Diversity against accidental and deliberate faults

Yves Deswarte, Karama Kanoun, Jean-Claude Laprie

Diversity against Accidental and Deliberate Faults

Yves Deswarte[1], Karama Kanoun and Jean-Claude Laprie
LAAS-CNRS
7 avenue du Colonel Roche
31077 Toulouse cedex 4 ÐFrance
{deswarte, kanoun, laprie}@laas.fr

Abstract

The paper is aimed at examining the relationship between the three topics of the workshops that gave rise to this book: security, fault tolerance, and software assurance. Those three topics can be viewed as different facets of dependability. The paper focuses on diversity, as a desirable approach for addressing the classes of faults that underlay all these topics, i.e., design faults and intrusion faults.

1. Introduction

The paper is aimed at examining the relationship between the three topics of the workshops that gave rise to this book: security, fault tolerance and software assurance. Those three topics can be viewed as different facets of dependability [29, 33], (see also the paper by Brian Randell in this volume). The second section is devoted to a fault classification, which identifies three major classes of faults: physical faults, design faults, (human-machine) interaction faults, where the latter two classes can be either accidental or deliberate. The classes of faults that come into play, when considering simultaneously security, fault tolerance and software assurance are the design faults and the interaction faults. Contributions of fault tolerance to security and software assurance necessitate diversity. Diversity can take place at a number of levels in a system: execution support (hardware plus operating system), execution conditions or design of the application software, human-machine interface, and operators. The third section is devoted to a close examination of these possibilities, with indications on their effectiveness with respect to the classes of faults of interest. Diversity is also commonly used for the validation of dependable systems all along its development, as presented in the fourth section. However, some faults can defeat fault-tolerance techniques (e.g., those faults resulting from tradeoffs between security and usability, or faults giving rise to common-mode failures). It is thus necessary to make an evaluation of the risk that is incurred, which is the topic of the fifth part of the paper.

[1] Yves Deswarte is currently on sabbatical at Microsoft Research, Cambridge, UK.

2. Faults [29, 33]

Faults are the adjudged or hypothesized causes of system failures, i.e. deviations from delivery of correct service to the system user(s). Faults and their sources are extremely diverse: a) their phenomelogical cause can be physical or human-made, b) they can be accidental or deliberate, with or without malicious intent, c) they can be created or occur during the system development or during its operational life, d) they can be internal or external to the system, and e) they can be permanent or transient. However, the many resulting classes of faults can be grouped into three major categories (Figure 1): physical faults (adverse physical phenomena), design faults, interaction faults (operational misuses).

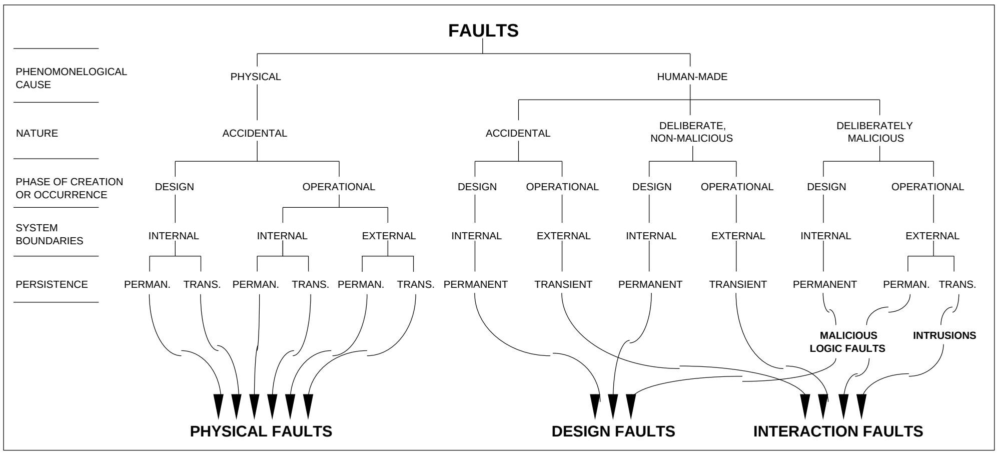

Figure 1 - Classes of faults

The causal chain from faults to failures (figure 2) involves errors, i.e. that part of system state that may lead to subsequent failure.

$$\cdots \longrightarrow \text{fault} \xrightarrow{\textit{activation}} \text{error} \xrightarrow{\textit{propagation}} \text{failure} \xrightarrow{\textit{causation}} \text{fault} \longrightarrow \cdots$$

Figure 2 - Causal chain from faults to errors, to failures

Failures can be classified according to a) their domain, i.e. value or timing, b) their perception by system users, i.e. consistent or inconsistent, usually called Byzantine, c) their consequences upon system environment, from minor to catastrophic, with usually intermediate grading, such as significant or major. From the very existence of the causal chain from faults to failures, it is relatively common usage to classify faults according to the failures they cause.

The ability to identify the activation pattern of a fault that caused one or more errors is the activation reproducibility of a fault. Faults can be categorized according to their activation reproducibility: faults whose activation is reproducible are called solid, or hard, faults, whereas faults whose activation is not systematically reproducible are elusive, or soft, faults. Most residual design faults in large, complex, software are elusive faults (or "Heisenbugs"[23]): they are subtle enough that their activation conditions depend on equally subtle combinations of internal state and external solicitation, which occur rarely and can be very difficult to reproduce.

Situations involving multiple faults and/or failures are frequently encountered. Given a system with defined boundaries, a single fault is a fault caused by one adverse physical event or one harmful human action. Multiple faults are two or more concurrent, overlapping, or sequential single faults whose consequences, i.e., errors, overlap in time, that is, the errors due to these faults are concurrently present in the system. Consideration of multiple faults leads one to distinguish a) independent faults, which are attributed to different causes, and b) related faults, which are attributed to a common cause. Related faults generally cause similar errors, i.e., errors that cannot be distinguished by whatever detection mechanisms are being employed, whereas independent faults usually cause distinct errors; it may however happen that independent faults lead to similar errors [5], or that related faults lead to distinct errors. The failures caused by similar errors are common-mode failures.

3. Diversity for fault tolerance

Many techniques commonly used in security or safety critical application domains can be identified as different means of implementing diversity for fault-tolerance. All these techniques are aimed at tolerating some kind of design faults, but they can be classified according to where they are implemented:

- at the level of users or operators,
- in the human-computer interfaces,
- at the application software level,
- at the execution level, or
- at the hardware or operating system level.

In this section, we identify these levels, and for each of them we analyze the fault classes they intend to tolerate.

Since every good concept is better if recursive, diversity can be successfully applied at diverse levels. For instance, the AIRBUS A-300/600 digital fly-by-wire system [41] is run by two classes of computers, with different microprocessors (designed independently and provided by different vendors), the application software being developed by two different companies, using different languages (and compilers), each computer being self-checking with independent channels for functional processing and monitoring. If we add the fact that a) some pilot interfaces implement diversity, b) two pilots are in the cockpit and c) the diverse computers control different axes of the plane (one controls the pitch axis and the other the roll axis), this example covers all levels of diversity implementation.

3.1. Diversity at the level of users or operators

If some privileged users or operators are not blindly trusted[2], or if they can be impersonated by some intruders, it is useful to require the cooperation of several independent users or operators to perform a sensitive operation. This can be interpreted as an application of the diversity approach to users and operators: as long as there is no

[2] In a recent survey by Ernst and Young concerning computer-related fraud involving 1200 companies in 32 countries, 66% of the surveyed companies had experienced at least one computer-related fraud in the previous 12 months; 17% of the companies had even experienced more than 5 such frauds. The survey reported that 84% of frauds were perpetrated by company employees.

common-mode failures of these independent persons (i.e., collusion), the system remains secure. On the contrary, if a single user or operator can perform such a sensitive operation, you system is insecure if this user is not perfectly trustworthy.

The *separation of duty* proposed by Clark and Wilson [13] is one implementation of user-level diversity. In the separation of duty model, complementary roles are assigned to different users, and a sensitive operation can be realized only by the successive executions of several programs that can be run only by different roles. A direct implementation of this approach can tolerate the malicious behavior of some users (at least commission faults, rather than omission faults), but no other class of faults (e.g., physical faults or software design faults, including malicious logic).

Distributing trust is another user-level diversity implementation. In this approach, sensitive operations, e.g. authentication and access control, can only be realized by running similar programs on different machines under the control of independent operators [16, 37]. In this case, the operators have similar privileges, and an operation is securely realized if at least a majority of the program copies are executed correctly. No single operator is trusted, but there is a reasonable confidence that a majority of them are not malicious. Secret sharing [40], and more generally Òthreshold cryptographyÓ can also be viewed as a cryptographic implementation of trust distribution. This approach is able to tolerate the malicious behavior of a minority of operators (omission faults as well as commission faults), but also accidental interaction faults (operator mistakes) and physical faults. Design faults are tolerated only if diversity is also applied at the application software level and/or at the support level (see Sections 3.3 and 3.5).

Trust distribution approach is efficient against malicious operators, but its efficiency against external attacks depends on the difficulty for the intruder to gain the control of a majority of the machine pool running the sensitive operation before being detected and neutralized. To increase this difficulty, it is advisable to implement also some sort of execution diversity (see Section 3.4).

Operator-level diversity is also commonly used in safety-critical applications to tolerate operator disability or mistakes. This is, for instance the reason of the presence of two pilots in commercial airplane cockpits. More generally, teamwork is often organized to prevent and tolerate independent operator errors [38].

3.2 Human-computer interface diversity

Operator-level diversity necessitates the cooperation of multiple operators. But even if only one operator is involved, diversity can be applied to human-computer interfaces to counter interaction deficiencies. These deficiencies can be caused by interface design faults, but also by possible operator inability to interact correctly[3]. Another cause of deficiency is the intrinsic inefficiency of some -human-computer interfaces. For instance, authentication mechanisms are based on something the user knows (e.g., a password), something he owns (e.g., a token) or something he is (biometric authentication). All these mechanisms have their limits: passwords can be guessed or

[3] If the operator misinterprets displayed data or is unable to enter correct information in time, this may be due to bad interface design. Accident cause analysis is the source of endless arguments on responsibility sharing between interface design faults and operator mistakes.

disclosed deliberately, tokens can be stolen or borrowed, and all biometric techniques have some false positive (authenticating the wrong person) and false negative (not authenticating the right person) rates. The most efficient authentication systems use diverse mechanisms, e.g. smartcard activated by keying a Personal Identification Number (PIN), or fingerprint matching with patterns stored on a smartcard (this kind of techniques can also ease privacy concerns raised by uncontrolled storage of biometric data).

Diverse interfaces can also provide back-up facilities to cope with physical faults affecting some parts of the interfaces.

3.3 Application software diversity

This is the most common form of diversity. The notion of software diversity has been formulated in the seventies [11, 22, 36]. It has been significantly used in safety-critical systems to provide either a fail-halt property or service continuity.

The fail-halt property, with respect to design faults, can be simply achieved by self-checking modules consisting of two parts: a functional part and a monitor part running an acceptance test based on assertions checked on input data, intermediate data or result data. Acceptance test can also be implemented by comparing two versions of the functional part. If the acceptance test detects an error, the component is isolated or halted, in order to prevent disturbance of other parts of the system.

Continuity of service can be provided by recovery blocks [36], N-version programming [4] or N-self-checking programming [30].

Application software diversity is of course primarily targeting at (accidental or deliberate) design faults in the application software. But they can be efficient also to tolerate physical faults [30], and also some hardware or operating system design faults. For instance, the ELEKTRA railway interlocking control system [28] is composed of two channels: one channel executes the interlocking control software, the other one executes the monitoring software (i.e., the safety bag). Both channels are made of identical hardware, with identical operating systems. According to the very high safety requirements, operating system and hardware cannot be considered as exempt of design faults. But, since the application programs running on the two channels are different, it can be considered as very unlikely that the same (hardware or OS) design fault can be activated at the same execution step on both channels and produce consistent errors leading to an unsafe state.

N-version programming can also be efficient against viruses, if inter-process communications are checked efficiently [26].

3.4. Diversity at the execution level

The same software run on the same hardware but with a different execution context may behave differently with respect to accidental design faults, and this kind of diversity can have a surprising high efficiency. Practical experience of Tandem systems [7, 23] has shown that rollback mechanisms designed to tolerate physical faults turn out to be equally efficient for the software faults, owing to the loose coupling between process executions: since most ÒHeisenbugsÓ appear to be context sensitive, those affecting primary execution are very unlikely to be activated during the rollback.

This kind of diversity would deserve more attention.

3.5. Diversity at the hardware and OS level

Diversity can be applied at the hardware level, for instance by designing independently different processors able to run identically the same software [3], and by comparing bit-by-bit the results. This kind of diversity has been primarily intended to tolerate hardware component failures and external physical faults: such faults are very unlikely to produce identical errors on diversely implemented hardware. But nowadays, such diversity should be still more useful to tolerate the numerous design faults induced by the increasing complexity of recent microprocessors. See for instance the Pentium Specification Updates published by Intel¨ , with many ÒerrataÓ defined as Òdesign defects or errorsÓ.

To tolerate compiler (design or execution) faults, but also to increase the execution diversity, identical application software modules can be compiled by independently developed compilers, as is the case for the Boeing 777.

Operating systems are also prone to design flaws, and diversity can help to tolerate them. In particular, it is possible to run application software replicas on different Commercial-Off-The-Shelf (COTS) operating systems. This should be especially efficient for security critical software, which could be attractive targets for intrusions (e.g., certification authorities, name and directory services, electronic commerce, etc.). Indeed, most intrusions exploit flaws in OS platforms rather than flaws in the security-related software itself. But a particular attention must be given to prevent Òcorrelated faultsÓ which could appear in several COTS OS. For example, the buffer overrun attack is common on many Unix-based systems, as well as on Microsoft Windows-NT. In some cases, such correlated faults cannot be easily avoided, e.g., when they are features of standard protocols. Such features are often exploited by denial-of-service attacks (see for instance [12]).

4. Diversity and validation

This section addresses the role of diversity in the validation process in two ways: on one hand, design diversity aids the validation of the software variants and, on the other hand, a good validation method necessitates a set of diverse verification and validation techniques.

Design diversity is generally used to check the dynamic behavior of the software during execution. In addition to its ability to detect or tolerate faults in operation, it has been observed that design diversity i) aids the validation of software variants, thanks to back-to-back testing and ii) helps in detecting certain errors that are unlikely to be detected with other methods. For example, in a controlled experiment performed in [42], 14 % of faults were detected by back-to-back testing of three variants after extensive individual tests, whereas in [8] nine faults were detected after extensive use of other testing methods. It can be argued that design diversity is costly. However, previous experiments and evaluations showed that design diversity does not double the cost (see e.g., [1, 6, 24, 31]), and more recently a study performed on an industrial software [27] confirmed that the cost of one diverse variant is between 0.7 and 0.85 the cost of a non fault-tolerant software. This is due to the fact that even though some development activities (e.g., detailed design, coding, unit and integration tests) are performed separately for each unit, several activities (e.g., specifications, high level design and system test) are performed globally and even take advantage of the existence of more than one variant.

Considering now, more generally, the validation of software systems. It is well known (and this is confirmed in the survey carried out in [32], Section 2.2.3) that despite the large number of available validation techniques (that are very valuable), none of them is perfect. The authors of this survey argue for the combination of various (i.e., diverse) techniques to obtain a high level of trust. They recommend combining static analysis with testing for all categories of software systems. For critical software, these techniques should be reinforced by formal specification, behavioral analysis and proof-of-correctness. Indeed, the diverse techniques allow different types of faults to be revealed. Static analysis (e.g., walk-through, inspection) can be applied to specifications and code and allow detection of a large number of faults before software execution: they could reveal as high as 84% and 95% of faults in the software (see respectively [10, 39]). Testing uncovers faults that static analyses have failed to reveal. Even within testing, diverse methods are recommended: for example it is worth combining functional and structural testing as well as strategies with randomly selected inputs and deterministic inputs. The behavioral analysis complements the static analysis as the dynamic properties can be verified. Formal specification and proof-of-correctness should be used for critical components. Another matter related to diversity in the validation process is the aim of the various activities. Indeed, even though the overall aim of validation is to check the software correct behavior, static analysis, behavioral analysis and testing aim at uncovering faults (they succeed when they uncover faults), whereas formal specification and proof-of-correctness aim at demonstrating the correct behavior (they fail when they uncover faults). Also, diversity within the same validation activity may be beneficial: in the experiment conducted in [21] two inspection teams, following the same rules, found different types of faults in the same software system (for each team around 75 % of faults were not found by the other team).

5. Efficiency evaluation

Diversity efficiency is mainly limited by the related faults defined in Section 2: even independently developed hardware of software can exhibit faults generating similar errors, as has been experienced by many experimental studies such as [5]. The evaluation of the impact of such faults on the overall dependability of the system can be evaluated as in [31]. This Section examines the efficiency of diversity a) on the development process and b) with respect of security/usability tradeoffs.

5.1 Efficiency of diversity on the development process

High quality software necessitates a controllable and well-defined production process as it assumed that there is a direct relationship between the quality of the development process and the quality of the resulting product. Programs aimed at improving and maturing the development process are usually referred to as reliability improvement programs. Such programs are generally based on several diverse actions: well-defined specification process, combined use of validation techniques, data collection, feedback to the development process and actions performed to improve the process, etc. The evaluation of the efficiency of these methods is needed to check their impact on the product quality and on the development process productivity and efficiency.

Indeed most of the companies that have followed a reliability improvement program have already appreciated the progress accomplished. Among these companies we can quote without being exhaustive: AT&T, Bull, Fujitsu, Hewlett-Packard, IBM, Motorola, NASA, etc. In addition to specific experiences, several papers and books have already been published advocating and defining methods and models for improving software process (see e.g., [25, 34]).

One of the major objections to reliability programs is their cost that has been considered for a long time as increasing with the required dependability level. The relationship between the level of dependability required and the associated cost is further complicated when considering factors such as the supplier's rework cost, the maintenance cost, the consequence of failure for the customer, or the cost of fault correction. For example, past experience has showed that the cost of fixing a fault uncovered during operation to be one or two orders of magnitude higher than the cost of the same fault detected during development [9].

It is worth noting that all the companies that have followed a well-defined program for improving software process and quality agree on the fact that the benefits are important. However, it is very difficult to partition the gains according to the methods used (e.g., the relative impact of fault prevention and fault removal techniques is very difficult to be assessed). Another difficulty comes from the fact that the improvement are usually evaluated by comparing the results of the new methods with respect to the previous ones used by the company. Moreover the criteria of comparison vary from one company to the other. These criteria concern for instance software productivity, fault density in the field, reduction of test duration, etc. It is thus very complicated to draw general conclusions from the published work. The examples presented below are given as indication about some benefits observed; they have to be considered within the context in which they have been obtained.

¥ The results obtained through the quality program started at AT&T's International DEFINITY PBX [19] (using among other techniques the Cleanroom approach and software reliability objectives) show: a factor-of-10 reduction in customer-reported problems, a factor-of-10 reduction in program maintenance cost, a factor-of-2 reduction in the system test interval, and a 30% reduction in new product introduction interval.

¥ The experience reported in [20] by IBM, consisting in analyzing reliability-related data (trend tests and reliability growth models) and in the application of an economic model to determine optimal release time, shows that the benefit-to-cost ratio brought by such analyses is 6.14, 11.98, and 78.65 depending on whether the cost of a failure is 500, 5,000, or 50,000 monetary units.

¥ The experience reported by Fujitsu [2], using the concurrent development approach, shows the release cycle has been reduced by 75%.

¥ The study carried out over 15 projects by Raytheon Equipment Division [18] shows that the rework cost has shrunk to a quarter of its original value after completion of a five-year program aimed at process improvement.

¥ Motorola (Government Electronics Div.) [17] has followed an improvement and an evaluation program: they went from Capability Maturity Model (see e.g., [25]) level 2 to level 3 in three years and it took them three more years to reach level 4 for the whole process and level 5 for policies and procedures. From level 2 to 5, the number of faults has been divided by 7, the cycle time by 2.4 whereas productivity has been improved by 2.8.

5.2. Efficiency of diversity with respect of security/usability tradeoffs

In Section 3, it has been shown that diversity can contribute significantly to improve security. But quite often, even if they are implemented with diversity, security mechanisms can be defeated by a careless user, e.g., who writes his PIN on his smartcard.

Indeed, most computing systems users are not motivated sufficiently by security concerns to accept easily the unavoidable discomfort induced by any security measures. It may be difficult for them to imagine that, while they may have no access to sensitive information, their careless attitude can endanger other users who may have to deal with such sensitive data. But this is really true! Many attackers are exploiting badly protected user accounts to gain enough privileges to progress towards more sensitive targets. For instance, they can easily impersonate a careless user and abuse the trust that other persons place in the impersonated user.

In fact, trust relationships should not be transitive: if Alice trusts Bob, and Bob trusts Charlie, this does not mean that Alice trusts Charlie. Nevertheless, when implemented by computer mechanisms, such trust relationships can be transitively abused. For instance, if Alice trusts Bob and grants him the privilege to access her account (e.g., by using `.rhosts` on Unix), and in the same manner, if Bob gives Charlie an access to his account, Charlie can easily access AliceÕs account. Yet Alice may not wish to deny BobÕs access to her account or Bob deny CharlieÕs access to his, if these accesses are needed to ease their work. Most operating systems exhibit such facilities, e.g. to improve teamwork or to enable some users to benefit from other usersÕexpertise. If these features are correctly used, they can even improve security (examples are given in [15]).

It is thus important to assess the influence of user behavior on the system security. A quantitative evaluation method has been developed for this purpose [14]. Measurements provided by this approach aim at representing as accurately as possible the security of the system in operation, i.e. its ability to resist possible attacks, or equivalently, the difficulty for an attacker to exploit the vulnerabilities present in the system and defeat the security objectives assigned to the system. This method is based on 1) a theoretical model, the privilege graph, exhibiting the system vulnerabilities, 2) a definition of the security objectives, 3) a mathematical model to compute significant security measures.

In the privilege graph model [15], a node represents a set of privileges owned by a user or a set of users (e.g., a Unix group) and an arc represents a vulnerability. An arc exists from node X to node Y if there is a method enabling a user owning *XÕs* privileges to obtain *YÕs* privileges. Most of these vulnerabilities are induced by lax user behavior or by features activated to facilitate information sharing. Weights are assigned to each arc, according to the effort needed for a possible attacker to exploit the vulnerability. In the privilege graph, diversity-implemented security mechanisms are represented by arcs with weights corresponding to the difficulty to break each of these mechanisms.

The security objective definition is mostly used to identify in the privilege graph which nodes represent the privileges of possible attackers and which nodes represent the privileges of possible targets. The mathematical model, based on Markov chains, is used to compute measures representing the global effort associated with all the paths that connect possible attacker node to possible target nodes. A set of software tools has been developed to generate automatically privilege graphs for distributed Unix systems, to define security objectives, and to compute significant security measures [35]. Such tools can help a security administrator to identify those security flaws that can be eliminated for the best security improvement, with the least incidence to users. These tools can also enable the administrator to monitor the evolution of the global system security according to changes in the environment, in the configurations or in the user behavior.

6. References

[1] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, ÒSoftware Fault Tolerance: An Evaluation,Ó IEEE Trans. on Software Eng., vol. SE-11, pp. 1502-1510, 1985.

[2] M. Aoyama, ÒConcurrent-Development Process Model,Ó IEEE Software, vol. July, pp. 46-55, 1993.

[3] J. Arlat, ÒDesign of a Fault-Tolerant Microcomputer by Means of Functional Diversification,Ó Doctoral Thesis. In French: INP, Toulouse, France, 1979.

[4] A. Avizienis, ÒThe N-version Approach to Fault-Tolerant Systems,Ó IEEE Trans. on Software Eng., vol. 11, pp. 1491-1501, 1985.

[5] A. Avizienis and J. P. J. Kelly, ÒFault Tolerance by Design Diversity: Concepts and Experiments,Ó IEEE Computer, vol. 17, pp. 67-80, 1984.

[6] A. Avizienis, M. R. Lyu, W. Schutz, K. S. Tso, and U. Voges, ÒDEDIX 87 - A supervisory System for Design Diversity Experiments at UCLA,Ó in Software Diversity in Computerized Control Systems, Dependable Computing and Fault-Tolerant Systems, vol. 2, U. Voges, Ed. Vienna, New York, Springer-Verlag, 1988, pp. 127-168.

[7] J. Bartlett, J. Gray, and B. Horst, ÒFault Tolerance in Tandem Computer Systems,Ó in The Evolution of Fault-Tolerant Systems (Proc. IFIP Symp. on The Evolution of Fault-Tolerant Computing, Baden, Austria, July 1986), Dependable Computing and Fault-Tolerant Systems, A. Avizienis, H. Kopetz, and J.-C. Laprie, Eds., 1 ed. Vienna, Austria, Springer-Verlag, 1987, pp. 55-76.

[8] P. G. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahl, and J. Lahti, ÒPODS Ñ A Project on Diverse Software,Ó IEEE Trans. on Software Eng., vol. 12, pp. 929-940, 1986.

[9] B. W. Boehm, Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[10] M. Bush, ÒGetting Stared on Metrics - Jet Propulsion Laboratory Productivity and Quality,Ó Proc. 12th International Conference on Software Engineering, Nice, France, 1990, pp. 133-142.

[11] L. Chen and A. Avizienis, ÒN-Version Programming: a Fault Tolerance Approach to Reliability of Software Operation,Ó Proc. 8th Int. Symp. Fault-Tolerant Computing (FTCS-8), Toulouse, France, 1978, pp. 3-9.

[12] CIAC, ÒTools Generating IP Denial-of-Service Attacks,Ó US Department of Energy, Computer Incident Advisory Capability (CIAC), Bulletin I-019, 1997.

[13] D. D. Clark and D. R. Wilson, ÒA Comparison of Commercial and Military Computer Security Policies,Ó Proc. Proc. Int. Symp. on Security and Privacy, Oakland, CA, USA, 1987, pp. 184-194.

[14] M. Dacier, ÒTowards Quantitative Evaluation of Computer Security,Ó Doctoral Thesis. In French: INP, Toulouse, 1994.

[15] M. Dacier and Y. Deswarte, ÒPrivilege Graph: An Extension to the Typed Access Matrix Model,Ó Proc. European Symposium on Research in Computer Security (ESORICS 94), Brighton, UK, 1994, pp. 319-334.

[16] Y. Deswarte, L. Blain, and J.-C. Fabre, ÒIntrusion Tolerance in Distributed Computing Systems,Ó Proc. Int. Symp. on Security and Privacy, Oakland, CA, USA, 1991, pp. 110-121.

[17] M. Diaz and J. Sligo, ÒHow Process Improvement Helped Motorola,Ó IEEE Software, vol. Sept., pp. 75-81, 1997.

[18] R. Dion, ÒProcess Improvement and the Corporate Balance Sheet,Ó IEEE Software, vol. July, pp. 28-35, 1993.

[19] M. Donnelly, B. Everett, J. Musa, and G. Wilson, ÒBest Current Practice of SRE,Ó in Handbook of Software Reliability Engineering, M. Lyu, Ed., Mc Graw Hill, 1996, pp. 219-254 (Chapter 6).

[20] W. Ehrlich, B. Prasanna, J. Stampfel, and J. Wu, ÒDetermining the Cost of a Stop-Test Decision,Ó IEEE Software, vol. March, pp. 33-42, 1993.

[21] K. El Emam and I. Wieczorek, ÒThe repeatability of Code Defect Classifications,Ó Proc. 9th International Symposium on Software Reliability Engineering (ISSRE'98), Paderborn, Germany, 1998, pp. 322-332.

[22] W. R. Elmendorf, ÒFault-tolerant ProgrammingÓ, Proc. 2nd IEEE Int Symp. Fault-Tolerant Computing, Newton, Massashusetts, 1972, pp. 79-83.

[23] J. Gray, ÒWhy Do Computers Stop and What Can be Done About it ?Ó, Proc. 5th Int. Symp. on Reliability in Distributed Software and Database Systems, Los Angeles, CA, 1986, pp. 3-12.

[24] G. Hagelin, ÒERICSSON Safety System for Railway ControlÓ, in Software Diversity in Computerized Control Systems, Dependable Computing and Fault-Tolerant Systems, vol. 2, U. Vogues, Ed. Vienna, New York, Sringer-Verlag, 1988, pp. 9-21.

[25] W. S. Humphrey, Managing the Software Process: Addison-Wesley, 1989.

[26] M. K. Joseph and A. Avizienis, ÒA Fault Tolerance Approach to Computer VirusesÓ, Proc. Int. Symp. on Security and Privacy, Oakland, CA, USA, 1988, pp. 52-58.

[27] K. Kanoun, ÒCost of Software Design Diversity Ñ An Empirical EvaluationÓ, Proc. 10th Int. Symp. on Software Reliability Engineering (ISSRE'99), Boca Raton, FL, USA, 1999.

[28] H. Kantz and C. Koza, ÒThe Elektra Railway Signaling System: Field Experience with an Activity Replicated System with DiversityÓ, Proc. 25th Int. Symp. on Fautlt Tolerant Computing (FTCS25), Pasadena, CA, 1995, pp. 453-458.

[29] J.-C. Laprie, ÒDependability: Basic Concepts and Terminology,Ó in Dependable Computing and Fault-Tolerant Systems, vol. 5, E. J.C. Laprie, Ed. Wien-New York: Springer Verlag, 1992, pp. 265.

[30] J.-C. Laprie, J. Arlat, C. B□ounes, and K. Kanoun, ÒDefinition and Analysis of Hardware-and-Software Fault-tolerant ArchitecturesÓ, IEEE Computer Magazine, vol. 23 (7), pp. 39-51, 1990.

[31] J.-C. Laprie, J. Arlat, C. B□ounes, and K. Kanoun, ÒArchitectural issues in Software fault ToleranceÓ, in Software Fault Tolerance, M. Lyu, Ed., Wiley, Trends in Software, 1996, pp. 45-80 (Chapter 3).

[32] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Ka□niche, K. Kanoun, C. Mazet, D. Powell, C. Rabejac, and P. Th□venod, Dependability Handbook. in French, Toulouse, France: Cepadues - Editions, 1995.

[33] J.-C. Laprie, A. Avizienis, and B. Randell, ÒDependability of Computing Systems: Fundamental Concepts, Terminology, and ExamplesÓ, LAAS-CNRS Report: 99-293, 1999.

[34] J. Musa, Software Reliability Engineering: Computing McGraw-Hill, 1998.

[35] R. Ortalo, Y. Deswarte, and M. Ka□niche, ÒExperimenting with Quantitative Evaluation Tools for Monitoring Operational SecurityÓ, IEEE Transactions on Software Engineering, vol. 25, 1999.

[36] B. Randell, ÒSystem Structure for Software Fault ToleranceÓ, IEEE Transactions on Software Engineering, vol. SE-1, pp. 220-232, 1975.

[37] M. K. Reiter, ÒDistributed Trust with the Rampart ToolkitÓ, Communications of the ACM, vol. 39 (4), pp. 71-74, 1996.

[38] L. Rognin and B. Pavard, ÒCollective activities and ReliabilityÓ, Proc. XIII Conference on Human Decision Making and Manual Control, June 13-14, Espoo, Finland, 1994, pp. 177-187.

[39] C. Sayet and E. Pilaud, ÒAn Experience of a Critical Software DevelopmentÓ, Proc. 20th IEEE Int. Symp. on Fault-Tolerant Computing (FTC-20), Newcastle, UK, 1990, pp. 36-45.

[40] A. Shamir, ÒHow to Share a SecretÓ, Communications of the ACM, vol. 22 (11), pp. 612-613, 1979.

[41] P. Traverse, ÒDependability of Digital Computers on Boards AirplanesÓ, in Dependable Computing for Critical Applications, Dependable Computing and Fault-Tolerant Systems, vol. 1, A. Avizienis and J. C. Laprie, Eds., Springer-Verlag, 1987, pp. 133-152.

[42] U. Voges, ÒUse of Diversity in Experimental Reactor Safety SystemsÓ, in Software Diversity in Computerized Control Systems, Dependable Computing and Fault-Tolerant Systems, vol. 2, U. Voges, Ed. Vienna, New York, Sringer-Verlag, 1988, pp. 29-49.