Analysis of Techniques For Building Intrusion Tolerant Server Systems

Feiyi Wang, Raghavendra Uppalli, Charles Killian
Advanced Networking Research
MCNC Research & Development Institute
Research Triangle Park, North Carolina, NC 27709

***Abstract*— The theme of intrusion detection systems (IDS) is detection because prevention mechanism alone is no guarantee to keep intruders out. The research focus of IDS is therefore on how to detect as many attacks as possible, as soon as we can, and at the same time to reduce the false alarm rate. However, a growing recognition is that a variety of mission critical applications need to continue to operate or provide a minimal level of services even when they are under attack or have been partially compromised; hence the need for *intrusion tolerance*. The goal of this paper is to identify common techniques for building highly available and intrusion tolerant server systems and characterize with examples how various techniques are applied in different application domains. Further, we want to point out the potential pitfalls as well as challenging open research issues which need to be addressed before Intrusion Tolerant Systems (ITS) become prevalent and truly useful beyond a specific range of applications.**

I. Introduction

Intrusion tolerance research has attracted more and more attention recently. Similar to intrusion detection, it belongs to the so-called second line of defense mechanisms. It is always better to keep intruder out of door in the first place, but the goals of complete detection, and correct and timely response remain elusive. Unlike intrusion detection, intrusion tolerance is motivated by the recognition of the fact that intrusions will occur and some will be successful. There is a wide range of mission-critical applications that require continuation of services even when the system is under active attack or some of its components have been partially compromised. A major assumption of intrusion tolerance is that protected system can be faulty and compromisable, and challenge is how to tolerate faults, be it natural or malicious and to continue to provide (possibly degraded) services.

DARPA has been sponsoring intrusion tolerance research through OASIS [1] program for several years. The goal of this paper is to analyze primary techniques employed for building highly available, intrusion tolerant systems (ITS). Large pieces of the foundation of intrusion tolerance are built upon traditional fault tolerance (software and hardware) techniques such as diversity, redundancy, acceptance testing, and ballot voting etc. This paper attempts to characterize how these techniques are being applied in this new environment where malicious faults, not natural faults, dominate and what the potential pitfalls are.

This work is sponsored by DARPA under contract N66001-00-C-8057

The rest of the paper is organized as follows. First, we present major techniques available for building an ITS. Along the way, we will discuss how these techniques are being utilized in various research projects and the associated drawbacks and limitations. Finally, we identify both troubling spots that require more attention in the community as well as promising research directions.

II. Fundamental Techniques

In this section, we will present the common techniques employed by intrusion tolerant systems. Though we concentrate on tolerance here, we should recognize that identification of faults is a key to tolerance. In other words, intrusion detection of some form is required to provide intrusion tolerance. Most tolerant systems require certain mechanisms to provide triggers indicating possible intrusions. In our discussion of the techniques below, we give examples of the past and current research projects that employ the techniques in different ways to provide intrusion tolerance. We also identify the limitations of these techniques where applicable. We acknowledge the fact that poorly designed mechanisms can themselves be targets of attacks or can be used as the road to attacks. Where applicable we mention the mechanisms necessary to alleviate such a possibility. For conciseness, in this paper, we mainly focus on intrusion tolerant server technologies, though other areas like middleware, mobile code and agents have seen active research in intrusion tolerance.

A. Redundancy and Diversity

The term redundancy has its root in both fault tolerant hardware/software and distributed systems. It generally refers to the extra resources allocated to a system that are beyond its need in normal working conditions. Redundancy is different from replication, which is just one type of redundancy which we have all accustomed to: physical resource redundancy. In [2], Kopetz pointed out two other types of redundancy: time redundancy and information redundancy. A good example of time redundancy is the well known time-out technique used in communication protocol design: if you send a message and wait for an acknowledgment in response, there is usually a timeout value and maybe repeated action associated with it. The timing latitude allowed in this case can tolerate the temporary faults in communication link etc. For information redundancy, we need to look no further than modern data

transmission: original data is encoded with extra bits embedded to provide functionality such as synchronization and error correction, i.e. better tolerance to faults. The threshold scheme described in Section II-D is just an example of information redundancy: for k shares of data (from n pieces of divided data) to be able to reconstruct original data D, there must be certain information redundancy embedded in each share.

If we must single out one essential technique for achieving intrusion tolerance, that would be redundancy technique. In other words, *no redundancy, no tolerance*. Within the scope of intrusion tolerance, however, redundancy alone is seldom sufficient. This is because if redundant components are pure replicas of each other, then the uncorrelated fault assumption that many intrusion tolerant systems make is violated – namely, if the attacker has found a technique to subvert one component and all are pure replicas, it is likely that *all* components are likewise vulnerable. To combat this, another common technique used is diversity. Diversity is the property that the redundant components should be substantially different in one or more aspects, from hardware diversity and operating system diversity, to software implementation diversity. Additionally, diversity is also applied to time and space, in that diverse services should be co-located at multiple sites to protect against local disasters, and that clients may use time diversity by requesting service at different times.

Given above discussion, it won't be a surprise to see that almost all intrusion tolerant systems utilize redundancy in one form or another. For example, Intrusion Tolerant Server Infrastructure (ITSI) [3], Hierarchical Adaptive Control QoS Intrusion Tolerant (HACQIT) System [4], Scalable Intrusion Tolerant Architecture for Distributed Systems (SITAR) [5] and Distributed Intrusion Tolerance (DIT) [6] are all designed with the capability to accommodate one or more COTS web servers, and each server can run different implementations, be it Apache, IIS or something else, for diversity. COCA [7] is another project which aims to design a fault tolerance online certification authority and claims to be the first system to integrate a Byzantine quorum system used to achieve availability: with $3t + 1$ COCA servers up to t maybe faulty or compromised.

Remarks: Increased diversity reduces the risk of correlated faults, but increases the complexity of the system. To achieve the highest degree of diversity, it is necessary to have multiple completely distinct implementations of a service. This would involve running a mix of operating systems on a mix of hardware, with separate implementation teams developing services, at geographically diverse sites. However, the cost of this diversity quickly becomes prohibitively expensive, and immature diversity may only increase potential points of failure, so each party must evaluate for themselves what level of diversity is appropriate. This can be viewed as a trade-off between prevention and tolerance.

B. Voting

Redundancy is a key component in providing any kind of tolerance. As a consequence of having redundant components in ITS system, it is also paramount that the systems' non-faulty components can agree on valid output data in the presence of the faulty components. While all the replicas of a response are considered equally reliable, the output must be based on cross-comparison of available replicas, possibly augmented by knowledge of the application. Voting is used to resolve any differences in redundant responses and to arrive at a consensus result based on the responses of perceived non-faulty components in the system. It has two complementary goals: masking of intrusions, thus tolerating them, and providing integrity of the data. The process involves comparing the redundant responses and reaching agreement on the results to find the "correct" response.

Common metrics for comparison are *Edit Distance* and *Hash Code*. Edit Distance is useful for comparing data where we need to consider modification (insert/delete/replace) costs. A number of variants of the edit distance computations exist [8]: simple edit distance, hamming distance, episode distance etc. The common Unix utility *diff* uses such an approach. Hash Code is a useful metric for large data streams. When computing edit-distance is not-possible or computationally intensive, a digest of the data can be used as a metric. The hash code can be computed using some digest function such as CRC, MD5, or SHA.

These metrics are used in agreement algorithms to arrive at a plausible "correct" response. A leader/delegate usually passes on the chosen replica to the client. Common voting algorithms [9] include:

- *Formalized Majority Voting*: This is the most commonly used algorithm, also known as consensus or majority voting. Here, the replicas are partitioned such that the difference between no two replicas in a partition is greater than a threshold. If the partition with the highest number of replica entries forms the absolute majority, one output from that partition is chosen as the final response.
- *Generalized Median Voting*: In this method, a middle value is selected from the set of N replicas by systematically locating those which differ by greatest amount and eliminating them from consideration.
- *Formalized Plurality Voting*: This algorithm is similar to formalized majority voting algorithm but but for the fact that a relative majority is considered instead of absolute majority.

Voting can be applied at various layers of the networking stack including application layer as well as the middleware layer [10]. SITAR uses edit distance comparison [11] and formalized majority voting as the primary algorithms. DIT [6] uses hash code comparison and formalized majority voting in its architecture. However, both architectures adapt to different algorithms based on the security posture at any given time. Some of the common mechanisms used to thwart attacks against this mechanism include diversity, unpredictable leader election and more redundancy.

Remarks: There is a trade-off between performance and confidence in choosing comparison and agreement strategies. So a balance has to be struck to maximize both. For example, one could compare the distance of the whole responses to be completely sure that the responses are same, or do it over just a part of the responses to be "mostly" sure. Given the

unpredictable nature of intrusions, these decisions are hard to make. Another limitation of these approaches is that they require a significant amount of redundancy, hence increased cost.

C. Acceptance Test

The term "acceptance testing" has its root in software fault tolerance study and generally defined as a developer-provided error detection measure in a software module [12], in the form of a check on the reasonableness of the results calculated. It usually consists of a sequence of statements that will raise an exception if the state of the system is not acceptable. If any exception is raised by the acceptance test, the module is said to have failed or been compromised. We can broadly classify the testing measures into following categories.

- *Requirement test:* In many cases, some conditions are imposed to complete a task. These conditions can be represented as an expected sequential order of events or a subset of given events. The requirement test is to make sure that the imposed conditions are satisfied.
- *Reasonableness test:* Reasonableness test is used to detect software/system failures through pre-computed ranges, expected sequences of program states, or other relationships that are expected to be satisfied. Reasonableness checks are based on physical constraints, while satisfaction of requirements tests are based on logical or mathematical relationships.
- *Timing test:* Timing test is used in systems with time-sensitive components to determine whether the execution time meets the constraints. However, finding out reasonable parameters is not trivial task. In many cases, certain learning or profiling techniques need to be employed to estimate reasonable time parameters.
- *Accounting test:* It is used for transaction-based applications that involve simple mathematical operations. Examples are airline reservation, inventory control systems etc. A tally for both the total number of records and sum over all records of a particular data field can be compared between source and destination, whenever a large number of records are transmitted or reordered.
- *Coding Test:* Essentially, it is a validation process using some form of checksum of current information against prior signature or knowledge of "correct" information.

SITAR uses acceptance testing on both the request and the response streams as a sanity check on them. This mechanism is very useful for detecting known attacks and also some well crafted attacks like timing attacks which are otherwise hard to detect. DIT does semantic checking on its request stream. Most architectures are augmented by IDSes, therefore they perform at least rudimentary analysis on the request streams. Requirement test can be seen as a variant of specification-based check used in IDS and ITS projects, which essentially aim to establish a normal profile based on system specification.

Remarks: One limitation of acceptance testing is that the testing rules are difficult to design and tend to be very application specific. In other words, we can design certain generic testing framework, but the real workload is to analyze the application workflow and come up with reasonable set of tests. So there is nontrivial development cost associated with this. However our experience showed that a highly modular architecture can help a great deal in this regard.

D. Threshold Scheme and Distributed Trust

Threshold scheme is also known as secret sharing, as proposed by Adi Shamir in his classical paper "How to share a secret". The general idea is to devise a method to divide data D into n pieces in such a way that it needs k shares to reconstruct original data D; Anything less, reveals no information at all. This elegant idea has found many applications in key management schemes as well as cryptography.

In terms of its application in ITS, there are two primary ways of using it. First and in its very native form, data shares can be stored in distributed physical locations such that even if $n - k - 1$ shares were attacked and compromised, the confidentiality are still kept and original data can be reconstructed, therefore the tolerance. In fact, this form not only employ threshold scheme, redundancy technique also comes into play due to the nature of dispersion of data. Second, data itself can be encrypted with a secret key, and this key is to be divided into n shares using threshold scheme. This form doesn't exactly provide any redundancy to the original data per se, however, to gain access of the information, you do need k shares of encryption *key* to construct original key, which essentially provide "joint control or custody" of information.

Threshold schemes help ensure confidentiality and survivability. One of the most representative projects is PASIS [13], a survivable storage system developed at CMU. PASIS makes use of threshold schemes to analyze trade-offs among security, availability and performance. Draper Laboratory's CONTRA [14] provides protection and tolerance by camouflaging the messages sent from the source to destination using threshold schemes. Aforementioned COCA also relies on threshold scheme to tolerate faults.

Remarks: A major limitation of these schemes is the choice of n and k. There is a trade-off between performance, availability, confidentiality and storage requirements. A high n ensures high availability at the cost of low performance and high storage requirements. A lower k yields high performance but at the cost of low confidentiality. Threshold schemes are hard to attack if the values of n and k are appropriately chosen and are well guarded by conventional means.

E. Dynamic Reconfiguration

Traditional intrusion detection systems are mostly reactive. The usual response after an intrusion is detected is to perform a post-mortem and take corrective and recovery actions. This is generally a manual task for the administrator and involves some downtime for the server. Survivable systems on the other hand, aim to have none or minimal downtime for the service as far as clients are concerned. They dynamically and adaptively reconfigure the system so that the service can be uninterrupted. Reconfiguration can be proactive or reactive and can help in prevention, elimination as well as tolerance. Reconfiguration can be effected in several different forms:

- *Rollover:* The affected component is transparently replaced by a pristine replica of it.
- *Shifting:* All the traffic directed to the affected server is routed to another safe server.
- *Load sharing:* If the in-availability or degradation in performance is caused by high load, some form of load sharing or balancing may be employed.
- *Blocking:* If a client is perceived to be offending or is suspicious, the system may decide not to service it.
- *Fishbowling:* Fishbowling is similar to blocking. Unlike blocking, however, fishbowling allows the targeted user to continue receiving service. But, it protects normal users from being effected by the attackers intent.
- *Changing the system's posture:* The system's multiple layers of defense can be turned off/on based on the current operating environments and threat indication.
- *Rejuvenation:* The affected component is restarted to restore it to a pristine state wiping out any memory resident or volatile attacks.

Willow [15], DIT and SITAR do both, proactive and reactive reconfiguration. Willow's reconfiguration adds, removes and replaces components and interconnections, or changes the modes of operation. DIT reconfigures the system to use different leader proxy and agreement regime. Additionally DIT recommends periodic rejuvenation as a proactive mechanism. SITAR assigns incoming connections to random circuits, changes the redundancy level based on perceived threat and uses different algorithms for comparison and voting. SITAR reactively rejuvenates components when a fault that is correctable by rejuvenation is detected. ITSI shifts, blocks or fishbowls the traffic when an attack is detected. HACQIT performs reconfiguration by rollover of the faulty components using alternate available hot-spare components. RFITS [16] effects reconfiguration by virtually moving the protected target.

Remarks: As we can see, a wide variety of reconfiguration strategies are employed. A challenge in devising the reconfiguration mechanism is to protect the mechanism from being (mis)used by the attacker. It is important that this process be not very predictable thus making the system vulnerable to attacks on its reconfiguration mechanism. A major challenge for reconfiguration systems is to make them unpredictable and resilient to oscillations in transient effects that may lead to reconfigurations. Reconfiguration can come with a significant performance penalty. Hence, if oscillations are not properly accounted for, it is easy to see that it can trigger a streak of oscillations thereby driving the system to an inconsistent state.

F. Indirection

Indirection is a common technique in computer science. In intrusion tolerance, it is often layered, and occurs at several levels. Indirection allows designers to insert protection barriers and fault logic between clients and servers. Also, since the indirection is hidden outside of the black box system, clients see only what looks like a COTS server. There are at least four main types of indirection used by intrusion tolerant systems: proxies, wrappers, virtualizations, and sandboxes. We will briefly summarize each.

Proxies: A proxy server, usually transparent, is often the first line of defense of a system. The proxy server accepts all client requests, and uses its own logic to perform a variety of functions, including load-balancing, validity testing, signature-based testing, and fault masking. The proxy acts as the sole client access point, hiding all behavior behind the proxy from clients. However, one caveat is that proxy efficiency is paramount to prevent performance bottlenecks.

SITAR and HACQIT use proxies to export the server interface to clients, protecting their many functions. Both are a kind of firewall and load balancing proxy. Since the proxy is the client endpoint, it is a likely target of attack. Though current implementations of SITAR and HACQIT do not appear to have extra protections to guard these, they can benefit from virtualization and redundant proxies.

Wrappers: Wrappers are most commonly placed directly around servers (or other wrappers), and inspect requests and responses before sharing them with other components (but not end clients). The wrapper also differs from the proxy in its intimate knowledge of the server. Though a single proxy/wrapper can be the sole line of defense, commonly the wrapper is behind other indirections, and is used to add functionality to a server without changing the COTS server itself.

Wrappers are commonly employed, such as in SITAR and Willow. SITAR uses wrappers to allow COTS servers to speak a SITAR internal language, and Willow's uses wrappers to augment the abilities of servers. Since the wrappers are treated by the rest of the system as the COTS servers, arguably this addition does not add substantial burden to protection of the COTS servers.

Virtualizations: Generally speaking, virtualizations are naming indirections and are often used subtly, without glorification. When requesting a new virtualized service, the indirection happens as the virtual name is translated into a real name, allowing the real service to be referenced. Thereafter, direct access is allowed, reducing the performance cost of indirections (but then references are not moderated). Some examples of common virtualizations include memory subsystems, the DNS system, and RPC. By virtualizing the names, the details are delayed until needed, and changed as appropriate.

In our review, RFITS and ITSI exhibited noteworthy virtualizations. RFITS is essentially based on virtualization (specifically, the existence of a large namespace from which mappings can be dynamically created and changed between an endpoints virtual channel and the actual channel they communicate with). By detecting flood attacks and negotiating these changes unpredictably between endpoints, RFITS can survive many denial of service flood attacks. RFITS uses cryptography to protect these negotiations. ITSI also uses virtualization, as an alternative to a proxy, by having multiple hardware interfaces share the same MAC address, and using another technique to determine which interface is the true recipient. Since ITSI uses a hardware implementation, its virtualization is not vulnerable to many types of attacks.

Sandboxes: Sandboxes are common tools used to separate users, servers, and other untrusted components. Essentially,

the idea is to run each untrusted component within a sandbox, where all interactions with other systems and subsystems are moderated (and usually significantly restricted). Faults can then be tolerated by the sandbox, by rolling back system state, instantiating a new component to respond, or rejecting the requester (or a variety of other methods). Sandboxes thus provide a window between the untrusted execution, and its results taking effect. If a fault can be detected before the results are committed, they can be safely aborted. Sandboxes are commonly used to protect against faulty mobile code, and to test and diagnose suspected attacks, because the faulty behavior can be limited to within the sandbox.

Two of the projects reviewed, ITSI and HACQIT, use sandboxes. ITSI's fishbowling is one of the possible outcomes of reconfiguration, where the communication with servers is protected within its fishbowl. HACQIT uses a sandbox as an analysis workbench. Whenever a possible intrusion is detected, the logs leading up to it and the compromised server are transferred to the sandbox, and it determines an attack signature within the safety of a sandbox without further risk to critical systems.

Remarks: There are a wide variety of indirections, but they all have a common goal: protection by separating clients and servers by an additional layer. This comes with two main costs: first, indirections often add additional overhead and thus latency, and second, a layer of indirection can violate the End-to-End argument [17], since client communication ends at the edge of the IT system, not the server. Developers should be wary of this when designing these systems.

III. Open Issues and Challenges

In the previous section, we have discussed a wide range of techniques commonly employed when building a highly available and intrusion tolerant system. In some cases, we also mentioned potential problems when applying particular techniques. In this section, we will present a more elaborate summary of these open research issues and challenges based on our experience of working on the SITAR project.

A. Dynamic Content Handling

Many previously discussed intrusion tolerant server projects assume a transaction-based service model: requests come in, responses go out. However, in most real-world application environments, there is usually some kind of back-end database deployed for providing dynamic content, such as a weather or financial information database. Things become tricky when we consider **dynamic content** and **redundancy and diversity** together. One problem is the lack of semantic encoding in user data which makes validation difficult. As an example, consider web services. Although HTML can be validated grammatically using DTDs, there are two substantial problems with using this for validation. (1) A large portion of HTML world-wide is not well formed, causing it to fail this basic validation. (2) This grammatical validation does not take into account the semantic meaning of the document. Therefore, it is inherently difficult to verify if an HTML page correctly expresses its semantic meaning. Dynamic content also presents additional challenges to acceptance testing and ballot voting techniques since responses from servers can differ due to implementation differences or variances in communication delays. Additionally, consistency presents a significant challenge to a project protecting COTS servers. Assume the protected service supports both read and write operations by the client (picture a banking system where you can transfer money, etc.). On a write operation, the COTS server will modify its state, without knowledge of what decision was made by the protection system. The decision actually taken by the protection system may be contrary to the state change made by the COTS server. Since the protection system strives not to modify the COTS server, this seems to present an uncorrectable inconsistency. Transient failures under such conditions will either be challenging to recover from, or cause costly reconfigurations. We investigated alternative options to add semantic meaning to HTML content, such as Dynamic HTML, XHTML, Zope Page Templates, XML/XLST, signed XML etc. They all have pros and cons and some schemes can mitigate the problem to a certain degree. However, dynamic content handling overall remains an open issue and need to be further studied.

B. Tolerance Quantification

Unlike other scientific disciplines in the security world, the question "How secure is the system?" tends to be described in qualitative terms. "Qualitative" in the sense that we claim our system is better than others because we make use of certain mechanisms, follow particular design procedures or methodologies, or use certain algorithms, etc. The open question is whether there are ways to measure security or tolerance capability quantitatively. An earlier paper by Dacier [18] deals with quantitative assessment of operational security by modeling the system as a privilege graph exhibiting operational security vulnerability and transforming the graph into a Markov chain to quantify *mean effort and mean time for attack to succeed*. In SITAR [19], we explored the concept further and came up with a nine-state finite state machine to characterize the transition of intrusion tolerant systems, and use semi-Markov chains to model and quantify. Other research efforts include Sanders' probabilistic validation of intrusion tolerance [20], etc. Despite these efforts, an often raised question is how realistic these measurements are due to the difficulty of parameterization when instantiating the individual models, and the problem at large remains unresolved.

C. Adaptation and Its Fragility

By definition, adaptation is a process of adjusting to environmental conditions. Specifically, many intrusion tolerant systems utilize feedback-loop type of control to monitor the system states and perform pre-defined actions, known as *reconfiguration process* to adapt to current condition. There are two issues with this approach, first real time and step-less adaptation are extremely difficult if not impossible. Taking the SITAR project as an example, the ARM module (adaptive reconfiguration monitor) is designed to dynamically adjust the system's resource allocation to maintain high availability when under high threat level and maximize performance when under

low threat level. Our experience shows that even carefully tuned adaptation is often prone to oscillation, and the practical way to deal with the problem is to carefully preset the mapping between threat levels and allocated resources. At best, this is a limited version of adaptation.

A more severe potential problem is associated with security compromise on these critical modules themselves: what if adaptation/reconfiguration is the target of attack? Many systems, including SITAR, make explicit or implicit assumptions that the protection mechanisms provided are trusted, or protected by "other" mechanisms, which might be too naive an assumption. Knight's paper [21] discusses the issue in a larger context, stressing the importance of securing these survivability mechanisms, while acknowledging the challenges involved. Whether or not this is wholly possible remains to be answered.

D. Cost and Benefits

We all know that security comes with a price. The problem is how to justify the trade-off between cost and benefits. Intrusion tolerance associated costs come in a variety of forms: performance, hardware, administration and maintenance, etc. Let us consider performance costs. Using SITAR system as an example, for the flexibility and fault tolerance of individual modules, we have built inter-component communication using a distributed shared memory architecture called JavaSpaces so that individual component can *join* or *leave* the system both locally and remotely, and they can do so at any time without significantly disrupting the overall operation. By our analysis, one incoming message will pass about a dozen intermediate checking points before it reaches the target server. The requirement to accommodate **legacy systems and COTS servers** only aggravates the problem. Moreover, **diversity and redundancy** techniques in SITAR context also mandate acceptance test modules to reconcile the time difference of returned responses as well as be responsible for message reassembly, which further degrade the overall system throughput. Redundancy and diversity add further costs to an institution's infrastructure budget. Is it worth it?. The general consensus seems to be that it is in the eye of beholder, i.e., if one truly cares for security and intrusion tolerance, this is the price to pay. Therefore, unless we have better techniques to achieve better balance between performance and security, these architectures and techniques might have quite limited applicability to a specific range of mission-critical applications.

IV. Conclusion

In this paper, we studied various fundamental techniques for building highly available, intrusion tolerance systems. We analyzed how a wide range of past and ongoing projects use these different forms of these techniques to provide various aspects of intrusion tolerance. In particular, we focused our attention on the service/server oriented intrusion tolerance systems. We briefly noted the advantages and limitations of various approaches. Based on our implementation experience gained from SITAR project, we also made an effort to discuss open issues and particular challenging problems as well as future research directions.

References

[1] "Organically assured and survivable information system (OASIS)." `http://www.tolerantsystems.org`.

[2] H. Kopetz and P. Veríssimo, *Real Time and Dependability Concepts*, ch. 16, pp. 411–446. Addison-Wesley, 1993.

[3] D. O'Brien, R. Smith, T. Kappel, and C. Bitzer, "Intrusion tolerant via network layer controls," in *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX'03)*, pp. 90–96, April 2003.

[4] J. E. Just, J. C. Reynolds, and K. Levitt, "Intrusion tolerance through forensics-based attack learning," in *Intrusion Tolerant System Workshop, Supplemental Volume on 2002 International Conference on Dependable System and Networks*, pp. C–4–1, 2002.

[5] F. Wang, F. Gong, F. Jou, and R. Wang, "SITAR: A scalable intrusion tolerance architecture for distributed service," in *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, (United States Military Academy, West Point, New York), pp. 38–45, June 4-5 2001.

[6] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou, and T. E. Uribe, "An architecture for an adaptive intrusion tolerant server," *Springer-Verlag*, 2002.

[7] L. Zhou, F. Schneider, and R. van Renesse, "Coca: A secure distributed on-line certification authority," *ACM Transactions on Computer Systems*, vol. 20, pp. 329–368, nov 2002.

[8] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[9] P. R. Lorczak, A. K. Caglayan, and D. E. Eckhardt, "A Theoretical Investigation of Generalized Voters for Redundant Systems," in *The Nineteenth International Symposium on Fault-Tolerant Computing*, pp. 444 – 451, 1989.

[10] A. Franz, R. Mista, D. Bakken, C. Dyreson, and M. Medidi, "Mr. fusion: A programmable data fusion middleware subsystem with a tunable statistical profiling service," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, pp. 273–278, 2002.

[11] R. Uppalli, R. Wang, and F. Wang, "Design of a ballot monitor for an intrusion tolerant system," in *Supplemental Volume of the International Conference on Dependable Systems and Networks (DSN-2002)*, pp. B60–B61, 2002.

[12] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*. Springer Verlag, 1990.

[13] G. R. Ganger, P.Khosla, M. Bakkaloglu, M. Bigrigg, G. Goodson, S. Oguz, V. Pandurangan, C. Soules, J. Strunk, and J. Wylie, "Survivable storage systems," in *DARPA Information Survivability Conference and Exposition, DISCEX II*, vol. 2, pp. 184–195, 2001.

[14] J. Lepanto and W. Weinstein, "Contra: Camouflage of Network Traffic to Resist Attacks." `http://www.tolerantsystems.org`, 2000.

[15] J. Knight, D. Heimbigner, and A. Wolf, "The willow architecture: Comprehensive survivability for large-scale distributed applications," in *Intrusion Tolerant System Workshop, Supplemental Volume on 2002 International Conference on Dependable System and Networks*, pp. C–7–1, 2002.

[16] R. S. Ramanujan, "Project summary: Randomized failover intrusion tolerant systems." `http://www.tolerantsystems.org`, 2000.

[17] J. Salzer, D. Reed, and D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 195–206, 1984.

[18] M. Dacier, Y. Deswarte, and M. Kaâniche, "Quantitative assessment of operational security: Models and tools," Tech. Rep. 96493, LAAS Rsearch Report, May, 1996.

[19] K. Goseva-Postojanova, F. Wang, R. Wang, F. Gong, K. Vaidyanathan, K. Trivedi, and B. Muthusamy, "Characterizing Intrusion Tolerant Systems Using A State Transition Model," in *DARPA Information Survivability Conference and Exposition, DISCEX II*, vol. 2, pp. 211–221, 2001.

[20] W. H. Sanders, M. Cukier, F. Webber, P. Pal, , and R. Watro, "Probabilistic validation of intrusion tolerance," in *Digest of Fast Abstracts: The International Conference on Dependable Systems and Networks*, 2002.

[21] J. C. Knight, K. J. Sullivan, M. C. Elder, and C. Wang, "Survivability architectures: Issues and approaches," in *DARPA Information Survivability Conference and Exposition, DISCEX I*, vol. 2, pp. 157–171, 2000.