On the Performance of a Survivability Architecture for Networked Computing Systems

W.S. Harrison, A.W. Krings, N. Hanebutte
University of Idaho
{*harrison,krings,hane*}*@cs.uidaho.edu*

M. McQueen
INEEL
amm@inel.gov

Abstract— **This research focusses on the performance and timing behavior of a two level survivability architecture. The lower level of the architecture involves attack analysis based on kernel attack signatures and survivability handlers. Higher level survivability mechanisms are implemented using migratory autonomous agents. The potential for fast response to, and recovery from, malicious attacks is the main motivation to implement attack detection and survivability mechanisms at the kernel level. A timing analysis is presented that suggests the real-time feasibility of the two level approach. The limits to real-time response are identified from the host and network point of view. The experimental data derived is important for risk management and analysis in the presence of malicious network and computer attacks.**

I. Introduction

The number of networked computers on the Internet, as well as the number of applications involving the Internet, has increased constantly over the last few years. Unfortunately, the number of incidents involving malicious attacks on computer systems has increased dramatically as well [5]. These attacks on computers and networks have resulted in huge financial losses, time, and exposure of confidential data [8].

One does not have to be an expert in order to attack systems. A typical attack sequence consists of probing a system for vulnerabilities using vulnerability analyzers, e.g. *nmap*, *satan* or *saint*, and finding the software that attacks the exposed vulnerabilities. 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], [38]. In general, the effects of attacks range from benign activity, e.g. port sweeps, to harmful Denial of Service (DoS), which may be coordinated to create Distributed Denial of Service (DDoS) attacks [1], [5]. The latter is often started from computers that have been compromised by malicious hackers.

In order to detect when a system has been compromised, various intrusion detection systems (IDS) have been created as the result of extensive research in intrusion detection (ID) [37]. Examples of IDSs are EMERALD [34] or NetStat, a successor of NSTAT [33].

The objective of an IDS is usually recognition and not reactionary measures, for recovery or survivability. *Survivability*, on the other hand, is the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents [10]. 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 [35].

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) [30]. General network information system survivability at a high level has been studied extensively, e.g. in [3], [10], [12], [15], [24], [28], [31], [32]. Conceptually, however, survivability covers a wide spectrum of issues at many different levels of abstraction [35] and applications [25].

Survivability is strongly related to the topic of ultra-reliable system design. However, the main focus of ultra-reliable systems design is usually not malicious faults. Instead, hybrid fault models are employed to account for different fault behavior, e.g. benign, symmetric or asymmetric behavior [41]. Here benign faults are globally verifiable faults, symmetric faults indicate that everybody receives the *same* faulty result, whereas asymmetric faults have no restriction on the behavior. Malicious attacks can therefore be categorized as asymmetric faults, in that they usually exercise pathological behavior. It should be noted that in fault-tolerant system design it is usually assumed that asymmetric failure of components is statistically unlikely, or it may be attempted to eliminate these fault types altogether [45]. Contrary to the statistical assumptions of fault-tolerant system design, in computer and network security and survivability it is fair to assume that the most likely scenarios caused by malicious hacker will exhibit pathological behavior, e.g. asymmetric behavior.

Unfortunately, achieving tolerance to diverse faults comes at a high cost. For example, special purpose ultra reliable system architectures such as SIFT or MAFT [20] exhibit runtime overhead costs of up to 80%. However, today's commercial applications cannot tolerate such overhead to achieve tolerance to failure or malicious attacks.

Besides run-time overhead, issues of fault propagation, containment and recovery time are of concern. In general, one can assert that solutions that are implemented at higher levels of abstraction may have inverse effects on fault propagation, containment and recovery. For example, if an IDS indicates that an intrusion has occurred based on

the analysis of system log files, much time may have passed since the actual intrusion. As a result, the attack may have caused extensive damage and recovery may degress to complete re-installations of system and application software.

The general awareness about the criticality of real-time attack detection and recovery, is the main motivation for employing survivability features at the lowest level of abstraction. The survivability architecture discussed here considers the lowest level, i.e. at the kernel function level. Kernel based attack recognition has the potential to optimally address issues of fault propagation and damage containment.

The research presented here describes timing issues of a general two-layer approach to survivability of networked computer systems. We investigate the real-time feasibility and analyze the timing behavior 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 the model presented in [22]. Details about the entire survivability architecture can be found in [23]. Section II describes the motivation and gives background information. Section III describes the Survivability Architecture from [23]. Section IV describes the methods used, implementation details, and results. Finally, Section V concludes the paper with a summary.

II. Overview of Survivability Architecture

This section presents an overview of the two level survivability architecture tailored to performance analysis. A full description of the architecture can be found in [23]. Some details are partially restated to aid readability and overall understanding.

In order to justify the survivability architecture it is important to understand the target environment. The target is a single networked computer operating in a *standard user environment* [11]. We view such an environment as a typical desktop computer, operated mostly by single individuals running standard applications. 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 previous environments where, for example, small numbers of users connect to the same Unix host via inexpensive xterminals. The usage of standard, "dedicated", workstations 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 lower level of the architecture implements kernel-level instrumentation. This process involves monitoring the usage of functions within the kernel itself. This is the lowest possible level in which instrumentation can be inserted without resorting to hardware-level solutions. Instrumentation at this level has the capability to be very low-overhead. This makes it an ideal candidate for a real-time survivability architecture.

At the higher layer, autonomous migratory agents are an integral component of the survivability architecture. There are several 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 ([29], [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. 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. The 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.

III. Survivability Architecture

Figure 1 gives an overview of the survivability architecture. At the heart of the architecture is the Signature Analysis engine. It is directly involved in the generation and 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.

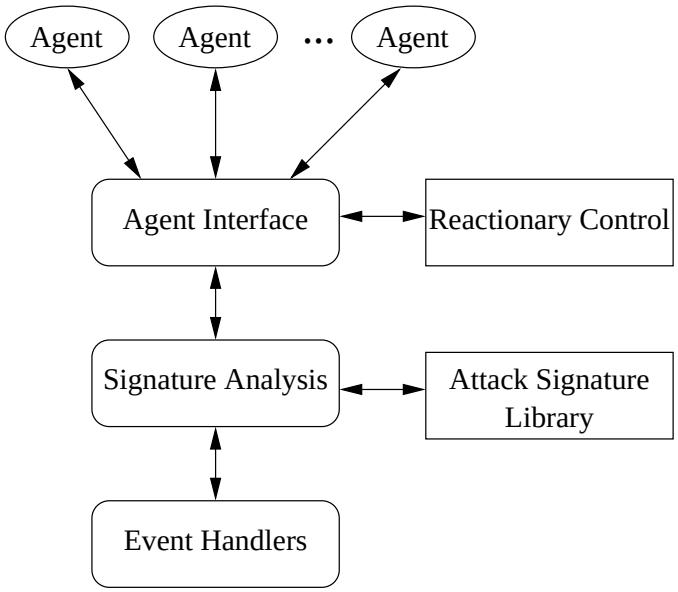

Fig. 1. Survivability Architecture Overview

A. Profiles and Signatures

The attack detection component of the survivability architecture is based on the manipulation of 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 $f_j(\Delta t)$, $1 \leq j \leq n$, is the number of times function F_j has been invoked 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.

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. The set of functions of S_i is denoted by $\mathbf{S}_i$, and the cardinality of $\mathbf{S}_i$ is denoted by $|\mathbf{S}_i|$. Thus, given S_i, $|\mathbf{S}_i| = s_i$.

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.

Attack signatures S_i are collected a priori in an *Attack Signature Library* by running atomic attacks against the idle system. The attacks currently considered range from DoS attacks to scanners. It seems obvious that the size of the library should will impact the attack recognition overhead. However, as will be shown in Subsection IV-B, the analysis overhead may be constant in the absence of attacks.

B. High Level Agent Architecture

Whereas the signature model described above is situated at the low level, i.e. at the kernel level, the high level survivability features are implemented using agents. The agent migration system chosen for this architecture was *Aglets* [27], a Java-based migration system developed by IBM Research Laboratory Japan.

Figure 2 gives an overview of the specific agent architecture.

B.0.a Signature Analysis Engine. The Signature Analysis 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 [23], 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 being underway.

B.0.b Local Agent Interface. The Local Agent Interface is the component responsible for communicating between the Signature Analysis 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, a "port sweep" is not likely to warrant much response, whereas a "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.

B.0.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 the specific attack it is defending against. 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 specific observed exploit might be to apply a patch.

B.0.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 additional filtering action. However, a long-standing issue with respect to migratory agents deals with the security and trustworthiness of these agents [16]. This issue is addressed in the current architecture by not allowing migratory agents any direct control over a host. This is the function of the Remote Agent Interface. It evaluates the trustworthiness of the agent and takes action on the local machine if deemed necessary. Response agents themselves have no capability to control a host.

IV. Performance and Measurement Analysis

A. Attack Description

For purposes of measuring overhead, two isolated networks of machines connected by a router were used. Fig-

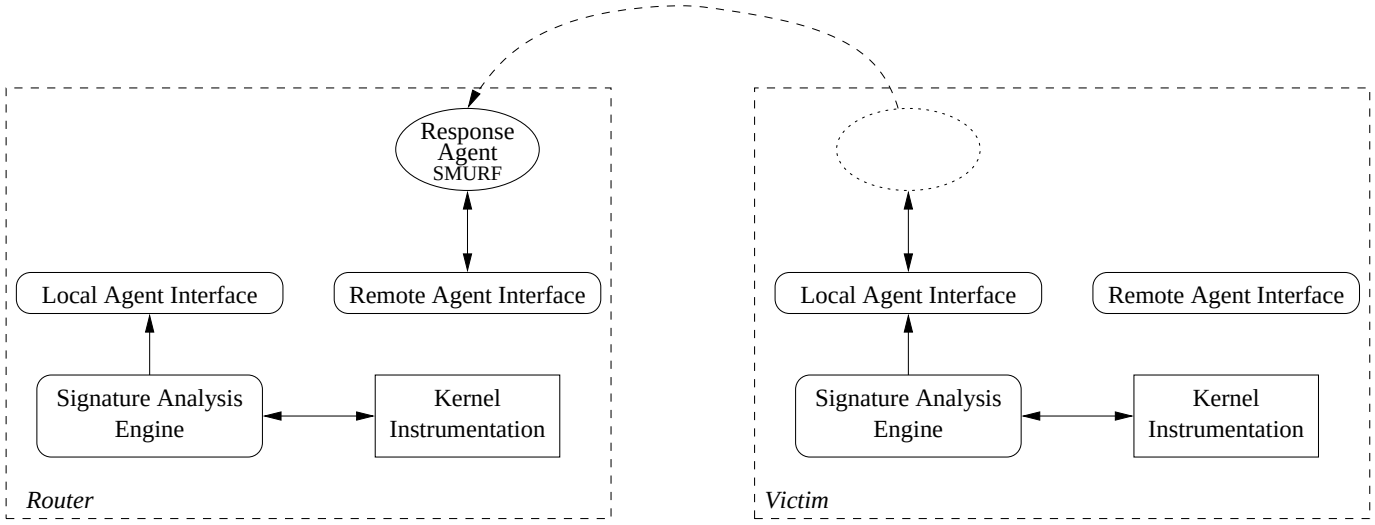

Fig. 2. Survivability Architecture

ure 3 shows the setup of the attack network.

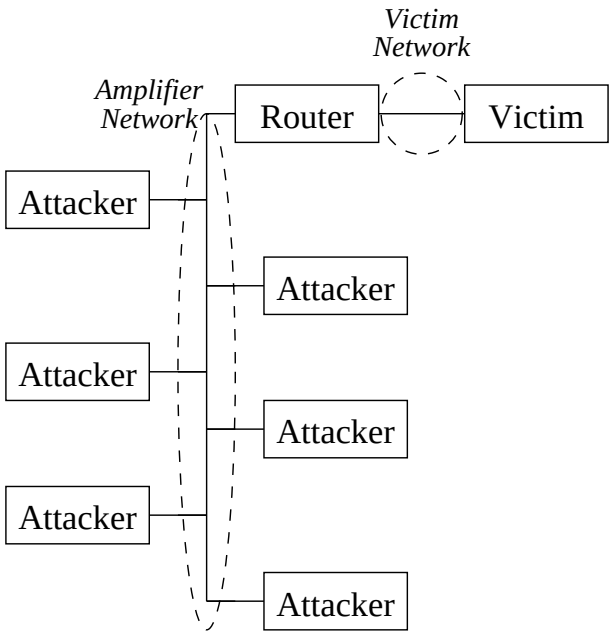

Fig. 3. Attack Network Setup

The victim network contained only the victim machine and a router. The victim machine was attacked by a smurf attack. This would trigger a response agent which would migrate to the router and contact a Remote Interface agent on the router. The Remote Interface agent would acknowledge the request and turn off packet forwarding from the amplifier network to the victim machine. The response agent would then migrate back to the source machine and sleep. More details on the smurf attack can be found in [6].

B. Overhead Identification

There are basically two different types of overhead associated with the survivability architecture. Firstly, the low-level instrumentation constitutes overhead in terms of code which is instrumented in the operating system. This adds additional code which must be executed and therefore will degrade system performance. Secondly, the different components of the survivability architecture produce runtime overhead at different levels of abstraction. It is crucial to analyse the performance cost of each architectural component in order to identify bottlenecks and critical events.

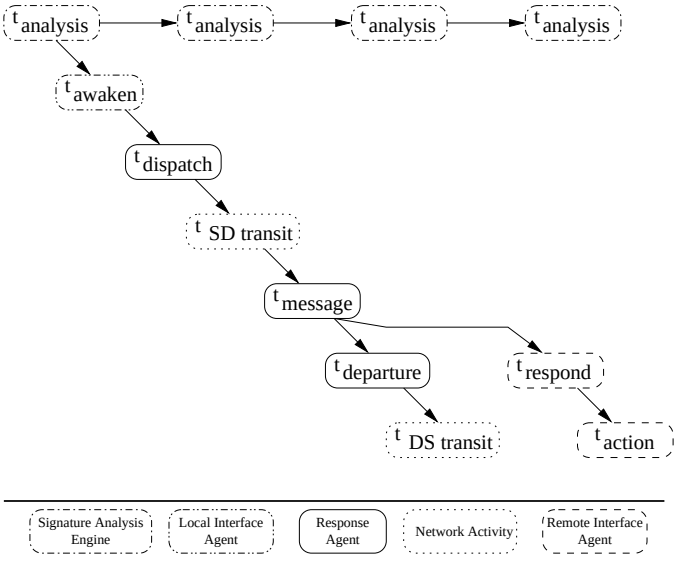

Fig. 4. Event Sequence Overview

Figure 4 presents the event chain of a typical attack and the specific architectural component that is executed.

1. The system profile $P_{sys}(\Delta t)$ is read and analyzed. The profiling interval is fixed to $\Delta t = 1s$ in the current implementation. At each Δt a new profile is read, thus overlapping the profile generation with the analysis and responses.
2. The time consumed during the signature analysis is denoted by $t_{analysis}$.

3. Conditioned on the outcome of the analysis an appropriate agent needs to be woken up, which takes time t_{awaken}.
4. The time it takes to dispatch the agent, i.e. the time from awaking to leaving the local computer, is denoted by $t_{dispatch}$.
5. Next, the agent transit time from the source to the destination, $t_{SDtransit}$ is considered.

On the destination computer, the following timing behavior is observed:

1. The time spent by the Response Agent between arrival at the router and sending a message is denoted by $t_{message}$.
2. The Response Agent then returns to its source computer. The time consumed between sending the message to the Remote Agent Interface and leaving the computer is $t_{departure}$.
3. The time it takes the Remote Agent Interface to receive the message from the Response Agent is $t_{respond}$. This time is independent of $t_{departure}$.
4. The time it takes the Remote Agent Interface to take appropriate action is denoted t_{action}.
5. The destination to source transit of the Response Agent is denoted by $t_{DStransit}$. It should be noted that $t_{DStransit}$ may differ greatly from $t_{SDtransit}$, since now the survivability action has been finished, e.g. a packet filter has been enabled in response to a DoS attack.

There are several different types of overhead to be considered.

1. *Instrumentation Overhead* consists of *code overhead.* This is strictly the number of lines added to the kernel itself. This code will be present in the kernel source code as well as change the size of a compiled kernel. This is not necessarily indicative of kernel performance; lines of code can be added to the kernel which are never called. Such code creates no overhead in terms of time.
2. *Profiling Overhead* is the overhead needed to generate $P_{sys}(\Delta t)$. This is the slowdown of the kernel due to the instrumentation. This overhead can be calculated by comparing execution times in an instrumented kernel to an non-instrumented kernel. Any slowdown between the two is due to the profiling overhead.
3. *Signature analysis* is the overhead used to compare $P_{sys}(\Delta t)$ to all signatures S_i. This overhead is added to the profiling overhead and both together must be less that Δt. If the sum of this combined overhead is greater than Δt, signatures cannot be effectively compared at runtime. The performance of the Signature Analysis Engine is not constant. When the signature analysis engine is started, all signatures are analyzed and a master set (called a **VIP** set) containing the functionalities contained in all signatures is generated i.e. $\mathbf{VIP} = \bigcup\{\mathbf{S}_j\}|1 \leq j \leq a$.
At runtime, a check is made to see if any non-zero elements of the $P_{sys}(\Delta t)$ correspond to a functionality in **VIP**. If there are all elements of $P_{sys}(\Delta t)$ corresponding to the functions in **VIP** are zero, then no known attack can be taking place. This means no further vector comparisons are needed. Thus, in the absence of attack, the the efficiency of the signature analysis engine is constant: $O(1)$, as only one set of vector comparisons is needed.
If, however, some elements of **VIP** correspond to non-zero elements of $P_{sys}(\Delta t)$, an attack may be taking place. Thus, $P_{sys}(\Delta t)$ must be compared against every signature. As the number of signatures increases, the efficiency of the current implementation is $O(k)$ where k is the number of signatures stored. This could be reduced by improving the comparison algorithm used. For example, a comparison mechanism using binary trees would result in efficiency $O(log_2 k)$.
4. *Agent Execution Overhead* is the time it takes an agent to do its task, for example, to awaken when a message is received, or to cut off an attack by engaging a filter.
5. *Agent Communication Overhead* is due to agents communicating with one another. This is dependent on the mechanism used, but, in general, is quite low. It varies slightly according to machine load.
6. *Transit Overhead* is the overhead due to a response agent migrating from one machine to the other. It is primarily dependent on two factors, the size of the response agent itself and the network traffic.

C. Experimental Setup

In the laboratory, a total of eight machines were used. The machines of interest are the victim machine and the router. The victim was a Pentium 3/850 MHz machine with 512 MB of memory. The router machine was the type of low-end machine typically employed in this role. In this case, the router was a Pentium 75 with 48 MB of memory and two Ethernet cards. The amplifier machines varied from 1 GHz Pentium 4 machines to Pentium 75 machines. All machines were running RedHat linux version 6.2. The machines were connected using 10MB Ethernet.

The measurements (except the instrumentation overhead and analysis measurements) were conducted under four sets of conditions:

1. *No Load/No Traffic*: No traffic was introduced into the network, and no additional load was generated on the victim machine.
2. *No Load/50% Traffic*: The load was left normal on the victim machine, but traffic was inserted into the victim network using a Hewlett-Packard Ethernet Network Analyzer Model J2522B. The traffic was rated at 50% of the network capacity.
3. *No Load/95% Traffic*: Again, no load was introduced on the victim machine, but traffic was inserted into the victim network as before, but at 95% of the network capacity. This was the maximum the Network Analyzer was capable of generating.
4. *High Load/95% Traffic*: Load was introduced on the victim machine in the form of several simultaneous mathematically oriented programs. In general, the load on the machine was approximately 7.5 (using the `top` command) although values above 8.0 could be seen at times. These numbers represent jobs in the run queue of the operating system. 1.0 is the optimal value. Further, the network was loaded as before at 95% capacity.

D. Experimental Results

D.1 Instrumentation Overhead

In the linux kernel used in this experiment, (2.2.16), the total number of lines of code in the non-instrumented network portion is 362,749. The number of lines of code in the network portion of the instrumented kernel is 372,992. Thus, the increase is not that significant (approximately 2.7%). The size of the executable kernel changed from 528,268 to 556,268 bytes, also not a significant increase (5%).

D.2 Profiling Overhead

This overhead can be considered the difference in speed between an instrumented kernel and an identical non-instrumented kernel. To determine this, the following test was run on both an instrumented and non-instrumented kernel on the same machine. The kernels were compiled using the same configuration file to ensure they were identical except for the instrumentation.

The test performed was downloading a 50MB file using the http protocol (the transfer was local). The file was downloaded over the loopback interface instead of Ethernet to avoid any network delays. The test was performed 100 times on both the instrumented and non-instrumented kernel.

The total time to execute a single iteration of the test (using the *time* command) is broken into three categories: *user time*, *system time*, and *elapsed time*. The user time is the time spent in user functions, the system time is the time spent in system (kernel) functions, and the elapsed time is the start-to-finish time of the task (including time when the task was swapped out and not executing). The average user time for both kernels was identical, 27.21 seconds. This is exactly what was expected as the only difference between the kernels is in the system function calls. The non-instrumented kernel had an average system time of 1.49 seconds versus 1.74 seconds for the instrumented version. Thus, on average, the system time increased by 14.3%. The elapsed time is relatively unpredictable since it is determined by system load and other external variables. The average elapsed time in the non-instrumented kernel was 29.56 seconds, and the average elapsed time in the instrumented kernel was 29.85 seconds. This is an increase of .971%, which is the result a user will see.

D.3 $t_{analysis}$

Several measurements were run to determine signature analysis time. Tests were conducted using signature libraries of 2, 20, 50, and 100 signatures. During each test, every signature in the library was compared to $P_{sys}(\Delta t)$ and the total comparison time was recorded. A typical result is shown in Figure 5. As can be seen, the predicted linear scaling is apparent.

The average $t_{analysis}$ varies according to the number of signatures. With 2 signatures in the library, the average is 6,640.07 microseconds. With 20 signatures, the average is 13,179.21 microseconds, with 50 the average is 24,724.02 microseconds, and with 100 signatures, the average $t_{analysis}$ is 43,000.79 microseconds. These numbers reflect linear scaling as the pure overhead of the Signature Analysis Engine is approximately 6,000 microseconds (determined by running it with no comparisons whatever).

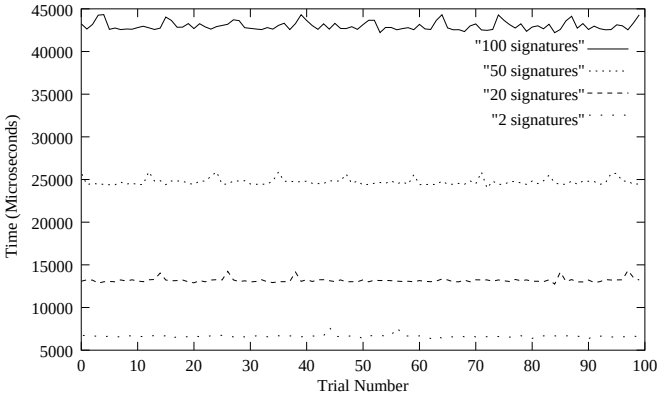

Fig. 5. Signature Analysis Scalability

As mentioned in Section IV-B, when no attack is underway, the $t_{analysis}$ should be constant. This is the case. With signature libraries of 2, 20, 50, and 100 signatures (with no attack underway), the average analysis times are 6109.88 microseconds, 6149.49 microseconds, 6097.94 microseconds, and 6064.94 microseconds, respectively. Figure 6 shows a comparison of $t_{analysis}$ with 100 signatures and no attack underway, 100 signatures with an attack underway, and 2 signatures with an attack underway. It can be seen that the analysis time for 100 signatures with no attack is less than the analysis time for 2 signatures with an attack underway.

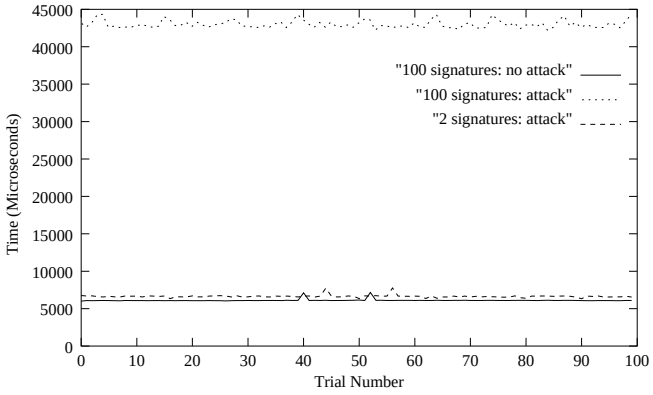

Fig. 6. Signature Analysis Comparison

D.4 t_{awaken}

t_{awaken} consists of two parts: the time it takes the Signature Analysis Engine to contact the Local Agent Interface, and the time it takes the Local Agent Interface to awaken a Response Agent (if needed). For purposes of testing, a Response Agent was always awoken by the Local Agent Interface. The Signature Analysis Engine communicates

via a FIFO pipe to the Local Agent Interface. The Local Agent Interface communicates with the Response Agent via Aglets messaging.

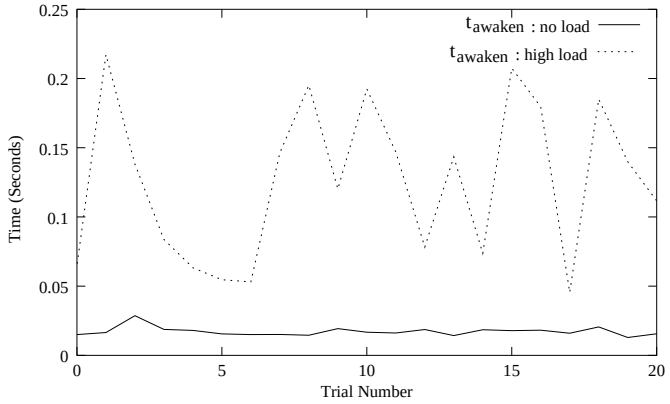

Fig. 7. t_{awaken} Comparison

Figure 7 shows t_{awaken} under both high and normal load conditions. Note the Y-Axis scale has changed to seconds. Under normal load circumstances, t_{awaken} is fairly stable, and has an average time of 0.0172 seconds, with a high of 0.0286 seconds and a low of 0.0129 seconds. The response time under high load is wildly variable, with an average of 0.126 seconds, a high of 0.2172 seconds and a low of 0.0455 seconds.

As expected, network load has no effect on t_{awaken}. The averages under different traffic conditions were: 0.0203 seconds (no load), 0.0172 seconds (50% load, plotted above), and 0.0165 seconds (95% load).

D.5 $t_{dispatch}$

$t_{dispatch}$ is the time it takes the Response Agent to leave the local machine after receiving a signal. This signal comes from the Local Agent Interface via Aglets messaging. This time is affected in the same ways as t_{awaken}. Figure 8 shows $t_{dispatch}$ under high and normal load conditions.

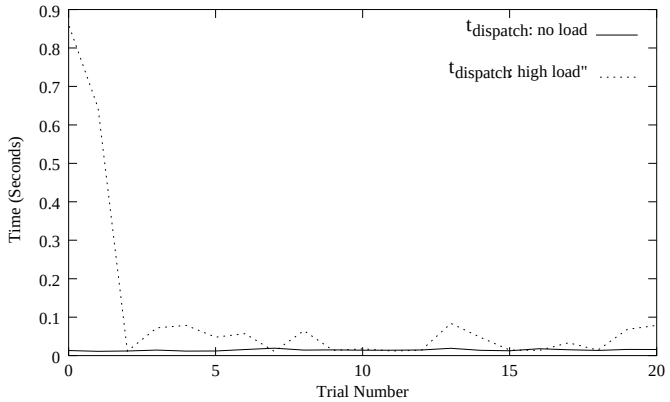

Fig. 8. $t_{dispatch}$ Comparison

As with t_{awaken}, the 0.1076 second $t_{dispatch}$ is much higher than the 0.0145 seconds under normal load. The high values for $t_{dispatch}$ under high load are extremely high, 0.8607 seconds and 0.6445 seconds. Averages under differing traffic conditions (with no machine load) were similar: 0.0159 seconds under no network load, 0.0145 seconds under 50% network load (plotted), and 0.0153 seconds under 95% network load.

D.6 Router Activity Times

Once the Response Agent arrives at the Router, it sends the Remote Interface Agent a message. This message is sent using the standard Aglets messaging facility. The router was never placed under any load, since routers are typically dedicated machines with no interactive logins. Thus, times relating to the router ($t_{message}$, $t_{departure}$, $t_{respond}$, and t_{action}) are similar in all test cases.

The average $t_{message}$ varied little under the four test conditions: 0.0719 seconds, 0.0773 seconds, 0.0706 seconds, and 0.0821 seconds with very little variance (less than 10%) between maximum and minimum times. Likewise, all values for $t_{departure}$ were similar: 0.0792 seconds, 0.0779 seconds, 0.0823 seconds, and 0.0814 seconds.

$t_{message} + t_{departure}$ comprises the total amount of time the Response Agent spent on the router. These times were 0.1604 seconds, 0.1630 seconds, 0.1579 seconds, and 0.1680 seconds.

$t_{repsond}$ reflects the elapsed time from when the Response Agent sent a message and the router responded. These will be similar to t_{awaken} as they reflect the time required for the same type of communication. There will be some difference in speed, however, as $t_{respond}$ is a single event (t_{awaken} consists of two events) and the router is a much slower machine than the victim. The test averages for $t_{respond}$ are 0.1328 seconds, 0.1314 seconds, 0.1285 seconds, and 0.1292 seconds.

Finally, t_{action} represents the elapsed time between $t_{respond}$ and when action is taken. In this case, the action is configuring the linux machine to stop packet forwarding between the two networks. This is done by writing a value in the `/proc` filesystem. $t_{respond}$ had test averages of 0.2272 seconds, 0.1710 seconds, 0.1546 seconds, and 0.2349 seconds.

D.7 $t_{SDtransit}$ and $t_{DStransit}$

Transit time constitutes the time required for a Response Agent to travel from one machine to another. No general-usage clock synchronization method is sufficient for the type of accuracy required (accurate to less than a millisecond to measure these times separately). Therefore, $t_{SDtransit}$ and $t_{DStransit}$ are discussed together.

In the experiments, network traffic was injected on the victim network. This was so that even after the attack had ceased, the network load would remain the same. This alleviates the problem of having to determine a very long $t_{SDtransit}$ and a very short $t_{DStransit}$, although this is the more realistic situation. When the attack has ceased, there will be little load on the network. The realistic situation can be simulated by taking a high traffic $t_{SDtransit}$ time

and a low-traffic $t_{DStransit}$ time. For purposes of this discussion, both of these times will be referred to as $t_{transit}$. The numbers were generated by taking the round-trip time and dividing by two.

$t_{transit}$ varies as can be expected under differing conditions of network load. Figure 9 shows the return time of the Response Agent under varying network and machine loads.

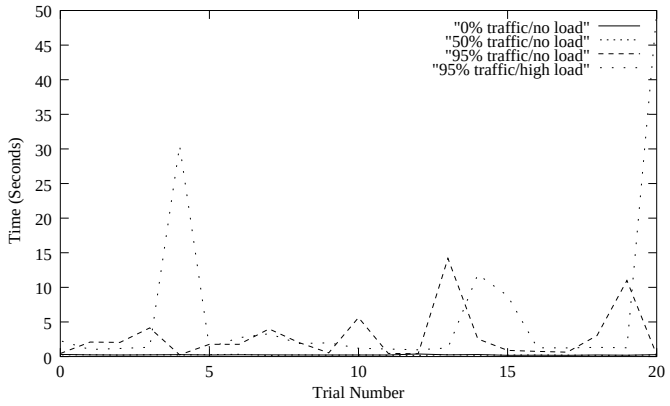

Fig. 9. $t_{transit}$ Comparison

Machine load has more of an effect than might be expected. This is because the Agent server must re-assemble the Response Agent after receiving it. It also appears that a 50% loaded network is not significant enough to add extra transit time to the Response Agent (the size of the Response Agent is approximately 3K).

With an unloaded machine and network, the average transit time was 0.2433 seconds, compared to a similar time of 0.2224 seconds with 50% network loading. With 95% network load and no machine load, $t_{transit}$ rose to an average of 2.8013 seconds. There was a large variation, from a low time of 0.2435 seconds to a peak time of 14.1884 seconds. When the computational load was introduced to the machine along with with high network loading, the average $t_{transit}$ rose to 6.0632 seconds. There was also large variation present, with a low time of 0.9696 seconds to a high time of 49.6094 seconds.

E. Discussion of Results

The results indicate good performance. Based on system call time, overall performance of the kernel is reduced by 14.3%. This degradation only occurs when instrumented functions are called, and so appears worse than it is. If no network functions are not used, there will be no performance degradation. Elapsed time is in some ways more accurate, as this is what a user sees. Using elapsed time instead of system time, overall degradation appears to be just .971%.

The Signature Analysis Engine performs at $O(1)$ in the best case, and $O(k)$ in the worst case, where k is the number of signatures being compared. This worst case could perhaps be made more efficient by using a more sophisticated comparison algorithm. However, even with 100 signatures in the library, the comparisons took very little time.

The execution times for the various agents were, overall, quite low, even under high load conditions. Although the router was not loaded, similar results as loading the victim will be seen if the router is loaded.

The true bottleneck in the system for this test is the transmit time of the migratory agent. This is due to the fact that this represents a "worst-case" situation. A DoS attack (like smurf) is specifically designed to congest the network is going to degrade overall performance greatly. This is due to the fact that migratory agents are used which must contend with the attack for network bandwidth. Even in these situations of extreme network loading (95% load), it is reasonable to assume that the attack will be responded to within one minute.

V. Conclusions

The research presented demonstrates the real-world overhead cost effectiveness of a two-layer approach to system and network survivability. The lower level presents a low-overhead method of determining whether a system is under attack, and the high level presents a higher-overhead method of dealing with different types of attacks.

The architecture presented causes little performance degradation in real-world situations as evidenced by the performance benchmarks given. In the face of realistic attack scenarios, the system performs quite well. Response time is not excessive even under extreme network load, and the approach is feasible for real-world enterprises.

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] Kieckhafer, R.M., et al, *The MAFT Architecture for Distributed Fault-Tolerance*, IEEE Transactions on Computers, V. C-37, No. 4, pp. 398-405, April, 1988.

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

[22] 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.

[23] A. Krings, S. Harrison, N. Hanebutte, C. Taylor, and M. McQueen, *A Two-Layer Approach to Survivability of Networked Computing Systems*, to appear in Proc: 2001 International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2001), L'Aquila, Italy, August 2001.

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

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

[26] 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.

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

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

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

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

[39] 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.

[40] 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

[41] P.M. Thambidurai and Y.K. Park Y.K., *Interactive Consistency with Multiple Failure Modes*, Proc. 7th Reliable Distributed Systems Symposium, Columbus, OH, pp. 93-100, Oct. 1988.

[42] 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.

[43] J.H. Wensley, et.al., *SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control*, Proceedings of the IEEE, pp. 1240-1255, Vol. 66, No. 10, October, 1978.

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

[45] Y.C. (Bob) Yeh, "Triple-Triple Redundant 777 Primary Flight Computer", *IEEE Aerospace Applications Conference*, pp. 293-307, 1996.