On-Line Intrusion Detection and Attack Prevention Using Diversity, Generate-and-Test, and Generalization

James C. Reynolds, James Just, Larry Clough, Ryan Maglich
Teknowledge Corporation
{reynolds, jjust, lclough, rmaglich}teknowledge.com

Abstract

We have built a system for protecting Internet services to securely connected, known users. It implements a generate-and-test approach for on-line attack identification and uses similarity rules for generalization of attack signatures. We can immediately protect the system from many variants of previously unknown attacks without debilitating waits for anti-virus updates or software patches. Unique to our approach is the use of diverse process pairs not only for isolation benefits but also for detection. The architecture uses the comparison of outputs from diverse applications to provide a significant and novel intrusion detection capability. With this technique, we gain the benefits of n-version programming without its controversial disadvantages. The isolation of intrusions is mainly achieved with an out-of-band control system that separates the primary and backup system. It also initiates attack diagnosis and blocking, and recovery, which is accelerated by continual repair.

1 Introduction

A potential solution to the problem of building more secure but still affordable and timely systems is to combine Commercial-Off-The-Shelf (COTS) hardware and software with proven techniques from the fault tolerant community. COTS software and hardware can provide cheap (though unreliable) components to build information systems. Fault tolerant techniques can build reliable systems from unreliable components despite intermittent or transient faults. In fact, highly available systems have been built with this approach [1]. There have been many other explorations of fault-tolerant approaches to providing reliable systems based on COTS hardware and software [2, 3, 4, 5].

Most fault tolerant techniques work against faults that can be modeled as rare events occurring at random. The external faults that pertain specifically to computer and network security have different characteristics. These "faults," namely, computer and network attacks, can occur frequently and repeatedly. Their success depends on internal faults that are usually called vulnerabilities. These vulnerabilities are most often design, programming, or configuration mistakes, which cause software components to exhibit unintended behavior when presented with data or circumstances not foreseen by the developer or administrator. Vulnerabilities can be exploited by an attacker to obtain privileges or to inject additional errors into the system or to deny service to the system's legitimate users. An attacker can explore a series of potential vulnerabilities until successful, and when successful, he can repeatedly use the exploit that provided success against the same system or other systems with the same vulnerability.

These security-related faults not only can propagate from one machine to another (most dangerously, from a primary system to its backup system) but they are highly likely to repeat in time. This implies a significant additional value for fault diagnosis, including machine learning techniques, and system adaptation for intrusion prevention. Early in the development of fault tolerant computing, considerable attention was paid to fault diagnosis [6, 7]. With the recognition that most software faults were Heisenbugs and the importance of failfast semantics to minimize mean time to repair, fault diagnosis became a luxury [10].

To combat the long-lived, persistent, and malicious characteristics of computer and network attacks, additional innovative techniques are required. One area of research, which has been applied to this domain by others, is the study of Byzantine faults [8]. Byzantine faults are arbitrary faults, that is, faults not defined by the fault model of the system. As such, they represent a large, indeed, infinite class of faults. A fundamental result from the study of Byzantine faults is that fault tolerance for n Byzantine faults requires $3n+1$ redundant components [9]. Therefore, implementations of the algorithms for Byzantine fault tolerance may quickly become impractical, both because of cost and because of system complexity.

Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)
0-7695-1874-5/03 $17.00 © 2002 IEEE

Our approach is to augment standard fault tolerant techniques with active defenses and design diversity. Repeatable errors are prevented by an out-of-band control system, which modifies the system security posture in response to detected errors. Where feasible, the system employs COTS-supplied design diversity (different operating systems and server applications like web servers or Data Base Management Systems). These measures are combined with fault tolerant techniques (including failure detection, failfast semantics, redundancy, and failover) to tolerate intrusions.

This paper describes the system called HACQIT ("hack-it")[1]. In the remainder of this section, we describe our assumptions and the system hardware and software architecture. Section 2 describes our attack identification and learning mechanisms. Section 3 describes a special type of detection by diversity. Section 4 describes isolation mechanisms. Section 5 describes continual repair. Section 6 contains the conclusions supported by our work so far.

1.1 Assumptions

HACQIT is not designed to be a general-purpose server connected to the Internet. Anonymous users are not allowed. All connections to the system are through a Virtual Private Network (VPN), and therefore during any period of operation the number of users is bounded. We assume that configuration or setup of the system has been done correctly, which includes the patching of all known vulnerabilities. We assume the Local Area Network is reliable, cannot be flooded, and is the only means of communications between users and the system.

An attacker can be any agent other than legitimate users who can authenticate through the VPN or HACQIT system administrators, all of whom are trusted. Attackers may be expert and well financed, perhaps by foreign intelligence agencies. They even may have prior access to HACQIT design documentation. While the financial supporters of such individuals could employ or subvert insiders, insiders are assumed to be handled by other means than HACQIT.

Finally, we assume attackers do not have physical access to HACQIT cluster hardware. Attacks must entail communication through the LAN either with software already resident on a host or new software downloaded by the attacker to that host

Our project focus is software fault tolerance. Our model of software faults is that they are either repeatable or not. Repeatable software faults include attacks (maliciously devised inputs) that exploit the same vulnerability (bug) in one of our software components. However, we do not presume to be able to divine intent, e.g., malice, so all inputs that cause repeated failures are treated the same -- they are blocked. On the other hand, we recognize that the system may fail intermittently from certain inputs, in which case we allow retry of those inputs.

1.2 Hardware and Software Architecture

The focus of this paper is the software on the HACQIT "Cluster," shown in the lower half of Figure 1 (below the "Corporate LAN"). A HACQIT cluster consists of at least five computers: a gateway computer running a commercial firewall and additional infrastructure for failover and attack blocking; two or more servers of critical applications, only one of which at any time is the primary server and, together with a designated backup, forms a "process pair;" an Out-Of-Band (OOB) machine for overall control and monitoring; and a "sandbox" for attack identification (not shown), which is connected to the OOB machine. The machines in the cluster are connected by two separate LANs.

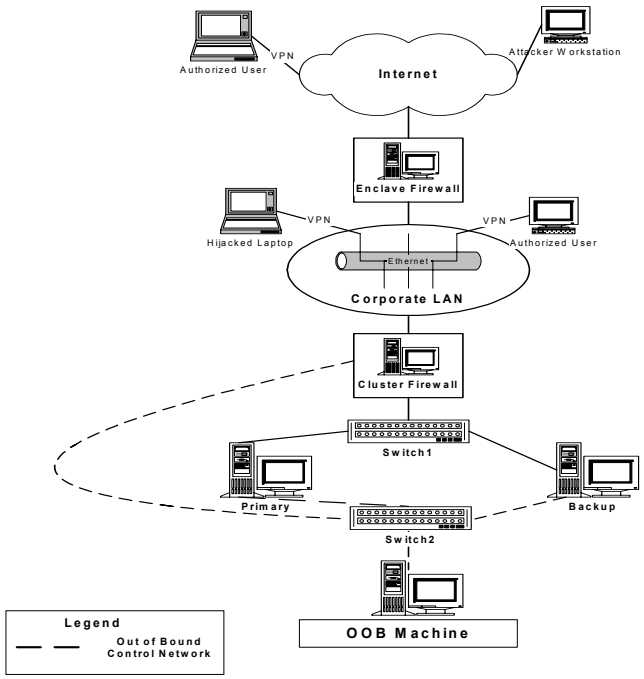

Figure 1. Intrusion tolerant hardware architecture

HACQIT uses primary and backup systems, but they are unlike ordinary primary and backup systems for fault tolerance. The two systems are isolated by the OOB computer, so no propagation of faults, for example, by a worm or an email virus, directly from the primary to the backup, is possible. (The hosts are connected by switches

[1] HACQIT stands for Hierarchical Adaptive Control of Quality of service Intrusion Tolerance, but the thrust of our project is much broader.

not hubs, and the switches are configured so that no direct communication is possible between the primary and backup.) The potential for propagation from the primary to the OOB machine is limited by sharply constraining and monitoring the services and protocols by which the OOB communicates with the primary. Failover is entirely controlled by the Mediator/Adapter/Controller (MAC) on the OOB computer. When failure is detected on the primary (possibly caused by intrusion), continued service to the end user is provided by promoting the backup to be the new primary.

The software architecture is shown in Figure 2. The system administrator controls the cluster through the Policy Editor, which runs on the OOB machine. Policy parameters determine normal and contingent behavior and responses to error conditions for all the components. Once policy has been set, it is disseminated to all components by the Policy Server on the OOB machine. Overall monitoring and control responsibility resides with the MAC.

We will discuss in detail the Forensics Agent, Sandbox, and Content Filter in Section 2. Other software components include the following:

- Web Server Protective Wrapper: This wrapper monitors calls to Dynamic Link Libraries (DLLs) for file access, process execution, memory protection changes, and other potentially malicious functions. When it detects a violation of specified behavior, it will alert, disallow, or modify the call depending on policy [16].
- Application Monitor: This software implements specification based behavior monitoring of the critical application in conjunction with the above wrapper. It starts and stops its application according to policy.
- Host Monitor: The HM communicates with the MAC and sends either heartbeats or alerts. Alerts are divided into two categories, either severe enough to cause failover or less severe, which requires further analysis by the MAC. It receives policies from the Policy Server through the Native Bridge and DLL module. The HM implements a capability to restore a failed primary to a healthy backup and is also responsible for continual repair.
- Firewall Controller: The Firewall Controller runs on the HACQIT cluster gateway machine, which hosts the firewall to the cluster. It controls whether requests/replies are sent through the switch to the primary or to the backup and uses the Suspicious Activity Monitor (SAM) Client to block and add users.

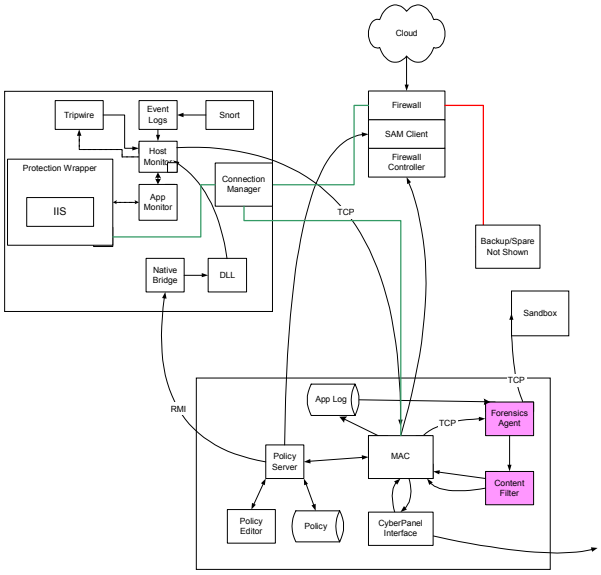

Figure 2. Intrusion tolerant software architecture

2 On-line Attack Learning and Generalization

HACQIT's on-line attack identification and generalization software provides a qualitative improvement over standard attack prevention. Currently, for most systems, two inadequate activities are performed for attack prevention. First, new attack signatures are downloaded to prevent newly discovered attacks (worms, viruses, etc.). This takes at least a few hours for humans to analyze the attacks and craft rules for the attack prevention software (anti-virus programs, firewalls, etc.) Second, patches for newly discovered vulnerabilities are downloaded and applied to critical software. This takes at least a few days for the vendor to identify the vulnerability, correct the software, and run extensive regression testing; it may also require a reboot to install the patch. Our approach provides immediate protection by automatically preventing repeated attacks and their variants without software updates.

We use on-line testing to positively identify attacks, eliminating false positives, and rule-based similarity reasoning to use a single attack to recognize a subset of attacks that exploit the same vulnerability. When a failover occurs, the MAC asks the Forensics Agent to start attack diagnosis. The FA analyzes the "App Log" containing recent requests to determine which request(s) may have caused the failure. This is the "generate" phase of the "generate-and-test" approach, one of the oldest problem-solving paradigms from Artificial Intelligence [17]. It then tests each suspicious request by sending it against a "Sandbox." The Sandbox is an exact duplicate

of the machine and application that failed. If the suspicious request causes the same conditions on the Sandbox that resulted in failover of the primary or backup, then it is identified as a "Bad Request." Through testing the identification *avoids* false positives, which makes the most common approach to unknown attack identification, anomaly detection [11, 12], so impractical. "Bad Requests" are put on a list the Content Filter uses to tell the MAC if a future request should be blocked. The FA learns attacks from failures, while the Content Filter generalizes bad requests identified by the FA, so that simple variants are also blocked. In this way, previously unknown attacks (and yet unseen variants) are automatically and immediately prevented from repeatedly causing failover.

Every time a request is received on the primary, it is forwarded to the MAC. The MAC calls the Content Filter's AllowRequest function with the new request as the argument. If there is an exact match with a "Bad Request," the function returns false to the MAC, meaning block the request. If an exact match is not found, the Content Filter analyzes the components of the requests (both new and bad) and determines if the new request is "similar" to a bad request. If it is similar, AllowRequest also returns false; otherwise, it returns true. In this way, learning is generalized from specific requests that have been identified as bad.

Similarity is rule based. The one rule currently implemented is the following: If (1) the query length of the bad request is greater than 256 and (2) the methods of the new request and the bad request are the same and (3) the file extensions of the new and bad requests are the same and (4) the query length of the new request is greater than or equal to the query length of the bad request, then return false (block the request).

With this rule, many variants of Code Red[2] I and II are blocked. The characters in the query are irrelevant to how Code Red works; the length is critical; so whether "XXX...", "NNN...", or "XNXN..." are in the query of the attack, the attack is blocked. In addition, the name of the file (minus the extension) is also irrelevant to how Code Red works, because it is the file extension that identifies the resource (Index Server) that is vulnerable to a buffer overflow, and it is the query that causes the buffer overflow, not the entire Universal Resource Identifier (URI). (The URI contains the path identifying the resource and, optionally, the query.)

Although this rule has been constructed from extensive ad hoc analysis of Code Red, it only generalizes "learned" behavior. That is, if the HACQIT cluster has never been attacked by Code Red, it will not stop the first Code Red attack. It will also not stop the first case of a variant of Code Red that uses the .IDQ extension instead of the IDA extension usually found in a Code Red attack. (Index Server uses files of types indicated by the extensions, ".IDA" and ".IDQ"; these two extensions are used by IIS to identify the Index Server resource and are passed to the Index Server interface.) This variant would first have to be "experienced," that is, learned as a bad request, and then generalized by the above rule. Most importantly, the rule does not prevent use of a resource like Index Server; it only prevents a wider variety of attacks that exploit an identified vulnerability in it from reducing availability of the web server.

It is worth comparing this automatic generalization with Snort's[3] hand-coded rules for preventing Code Red attacks. Immediately after the flurry of initial Code Red attacks, Snort aficionados began crafting rules to block these attacks. It took at least two days before rules were posted on the Snort site. These were not generalized and did not work against trivial variants. Some three months later, the rules block on ".ida" and ".idq" in the URI and "payload size" greater than 239. The use of the file extensions shows some generalization but the use of 239 as a limit on legitimate requests intended for Index Server in fact cause false positives because the payload can be much greater than 239 (at least 373) without causing the web server to fail. Our investigation with Code Red II shows the padding in the query that causes the buffer overflow is no more than five bytes over the minimum required. We think this indicates finding the minimum may not be difficult, either.

The reason for the first condition in the rule is to differentiate in a trivial way between bad requests that are buffer overflow attacks and bad requests that are some other type of attack, like remote command execution. Unfortunately, it introduces the possibility of false negatives, that is, a bad request that was a buffer overflow attack, but with the overflow occurring after less than 256 characters, would be ignored as an example to be generalized. We think the likelihood of this type of false negative is very small.

Note that although this rule seems very Microsoft-oriented, as the concept of file extensions does not exist under Unix, it should be possible to generalize this to file types under Unix. The key distinction to be made is, does the path in the URI identify a document to be returned to the requester or does it identify an executing resource such as a search engine, a DBMS, etc.? Other improvements to generalization would use analysis based on HTTP headers and body content.

One additional aspect of the design of the Content Filter software is worth discussing. Its UpdateBadReq

[2] Code Red I and II were used to successfully attack hundreds of thousands of computers in August 2001. It exploited a previously disclosed vulnerability (a buffer overflow) in Microsoft's web server, IIS, when it was used with another software component called Index Server.

[3] Snort is the most widely used Intrusion Detection System.

function uses AllowRequest in a very clever way. It first calls AllowRequest with the bad request received from the MAC. If AllowRequest returns true, that means the bad request is not on the bad request list, so it is added. If AllowRequest returns false, that means it is on the bad request list, so it is not added to the list. This prevents duplication. However, with the addition of generalization, not only will duplicates be prevented, but also trivial variants will not extend the bad request list to a performance-crippling length. As there are over 2^{1792} (or more than the number of atoms in the universe) variants of Code Red, this is an important and effective design.

3 Detection by Comparison of Outputs from Diverse Software

Although we detect intrusions through various sensors like Snort and the Windows NT/2000 Security Event Log and begin attack diagnosis for many different cases, we use diversity to provide a unique and powerful discriminator. Duplicate and compare is an old and effective strategy for hardware error detection [15]. The same input is sent to identical components and the outputs from the components are compared. If the outputs are different, an error has been detected. If the input is retried and the outputs are the same, the failure was transient; otherwise, the components must be repaired or replaced.

Typically, the duplicated components are within one single device, such as a CPU, which, in the case of permanent (hard) failure, is the unit repaired and replaced. If this is not the case, the module or system consisting of the paired components, must stop. (Following Gray and Reuter [10], we call this behavior failfast, because the delay between detecting the error and stopping must happen as quickly as possible to prevent fault propagation.) If a system built on paired components simply stops, then the redundancy has supported error detection but not failure masking, which is necessary to provide continued service despite failures.

HACQIT combines redundancy and diversity to both detect errors and mask failures for software components. This is a unique innovation for software fault tolerance. HACQIT uses two software components with identical functional specifications but different implementations. It can be seen as combining the benefits of N-version programming [13] without its controversial disadvantages [14], namely higher cost overall and lower quality components (versions).

We can do this with web server and other well known, TCP/IP based applications like mail servers, because these applications must implement a single specification, and there are already many implementations of them. The HACQIT version discussed in this paper uses two web servers, Microsoft's Internet Information Service (IIS) and the open source Apache web server, to implement this design approach. We also have evidence that multiple implementations of an identical specification do result in a small percentage of common errors [18][4].

Every HTTP request to the primary is intercepted and sent to the MAC. After checking that the request is allowed, the request is sent to both web servers, one on the primary and one on the backup, one IIS and one Apache. The HTTP specification defines the status codes with which a web server must respond. The MAC compares the responses (analogous to the outputs from duplicate hardware). If the status codes are different, the result indicates that a failure may have occurred.

In this case, failfast semantics demands failover as quickly as possible, especially because the danger from intrusion propagation is so much greater than the danger from transient fault propagation. The question, then, is how to determine which web server was successfully attacked. An analysis of HTTP status codes makes this possible.

Status codes lie in five general categories:

Status codes 100 – 199: Informational. Request received, continuing process

Status codes 200 – 299: Success. Action was received, understood and accepted.

Status codes 300 – 399: Redirection. Further action must be taken in order to complete the request.

Status codes 400 – 499: Client Error. The request contains bad syntax or cannot be fulfilled.

Status codes 500 – 599: Server Error. The server failed to fulfill an apparently valid request.

There are only ten different combinations of different status code combinations. Most are impervious to analysis or simply not indicative of an error. One meaningful combination for detection that can occur is 200/400. This means one web server responded with success while the other responded with client error. For the latter case, the cause is usually a malformed request or a request for a resource that does not exist on one machine. Either cause would indicate that the machine responding with success should be taken off-line (stopped). For example, this is the case when the well-known attack, Code Red, is sent in a request to IIS and Apache: IIS is taken over by Code Red but responds with a success code (200) while Apache is unharmed and responds with an client error code (400).

Another meaningful combination is 200/300 (success/redirection). This indicates one web server (the one responding with success) sent back different content than the other, which responded with redirection, because the client already had the content requested. In this case, the cause is assumed to be that content has changed on the first web server indicating a defacement may have occurred. That server would be taken off-line.

[4] In the study cited, for the two most diverse implementations of the POSIX specification, their errors in common were less than ten percent of the total errors.

Other combinations of different status codes are ignored. Different status codes in general are rare, including the two combinations analyzed in the preceding paragraphs. If those combinations occur but are not caused by an intrusion, service will still continue, as a primary suspected of compromise will be immediately replaced by its backup.

Besides comparing status codes, failures can be detected by other sensors in the system. For example, the HM monitors system health rules, which include detections of Quality of Service (QoS) violations to the HM, e.g., each process has a specified maximum CPU utilization percentage. The rule allows authorized processes to exceed the percentage parameter for short periods but will kill the offending process after that, including the web server, which might be appropriate if its process is taken over by an intruder.

4 Software Isolation

4.1 Diversity

As described in the section about "Detection", COTS-provided diversity is part of HACQIT's design. Not only does it help with the detection through comparison mechanism, but it also is part of our isolation strategy. Most attacks used exploits of particular vulnerabilities in a software product, either a server application like a web server or an operating system or one of its major components like the networking protocol stack. An exploit that works against one product of a type of software (web server, OS, etc.) will seldom work against another product of the same type. Consequently, as long as we have two different web servers operating on our primary and backup, an exploit that succeeds on one should not propagate to the other, even though we are passing the request that contains the exploit to both.

4.2 Random Rejuvenation

It is possible for an intrusion to become part of a legitimate process (create a new thread that lives within the process) but not do anything ("sleep") for an indefinite length of time. In this case, our detection mechanisms may not detect a failure until the malign thread(s) "wake up" and attempt to do some damage. Random rejuvenation is a counter-measure for this type of intrusion. The MAC randomly initiates a failover with the average interval between random failovers set through the Policy Editor. This should minimize the effectiveness of "stealth" attacks that sleep before they cause errors we can identify. Typically, this value should be set at a few hours or more.

5 Recovery

The wrappers on our web servers mediate every attempt to access the file system through the web server process, allowing only policy-authorized operations, for example, appending messages to the message board store, which is the system's implemented application. Thus, every file access by a wrapped process should be legitimate. Even if the process is taken over by a buffer overflow attack, the wrapper continues to mediate file access and prevent unauthorized file access. For example, Code Red I can succeed in starting a new thread within the IIS process, but it will not be allowed to read the root directory, where it looks for a file named "NOWORM."

However, we have designed HACQIT to incorporate multiple, layered defenses. In particular, we would like to detect unauthorized file accesses due to wrapper failure or other unknown vulnerabilities, to accelerate recovery. We call the overall capability (detection and correction) Continual Repair.

This is accomplished using the native security event auditing of Windows NT/2000. With this feature, the HM can detect any process creation and every access to the file system almost immediately (within a second). File accesses include reads, writes, appends, executes, lists of directories, changes to the directory, and access to file attributes. In addition, the HM knows the system files that are allowed to execute and the specific pre-existing directory of system data files they must be able to access. Finally, there are uncompromised copies of all files on a separate disk connected to the out-of-band machine.

Continual Repair performs the following actions: 1) If a file is executed which is not either (a) on the approved list of processes allowed to execute, or (b) executed by a wrapped process, it is an unauthorized process, and the HM will terminate it; 2) If an unauthorized process has created or deleted a file, the HM sets the server state as "Unhealthy" and notifies the MAC; the MAC will then set the server off-line and failover; 3) If the unwrapped process created a file, the HM will remove the file; 4) If a file is modified or deleted by other than a wrapped process, the HM will retrieve a copy of the file from the OOB machine to replace the one that was modified or deleted. The copy will be valid as of the last failover.

The last two actions precede what occurs after every failover: the wrapped processes are terminated, and an integrity check is performed on the allowed executable files and the directory of system data files. If no integrity violations are found, the MAC is notified that the server is "Healthy." When the server is promoted to on-line spare, backup, or primary, its data files are re-synchronized with the files from the previous instantiation of process pairs.

Continual Repair prevents two common goals of network attacks: to leave Trojans or "Zombies" for later exploitation or to delete necessary executable or data files

to deny service to the end users. Continual Repair does not mediate all potential changes to persistent data (files) to create a transactional file system where, if the persistent data begins in a correct state, no change violates correctness. However, it is worth noting that even with a transactional system, system delusion is possible via insider or hijacked trusted clients manipulating data. Rollback is used to mitigate this problem, but does not solve it.

6 Plans

We would like to increase the complexity of our web server based application to see if it scales to real-world use. We would like to host a second TCP/IP application, for example, email, which is inherently stateful, and see if the techniques we have developed would extend successfully. We would like to checkpoint the persistent stores for our applications using a commercial Database Management System, so that we could provide fine-grained rollback once an error in data, perhaps caused by an undetected intrusion, has been discovered.

7 Conclusion

We have developed an intrusion tolerant system for a dynamic but simple web server application. We have developed a novel technique that uses application diversity to support both detection (through comparison of status codes) and isolation (by significantly reducing the likelihood of the same attack succeeding on both the primary and backup).

We believe at least four conclusions are supported by our work:

1) Even for the general case (any server on the Internet), it is possible to prevent repeated attacks from succeeding, even when the attacks can be varied, with a combination of attack learning and generalization as part of a control loop that filters out bad requests, which have been verified using a sandbox approach. Note that the sandbox could run on a virtual machine and not require any additional hardware.
2) Diversity (or n-version programming) can in fact be effectively used for intrusion tolerance, but its main value is in detection not isolation.
3) For a bounded problem space (not an unbounded number of anonymous uses), it is possible to use techniques from the fault tolerance field, suitably modified, to increase the availability of our systems in the face of concerted cyber attacks.
4) For a problem space with less restrictive assumptions, it is possible to significantly improve on how fast and how cheaply we can recover from intrusions with the implementation of continual on-line repair.

Of course, these benefits have a cost in additional hardware, software, and administration.

8 References

1. T. Barclay, J. Gray, and D. Slutz, *Microsoft TerraServer: A Spatial Data Warehouse,* MS-TR-99-29, Microsoft Research, Advanced Technology Division, One Microsoft Way, Redmond WA 98052, June 1999.
2. H. Abdel-Shafi, E. Speight, and J. K. Bennet, "Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters," *Proceedings of the 3RD USENIX NT Symposium*, Seattle WA, July 1999.
3. J. Strouji, P. Schuster, M. Bach, and Y. Kuzmin, "A Transparent Checkpoint Facility on NT," *Proceedings of the 2nd USENIX NT Symposium*, Seattle WA, pp. 77-85, August 1998.
4. Y. Huang, P. E. Chung, C. Kintala, C. Y. Wang, and D. R. Liang, "NT-SwiFT: Software Implemented Fault Tolerance on Windows NT," *Proceedings of the 2nd USENIX NT Symposium*, Seattle WA, pp. 47-54, August 1998
5. J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," *Proceedings of USENIX Winter Technical Conference*, New Orleans LA, pp. 213-223, January 1995.
6. F. P. Preparata, G. Metze, and R. T. Chien, "On the Connection Assignment Problem of Diagnosable Systems," *IEEE Transactions on Electronic Computers,* EC16, pp. 848-854, 1967.
7. T. Dahbura, "System-Level Diagnosis: A Perspective for the Third Decade," *Concurrent Computations: Algorithms, Architecture, and Technology,* (Tewksbury, Dickson, and Schwartz, eds.) Plenum Press, Chapter 21, pp. 411-434, 1988.
8. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, New Orleans LA, February 1999.
9. L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 3, pp. 382-401, July 1982.
10. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, San Francisco, CA: Morgan Kaufmann Publishers, 1993.
11. R. A. Maxion and K. M.-C. Tan, "Benchmarking Anomaly-Based Detection Systems," *Proceedings of the International Conference on Dependable Systems and Networks,* pp. 623-630, New York NY, June 2000.
12. Schwartzbard and A. K. Ghosh, "A Study in the Feasibility of Performing Host-Based Anomaly Detection in Windows NT," *Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection,* West Lafayette IN, September 1999.

13. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *Transactions on Software Engineering,* Vol. SE-22, No. 12, pp. 1491-1501, December 1985.
14. S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of Faults in an N-Version Software Experiment," *IEEE Transactions on Software Engineering,* Vol. SE-16, No. 2, February 1990.
15. L. Spainhower and T. A. Gregg, "IBM S/390 Parallel Enterprise Server G5 fault tolerance: A historical perspective," IBM reprint, 0018-8646/99, 1999.
16. Goldman, N. M. and Balzer, R. M., "Virtual Libraries and Active Sandboxes," *Proceedings of ICSE 2000*, Limerick, Ireland.
17. Lindsay, R., B. G. Buchanan, E. A. Feigenbaum, and J. Lederberg, *Applications of Artificial Intelligence for Chemical Inference: The DENDRAL Project*, McGraw-Hill Book Company, New York, 1980.
18. Koopman, P. and J. DeVale, "Comparing the Robustness of POSIX Operating Systems," *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems (FTCS-29)*, Madison, WI, June 1999.