INOCULATING SOFTWARE FOR SURVIVABILITY

An old adage holds true for software: you can build a stronger system by first breaking it.

• ANUP K. GHOSH AND JEFFREY M. VOAS

In early 1998, several dozen computer systems in U.S. military installations and government facilities were successfully hacked, resulting in a full-scale Defense Department response now known as Operation Solar Sunrise. The attacks successfully broke into systems belonging to the Navy and Air Force as well as to federally funded research laboratories including Oak Ridge National Laboratory, Brookhaven National Laboratories, U.C. Berkeley, and MIT. Although no classified systems were allegedly compromised, the attackers were able to obtain system privileges that could be used to read password files, delete files, or create back doors for later re-entry. Despite being called "the most organized and systematic attack" to date against the Department of Defense systems by the U.S. Deputy Defense Secretary, these attacks were not the work of an organized terrorist group or nation; rather, authorities believe two northern California teenagers under the tutelage of an Israeli computer hacker were responsible for breaking into these systems, simply because they could.

Foretelling these attacks, the President's Commission on Critical Infrastructure Protection (PCCIP) announced in October of 1997 that the increasing dependence of U.S. critical infrastructures on information and communications has made them vulnerable to information warfare attacks. The commission found that while the resources needed to conduct a physical attack against these infrastructures have not dramatically changed, the resources necessary to launch a comparable-scale attack via information warfare are commonplace and consist of a personal computer and an Internet connection. Furthermore, the ubiquity of Internet access and the easy availability of hacker tools on underground Internet sites have significantly reduced both financial and intellectual barriers to launching effective attacks against critical systems.

With roughly 95% of Defense Department communications relying on commercial infrastructures, the government finds itself as a major stakeholder in the security of commercial systems and is now proposing to spend $1.46 billion in fiscal year 2000 to directly address the threat of cyberterrorism.

The federal sector is not alone, however, in its concerns over information warfare. Wholesale payment systems such as the Federal Reserve's FedWire and

automated clearinghouses (ACHs) move trillions of dollars over electronic networks. A compromise of the Federal Reserve system could dissolve trust in the electronic payments and clearing system on which all banking transactions rely. As more corporations move their business to the Internet—an inherently insecure medium—their trade and financial secrets are being exposed and placed at risk.

As societies transition to paperless commerce, individual privacy is threatened with each transaction. In short, as society becomes more "wired," the security, privacy, and integrity of information becomes paramount. Likewise, the threat of information warfare looms ever larger.

At the heart of the U.S. national information infrastructure (NII) is software. Software is used to enable the entire information infrastructure from the ubiquitous Web browser to telecommunications switching software to front-end network servers, middleware components, and back-office computing. Software is pervasive in every component that enables the information economy. The greatest risk to our NII is failing software, be it from inadvertent flaws or from malicious attacks. The most dangerous attacks against the information infrastructure are attacks against the software that comprises it.

In this article, we are concerned with the survivability of the infrastructure to software flaws, anomalous events, and malicious attacks. In the past, finding and removing software flaws have traditionally been the realm of software testing. Software testing has largely concerned itself with ensuring that software behaves correctly—an intractable problem for any nontrivial piece of software. In this article, we present "off-nominal" testing techniques that are not concerned with the correctness of the software, but with the survivability of the software in the face of anomalous events and malicious attacks. Software testing is focused on ensuring that the software computes the specified function correctly. We are concerned that the software continues to operate in the presence of unusual system events or malicious attacks.

The off-nominal testing approach uses fault injection analysis to determine the effect of unusual or malicious attacks against software. Fault injection is the process corrupting a data state during program execution. Fault injection analysis is the process of determining the effect of that corruption. The analysis may consist of simply measuring whether the corrupted state affected a particular output, or the analysis may determine whether system attributes such as safety, security, or survivability have been affected [12].

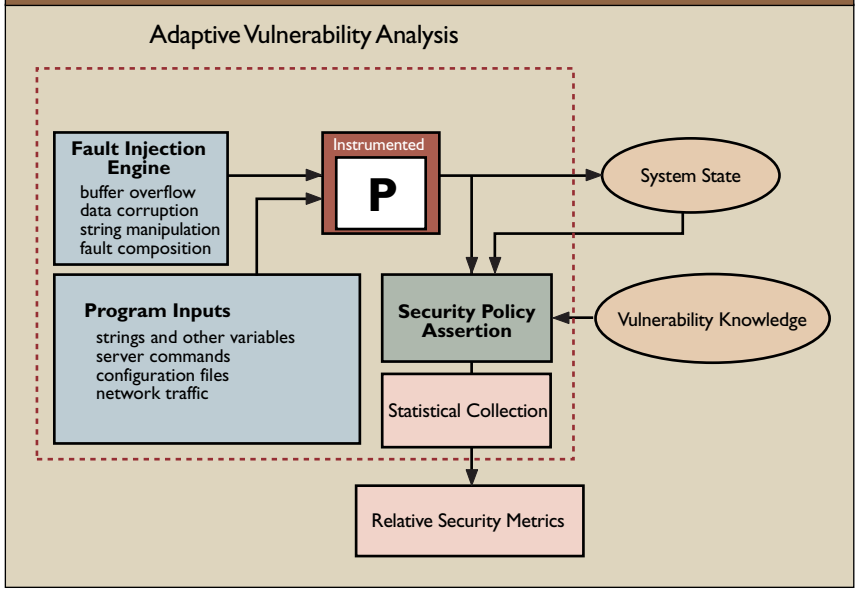
We describe two applications of fault injection analysis: one to improve the survivability of software before release and one to test the survivability of software once deployed in a fielded system. The former approach is aimed at software vendors to provide additional assurance prior to releasing software (as a complement to standard testing) that the software has been exercised under unusual conditions that might be otherwise unattainable via standard testing. Fault injection analysis is performed via software source code instrumentation in order to identify vulnerabilities in the source that can be potentially exploited to compromise system security and survivability. The results from the analysis can be used to harden the software against anomalous events or malicious attack, as we illustrate here with several case studies.

The second approach of using fault injection analysis addresses the growing need to provide assurance of survivability in systems comprised of commercial off-the-shelf (COTS) software. The purpose of using fault injection analysis is, as before, to simulate anomalous conditions that would otherwise be difficult to obtain via standard testing, and to observe the resulting effect on system survivability. Today, few systems are built from the ground up, using custom-written software components. Instead, today's software systems are a mixture of COTS software, legacy software, and custom-written software. Therefore, we must develop techniques that can provide assurance of survivability without requiring access to source code.

We describe an approach and a tool that permit assessment for how robust a software program is under anomalous system resource conditions. For instance, if the operating system throws an exception during operation, the analysis can determine a priori how robust the software is to these anomalous conditions. The discussion describes a prototype tool for testing the robustness of Windows NT application software under anomalous operating system conditions.

JAMES GARY

Figure 1. Overview of the fault injection security tool.



Developing More Survivable Systems

Two approaches to improving the survivability of the NII are to: (1) develop more survivable systems before releasing them, and (2) make fielded systems more robust and survivable. In an ideal world, software development firms would spend an appropriate amount of time and resources to developing more survivable systems. In reality, however, the commercial pressures to bring a product to market usually override concerns over providing rigorous assurance of security or survivability. As a result, little security/survivability testing is performed in software products prior to their release, in spite of the historical evidence that software flaws adversely affect system security and survivability. However, even without market pressures, there is little tool support for security and survivability-oriented testing that even the best intentioned software development firms can use.

We present an approach and a tool that support the first approach for improving the survivability of the NII, that is, enabling the development of more survivable systems by providing security and survivability assurance technologies during software development. (See [1, 3, 6–8] for other related security-oriented testing technologies.) Later in the article we describe an approach and tool for assessing and improving the survivability of fielded COTS systems. The approach recognizes that no matter how good (or inadequate) the efforts made to develop more survivable systems are, the complexity of today's systems combined with different operational environments in which software is deployed makes the deployment of perfectly survivable systems practically impossible. Thus, technologies that assess the survivability of fielded systems in a particular environment to anomalous conditions are essential for finding vulnerabilities and retrofitting software with survivable mechanisms such as software wrappers.

Fault injection originated out of testing of integrated circuits, but recent advances have allowed it to be applied to testing the safety properties of safety-critical systems. In [12], case studies of fault injection analysis are described that detected serious flaws in the safety-monitoring routine of a nuclear control application, potential safety-critical hazards in a computer-controlled surgical device, and safety-critical flaws in a metropolitan subway control system.

Having demonstrated value to safety-critical systems, fault injection analysis has since been developed to analyze security properties in security-critical software systems. Specifically, it was theorized in [11] that fault injection can be used to find locations in source code where security-related vulnerabilities might exist. The idea is simple: perform fault injection in locations throughout the software source code, and then observe whether the program exhibits insecure or non-robust behavior. Those locations in which fault injection resulted in undesirable behavior would require strengthening (or fault tolerance) to ensure that the corrupted internal states will not manifest during the actual use of the program. An application of fault injection analysis to security-critical software is described in detail in [4].

By discovering where critical flaws may be during product development using automated fault injection analysis, the opportunity to develop more survivable software systems—before damage has occurred—is afforded to software vendors. If this proactive approach is employed by vendors of critical software within the information infrastructure (for example, operating systems, system utilities, network servers and clients) one result is that, on the whole, the information infrastructure will be more survivable.

A tool for fault injection security analysis. Fault injection analysis for identifying potential vulnerabilities in software has been implemented in a working tool named the Fault Injection Security Tool (FIST). The tool automates fault injection analysis of software

using program inputs, fault injection functions, and assertions in programs written in C and C++.

A schematic diagram of FIST is shown in Figure 1. The fault injection engine provides the analyst with the ability to instrument data variables with fault injection functions. The security policy assertion component provides the ability to capture security constraints on the software and to determine if a security violation occurs during testing. As shown in Figure 1, a program, *P*, is instrumented with fault injection functions and assertions of its security policy (based on the vulnerability knowledge of the program). The program is exercised using program inputs. The security policy is evaluated online by examining program and system states. If a security policy assertion is violated during the dynamic analysis, the specific input and fault injection function that triggered the violation is identified.

When a program is loaded into FIST, it is automatically instrumented with fault injection functions to corrupt all possible data variables. Fault injection functions are instrumented by default for each different data type. For example, Booleans are corrupted to their opposite value during execution, integers are corrupted using a random function with a uniform distribution centered around their current value, character strings are corrupted using random values. In addition to these default settings, the user can programmatically instrument functions such as appending a particular string command to the end of a random string or instrumenting buffer overrun functions.

The buffer overrun function overwrites the return address of the stack frame in which the buffer variable is allocated with the address of the buffer itself. By tracing the frame pointer back through the stack, the fault injection function is able to determine where to overwrite the return address. The opcodes for machine instructions are written into the buffer being corrupted. If the program is vulnerable to a buffer overrun attack, the activation record containing the modified return address will be popped off the program stack and the program will jump to the machine instructions embedded by the fault injection function. These instructions will be executed as if they were a part of the normal operation of the program. This fault injection function can determine whether buffer variables are susceptible to buffer overrun attacks.

Once instrumented, the program is iteratively run. For each test case, a different fault injection function is triggered on each run until all test cases and all feasible fault injection functions are executed (see [4] for the algorithm). This process is automated in an iterative execution environment. The effect of a single injected fault on program security is assessed by determining which assertions fire. The specific fault injection function that triggers a particular security assertion is identified. As a result, the analyst can tie the violation of security policy to a specific line of source code that when corrupted violates security. This information allows the developer to harden the code with fault-tolerant or survivable mechanisms such as assertions or stack guards [2].

AS SOCIETY BECOMES MORE "WIRED," THE SECURITY, PRIVACY, AND INTEGRITY OF INFORMATION BECOMES PARAMOUNT. LIKEWISE, THE THREAT OF INFORMATION WARFARE LOOMS EVER LARGER.

Case studies of fault injection analysis. In a case study of fault injection analysis for software security, five common network services were analyzed [4]. Network daemons are interesting from a security standpoint because they provide services to untrusted users. Most network daemons allow connections from anywhere on the Internet, opening them up to attack from malicious users anywhere. Network daemons sometimes run with super-user, or `root`, privilege levels in order to bind to sockets on reserved ports, or to navigate the entire file system without being denied access. Successfully exploiting a weakness in a daemon running with high privileges can allow the attacker complete access to the server. Therefore, it is imperative that network daemons be free from security-related flaws that could permit untrusted users access to high privilege accounts on the server.

The programs analyzed were NCSA `httpd` version 1.5.2.a, the Washington University `wu-ftpd` version 2.4, `kfingerd` version 0.07, the Samba daemon version 1.9.17p3, and `pop3d` version 1.005h. The source code for these programs is publicly available on the Internet. Samba, `httpd`, and `wu-ftpd` are popular programs and can be found running on many sites on the Internet. The analysis of those programs was performed on a Sparc machine running SunOS 4.1.3 U. The other programs, `pop3d` and `kfingerd`, are Linux programs found in public repositories for Linux source code on the Internet. The analysis of those pro-

Table 1. Results from fault injection analysis of network daemons.				
Program	**Instrumented Locations**	**Successful Simple Corruptions**	**Successful Buffer Overruns**	**Function Coverage**
Samba v1.9.17p3	1264	12	15	45.5%
NCSA http v1.5.2a	463	27	3	40.14%
wu-ftpd v2.4	476	11	3	58.62%
pop3d v1.005h	73	2	1	63.64%
kfingerd v0.07	146	12	5	38.1%

grams was performed on a Linux 2.0.0 kernel. The programs were instrumented with both simple fault injection functions as well as the buffer overrun functions where applicable.

A summary of results from the analysis is shown in Table 1. The table shows the total number of instrumented locations together with the number of simple corruptions and buffer overrun corruptions that resulted in security violations. Clearly, the automated analysis shows a number of "trouble" spots in which fault injection functions violated the security policy of the software. In the case of buffer overruns, the security policy was simply that the program did not allow the buffer overrun function to execute its own code. In the case of the simple corruptions, the security policy involved illegal accesses to protected files.

The last column of Table 1 shows the percentage of the functions in the source code that were executed as a result of the test cases employed. Higher coverage results can be achieved through more testing and may result in more potential security hazards flushed out through the analysis.

Assessing the Survivability of COTS-based Systems

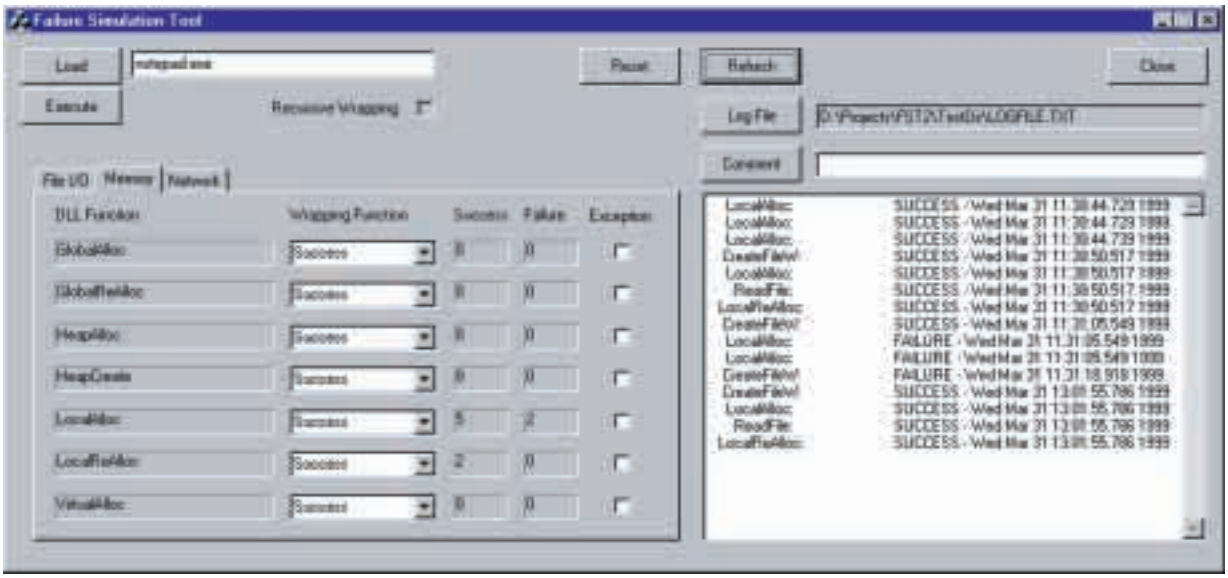
The preceding section described fault injection analysis as a viable technique for improving the survivability of software before its release. The approach, however, is not a silver bullet solution for survivability, and market pressures tend to reward quicker release cycles of products with more features, that is, complexity rather than stronger security and survivability. As a result, we cannot depend on software development and testing to produce survivable systems. Furthermore, even with the open-source software movement currently afoot, most commercial software firms are reluctant to release source code, which would permit peer review to identify bugs and vulnerabilities. As a result, we are bound by the practical constraints of commercial software releases to develop assurance technologies that can work with COTS software.

Here we describe an approach to software assurance that is designed for COTS software when the source code is not available, but executable binaries and application programming interfaces (APIs) are. The approach leverages fault injection analysis of software interfaces to analyze a critical attribute of survivability—robustness of software to anomalous events. Robustness is defined by the *IEEE Standard Glossary of Software Engineering Terminology* as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions." In order to assure survivability of deployed software systems, we are concerned with unusual or stressful conditions that often arise in the field that are rarely tested by the software vendor.

In a break from traditional dependability research, we have applied this approach to the Microsoft Windows 32-bit (Win32) platform—Windows 95, 98, NT, CE, and 2000—which represents the most popular commercial platform and is increasingly being used in critical applications. For example, under the information technology in the 21st century (IT-21) directive, the U.S. Navy requires its ships to migrate to Windows NT workstations and servers. While modernizing the fleet's technology base is appropriate, the risks of migrating to new platforms are great, particularly in mission-critical applications. A stark example of the risks is illustrated by the saga of the USS *Yorktown*, a U.S. Navy Aegis missile cruiser, which suffered a significant software problem in the NT-based systems that control the "smart ship." Reportedly, an exception thrown by the NT platform crashed the ship's propulsion system software [10]. The end result: the ship had to be towed back to the Norfolk Naval shipyard.

In order to assess the survivability of COTS-based systems, we employ fault injection analysis on the interfaces between the software application and the operating system (OS). The fault injection functions simulate the effect of failing system resources, such as memory allocation errors, network failures, file input/output (I/O) problems, as well as the range of exceptions that can be thrown by OS functions when improperly used. The fault injection analysis tests the robustness of the application to unusual, anomalous,

Figure 2. The graphical interface to the failure simulation tool that wraps Windows NT executable programs.



potentially malicious, and stressful environment conditions. An application is considered robust when it does not hang, crash, or disrupt the system in the presence of anomalous or invalid inputs, or stressful environmental conditions.

In our previous studies of the Windows platform, we analyzed the robustness of Windows NT OS functions to unexpected or anomalous inputs [5, 9]. We developed test harnesses and test data generators for testing OS functions with combinations of valid and anomalous inputs. Results from these studies show non-robust behavior from a large percentage of tested DLL functions when anomalous inputs were presented. This information is particularly relevant to application developers that use these functions. Unless application developers are building in robustness to handle exceptions thrown by these functions, their applications may crash if they use these functions in unexpected ways.

Using nominal testing techniques to test an application's robustness to exceptional OS behavior is very difficult in practice because of the difficulty in triggering exceptional OS behavior via application testing. As a result, we employ fault injection functions at the software interface between the application and the operating system to artificially trigger non-robust operating system behavior in order to assess the robustness of the application.

A failure simulation tool for Windows applications. The approach to fault injection analysis of binary executables involves "wrapping" the software's interface to the Win32 API with our own functions. The Win32 API is a set of functions standard on Windows 95, 98, NT, and CE platforms. These functions exist in dynamically linked libraries (DLLs) and represent the programmer's interface to the Windows operating system. The application's import address table (IAT), which is used to look up the addresses of imported DLL functions, is modified to point to our own wrapper DLL. When a target function is called by the application, the wrapper DLL is called instead. The wrapper DLL in turn executes, providing the ability to replace the value returned by the requested DLL function with an exception. Only exceptions that have been documented as part of the function's interface or verified through actual testing are returned by the wrapper.

A failure simulation tool has been written to enable the user to interactively fail system functions during testing. If the application crashes, then we know that the application is non-robust to these exceptions thrown by the OS function.

Figure 2 shows the graphical user interface to the failure simulation tool that allows selective failing of operating system resources. The tool allows the user to interactively fail OS functions when they are called. The panel on the right side of the image shows a log of the calls made and the failures simulated, if any, for each cell. The window shows the example memory functions that can be wrapped with failure or success functions. Other functions (such as file I/O functions) are available for instrumentation via the `System` tab shown in the window in Figure 2. The tool can be applied to any Win32 program.

Retrofitting Survivability into COTS-based Systems

By using the failure simulation tool described in the preceding section, we can determine how robust or survivable a given application is to unusual or stressful environmental conditions (such as those one might encounter in a mission-critical environment). There are two options that would increase the survivability of the vulnerable software: (1) inform the software vendor of robustness/survivability problems and hope for a patch, or (2) harden the application with software wrappers. The former option is attractive, as it fixes the problem at its source; however, the response might be less than desirable. Unless it can be demonstrated that the failure of the application occurred in a non-simulated environment, that is, a *real* mission-critical failure, *and* that the non-robustness will impact a significant number of users, it is unlikely that the vendor will rectify the problem. Of course, waiting for a mission-critical failure to occur before complaining about a problem does not ensure survivability. Thus the second option is attractive and we pursue it briefly here.

The approach leverages the wrapping method described in the preceding section. In a case in which an application is non-robust to an exception thrown by an operating system function, the program can be wrapped to make it more robust. Instead of throwing exceptions to test robustness, the wrapper will catch any exceptions thrown by OS functions and return them as a specified error code. While many programmers will not handle exceptions, the return value from a function is almost always checked for specified error values. Thus handling an exception thrown by an OS function at the wrapper and returning a specified error value is a robust way of dealing with a non-robust OS function.

Using the wrapping approach to test for robustness as described in the preceding section, the robustness of an application to error codes can be verified a priori. Therefore, the robustness of the wrapping approach to handling exceptions can be known before online deployment. The wrapper approach is particularly useful for mission-critical COTS software, when access to the source code is not available, but when robustness is important. The wrapper can be deployed with the application such that whenever the application is started, it is started with the wrapper in place.

Conclusions

The fault injection approaches we described in this article as off-nominal testing can be thought of as a means of inoculating software for survivability. The analogy to vaccinations is apt. People are inoculated against disease by injecting infectious matter into the body in nonlethal forms. The body builds appropriate antibodies to the infectious matter in order to combat future infections of a more lethal instance of the disease. In the same way, fault injection analysis injects faults into an executing program in order to determine where it is vulnerable. Unfortunately, today we do not have automatic learning systems for protecting software states, though it is the subject of ongoing research. Instead, once the program is found to be vulnerable through fault injection analysis, it can be retrofitted with fault-tolerant mechanisms in order to increase its likelihood of survivability. C

References

1. Bishop, M. and Dilger, M. Checking for race conditions in file accesses. In *The USENIX Association, Computing Systems*, Spring 1996, 131–152.
2. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P. and Zhang, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, TX, Jan. 1998), 63–78.
3. Fink, G. and Bishop, M. Property-based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes 22*, 4 (July 1997).
4. Ghosh, A.K., O'Connor, T., and McGraw, G. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, CA, May 3–6, 1998), 104–114.
5. Ghosh, A.K., Schmid, M., and Shah, V. Testing the robustness of Windows NT software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering* (ISSRE'98), (Los Alamitos, CA, Nov. 1998). IEEE Computer Society, 231–235.
6. Hamlet, R. Testing programs to detect malicious faults. In *Proceedings of the IFIP Working Conference on Dependable Computing* (Feb. 1991), 162–169.
7. Ko, C., Fink, G., and Levitt, K. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Application Conference*, (Dec. 1994, Orlando, FL), 134–144.
8. Lo, R., Levitt, K., and Olsson, R. MCF: A malicious code filter. *Computers and Security 14*, 6 (1995), 541–566.
9. Schmid, M. and Hill, F. Data generation techniques for automated software robustness testing. In *Proceedings of the International Conference on Testing Computer Software*, 1999.
10. Slabodkin, G. Software glitches leave Navy smart ship dead in the water, July 13, 1998; www.gcn.com/gcn/1998/July13/cov2.htm
11. Voas, J., Ghosh, A., McGraw, G., Charron, F., and Miller, K. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Proceedings of the 11th Annual Conference on Computer Assurance*, (June 1996), 250–263.
12. Voas, J.M. and McGraw, G. *Software Fault Injection: Inoculating Programs Against Errors*. Wiley, NY, 1998.

Anup K. Ghosh (anup.ghosh@computer.org) is Director of Security Research at Reliable Software Technologies in Sterling, VA.
Jeffrey M. Voas (jmvoas@rstcorp.com) is Chief Scientist at Reliable Software Technologies in Sterling, VA.

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) under Contracts F30602-95-C-0282 and F30602-97-C-0117. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.