

Survivability: Synergizing Security and Reliability

Crispin Cowan
Immunix, Inc.

September 7, 2003

Abstract

In computer science, reliability is the study of how to build systems that continue to provide service despite some degree of random failure of the system. Security is the study of how to build systems that provide privacy, integrity, and continuation of service. Survivability is a relatively new area of study that seeks to combine the benefits of security and reliability techniques to enhance system survivability in the presence of arbitrary failures, including security failures. Despite apparent similarities, the combination of techniques is not trivial. Despite the difficulty, some success has been achieved, surveyed here.

1 Introduction

At first glance, reliability and security would seem to be closely related. *Security* is defined [65] as privacy, integrity, and continuation of service, the latter seeming to encompass a degree of reliability. Conversely, *reliability* is defined as systems that *mask faults* to prevent *failures* of systems to provide their specified services [64].

The combination reliability and security is a natural fit: both seek to improve system availability, both must deal with failures and their consequences. Techniques to ensure software quality such as source code auditing, type safe languages, fault isolation, and fault injection all work well for both security and reliability purposes; improving one tends to improve the other.

However, interpreting security “faults” (vulnerabilities) as failures in the reliability sense has proven to be problematic. *Failure* is defined as “deviation from specification” which is not helpful if the specification itself is wrong. More over, many real systems are implemented in type-unsafe languages (especially C) and so correspondence to a formal specification cannot easily be assured. Thus reliable software does what it is supposed to do, while secure software does what it is sup-

posed to do, and *nothing else* [1]. The surprising “something else” behaviors form the crux of the software vulnerability problem.

So security and reliability cannot be trivially composed to achieve the benefits of both. *Survivability* is the study of how to combine security and reliability techniques to actually achieve benefits combination of the two. The intersection of the two is to be able to survive failures of the security system. The union of the two is to survive arbitrary failures, including the security system, not just random failures.

The rest of this chapter is organized as follows. Section 2 describes the problem of composing security and reliability techniques in greater detail. Section 3 surveys and classifies survivability techniques. Section 4 surveys methods of assessing survivability. Section 5 describes related work surveying survivability. Section 6 presents conclusions.

2 The Problem: Combining Reliability and Security

Reliability defines a *fault* to be when something goes wrong, and a *failure* as when a fault manifests a consequence to a user, such that the system no longer performs its required function. Thus the principle approach to building reliable systems is *fault masking* in which some provisions are made to prevent individual faults from producing failures. *Fault tolerance* techniques notably achieve fault masking by providing *redundant* components, so that when one component fails, another can take up its burden.

A crucial assumption to fault tolerance is that faults are *independent*: that there is no causal relationship between a fault in one component and a fault in another component. This assumption breaks down with respect to *security faults* (vulnerabilities) because replication of components replicates the defects. Attackers seeking to exploit these vulnerabilities can readily compromise all replicas, inducing failure. Thus survivable systems must provide something more than replication to be able to survive the design and implementation faults that are at the core of the security problem.

Reliability also assumes that faults are random, while security cannot make such an assumption. For instance, a system that depends on random memory accesses not hitting the address 0x12345678 can be highly reliable (assuming no data structures are adjacent) but is not secure, because the attacker can aim at an address that is otherwise improbable. In the general case, the attacker can maliciously induce faults that would otherwise be improbable. Thus the traditional reliability techniques of redundancy and improbability do not work against security threats.

The traditional security approach to masking security faults is prevention: ei-

ther *implement* with such a high degree of rigor that vulnerabilities (exploitable bugs) do not occur, or else *design* in such a way that any implementation faults that do occur cannot manifest into failures. Saltzer&Schroder canonicalized these prevention techniques in 1975 [65] into the following security principles:

1. **Economy of mechanism:** designs and implementations should be as small and simple as possible, to minimize opportunities for security faults, i.e. avoid bloat.
2. **Fail-safe defaults:** access decisions should default to “deny” unless explicitly specified, to prevent faults due to unanticipated cases.
3. **Complete mediation:** design such that *all* possible means of access to an object are mediated by security mechanisms.
4. **Open design:** the design should not be secret, and in particular, the design should not *depend on secrecy* for its security, i.e. no “security through obscurity.”
5. **Separation of privilege:** if human security decisions require more than one human to make them, then faults due to malfeasance are less likely.
6. **Least privilege:** each operation should be performed with the least amount of privilege necessary to do that operation, minimizing potential failures due to faults in that privileged process, i.e. don’t do everything as `root` or `administrator`.
7. **Least common mechanism:** minimize the amount of mechanism common across components.
8. **Psychological acceptability:** security mechanisms must be comprehensible and acceptable to users, or they will be ignored and bypassed.

These principles have held up well over time, but some more than others. Least privilege is a spectacular success, while least common mechanism has failed to compete with an alternate approach of enhanced rigor applied to common components that are then liberally shared.

Unfortunately, these techniques also turn out to be too expensive. They are hard to apply correctly, succeeding only rarely. When they do succeed in building highly secure (invulnerable) systems, the result is so restricted and slow that it tends to fail in the market place, having been eclipsed by less secure but more featureful systems.

So in practice, Saltzer&Schroeder's techniques fail most of all for lack of being applied in the first place. Security faults are thus inevitable [25]. Survivability is then the study of how to mask security faults, and do so such that attackers cannot bypass the fault masking. Section 3 examines how security faults can be effectively masked.

3 Survivability Techniques

In Section 2 we saw that redundancy and improbability are insufficient to mask security faults against an intelligent adversary, because the adversary can deliberately invoke *common mode failures*. How then to mask unknown security faults?

Colonel John R. Boyd, USAF, defined a strategy called *OODA*: Observe, Orient, Decide, and Act [39]. These four steps describe specifically how a fighter pilot should respond to a threat, and the approach generalizes to other fields of conflict, including computer security, which plays out as follows:

1. **Observe** data that might indicate a threat.
2. **Orient** by synthesizing the data into a plausible threat.
3. **Decide** how to react to that threat.
4. **Act** on the decision and fend off the threat.

This strategy is often used in computer survivability research to build *adaptive intrusion response* systems. These systems detect intrusions (using IDS/Intrusion Detection Systems) and take some form of dynamic action to mitigate the intrusion. They precisely follow Boyd's OODA loop.

Survivability techniques vary in the *time frame* in which the intrusion detection and response occur. Boyd advocated *tight* OODA loops to "get inside" the adversary's control loop, acting before the adversary can respond. In computer systems, tight response loops have the distinct advantage of preventing the intrusion from proceeding very far, and thus prevent most of the consequent damage.

However, larger OODA loops are not without merit. Taking a broader view of intrusion events enables more synthesis of what the attacker is trying to do, producing better "orientation" (in Boyd's terms) and thus presumably better "decisions."

In the following sections, we examine survivability techniques in terms of when they mitigate potential security faults. Section 3.1 looks at design time mitigation. Section 3.2 looks at implementation time techniques. Section 3.3 looks at run time intrusion prevention. Section 3.4 looks at intrusion recovery techniques.

3.1 Design Time: Fault Isolation

Despite the problems described in Section 2, security and reliability techniques do have overlap, especially in the area of design: to minimize the scope of faults. The Principle of Least Privilege can be applied in many contexts to enhance both security and reliability, such as:

- address space protection, which isolate wild pointer process faults to that process
- microkernels, which further compartmentalize facilities such as file systems into separate processes, rather than placing them in a single monolithic kernel
- mandatory access controls, which limit the scope of damage if an attacker gains control of a process

Mandatory access controls were the major security result of the 1970s and 1980s. Firewalls were the major security result of the late 1980s and early 1990s, in effect providing mandatory access controls for entire sites at the network layer. This is a natural consequence of the shift to decentralized computing: mandatory access controls are effective when all of the users are logged in to a single large time-share system. Conversely, firewalls are effective if all of the users have their own computer on the network, and thus access must be controlled at the network level. Section 3.1.1 elaborates on mandatory access controls, and Section 3.1.2 elaborates on firewalls.

3.1.1 Mandatory Access Controls

Access controls have always been a fundamental part of secure operating systems, but the devil is in the details. One can achieve true *least privilege* [65] operation by precisely specifying permissions for *every* operation, applying Lampson's access control matrix [50] to the extreme. Unfortunately, the result is an unmanageably complex access control matrix, leading to errors in *authorization*.

Thus the goal in access control is to create an *abstraction* for access control that closely matches the anticipated usage patterns of the system, so that access control specifications can be created that are both *succinct* (easy to create and to verify) and *precise* (closely approximate least privilege).

Access control schemes originally used an abstraction of controlling interactions among *users* on a time-share system. But by the mid-1990s, most computers had become single-user: either single-user workstations, or no-user network servers that do not let users log in at all and instead just offer services such as

file service (NAS), web service, DNS, etc., and thus user-based access controls schemes became cumbersome. To that end, survivability research has produced several new access control mechanisms with new abstractions to fit new usage patterns:

- **Type enforcement and DTE:** Type enforcement introduced the idea of abstracting users into domains, abstracting files into types, and managing access control in terms of which domains can access which types [10]. DTE (Domain and Type Enforcement [2, 3]) refined this concept.
- **Generic Wrappers:** This extension of DTE [36] allows small access control policies, written in a dialect of C++, to be dynamically inserted into a running kernel. A variation of this concept [4] makes this facility available for Microsoft Windows systems, but in doing so implements the access controls in the DLLs (Dynamically Linked Libraries) instead of in the kernel, compromising the non-bypassability of the mechanism.
- **SubDomain:** SubDomain is access control streamlined for server appliances [22]. It ensures that a server appliance does what it is supposed to and nothing else by enforcing rules that specify which files each program may read from, write to, and execute. In contrast to systems such as DTE and SELinux, SubDomain trades expressiveness for simplicity. SELinux can express more sophisticated policies than SubDomain, and should be used to solve complex multiuser access control problems. On the other hand, SubDomain is easy to manage and readily applicable. For instance, Immunix Inc. entered an Immunix server (including SubDomain) in the Defcon Capture-the-Flag contest [19] in which SubDomain profiles were wrapped around a broad variety of badly vulnerable software in a period of 10 hours. The resulting system was never penetrated.

An unusual new class of access controls emerging from survivability research is *randomization*, in which some aspect of a system is deliberately randomized to make it difficult for the attacker to read or manipulate. The secret “key” of the current randomization state is then distributed only to known “good” parties [24]. To be effective, the system aspect being randomized must be something that the attacker depends on, and must also be something that can be randomized without breaking the desired functionality of the system. System aspects that have been randomized are:

- **Address space layout:** Forrest *et al* [35] introduced address space randomization with random amounts of padding on the stack, with limited effec-

tiveness. The PaX project [73] introduced the ASLR (Address Space Layout Randomization) feature, which randomized the location of several key objects that are often used by buffer overflow and `printf` format string exploits. Bhatkar *et al* [9] extended ASLR by randomizing more memory objects, but with a limited prototype implementation. Xu *et al* [86] extended in the other direction, providing a degree of address space randomization using *only* a modified program loader.

- **Pointers:** PointGuard [21] provides the exact dual of address space randomization by encrypting pointer values in memory, decrypting pointer values only when they are loaded into registers for dereferencing. Memory layout is left unchanged, but pointers (which act as the effective *lens* through which memory is viewed) are hashed so that only *bona fide* code generated by the compiler can effectively dereference a pointer.
- **Instruction Sets:** Kc *et al* [45] provide a mechanism for encrypted instruction sets. In one instance, they encrypt the representation of CPU instructions to be locally unique, so that foreign binary instructions are not feasible to inject. In another instance, they apply similar encryption to PERL keywords, so that script commands cannot be injected over e.g. web interfaces. The former is largely limited to hypothetical CPUs (other than perhaps Transmeta) while the latter is feasible as a normal feature of conventional software systems.
- **IP Space** Kewley *et al* built a system that randomizes the IP addresses within a LAN. Defenders know the secret pattern of changing IP addresses from time to time, while attackers have to guess. Defenders reported this technique as effective, while attackers reported that it was confounding when they did not know it was being used, but once discovered, the limited address space of IPs (256 for a small LAN) made the guessing attack relatively easy.

Randomization defenses are effectively cryptographic session keys, applied to implicit interfaces. Some form of defense is called for when the attacker has access to one side of the interface, i.e. the attacker supplies data to that interface, making the software processing that input subject to attack. Randomization is an effective defense where the defense is *implicit* and thus classical encrypted communication would be difficult to employ, e.g. within an address space.

Randomization is less effective when the interface in question is *explicit* because classical encryption may well be more effective. The singular advantage that network interface randomization offers over network encryption (Virtual Private

Networks) is resistance to DoS (Denial of Service) attacks: attackers can flood encrypted network ports with traffic without having any encryption keys, but it is relatively difficult to flood an unknown IP address.

3.1.2 Firewalls

Initially designed to protect LANs from the outside Internet [18], firewalls progressed to being deployed within LANs to compartmentalize them, much the way mandatory access control systems compartmentalized time-share users [7]. In recent self-identified survivability research, the Secure Computing ADF card [61, 59] isolates security breaches inside insecure PCs by providing an enforcing firewall NIC managed from somewhere other than the PC itself. Thus compromised machines can be contained from a management console.

In early 2000, a new class of attacks appeared: DDoS (Distributed Denial-of-Service) attacks, in which an attacker co-opts a large number of weakly defended computers around the Internet, and then commands them all to flood a victim's machine with traffic. These attacks are very difficult to defend against, especially for public web sites intended to allow anyone on the Internet to submit requests. Defenses against such attacks are either to trace back the origin of the attacks [66] or to attempt to filter the DDoS traffic from legitimate traffic somewhere in the network to block the flood [33, 60, 75]. Work in this area has led to commercial ventures such as Arbor Networks (www.arbornetworks.com) and Mazu Networks (www.mazunetworks.com).

The problem with DDoS defenses is that they are subject to an arms race. The detection technologies are relying upon synthetic artifacts of the bogus data that the DDoS agents are generating. As the DDoS agents become more sophisticated, the data will come to more closely resemble legitimate traffic. In principle, there is no reason why DDoS traffic cannot be made to look exactly like a heavy load of real traffic: this is a fundamental difference between DoS attacks and misuse attacks, that DoS attacks need not deviate *at all* from real traffic. When faced with DDoS traffic that is identical to real traffic, filtering will become either ineffective or arbitrary. Traceback has a better chance of success in the face of sophisticated attack, but will be labor-intensive until the Internet itself changes to support effective traceback of traffic.

3.2 Implementation Time: Writing Correct Code

The least damaging vulnerability is one that never ships at all. To that end, many tools have been developed to help find and eliminate software vulnerabilities during the development cycle. Three classes of tools have emerged:

- **Syntactic checkers:** These tools scan the program source code looking for syntactic patterns associated with vulnerabilities, e.g. calls to unsafe functions like `gets ()` [78, 83].
- **Semantic checkers:** These tools do deeper analysis of program semantics looking for vulnerabilities such as buffer overflows and TOCTTOU (Time of Check to Time of Use) vulnerabilities [80, 17, 69].
- **Safer Language Dialects:** These are variants of the C programming language intended to make it safer to write programs by making it more difficult to write vulnerable code, e.g. by removing dangerous features like pointer arithmetic [43, 58].

3.3 Run Time: Intrusion Prevention

Regardless of fault containment designs, some code must be trusted, and regardless of the implementation techniques used, some vulnerabilities (bugs) will slip through, and so vulnerabilities are inevitable. Thus something must be done to detect the exploitation of these vulnerabilities and (hopefully) halt it. The reliability community calls this *fail-stop* behavior (assuming that the failure is detected). The survivability research community calls it *intrusion detection* and *adaptive response*. The security commercial sector, which has recently become interested, calls it *intrusion prevention*.

There are several dimensions in which intrusion prevention techniques can be classified. We present a 3-dimensional view, intended to classify together technologies that achieve similar goals:

- **Network vs. Host:** Intrusion prevention can be done either at the network layer or within the host. Network intrusion detection is much easier to deploy, but because there is little context in network traffic, it is possible for attackers to evade network intrusion detection methods [63, 70].
- **Detection vs. Prevention:** Some tools only detect intrusions, while others respond to intrusion events and shut the attackers down (closing the OODA loop). Prevention is effectively detection + response.
- **Misuse vs. Anomaly:** Some systems characterize and arrest known system misuse and allow everything else (*misuse detection*) while others characterize normal system behavior and characterize *anomalous* behavior as an intrusion. Misuse detection is fast & accurate, but fails to detect *novel* attacks. Anomaly detecting can detect novel attacks, but is subject to a high *false positive* rate (complaints about traffic that is actually legitimate) [53].

Populating this array of properties, we get:

- **Network**

- **Detection**

- * **Misuse:** This area is dominated by commercial NIDS (Network Intrusion Detection) products such as the commercial ISS RealSecure and the open source SNORT.
 - * **Anomaly:** The ability to detect novel attacks has generated keen interest in this area of research [51, 77], but little of it has had real-world impact due to high false positive rates. Industrial applications of intrusion detection demand *very* low false-positive rates, because non-trivial false-positive rates combined with high bandwidth lead to high staffing requirements.

- **Prevention**

- * **Misuse:** Network misuse prevention emerged as a commercial market in 2002, taking the more reliable parts of network intrusion detection systems and placing them *in-line* with the network connection, acting much like an application-level firewall with misuse prevention rules. Example systems include the Hogwash and Inline-SNORT NIPS (Network Intrusion Prevention Systems).
 - * **Anomaly:** Network anomaly prevention is a new way of looking at classic firewalls, which permit traffic with specified source and destination IP addresses, ports, and protocols, and deny all other traffic.

- **Host**

- **Detection**

- * **Misuse:** This area is referred to as HIDS (Host Intrusion Detection System) typified research projects such as EMERALD [62], STAT [79]. These systems actually use a combination of misuse and anomaly detection. Commercial HIDS similarly use a combination of anomaly and misuse detection, and also provide for both detection and prevention, as exemplified by products such as Zone Alarm and Norton Personal Firewall.
 - * **Anomaly:** There are a variety of ways to do host anomaly detection, depending on which factors are measured, and how “normal” and “anomalous” are characterized. Forrest *et al* [34] monitor sequences of system calls, ignore the arguments to system calls, and

look for characteristic n -grams (sequences of length n) to distinguish between “self” (normal) and “non-self” (anomalous). Eskin *et al* [31] generalize this technique to look at dynamic window sizes, varying n . Ghosh *et al* [38] use machine learning to characterize anomalous program behavior, looking at BSM log records instead of system call patterns. Michael [57] presents an algorithm to find the vocabulary of of Program Behavior Data for Anomaly Detection, so as to substantially reduce the volume of raw, redundant anomaly data to be considered. Tripwire [47, 42] does not look at program behavior at all, and instead detects changes in files that are not expected to change, based on a checksum; files are profiled in terms of their probability of change, so that e.g. changes in `/var/spool/mail/jsmith` are ignored (e-mail has arrived) while changes in `/bin/login` are considered very significant.

– Prevention

- * **Misuse:** The most familiar form of host misuse prevention is antivirus software that scans newly arrived data for specific signatures of known malicious software. However, this very narrow form of misuse is also very limited, in that it must be constantly updated with new virus signatures, a limitation made obvious every time a virus becomes widespread before the corresponding signature does, causing a viral bloom of Internet mail such as Melissa [13], “I Love You” [14], or Sobig.F [16]. Host misuse prevention can be either provided by the environment, or compiled in to software components.
 - **Kernel:** In the kernel environment, an exemplary system is the Openwall Linux kernel patch [28] which provides both a non-executable stack segment to resist buffer overflow attacks, and also prevents two pathological misuses of hard links and symbolic links. PaX [73] generalizes Openwall’s non-executable stack segment to provide non-executable heap pages by manipulating the TLB, which is otherwise problematic on x86 CPUs. RaceGuard [23] detects violations of the atomicity of temporary file creation to fend off temporary file race attacks.
 - **Library:** At the library level, Libsafe [5] provides a version of `glibc` that does plausibility checks on the arguments to string manipulation functions to ensure that buffer overflows and format string attacks [74] do not corrupt the caller’s acti-

vation records.

- **Compiled In:** Compiled into programs, StackGuard [26] and derived work [32, 11] compile C programs to produce executables that detect attempts at “stack smashing” buffer overflow attacks that try to corrupt activation records to hijack the victim program, and fail-stop the program before the attack can take effect. PointGuard [21] generalizes this concept to provide broader coverage by encrypting pointers in memory and decrypting pointer values only when they are loaded into registers for dereferencing. FormatGuard [20] provides a similar protection against `printf` format string vulnerabilities.
- * **Anomaly:** Operating systems can profile user or application behavior, and arrest anomalous deviations. Similar to network anomaly prevention, this induces sufficient false positives that it is not often deployed. For instance, to address the limitations of antivirus filters described above in Host Misuse Prevention, various survivability research projects [4, 40, 67] have built systems to limit the behavior of workstation applications that process network input (e.g. mail clients and web browsers) to minimize potential damage if the application is hijacked. Mandatory access controls (MAC) also fit in here (see section 3.1.1) by using the MAC system to describe what applications or users may do, and then prohibit everything else.

Many of the above technologies require modification to the operating system kernel to be effective. Because of that need, the open source license and wide popularity of the Linux kernel make it a popular target for survivability research. Unfortunately, the result of a research project that customizes the kernel to include survivability features is a “private fork” which is easy enough for researchers to build for their own use, but not sufficiently convenient for IT workers to deploy.

To address this need, the Linux Security Modules project (LSM [85, 84]) was built to provide a sufficiently rich loadable kernel module interface that effective access control modules could be built without needing to modify the standard Linux kernel. LSM has now been accepted as a standard feature, first appearing for production in Linux 2.6. As a result, IT workers should be able to obtain survivability-enhancing modules and load them into the standard kernels they get with commercially distributed Linux systems.

Finally, we return to Boyd’s OODA Loop. In the above technologies, those marked as “prevention” provide their own built-in intrusion mitigation mechanisms (usually fail-stop) and thus provide a very tight OODA loop. Those marked as

“detection” need to be composed with some other form of intrusion mitigation to actually be able to enhance survivability.

This longer OODA loop sacrifices the responsiveness that Boyd so highly prized, in favor of more sophisticated analysis of intrusion events, so as to gain greater precision in discerning actual intrusions from false-alarms due to subtle or ambiguous intrusion event data. For instance, IDIP [68] provides infrastructure and protocols for intrusion sensors (network and host IDS) to communicate with analyzers and mitigators (firewalls embedded throughout the network) to isolate intrusions.

The CIDF (Common Intrusion Detection Framework) project was a consortium-effort of DARPA-funded intrusion detection teams to build a common network language for announcing and processing intrusion events. CIDF tried to provide for generality using Lisp-based S-expressions to express intrusion events. Unfortunately, CIDF was not adopted by intrusion detection vendors outside the DARPA research community. Subsequent attempts to build IETF [8] standards for conveying intrusion events have yet to achieve significant impact.

3.4 Recovery Time: Intrusion Tolerance

An explicit goal of survivability research is to build systems that are *intrusion tolerant*: that an intruder may partially succeed in compromising privacy, integrity, or continuation of service, and yet the system tolerates this intrusion, carrying on providing critical services at a possibly reduced rate, rather than catastrophically failing all at once.

The classic reliability approach to *fault* tolerance is redundancy, so that when one replica fails, another can take up its load until repairs are made. However, as discussed in Section 2, the problem here is that while hardware faults are *independent*, the *vulnerabilities* that lead to security faults (intrusions) are largely *not* independent. Thus the attacker can sweep through an array of redundant services, quickly compromising all replicas.

Therefore, to provide intrusion tolerance, something must be done to prevent the attacker from quickly compromising all replicas, by ensuring that they do not all share the same vulnerabilities. *N*-version programming, where independent teams are given identical specifications to implement, is a classic fault tolerance technique, and would seem to provide for sufficiently diverse implementation to prevent common vulnerabilities. Unfortunately, *N*-version programming suffers from several severe limitations:

- It is *very* expensive, multiplying software development costs nearly by *N*.

- Theoretically independent software teams unfortunately tend to make similar (and wrong) assumptions, leading to common bugs [48]. Independent implementations of TCP/IP stacks and web browsers have often been shown to have common security vulnerabilities. For instance, nearly all TCP/IP implementations crashed when presented with a datagram larger than 64KB in size (the “Ping of Death”) because most developers read the IP specification, which clearly states that all packets have a maximum size of 64KB, and few thought to check for over-sized packets.

A related approach is to exploit *natural diversity* by deploying a redundant array comprised of e.g. a Windows machine, a Linux machine, and a FreeBSD machine. Natural diversity has the advantages of lower cost (because it leverages existing diversity instead of paying for additional redundant effort) and sharing fewer common design and implementation vulnerabilities, because the independent teams did not share a specification. Conversely, natural diversity has no design assurance that diversity has been *delivered* in key areas, e.g. many naturally diverse systems abruptly discovered that they depended on the same version of `zlib` (a data compression library) when it was discovered to be vulnerable [15].

Several survivability projects [54, 44, 59, 87] employ the natural diversity defense. They share a common architecture, in which services are replicated across *heterogeneous* servers, e.g. one Linux server, one Windows server, and one BSD server. A router or load balancer of some kind sits in front of the replicas, routing service requests to one or more of the replicas. The hope is that while one flavor of server might be vulnerable, that the vulnerability will not be common to all of the replicas. The system may use some form of intrusion detection to detect infected servers and remove them from the replica pool. The system may use some form of consensus voting to determine correct results, and to remove minority-view servers from the pool as infected.

Often omitted from such systems is a mechanism to share dynamic state across the replicas in a consistent and coherent manner, so that they can provide active content. This is an important issue, because providing static content in a survivable fashion is relatively simple: burn it to read-only media, and reboot periodically.

Another problem with natural diversity is that it is *expensive*, which is why the IT industry is moving in the opposite direction, consolidating on as few platforms as possible to reduce costs. Heterogeneity costs money for several reasons:

- The site must employ staff with expertise in each of the heterogeneous systems present. Many systems administrators know only a few systems, and those that know many systems tend to be senior and cost more.

- The site must patch each of these systems, and to the extent that vulnerabilities are not common, the patching effort is multiplied by the number of heterogeneous systems present. One study [41] found that a site with an infrastructure of nine NT servers and eight firewalls, for example, would have needed 1,315 updates during the first nine months of 2001.
- Multiple versions of applications must be purchased or developed, incurring additional capital and support expenses.

So while heterogeneity can be effective at providing survivability, it comes at a substantial cost. It is not yet clear whether the costs of heterogeneity outweigh the benefits. However, this problem is not specific to the heterogeneity defense; actual survivability is difficult to assess regardless of the methods employed. Section 4 looks at survivability evaluation methods.

4 Evaluating Survivability

The security assurance problem is “How can I tell if this system is secure?” To solve that, one must answer “Will this program do something bad when presented with ‘interesting’ input?” Unfortunately, to solve that one must solve Turing’s Halting Problem [76], and Turing’s theorem proves that you cannot, in general, write a program that will examine arbitrary other programs and their input and determine whether or not they will halt.

Thus in the fully general case, the security assurance problem cannot be statically decided automatically, and so other means must be employed to determine the security assurance of a system. Because determining the *actual* security of systems is so problematic, security standards such as the TCSEC (“Orange Book”) and Common Criteria turned instead to documentation of how hard the developers *tried* to provide security, by verifying the inclusion of security enhancing features (access controls, audit logs, etc.) and the application of good software engineering practice (source code control, design documentation, etc.) and at higher levels of certification, a degree of testing.

The question of “How survivable is this system?” is even more problematic, because the survivability question entails assuming bugs in the software, further weakening assumptions on which to base assurance arguments. Section 4.1 looks at studies to evaluate survivability through formal methods, and Section 4.2 looks at empirical evaluations of survivability.

4.1 Formal Methods

Knight *et al* [49] investigated the problem of a rigorous definition of “survivability.” They concluded a system is “survivable” if it “meets its survivability specification” Since most systems are not formally specified in the first place, and some of the most pressing survivability problems occur in systems largely written using informal languages such as C [46], Knight’s survivability metric is not yet of practical use, and there is a great deal of work yet to be done before we can specify just how “survivable” a given system really is.

Similarly, Dietrich and Ryan [29] find that survivability is, by definition, poorly defined, and therefore ill-suited to formal methods of evaluation. They observe that survivability is a relative property, not an absolute, and so must be evaluated in the context of “more survivable than what?”

Gao *et al* [37] propose a survivability assessment method that models dependencies of components on one another, with the mission objective as the root. They can then determine which component failures will lead to a failure of the mission. The limitation of this approach, apart from the cost of constructing such a model for large systems, is that for many practical systems, the model would quickly indicate that exploiting a failure in a trusted software component can compromise the mission, that a very large fraction of the software is trusted, and thus the survivability of the system against security attack reduces to the probability of exploitable vulnerabilities in a large software base, which is hard to assess.

Many survivability systems depend critically on the quality of intrusion detection. Tan and Maxion [72] present a thorough statistical analysis that highlights the pitfalls of statistical anomaly detection, especially with respect to the quality of the “good” training data. They show that emergent values from previous anomaly training studies such as Forrest *et al*’s claim that 4 to 6 events is an optimal window of events to consider [34] are in fact properties of the training data, and not constants at all. This brings into question whether such anomaly training can be effective in the face of intelligent adversaries, as highlighted by Wagner and Soto’s Mimicry attack [81].

4.2 Empirical Methods

The limitations of formal methods of assuring survivability make empirical methods of measuring survivability attractive. Unfortunately, for reasons similar to Turing’s Halting problem above making static security assurance problematic, it is also not possible to purely test for security. Software testing is comprised of testing probable inputs, random inputs, and boundary conditions. This is inadequate for security testing, because *obscure* input cases can induce security faults and go

undetected for many years, as exemplified by `printf` format string vulnerabilities [20]. Just as Turing shows that you cannot write a static analyzer to determine whether a program will behave badly when given arbitrary input, you cannot test for whether a program will behave badly when given arbitrarily bad input.

So rather than attempt to exhaustively test software for vulnerability, security testing takes the form of measuring the *attacker's work factor*; how much effort must the attacker apply to break the system. *Red team* experimentation (also known as *ethical hackers*) is where a would-be defender deliberately hires an adversarial team to attempt to break into the defender's systems. Red teams and actual attackers use largely the same methods of attack. The critical differences are:

- That red teams can be given boundaries. Defenders can ask red teams to attack only selected subsets of the actual system (e.g. don't attack the production payroll system on payday) and expect these boundaries to be respected.
- That red teams will explain what they have done. Actual attackers may leave an amusing calling card (web site defacement) but they often do not explain the exact details of how they succeeded in compromising security, making it rather expensive to first discover and then repair the holes. Professional red teams, in contrast, will report in detail what they did, allowing defenders to learn from the process.

DARPA funded red team experimentation on the effectiveness of the DARPA-funded survivability technologies. For example, Levin [52] describes several red team experiments testing the validity of a scientific hypothesis. Ideally the experiment should be repeatable, but that is problematic: Levin found that it is important to set the goals of the experiment and the rules of engagement clearly, because undocumented claims and attacks cannot be validated.

The results are very sensitive to what the red team knows about the defender's system at the time of the experiment: if the defender's system is obscured, then the red team will spend most of their time discovering what the system is doing rather than actually attacking it. While this is arguably similar to the situation faced by actual attackers, it is an expensive use of the red team's time, because it can be fairly safely assumed that actual attackers will be able to learn whatever they want about the defender's system. The experimental results also depend heavily on the skills of the red team, which are hard to reproduce exactly: a different team will assuredly have a different set of skills, producing different rates of success against various defenses.

An alternate approach to red team experimentation is symmetric hacker gaming, in which the individual designated attacker and defender teams of a classical red team engagement are replaced by a handful of teams that are competitively set

to attack each other while simultaneously defending themselves [19]. Each team is required to maintain an operational set of network services and applications, and a central score-keeping server records each team's success at keeping services functional, as well as which team actually "owns" a given server or service. This symmetric threat model tends to reduce disputes over the rules of engagement, because all teams are *equally* subject to those rules. This results in a nearly no-holds-barred test of survivability.

We entered an Immunix server in the Defcon "Root Fu" (nee "Capture the Flag") games in 2002 and 2003, with mixed results. In both games, we placed 2nd of 8 teams. In the 2002 game, the Immunix machine was never compromised, but it did take most of the first day to configure the Immunix secure OS such that it could earn points, because the required functionality was obscured, being specified only as a reference server image that did provide the required services, but was also highly vulnerable. However, the Immunix server was also DoS'd to the point where it no longer scored points; in retrospect not very surprising, as Immunix was designed to prevent intrusion, not DoS. The 2003 game explicitly prohibited DoS attacks, but teams deployed DoS attacks anyway. So even in symmetric red teaming, rules of engagement matter, if the rules are not uniformly enforced.

Another form of empirical testing is to measure the precision of intrusion detection using a mix of test data known to be either "good" or "bad." DARPA sponsored such a study in the 1998 MIT/Lincoln Labs intrusion detection test [53, 56]. The goal of DARPA's intrusion detection research was to be able to detect 90 (including especially *novel* attacks) while reducing false positive reports by an order of magnitude over present intrusion detection methods.

The results were mixed. False positive rates were significantly lowered versus previous generations of technologies, but were still high enough that intrusion detection still requires significant human intervention. Worse, the detectors failed to detect a significant number of novel attacks. Only a few of the tested technologies were able to detect a few of the novel attacks.

Yet another aspect of empirical measurement is to examine the behavior of attackers. This behavior matters because survivability is essentially fault tolerance against the faults that attackers will induce, and so expectations of survivability need to be measured against this threat. Browne, Arbaugh, McHugh and Fithen [12] present a trend analysis of exploitation, studying the rates at which systems are compromised, with respect to the date on which the vulnerabilities in question were made public. This study showed that exploitation spikes not immediately following *disclosure* of the vulnerability, but rather after the vulnerability is *scripted* (an automatic exploit is written and released).

Our own subsequent study [6] statistically examined the relative risks of patching early (risk of self-corruption due to defective patches) versus the risk of patch-

ing later (risk of security attack due to an unpatched vulnerability) and found that approximately 10 days after a patch is released is the optimal time to patch. However, the gap between the time a vulnerability is disclosed and when it is scripted appears to be closing rapidly [82] and so this number is expected to change.

5 Related Work

The field of Information Survivability dates back to the early 1990s, when DARPA decided to take a fresh approach to the security problem. In a field so young, there are not many survey papers. In 1997, Ellison *et al* [30] surveyed this emerging discipline, which they characterize as the ability of a system to carry out its mission while connected to an *unbounded network*. An unbounded network is one with no centralized administration, and thus attackers are free to connect and present arbitrary input to the system, exemplified by the Internet. They distinguish survivability from security in that survivability entails a capacity to recover. Unfortunately, they were *anticipating* the emergence of recovery capability, and that capability has yet to effectively emerge from survivability research. Self recovery capacity remains an area of strong interest [55]. They distinguish survivability from fault tolerance in that fault tolerant systems make failure statistically improbable in the face of random failures, but cannot defend against coincident failures contrived by attackers, as described in Section 2.

Stavridou *et al* [71] present an architectural view of how to apply the techniques of fault tolerance to provide intrusion tolerance. They propose that individual components should be sufficiently simple that their security properties can be formally assured, and that the entire system should be multilevel secure (so security faults are isolated) as in Section 3.1.

In 2000 we surveyed *post hoc* security enhancement techniques [24] which subsequently came to be known as intrusion prevention. This survey considered *adaptations* (enhancements) in two dimensions:

- **What** is adapted:
 - **Interface:** the enhancement changes the interface exposed to other components.
 - **Implementation:** the enhancement is purely internal, nominally not affecting how the component interacts with other components.
- **How** the enhancement is achieved:
 - **Restriction:** the enhancement *restricts* behavior, either through misuse detection or anomaly detection (see Section 3.3).

- **Randomization:** the enhancement uses natural or synthetic diversity to randomize the system so as to make attacks non-portable with respect to the defender’s system (see Section 3.4).

This two-dimensional space thus forms quadrants. We found that effective techniques exist in all four quadrants, but that in most cases, restriction is more *cost-effective* than randomization. Interestingly, we found that it is often the case that when one goes looking for a randomization technique, one finds a restriction technique sitting conceptually beside the randomization technique that works better: if a system attribute can be identified as something the attacker depends on, then it is better to restrict the attacker’s access to that resource than to randomize the resource.

We also conducted a similar study with narrower focus, examining buffer overflow attacks and defenses [27]. Responsible for over half of all security faults for the last seven years, buffer overflows require special attention. A buffer overflow attack must first arrange for malicious code to be present in the victim process’s address space, and then must induce the victim program to transfer program control to the malicious code. Our survey categorized attacks in terms of how these objectives can be achieved, defenses in terms of how they prevent these effects, and summarized with effective combinations of defenses to maximize coverage.

6 Conclusions

Reliability methods (redundancy, fault injection, etc.) provide protection against independent faults. Security methods (least privilege, type checking, etc.) defend against malicious exploitation of software design and implementation defects.

Survivability methods combine these two techniques to provide survivability of systems against combinations of accidental and malicious faults, and also to defend against failures in security systems. Survivability techniques defend against security faults occurring at various times throughout the software life cycle, from design time, to implementation time, to run time, to recovery time.

References

- [1] Ivan Arce. Woah, please back up for one second. <http://online.securityfocus.com/archive/98/142495>, October 31 2000. Definition of security and reliability.

- [2] Lee Badger, Daniel F. Sterne, and et al. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995.
- [3] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the USENIX Security Conference*, 1995.
- [4] Robert Balzer. Assuring the Safety of Opening Email Attachments. In *DARPA Information Survivability Conference and Expo (DISCEX II)*, Anaheim, CA, June 12-14 2001.
- [5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *2000 USENIX Annual Technical Conference*, San Diego, CA, June 18-23 2000.
- [6] Steven M. Beattie, Crispin Cowan, Seth Arnold, Perry Wagle, Chris Wright, and Adam Shostack. Timing the Application of Security Patches for Optimal Uptime. In *USENIX 16th Systems Administration Conference (LISA)*, Philadelphia, PA, November 2002.
- [7] Steven M. Bellovin. Distributed Firewalls. *login.*, 24, November 1999.
- [8] J. Bester, A. Walther, M. Erlinger, T. Buchheim, B. Feinstein, G. Mathews, R. Pollock, and K. Levitt. GlobalGuard: Creating the IETF-IDWG Intrusion Alert Protocol(IAP). In *DARPA Information Survivability Conference and Expo (DISCEX II)*, Anaheim, CA, June 12-14 2001.
- [9] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Approach to Combat Buffer Overflows, Format-String Attacks, and More. In *12th USENIX Security Symposium*, Washington, DC, August 2003.
- [10] W.E. Bobert and R.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, 1985.
- [11] Brandon Bray. How Visual C++ .Net Can Prevent Buffer Overruns. Report, Microsoft, 2001.
- [12] Hilary K. Browne, William A. Arbaugh, John McHugh, and William L. Fithen. A Trend Analysis of Exploitations. In *Proceedings of the 2001 IEEE Security and Privacy Conference*, pages 214 – 229, Oakland, CA, May 2001. <http://www.cs.umd.edu/~waa/pubs/CS-TR-4200.pdf>.

- [13] CERT Coordination Center. CERT Advisory CA-1999-04 Melissa Macro Virus, March 27 1999. <http://www.cert.org/advisories/CA-1999-04.html>.
- [14] CERT Coordination Center. CERT Advisory CA-2000-04 Love Letter Worm, May 4 2000. <http://www.cert.org/advisories/CA-2000-04.html>.
- [15] CERT Coordination Center. CERT Advisory CA-2002-07 Double Free Bug in zlib Compression Library, March 12 2002. <http://www.cert.org/advisories/CA-2002-07.html>.
- [16] CERT Coordination Center. CERT Incident Note IN-2003-03, August 22 2003. http://www.cert.org/incident_notes/IN-2003-03.html.
- [17] Hao Chen and David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings fo the ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.
- [18] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.
- [19] Crispin Cowan, Seth Arnold, Steven M. Beattie, , and Chris Wright. Defcon Capture the Flag: Defending Vulnerable Code from Intense Attack. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [20] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.
- [21] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2003.
- [22] Crispin Cowan, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, December 2000.
- [23] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

- [24] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proceedings of the 19th National Information Systems Security Conference (NISSC 2000)*, Baltimore, MD, October 2000.
- [25] Crispin Cowan, Calton Pu, and Heather Hinton. Death, Taxes, and Imperfect Software: Surviving the Inevitable. In *Proceedings of the New Security Paradigms Workshop*, Charlottesville, VA, September 1998.
- [26] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.
- [27] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000. Also presented as an invited talk at SANS 2000, March 23-26, 2000, Orlando, FL, <http://schafercorp-ballston.com/discecx>.
- [28] “Solar Designer”. Non-Executable User Stack. <http://www.openwall.com/linux/>.
- [29] Sven Dietrich and Peter Y. A. Ryan. The Survivability of Survivability. In *Proceedings of the Information Survivability Workshop (ISW 2002)*, Vancouver, BC, March 2002.
- [30] R.J. Ellison, D.A. Fisher, R.C. Linger, H.F. Lipson, T. Longstaff, and N.R. Mead. Survivable Network Systems: An Emerging Discipline. Report CMU/SEI-97-TR-013, Software Engineering Institute, November 1997. <http://www.cert.org/research/tr13/97tr013title.html>.
- [31] Eleazar Eskin, Wenke Lee, and Salvatore J. Stolfo. Modeling System Calls for Intrusion Detectin with Dynamic Window Sizes. In *DARPA Information Survivability Conference and Expo (DISCEX II)*, Anaheim, CA, June 12-14 2001.
- [32] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. <http://www.tr1.ibm.com/projects/security/ssp/>, November 21 2000.

- [33] Laura Feinstein, Dan Schnackenberg, Ravindra Balupari, and Darrell Kindred. Statistical Approaches to DDoS Attack Detection and Response. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [34] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996.
- [35] Stephanie Forrest, Anil Somayaji, and David. H. Ackley. Building Diverse Computer Systems. In *HotOS-VI*, May 1997.
- [36] Tim Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [37] Zhixing Gao, Chen Hui Ong, and Woon Kiong Tan. Survivability Assessment: Modelling Dependencies in Information Systems. In *Proceedings of the Information Survivability Workshop (ISW 2002)*, Vancouver, BC, March 2002.
- [38] Anup K Ghosh, A. Schwatzbard, and M. Shatz. Learning Program Behavior Profiles for Intrusion Detection. In *Proceedings of the First USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 1999.
- [39] Keith H. Hamonds. The Strategy of the Fighter Pilot. *Fast Company*, (59), June 2002.
- [40] Tim Hollebeek and Dur Berrier. Interception, Wrapping and Analysis Framework for Win32 Scripts. In *DARPA Information Survivability Conference and Expo (DISCEX II)*, Anaheim, CA, June 12-14 2001.
- [41] Edward Hurley. Keeping Up With Patch Work Near Impossible, January 17 2002. SearchSecurity, http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci%796744,00.html.
- [42] Tripwire Security Incorporated. Tripwire.org: Tripwire for Linux. <http://www.tripwire.org/>.
- [43] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of USENIX Annual Technical Conference*, Monterey, CA, June 2002.

- [44] James E. Just and James C. Reynolds. HACQIT: Hierarchical Adaptive Control of QoS for Intrusion Tolerance. In *Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, December 10-14 2001.
- [45] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering CodeInjection Attacks With InstructionSet Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, Washington, DC, October 2003.
- [46] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.
- [47] Gene H. Kim and E.H. Spafford. Writing, Supporting, and Evaluating Tripwire: A Publicly Available Security Tool. In *Proceedings of the USENIX UNIX Applications Development Symposium*, pages 88–107, Toronto, Canada, 1994.
- [48] John C. Knight and Nancy G. Leveson. An Experimental Evaluation of the Assumptions of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.
- [49] John C. Knight, Elisabeth A. Strunk, and Kevin J. Sullivan. Towards a Rigorous Definition of Information System Survivability. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [50] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, Princeton, NJ, 1971. Reprinted in *ACM Operating Systems Review* 8(1) January 1974 pages 18-24.
- [51] Wenke Lee, Salvatore J. Stolfo, Philip K. Chan, Elezan Eskin, Wei Fan, Matthew Miller, Shlomo Hershkop, and Junxin Zhang. Real Time Data Mining-based Intrusion Detection. In *DARPA Information Survivability Conference and Expo (DISCEX II)*, Anaheim, CA, June 12-14 2001.
- [52] D. Levin. Lessons Learned in Using Live Red Teams in IA Experiments. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [53] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 DARPA Off-Line Intrusion Detection Evaluation. In *Recent Advances in Intrusion Detection (RAID)*, Toulouse, France, October 2-4 2000.

- [54] Peng Liu. Engineering a Distributed Intrusion Tolerant Database System Using COTS Components. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [55] Peng Liu and Partha Pal. Workshop on Survivable and Self-Regenerative Systems, October 31 2003. in conjunction with the ACM International Conference on Computer and Communications Security (CCS-10).
- [56] John McHugh. The 1998 Lincoln Lab IDS Evaluation - a critique. In *Recent Advances in Intrusion Detection (RAID)*, Toulouse, France, October 2-4 2000.
- [57] C.C. Michael. Finding the Vocabulary of Program Behavior Data for Anomaly Detection. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [58] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, London, England, January 2002. Also available at http://raw.cs.berkeley.edu/Papers/ccured_popl02.pdf.
- [59] D. O'Brien. Intrusion Tolerance via Network Layer Controls. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [60] Christos Papadopoulos, Robert Lindell, John Mehringer, Alefiya Hussain, and Ramesh Govindan. COSSACK: Coordinated Suppression of Simultaneous Attacks. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [61] C. Payne and T. Markham. Architecture and Applications for a Distributed Embedded Firewall. In *Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, December 10-14 2001.
- [62] P. Porras and P. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 20th National Information Systems Security Conference (NISSC 1997)*, Baltimore, MD, October 1997.
- [63] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evation, and Denial of Service: Eluding Network Intrusion Detection. Report, Network Associates, Inc., January 1998. <http://www.nai.com/products/security/advisory/papers/ids-html/doc001.asp>.

- [64] John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.
- [65] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.
- [66] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Network Support for IP Traceback. *IEEE/ACM Transactions on Networking*, 9(3):226–237, June 2001.
- [67] Mathew Schmid, Frank Hill, A.K. Ghosh, and J.T. Bloch. Preventing the Execution of Unauthorized Win32 Applications. In *DARPA Information Survivability Conference and Expo (DISCEX II)*, Anaheim, CA, June 12-14 2001.
- [68] Dan Schnackenberg, Kelly Djahandari, and Dan Sterne. Infrastructure for Intrusion Detection and Response. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000.
- [69] Secure Software. RATS: Rough Auditing Tool for Security. http://www.securesoftware.com/download_rats.htm, July 2002.
- [70] Dug Song. Fragroute, May 27 2002. <http://monkey.org/~dugsong/fragroute/>.
- [71] Victoria Stavridou, Bruno Dutertre, R. A. Riemenschneider, and Hassen Saldi. Intrusion Tolerant Software Architectures. In *DARPA Information Survivability Conference and Expo (DISCEX II)*, Anaheim, CA, June 12-14 2001.
- [72] Kymie M.C. Tan and Roy A. Maxion. “Why 6?” Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.
- [73] ‘The PaX Team’. PaX. <http://pageexec.virtualave.net/>, May 2003.
- [74] “tf8”. Wu-Ftpd Remote Format String Stack Overwrite Vulnerability. <http://www.securityfocus.com/bid/1387>, June 22 2000.
- [75] Roshan Thomas, Brian Mark, Tommy Johnson, and James Croall. Net-Bouncer: Client-legitimacy-based High-performance DDoS Filtering. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.

- [76] Alan Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Math Society*, 42(2):230–265, 1937.
- [77] Alfonso Valdes. Detecting Novel Scans Through Pattern Anomaly Detection. In *DARPA Information Survivability Conference and Expo (DISCEX III)*, Washington, DC, April 22-24 2003.
- [78] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, December 2000. <http://www.digital.com/its4/>.
- [79] Giovanni Vigna, Steve T. Eckmann, and Richard A. Kemmerer. The STAT Tool Suite. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000.
- [80] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS (Network and Distributed System Security)*, San Diego, CA, February 2000.
- [81] David Wagner and Paolo Soto. Mimicry Attacks on HostBased Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, DC, October 2002.
- [82] Lawrence M. Walsh. Window of Opportunity Closing for Patching. *Security Wire Digest*, 5(66), September 4 2003. http://infosecuritymag.techtarget.com/ss/0,295812,sid6_iss82,00.html#ne%ws2.
- [83] David Wheeler. Flawfinder. <http://www.dwheeler.com/flawfinder/>, July 2 2002.
- [84] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Module Framework. In *Ottawa Linux Symposium*, Ottawa, Canada, June 2002.
- [85] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security Symposium*, San Francisco, CA, August 2002. <http://lsm.immunix.org>.

- [86] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS'2003)*, Florence, Italy, October 2003.
- [87] Yongguang Zhang, Son K. Dao, Harrick Vin, Lorenzo Alvisi, and Lorenzo Alvisi Wenke Lee. Heterogeneous Networking: A New Survivability Paradigm. In *Proceedings of the New Security Paradigms Workshop*, Cloudcroft, NM, September 2001.