Enhancing Fault / Intrusion Tolerance through Design and Configuration Diversity

Alysson Bessani[1] Alessandro Daidone[2] Ilir Gashi[3]
Rafael Obelheiro[4] Paulo Sousa[1] Vladimir Stankovic[3]
[1]LaSIGE, University of Lisbon, Portugal
[2]University of Florence, Italy
[3]Centre for Software Reliability, City University London, UK
[4]Universidade do Estado de Santa Catarina, Brazil
{bessani,pjsousa}@di.fc.ul.pt, daidone@dsi.unifi.it, {i.gashi, v.stankovic}@city.ac.uk, rro@joinville.udesc.br

Abstract

Fault/intrusion tolerance is usually the only viable way of improving the system dependability and security in the presence of continuously evolving threats. Many of the solutions in the literature concern a specific snapshot in the production or deployment of a fault-tolerant system and no immediate considerations are made about how the system should evolve to deal with novel threats. In this paper we outline and evaluate a set of operating systems' and applications' reconfiguration rules which can be used to modify the state of a system replica prior to deployment or in between recoveries, and hence increase the replicas chance of a longer intrusion-free operation.

1 Introduction

In this paper we will present a set of configuration rules which are used to diversify the implementation and/or runtime configuration of operating systems and applications. These rules can be applied before an operating system or application is deployed in a system or in between the system recovery actions. The use of these rules is presented as part of an architecture which utilises a service we named FOREVER - Fault/intrusiOn REmoVal through Evolution & Recovery. The FOREVER service incorporates both reactive and proactive methods of recovery as well as being adaptable to work with traditional fault-tolerant architectures such as those based on design diversity. This work aims to offer designers of fault/intrusion tolerant systems guidelines and heuristics on the architectural evolution of these systems when they are faced with ever-evolving threats. The use of various intrusion or fault tolerance mechanisms to enhance the resilience of software components against either malicious or non-malicious faults and failures is well known in the literature [22, 12], with various architectural and design approaches proposed [23, 22]. Many of these solutions concern a specific snapshot of the production or deployment of a fault-tolerant system. Hence, when a decision is made to deploy a fault-tolerant diverse system made up of, for example, two components, once the two versions of the components used in the system are selected, no immediate considerations are made about how the system should evolve. This becomes especially important to effectively tolerate intrusions and other malicious faults since these evolve during the system life-cycle. Fault/intrusion tolerant systems need to counteract the evolution of threats and exploits against a running system. The counter-measures in current practice are largely *reactive* - issuance of patches and upgrades of software components to protect against known faults, threats and vulnerabilities: but they do not happen in real-time and the "at-risk-time" may be significantly longer for the system while a patch is being developed for a publically known vulnerability. Hence, the designers need *proactive* methods to counteract the evolution of faults, threats and vulnerabilities. With proactive methods design *novelty* is used as a defence. The use of a proactive approach is not a substitute for the reactive methods such as patching and upgrading, or the fault tolerant architectures proposed in the literature. It rather aims to complement these approaches with a more dynamic response to evolving threats and vulnerabilities.

The rest of the paper is organised as follows: section 2 briefly describes the FOREVER service; section 3 presents the configuration diversity rules; section 4 contains evaluation results on the effectiveness of configuration diversity and other dependability aspects of the FOREVER service; section 5 contains conclusions and provisions for further work.

2 Architecture of FOREVER

The main goal of the FOREVER service is to enhance the resilience of fault/intrusion-tolerant replicated systems [5, 15, 13] by allowing these systems to tolerate an arbitrary number of replica failures without increasing the total number of replicas. Such an ambitious goal is achieved through the combination of two important and complementary mechanisms: recovery and evolution. FOREVER allows an intrusion-tolerant system to *recover* from past malicious actions/faults, by cleaning the effects of such actions through periodic and on-demand recoveries that neutralize the effects of both undetected and detected faults and intrusions. Moreover, when FOREVER triggers a recovery of a certain replica, it not only cleans the effects of previous malicious actions/faults, but also *evolves* the replica, modifying the vulnerabilities that may be exploited by a malicious adversary, by applying a set of configuration diversity rules (e.g., changes OS access passwords, randomizes open ports, switches between different authentication methods). These rules are explained in Section 3.

In order to avoid the possibility that FOREVER itself becomes a victim of malicious attacks, a fault/intrusion-tolerant system enhanced with FOREVER should be built under a hybrid system model and architecture [24] in which the system is composed of *two parts*, with distinct properties and assumptions. These two parts are typically called *payload* and *wormhole*. The fault/intrusion-tolerant application (and replication library) runs in the payload part exposed to arbitrary faults and asynchrony. The FOREVER service runs in the wormhole part that is guaranteed to be secure and timely by construction.

A more detailed description of the FOREVER service is available in [21].

3 Diversity aspects

The main resilience goal of using any redundancy in a software architecture is to minimise the probability that a failure/intrusion of one of the components will lead to a system failure/intrusion. To pursue *failure/intrusion diversity* between components a designer building a fault-tolerant system can use various forms of diversity:

- *simple separation of redundant executions*. This is the weakest form, but it may yet tolerate some faults/intrusions. In the database research community it is well known that many bugs in complex, mature software products are "Heisenbugs" [11], i.e., they cause apparently non-deterministic failures. When a database fails, its identical copy may not fail, even with the same sequence of inputs. The reported phenomena of Heisenbugs that we are aware of concern non-malicious activity, but may also be applicable for malicious behaviour (e.g., some sort of brute force attack against a system which only leads to a successful penetration under certain non-deterministic combinations of behaviours in the running system);

- *design diversity*, the typical form of parallel redundancy for fault tolerance against design faults (either accidental or intentional): the multiple replicas of the system are handled by diverse software components;

- *data diversity* ([1]): for some systems there may be a natural redundancy in the input language which allows the demands to the system to be expressed in syntactically different but logically equivalent forms. A practical example is the SQL language for databases where a sequence of one or more SQL statements can be "rephrased" into a different but logically equivalent sequence to produce redundant executions (see [9] for a recent study with SQL database servers);

- *configuration diversity* (which can be seen as a special form of data diversity). Software products often come with many configuration parameters affecting for example the amount of system resources they can use (amount of RAM, CPU time), port number used for communication, authentication method etc. Given the same software product, varying these parameters between two installations can produce different implementations of the data and the operation sequences on them, and thus decrease the risk of the same bug/vulnerability being triggered in two installations of the same software.

These precautions can in principle be combined. For instance, data diversity can be used with diverse software products; diverse software products can be deployed with configuration diversity of the different software. The choice will depend on the cost (e.g. purchasing the software), maintenance costs from increased complexity, time to deployment etc. There has been significant amount of research on the architectures for exploiting design diversity and experimental studies of evaluating its effectiveness: for a useful survey of architectural options the reader is referred to [22]; the assessment issues are discussed in [14]; diverse intrusion-tolerant architectures are reported in [20]. We already mentioned earlier in the paper research on data diversity as a defence against non-malicious faults [1, 9]; recently there has also been efforts to apply the data diversity concepts as protection against malicious behavior [17].

In what follows we summarise rules of configuration diversity which purport to enhance the resilience of software in between recoveries. A summary of the rules defined is given in Table 1. Full details can be found in a technical report [10], where the types of attacks the rule would help

ID	Rule name	Design Implications		Security Category
		Impl. intrusiveness	Client notification required?	
1	Password change	B	Yes	C
2.1	Different authentication protocols	W, B or G	Yes	C
2.2	Different Trusted Third Parties	W, B or G	No	C and A
3	Different "factors" in n-factor authentication methods	W, B or G	Yes	C
4.1	Address Space Layout Randomisation (ASLR)	W	No	I
4.1.1	Pointer obfuscation	W	No	I
4.1.2	Randomisation of global variables and local variables offsets	W	No	I
4.2	Address Space Partitioning	W, B or G	No	I
4.3	Stack Frame Padding	W, B or G	No	I
4.4	Basic Block reordering	W, B or G	No	I
5.1	Instruction set randomisation	W, B or G	No	I
5.2	Instruction set tagging	W, B or G	No	I
5.3	Instruction Reordering	W, B or G	No	I
6.1	Diverse Linux User IDs (UID)	W, B or G	No	I and C
7	Change IP addresses of the hosts	B	Yes	C and A
8	Changing listening port numbers	W, B or G	Yes	C and A
9	Adding or deleting non-functional code	W, B or G	No	I
10.1	Varying dynamic libraries and system calls	W	No	I
10.2	Varying unique names of system files	W	No	I
10.3	Varying magic numbers in certain files (e.g., executables)	W	No	I

Table 1. Configuration rules for diversifying the deployment of an OS or an application. Abbreviations: (W)hite-, (B)lack-, (G)rey-box; (C)onfidentiality, (I)ntegrity, (A)vailability

in alleviating are also listed. The structure of Table 1 is as follows:

- ID – the rule identifier for easier traceability to the technical report [10];
- Rule name – a concise description of the rule;
- Design implications – implications on the architectural design of the operating system (OS) or application to which the rule is applied:

  Implementation intrusiveness – is access to the internal implementation of the OS or applications required to apply the rule (*white box*) or can the rule be applied simply through the utilisation of the OS or application's configuration parameters (*black box*)? There might also be *grey box* solutions for which the implementation of the rule may not need to have access to the internal implementation of the OS or application but it can be built on top of its API (Application Programming Interface) - this is especially useful for proprietary OSs and applications for which access to the implementation is not provided. For some of the rules, depending on the application or OS implementation, any of these implementation types (white-, black- or grey-box) are possible; in those cases we list all (or a subset of the options) for a given rule and we provide more detailed explanation of the scenarios under which a given implementation solution is possible in the technical report [10].

  Client notification required? – should client applications of the OS or application to which the rule is applied be notified once the rule is applied?

- Security framework category – under which CIA (confidentiality, integrity, availability) category does the rule fall into.

The rules were generated in the following ways:

- Bottom-up – exploring the implementations of operating systems, such as Linux and MS Windows, and applications, such as database servers (e.g., PostgreSQL, Interbase, Oracle, MS SQL), and identifying the features and interfaces that can be diversified and configured in different ways.
- Top-down – exploring reported vulnerabilities (such as NVD [18]) and defining rules that can "workaround" or protect against the types of vulnerabilities and attacks listed in these sources.
- Lateral – reviewing existing literature of configuration rules for protection against "malicious" behaviour

(such as [8]).

Which rule may be most effective at improving the security of a given system will depend on the operational and threat profile of the system. For example, attacks that exploit memory programming errors (e.g., buffer overflows) are one of today's most serious security threats. These attacks require an attacker to have an in-depth understanding of the internal details of the system being attacked, including the locations of critical data and/or code. Address obfuscation techniques such as *Address Space Layout Randomisation*- listed as rule 4.1 in Table 1 - randomize the location of program data and code each time a program is executed. It has been shown that address obfuscation can greatly reduce the probability of successful attacks [3, 19]. Given that recoveries force the restart of every program in the system, they are a perfect way of introducing address obfuscation both at the OS and application level.

4 Evaluation

This section summarizes the assessment of the FOREVER service, aiming to quantify how much this service enhances the resilience of the system in which it is implemented by evaluating the probability of system failure through variation of i) time between recoveries, ii) probability of common vulnerabilities and iii) mean effectiveness of configuration diversity rules applied.

The quantitative evaluation of the FOREVER service was performed using a modelling methodology based on the following argument: recovery actions determine a change in the overall system configuration, therefore it is possible to represent the entire operational life split into different periods of deterministic duration called "phases". This feature allows a reconfiguration strategy belonging to the Multiple Phased System (MPS) class for which a modelling and evaluation methodology exists [16], supported by the DEEM tool [4]. The details about the model were omitted for space reasons, but the interested readers can find the full details in [6].

System overview and assumptions The replicated system used to assess the FOREVER service was composed of $n = 4$ replicas and hence was able to tolerate up to $f = 1$ failed replicas ($n \geq 3f + 1$). Replicas were assumed to suffer permanent arbitrary faults and to have one single failure mode: "failed". For ease of modelling, we assumed that a replica, as soon as it was hit by a fault, explicitly manifested a permanent failure. The (overall) system was considered failed as soon as the number of failed replicas was greater than f.

Replicas were diverse both in space (design diversity) and in time (application of diversity rules). Design diversity was modelled assuming that each replica has its own failure rate λ_A^i obtained by multiplying a basic value $\lambda_A = 10^{-5}$ (about one failure per day, as in [7]) with a replica-specific multiplier[1] obtained from the results of a NVD study reported elsewhere [2]. We pessimistically assumed that λ_A^i was increased by an "aging penalty" value $\delta_\lambda = 10^{-6}$ (10% of the basic failure rate) when no configuration diversity rule was applied during a recovery. We modelled diversity in time domain updating the basic replica failure rate after each recovery in this way: $\lambda^i_{A(after)} = \lambda^i_{A(before)} + \delta_\lambda (1 - \delta_x)$, where $\delta_x \in \{0, 1\}$ is an "effectiveness" parameter[2] obtained as the mean value for the effectiveness parameters of all the applied rules.

Despite diversity, we assumed the existence of common faults (e.g., common vulnerabilities) in pairs of replicas. Common faults were modelled taking into account the conditional probability δ^{ij} that replica i is faulty, given that replica j is hit by the same fault. The basic values for δ^{ij} were set[3] based on the results of the NVD study [2], assuming that a common fault between two replicas was mainly due to the exploitation of a common vulnerability between the corresponding operating systems.

The modelled recovery strategy managed proactive recoveries only (as in PBFT [5] and COCA [25]), performed sequentially "one-at-the-time" on a round robin basis. Each recovery action lasted for $T_R = 120$ seconds[4]; a waiting time T_W took places between recoveries, so that the recovery period was $T_p = n(T_R + T_W)$ seconds . The recovery process was assumed fault-free (a replica is correct after its recovery).

Evaluation Results We evaluated the (overall) system failure probability p_F (i.e. the probability of having more than f failed replicas) over mission time t, varying the following parameters: i) recovery period T_P (acting on the waiting time T_W), ii) probability δ_{ij} of common faults and iii) mean value δ_x for the effectiveness parameters of the configuration diversity rules. The measure of interest was evaluated using the analytic solver, with[5] $\varepsilon = 10^{-10}$ and $Maxiter = 10^4$.

The results of the assessment are given in Figure 1; a more extensive discussion of the analysis is given in [6]. The main outcomes can be summarised as follows. First (Figure 1(a)), the shorter the time between replica recoveries the longer the system can survive failure-free; second

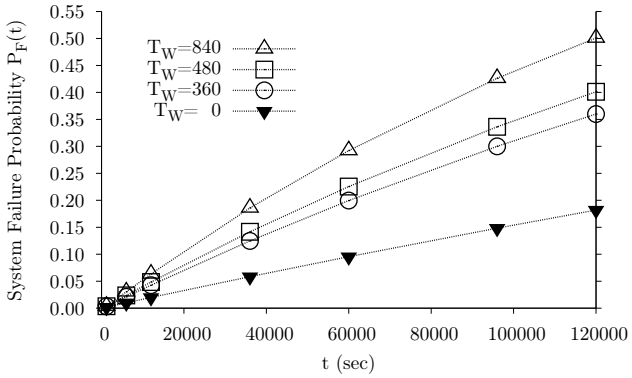
[1] The multipliers used here were 1.0, 1.8, 1.5 and 1.9.

[2] $\delta_x = 0$: the rule is not effective at all, e.g., it changes the number of a listening port that is not used; $\delta_x = 1$: the rule has the optimal effect, e.g., it changes the root password following a root password compromise.

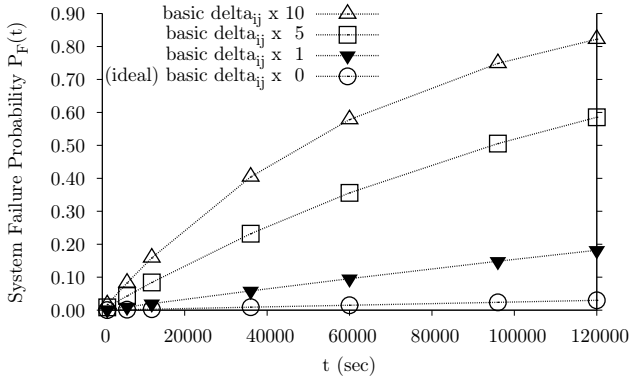
[3] The values for δ^{ij} spanned between 0.004 and 0.029.

[4] As in [7], where a prototype of a system replicated with diversity, possible target for the FOREVER service, was studied.

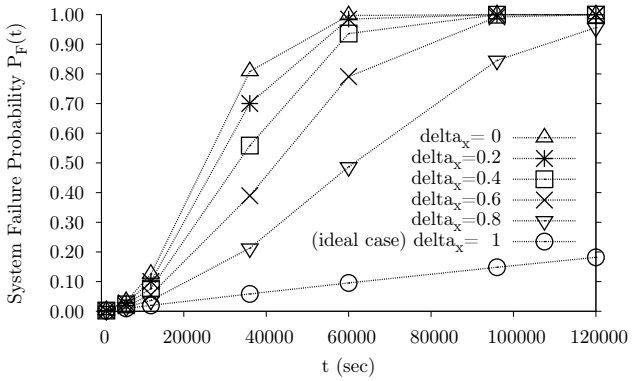
[5] ε represents the error tolerance, *Maxiter* the maximum number of iterations that has to be considered by the transient solution method.



(a) At varying the time T_W between recoveries (with $\delta_\lambda = 0$, $\delta_x = 0$)



(b) At varying the probability of common fault δ_{ij} (with $\delta_\lambda = 0$ and $\delta_x = 0$). NOTE: we use 'delta' instead of δ in the figure for font embedding purposes.



(c) At varying the mean effectiveness δ_x of the configuration diversity rules ($\delta_\lambda = 10^{-6}$). NOTE: we use 'delta' instead of δ in the figure for font embedding purposes.

Figure 1. System failure probability p_F over time for different evaluation studies

(Figure 1(b)), the lower the common failure rate between the replicas deployed in the system the lower the overall system failure rate is (this is an obvious observation, but the assessment also gives measures of how much worse the overall system failure becomes when the common failure rates of the replicas increases); and, third (Figure 1(c)), under the assumption that the configuration rules can only improve security, then the obvious observation from the assessment is that it is better to apply any rule than no rule at all; but we also see that the gains from lower failure rates greatly depend on the effectiveness of the rule (or the ruleset) applied, and measures of these gains can be estimated.

5 Conclusions

In this paper we have addressed the issue of how systems should evolve to tolerate the effects of ever-growing and ever-present security threats. We defined a set of twenty configuration rules for diversifying the implementation and/or the runtime environment of operating systems and applications before deployment in a system as well as in between system recoveries. These rules are then used as part of a wider intrusion-tolerant architecture which utilises a service we call FOREVER - Fault/intrusiOn REmoVal through Evolution & Recovery. The FOREVER service incorporates both reactive and proactive methods of recovery and is adaptable to work with traditional fault-tolerant architectures, for instance those based on design diversity.

We also performed an assessment of several parameters of the FOREVER service and the underlying architecture and explained how the overall system failure rate is affected from varying the parameters.

There are several provisions for extending the work presented in this paper. First, the configuration rules for diversifying the operating systems and applications, presented in section 3, can be extended with further new rules, proof of concept implementations and deployments of these rules in real applications.

Second, evaluation of the effects on system performance (in terms of overall system response time and / or throughput rates) that the application of these rules will cause.

Finally, the assessment model presented in section 4 could be refined to model reactive recovery actions [21, 7] too, allowing us to study how effective is the trade-off between proactive and reactive recoveries, as well as to further research the effectiveness of the configuration rules.

Acknowledgments

This work was partially supported by the EC through project IST-2004-27513 (CRUTIAL) and NoE IST-4-026764-NOE (RESIST), and by the FCT through the Multiannual Funding and the CMU-Portugal Programs.

References

[1] P. Ammann and J. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, C-37(4):418–425, 1988.

[2] A. N. Bessani, R. R. Obelheiro, P. Sousa, and I. Gashi. On the effects of diversity on intrusion tolerance. DI/FCUL TR 08–30, Dep. of Informatics, Univ. of Lisbon, Dec. 2008.

[3] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of the 14th USENIX Security Symp.*, pages 271–286, 2005.

[4] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli, and F. Sandrini. DEEM: a tool for the dependability modeling and evaluation of multiple phased systems. In *IEEE Int. Conf. on Dependable Systems and Networks (DSN 2000)*, pages 231–236, 2000.

[5] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[6] A. Daidone. Forever assessment: modelling details. Technical Report rcl080508, Univ. of Florence, Dip. Sistemi Informatica, RCL group, Dec. 2008.

[7] A. Daidone, S. Chiaradonna, A. Bondavalli, and P. Veríssimo. Analysis of a redundant architecture for critical infrastructure protection. In R. De Lemos, F. Di Giandomenico, C. Gacek, H. Muccini, and M. Vieira, editors, *Architecting Dependable Systems V*, volume 5135 of *LNCS*, pages 78–100. 2008.

[8] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of The 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Cape Cod, MA, USA, 1997. IEEE Computer Society Press.

[9] I. Gashi and P. Popov. Rephrasing rules for off-the-shelf sql database servers. In *6th European Dependable Computing Conf. (EDCC-6)*, pages 139–148, Coimbra, Portugal, 2006.

[10] I. Gashi and V. Stankovic. List of rules for diversifying operating systems and applications and their respective runtime environment(s). http://www.csr.city.ac.uk/people/ilir.gashi/ConfigDiv/, 2008.

[11] J. Gray. Why do computers stop and what can be done about it? In *5th Symposium on Reliability in Distributed Software and Database Systems (SRDSDS-5)*, pages 3–12, Los Angeles, CA, USA, 1986. IEEE Computer Society Press.

[12] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong. Survivability through customization and adaptability: The cactus approach. In *DARPA Information Survivability Conference & Exposition*, 2000. 04900294.pdf.

[13] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of the 21st ACM Symp. on Operating Systems Principles (SOSP'07)*, Oct. 2007.

[14] B. Littlewood, P. Popov, and L. Strigini. Modelling software design diversity - a review. *ACM Computing Surveys*, 33(2):177–208, 2001.

[15] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo. Randomized intrusion-tolerant asynchronous services. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'06)*, pages 568–577, jun 2006.

[16] I. Mura and A. Bondavalli. Markov regenerative stochastic Petri nets to model and evaluate the dependability of phased missions. *IEEE Transactions on Computers*, 50(12):1337–1351, 2001.

[17] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson. Security through redundant data diversity. In *38th IEEE/IFPF International Conference on Dependable Systems and Networks (DSN'08),*, Anchorage, Alaska, USA, 2008.

[18] National vulnerability database. http://nvd.nist.gov/, 2008.

[19] R. Pucella and F. B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *Proc. of the 19th IEEE Work. on Computer Security Foundations*, pages 230–241, 2006.

[20] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt. The design and implementation of an intrusion tolerant system. In *Dependable Systems and Networks (DSN-02)*, pages 285–292, Washington, D.C., USA, 2002. IEEE Computer Society Press.

[21] P. Sousa, A. N. Bessani, and R. R. Obelheiro. The FOREVER service for fault/intrusion removal. In *2nd Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS'08)*, Apr. 2008.

[22] L. Strigini. Fault tolerance against design faults. In H. Diab and A. Zomaya, editors, *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, pages 213–241. J. Wiley & Sons, 2005.

[23] M. van der Meulen, S. Riddle, L. Strigini, and N. Jefferson. Protective wrapping of off-the-shelf components. In *4th Int. Conf. on COTS-Based Software Systems (ICCBSS '05)*, volume LNCS, pages 168–177, 2005.

[24] P. Verissimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1), 2006.

[25] L. Zhou, F. Schneider, and R. Van Rennesse. COCA: A secure distributed online certification authority. *ACM TOCS*, 20(4):329–368, Nov. 2002.