Computer System Intrusion Detection: A Survey[1]

Anita K. Jones and Robert S. Sielken
Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903
jones@cs.virginia.edu, rsielken@alumni.virginia.edu

Abstract

The ability to detect intruders in computer systems increases in importance as computers are increasingly integrated into the systems that we rely on for the correct functioning of society. This paper reviews the history of research in intrusion detection as performed in software in the context of operating systems for a single computer, a distributed system, or a network of computers. There are two basic approaches: anomaly detection and misuse detection. Both have been practiced since the 1980s. Both have naturally scaled to use in distributed systems and networks.

1 Introduction

When a user of an information system takes an action that that user was not legally allowed to take, it is called *intrusion*. The intruder may come from outside, or the intruder may be an insider, who exceeds his limited authority to take action. Whether or not the action is detrimental, it is of concern because it *might* be detrimental to the health of the system, or to the service provided by the system.

As information systems have come to be more comprehensive and a higher value asset of organizations, complex, *intrusion detection* subsystems have been incorporated as elements of operating systems, although not typically applications. Most intrusion detection systems attempt to detect suspected intrusion, and then they alert a system administrator. The technology for automated reaction to intrusion is just beginning to be fashioned. Original intrusion detection systems assumed a single, stand-alone processor system, and detection consisted of post-facto processing of audit records. Today's systems consist of multiple nodes executing multiple operating systems that are linked together to form a single distributed system. Intrusions can involve multiple intruders. The presence of multiple entities only changes the complexity, but not the fundamental problems. However, that increase in complexity is substantial.

This survey states the basic assumptions and illuminates the alternative technical approaches used to detect intrusions. There have been a number of surveys of intrusion detection

[1] This effort was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement F30602-99-1-0538. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

that essentially catalog different systems [Anderson80, Cannaday96, Liepens92, Lunt93b, Kumar94, Smaha94]. In this survey we attempt to determine the fundamental approaches and describe the essence of each approach. To be concrete, we use existing implementations to illustrate the mechanics of implementation of each approach.

Intrusion *detection* involves determining that some entity, an *intruder*, has attempted to gain, or worse, has gained unauthorized access to the system. None of the automated detection approaches of which we are aware seeks to identify an intruder before that intruder initiates interaction with the system. Of course, system administrators routinely take actions to prevent intrusion. These can include requiring passwords to be submitted before a user can gain any access to the system, fixing known vulnerabilities that an intruder might try to exploit in order to gain unauthorized access, blocking some or all network access, as well as restricting physical access. Intrusion detection systems are used in addition to such preventative measures.

Intruders are classified in two groups. *External intruders* do not have any authorized access to the system they attack. *Internal intruders* have some authority, but seek to gain additional ability to take action without legitimate authorization. J. P. Anderson divided internal intruders into three subgroups: masqueraders, clandestines, and legitimate [Anderson80]. In later related work Brownell Combs has divided internal intruders into two categories. He separates internal users who have accounts on the system from pseudo-internal intruders who are, or can be thought of as being, physically in space of legitimate users, but have no accounts [Combs98]. They do however have physical access to the same equipment used by those who have accounts. He shows how distinguishing the two categories can be distinguished enables better defense against the pseudo-internal intruders.

To limit the scope of the problem of detecting intrusions, system designers make a set of assumptions. Total physical destruction of the system, which is the ultimate denial of service, is not considered. Intrusion detection systems are usually based on the premise that the operating system, as well as the intrusion detection software, continues to function for at least some period of time so that it can alert administrators and support subsequent remedial action.

It is also assumed that intrusion detection is not a problem that can be solved once; continual vigilance is required. Complete physical isolation of a system from all possible, would-be external intruders is a simple and effective way of denying external intruders, but it may be unacceptable because physical isolation may render the system unable to perform its intended function. Some possible solution approaches cannot be used because they are in conflict with the service to be delivered.

In addition, potential internal intruders legitimately have access to the system for some purposes. So, it is assumed that at least internal intruders, and possibly external intruders, have some access and therefore have some tools with which to attempt to penetrate the system. It is typically assumed that the system, usually meaning the operating system, does have flaws that can be exploited. Today, software is too complicated to assume otherwise. New flaws may be introduced in each software upgrade. Patching the system could eliminate known vulnerabilities. However, some vulnerabilities are too expensive to fix, or their elimination would also prevent desired functionality.

Vulnerabilities are usually assumed to be independent. Even if a known vulnerability is removed, a system administrator may run intrusion detection software in order to detect *attempts* at penetration, even though they are guaranteed to fail. Most intrusion detection systems do not depend on whether specific vulnerabilities have been eliminated or not. This use of intrusion detection tools can identify a would-be intruder so that his or her other activities may be monitored. New vulnerabilities may, of course, be discovered in the future.

2 Approaches

Currently there are two basic approaches to intrusion detection. The first approach, called *anomaly detection*, is to define and characterize correct static form and/or acceptable dynamic behavior of the system, and then to detect wrongful changes or wrongful behavior. It relies on being able to define desired form or behavior of the system and then to distinguish between that and undesired or anomalous behavior. The boundary between acceptable and anomalous form of stored code and data is precisely definable. One bit of difference indicates a problem. The boundary between acceptable and anomalous behavior is much more difficult to define.

The second approach, called *misuse detection*, involves characterizing known ways to penetrate a system. Each one is usually described as a pattern. The misuse detection system monitors for explicit patterns. The pattern may be a static bit string, for example a specific virus bit string insertion. Alternatively, the pattern may describe a suspect set or sequence of actions. Patterns take a variety of forms as will be illustrated later.

Intrusion detection systems have been built to explore both approaches – anomaly detection and misuse detection – for the past 15 to 20 years. In some cases, the two kinds of detection are combined in a complementary way in a single system. There is a consensus in the community that both approaches continue to have value. In our view, no fundamentally different alternative approach has been introduced in the past decade. However, new forms of pattern specifications for misuse detection have been invented. The techniques for single systems have been adapted and scaled to address intrusion in distributed systems and in networks. Efficiency and system control have improved. User interfaces have improved, especially those for specifying new misuse patterns and for interaction with the system security administrator. Essentially all the intrusion detection implementations that will be discussed are extensions of operating systems. They use operating system notions of events, and operating system data collection, particularly audit records, as their base.

The concept of intrusion detection appears to have been first used in the 1970s and early 1980s [Anderson80]. In what we will call the first generation of intrusion detection, the emphasis was on single computer systems. Operating system audit records were post-processed. Both anomaly detection and misuse detection approaches were invented early. In the second generation, the processing became more statistically sophisticated, more behavior measures were monitored, and primitive real-time alerts became possible. A seminal paper defining an early second generation intrusion detection system implementation (IDES) appeared in 1987

[Denning87].[2] Intrusion detection was expanded to address the multiple computers in a distributed system.

We are now in the third generation of operating system based intrusion detection where networks are a major focus. The challenges are to

- manage the volume of data, communications, and processing in large scale networks,
- increase coverage (i.e. be able to recognize as much errant behavior as possible),
- decrease false alarms (benign behavior reported as intrusion),
- detect intrusions *in progress*, and
- react in real-time to avert an intrusion, or to limit potential damage.

The latter challenges are the most daunting.

3 Anomaly Detection

By definition anomalies are not nominal or normal. The anomaly detector must be able to distinguish between the anomaly and normal. We divide anomaly detection into static and dynamic. A *static* anomaly detector is based on the assumption that there is a portion of the system being monitored that should remain constant. All the static detectors that we have studied address only the software portion of a system, and are based on the tacit assumption that the hardware need not be checked. There exist system administration tools to check physical component configurations and report change; we do not treat such tools here. The static portion of a system is composed of two parts: the system code and that portion of system data that remains constant. Static portions of the system can be represented as a binary bit string or a set of such strings (such as files). If the static portion of the system ever deviates from its original form, either an error has occurred or an intruder has altered the static portion of the system. Static anomaly detectors are said to check for *data integrity*.

Dynamic anomaly detectors must include a definition of behavior. Frequently, system designers employ the notion of an *event*. System behavior is defined as a sequence (or partially ordered sequence) of distinct events. For example, many intrusion detection systems use the audit records that are (already) produced by the operating system to define the events of interest. In that case, the only behavior that can be observed is that which results in the creation of audit records by the operating system. Events may occur in a strict sequence. More often, such as with distributed systems, partial ordering of events is more appropriate. In still other cases, the order is not directly represented; only cumulative information, such as cumulative processor usage during a time interval, is maintained. In this case, thresholds are defined to separate nominal resource consumption from anomalous resource consumption.

Where it is uncertain whether behavior is anomalous or not, the system may rely on parameters that are set during initialization to reflect behavior. Initial behavior is assumed to be normal. It is measured and then used to set parameters that describe correct or nominal behavior. There is typically an unclear boundary between normal and anomalous behavior as depicted in Figure 1. If uncertain behavior is not considered anomalous, then intrusion activity may not be

[2] It should be noted that several seminal systems, such as IDES, came from researchers at the Stanford Research Institute.

detected. If uncertain behavior is considered anomalous, then system administrators may be alerted by false alarms, i.e. in cases where there is no intrusion.

Figure 1: Anomalous behavior must be distinguished from normal behavior.

The most common way to draw this boundary is with statistical distributions having a mean and standard deviation. Once the distribution has been established, a boundary can be drawn using some number of standard deviations. If an observation lies at a point outside of the (parameterized) number of standard deviations, it is reported as a possible intrusion.

A dynamic anomaly detector must define some notion of the "actor", the potential intruder. An actor is frequently defined to be a user, i.e. activity that is identified with an account and presumably then with one specific human being. Alternatively, user or system processes are monitored. The mapping between processes, accounts, and human beings is only determined when an alert is to be raised. In most operating systems there is clear traceability from any process to the user/account for which it is acting. Likewise, an operating system maintains a mapping between a process and the physical devices in use by that process. In the next sections we describe in more detail how both static and dynamic anomaly detectors have been implemented.

Anomaly detection is performed statically and dynamically by first and second generation detectors. The second generation detectors are more sophisticated on several dimensions: definitions of events or behavior of interest, compact representation of signatures, compilation of initial behavior profiles that characterize behavior of interest (whether normal or anomalous), and statistical processing techniques for divining the difference between normal and anomalous behavior. However, one difficult problem continues: coverage, i.e. the percentage of the kinds of intrusions that a specific detector will identify. In a later section we will discuss extensions of these second generation techniques from single computer and distributed processor operating systems to the more loosely connected networked systems.

It is important to note that anomaly detection which is restricted to events visible to an operating system, or behavior of that operating system in reaction to users is limited to activity of import to the operating system. Because one of the most insidious intrusions is for a user to gain the privilege of a system administrator, the semantics of the operating system are precisely those which need to be monitored for this intrusion.

3.1 Static anomaly detection

Static anomaly detectors define one or several static bit strings to define the desired state of the system. They archive a representation of that state, perhaps compressed. Periodically, the static anomaly detector compares the archived state representation to a similar representation computed based on the current state of the same static bit string(s). Any difference signals an error such as hardware failure or intrusion.

The representation of static state could be the actual bit strings selected as the definition of the static system state. However, that is quite costly in both storage and comparison computation. Because the primary objective is to determine *if* there is a difference, not to identify precisely what that difference might be. The compressed representation is called a *signature*. It is a "summary" value computed from a base bit string. The computation is designed so that a signature is computed from a different base bit string would – with high probability – have a different value. Signatures include checksums, message-digest algorithms, and hash functions[3].

Some anomaly detectors incorporate *meta-data*, or knowledge about the structure of the objects that are being checked. For example, the meta-data for a log file includes its size. A log file that has decreased in size might be a sign of an intrusion, while a log file that has increased in size would be consistent with normal operation. We will shortly describe an intrusion detector that stores information about Unix file and directory objects. The following sections provide implementation detail on two (contrasting) anomaly detection designs. Tripwire performs integrity checks using the signatures and meta-data that describe files. A second system, Self-Nonself, takes a quite different approach to signatures.

3.1.1 Tripwire

Tripwire [Kim93, Kim94] is a file integrity checker that uses signatures as well as Unix file meta-data. A configuration file, *tw.config*, specifies the static system state to be some portion of the hierarchical Unix file system. Each file defines a bitstring on which one – or more – signatures is computed. Tripwire periodically monitors the file system for added, deleted, or modified files. For each file or directory of interest, the Tripwire configuration file specifies attributes (meta-data) that are expected to remain constant. A substantial portion of the meta-data is fields of the file's Unix *inode*. More specifically, a "selection mask" is associated with each file or directory. It contains a flag for each distinct field stored in a Unix *inode*. The mask specifies which attributes can change, without being reported as an exception, as well as which should not change. Attributes include access permission to the file, *inode* number, number of links, user id, group id, size of the file, modification timestamp, and access timestamp. In addition, ten or less signature algorithms are specified. Checksums, hash functions and message digests can be used in concert. Each signature is computed based on file content.

Tripwire is initialized by building a *baseline database* based on the then-current file system state and the configuration file. It is assumed that the baseline database describes a "clean", unpenetrated system. Based on *tw.config*, Tripwire builds a baseline database containing one record per file. While the configuration file may simply name directories, the database contains a record per file in each (recursively) named directory. Note that a *tw.config* file that names the directory "/", effectively names the entire file system because Tripwire expands directories recursively. Each database entry holds a selection mask and a set of signatures.

[3] A checksum is a count of the number of bits in the string (usually used in transmissions so that the receiver can verify that it received the appropriate number of bits). A hash function computes a string of characters, usually a shorter and fixed in length, that represents the original string. A message-digest algorithm is one type of hash function.

Periodically thereafter, Tripwire re-computes elements of a current value of database entries in the same manner. This requires computing each specified signature based on current file contents. Using the selection masks from the configuration file to determine what *inode* attributes to ignore, Tripwire compares the current database to the baseline database (record by record) and issues alerts where unexpected mis-matches are found.

Tripwire assumes a Unix system; in particular it is tailored to check attributes of Unix *inodes* for files. However, the configuration file is designed to be generic and reusable for many instances of similar nodes in a networked system. The strategy could be mapped to any operating system file structure. Tripwire supports distributed systems as conveniently as single computer systems. It offers other optimizations and usability features to simplify the job of the system administrator who oversees a network of hundreds of machines. Note that while many distributed nodes may share the same generic *tw.config* configuration file, each will have a unique baseline database computed from actual node file structure and contents. Other static anomaly detection systems exist, such as COPS [Farmer94], TAMU [Safford93], and ATP [Cotrozzi93]. We chose Tripwire as representative of this class of systems.

3.1.2 Virus checkers

Virus-specific checkers [Skardhamar96] are another example of static anomaly checkers. They maintain a database of strings, each representing a telltale portion of virus code and data that is inserted as part of the virus infection. Typically, virus checkers record a modicum of meta-data so that they recognize files, or individual memory objects. The strings are short. The virus checker looks for an exact match to the specific string. In this case the presence of the string indicates virus infection. Note that in this case, the virus checker is searching for a signature strung that signifies corruption of data, not the presence of correct data as was the case for Tripwire. Most virus checkers use the actual bit string inserted by the virus. That string is short enough that there is no efficiency to be gained by compressing the string to make a shorter length signature.

3.1.3 Self-Nonself

Self-Nonself, like Tripwire, addresses the problem of assuring that static strings do not change [Forrest94]. Again, some unchanging portion of code and data is defined to be the static string to be protected. While Tripwire uses signatures, such as checksums and hash functions, to compute a value directly based on the content of a string, the Self-Nonself signatures are for *unwanted* string values, that is, strings that might result if an unwanted change were to be made to the static system state.

The Self-Nonself developers describe an analogy between their approach and that of the human immune system to ward off foreign material [Forrest94]. The human body creates T cells in the thymus and then a “censoring” process takes place: if the T cells bind with proteins, or actually peptides that are sub-units of proteins, then they are disposed of because they bind with “self”. Those T cells that do not bind are released to monitor the body for foreign material. Presumably, they will bind with that foreign material, and be a vehicle to remove it from the system.

Using the Self-Nonself technique, one views the static state as a single string. It is divided into substrings of equal length *k*. These substrings comprise a collection, *Self*, that should contain some, but not all, of the possible 2^k strings. Note that *Self* is a collection, not a set, because it may contain duplicates. The Self-Nonself approach is to generate (only) a portion of the set { 2^k – *Self* } or the complement of *Self*, called *Nonself*. The set *Nonself* contains *n detector strings*, each of length *k* and not in the collection *Self*. Efficiency calls for *Nonself* to be a set[4]. *Nonself* is the baseline database containing "negative signatures" for the static string whose integrity is to be protected. Self-Nonself monitoring consists of periodically comparing substrings of length k of the current static state to the detector strings in *Nonself*. If a match is ever found, then it indicates unexpected change, and possibly an intrusion.

Nonself could be generated in many ways. The recommended implementation is to randomly generate strings of length *k* and then to censor any that are in the collection *Self*. A substring that is not in *Self* is a detector and is added to *Nonself*. The size of *Nonself* determines a tradeoff between the efficiency of the execution with the probability of detecting anomalous behavior. The larger the size of *Nonself*, the more likely it is that an intrusion will be detected in an individual node. However, with each addition to *Nonself*, the cost of monitoring increases. The counterpart to the choice of the size of *Nonself* in Tripwire is the choice of the number of signatures for content to use in the monitoring comparisons. Both Tripwire and Self-Nonself are probabilistic in that any change which does not generate a change in the signature of the current static state will not be detected by either. One would expect that signatures which can be tailored to maximally detect changes in the underlying string will be more efficient in detecting change than randomly selected detector strings that are selected merely to be outside *Self*.

The initial generation of the set the *Nonself* is computationally expensive, but the monitoring comparisons are cheaper. In Tripwire monitoring requires computing anew signatures of the current static state for comparison. Self-Nonself requires that the current static state string be broken up into substrings and compared to detector strings. In either case, the entire current static state must be visited, so the monitoring costs appear roughly comparable.

The developers of Self-Nonself comment that perfect matching between strings of non-trivial length is rare [Forrest94]. Therefore, a partial matching rule: "for any two strings *x* and *y*, . . . *match(x,y)* is true if *x* and *y* agree (match) in at least *r* contiguous locations". Consider the following example where

x = ABADCBAB , and

y = CAGDCBBA.

With $r>3$, *match(x,y)* is false, but with $r<4$, *match(x,y)* is true. Because perfect matches are too rare, partial matching is used to increase the probability of detecting anomalous behavior.

The developers of the Self-Nonself approach were motivated by the problem of detecting virus infections, that is changes in code that should remain constant. As mentioned earlier, most virus detection systems rely on known bit strings inserted by the virus to detect the unwanted change. They are similar to the Self-Nonself system in that they monitor, seeking to find the

[4] The reference [Forrest94] uses slightly different terminology in describing the Self-Nonself approach. Because the objective of this survey is to illuminate the fundamental ideas, we introduce terminology that can be reused when describing different systems.

unwanted strings. However, they are decidedly less robust than the Self-Nonself approach in that the viral signature must be known a priori bit for bit. The Self-Nonself signatures do not rely on prior knowledge of the virus!

While the Self-Nonself signature is analogous to the signature portion of Tripwire. Self-Nonself does not check or depend upon properties of the file system, or actual (meta) data stored in the file system. As a result it will not detect improper deletion of files; it has no notion of a file. Because a deleted file is no longer present (has no strings at all), it will never contain a substring that is in *Self*. However, the Self-Nonself technique does address both string modifications and additions to the static state.

The choice of the value of *k* is important for implementation. If *Nonself* were to be empty, then all possible strings are already contained in *Self*. But detectors are needed for monitoring, so an empty *Nonself* is not useful. The solution is to increase the substring length *k* so that *Nonself* will not be computed to be empty.

Consider the use of Self-Nonself for a distributed system. Each of the separate cooperating operating systems (nodes) will be separately monitored. It is attractive to have the maximum size of *Nonself* be substantially larger than that used by a single computer. The suggested implementation is not to build the maximum size of *Nonself*, but to cause each different instance of the operating system in a distributed system to randomly generate strings and thus create its own (possibly unique) set, *Nonself*. This provides more coverage in monitoring for undesired changes. If the maximum size of *Nonself* were large enough, the distributed Self-Nonself system could be constructed to ensure a unique *Nonself* signature for each node.

Tripwire is implemented so that to have different nodes use different signatures, each node must have differences in the *tw.config* configuration file. The Tripwire developers cited the use of a common configuration file for multiple nodes as a usability advantage.

3.2 Dynamic anomaly detection

Dynamic anomaly detection requires distinguishing between normal and anomalous activity. It is intrinsically harder that distinguishing changes in static strings. Dynamic anomaly detection systems typically create a *base profile* to characterize normal, acceptable behavior. A profile consists of a set of observed measures of behavior for each of a set of dimensions. Frequently used dimensions include

- preferred choices, e.g. log-in time, log-in location, and favorite editor,
- resources consumed cumulatively or per unit time, e.g. length of interactive session or number of messages emitted into a network per unit time, and
- representative sequences of actions.

Dimensions may be specific to the type of the entity with which behavior is associated. Typical entity types are users, workstations, or remote hosts as in NIDES [Anderson95a,b, Javitz93, Lunt93a] or even applications, as in SRI Safeguard [Anderson93]. An intrusion

detection system develops a unique base profile (typically based on observed behavior) for each individual entity that it recognizes. It assumes that the profile is untainted by intrusive activity.

After base profile initialization, intrusion detection can commence. Dynamic detectors are similar to the static detectors in that they monitor behavior by comparing current characterization of behavior to the initial characterization of expected behavior (the base profile). They seek divergence. As the intrusion detection system executes, it observes events that are related to the entity or actions that are attributed to the entity. It incrementally builds a *current* (possibly always incomplete) *profile*. Early generation systems depended upon audit records to capture the events or actions of interest. Some later generation systems record a data base specific for intrusion detection. Some operate in real-time, or near real-time, and more directly observe the events of interest during their occurrence, rather than waiting for the operating system to make a record describing an event.

Different sub-sequences of events relate to different profile dimensions. So, for example, if audit records are used to define events of interest, one audit record may reflect initiation of an interactive session for an entity. Then, for the purposes of the "session duration" dimension of the entity's profile, all audit records until a session termination event may be ignored. One portion of the detector sequentially processes audit records, updating the profile dimensions that each one affects.

Because there is typically wide variation in acceptable entity behavior, deviation from the base profile is often measured in statistical terms. Normal behavior is distinguished from abnormal behavior based on empirically determined thresholds, or standard deviation measures. In some systems the profiles are slowly aged to reflect gradual behavioral changes. Another portion of the dynamic anomaly detector periodically compares the incrementally built, current profiles to base profiles. The "difference" will be computed. It will be compared to a statistically defined threshold to determine whether the difference is so great that it indicates anomalous behavior. For example, an inordinately long session will eventually appear to be anomalous when "session duration" exceeds some threshold.

The main difficulty with dynamic anomaly detection systems is that they must build sufficiently accurate base profiles and then recognize deviant behavior that is in conflict with the profile. Base profiles can be built by synthetically running the system or by observing normal user behavior over a sufficiently long time.

Errant behavior of an entity whose behavior typically varies within tight bounds will be easier to detect when it starts to deviate. On the other hand, an entity that exhibits diverse activities will be characterized by a base profile with wider bounds. If genuinely anomalous behavior falls within observed base profile bounds, it will not be recognized as anomalous. An intruder masquerading as a diverse user would be much more difficult to detect because that user's base profile bounds are larger.

3.2.1 NIDES

The Next-generation Intrusion Detection Expert System (NIDES) [Anderson95a,b, Javitz93, Lunt93a], developed by SRI, contains a statistical dynamic anomaly detector. NIDES

builds statistical profiles of users, though the entities monitored can also be workstations, network of workstations, remote hosts, groups of users, or application programs. NIDES uses statistically unusual behavior to detect an intruder masquerading as a legitimate user.

NIDES reads audit records written by the operating system. NIDES measures fall into several generic classes:

- Audit record distribution – tracks the types of audit records that have been generated in some specified period of time.
- Categorical – transaction-specific information, such as user name, names of files accessed, and identity of machines onto which user has logged.
- Continuous – any measure for which the outcome is a count of how often some event occurred, such as elapsed user CPU time, total number of open files, and number of pages read from secondary storage.

For continuous measures, NIDES defines a sequence of intervals or "bins"; thirty-two is often cited. The "bins" contain a count of the number of observations with values in the interval represented by the "bin". For example, one profile dimension of a user includes a distribution of the total memory size of the user's processes during execution. At any point the user's current profile is the distribution of periodically sampled total memory size. The current and base distributions can be compared for similarity.

To maintain each statistical profile dimension, NIDES stores only statistics such as frequencies, means, variances, and covariances of the profile since storing the audit data itself is too cumbersome. Given a profile with *n* measures, NIDES characterizes any point in the *n*-space of the measures to be anomalous if it is sufficiently far from an expected or defined value. In some cited experiments, this was defined as two standard deviations. In this manner, NIDES evaluates the total deviation and not just the deviation of each individual measure.

The profile measures of each entity are subject to an exponential decay factor so that the older audit records have less of an impact on the statistical measures while the newer records have the most weight in the determination of the statistical distributions.

3.2.2 UNM Pattern Matching

Modern programming for decades has used the notion of a function, procedure or method as a way to package units of code for use, by invocation. That means that a sequence of invocations characterizes the sequential behavior of a program. Invocation sequences offer an alternative to audit record events. In the section discussing misuse detection, we will see repeated use of invocations to characterize known intrusions.

Researchers at the University of New Mexico (UNM) have taken operating system routine invocations as the definition of system behavior [Hofmeyer97]. They elect to monitor only those system routines that execute with privileges over and above those of an ordinary user. (Ordinary users can, of course, invoke system routines.) The researchers associate with each system routine a profile that consists of fixed length, *k*, sequences of calls made by the (privileged) system routine.

To illustrate the construction of the profile, the UNM researchers created traces of system calls by a selected system routine, such as Unix *sendmail*. Parameters were ignored. So for example, assume that the sequence length is selected to be three and given the trace of systems calls made by *sendmail*:

open, read, mmap, mmap, open, read, mmap

The resulting sequences are:

open, read, mmap
read, mmap, mmap
mmap, mmap, open
mmap, open, read

Two techniques were used to develop traces. The first method involved creating a variety of synthetic invocations. For example, invoking *sendmail* for different situations, will induce different calls by *sendmail*. Sending messages of different sizes results in different traces as does forwarding, bounced mail, vacation, and sending messages to multiple recipients. The resulting size of the profile database for $k = 10$ for three Unix programs was

Program[5]	**Profile database size**
sendmail	1318
lpr	198
ftpd	1017

The second method for building the database is to observe normal usage for a period of time. In both cases the database is not guaranteed to hold all sequences derivable from legal functioning of the routine at hand.

In experiments, the UNM researchers report that the sequence length of ten proved useful. When *k* equals one, there were too few mismatches possible. Once *k* had the value between 6 and 10, empirical observation showed that increasing the sequence length was not particularly useful.

A set of experiments was performed to detect intrusions that exploit flaws in the three programs based on five known intrusions (three for *sendmail*, one for *lpr*, and one for *ftpd)* and three intrusions for which the system had been configured, so that the intrusion attempt would be unsuccessful. Based on the intrusion code, sequences of length *k* were generated and compared against the profile database. The number of mismatches – sequences not in the database – is an indicator of anomalous behavior. UNM researchers report that the majority of intrusions were detectable due to notable numbers of mismatches between the *k*-length sequences of the intrusions and the synthetic database sequences.

Whether the profile database is built synthetically or is built based on sequences observed during initialization, there is a question of how large the database should be to ensure that anomalies will be detected, but false alarms will not raised. It is a matter for experimentation. Reported experiments indicated that if the profile database was built from 700 to 1400 invocations of *lpr* and then roughly 1000 to 1300 invocations were tested, the false alarm rate was between one and two. However, it is interesting to note that a number of the false alarms were

[5] *sendmail* sends messages to designated recipients, *lpr* prints a file to a device and *ftpd* handles connections to a file transfer protocol (FTP) server.

based on unusual circumstances. They included printing on a machine on which a printer did not exist, printing a job so large that the print spooler ran out of disk space to store the log file, and printing from a separately administered machine whose configuration differed. An intrusion detection system that flags rare events could be useful to system administration and to users.

4 Misuse Detection

Misuse detection is concerned with catching intruders who are attempting to break into a system using some known technique. Ideally, a system security administrator would be aware of all the known vulnerabilities and would eliminate them. However, as was mentioned earlier, an organization may decide that eliminating know vulnerabilities is cost prohibitive or unduly constrains system functionality. In practice, many administrators do not remove vulnerabilities even when they might. Note that users who slowly modify their activity so that their profiles contain the abusive activity are nearly impossible to detect with anomaly detectors. So, misuse detection systems look for known intrusions irrespective of the user's normal behavior.

We use the term *intrusion scenario* to mean a description of a fairly precisely known kind of intrusion; it is typically defined as a (partial) sequence of actions, that when taken, result in an intrusion, unless some outside intervention prevents completion of the sequence. A misuse detection system typically will continually compare current system activity to a set of intrusion scenarios in an attempt to detect a scenario in progress. The model or description of the intrusion scenario will substantially determine how efficient monitoring can be. Current system activity as seen by the intrusion detection system may be real time observations strictly for the use of the intrusion detection system, or it can be the audit records as recorded by the operating system. Although the systems described below use audit records, they would be fundamentally the same if they were collecting unique, real-time system information.

The main difference between the misuse detection techniques described below is in how they describe or model the bad behavior that constitutes an intrusion. First generation misuse detection systems used rules to describe what security administrators looked for within the system. Large numbers of rules accumulated and proved to be difficult to interpret and modify because they were not necessarily grouped by intrusion scenario.

To overcome these difficulties, second generation systems introduced alternative scenario representations. These include model-based rule organizations and state transition representations. These have proved to be more intuitive for the misuse detection system user who needs to express and understand the scenarios. Since the system will need to be constantly maintained and updated to cope with newly discovered intrusion scenarios, ease of use is a major concern.

Since the intrusion scenarios can be specified relatively precisely, misuse detection systems can track the intrusion attempt against the intrusion scenario action by action. During the sequence of actions, the intrusion detection system can anticipate the next step of the possible intrusion. Given this information, the detector can more deeply analyze system information to check for the next step or can determine that the intrusion attempt has proceeded far enough and intervene to mitigate possible damage. The model-based and state transition characterizations lend themselves to anticipation of intrusion.

4.1 Rule-Based Systems

Expert systems detect intrusions by encoding intrusion scenarios as a set of rules. These rules reflect the partially ordered sequence of actions that comprise the intrusion scenario. Some rules may be applicable to more than one intrusion scenario.

The system state is represented in a *knowledge base* consisting of a *fact base* and a *rule base*. A *fact base* is a collection of assertions that can be made based on accumulated data from the audit records or directly from system activity monitoring. The *rule base* contains the rules that describe known intrusion scenario(s) or generic techniques. When a pattern of a rule's antecedent matches the asserted fact, a *rule-fact binding* is created. After this binding is made, if all the patterns of the rule have been matched, then a binding analysis is performed to make sure that all the associated variables with the rule are consistent with the binding. The rules with rule-fact bindings that meet the binding analysis requirements are then gathered into a set from which the "best" rule is picked, through a process called conflict resolution. The rule then *fires*. It may cause an alert to be raised for a system administrator. Alternatively, some automated response, such as terminating that user's session, will be taken. Normally, a rule firing will result in additional assertions being added to the fact base. They, in turn, may lead to additional rule-fact bindings. This process continues until there are no more rules to be fired.

Consider the intrusion scenario in which two or more unsuccessful login attempts are made in a period of time shorter than it would take a human to type in the login information at a conventional keyboard. If the rule or rules of this scenario fire, then a specific user's suspicion level can be increased. The system may raise an alarm or freeze the named user's account. Account freeze would be entered into the fact database.

The following sections demonstrate the progress of the development of rule-based misuse detection. The first system, MIDAS, uses the basic rule-based system. The second, IDES/NIDES, was originally designed to be rule-based, but then the design was changed to the model based organization of the rule base.

4.1.1 MIDAS

MIDAS (Multics Intrusion Detection and Alerting System) [Sebring88] was designed and written to perform rule-based intrusion detection. For developing, compiling, and debugging the rules, MIDAS uses the Production-Based Expert System Toolset (P-BEST) that is a forward-chaining, LISP based development environment [Lindqvist99]. The P-BEST compiler produces primitive LISP functions that embody the semantics of the rules. The MIDAS rule base grew to be very large, so it was subdivided by the type of intrusion for which each rule was designed to detect. The system was designed to take some predefined action once it detected an intrusion. A secondary set of rules determine what action should be taken by the system. These secondary rules are kept separate from the primary rules to help keep the rule base size manageable for user maintainability.

The following figures (2a and 2b) illustrate MIDAS rules. The first rule deals with an intrusion scenario dealing with attempted privileged account intrusions. The rule monitors the knowledge base waiting for an assertion of a failed login attempt to an account with an account

name that is either privileged on this system or is the common name of privileged accounts on other systems. When the rule fires, it warns the MIDAS operator and adds a "remember" fact to the fact base stating that there is a high probability that an intrusion attempt occurred. Words preceded by a question mark denote variables that are during rule antecedent matching.

(*defrule* illegal_privileged_account *states*
if there exists a failed_login_item
such that name *is* ("root" or "superuser" or "maintenance" or "system") *and*
time *is* ?time_stamp *and*
channel *is* ?channel
then
(print "WARNING: ATTEMPTED LOGIN TO PRIVILEGED ACCOUNT")
and remember a breakin_attempt
with certainty *high*
such that attack_time *is* ?time_stamp
and login_channel *is* ?channel)

Figure 2a. Illegal privileged account access rule

The second example rule defines an intrusion scenario involving unusual login times. This rule is used to determine when a login to an account was made outside of "normal" hours. This example illustrates that some implementations use intrusion scenarios described as rules in lieu of statistical anomaly detection, but the rule can only look for specifics (a particular value) and not a parameterized range of values. This example also illustrates that the intrusion scenario describes unusual behavior that does not necessarily constitute an intrusion.

(*defrule* unusual_login_time *states*
if there exists a login_entry
such that user *is* ?userid *and*
time_stamp *is* ?login_time *and*
(unusual_login_time ?userid ?login_time)
then
remember a user_login_anomaly
such that user *is* ?userid *and*
time_stamp *is* ?login_time)

Figure 2b. Unusual login time rule

4.1.2 IDES/NIDES

Initially, IDES [Lunt89] was designed with a simple rule-based system to detect intrusion attempts using intrusion scenarios described by rule sets. The rule-based component was based on the same Production-Based Expert System Toolset (P-BEST) that MIDAS used. The *rule base* was divided into two parts for easier maintainability and understanding. The generic rules are those that can be applied to many different types of target systems under a number of configurations. The second part of the rules are those that are either operating system or implementation dependent. IDES was a predecessor of NIDES.

Although simple rule-based systems can be useful, they are reported to be hindered by a lack of support for developing the intrusion scenarios. It is difficult to determine the relations between rules. The sheer magnitude of the rule sets make it difficult to isolate a subset in order to make a change. To overcome this difficulty, the concept of *model-based* intrusion detection [Garvey91] was developed in conjunction with the IDES project at Stanford Research Institute. Each intrusion scenario was separately modeled so that the number of rules that need to be considered in making a change is a more manageable size.

A performance issue is involved here as well. Since the model-based approach organizes the rules by intrusion scenario, only the rules used to check for the initial steps in the intrusion need to be fired. The other rules remain dormant. Once an intrusion scenario is begun (by the first rule of that scenario being satisfied), additional rules for detecting the subsequent steps of the intrusion can be added to the set of rules that must be evaluated. In the initial rule-based approach, none of the rules were dormant, so they were all constantly being evaluated. Therefore, the model-based approach gains both efficiency and improved maintainability.

4.2 State-based intrusion scenario representations

In state based representations, attribute-value pairs characterize systems states of interest. Actions that contribute to intrusion scenarios are defined as transitions between states. Each action changes the value of attribute(s) of interest. Intrusion scenarios are defined in the form of state transition diagrams. Nodes represent system states and arcs represent relevant actions. The action causes a transition between states and determines how the attribute values of the prior state change as a result of a transition.

The state of the system is a function of all the users, processes, and data present in a system at any given point. A state transition diagram that defines an intrusion scenario consists

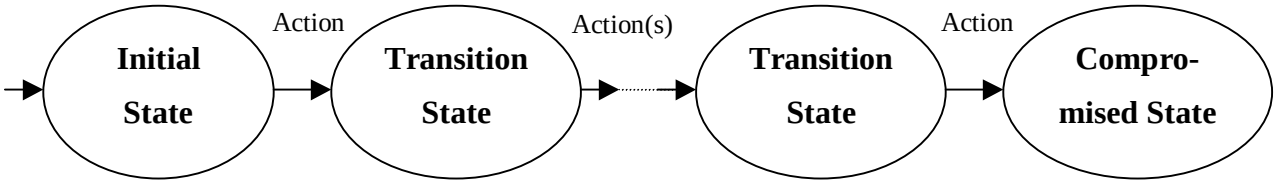

Figure 3. Generic State Transition Diagram

of an *initial state*, the state before the intrusion, and a *compromised state*, the state after the intrusion has been completed, as illustrated in figure 3. In between are some number of *transition states*. Actions of interest are those taken by the would-be intruder to attain the compromised state. Actions that do not involve a labeled arc emanating from a current state (initial or transition) are ignored for the purposes of a specific intrusion scenario. If a compromised (final) state is ever reached, an intrusion is said to have occurred.

4.2.1 USTAT

USTAT (UNIX State Transition Analysis Tool) provides an excellent illustration of the implementation of the state-based approach [Porras92]. USTAT is tailored to the UNIX

environment [Ilgun93]. Each known penetration, or intrusion scenario, is represented in the form of a state transition diagram. Some action, for example UNIX system routines that change system state, are the transitions from one state to the next.

USTAT processes audit records from the particular UNIX system on which it was implemented. The more-than-200 audit events were mapped onto ten USTAT actions, such as read(file_var), modify_owner(file_var) and hardlink(file_var, file_var), where each instance of "file_var" is the name of some file.

States are each defined by as *set* of assertions, each of which evaluates to true or false. An example assertion is of the form: member(file_set, file_var), which evaluates to true if file_var is a member of the file_set.

To monitor for intrusions, an inference engine maintains a table that holds a *row* for each possible intrusion that may be in progress. The inference engine processes audit events sequentially. It maps each event to a corresponding USTAT action. It then checks all rows in the table to determine if that action causes a transition from the current state (of a diagram that is represented by a row in the table) to its successor state. If so, the inference engine adds a copy of that row to the table and marks it as being in the successor state. The original row remains because another later action could repeat the same action in another penetration attempt using the same scenario.

An action that causes transition to the final state of a diagram indicates an intrusion. A separate decision engine determines what action to take.

5 Extensions – distributed systems and networks

Intrusion detection for a distributed system, or for a (more loosely coupled) network of machines is basically similar to that of intrusion detection for a single operating system. We refer to them as networked systems from now on. Intrusion scenarios are still based on actions taken by entities. However, the multiple users of a network system can work together as part of a *cooperative* intrusion in which multiple entities collaborate to implement the intrusion. These entities may represent different humans or may be the same human using different identifications, possibly on different machines. Casual experience shows that cooperative intrusions in a network are more frequent than single entity intrusions and provide more options for intrusive activity. The intruder(s) can use the multiple nodes in an attempt to disguise their activities. They take advantage of the fact that different operating systems may be unaware of each other's state. To detect network-based intrusion, the detector must be able to correlate actions, and possibly users, from multiple nodes involved in a cooperative intrusion.

The single-system approaches discussed earlier, anomaly detection and misuse detection, have been scaled up to deal with intrusion in network systems. Audit data, system routine invocations, and system state information are collected and then analyzed in a very similar way as for single operating systems. As one would expect, the system calls that result in network activity figure prominently in the definition of normal/anomalous behavior and intrusion scenarios. The difference between the singular case and the network case is that the intrusion detection system must aggregate and correlate the information from multiple hosts and the

network. To accomplish this task, the detector can either apply a centralized approach in which all the information is collected in a single location and then analyzed, or it may use a decentralized (hierarchical) approach where local information is analyzed and only selected, vital information is shared between the intrusion detection components across nodes.

By correlating each action to a particular user, the intrusion detection system encounters the problem commonly called the Network-user Identification (NID) problem. It is the problem of tracking a user moving around in the network using possibly many different user-ids on each machine. There is some disagreement on how much of a problem this presents, but it is a problem to some degree in all the systems. [Snapp91] argues that the NID problem exists in both detecting the intrusion and knowing on whom to focus mitigation; [Kemmerer97] claims that the NID problem is only a problem for the mitigation aspect. No matter how many times a human logs-in on different machines through the network with different ids, there is always only one human from which all the logins originated. The key to solving this problem is being able to find the initial log-in and use that identification as the originating-id for all the other different ids that have been "derived" from that originating-id. One way to do this it to check for sequences of actions that a user takes after logging in. Frequently, a user will run certain scripts or commands in a distinguishable sequence immediately after login.

Other difficulties in performing network intrusion detection include the classical problem of synchronization of either the clocks for different nodes, or just the audit record time stamps from different nodes. Since actions may be temporally dependent, keeping clocks in the distributed system synchronized is essential to being able to match sequences of actions in the system with the sequences of actions in a defined intrusion scenario.

5.1 Centralized analysis

Centralized network intrusion detection systems are characterized by distributed audit collection and centralized analysis. Most, if not all, of the audit data is collected on the individual systems and then reported to some centralized location where the intrusion detection analysis is performed. This approach works well for smaller network situation but is inadequate for larger networks due to the sheer volume of audit data that must be analyzed by the central analysis component.

An example of an intrusion that is operating system dependent is the *setuid* shell intrusion that is possible in SunOS but not in Solaris. The intrusion detector must be able to distinguish between different audit trails since some different intrusion scenarios may apply to each different operating system being run. This is the problem with performing centralized analysis on information collected from a collection of heterogeneous system components.

5.1.1 DIDS

Distributed Intrusion Detection System (DIDS) illustrates the centralized approach to network intrusion detection [Snapp91]. DIDS is basically a collection of multiple intrusion detection systems running on individual systems that cooperate to detect network-wide intrusions. The intrusion detection components on the individual systems are responsible for collecting the system information and converting it into a homogeneous form to be passed to the

central analyzer. By converting the audit data into a homogeneous format, DIDS is able to handle heterogeneous individual systems with just one centralized intrusion detection system.

DIDS does extend the non-distributed intrusion detectors by monitoring network traffic and aggregating all the information from the individual intrusion detectors. Once the information about the individual systems and the network itself has been collected in the centralized location, it can be analyzed as if it were a single system using some combination of anomaly and misuse detection approaches. NADIR (Network Anomaly Detection and Intrusion Reporter) follows a similar approach to that taken by DIDS [Hochberg93].

5.1.2 NSTAT

NSTAT (Network State Transition Analysis Tool) also performs centralized network intrusion detection [Kemmerer97]. NSTAT collects the audit data from multiple hosts and combines the data into a single, chronological audit trail to be analyzed by a modified version of USTAT. To chronologically maintain the audit trail, each component sends a *sync* message periodically to make sure that the clocks are synchronized within some threshold. Like DIDS, NSTAT can handle many heterogeneous audit trail formats since the local audit trail is converted to a common NSTAT format before it is sent across the network via an encrypted socket connection. The intrusion detection analysis is similar to that described for USTAT, a predecessor of NSTAT.

5.2 Decentralized (hierarchical) analysis

Decentralized network intrusion detection systems are characterized by distributed audit data collection followed by distributed intrusion detection analysis. These systems can be modeled as hierarchies. Unlike the centralized network intrusion detection systems, these systems are better able to scale to larger networks because the analysis component is distributed and less of the audit information must be shared between the different components.

For the decentralized approach, there must be some way of partitioning the entire system into different, smaller domains for the purpose of communication. A *domain* is some subset of the hierarchy consisting of a node that is responsible for collecting and analyzing all the data from all the other nodes in the domain; this analyzing node represents the domain to the nodes higher up in the hierarchy. Domains can be constructed by dividing the system based on

- geography,
- administrative control,
- collections of similar software platforms, or
- partitions based on anticipated types of intrusions.

For example, audited events from nodes running the same operating system can be sent to a central collection point so that the homogeneous systems can be analyzed in concert.

5.2.1 GrIDS

The Graph Based Intrusion Detection System (GrIDS) uses a decentralized approach [Staniford-Chen96]. GrIDS is concerned with detecting intrusions that involve connections between many nodes. It constructs *activity graphs* to represent host activity in a network. The system being observed is broken into domains as described above ([Staniford-Chen96] calls

these *departments*). Graphs consist of nodes representing the domains and edges representing the network traffic between them. If a domain Z is the parent domain of domains A, B, and C, then Z collects the information from the three (A,B, and C) and analyzes it. Z then represents the domain of A, B, and C to the next level in the hierarchy. Therefore, the parent domain of Z will only collect information from Z and not the individual domains. For scalability reasons, each domain builds its own graph and then passes the graph and summary information up to its parent domain. As the information passes up the hierarchy, the graphs become coarser and coarser with each child node representing a lower domain that may have numerous nodes and/or sub-domains.

Although it is a decentralized intrusion detection system, GrIDS uses a rule set to determine how the graphs are built from the incoming and previous information. Rules are applied to determine whether or not a graph is suspicious – i.e. whether it represents a possible intrusion. The rule set also specifies how graphs can be combined, based on common nodes or edges. GrIDS allows multiple rule sets. Each may operate independently of others. Since rule sets can be complicated and difficult to write, GrIDS includes a policy specification tool that more easily allows the specification of acceptable and unacceptable behavior. From this policy specification, GrIDS builds the appropriate rule sets.

5.2.2 EMERALD

EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances) uses a three-layer hierarchical approach to large-scale system intrusion detection [Porras97]. Each of the three layers consists of monitors. Each monitor may have its own anomaly and misuse detectors. The layers are named: service (lowest), domain-wide, and enterprise-wide (highest). The service layer monitors a single domain. The monitors in the domain-wide layer accept input from the service layer monitors and attempts to detect intrusions across multiple, service domains. Likewise, the enterprise-wide monitor accepts input from the domain-wide monitors and attempts to detect intrusions that cross the entire system.

Information exchange can go up and down the hierarchy. Monitors may subscribe to information from other monitors at the same level and lower. This "push-pull" information structure allows the system to scale better than the centralized network intrusion detection systems.

5.2.3 Common Intrusion Detection Framework (CIDF)

A natural extension of the hierarchical approach to intrusion detection is using multiple intrusion detection systems to form a new intrusion detection system that can utilize the best portions of each intrusion detection system. For these individual systems to cooperate with each other, there must be some standardization between the heterogeneous intrusion detection subsystems on issues such as deciding on a common vocabulary, information format, and protocols for sharing information.

One such formalization is the Common Intrusion Detection Framework (CIDF) [Kahn98] sponsored by the Defense Advanced Research Projects Agency (DARPA). The CIDF working group is composed of numerous researchers collaborating in an effort to allow their respective intrusion detection systems to interoperate. The CIDF already includes the Common Intrusion Specification Language (CISL) for expressing event data, analysis results, and responses to directives from other intrusion detection systems.

6 Vulnerabilities

Intrusion detection software mechanisms themselves are not inherently survivable; they too require some protection to prevent an intruder from manipulating the intrusion detection system to avoid detection. Most systems depend upon the assumption that the intrusion detection system itself, including executables and data, cannot be tampered with. Fortunately, many of these problems are classical security problems that have been studied in depth. Physical security of the system itself must be maintained. Also, the data files from which the intrusion detection system operates must be kept secure. Well-guarded access and physical measures such as read-only data storage are used.

Some intrusion detection systems initialize by creating a database intended to define "normal" behavior. That initialization will be flawed if the intrusion(s) are present.

Since many of the current intrusion detection systems rely on audit trail information, audit data must be available to the intrusion detection system in a timely manner. Long gaps between receiving audit records can render an intrusion detector less useful because an intrusion can take place in a relatively short time. The intrusion detector should have some built-in survivability to handle the case of infrequent audit records.

Intrusion detection system designers have to be conscious about the coexistence of the intrusion detector with the rest of the system. The system being guarded and the intrusion detector should not compete for the same processor, because doing so would make the intrusion detector vulnerable to denial-of-service attacks. Executing the intrusion detection system on a separate computer with its own processor and memory can solve most of these problems.

7 Conclusions

About twenty years of research have produced efficient, effective intrusion detection systems. They are based on two fundamental approaches: the detection of anomalous behavior as it deviates from normal behavior, and misuse detection. These two approaches were originally developed for single operating systems. In the second generation, they were extended and scaled to address distributed systems. In the third generation, they were extended to address loosely coupled networks of otherwise unrelated systems. While the approaches for networked systems are basically the same as for single operating systems, there are two primary challenges: tracking users as they move through nodes in the network and managing the data needed by the intrusion detectors as the size of the network scales up.

We posit that for the next/fourth generation of intrusion detectors, it is urgent to find some new approaches. We expect that the current approaches will become more accurate because the semantics of operating systems, and the protocols that knit multiple computers together into an interdependent network, will be more precisely defined. It will be possible to monitor for and detect unusual behavior based on more precise and more formal descriptions of behavior. The University of New Mexico research that devised patterns that, in effect, recognized "unusual" behavior provides a particularly creative approach for characterizing system behavior.

Network intrusion attempts will be more easily detected if it is possible to actively trace back from messages in one computer through intermediaries to the originating computer with high assurance. TCP/IP v6 will raise the associated issues of policy, privacy, and administrative control.

Intrusion detection in the first three generations has almost wholly focused on intruders who seek to penetrate the operating system, and in the jargon of Unix, attempt to gain "root" privilege. Operating system based intrusion detection is well understood at this point. The weakest aspect is the problem tackled in the third generation: networks. If new network protocols permit active tracing and identification of (external) intruders through multiple network nodes, the network problems of today will be dramatically reduced.

At that point the main threat will come from internal intruders, those with limited authority, seeking to extend that authority, particularly in the context of their applications. Users who seek to gain application privileges will likely be invisible at the operating system level, and thus invisible to the most of the kind of intrusion detectors that this survey addresses. We envision the need for application intrusion detection systems that relate to and exploit the semantics of the application, not to those of the operating system. These types of detectors will be the keystone of the fourth generation intrusion detectors that are still to come. An initial analysis of how to approach application intrusion detection can be found in [Sielken99a, 99b].

8 References

[Anderson80] Anderson, J.P. "Computer Security Threat Monitoring and Surveillance." Technical Report, James P. Anderson Co., Fort Washington, Pennsylvania, April 1980.

[Anderson93] Anderson, D. T. Lunt, H. Javitz, A. Tamaru, and A. Valdes. "Safeguard Final Report: Detecting Unusual Program Behavior Using the NIDES Statistical Component." *SRI International Computer Science Laboratory Technical Report*, December 1993.

[Anderson95a] Anderson, D., T. Lunt, H. Javitz, A. Tamaru and A. Valdes. "Detecting Unusual Program Behavior Using the Statistical Component of the Next-generation Intrusion Detection Expert System (NIDES)." *SRI International Computer Science Laboratory Technical Report SRI-CSL-95-06*, May 1995.

[Anderson95b] Anderson, D., T. Frivold and A. Valdes. "Next-generation Intrusion Detection Expert System (NIDES): A Summary." *SRI International Computer Science Laboratory Technical Report SRI-CSL-95-07*, May 1995.

[Cannady96] Cannady, J. and J. Harrell. "A Comparative Analysis of Current Intrusion Detection Technologies." *4th Technology for Information Security Conference (TISC'96)*, May 1996.

[Combs98] Combs, Brownell. "The Pseudo-Internal Intruder; a new Access Oriented Intruder Category." University of Virginia Technical Report, 1999.

[Cotrozzi93] Cotrozzi, M. and D. Vincenzetti. "ATP – Anti-Tampering Program." *UNIX Security IV Symposium (USENIX)*, October 1993.

[Debar92] Debar, H., M. Becker and D. Siboni. "A Neural Network Component for and Intrusion Detection System." *Proceedings of the IEEE Symposium on Research in Computer Security and Privacy*, 1992.

[Denning87] Denning, D. "An Intrusion Detection Model." *IEEE Transactions on Software Engineering*, 13.2 (1987) 222.

[Farmer94] Farmer, D. and E. Spafford. "The COPS Security Checker Systems." *Purdue Technical Report CSD-TR-993*, January 1994.

[Forrest94] Forrest, S., L. Allen, A.S. Perelson, and R. Cherukuri. "Self-Nonself Discrimination in a Computer." *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, 1994.

[Garvey91] Garvey, T.D. and T.F. Lunt. "Model-Based Intrusion Detection." *Proceedings of the 14th National Computer Security Conference*, October 1991.

[Hochberg93] Hochberg, J., K. Jackson, C.Stallings, J.F. McClary, D. DuBois, and J. Ford. "NADIR: An Automated System for Detecting Network Intrusion and Misuse." *Computers and Security*, 12.3 (1993) 235-248, http://nadir.lanl.gov/libLA-UR-93-137.html.

[Hofmeyer97] Hofmeyer, S.A., S. Forrest, and A. Somayaji. "Intrusion Detection using Sequences of System Calls." Revised: December 17, 1997. http://www.cs.unm.edu/~steveah/publications/ids.ps.gz.

[Ilgun93] Ilgun, K. "USTAT: A Real-Time Intrusion Detection System for UNIX." *Proceedings of the IEEE Symposium on Research in Security and Privacy*, May 1993.

[Javitz93] Javitz, H.S. and A. Valdes. "The NIDES Statistical Component: Description and Justification." ftp://ftp.csl.sri.com/pub/nides/reports/statreport.ps.gz, March 1993.

[Kahn98] Kahn, C., P. Porras, S. Staniford-Chen, and B. Tung. "A Common Intrusion Detection Framework." Submitted to *Journal of Computer Security*, July 1998.

[Kemmerer97] Kemmerer, R.A. "NSTAT: A Model-based Real-time Network Intrusion Detection System." *University of California-Santa Barbara Technical Report TRCS97-18*, November 1997.

[Kim93] Kim, G.H. and E.H. Spafford. "A Design and Implementation of Tripwire: A File System Integrity Checker." *Purdue Technical Report CSD-TR-93-071*, November 1993.

[Kim94] Kim, G.H. and E.H. Spafford. "Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection." *Purdue Technical Report CSD-TR-94-012*, February 1994.

[Kumar94] Kumar, S. and E. Spafford. “An Application of Pattern Matching in Intrusion Detection.” *Purdue Technical Report CSD-TR-94-013*, June 1994.

[Lindqvist99] Lindqvist, U. and P. Porras. “Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST).” *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[Liepens92] Liepens, G. and H. Vaccaroo. “Intrusion Detection: Its Role and Validation.” *Computer & Security* 11 (1992) 347-355.

[Lunt89] Lunt, T., R. Jaganathan, R. Lee, A. Whitehurst and S. Listgarten. “Knowledge-Based Intrusion Detection.” *Proceedings of the 1989 AI Systems in Government Conference*, March 1989.

[Lunt93a] Lunt, T.F. “Detecting Intruders in Computer Systems.” *1993 Conference on Auditing and Computer Technology*, 1993.

[Lunt93b] Lunt, T.F. “A Survey of Intrusion Detection Techniques.” *Computers & Security* 12 (1993) 405-418.

[Mukherjee94] Mukherjee, B., L.T. Heberlein and K.N. Levitt. “Network Intrusion Detection.” *IEEE Network*, May/June 1994, 26-41.

[Porras92] Porras, P.A. and R.A. Kemmerer. “Penetration State Transition Analysis: A Rule-Based Intrusion Detection Approach.” *Proceedings of the Eighth Annual Computer Security Applications Conference*, December 1992.

[Porras97] Porras, P.A. and P.G. Neumann. “EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances.” *19th National Information System Security Conference (NISSC)*, 1997, http://www.csl.sri.com/emerald/emerald-niss97.html.

[Safford93] Safford, D.R., D.L. Schalem, and D.K. Hess. “The TAMU Security Package: An Outgoing Response to Internet Intruders in an Academic Environment.” *Proceedings of the Fourth USENIX Security Symposium*, 1993.

[Sebring88] Sebring, M.M., E. Shellhouse, M. Hanna and R. Whitehurst. “Expert Systems in Intrusion Detection: A Case Study.” *Proceedings of the 11th National Computer Security Conference*, October 1988.

[Sieken99a] Sielken, R. “Application Intrusion Detection.” *University of Virginia Computer Science Technical Report CS-99-17*, June 1999.

[Sieken99b] Sielken, R. and A. Jones “Application Intrusion Detection Systems: The Next Step.” *ACM Transactions on Information and System Security*, Submitted 1999.

[Skardhamar96] Skardhamar, R. *Virus: Detection and Elimination*. AP Professional, 1996.

[Smaha88] Smaha, S.E. "Haystack: An Intrusion Detection System." *Fourth Aerospace Computer Security Applications Conference*, December 1988.

[Smaha94] Smaha, S.E. and J. Winslow. "Misuse Detection Tools." *Computer Security Journal* 10.1 (1994) 39-49.

[Snapp91] Snapp, S.R., 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." *Proceedings of the 14th National Computer Security Conference*, October 1991.

[Staniford-Chen96] Staniford-Chen, S., 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." *20th National Information System Security Conference (NISSC)*, October 1996, http://olympus.cs.ucdavis.edu/arpa/grids/nissc96.ps.

[Sundaram96] Sundaram, A. "An Introduction to Intrusion Detection." *ACM Crossroads* 2.4 (1996), http://www.acm.org/crossroads/xrds2-4/intrus.html.