A Two-Layer Approach to Survivability of Networked Computing Systems

A.W. Krings, W.S. Harrison, N. Hanebutte, C. Taylor, and M. McQueen

Abstract— **An approach to system survivability is presented that uses low level attack recognition and high level migratory autonomous agents as a mechanism for recognition, early warning, and response to malicious attacks. Attack recognition is based on attack signatures at the kernel level. Signature analysis triggers agents capable of propagating attack information and facilitating reactionary measures. The approach is intended to augment existing intrusion detection systems.**

Keywords— **Survivability, autonomous agents, attack recognition, system profiling, intrusion detection, attack signature.**

I. Introduction

With the ever increasing number of networked computers on the Internet, the number of incidents involving malicious attacks on individual systems has increased dramatically. However, this does not seem to stop us from using computers in almost every aspect of life. Applications range from managing personal finances, thereby storing sensitive personal data on the computer, to critical applications where malicious manipulations might have severe consequences.

Attack software is readily available from countless web sites and can be downloaded with minimal effort even by novices. Furthermore, many books have been published on how hackers break into systems [18], [36]. However, information systems engineers do not seem to be learning from these documented security attacks [2].

A typical attack sequence may consist of probing the target system for vulnerabilities using vulnerability analyzers, e.g. *nmap*, *satan* or *saint*, and finding the software that attacks the vulnerabilities exposed. In general, the effects of attacks are ranging from benign activity, e.g. port sweeps, to Denial of Service (DoS), which may be coordinated to Distributed Denial of Service (DDoS) attacks [1], [5]. The latter is often started from computers that have been overtaken by malicious hackers.

It is crucial to recognize whether a computer has been compromised or is under attack. This has been the focus of much research in intrusion detection (ID) [35] and has resulted in a variety of intrusion detection systems (IDS), including projects like EMERALD [32] or NetStat, a successor of NSTAT [31]. However, the objective of IDS is usually recognition and not reactionary measures, recovery or survivability.

Survivability is the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents [10]. Intrusion detection or attack recognition are only specific tasks in a greater picture based on specific properties. It is usually not the entire system that is considered when addressing survivability, but certain critical functionalities or critical requirements for security, reliability, real-time responsiveness, and correctness [33].

Survivability has recently been recognized as an area of immense importance and criticality, due to the possibly staggering cost and consequences of malicious attacks on the nations resources and infrastructure [8]. It has also received national attention though documents such as the 1997 report of the President's Commission on Critical Infrastructure Protection (PCCIP) [28]. General network information system survivability at a high level has been studied extensively, e.g. in [3], [10], [12], [15], [22], [26], [29], [30]. However, survivability covers a wide spectrum of issues at different levels of abstraction [33] and applications [23].

Within a survivable system several key properties can be differentiated [10]. At the lowest level one finds *Resistance*, which implements strategies for repelling attacks. Resistance alone has proven to not be enough in order to secure a system. We therefore should assume that attacks will succeed. *Recognition* of an attack, and perhaps the assessment of the magnitude of damage, is the next logical step. In order to gain survivability of essential services after an attack, one has to consider *Recovery*, which implies restoration of essential services. Resistance, recognition and recovery have been termed the "three R's" [10]. Lastly, *Adaptation* addresses strategies for improving survivability based on an evolving process considering knowledge gained from the intrusions. The research presented here focuses on the latter three key properties, i.e. recognition, recovery and adaptation.

System survivability is usually defined with respect to *critical functionalities*. In [11] a top-town survivable network analysis method was presented that considered critical functionalities and derived a survivability map. This map considered intrusion scenarios, their resistance, recognition and recovery strategies at a high level. The survivability map indicates how to augment current strategies with recommendations that should improve survivability.

A different approach was taken in [20] where it was assumed that intrusions would happen sooner or later with possible access to root or superuser status. Critical functionalities were spatially distributed and operating system features were modified in order to mask the possible damage inflicted by an attacker.

The research presented here attempts to break away from high level survivability mechanisms and describes a general two-layer approach to survivability of networked computer systems. We investigate the feasibility of a layered

approach where low level attack analysis and survivability mechanisms are augmented with high level agent based survivability features. The low layer is based on [21], and some results are partially restated to aid readability. Section II describes the motivation and gives background information. Section III introduces the Survivability Architecture, methods used, implementation details, and results. Finally, Section IV concludes the paper with a summary.

II. Motivation and Background

A. Off-line and On-line Survivability

Survivability of a system can be seen from two different viewpoints. One can look at the architecture as the result of an off-line design process, or one can focus on on-line, real-time protective capabilities. The off-line approach focuses on designing an architecture based on general information of attack scenarios and protection of critical functionalities specified a priori. An example of such strategy is the one presented in [11], which introduced survivability maps. On-line survivability takes advantage of information gathered at run-time, which allows the architecture to adapt itself to new attack scenarios or changes in the overall computer and network environment.

Off-line and on-line approaches may have the same general goal, trying to adapt to attack scenarios. Whereas in the first case this is done in an iterative off-line design process, the second case is more flexible with respect to general real-time adaptation capabilities. However, there is a distinct difference in the operating environment of the two design strategies, the off-line approach can take advantage of a controlled operation environment, whereas the on-line approach operates within a nondeterministic run-time dependent operation environment. Therefore, the advantage of the off-line approach is the preciseness induced by the observability and determinism of the controlled environment, yet it lags the fast feedback loop of the on-line approach. Alternatively, the difficulty of the on-line approach is that, in order to work properly, one has to be able to clearly recognize normal from abnormal behavior in a very dynamic and noisy environment. This however essentially is the highest goal of anomaly detection.

The survivability architecture presented here represents a hybrid design, encompassing characteristics from both strategies. Specifically, the low layer of the architecture implements an off-line design process, whereas the high layer of the architecture is aided on-line by autonomous agents capable of adaptability and real-time change management.

B. Attack Recognition and Attack Signatures

As indicated above, recognition is at the lowest level of our considerations, i.e. it is the second of the "three R's". Recognition is based on the detection of suspicious or abnormal behavior. Two basic strategies are used to analyze both known and unknown attacks. The first is based on attack signature detection, the second considers anomaly detection. Ideally, attack recognition based on attack signatures requires a priori knowledge of *all* attack scenarios, whereas anomaly detection requires full knowledge of the expected behavior of the system to detect all attacks. Neither of these strategies can achieve complete realistic detection coverage [1].

Attack signatures are often also called intrusion signatures. Intrusion signatures can be specified as a sequence of events and conditions that lead to a break-in [24]. However, there is no consensus on this definition as signatures can contain different information, including or excluding the notion of time or event sequencing. Different interpretations exist, ranging from audit trail based signatures for use in expert systems [24], [34], to state oriented attack signatures [37]. Many projects or products use signatures, but, especially in commercial ID systems, due to the proprietary nature, it is often not clear what the signatures constitute, e.g. *RealSecure* (Internet Security Systems) or *NetRanger* (WheelGroup Inc.). In general, the lack of accessibility to signatures leaves a general desire to support the growing interest in open-source signatures [1].

In the off-line design process our view of attack recognition is based on the recognition of low-level signatures. We are not trying to determine abnormal behavior in the general sense, but to look at very specific attack signatures within the behavior of the system. Whereas this approach reduces the flexibility with respect to detecting unknown attacks, it provides information that can be used in the off-line design process in order to harden the system.

Hardening a system should not be confused with standard maintenance of a software system, e.g. patch or revision management. Software updates are a very common practice, often presenting fixes to known bugs or vulnerabilities. We assume that common sense is applied and focus our attention to inherently difficult problems, or unavoidable scenarios. Typical examples of difficult problems are DDoS attacks or attacks on resources from within a system, i.e. after a successful attack. An example of an unavoidable scenario is the probing of the system by a vulnerability analyzer, e.g. *saint*.

In general, signatures serve multiple purposes. Firstly, in the off-line design process they represent a mechanism that aids in the identification of critical functionalities in the operating systems. These functionalities are then subjected to fault-tolerant hardening techniques. Secondly, in the on-line architecture, they serve as real-time attack recognition indicators that may trigger survivability actions using agents.

C. Levels of Complexity

Signatures also can be defined at different levels of complexity. When signatures contain specific sequences of system events involved in malicious actions, one often speaks of traces. It is important to realize at which level of complexity signatures are formed. Most research and projects involving signatures has been based on discrete events that are derived from system log files. These data are however at a relatively high level of abstraction, since log files record only specific audit events. At a lower level of complexity one can derive signatures of system calls. Such an approach

has been taken in [14], [39].

We are interested in kernel level signatures. Kernel based attack detection is positioned at the lowest level of observation, just above hardware assisted detection. The benefits of kernel based intrusion detection or attack recognition have only been discussed in the literature in the recent past. Some kernel based intrusion detection systems are based on wrappers [19], or on dynamic profiling [9]. Our research is based on signatures derived from kernel level functional profiling.

Figure 1 displays the relationship of signatures and their analysis at different levels of abstraction. Kernel based signatures are at the lowest level that can be controlled by software. This level is considered in [9] and [21]. System call logs allow profiling, and thus signature generation, at much coarser granularity and are considered in [14] and [39]. At the highest level are signatures derived from log file manipulation. Most IDS research is based on log file analysis [1]. The level of abstraction of the signatures is inverse proportional to the potential real-time feasibility of recognition and reaction to attacks.

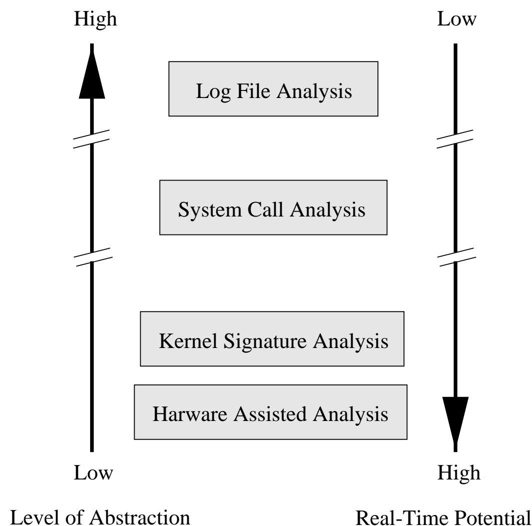


Fig. 1. Levels of Abstraction and Real-Time Potential

D. Clean Measurements

With respect to the attack signature generation process, we are attempting to turn the attack recognition problem into a basic dynamic measuring problem. At the heart of our approach to attack recognition is the process of clean measurement. We want to measure the impact of specific attack scenarios on the system as accurately as possible off-line. Once the measurements are taken and "cleaned up" they can constitute a frame of reference in on-line attack recognition. In order to identify when a system behavior is "abnormal", many IDSs attempt to establish what the "normal" system should look like. In [9] a normal system was considered as the activities of the kernel under four distinct application environments. The baseline profile was then established from these exercises. Attack recognition was based on the detection of (1) decreased activities of baseline profiles in conjunction with (2) increased activities of baseline profiles caused by an attack.

We can demonstrate that one can achieve attack recognition, independent of application environments, based on the quality and analysis of the attack signatures alone. The baseline of our investigations is the measurement of an idle system in its purest form.

We model our measurement procedure on measurements of physical phenomena. First, we focus our attention on the network portion of the target system, a Linux based host. Next, in order to reduce system noise, we conduct all measurements off-line, in an isolated network, eliminating any unrelated network activity. Specifically, we look at two connected Linux based systems in isolation. One system serves as an attacker, the other as the attack target. Lastly, there are no applications executing, i.e. the systems are running as console (without x-windows support). Now, signatures were extracted from an absolutely idle system as it was subjected to very narrowly defined attacks.

Whereas the signature generation process above is invoked off-line and in a clean environment, attack recognition is an on-line process, i.e. it is done in real-time. An off-line analysis determines the usefulness of the signature in identifying critical functions, and investigates their relationship to other, similar, attack signatures. It is only during run-time, in the on-line process, that we attempt to recognize the attack signatures in the noisy overall profile of the networked system. It is very important that the reader be clear about the differences of these two cases.

E. Two Layer Model

The survivability architecture presented here consists of two layers. At the low layer we focus on attack signatures, their generation and analysis. This layer is partly situated in the operating system kernel. Upon detection of possible attacks, a high layer agent system is invoked. This agent system represents the executive and reactionary component of the overall architecture.

Survivability may be defined with respect to an application, host, network or multi-network [1]. The low layer of our architecture, i.e. the attack signature related components, are implemented at the host level. However, the approach can be directly applied to applications as well. The agent based upper layer of the architecture builds the bridge from the host level to the network or multi-network level.

Before giving some background on agents in general, we need to define the typical system this survivability architecture targets.

F. Target System

Our research focus is to apply very specific attack recognition at the lowest level. As such, we implement host based attack recognition, and our target is a single networked computer operating in a *standard user environment.* We view such an environment as a typical desktop computer, operated mostly by single individuals. Given the affordability of desktop computers and the increasing

popularity of Linux, it is our experience in the academic environment that most students have fairly powerful desktop computers as their standard networked working console in private settings or university labs. This is quite different from recent environments where perhaps small numbers of users connected to the same Unix host via inexpensive xterminals. The usage of standard, "dedicated", work stations is in general very low. Most common user profiles include applications such as x-windows, browsers, email, compilers, or small web servers. However, unlike systems such as transaction systems or main servers, the actual utilizations are generally surprisingly low, as can be verified on individual systems using utilities such as `top`.

The basic system model can be seen in Figure 2. The target environment is a medium desktop computer, e.g. a Pentium III, running RedHat Linux 6.2 kernel version 2.2.16. Such a system consists of the general operating system, including file system or memory management, as well as some functionalities we may consider critical, e.g. accesses to password files or shared files. Our current interest in attack signatures is mainly focused on the network interface of the operating system, which is indicated as a separate block. The main protocol stack is TCP/IPv4.

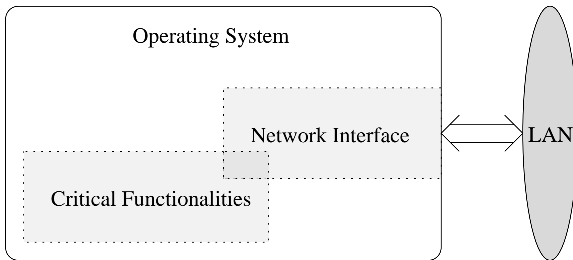


Fig. 2. System Architecture

G. Autonomous Migratory Agents

Autonomous migratory agents are an integral component in the survivability architecture described in Section III. This section gives some background for readers not familiar with agent technology.

There are many possible definitions of what constitutes a *software agent*, but most definitions tend to have similar characteristics. Bradshaw [4] defines the term agent to mean a software entity which functions continuously in a flexible and intelligent manner that is responsive to changes in the environment. Others ([27], [17]) are similar in the emphasis on *autonomy*, that is, the agent must not require constant human supervision, and must be able to assess circumstances and decide on the best course of action based on the current circumstances. Further, some argue that learning is also a requirement of agents [7], [40].

For the purposes of this research, we put a high emphasis on small, autonomous, modular agents. These agents are in general not capable of completing an entire task, but instead each can complete a sub-task and via cooperation, larger tasks can be completed by groups of agents. Our agents are also *migratory* in that they are capable of stopping execution on one machine, i.e. a host, packaging up their execution state, sending it to another machine, and resuming execution.

This model of agent programming has several advantages outlined in [17], including reduction of network load due to the fact that a mobile agent can execute directly at the source of the data. Other advantages are asynchronous execution, since the overall system will continue to execute even if one agent crashes. It can adapt dynamically, as the agents can sense and react to a new environment with every migration.

III. Survivability Architecture

Figure 3 give an overview of the survivability architecture. At the heart is the Signature Analysis engine. It is directly involved in the generation and attack analysis of the attack signatures. Signatures are collected in, and accessed from, an Attack Signature Library. At run-time, signatures are compared to the run-time system profile in an attempt to recognize attack signatures. Pending recognition, Event Handlers are called which implement the kernel based survivability mechanisms. Simultaneously, the Agent Interface selects specific agents in order to propagate reactionary survivability measures.

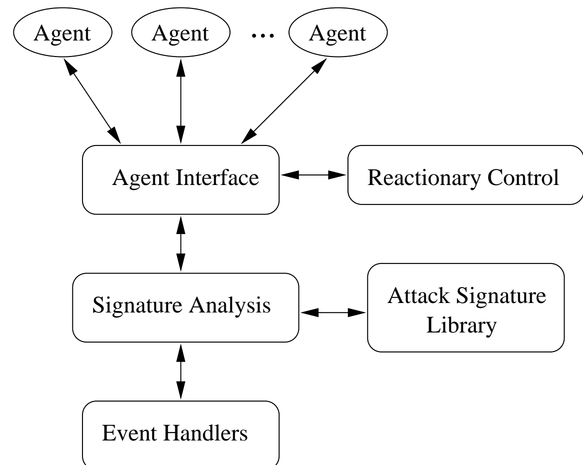


Fig. 3. Survivability Architecture Overview

A. Event Handlers, Reactionary Control

At the heart of survivability are methods to recover information or to filter information. Recovery implies that there must be sufficient information about a data set in order to recreate it in the case of loss, corruption or malicious tampering. The types of redundancy available are *time, information* and *spatial* redundancy. Time redundancy implies that a set of data is read or manipulated multiple times temporarily skewed. Whereas this is very useful for applications involving sensors, we consider it of little use in the context of operating systems. Information redundancy refers to a process in which information is added to the original data set in order to fulfill a specific objective. Typical widely used applications include

error codes and authentication. Lastly, spatial redundancy considers multiple versions of a code or data set that are spatially distributed. The results of the individual components are usually combined using voting, e.g. majority voting or agreement algorithms. In the context of this research, it is information and spatial redundancy that are the principle mechanisms for information recovery.

B. Profiles and Signatures

Similarly to [9], we view the system as a collection of functionalities. The functionalities are observed during specified time intervals Δt. Specifically, we view a system in terms of its system profile $P_{sys}(\Delta t)$, which is composed of the profiles of all functionalities $P_i(\Delta t)$ executing during Δt. Thus, for any Δt we have

$$P_{sys}(\Delta t) = \sum_{i=1}^{k} P_i(\Delta t)$$

where k is the number of functionalities active during the time interval. Each $P_i(\Delta t)$ is a vector of a length equal to the number of identities F, i.e. C functions, profiled. Therefore, if there are n identities profiled, then

$$P_i(\Delta t) = (f_1(\Delta t), f_2(\Delta t), ..., f_n(\Delta t)),$$

where $f_j(\Delta t)$, $1 \le j \le n$, is the number of times function F_j has been called (or activated) during Δt. A value of $f_j(\Delta t) = 0$ implies that function F_j has not been invoked at all, whereas $f_j(\Delta t) = x$, x a positive integer, implies that F_j has been invoked x times during Δt.

Attacks are malicious attempts to gain access to restricted resources or to gather information on a system in order to expose vulnerabilities. In practice, attacks may be mounted using individual programs or attack suites, e.g. *toast* (http://breedersec.virtualave.net/code.html). We assume attacks to be atomic. An atomic attack A_i is the smallest attack technology unit, e.g. a port sweep or a sequence of unsuccessful login attempts. Thus, an attack suite can be viewed as a collection of individual atomic attacks A_i. Limiting the scope of an attack to atomic units allows us to focus on a very narrow sets of affected functions in the OS, application and network.

Profiles during atomic attacks are of special interest and result in attack signatures, i.e. an attack signature is the portion of a profile that is attributable to the attack. Formally, the attack signature corresponding to A_i will be denoted by S_i. Only non-zero profile components are considered. Thus,

$$S_i = (f_{\alpha(1)}(\Delta t), f_{\alpha(2)}(\Delta t), ..., f_{\alpha(s_i)}(\Delta t)),$$

where α is a function that maps (one-to-one) the indices of S_i to the indices of the functions profiled. Note that s_i, the length of vector S_i, is signature dependent.

A signature of special interest is the signature of an idle system. The so-called *idle signature* is denoted by S_0. Signature S_0 corresponds to the system profile $P_{sys}(\Delta t)$ of an idle Linux system. Contrary to [9] our idle profile constitutes the system profile of a Linux system that was just booted up. No applications, e.g. x-windows, are executing.

C. Signature Analysis

Once a signature S_i has been extracted, the question arises how this signature will be recognized in the noisy profile of a running system, i.e. $P_{sys}(\Delta t)$. We limit our investigation on the effectiveness of the simplest analysis in order to minimize run-time overhead.

C.1 Ideal Signature Analysis

First, we assume an ideal scenario in which attack signatures are absolutely accurate, i.e. there is no system noise, and the time interval Δt precisely captures the entire attack. Obviously, such ideal scenario will be unachievable, however, it will serve as a frame of reference in the realistic environment discussed in Subsection III-C.2.

Let $\mathbf{S}_i$ be the set of functions F_j that are part of signature S_i, i.e. $\mathbf{S}_i = \{F_{\alpha(j)} \text{ in } S_i, 1 \le j \le s_i\}$. By the definition of a signature we have $f_{\alpha(j)} > 0$. Furthermore, let $|\mathbf{S}_i|$ denote the cardinality of $\mathbf{S}_i$. Let $\mathbf{P}_{sys}(\Delta t)$ be the set of functions F_j with non-zero frequencies, i.e. $f_j(\Delta t) > 0$, in the system profile. Thus, the number of functions represented in $\mathbf{P}_{sys}(\Delta t)$ is less than or equal to the number of functions in $P_{sys}(\Delta t)$, i.e. functions not used in Δt are excluded. Similar to $|\mathbf{S}_i|$, let $|\mathbf{P}_{sys}(\Delta t)|$ denote the cardinality of the run-time system profile. Now the following lemma, whose proof can be found in [21], can be stated:

Lemma 1: Given an ideal run-time profile $P_{sys}(\Delta t)$ and a signature S_i, if $\mathbf{P}_{sys}(\Delta t) \cap \mathbf{S}_i \ne \mathbf{S}_i$ then attack A_i cannot be present.

Lemma 1 does not consider the frequencies $f_{\alpha(j)}$ as it only considers sets. However, signatures are characterized by the implied frequencies of the functions in $\mathbf{S}_i$.

Profiles and signatures are vectors. As such, relations between them have to be considered in the context of vector operations. The difference in length of $P_{sys}(\Delta t)$ and S_i complicate matters of comparison. Therefore we introduce a function $\mathcal{T}_i(X)$, that translates a profile X into a profile of length s_i such that only the functions in S_i are reflected. Thus, $P = \mathcal{T}_i(P_{sys}(\Delta t))$ generates a profile P that is of the same length as S_i, whose elements reflect the frequencies of the same functions, i.e. the functions of S_i.

We use standard vector relation definitions: given two vectors $x = (x_1, ..., x_m)$ and $y = (y_1, ..., y_m)$, $x \ge y$ only if $x_i \ge y_i$ for all i, $x < y$ only if $x_i < y_i$ for all i, $1 \le i \le m$, and $x \ne y$ otherwise.

Before stating the lemma that considers the relationship of profiles and signatures we want to define two terms. A *false positive* is an event incorrectly identified by the IDS as an attack when none has occurred. Similarly, in the presence of an attack, failure to recognize the attack constitutes a *false negative* [1].

Lemma 2: Given an ideal run-time profile $P_{sys}(\Delta t)$ and a signature S_i, let $P = \mathcal{T}_i(P_{sys}(\Delta t))$. Then

a) if $P \ge S_i$ then A_i possible
b) if $P \ne S_i$ then A_i not possible
c) if $P < S_i$ then A_i not possible

The proof of the lemma is presented in [21].

C.2 Realistic Signature Analysis

The "no noise" assumptions above may be unrealistic. The quality of the attack signature depends on the quality of the process that generates S_i. Attack signatures are generated in the system environment described previously by repeating the measuring process numerous times and applying filtering. The signature can be verified against the call graph of the system.

The assumption about a perfect Δt during run-time profiling is unrealistic. Implications of Δt on $P_{sys}(\Delta t)$ need to be considered. One could set Δt to the duration determined during the measurement of S_i, and use a sliding window approach to determine $P_{sys}(\Delta t)$ in order to avoid the attack falling between two time intervals. This approach works under low system utilization. However, it is possible that system resources become saturated, e.g. the load of the system is high, resulting in functionalities to take longer than under low system load. Such behavior has direct implications to cases b) and c) of Lemma 2. The effect is that Δt needs to be adjusted to compensate for the slowdown. Such adjustments can be made under consideration of system load indicators, e.g. `uptime`.

In a realistic scenario that allows resource saturation one has to consider the implications of selecting an appropriate Δt. Define an array of random variables X_i, $1 \leq i \leq s_i$, that represents the invocations of F_i caused by A_i. Denote probabilities of event e with $Pr[e]$. Given a run-time profile $P_{sys}(\Delta t)$, signature S_i, and $P = \mathcal{T}_i(P_{sys}(\Delta t))$, then
a) if $P \geq S_i$, and no A_i has occurred, then a false positive has probability $\prod_{j=1}^{s_i} Pr[X_j \geq f_j^{S_i}]$,
b) if $P \neq S_i$ and A_i has occurred, then probability of a false negative is $\prod_{j=1}^{s_i} Pr[X_j < f_j^{S_i}]$, (note, that no functions with $f_j^P \geq f_j^{S_i}$ contribute to the product),
c) if $P < S_i$ and A_i has occurred, then probability of a false negative is $\prod_{j=1}^{s_i} Pr[X_j < f_j^{S_i}]$. Note that *all* functions contribute to the product.

D. Attack Signature Library

Numerous attacks have been considered, ranging from DoS attacks to scanners. Table I shows a sample of attacks that have been cataloged. The table indicates the name of the attack, the number of functions $|\mathbf{S}_i|$ in the corresponding attack signature, and the target operating system. It should be noted that, even though several of the attacks are targeting Microsoft Windows rather than Linux platforms, it is useful to consider even these signatures, since they can be part of an attack suite, e.g. *toast*. Table I furthermore gives a brief attack description followed by a classification, and notes on defenses.

The classification column indicates whether the attack is a scanner, constitutes a DoS, targets the consumption of bandwidth, lets the host crash, e.g. lock up or reboot, or causes the CPU to overload. Protocol flaws refer to flaws in a networking protocol, where a certain sequence of valid events aids an intrusion.

An attack signature library has been compiled, representing a collection of individual attack signatures S_i. An example of such S_i is presented in Figure 4, where a repetitive "teardrop" attack sequence is observed over time. The x-axis represents the number of functions invoked, in this case 58. The y-axis indicates the number of seconds the attack was observed, which was about 14 seconds. Finally, the z-axis shows the call frequencies of the functions, with maximum observed function frequency of 200. Each individual attack invocation can be seen by a "mountain range" in the three-dimensional space.

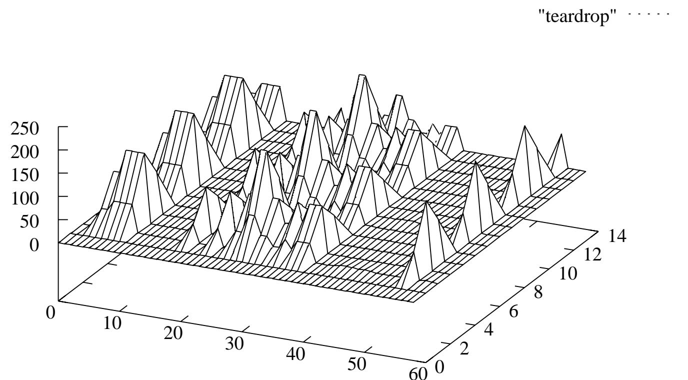


Fig. 4. Three Dimensional Teardrop Attack

The three-dimensional representation of the attack is condensed and run though a filter to give the single attack signature shown in Figure 5. With respect to Figure 4, the attack signature in Figure 5 is a rearranged "slice" of a representative "mountain range" after noise reduction. The x-axis indicates the function IDs; roughly 4000 functions have been instrumented.

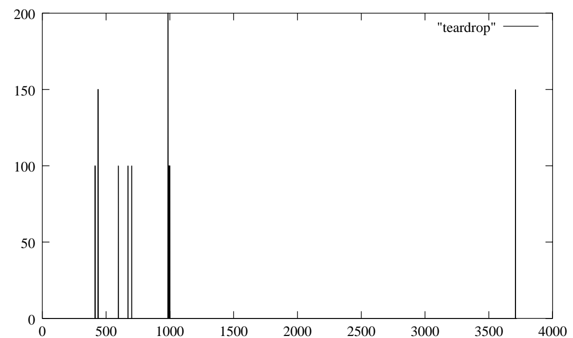


Fig. 5. Teardrop Attack

Other examples of attack signatures are shown in Figure 6 and Figure 7, which represent the "bonk" and "gewse" attack respectively. The reader should note the similarity between the signatures given in Figure 5 and Figure 6 based on visual inspection. On the other hand, the signature in Figure 7 looks less similar. In order to see how related the signatures are, with respect to the functions they share, we investigated their correlation.

Given two signatures S_i and S_j, $1 \leq i, j \leq a$, where a is the total number of attacks considered, we define correla-

TABLE I
SAMPLE ATTACKS

Attack	$\vert \mathbf{S}_i \vert$	OS	Attack Description	Classification	Defense
nmap	85	any	scan for vulnerabilities and OS setup with option "light"	scanner	secure your information
saint	200	any	scan for vulnerabilities and OS setup with option "-sS"	scanner	secure your information
telnetfp	236	any	scan for OS using telnet protocol	scanner	router/firewall
smurf	26	any	drown target host with multiplied ICMP packets	DoS; bandwidth; protocol flaw	filter; router config.
land	37	WIN; older UNIX	target gets SYN flags with its IP as src and dst	DoS; crash; exploit	filter packet with src IP = dest IP
teardrop	22	WIN; older UNIX	overlapping IP(TCP) fragments are formatted so that the reassembly crashes	DoS; crash; exploit	filter for malformed header
newtear	23	WIN	same as Teardrop, but using UPD	DoS; crash; exploit	filter for malformed header
bonk	24	WIN	malformed IP header leads to reassembly of to big packets	DoS; crash; exploit	filter; header checking
jolt	19	WIN	sends fragmented ICMP packets that can't be put together	DoS; crash; exploit	filter; header checking
nestea	23	older or precompiled Linux	exploits "off by one IP header" bug in refragmentation routine	DoS; crash; exploit	filter for malformed header
conseal	14	ConSeal Firewall	flood	DoS; crash; exploit	get update
DCD3C	25	NetQuake protocol for WIN	spoofs connect requests to all servers; host gets flooded with responses	DoS; bandwidth; protocol flaw	see smurf
fawx	19	WIN	oversized IGMP fragments	DoS; crash; exploit	filter; header check; disconnect
gewse	173	UNIX	floods identd on port 139	DoS; crash; protocol flaw	filter and router config.
ssping	19	WIN	aids sniffing through performing source routing; even if it's disabled	exploit for sniffing	filter for headers
syndrop	22	Linux pre kernel 2.0.34	IP fragment overlap results in passing of unwrapped parameters	DoS; crash; exploit	filter for header
hiperbomb	239	3Com HiperARC	sends specially formatted high volume of IACs	DoS; crash; exploit	get upgrade; filter for pattern
misfrag	14	WIN	IGMP TH_SYN and TH_ACK to every port; target IGMP fragmentation assembly routine	DoS; crash; exploit	filter for packets
trash	19	WIN	random ICMP error packets are sent as flood	DoS; CPU ; protocol flaw	filter; router config.
pepsi	46	UNIX	UPD floods to port 7,9,13, 19 and 113	DoS; CPU; protocol flaw	filter; router config.

tion $C(i,j)$ as

$$C(i,j) = \frac{|\mathbf{S}_i \cap \mathbf{S}_j|}{min(|\mathbf{S}_i|, |\mathbf{S}_j|)}.$$

Figure 8 illustrates the correlation between a sample set of signatures. The attacks A_i are listed from left-to-right and top-to-bottom, sorted by their size $|\mathbf{S}_i|$. The numbers indicate the correlation among functions. Note, only correlation of functions, *not* frequencies, is shown. Especially among the attacks with small signatures, a very high correlation can be observed.

	conseal	misfrag	fawx	jolt	ssping	trash	syndrop	teardrop	nestea	newtear	bonk	dcd3c	smurf	land	pepsi	nmap	gewse	saintl	telnetfp	hiperbomb
conseal	1.00	.85	.85	.85	.85	.85	.85	.85	.85	.85	.85	.85	.85	.78	.92	.85	.85	.85	.85	.85
misfrag	.85	1.00	.85	.85	.85	.85	.85	.85	.85	.85	.85	.92	.92	.71	.85	1.00	1.00	1.00	1.00	1.00
fawx	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.63	.63	.52	.63	.63	.63	.63	.63	.63
jolt	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.63	.63	.52	.63	.63	.63	.63	.63	.63
ssping	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.63	.63	.52	.63	.63	.63	.63	.63	.63
trash	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.63	.63	.52	.63	.63	.63	.63	.63	.63
syndrop	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.54	.54	.45	.54	.54	.54	.54	.54	.54
teardrop	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.54	.54	.45	.54	.54	.54	.54	.54	.54
nestea	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	.95	.95	.52	.52	.43	.52	.52	.52	.52	.52	.52
newtear	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	.95	1.00	1.00	.52	.52	.43	.56	.56	.56	.56	.56	.56
bonk	.85	.85	1.00	1.00	1.00	1.00	1.00	1.00	.95	1.00	1.00	.50	.50	.41	.54	.54	.54	.54	.54	.54
dcd3c	.85	.92	.63	.63	.63	.63	.54	.54	.52	.52	.50	1.00	.56	.64	.56	.52	.52	.56	.52	.52
smurf	.85	.92	.63	.63	.63	.63	.54	.54	.52	.52	.50	.56	1.00	.65	.53	.69	.69	.73	.69	.69
land	.78	.71	.52	.52	.52	.52	.45	.45	.43	.43	.41	.64	.65	1.00	.27	.40	.40	.40	.40	.40
pepsi	.92	.85	.63	.63	.63	.63	.54	.54	.52	.56	.54	.56	.53	.27	1.00	.65	.47	.73	.56	.56
nmap	.85	1.00	.63	.63	.63	.63	.54	.54	.52	.56	.54	.52	.69	.40	.65	1.00	.76	.91	.87	.88
gewse	.85	1.00	.63	.63	.63	.63	.54	.54	.52	.56	.54	.52	.69	.40	.47	.76	1.00	.97	.98	.97
saintl	.85	1.00	.63	.63	.63	.63	.54	.54	.52	.56	.54	.56	.73	.40	.73	.91	.97	1.00	.94	.93
telnetfp	.85	1.00	.63	.63	.63	.63	.54	.54	.52	.56	.54	.52	.69	.40	.56	.87	.98	.94	1.00	.99
hiperbomb	.85	1.00	.63	.63	.63	.63	.54	.54	.52	.56	.54	.52	.69	.40	.56	.88	.97	.93	.99	1.00

Fig. 8. Signature Correlation

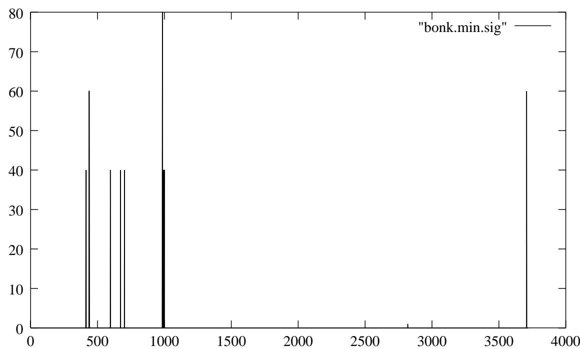


Fig. 6. Bonk Attack

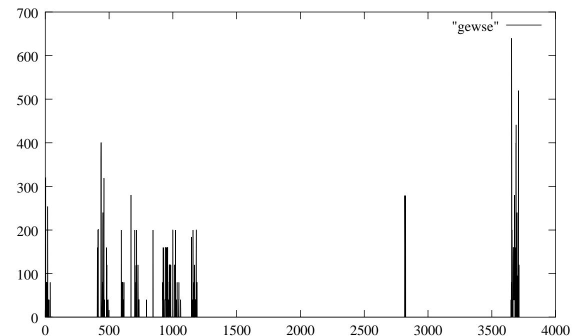


Fig. 7. Gewse Attack

E. Agent System

The agent system lies at the heart of the high layer of the survivability architecture, for it is with agents that these survivable features are implemented. This should not be confused with the survivability handlers, which facilitate survivability features at the kernel level. In general, agents can take one of three roles; they can patch an existing system, they can filter out DoS attacks, or they can simply warn other machines of the potential for a future attack.

E.1 The Aglet Agent System

The agent migration system chosen for this architecture was *Aglets* [25]. Aglets, a Java-based migration system developed by IBM Research Laboratory Japan, allows for a *coarse-grained* migration. Coarse-grained refers to the notion that the entire state of the migratory agent is not preserved. Instead, the agent program restarts after a migration. The values of variables, however, are preserved. Thus it is possible to design migratory agents in a state-machine fashion, using variables to checkpoint when key tasks are complete. Although there are other Java-based migration systems available, e.g. D'Agents [13] and Mole [38], Aglets gives a combination of asynchronous message passing, stability, and ease of programming, which make it the most desirable migration system for this architecture.

E.2 Agent Specification

Figure 9 gives an overview of our specific agent architecture. There have been several refinements to the larger overview presented in Figure 3.

E.2.a Signature Engine. The Signature Engine is the component of the survivability architecture which conducts real-time comparisons between the stored Attack Signature Library and the Kernel Instrumentation. Upon detecting an attack in progress, using the methods discussed in Section III, it passes the attack information, e.g. attack likelihood and attack type, to the Local Agent Interface. The attack likelihood refers to the probability of the attack ac-

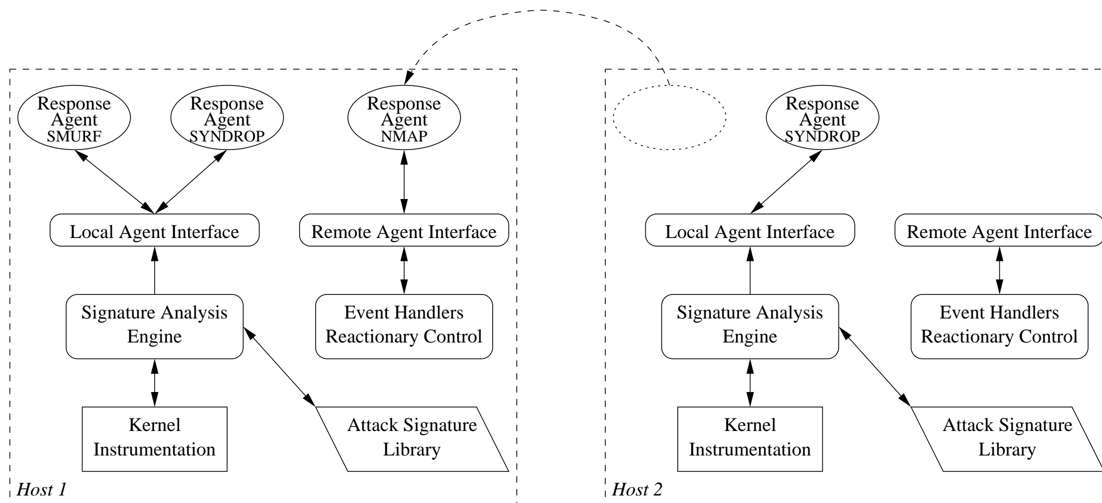


Fig. 9. Survivability Architecture

cording to Lemma 2.

E.2.b Local Agent Interface. The Local Agent Interface is the component responsible for communicating between the Signature Engine and the Response Agents discussed below. Given the information regarding the attack from the Signature Engine, this component will make a decision whether to respond to an attack or not. For example, an "unlikely port sweep" is not likely to warrant much response, whereas a "definite smurf attack" requires a quick response. If it chooses to respond to the attack, it directs one of several response agents to apply appropriate action. The Local Agent Interface, although it does not migrate, is capable of doing so.

E.2.c Response Agents. The Response Agents apply high level survivability features. They are individual, migratory agents, which respond to attacks. Each type of Response Agent has a unique response for an attack. For example, a smurf attack [6] might require a router to turn off packet forwarding, whereas a port sweep might simply require warning other hosts in the network that there is a potential future attack. A response to a particular observed exploit might be to apply a patch.

E.2.d Remote Agent Interface. Although Response Agents are created on one system, i.e. the so-called *home* host, in many circumstances, it is desirable for them to migrate to another host to take specific action. For example, a DoS attack might require the router to take filtering action. However, a long-standing issue with respect to migratory agents deals with the security and trustworthiness of these agents. It has been shown that migratory agents are themselves vulnerable to different types of attacks, including attacks wherein a hostile host examines or even modifies the contents of a migratory agent as it travels through that host [16].

For purposes of our experiment, we assume a secure network in which agents can be trusted. This however is quite limiting. Therefore, in order to facilitate the assumption we do not allow migratory agents any direct control over a host. Specifically, the responsibility of security is shifted to the Remote Agent Interface. When an agent migrates to another machine to prescribe corrective action, this action cannot be implemented by the agent itself. Instead, the Remote Agent Interface communicates with the Response Agent, and based on the trustworthiness of that remote agent, allows different levels of action to take place. This might apply the prescribed action, such as modifying a routing table. However, this could be limited to allow only a subset of the prescribed action to take place, e.g. making a backup of the password file, but not modifying the existing one. In the extreme case it may not allow any action to take place at all.

F. *Case Study: Smurf*

In order to describe the overall function and agent interaction of the survivability architecture, we will consider a DDoS attack, specifically a smurf attack.

F.1 Smurf Attack

A smurf attack is a DDoS attack which exploits a vulnerability in the TCP/IP protocol. This attack involves an attacker sending ICMP echo packets to generate multiple replies. The attacker forges the source address on the packet to the victim's address and, typically, sends these requests to a broadcast address. All machines within the broadcast subnet, also called the *amplifier* or *amplifier network*, then respond, deluging the victim with unwanted ICMP echo replies. There are few ways to defend against such an attack. One option is to have the local router filter all packets from outside the victim's network. This essentially shuts off the source of the attack, assuming the responses were generated external to the router. If the attack is from inside the local network this action will have no effect.

F.2 Smurf Attack Sequence

Assume an experiment where the victim of the smurf attack is a Linux machine running RedHat 6.2, and the amplifier network consists of three similarly configured machines. The victim and the amplifiers are separated by a router, also a Linux RedHat 6.2 machine.

F.2.a Before the Attack. The Signature Engine is running and compares the cataloged signatures to the current system profile every second. The experiments in [21] have determined $\Delta(t) = 1s$ to be a reasonable choice for a time interval between consecutive profiles. First the Signature Engine attempts to recognize $\mathbf{S}_i$, the set of C functions of S_i, in the set $\mathbf{P}_{sys}(\Delta t)$ of the current system profile $P_{sys}(\Delta t)$. If a match is detected, the frequency values of the profile are compared to the signatures, i.e. S_i and $P_{sys}(\Delta t)$ are compared. The possible attack is then categorized according to the criteria of Lemma 2. However, rather than assuming the ideal cases, i.e. $P \geq S_i$, $P \neq S_i$ and $P < S_i$, we have experimented with relaxing the relations using different functions. The resulting scenarios are termed *probable* for $P \geq S_i$, and *unlikely* for $P \neq S_i$ and $P \geq S_i$.

When an attack becomes probable, the Signature Engine informs the Local Agent Interface of the attack by writing to a named pipe. The message itself is a long integer, the bits of which correspond to the specific attack number. Currently, only "probable" attacks are reported to the Local Agent Interface and "likely" and "impossible" ones are not considered.

The Local Agent Interface, when started, is blocking on a read from this named pipe. When it does read a value, it declares the attack underway and determines if there is a response agent for that attack present on the system. It does this by connecting to the local context. This can be visualized as the local environment in which all agents on the same machine exist. Next, it examining the names of the Response Agents on the system. The agent which responds to the smurf attack is named "Survive.SMURF". If this agent does not exist for any reason, the Local Agent Interface refrains from action and simply listens again on the named pipe. If the Response Agent does exist on the system, the Agent Interface sends it a message that the home host is under attack, and returns to listening on the pipe.

The Response Agent for smurf is designed to migrate to the router and update the tables to filter the amplifier network out. When the Response Agents are created, they are waiting for a message from the Local Agent Interface. Until a message arrives, they are suspended causing no overhead.

When the Smurf Response Agent gets a message from the Local Interface Agent, the agent responds to the message and migrates to the router. Once at the router, it contacts the Remote Agent Interface and requests that the amplifier network be blocked. In the test environment, the Remote Agent Interface trusts the agent and changes the routing table to cut off the amplifier network. Once this is done, the Smurf Response Agent returns to the home host, and goes back to sleep. By this time, the attack has been terminated, as the amplifier network has been cut off from the victim by the router.

After the Response Agent leaves, but before the attack terminates, the Signature Engine will continually report the smurf attack, as well as any new attacks, to the Local Agent Interface. If the Smurf Response Agent has been dispatched and is no longer local, the Response Agent will not attempt to send a message to it.

G. False Positives and Negatives

As stated in [1], it is very difficult, if not impossible to give accurate percentages for false positives or negatives. Our testing assumed running the Attack Detection System under nominal system usage, including applications such as x-windows, telnet sessions, ftp sessions, and web page retrieval to and from the target machine.

In the current state of the project, we have only limited data on false negatives. However, so far we have not been able to produce them under the application mix assumed.

With respect to false positives, we have not succeeded in generating pathological scenarios for non-trivial attacks. However, we have reported false positives for attacks with very small numbers of functions, e.g. "misfrag", which were the result of benign activity. Furthermore, misclassification has been observed among highly correlated attacks indicated in Figure 8. This gives rise to investigating refinements in the classification scheme.

We recognize the false positive and negative data is still in a preliminary state. Only extensive field testing will reveal the actual percentages.

IV. Conclusions

This paper introduced a two-layer approach to computer and network survivability. The architecture consists of an off-line and an on-line component, facilitating an iterative design process.

At the low layer, attack signatures were generated in a controlled environment, which aided in the identification of critical functionalities. Survivability handlers were applied at the kernel level. The signature analysis also triggered response mechanisms at the high layer, implemented by an agent system.

The current research focus is on the expansion of the signature attack library, as well as improved reactionary mechanisms under consideration of security concerns of migratory agents. Whereas this architecture addresses survivability, it does not claim to give comprehensive intrusion detection, and is intended to be augmented by a general IDS.

References

[1] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel and E. Stoner, *State of the Practice of Intrusion Detection Technologies*, Carnegie Mellon, SEI, Technical Report, CMU/SEI-99-TR-028, ESC-99-028, January 2000.

[2] W. Arbaugh, W. Fithen, and J. McHugh, *Windows of Vulnerability: A Case Study Analysis*, IEEE Computer, Vol. 33, No. 12, Dec. 2000.

[3] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag and P. Uppuluri, *Building Survivable Systems: An Integrated Approach Based on Intrusion Detection and Damage Containment*, Proc. of the DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. II of II, Hilton Head Island, South Carolina, January 2000.

[4] J. Bradshaw, *An Introduction to Software Agents*, Software Agents, Ch. 1, pp. 3-46, AAAI/MIT Press, 1997.

[5] CERT Coordination Center, Carnegie Mellon, SEI, http://www.cert.org/advisories.

[6] CERT Coordination Center, *Smurf IP Denial-of-Service Attacks*, Advisory CA-1998-01, January 1998, http://www.cert.org/advisories/CA-1998-01.html

[7] M. Crosbie and E. Spafford, *Defending a Computer System using Autonomous Agents*, Technical Report CSD-TR-95-022, The COAST Project, Purdue University, 1994.

[8] Computer Security Institute, *Computer Security Issues and Trends*, 4, 1, Winter 1998.

[9] S. Elbaum, and J. Munson, *Intrusion Detection Through Dynamic Software Measurement*, Proceedings of the Eighth USENIX Security Symposium, 1999.

[10] E. Ellison, L. Linger, and M. Longstaff, *Survivable Network Systems: An Emerging Discipline*, Technical Report CMU/SEI-97-TR-013, 1997.

[11] E. Linger and M. Longstaff, *A Case Study in Survivable Network System Analysis*, Technical Report CMU/SEI-98-TR-014, 1998.

[12] D. Fisher, *Emergent Algorithms: A New Method for Enhancing Survivability in Unbounded Systems*, IEEE Proceedings of the Hawaii International Conference on Systems Sciences, Jan. 5-7, 1999, New York: IEEE Computer Society Press, 1999.

[13] R. Gray, *Agent Tcl: A Flexible and Secure Moble-Agent System*, Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96), pp. 9-23, 1996.

[14] S. Hofmeyr and S. Forrest, *Intrusion Detection using Sequences of System Calls*, Journal of Computer Security, Vol. 6, pp. 151-180, 1998.

[15] J. Howard, *An Analysis of Security Incidents on the Internet (1989- 1995)*, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1995.

[16] W. Jansen and T. Karygiannis, *Mobile Agent Security*, NIST Special Publication 800-19, National Institute of Standards and Technology, Computer Security Division, 1999.

[17] W. Jansen, P. Mell, T. Karygiannis, and D. Marks, *Mobile Agents in Intrusion Detection and Response*, 12th Annual Canadian Information Technology Symposium, Ottowa, Canada, June 2000.

[18] L. Klander and E.J. Renehan, Jr., *Hacker Proof: The Ultimate Guide to Network Security*, Delmar Publishers, 1997.

[19] K. Calvin, T. Fraser, L. Badger, and D. Kilpatrick, *Detecting and Countering System Intrusions Using Software Wrappers*, Proceedings of the Ninth USENIX Security Symposium, 2000.

[20] A. Krings and M. McQueen, *A Byzantine Resilient Approach to Network Security*, Proc. 29th International Symposium on Fault-Tolerant Computing, Fast Abstracts: (FTCS-29), Madison, Wisconsin, June 15-18, pp. 13-14, 1999.

[21] A. Krings, S. Harrison, N. Hanebutte, C. Taylor, and M. McQueen, *Attack Recognition Based on Kernel Attack Signatures*, to appear in Proc.: 2001 International Symposium on Information Systems and Engineering, (ISE'2001), Las Vegas, June 25-28, 2001.

[22] J. Knight, R. Lubinsky, J. McHugh and S. Kevin, *Architectural Approaches to Information Survivability*, http://www.cs.virginia.edu/ survive/publications.html

[23] J. Knight, et.al, *Topics in Survivable Systems*, University of Virginia, Computer Science Report, No. CS-98-22, August 1998.

[24] S. Kumar, and E. Spafford, *An Application of Pattern Matching in Intrusion Detection*, Technical Report CSD-TR-94-013, The COAST Project, Purdue University, 1994.

[25] D. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.

[26] R. Linger, N Mead and H. Lipson, *Requirements Definition for Survivable Network Systems*, Proceedings of the International Conference on Requirements Engineering, Colorado Springs, CO: April 6-10, 1998, New York, IEEE Computer Society Press.

[27] P. Maes, *Modeling Adaptive Autonomous Agents*, Artificial Life, Vol. 1, No. 1/2, Ed: Christopher Langton, MIT Press, 1993.

[28] T. Marsh (ed), *Critical Foundations: Protecting America's Infrastructures*, Technical report, President's Commission on Critical Infrastructure Protection, October 1997.

[29] D. Medhi and D. Tipper, *Multi-Layered Network Models, Analysis, Architecture, Framework and Implementation: An Overview*, Proc. of the DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. I of II, Hilton Head Island, South Carolina, January 2000.

[30] S. Moitra and S. Konda, *A Simulation Model for Managing Survivability of Networked Information Systems*, technical report, Carnigie Mellon, Software Engineering Institute, CMU/SEI-2000-TR-020, December 2000.

[31] R. Kemmerer, *NSTAT: A Model-Based Real-Time Network Intrusion Detection System*, (RTCS97-18), November 1997, http://www.cs.ucsb.edu/~kemm/netstat.html/documents.html.

[32] P. Neumann and P. Porras, *Experience with EMERALD to DATE*, Proc. 1st USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California, pp. 73-80, 1999.

[33] P. Neumann, *Practical Architectures for Survivable Systems and Networks*, (Phase-Two Final Report), Computer Science Laboratory, SRI International, June 2000.

[34] P. Porras and P. Neumann, *EMERALD: Event Monitoring Enabling Response to Anomalous Live Disturbances*, 1997 National Information Systems Security Conference, 1997, http://www.sdl.sri.com/emerald/emerald-niss97.html.

[35] Purdue University, *Coast intrusion detection*, http://www.cerias.purdue.edu/coast/intrusion-detection/ids.html.

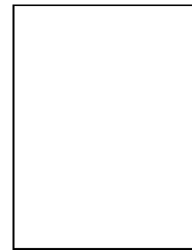
[36] J. Scambray, S. McClure, and G. Kurtz, *Hacking Exposed: Network Security Secrets & Solutions (Second Edition)*, McGraw-Hill, 2001.

[37] M. Sobirey, B. Richter, and H. Konig, *The intrusion detection system AID. Architecture, and Experiences in Automated audit Analysis*, Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security, pp. 278-290, September 1996.

[38] M. Strasser, J. Baumann, and F. Hohl, *Mole, A Java Based Mobile Agent System* Special Issues in Object Oriented Programming, pp. 391-307, Springer Verlag, 1997

[39] C. Warrender, S. Forrest and B. Pearlmutter, *Detecting Intrusions Using System Calls: Alternative Data Models*, 1999 IEEE Symposium on Security and Privacy, pp. 133-145, 1999.

[40] M. Wooldridge, *Intelligent Agents*, Multiagent Systems, Ed: Gerhard Weiss, pp. 1-27, MIT Press, 1999



Axel W. Krings received the Dipl.Ing. in Electrical Engineering from the FH-Aachen, Germany, in 1982 and his M.S. and Ph.D. degrees in Computer Science from the University of Nebraska - Lincoln, in 1991 and 1993, respectively. He is an associate professor of Computer Science and Computer Engineering at the University of Idaho and a member of the Center for Secure and Dependable Software (CSDS), the Initiative for Bioinformatics and Evolutionary Studies (IBEST), and the Microelectronics Research Center (MRC). His research interests include Survivability, Fault-Tolerance Systems, Scheduling Theory, Parallel and Distributed Systems, Data Communication and Real-Time Systems.

William S. Harrison received a B.A. in Computer Science from Southeastern Louisiana University in 1995 and a Ph.D in Computer Science from Tulane University in 1999. He is currently an Assistant Professor at the University of Idaho and is a member of the Center for Secure and Dependable Software (CSDS), the Initiative for Bioinformatics and Evolutionary Studies (IBEST), and the Center for Intelligent Systems Research (CISR). His research interests include Survivability, Agent Systems, and Artificial Intelligence.

Nadine Hanebutte received her Dipl.-Inf. (M.S.) degree in Computer Science from the Otto-von-Guericke-University, Magdeburg, Germany, in 1999. She is currently pursuing her Ph.D. degree in Computer Science at the University of Idaho, Moscow, USA. During 1999 she was a software engineer in the Risk Control Department of the HypoVereinsbank in Munich, Germany. Her research interests include software engineering, software metrics and system survivability.

Carol S. Taylor received her B.S. in Computer Science from the Colorado State University in 1985. From 1985 to 1995 she worked as a consultant, programmer and systems analysts for different companies and government agencies. She is currently pursuing her Ph.D. degree in Computer Science at the University of Idaho, Moscow, USA, where she is involved in research in the applicability of statistical methods in network security and software metrics.

Miles A. McQueen received a B.A. in Mathematics and a B.A. in Economics from UCLA in 1977, and his M.S. degree in Computer Science from the California State University at Northridge in 1980. Currently he is an advisory engineer at the Idaho National Engineering and Environmental Laboratory working with the Information Protection and Security group. He is also an affiliated faculty member of the Computer Science Department of the University of Idaho. His research interests include Concurrent Systems, Survivability, Real-Time Systems, and Scheduling Algorithms.