A Holistic Approach to Service Survivability*

Angelos D. Keromytis* Janak Parekh* Philip N. Gross* Gail Kaiser*
Vishal Misra* Jason Nieh* Dan Rubenstein† Sal Stolfo*

*Department of Computer Science †Department of Electrical Engineering
Columbia University
{*angelos,janak,phil,gail,misra,nieh,danr,sal*}*@cs.columbia.edu*

ABSTRACT

We present **SABER** (Survivability Architecture: Block, Evade, React), a proposed survivability architecture that blocks, evades and reacts to a variety of attacks by using several security and survivability mechanisms in an automated and coordinated fashion. Contrary to the *ad hoc* manner in which contemporary survivable systems are built–using isolated, independent security mechanisms such as firewalls, intrusion detection systems and software sandboxes– SABER integrates several different technologies in an attempt to provide a unified framework for responding to the wide range of attacks malicious insiders and outsiders can launch.

This coordinated multi-layer approach will be capable of defending against attacks targeted at various levels of the network stack, such as congestion-based DoS attacks, software-based DoS or code-injection attacks, and others. Our fundamental insight is that while multiple lines of defense are useful, most conventional, uncoordinated approaches fail to exploit the full range of available responses to incidents. By coordinating the response, the ability to survive successful security breaches increases substantially.

We discuss the key components of SABER, how they will be integrated together, and how we can leverage on the promising results of the individual components to improve survivability in a variety of coordinated attack scenarios. SABER is currently in the prototyping stages, with several interesting open research topics.

Categories and Subject Descriptors

C.2.0 [**Security and Protection**]: Denial of Service; D.2.0 [**Protection Mechanisms**]: Software Patching

General Terms

Reliability, Survivability, Overlay Networks, Intrusion Detection.

1. INTRODUCTION

A secure system meets or exceeds an application-specified set of security policy *requirements.* For example, in message delivery, the high-level requirements may be that the correct information gets to the right person, in the right place, *at the right time*. The details of "right" are determined by the application's needs. For example, during a crisis, the network can be used to carry communications between widely dispersed "static" sites (*e.g.*, various federal, state, and city agencies) and (semi-) roaming stations and users. Similarly, timely message delivery is crucial for battlefield or stock-trading tasks. Traditional security mechanisms have addressed the first two parts of this informal definition of security, but largely ignore the timeliness and/or service guarantee issues.

The U.S. military has embraced the concept of "getting inside the enemy's decision cycle", *i.e.*, fighting so fast that the enemy cannot organize a coherent defense. This strategy has proven extremely effective in the battlefield. The same strategy can be applied in the domain of communications, especially against organizations that increasingly rely on information sharing, such as financial firms or the US military itself. When such attacks are directed against widely-used and/or critical services or software (*e.g.*, the Internet DNS infrastructure or an information-sharing web-based service), they offer the potential for complete shutdown of the target's operational network. Thus, from an attacker's point of view, any of a number of attacks are sufficient (even if not equally effective) in gaining an advantage: network DoS attacks, software DoS attacks, code-injection attacks, *etc.*

Currently, most commercial responses to this diverse array of vulnerabilities has been to apply several discrete solutions:

- Utilization of network-based firewalls to avoid exposing services to the Internet. However, the increasing number of services whose data is piped over HTTP (either for dynamic web content or Web Services-based solutions [71]) rapidly reduces the utility of firewalls, especially against DDoS attacks, where service or pipe saturation is now not only feasible, but frequent [52].

- Deployment of network- and host-based intrusion detection systems (NIDS and HIDS, respectively). Both, however, suffer from the "chatty" problem: these services typically generate very extensive log reports of every potential attack, but typically require human inspection to see what may have been an actual attack [53]. Moreover, the human must devise and deploy a response.

- Manual installation and deployment of patches. This is a complex and tedious process on any deployed platform; modern server operating systems, for example, are shipping with hundreds of known vulnerabilities only a few months after

*Parts of this work are suported by DARPA contract No. F30602-02-2-0125 (FTN program), with additional support from Cisco and Intel Corp.

SSRS '03, October 31, 2003, Fairfax, VA, USA.

release [4, 7]; moreover, the turnaround time for patch creation and release is rapidly becoming insufficient.

All of these approaches as well as others have a more fundamental weakness: they require manual user intervention. In other words, the *self-survivability* of services is known to be weak, and human intervention is required when attacks penetrate or sometimes merely threaten existing safeguards. While this worked in older days, the pervasiveness of today's Internet solutions means that a large number of vulnerable hosts are present, from which attacks may be launched too rapidly for a human response. As the SQL Slammer worm recently demonstrated [13, 14] the response time of the average IT department was and is insufficient against the next generation of rapidly spreading exploits.

Auto-response systems are slowly becoming more pervasive; however, the majority of such approaches only solve limited subsets of the above challenges, such as automated patch installs [5]. While this helps to reduce latency, such mechanisms often have no fallback strategy, nor can they take advantage of the different tradeoffs in terms of effectiveness, performance, and availability that the different solutions offer. For example, if the information gleaned from a NIDS were to be used in an autonomic fashion, one might be able to trigger an auto-patching subsystem in faster-than-human time and repair the problem before significant damage is sustained. Should this process fail, we may be willing to sandbox the service, or migrate it to a new location.

We therefore propose a framework, SABER (Survivability Architecture: Block, Evade, React), which allows the synthesis of multiple mechanisms to not only prevent attacks, but to also maintain the survivability of services under active attack, and to do so in an autonomic fashion. By coordinating information gleaned from and actions performed between the different first-class core components, we can minimize unnecessary human intervention. We discuss how our existing components will work as security providers within SABER, and how they address these concerns.

Paper Organization. The remainder of this paper is organized as follows: first, the SABER model is presented, with each of the required constituent core components. We then describe our component implementations in more detail, followed by a section that demonstrates how these technologies can coordinate and collaborate in a variety of scenarios. Finally, we discuss future development of SABER as well as related work.

2. ARCHITECTURE

An overview of SABER is shown in Figure 1. SABER brings together and coordinates several components: it selects the most appropriate ones given the nature of the threat, state of the service and the network, and the ability of the individual components to counter the threat. Currently, SABER uses the following reaction/protection mechanisms:

- A network denial-of-service (DoS) resistant architecture [40];
- Intrusion and anomaly detection tools, placed within service contexts to detect both malicious activity as well as stealthy "scans and probes";
- A process migration system [56] that can be used to move a service to a new location that is not (currently) targeted by an attacker;
- An automated software-patching system [66] that dynamically fixes certain classes of software-based attacks, such as buffer overflows;
- A high-level coordination and control infrastructure, to correlate and coordinate the information and control flow of the aforementioned constructs.

We discuss each of these constructs in detail and motivate them in the following subsections. It is important to recognize that SABER is designed in a modular fashion, so each of these components can be adopted or discarded based upon the survivability requirements of the services being offered. Furthermore, we can integrate other components as they become available.

2.1 DoS Resistant Architecture

Denial-of-service attacks are among the most prevalent and successful form of attacks on the Internet today [52]. The fundamental problem with most services exposed on the Internet, and the reason they are vulnerable to this form of attack, is the inability to determine the difference between legitimate traffic (specific requests appropriate to the service's business) as opposed to illegitimate "junk" traffic (random requests just designed to induce load). This rarely has to do with poor coding practices (*e.g.*, buffer overflows), but rather points to the fact that the number of (high-bandwidth) clients on the Internet is scaling up faster than the capacity of providers to support a large number of them. Current approaches, including the common tactic of utilizing manual firewalling at farther-away points from the attacked network, only have had limited success and almost always require further human intervention, reducing the survivability of the service under attack.

Therefore, tools are required that can *rapidly* determine whether a service request is legitimate and directed, and therefore if a provider should consume resources processing request. In our architecture, we use a DoS-resistant network topology that pushes the "attack perimeter" into the Internet core, which is assumed to be non-DoSable, while letting legitimate traffic through to the service [40]. (If the core were to be successfully attacked in this manner, it seems unlikely that any network-based approach could avoid unavailability.) Furthermore, we combine that system with security protocols that are themselves hardened against DoS attacks [15].

2.2 Intrusion Detection Tools

While intrusion detection tools themselves cannot prevent attacks, their presence is necessary for two reasons: to help detect an attack *in progress*, as well as to determine probes of the service network, which may be a precursor sign to an impending attack [49]. SABER employs both the two most common forms of intrusion detection:

- *Surveillance detection* looks for known bad types of network traffic, an excess of which might indicate an attempt at a stealthy scan (for example, half-open TCP connections or ICMP scans).
- *Anomaly detection* uses a training approach, where "normal behavior" is learned and abnormal traffic or behavior generates an alert (such as a logged-in user executing unusual UNIX commands).

One of the known characteristics of intrusion detection is its penchant to generate many alerts, including a mix of legitimate alerts and "false positives". In the SABER architecture, the alerts are quickly coordinated with other components to help determine the legitimacy of the threat, and to help preemptively adjust behavior to match. For example, a large number of unusual TCP scans on a certain set of ports might indicate an attacker looking for vulnerabilities; the generated alerts might be communicated to the migra-

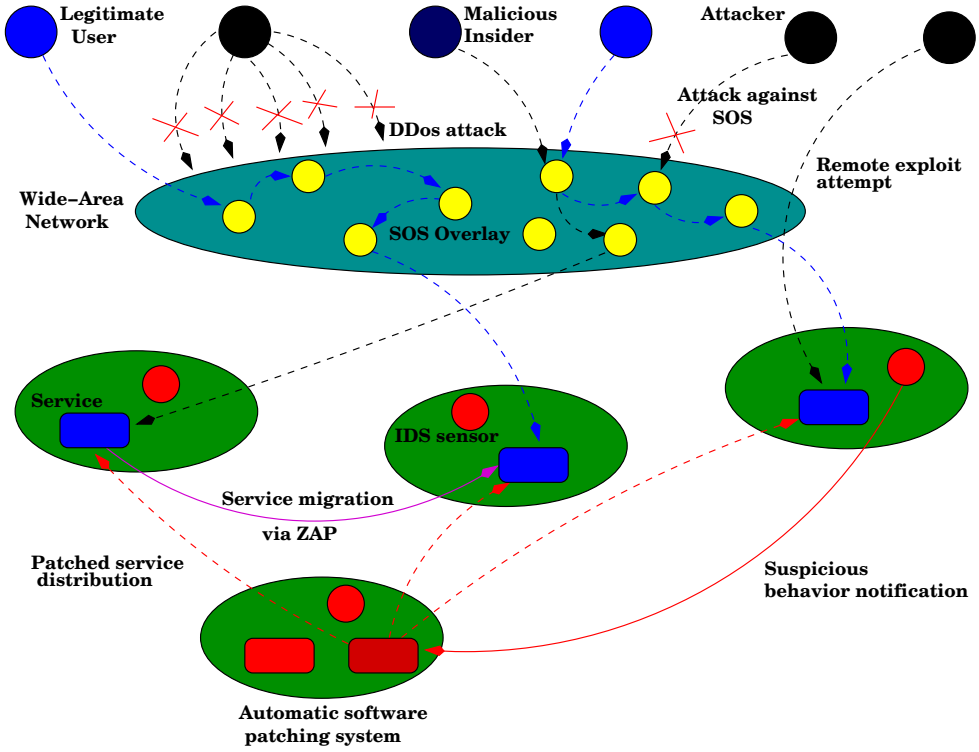

Figure 1: The SABER architecture.

tion component, to help the system respond in the most efficient way to maximize survivability.

2.3 Process Migration And Software Patching

Both process migration and software patching tools are critical in enhancing the survivability of the exposed service not only before an attack but especially during or after an attack, by either altering the process or moving it to a more secure location.

Process migration encompasses tools that automate the ability of processes to suspend state, move to another host, and to resume where computation was left off. By leveraging such tools, services under attack may either move to less vulnerable hosts or be suspended until such time as an attack has passed. Often, such migration is critical to survivability: instead of having to restart a service which may be in the middle of a (possibly distributed) computational task, we simply move it to a new location, which the surveillance/anomaly detection subsystem indicates as being less vulnerable to attackers at the moment. Naturally, the system should balance this against the possibility of attackers somehow being able to determine the (new) location of a service and cause a new migration (thereby causing an indirect DoS attack).

The goal of the automated software patching component, on the other hand, is to provide an autonomic equivalent to the manual task that most system administrators face on a daily basis: working around application vulnerabilities. While the canonical action is to wait for a manufacturer-supplied patch and to install it within a timely fashion, modern exploits and attacks leave little room for a manually-initiated response. Instead, autonomic patching aims to work around the vulnerable code by monitoring a copy of it for exploits (such as buffer overflows, whose violation of memory boundaries can be determined), and to automatically fortify or reroute around such code. This enhances survivability by deploying a solution that provides either no loss of functionality or, at worst, minimal impact, without having to wait for an administrator take down the service, patch it, and bring it up, which may also happen long after an attack has taken place. The downside of the approach is that it can only address specific types of software flaws, and that despite its high success rate (80%) [66] it is by no means a panacea.

2.4 Coordination and Control Infrastructure

The final critical component in the SABER model is infrastructure to enable the various core components to communicate and correlate efficiently and in a decentralized fashion. Without such an ability, SABER would be little more than a decentralized collection of strong security tools. But with this infrastructure, SABER can respond to attacks more rapidly, deploy solutions and enable services to maximize survivability.

We propose a distributed, network-scale publish-subscribe event-based infrastructure as the basis for communication. Events correspond naturally to low-level intrusion detection alerts and are lightweight, enabling rapid communication to other SABER components as soon as attacks are detected. We also propose an event-based correlation infrastructure that is able to track multiple inputs over time (*e.g.*, DoS and intrusion detection) and determine whether a multiple-method coordinated attack is in process, or if an attack is being carried out in stages.

In essence, this infrastructure forms a workflow component to the security capabilities developed in SABER and enables it to provide a comprehensive response with minimal latency, critical to enabling the framework to be cohesive and coordinated.

2.5 Putting It All Together

In a target service network "enhanced" by SABER, the DoS-resistant architecture is placed at the perimeter of the service net-

work, where the service itself might ordinarily be exposed to either bandwidth or computation-based DoS attacks. IDS sensors are placed in various nodes both inside the service network and within the DoS-resistant components on the perimeter. Process patching and migration services are available within the network, ready to be used on short notice based on data collected by sensors and as directed by the C2I mechanisms. These components are tied together via the publish-subscribe event-based information bus, which provides rapid, low-latency communication of alerts, attacks, repairs/changes and related security events. One last important consideration is the hardening of SABER nodes themselves; SABER is designed to "drink its own medicine" and support intrusion detection, self-migration, etc. on its own nodes within the network.

As this demonstrates, SABER remains highly modular and decentralized, thereby less vulnerable to a coordinated attack itself; *i.e.,* if the process patching component were to be bypassed, the remaining components could still function independently; and, as mentioned before, components can be added and removed depending on requirements. SABER is also designed to support "passive" operation, *e.g.,* brittle networks can choose to employ SABER in a non-autonomic fashion, where users can be involved in the process; however, SABER's technologies are truly designed to support low-latency, independent operation.

A concrete integration example is discussed in Section 4.

3. COMPONENTS

We now describe some of the components of the SABER architecture in more detail. For complete details, implementation and results, the reader is referred to the cited papers. Some of the components are in a fairly mature stage of research and development, while others are subject of ongoing research. Readers who are familiar with the material are encouraged to skip the appropriate sections and focus on Section 4, where we outline how these components work together in detail.

3.1 Secure Overlay Services (SOS)

SOS [40] addresses the problem of securing communication on top of today's existing IP infrastructure from DoS attacks, where the communication is between a pre-determined location and users, located anywhere in the wide-area network, who have authorization to communicate with that location. The focus is on protecting a site that stores information that is difficult to replicate due to security concerns or due to its dynamic nature. An example is a database that maintains timely or confidential information such as building structure reports, intelligence, assignment updates, or strategic information. SOS assumes that there is a pre-determined subset of clients scattered throughout the wide-area network who require (and should have) access to this information, although more recent work [24] somewhat relaxes this requirement.

The approach taken by SOS is *proactive.* In a nutshell, the portion of the network immediately surrounding the target (location to be protected) aggressively filters and blocks all incoming packets whose source addresses are not "approved". The small set of source addresses (potentially as small as 2-3 addresses) that are "approved" at any particular time is kept secret so that attackers cannot use them to pass through the filter. These addresses are picked from among those within a distributed set of nodes throughout the wide area network, that form a *secure overlay*: any transmissions that wish to traverse the overlay must first be validated at entry points of the overlay. Once inside the overlay, the traffic is tunneled securely for several hops along the overlay to the "approved" (and secret from attackers) locations, which can then forward the validated traffic through the filtering routers to the target. The two main principles behind this design are: *(i)* elimination of communication "pinch" points, which constitute attractive DoS targets, via a combination of filtering and overlay routing to obscure the identities of the sites whose traffic is permitted to pass through the filter, and *(ii)* the ability to recover from random or induced failures within the forwarding infrastructure or among the overlay nodes.

The overlays are secure with high probability, given attackers who have a large but finite set of resources to perform the attacks. The attackers also know the IP addresses of the nodes that participate in the overlay and of the target that is to be protected, as well as the details of the operation of protocols used to perform the forwarding. However, the assumption is that the attacker does not have unobstructed access to the network core. That is, the model allows for the attacker to take over an arbitrary (but finite) number of hosts, but only a small number of core routers. It is more difficult (but not impossible) to take control of a router than an end-host or server, due to the limited number of potentially exploitable services offered by the former. While routers offer very attractive targets to hackers, there have been very few confirmed cases where take-over attacks have been successful. Finally, SOS assumes that the attacker cannot acquire sufficient resources to severely disrupt large portions of the backbone itself (*i.e.,* such that all paths to the target are congested).

A stochastic analysis of the SOS architecture shows that even attackers that are able to launch massive attacks are very unlikely to prevent successful communication. For instance, attackers that are able to launch attacks upon 50% of the nodes in the overlay have roughly one chance in one thousand of stopping a given communication from a client who can access the overlay through a small subset of overlay nodes. Furthermore, use of SOS increases end-to-end latency by an average factor of 2, which we believe is an acceptable alternative to severely degraded or even no connectivity to the remote service. Attacks against the SOS infrastructure itself only cause a temporary disruption of communication, on the order of 10 seconds; furthermore, they must persist — otherwise the overlay will recover from component failures and re-integrate them seamlessly. More details can be found in [40, 24, 41].

3.2 Intrusion Detection Systems

3.2.1 Surveillance Detection

Security software must detect surveillance activities to counter the escalating sophistication and sheer prevalence of today's online attack procedures. Surveillance, the scanning of target IPs and ports for vulnerabilities, is the fundamental means to gather online attack intelligence, and is an increasingly common part of precise attack targeting. This is reflected by an alarmingly high proportion of connection attempts that are indeed surveillance probes. The origins of such attempts range across most countries of the world, initiated by human attack activities as well as worms and otherwise captured drones. The range of technical strategies to perform surveillance is growing in variety and sophistication as methods become more precise and more stealthy (*i.e.,* camouflaged against detection, such as by stretching slowly over time or using multiple source addresses) [30, 55, 67]. Only with the full-scale detection of surveillance activities can security systems be augmented to match this arms race, organizing the flood of detected surveillance attempts with watch lists, correlation and intelligence profiling.

Full-scale surveillance detection, *i.e.,* detecting this range of surveillance activities with high precision, presents a series of technical challenges. For example, real-time tracking of all prospec-

tive scanners within a high bandwidth network presents challenges with respect to memory use and speed, given the temporal analyses necessary to detect increasingly prevalent and stealthy scanning. Moreover, certain network tap points suffer from crippling information loss, such as the partial information accessible at a peering center due to unpredictably asymmetric routing.

System Detection (SysD)'s surveillance detection system [63] employs a cascading filter design that coordinates a series of specialized heuristics across extrapolated connection records, individual probes, scans and coordinated scanning groups. This design provides scalability via data reduction across aggregate filters, and detects scans and probes in high-bandwidth environments with high coverage and a low false positive (FP) rate. Two variations specialize over environment class: enclave surveillance detection (ESD) and peering center surveillance detection (PSD).

ESD is implemented and fully operational as a module in SysD's Antura Recon platform (formerly known as the Hawkeye Operational Platform). The infrastructure includes tools and APIs that allow various intrusion detection components to be "plugged in" and deployed. Antura modules include analysis algorithms (*e.g.,* ESD), feature extraction and data parsing procedures.

3.2.2 Anomaly Detection

Our work in anomaly detection has spanned multiple domains. We discuss two common features here: registry (configuration) and filesystem access, both of which are prevalent in situations where an attacker gains privileged access to a system.

3.2.2.1 Registry-based anomaly detection.

Microsoft Windows is one of the most popular operating systems today, and also one of the most often attacked. There are two widely deployed first lines of defense against malicious software on such hosts: virus scanners and security patches. Virus scanners attempt to detect malicious software on the host, and security patches are operating system updates to fix the security holes that malicious software exploits. Both of these methods suffer from the same drawback: they are effective against known attacks but are unable to detect and prevent new types of attacks.

A second line of defense is through IDS systems. Unfortunately, most commercial host-based IDS systems that are widely in use are based on signature algorithms, and require previous knowledge of an attack and are rarely effective on new attacks. Recently, however, there has been growing interest in the use of data mining techniques such as anomaly detection in IDS systems [45, 48]. Anomaly detection algorithms build models of normal behavior in order to detect deviant behavior and which may correspond to an attack [19, 26]. The main advantage of anomaly detection is that it can detect new attacks and can be an effective defense against new malicious software. Anomaly detection algorithms have been applied to network intrusion detection [26, 36, 47] and also to the analysis of system calls for host based intrusion detection [28, 29, 33, 46, 75]. However, there are two problems to the system call approach to host based IDS which inhibits their use in actual deployment. The first is that the computational overhead of monitoring all system calls is very high, which degrades the performance of a system. Additionally, the distribution of system calls is irregular by nature, which makes it difficult to differentiate between normal and malicious behaviors, which may cause a high false positive rate.

We have developed a new approach to host-based IDS that monitors a program's use of the Windows Registry, called RAD (Registry Anomaly Detection), which monitors the accesses to the registry in real-time and detects the actions of malicious software [16]. The Windows Registry is very heavily used, making it a good source of audit data. By building a sensor on the registry and applying the information gathered to an anomaly detector, we can detect activity that corresponds to malicious software. The main advantages of monitoring the Windows Registry is that the activity is regular by nature, can be monitored with low computational overhead, and almost all system activities interact with the registry.

Our anomaly detection algorithm is a registry-specific version of PHAD (Packet Header Anomaly Detection) [48]. An anomaly detection algorithm is then applied to this data to detect abnormal registry behavior which corresponds to the actions of malicious software. Modifications of the PHAD algorithm are also made in the RAD system. Results of experiments evaluating the RAD system show that it is effective in detecting attacks while maintaining a low rate of false alarms.

3.2.2.2 File-based anomaly detection.

In addition to Registry access, file-based anomaly detection is critical in Unix environments, since there is no central registry to monitor. Anomalous process executions (possibly those that are malicious) may not truly damage a system until the malicious execution attempts to alter or damage the machine's permanent store. Thus, a malicious attack that alters run-time memory is perhaps less important than actions that attempt to damage permanent store of the host in question.

We focus our auditing on the underlying file system, as any malicious execution intended to damage a host will ultimately attempt to manipulate it. The File Wrapper Anomaly Detection System (FWRAP) is a host-based anomaly detector that utilizes file wrapper technology to monitor file system accesses. It is the counterpart of the registry "wrapper" developed for RAD for the registry. The file wrappers implemented in FWRAP are based upon work described in [78] and operate in much the same fashion as the wrapper technology described in [42, 18]. The wrappers are implemented to extract a set of information about each file access including, for example, date and time of access, host, UID, PID, and filename, *etc.* Each such file access thus generates a record describing that access. Intuitively, these records provide the same type of information associated with a Windows Registry access, and as such can be modeled in the same fashion.

We also use the Probabilistic Anomaly Detection (PAD) algorithm to model file system accesses [32]. We apply PAD to analyze and model file access data, merged with information about the running processes that invoke such accesses, to train an anomaly detector in much the same fashion as accomplished with RAD. In the same way RAD modeled the actions of running programs via the System Registry, we are modeling running processes via the underlying file system.

3.3 Process Migration Using ZAP

Process migration [56] is the ability to transfer a process from one machine to another. It is a useful facility in distributed computing environments, especially as computing devices become more pervasive and Internet access becomes more ubiquitous. Among the potential benefits of process migration are fault resilience by migrating processes off of faulty hosts, data access locality by migrating processes closer to the data, better system response time by migrating processes closer to users, dynamic load balancing by migrating processes to less loaded hosts, and improved service availability and administration by migrating processes before host maintenance so that applications can continue to run with minimal down-time.

Although process migration provides substantial potential benefits and many approaches have been considered [50], achieving pro-

cess migration functionality has been difficult in practice. Toward this end, there are four important goals that need to be met. First, given the large number of widely used legacy applications, applications should be able to migrate and continue to operate correctly without modification, without requiring that they be written using uncommon languages or toolkits, and without restricting their use of common operating system services. For example, networked applications should be able to maintain their network connections even after being migrated. Second, migration should leverage the large existing installed base of commodity operating systems. It should not necessitate use of new operating systems or substantial modifications to existing ones. Third, migration should maintain the independence of independent machines. It should avoid creating residual dependencies that limit the utility of process migration by requiring machines where a process was previously executed to continue to service a process even after it has migrated to another machine. Fourth, migration should be fast and efficient. Overhead should be small for normal execution and migration.

To overcome limitations in previous approaches to general-purpose process migration, we have created Zap. Zap provides a thin virtualization layer on top of the operating system that introduces a PrOcess Domain (pod) abstraction. A pod provides a group of processes with a private namespace that presents the process group with the same virtualized view of the system. This virtualized view associates virtual identifiers with operating system resources such as process identifiers and network addresses. This decouples processes in a pod from dependencies on the host operating system and from other processes in the system.

Zap virtualization is integrated with a checkpoint-restart mechanism that enables processes within a pod to be migrated as a unit to another machine. Since pods are independent and self-contained they can be migrated freely without leaving behind any residual state after migration, even when migrating network applications while preserving their network connections. Zap can therefore allow applications to continue executing after migration even if the machine on which they previously executed is no longer available. In using a checkpoint-restart approach, Zap not only supports process migration, but also allows processes to be suspended to secondary storage and transparently resumed at a later time.

Zap is designed to support migration of unmodified legacy applications while minimizing changes to existing operating systems. This is done by leveraging loadable kernel module functionality in commodity operating systems that allows Zap to intercept system calls as needed for virtualization and save and restore kernel state as needed for migration. Zap's compatibility with existing applications and operating systems makes it simple to deploy and use. We have implemented a Zap prototype as a loadable kernel module in Linux that supports transparent migration, without any kernel modifications, among separate machines running independent Linux operating systems; it does not require a single-system image across all machines. Our experimental results on our Linux Zap prototype demonstrate that it can provide general-purpose process migration functionality with low overhead.

3.4 Autonomic Software Patching

Recent incidents [12, 13] have demonstrated the ability of self-propagating code, also known as "network worms" [65, 21], to infect large numbers of hosts, exploiting vulnerabilities in the largely homogeneous deployed software base [14, 79, 10]. Even when a worm carries no malicious payload, the direct cost of recovering from the side effects of an infection epidemic can be tremendous [1]. Thus, countering worms has recently become the focus of increased research, generally focusing on content-filtering mechanisms combined with large-scale coordination strategies [51, 68, 54, 35]. The same issues and mechanisms are relevant to the case of automatically-exploitable (scripted) vulnerabilities.

Despite some promising early results, we believe that this approach will be insufficient by itself in the future. We base this primarily on two observations. First, to achieve coverage, such mechanisms are intended for use by routers (*e.g.,* Cisco's NBAR [9]); given the routers' limited budget in terms of processing cycles per packet, even mildly polymorphic worms (mirroring the evolution of polymorphic viruses, more than a decade ago) are likely to evade such filtering. Network-based intrusion detection systems (NIDS) have encountered similar problems, requiring fairly invasive packet processing and queuing at the router or firewall. When placed in the application's critical path, as such filtering mechanisms must, they will have an adverse impact on performance. Second, end-to-end "opportunistic"[1] encryption in the form of TLS/SSL [27] or IPsec [39] is being used by an increasing number of hosts and applications [6]. We believe that it is only a matter of time until worms start using such encrypted channels to cover their tracks. Similar to the case for distributed firewalls [20, 34], we believe that these trends argue for an end-point worm-countering mechanism.

A preventative approach to the worm problem is the elimination or minimization of remotely-exploitable vulnerabilities, such as buffer overflows. Detecting potential buffer overflows is a very difficult problem, for which only partial solutions exist (*e.g.,* [23, 44]). "Blanket" solutions such as StackGuard or MemGuard [25] typically exhibit at least one of two problems: reduced system performance, and self-induced denial of service (*i.e.,* when an overflow is detected, the only alternative is to terminate the application). Thus, they are inappropriate for high-performance, high-availability environments such as a heavily-used e-commerce web server. An ideal solution would use expensive protection mechanisms only where needed and allow applications to gracefully recover from such attacks.

The autonomic software patching mechanism is an end-point first-reaction system that tries to automatically patch vulnerable software by identifying and transforming the code surrounding the exploited software flaw. Briefly, we use instrumented versions of an enterprise's important services (*e.g.,* web server) in a sandboxed environment. This environment is operated in parallel with the production servers, and is not used to serve actual requests. Instead, we use it as a "clean room" environment to test the effects of "suspicious" requests, such as potential worm infection vectors. Appropriate instrumentation allows us to determine the buffers and functions involved in a buffer overflow attack. We then apply several source-code transformation heuristics that aim to contain the buffer overflow. Using the same sandboxed environment, we test the produced patches against both the infection vectors and a site-specific functionality test-suite, to determine correctness. If successful, we update the production servers with the new version of the targeted program.

Our architecture makes use of several components that have been developed for other purposes. Like SABER itself but on a smaller scale, its novelty lies in the combination of all the components in fixing vulnerable applications without unduly impacting their performance or availability. Our major assumption is that we can extract a worm's infection vector (or, more generally, one instance of it, for polymorphic worms). We envision the use of various mechanisms such as honeypots, host-based, and network-based intrusion

[1]By "opportunistic" we mean that client-side, and often server-side, authentication is often not strictly required, as is the case with the majority of web servers or with SMTP over TLS (*e.g.,* sendmail's STARTSSL option).

detection sensors. Note that vector-extraction is a necessary precondition to any reactive or filtering-based solution to the worm problem. A secondary assumption is that the source code for the application is available.[2]

To determine the effectiveness of the approach, we tested a set of 17 applications vulnerable to buffer overflows, compiled by the Cosak project [8]. We simulated the presence of a worm (even for those applications that were not in fact network services) by triggering the buffer overflow that the worm would exploit to infect the process. Our experiments show that our architecture was able to fix the problem in 82% of all cases. An experiment with a hypothetical vulnerability in the Apache web server showed that the total time to produce a correct and tested fix was 8.3 seconds [66].

3.5 Event-Based Command and Control

3.5.1 MEET

The goal of the Multiply Extensible Event Transport (MEET) is to provide an extensible, survivable, and efficient publish/subscribe substrate for advanced distributed applications. While pub/sub is an elegant paradigm for any network program, it becomes a necessity when the number of components and interactions rises beyond the point of manual administration. MEET offers additional useful features to a distributed security system: decentralized architecture, modular and extensible architecture, secure communication, and high performance. These are described in detail below.

When a distributed system consists of many data sources, many data sinks, and multicast distribution patterns (multiple recipients for a single data item), the traditional network paradigm of unicasting data to a specific communication partner begins to break down. The complexity of managing the patterns of communication overwhelms the primary tasks of the distributed system. By offloading the problem of distributed multicast (or unicast) communication to a dedicated middleware component, the design, construction, and integration of the distributed system is vastly simplified. Publishers simply push data into the ether. Subscribers describe the data they're interested in. The pub/sub system ensures that the data goes to the right places, securely and efficiently. Pub/sub is particularly tolerant of nodes appearing and disappearing, a useful attribute for any large distributed system in the real world.

MEET's distributed architecture sets it apart from other pub/sub systems such as Elvin [64] or Siena [22]. Elvin passes all traffic through a set of one or more core nodes, and Siena distributes all data along a statically defined tree, but MEET allows participating nodes to be configured in an arbitrary graph. While any individual data publication is distributed along a tree, MEET's awareness of the full network topology allows it to instantly and transparently reconfigure and recover in cases of network partition. Nodes can auto-discover peers, and merge themselves into the fabric automatically. The system can be configured for varying degrees of routing information redundancy. Overall, MEET provides a high degree of infrastructure survivability for higher-level applications.

MEET is designed as a set of cooperating components that can be replaced at runtime, *e.g.*, routing modules (BGP, OSPF), channel modules (TCP, UDP, IPsec), message types, and so on. This allows MEET to reconfigure itself to run on low-end PDAs or high-end servers, to adjust to changing environments (*e.g.*, a laptop removed from its docking station), and to add new capabilities, while continuing to route messages. The extensibility of MEET extends pub/sub's tolerance of changes in network topology to the software itself, allowing new features, protocols, and standards to be added to the system without downtime. As security protocols and standards are constantly evolving, this feature greatly eases the administration of the network. Additionally, advanced application-specific features (*e.g.*, bloom-filter-based subscriptions) can be integrated directly into the messaging framework.

MEET offers a range of security support, under the end-to-end assumption that higher-level applications will have their own security architecture, which shouldn't be duplicated at the lowest level. MEET supports a number of security protocols for inter-node communication, including IPsec and TLS/SSL, and of course more can be added. MEET also provides hooks for private channel support. In addition, pub/sub greatly simplifies the use of shared secret key systems, where a key is split into multiple pieces, and majority of subkey holders are needed to decrypt a secret. By requiring, for instance, that a majority of nodes must agree to a change in policy, a distributed system can become much more resistant to the compromise of a few nodes.

Finally, MEET is designed to run in embedded, hard real-time environments. The central core routes messages with high efficiency, guaranteeing bounded-time performance (assuming bounded-time subscription tests). The native message format is optimized for fast routing, with key bits for classification at the front. Messages with large or variable-length fields have an index structure to speed processing.

3.5.2 XUES Event Correlation

Most commercial event-based cross-platform problem detection and repair tools are largely application-neutral, leaving understanding of what the system is supposed to be doing (and how and why) to the human administrator. Thus only the simplest general-purpose analyses and repairs can be automated, and complex cross-domain security scenarios are difficult to support.

Our solution to this problem is what we call XUES [38], or XML Universal Event Service. XUES runs as a lightweight, decentralized, easily integrable collection of active middleware components, tied together via a publish/subscribe (content-based messaging) event system. We have used XUES to monitor a variety of target applications employing application-level semantics. XML is used as a native data format, providing rich expressiveness.

XUES consists chiefly of two services. The Event Packager provides event translation and "flight recorder" services to standardize and log all incoming event streams. The Event Distiller performs sophisticated cross-stream temporal event pattern analysis and correlation to monitor desirable (and, correspondingly, undesirable) behavior; we describe it in further detail.

3.5.3 Event Distiller

In many monitored systems, the key is to determine what original failure ("root cause") started a cascading problem [59]. The Event Distiller is the component responsible for detecting causality among the events in significant event sequences, by performing time-based pattern matching. Internally, it uses a collection of nondeterministic state engines for temporal complex event pattern matching. While this is memory-intensive, it allows a richer representation of event sequences: logic constructs are supported, as are loops, rule chaining, and variable binding. We also mitigate memory usage by supporting timeouts and automatic garbage collection. Timestamped event reordering is also supported, so if events arrive out-of-order within a certain window, the Event Distiller will rearrange them appropriately so that sequences, and causality, can still be recognized correctly.

[2] Although our architecture can use binary rewriting techniques [60], we focus on source-code transformations. We should also note that several popular server applications are in fact open-source (*e.g.*, Apache [10], Sendmail, MySQL, and BIND).

Event Distiller rules may be populated in one of several ways: First, an XML rulebase is supported, where event sequences are specified, along with time-bound parameters as well as "success" and "failure" notifications; we have also developed a GUI to assist a XUES integrator; it also works as a systems management console for human engineers, although our goal is to automate many repairs within a XUES feedback loop. Second, the Event Distiller supports dynamic rule generation – messages can be sent to the Event Distiller with XML snippets specifying a rule or a segment of a rule (*e.g.,* to construct new rules on the fly or modify existing rules). Third, as with the Event Packager, other sources can be easily integrated; for example, new SABER components can be added and rules modified on-the-fly.

4. SCENARIOS

In this section, we discuss deployment possibilities and look at various category of attacks, and how SABER is well-equipped to handle them.

4.1 Deployment scenario

For the context of this section, we envision an Internet-enabled bank whose goal is to provide both customers and peers with the ability to securely conduct transactions.

The bank's commodity Internet connection is heavily firewalled, but more importantly, firewalling is also done at their ISP to only allow legitimate SOS "overlay traffic" to prevent saturation of the link by a DoS attack. At the same time, IDS sensors are placed in the firewall at both the ISP and bank's endpoints to monitor incoming traffic.

Within the bank's LAN, a number of servers and a number of teller machines (frontends) are scattered throughout. IDS sensors are also placed here. Critical server services (such as customer databases) are embedded within Zap pods to make process migration simple. Worm vaccine monitors are deployed on servers that provide services to either internal bank clients or to teller machines. Extra servers are provided for redundancy's sake; these servers are configured to provide minimal external interfaces, but still have Zap services and IDS sensors located on them to continue monitoring and to ease service migration.

Each of the SABER components has two communication mechanisms: the ability to report alerts and to accept controls. These messages are communicated through a decentralized MEET network, whose nodes are scattered redundantly throughout the LAN. XUES is colocated with MEET at critical points to aggregate and monitor alerts.

Note that this scenario can be scaled up to multiple-site scenarios, depending on the requirements. The sites may act as independent SABER deployments, or can communicate attack information via external MEET nodes.

4.2 Simple attack scenarios

We outline a sample of single-attack scenarios, how SABER would handle them, and the net effect on survivability.

1. Bandwidth denial-of-service attack: A cluster of computers, potentially distributed across the Internet, issue a brute-force DoS attack against the bank (*e.g.,* SYN floods). Several aspects of the system prevent the DoS. First, the overlay network will drop the vast majority of these packets as the machines do not have the necessary credentials to perform a transaction. Any traffic directly hitting the bank's ISP will also be dropped, as it's not sent from a trusted secret servlet.

2. Service denial-of-service attack: A credentialed "user" attempts a large number of multiple transactions with the bank, hoping to consume the resources of the webserver. Apart from standard webserver rate-limiting and IP threshholding, protocols such as JFK [15] prevent the server from being computationally bound for secure transactions. Application-specific instances of SOS, such as WebSOS [24], can be used in a similar fashion for particular applications.

3. Outside attacker: Posing as a "credentialed" user, a malicious entity attempts to exploit the web service, via buffer overflow or other known vulnerability. Possibilities include:

   - The outside attacker is thwarted because the IDS has already detected suspicious scan behavior, and SABER triggered the revocation of the user's credentials, essentially firewalling the user away as SOS no longer tolerates the user.
   - The outside attacker is thwarted because the IDS has already detected suspicious scan behavior and the port it was occurring on, which prompted the worm vaccine to simulate, trap, and patch the buffer overflow or similar vulnerability.
   - The outside attacker manages to compromise the web server, but the software patcher shortly thereafter patches and restarts the web server, eliminating the outside attacker's access.
   - The outside attacker manages to compromise the web server (and, possibly, the database server), but the IDS detects anomalous behavior on the web server and, as a last resort, triggers a safety lockdown, where the database process is suspended by Zap and safely migrated to a secure spare server, thereby denying users' data but keeping the bank's critical infrastructure secure and protected from the attacker.

4. Inside attacker: Posing as an "employee", a malicious entity attempts to exploit services within the network, particularly but not limited to teller services, to gain access to unauthorized information or to cause service damage[3]. A number of possibilities may occur here as well:

   - The inside attacker succeeds in gaining general access on the teller service server. The IDS detects anomalous behavior and triggers the shutdown down the teller service, and optionally triggers a Zap migration of the database to a more secure, but less available, location, preventing intrusion into the database server.
   - The inside attacker fails in gaining general access; a known vulnerability on the teller server is automatically patched using the vaccination facilities.
   - The inside attacker gains general access to a workstation, and attempts to launch an attack from there. The IDS detects the attack and triggers the revocation of the workstation user account, and possibly triggers a Zap migration of the sensitive database information.
   - The inside attacker unplugs network wires to try and trigger a partition, thereby reducing SABER's ability to coordinate. The redundant MEET and XUES components continue to function on the partition not entirely

[3]A recent FBI study determined that as much as 70% of all successful attacks are launched from insiders.

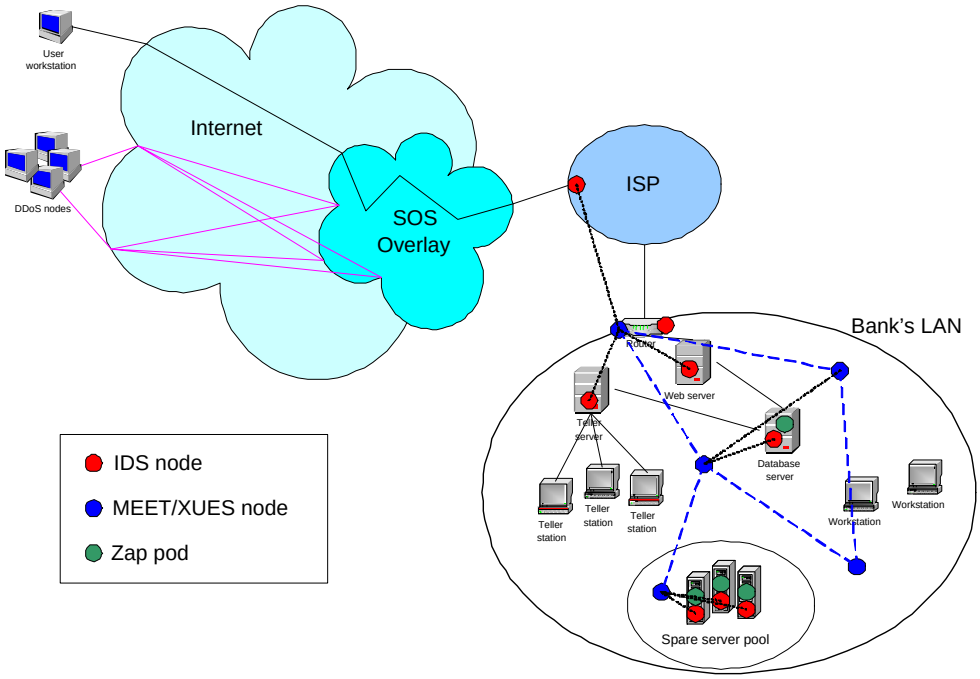

Figure 2: Example SABER deployment.

isolated, including the server banks, and automatically trigger a shutdown of the servers.

These list of possible attacks is not meant to be comprehensive. However, it is notable that in the vast majority of cases the service largely survives unscathed: 6 of the above 10 possibilities ultimately result in little or no degradation of service. In the other 4 intrinsically nonrecoverable attacks, data compromise is averted, which is often an acceptable alternative. If this is not sufficient, service migration across redundant, disjoint networks is a possibility, as is better physical access restrictions for the inside attacker.

4.3 Complex attack scenarios

Complex attacks often involve multiple attack approaches. This is common in a number of situations: the attacker may actually be comprised of a distributed yet organized team taking different approaches, there may be an "attack competition", or there might be one attacker who has managed to gain "zombie control" over a number of unsuspecting machines distributed around the Internet, enabling him to launch a DDoS to cover his tracks while performing actual attacks.

SABER stresses redundancy and decentralization throughout its core, which enables it to degrade gracefully, even with multiple attackers. By employing a publish/subscribe event system capable of self-healing and self-management, network partitions or packet loss triggered by an attack need not be the end of coordination. Instead, given sufficient redundancy, SABER supports a broad variety of worst-case scenarios.

5. RELATED WORK

Related work naturally draws from a number of fields, due to the broad set of tools presented in this paper. We do not address related work for each of the individual components here, but rather just the framework; see the respective cited papers for more details on the appropriate subjects.

Secure survivable architectures are typically very application- or domain-specific. Ghosh, *et al.* [31] propose "fault injection analysis" applied to software, while Strunk, *et al.* [70] apply a low-level approach: they propose an intrusion detection and recovery model at the storage layer. Kreidl, *et al.* [43] propose a formalized feedback-driven model for individual COTS applications. In contrast, SABER is a generalized, system-level, application-neutral architecture that encompasses a broad array of tools, yet can be integrated into existing third-party solutions, although we may investigate integrating application-level semantics in the future.

Some recent preliminary research has begun on building frameworks to support a wider variety of software. The DARPA OASIS [2] ("Organically Assured and Survivable Information System") program sponsored several projects that address a number of related issues. In particular, the Willow project [76] supports intrusion tolerance by establishing a high-level language for specifying intrusion tolerance policies that are translated into control code as well as the notion of "postures" which dictate the state of a system (*e.g.*, in neutral mode, under attack, *etc.*). The SITAR project [73] utilizes components that wrap COTS services and utilizes a set of policies to determine legitimate requests. In contrast to Willow, SITAR, and a number of the other OASIS projects, SABER aims to be a more *ad hoc* framework designed to be applied in already-deployed service environments with minimal disruption.

The HACQIT architecture [37, 62, 61] uses various sensors to detect new types of attacks against secure servers, access to which is limited to small numbers of users at a time. Any deviation from expected or known behavior results in the possibly subverted server to be taken off-line. Similar to our approach on autonomic patching, a sandboxed instance of the server is used to conduct "clean room" analysis, comparing the outputs from two different implementations of the service (in their prototype, the Microsoft IIS and

Apache web servers were used to provide application diversity). Machine-learning techniques are used to generalize attack features from observed instances of the attack. Content-based filtering is then used, either at the firewall or the end host, to block inputs that may have resulted in attacks, and the infected servers are restarted. Due to the feature-generalization approach, trivial variants of the attack will also be caught by the filter.

The APOD project [17, 57, 58] uses a combination of intrusion detection, firewalls, TCP stack probes, virtual private networks, bandwidth reservation, and traffic shaping mechanisms, to allow applications to detect attacks and contain the damage of successful intrusions by changing their behavior. They also discuss the use of fine-grained access control at the object level (using CORBA), as well as the use of randomizing techniques, such as changing the TCP ports applications listen to, and service replication. Although APOD shares some similarities with SABER, our architecture focuses on fixing the vulnerabilities at the software level, as well as using scalable DoS-protection mechanisms. Thus, it could be combined with some of the mechanisms developed for use in APOD.

Other related work includes Secure Computing Corporation's "Intrusion Tolerant Server Infrastructure" [11], the ITS project [77], and SITAR [74]. These architectures emphasize intrusion detection, filtering of known attacks, and reconfiguration. Although they share these components with SABER, the latter better addresses root causes (such as software flaws), and uses scalable distributed mechanisms to achieve protection against some common network-based attacks. The MAFTIA project [3] is developing an open architecture for transactional operations on the Internet, modeling (successful) attacks as faults and applying approaches developed in the realm of fault tolerance.

Finally, autonomic computing is rapidly growing as its own field. Sterritt, *et al.* [69] suggest an overall model of how survivable systems require a number of properties, including *self-protecting*, *self-configuring*, *self-healing*, and *self-optimizing* capabilities. We feel that the application of SABER to a target system enable it to be self-protecting as well as self-healing.

6. FUTURE WORK

We have identified the next main steps in the development of the SABER architecture to make it more autonomic and survivable, and discuss them briefly.

- **Coordination language**: In the current model, inter-component logic has to be manually developed, e.g., the communication of IDS results to Zap via MEET. We envision the development of higher-level semantics to improve an organization's ability to deploy SABER components. Such a higher-level language would help define what "approved services" and "approved operations" are, and would then compile these down for use by the individual components. Learning approaches could also be utilized to assemble these lists. Finally, various workflow process approaches could be adopted as effectors for such a language, such as our previous work on Workflakes [72].

- **Recovery assistance**: SABER's architecture is currently architected to prevent attacks and, given a successful attack, to limit damage with a minimum of human intervention. However, the semantics captured by SABER could additionally be useful after human intervention, *i.e.*, SABER could support the reverse migration of a process back to a patched server.

  Future work could also support a "staged" autonomic recovery, where process are migrated off of a machine which has been attacked until such time SABER's patching components could develop a solution, at which point the component would be seamlessly migrated back. Ongoing work in the individual components that support our proposed SABER implementation will also make it easier to develop such a solution.

7. CONCLUSIONS

SABER has tremendous utility as a general framework to support the collaboration of best-of-breed security technologies to maximize service survivability not only in preventing attacks, but also to minimize loss in networks where attack has been successful. Our proposed implementation of uses a number of unique tools to support a broad array of scenarios. Given more coordination support, we feel that SABER will be adequately prepared to meet the next generation of attacks and security vulnerabilities.

8. REFERENCES

[1] 2001 Economic Impact of Malicious Code Attacks. http://www.computereconomics.com/cei/press/pr92101.html.

[2] DARPA OASIS (Organically Assured and Survivable Information System). http://www.tolerantsystems.org/index.html.

[3] Malicious- and Accidental-Fault Tolerance for Internet Applications. RTD Research Project IST-1999-11583, IST Programme. http://maftia.org/.

[4] Microsoft Security Tool Kit: Installing and Securing a New Windows 2000 System. Microsoft TechNet. http://www.microsoft.com/technet/security/tools/tools/w2knew.asp.

[5] Microsoft Windows Software Update Services. http://www.microsoft.com/windows2000/windowsupdate/sus/.

[6] OC48 Analysis – Trace Data Stratified by Applications. http://www.caida.org/analysis/workload/byappli\-cation/oc48/port_analy%sis_app.xml.

[7] RedHat 9 Security Advisories. https://rhn.redhat.com/errata/rh9-errata-security.html.

[8] The Code Security Analysis Kit (CoSAK). http://serg.cs.drexel.edu/cosak/index.shtml/.

[9] Using Network-Based Application Recognition and Access Control Lists for Blocking the "Code Red" Worm at Network Ingress Points. Technical report, Cisco Systems, Inc.

[10] Web Server Survey. http://www.securityspace.com/s_survey/data/200304/.

[11] Intrusion Tolerant Server Infrastructure. http://www.tolerantsystems.org/ProjectSummaries/Intrusion_Tolerant_Serv%er_Infrastructure.html, 2000.

[12] CERT Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-19.html, July 2001.

[13] Cert Advisory CA-2003-04: MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html, January 2003.

[14] The Spread of the Sapphire/Slammer Worm. `http://www.silicondefense.com/research/worms/slammer.php`, February 2003.

[15] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A. D. Keromytis, and O. Reingold. Efficient, DoS-Resistant, Secure Key Exchange for Internet Protocols. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 48–58, November 2003.

[16] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. J. Stolfo. Detecting malicious software by monitoring anomalous windows registry accesses. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID-2002)*, Zurich, Switzerland, October 2002.

[17] M. Atighetchi, P. Pal, C. Jones, P. Rubel, R. Schantz, J. Loyall, and J. Zinky. Building Auto-Adaptive Distributed Applications: The QuO-APOD Experience. In *Proceedings of the 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with the 23rd International Conference on Distributed Computing Systems*, May 2003.

[18] R. Balzer. Mediating connectors. In *19th IEEE International Conference on Distributed Computing Systems Workshop*, 1994.

[19] V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley and Sons, 1994.

[20] S. M. Bellovin. Distributed Firewalls. *;login: magazine, special issue on security*, pages 37–39, November 1999.

[21] J. Brunner. *The Shockwave Rider*. Del Rey Books, Canada, 1975.

[22] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. In *ACM Transactions on Computer Systems*, volume 19(3), pages 332–383, August 2001.

[23] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.

[24] D. L. Cook, W. G. Morein, A. D. Keromytis, V. Misra, and D. Rubenstein. WebSOS: Protecting Web Servers From DDoS Attacks. In *Proceedings of the IEEE International Conference on Networks (ICON)*, September/October 2003.

[25] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[26] D. E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, SE-13:222–232, 1987.

[27] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, January 1999.

[28] E. Eskin. Anomaly detection over noisy data using learned probability distributions. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000.

[29] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. pages 120–128. IEEE Computer Society, 1996.

[30] Fyodor. The art of port scanning. *Phrack 51*, 7, September 1997. `http://www.phrack.com/phrack/51/P51-11`.

[31] A. K. Ghosh and J. M. Voas. Inoculating software for survivability. *Communications of the ACM*, 42(7):38–44, 1999.

[32] S. Hershkop, R. Ferster, L. H. Bui, K. Wang, and S. J. Stolfo. Host-based anomaly detection by wrapping file system accesses. Technical report, Columbia University Department of Computer Science, April 2003.

[33] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detect using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[34] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing a Distributed Firewall. In *Proceedings of Computer and Communications Security (CCS)*, pages 190–199, November 2000.

[35] R. Janakiraman, M. Waldvogel, and Q. Zhang. Indra: A peer-topeer approach to network intrusion detection and prevention. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, June 2003.

[36] H. S. Javitz and A. Valdes. The nides statistical component: Description and justification. Technical report, SRI International, 1993.

[37] J. E. Just, L. A. Clough, M. Danforth, K. N. Levitt, R. Maglich, J. C. Reynolds, and J. Rowe. Learning Unknown Attacks – A Start. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.

[38] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kinesthetics eXtreme: An external infrastructure for monitoring distributed legacy systems. In *Proceedings of the Autonomic Computing Workshop, Fifth Annual Workshop on Active Middleware Services*, 2003.

[39] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, Nov. 1998.

[40] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM*, pages 61–72, August 2002.

[41] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: An Architecture For Mitigating DDoS Attacks. *IEEE Journal on Selected Areas of Communications (JSAC)*, 2003. (to appear).

[42] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *10th Annual Computer Security Applications Conference*, pages 134–144, December 1994.

[43] O. Kreidl and T. Frazier. Feedback control applied to survivability: a host-based autonomic defense system. *IEEE Transactions on Reliability*, Vol. 52, No. 3, September 2003.

[44] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, August 2001.

[45] W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. 1999.

[46] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix processes execution traces for intrusion detection. pages 50–56. AAAI Press, 1997.

[47] W. Lee, S. J. Stolfo, and K. Mok. Data mining in work flow environments: Experiences in intrusion detection. In *Proceedings of the 1999 Conference on Knowledge Discovery and Data Mining (KDD-99)*, 1999.

[48] M. Mahoney and P. Chan. Detecting novel attacks by

identifying anomalous network packet headers. Technical Report CS-2001-2, Florida Institute of Technology, Melbourne, FL, 2001.

[49] M. V. Mahoney and P. K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 376–385. ACM Press, 2002.

[50] D. Milojicic, F. Douglis, and R. Wheeler. *Mobility: Pro-cesses, Computers, and Agents*. Addison Wesley Longman, February 1999.

[51] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference*, April 2003.

[52] D. Moore, G. M. Voelker, and S. Savage. Inferring internet Denial-of-Service activity. In *Proceedings of the 10th Usenix Security Symposium*, pages 9–22, 2001.

[53] D. Newman, J. Snyder, and R. Thayer. Crying wolf: False alarms hide attacks. *Network World*, June 2002. `http://www.nwfusion.com/techinsider/2002/0624security1.html`.

[54] D. Nojiri, J. Rowe, and K. Levitt. Cooperative Response Strategies for Large Scale Attack Mitigation. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX)*, pages 293–302, April 2003.

[55] S. Northcutt. *Network Intrusion Detection: An Analyst's Handbook*, pages 122–139. New Riders, Indianapolis, 1999.

[56] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, Boston, MA, December 2002.

[57] P. PAl, M. Atighetchi, F. Webber, R. Schantz, and C. Jones. Adaptive Use of Netwokr-Centric Mechanisms in Cyber-Defense. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, May 2003.

[58] P. Pal, M. Atighetchi, F. Webber, R. Schantz, and C. Jones. Reflections on Evaluating Survivability: The APOD Experiments. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications*, April 2003.

[59] L. Perrochon. Using context-based correlation in network operations management. Technical report, Stanford University Department of Computer Science, 1999. `http://pavg.stanford.edu/cep/cidf.ps.gz`.

[60] M. Prasad and T. Chiueh. A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[61] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Generate-and-Test, and Generalization. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS)*, January 2003.

[62] J. C. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The Design and Implementation of an Intrusion Tolerant System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[63] S. Robertson, E. V. Siegel, M. Miller, and S. J. Stolfo. Surveillance detection in high bandwidth environments. In *Proceedings of the 2003 DARPA DISCEX III Conference*, April 2003.

[64] B. Segall, D. Arnold, J. Boot, et al. Content-based routing with Elvin4. In *Proceedings of AUUG2K*, June 2000.

[65] J. F. Shoch and J. A. Hupp. The "worm" programs – early experiments with a distributed computation. *Communications of the ACM*, 22(3):172–180, March 1982.

[66] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, June 2003.

[67] S. Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy portscans. In *Proceedings of the Seventh ACM Conference on Computer and Communications Security*, Athens, Greece, 2000.

[68] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[69] R. Sterritt and D. Bustard. Autonomic computing–a means of achieving dependability? In *Proceedings of IEEE International Conference on the Engineering of Computer Based Systems (ECBS'03)*, pages 247–251, April 2003.

[70] J. D. Strunk, G. R. Goodson, A. G. Pennington, C. A. Soules, and G. R. Ganger. Intrusion detection, diagnosis, and recovery with self-securing storage. Technical report, Carnegie Mellon University, 2002.

[71] P. Thompson. Web services – beyond http tunneling. In *W3C Workshop on Web Services*, April 2001.

[72] G. Valetto and G. Kaiser. Using process technology to control and coordinate software adaptation. In *Proceedings of International Conference on Software Engineering*, May 2003.

[73] F. Wang, F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi, and F. Jou. Sitar: A scalable intrusion tolerance architecture for distributed servers. In *Proceedings of the IEEE 2nd SMC Information Assurance Workshop*, 2001.

[74] F. Wang and R. Uppalli. SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services. In *Volume II of the Proceedings of DISCEX III*, pages 153–155, April 2003.

[75] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. pages 133–145. IEEE Computer Society, 1999.

[76] A. Wolf, D. Heimbigner, A.Carzaniga, J. Knight, P. Devenbu, and M. Gertz. Bend, don't break: Using reconfiguration to achieve survivability. In *Proceedings of the Third Information Survivability Workshop (ISW2000)*, pages 187–190, October 2000.

[77] A. Wolf, D. Heimbigner, A. Carzaniga, J. Knight, P. Devenbu, and M. Gertz. Bend, Don't Break: Using Reconfiguration to Achieve Survivability. In *Proceedings of the 3rd Information Survivability Workshop*, pages 187–190, October 2000.

[78] E. Zadok and I. Badulescu. A stackable file system interface for Linux. In *LinuxExpo 99*, May 1999.

[79] C. C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, November 2002.