation_time, num_of_packets, num_of_bytes}.

3.3.3 Productions

Productions define the relationship between the different object types of a system. They define which types of objects compose a complex object, and they indicate how to determine which set of objects from an object type compose the complex object. A production rule has the form:

complex_object_type -> list of *object_composition*

The complex_object_type is simply the name of a complex object type (e.g., stream). An object_composition is a set defined by a tuple of the form <object_type, restrictions>. The object_type is simply the name of one of the defined object types (basic or complex), and the restrictions determine which of all possible objects of type object_type are actually used to compose the complex object.

For example, let the complex object type called stream be define as above, and let the object type called packet be defined as follows:

```
basic:          packet{
                              inet_addr     src_addr
                              inet_addr     dst_addr
                              int           src_port
                              int           dst_port
                              int           num_of_bytes
                              int           time
                }
```

A production rule for the stream object can now be defined as follows:

```
stream -> packet
    where for all e ∈ packet
        e.src_addr = stream.src_addr
        & e.dst_addr = stream.dst_addr
        & e.src_port = stream.src_port
        & e.dst_port = stream.dst_port
```

Finally, each element of packet which composes a particular stream object is called a sub-component of the stream object, and the stream object is called a super-component of the packet objects. The concepts of sub-components and super-components will be used in section 4.2 to define integrated object analysis functions.

3.3.4 Attribute Functions

The attributes of a complex object which are not defined by the monitor when the object is inferred are defined by attribute functions. The attribute functions for a structural language are defined as they are for attribute grammars; however, special attention must be given to the format of the production and the restriction for the production. For example, an attribute function to determine the value for the attribute "num_of_bytes" of a stream object could be as follows:

$$\text{stream.num\_of\_bytes} = \sum_{i=1}^{n} e_i.\text{num\_of\_bytes}$$

Where n = | packet |, and each e ∈ packet is assumed to be a sub-component of the stream object as defined by the restrictions in the production rule for stream objects.

4. Detecting Behaviors in Systems

Once the structural grammar, attributes, and attribute functions have been defined, a second set of functions, called behavior-detection functions, must be defined for each object in the structural grammar. Behavior-detection functions determine whether an object is associated with the particular behavior of interest. Because a behavior may manifest itself differently or more clearly in different object types, each object in a system parse tree (the snapshot of the system) must be examined for the behavior by particular behavior-detection functions designed for that object type. For each type of object, there will be two behavior-detection functions: the isolated behavior-detection function and the integrated behavior-detection functions. These two function types are discussed below.

4.1 Isolated Object Analysis

An isolated behavior-detection function for an object uses the attributes of that object to calculate the probability that the object is associated with the behavior of interest. In short, an isolated behavior-detection function is a classifier. With some preprocessing to transform the attribute types, a large number of classifiers can be used.

Unfortunately, classifiers generally have to be trained with sample data, and the behavior of interest is often quite rare. There are at least two possible solutions to the problem of lack of sample data: expert systems and single behavior classifiers. An expert system, designed by people knowledgeable about the problem domain, can use heuristics to determine how close an object's behavior is to the behavior of interest. A single behavior classifier is built around the assumption that a rare behavior will be significantly different than normal behavior [2]. If this is true, a single classifier can profile normal behavior, and then it could report any behavior which does not strongly resemble normal behavior. Work on such single behavior classifiers have been performed by SRI for IDES [13] and Los Alamos National Laboratory for Wisdom and Sense [17]. For our particular problem environment, we combined the efforts of both an expert system and a single behavior classifier.

4.2 Integrated Objects Analysis

An integrated behavior-detection function for an object modifies the result of the isolated behavior-detection function for the object by including the analysis of the isolated behavior-detection functions for sub-components and super-components of that object. The modification by an integrated behavior-detection function allows the inclusion of both the results of aggregated analysis (those from super-components) and the results of more detailed levels of analysis (those from sub-components). The integrated behavior-detection function can be implemented by a weighted average function such as:

$$\frac{W_1 * \text{Object\_if} + W_2 * \text{Super\_if} + W_3 * \text{Sub\_if}}{W_1 + W_2 + W_3}$$

Where Object_if is the value calculated by the object's isolated behavior-detection function, Super_if is the average isolated behavior-detection function value for all the super-components, Sub_if is the average isolated behavior-detection function value for all the sub-components, and W_1, W_2, and W_3 are the weights.

The relationship between an object's attributes, isolated behavior-detection functions, and integrated behavior-detection functions can be seen in figure 5. In this example, we are interested in analyzing the object B_1 for a particular behavior. The object B_1 is composed of objects C_1 and C_2, and it is part of the object A_1. Result B_{1v} is the analysis of object B_1 in isolation, and result $B_{1v'}$ is the result after combining the result of B_{1v} with the results from objects C_1, C_2, and A_1.

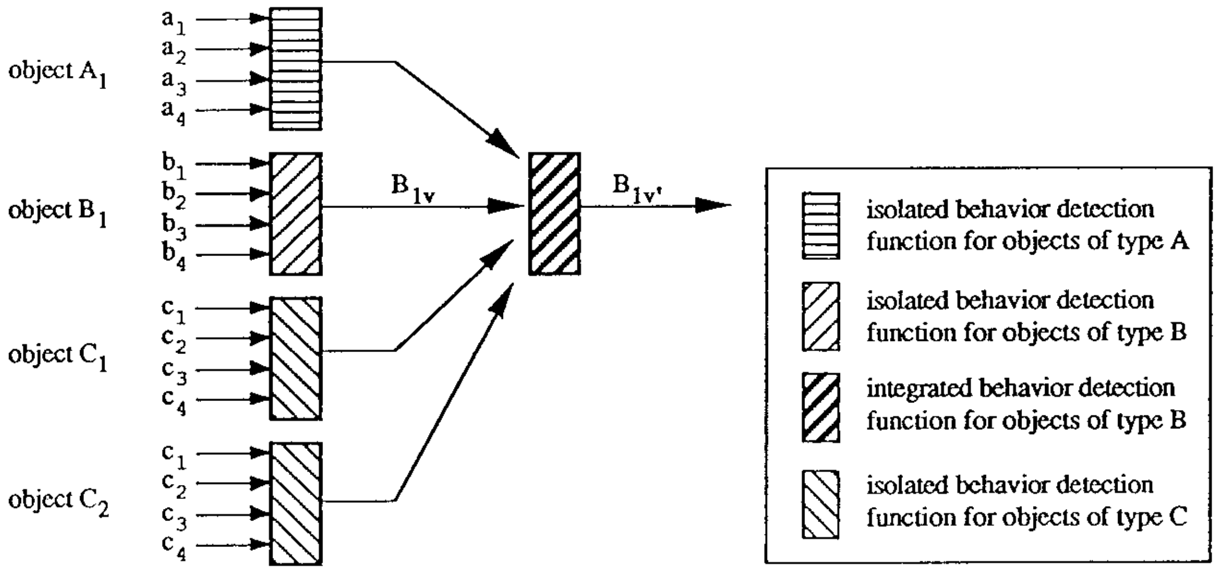

object A_1
a_1
a_2
a_3
a_4
object B_1
b_1
b_2
b_3
b_4
object C_1
c_1
c_2
c_3
c_4
object C_2
c_1
c_2
c_3
c_4
B_{1v}
$B_{1v'}$
isolated behavior detection function for objects of type A
isolated behavior detection function for objects of type B
integrated behavior detection function for objects of type B
isolated behavior detection function for objects of type C

Figure 5

5. Results and Future Research

By using the system language definition for the networked environment described in [7], one programmer was able to code both the object detector and object analyzer modules in less than two weeks. Since the coding of these modules is a straight forward implementation of the system language definition, we hope to provide automatic development tools in the future which will automatically generate object detector and object analyzer modules from a system language definition.

We have concentrated our analysis efforts on an isolated behavior-detection function for connections. This function combines a simple anomaly detector, an attack model, and an expert system to arrive at a single suspicion value. The higher the suspicion value is, the more likely our monitor believes the connection is associated with intrusive activity.

We monitored the Electrical Engineering and Computer Science LAN at UCD for a period of approximately three months. During this time over 400,000 connections were detected and analyzed, and among these connections, over 400 were identified as being associated with intrusive behavior.

Our future work includes continual improvement of the isolated behavior-detection function for connections as well as other objects in the model (i.e. service-sets, hosts, and streams). We would like to take advantage of semantic knowledge about known system vulnerabilities, and we would also like to develop profiles of intrusive activity as well as normal activity.

As mentioned previously, we are also moving towards automatic code generation for the object detector and object analyzer components of the monitor. We are currently developing a system language definition for a stand alone host based monitor too, and if we can develop automatic code generators for object detector and analyzer modules, then porting the monitor to a different operating system should be greatly simplified.

Finally, we are incorporating our network monitor into a distributed intrusion detection system called DIDS [15]. DIDS combines both host based as well as network based monitors to take advantage of the benefits of both systems.

References

1. G.V. Bochmann, "Semantic Evaluation from Left to Right," Communications of the ACM, vol. 19, no. 2, pp. 55-62, Feb. 1976.

2. D.E. Denning, "An Intrusion Detection Model," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 2, pp. 222-232, Feb. 1987.

3. P.J Denning, ed. Computers Under Attack: Intruders, Worms, and Viruses. New York: ACM Press, 1990.

4. Department of Defense Trusted Computer System Evaluation Criteria, Dept. of Defense, National Computer Security Center, DOD 5200.28-STD, Dec. 1985.

5. G.V. Dias, K.N. Levitt, B. Mukherjee., "Modeling Attacks on Computer Systems: Evaluating Vulnerabilities and Forming a Basis for Attack Detection," Technical Report CSE-90-41, University of California, Davis.

6. C. Dowell and P. Ramstedt, "The COMPUTERWATCH Data Reduction Tool," *Proc. 13th National Computer Security Conference*, pp. 99-108, Washington, D.C., Oct 1990.

7. L.T. Heberlein, "Towards Detecting Intrusions in a networked Environment," Technical Report CSE-91-23, University of California, Davis.

8. L.T. Heberlein, B. Mukherjee, K.N. Levitt, D. Mansur., "Towards Detecting Intrusions in a Networked Environment," *Proc. 14th Department of Energy Computer Security Group Conference*, May 1991.

9. L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood., "Network Attacks and an Ethernet-based Network Security Monitor," *Proc. 13th Department of Energy Computer Security Group Conference*, pp. 14.1-14.13, May 1990.

10. L.T. Heberlein, G.V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, D. Wolber., "A Network Security Monitor," *Proc. 1990 Symposium on Research in Security and Privacy*, pp. 296-304, May 1990.

11. B.W. Kernigan, D.M. Ritchie., The C Programming Language, 2nd ed. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

12. D.E. Knuth, "Semantics of Context-Free Languages," *Math Systems Th. 2* (1968), 127-145. Correction appears in *Math Systems Th.5* (1971),95.

13. T.F. Lunt, et al., "A Real Time Intrusion Detection Expert System (IDES)," Interim Progress Report, Project 6784, SRI International, May 1990.

14. S.E. Smaha, "Haystack: An Intrusion Detection System," *Proc. IEEE Fourth Aerospace Computer Security Applications Conference*, Orlando, FL, Dec. 1988.

15. 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, D.L. Mansur., "DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and an Early Prototype," to be published in *Proc. 14th National Computer Security Conference*, Oct. 1991.

16. W.T. Tener, "Discovery: an expert system in the commercial data security environment," *Security and Protection in Informations Systems: Proc. Fourth IFIO TC11 International Conference on Computer Security*, North-Holland, May 1988.

17. H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity," *Proc. 1990 Symposium on Research in Security and Privacy*, pp. 280-289, Oakland, CA, May 1989.

18. J.R. Winkler, "A Unix Prototype for Intrusion and Anomaly detection in Secure Networks," *Proc. 13th National Computer Security Conference*, pp. 115-124, Washington, D.C., Oct. 1990.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY/
NATIONAL COMPUTER SECURITY CENTER

14TH NATIONAL COMPUTER SECURITY CONFERENCE

October 1-4, 1991
Omni Shoreham Hotel
Washington, D.C.

PROCEEDINGS
VOLUME II

Information Systems Security:
Requirements & Practices