Proactive Resilience through Architectural Hybridization

Paulo Sousa
Faculdade de Ciências
Univ. Lisboa
pjsousa@di.fc.ul.pt

Nuno Ferreira Neves
Faculdade de Ciências
Univ. Lisboa
nuno@di.fc.ul.pt

Paulo Veríssimo
Faculdade de Ciências
Univ. Lisboa
pjv@di.fc.ul.pt

ABSTRACT

In a recent work, we have shown that it is not possible to dependably build any type of distributed f fault or intrusion-tolerant system under the asynchronous model. This result follows from the fact that in an asynchronous environment one cannot guarantee that the system terminates its execution before the occurrence of more than the assumed number of faults.

Some systems resorted to proactive recovery as a way to address this problem, by attempting to ensure that no more than f faults ever occur: nodes are periodically rejuvenated to remove the effects of faults or malicious attacks. However, asynchronous systems with proactive recovery also suffer from the same problem. In fact, proactive recovery protocols usually require stronger assumptions (e.g., synchrony, security) than the system that is proactively recovered.

To solve this contradiction, we work with a hybrid distributed system model. We propose *proactive resilience* as a new and more resilient approach to proactive recovery, based on *architectural hybridization*: proactive recovery functions are encapsulated in architectural devices that meet the required stronger assumptions, and have a well-defined interface with the recovered system.

We present the Proactive Resilience Model (PRM) and describe a design methodology under the PRM. This methodology is a way of building systems which guaranteedly do not suffer more than the assumed number of faults, and we use it to derive a distributed intrusion-tolerant secret sharing system.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

SAC'06 April 23-27, 2006, Dijon, France

General Terms

Security, Reliability, Design, Algorithms

Keywords

Intrusion tolerance, proactive recovery, secret sharing

1. INTRODUCTION

A distributed system built under the asynchronous model makes no timing assumptions about the operating environment: local processing and message deliveries may suffer arbitrary delays, and local clocks may present unbounded drift rates [9, 6]. Thus, in a (purely) asynchronous system one cannot guarantee that something will happen before a certain time.

Consider now that we want to build a dependable intrusion-tolerant distributed system, i.e., a distributed system able to tolerate arbitrary faults, including malicious ones. Can we build such a system under the asynchronous model?

This question was partially answered, twenty years ago, by Fischer, Lynch and Paterson [7], which proved that there is no deterministic protocol that solves the consensus problem in an asynchronous distributed system prone to even a single crash failure. This impossibility result (commonly known as FLP) has been extremely important, given that consensus lies at the heart of many practical problems, including membership, ordering of messages, atomic commitment, leader election, and atomic broadcast. In this way, FLP showed that the very attractive asynchronous model of computation is not sufficiently powerful to build *certain* types of fault-tolerant distributed protocols and applications.

What are then the minimum synchrony requirements to build a dependable intrusion-tolerant distributed system?

If the system needs consensus (or equivalent primitives), then Chandra and Toueg [4] showed that consensus can be solved in asynchronous systems augmented with failure detectors (FDs). The main idea is that FDs operate under a more synchronous environment and can therefore offer a service (the failure detection service) with sufficient properties to allow consensus to be solved.

But what can one say about intrusion-tolerant asynchronous systems that do not need consensus? Obviously, they are not affected by the FLP result, but are they dependable?

Independently of the necessity of consensus-like primitives, we have recently shown that assuming a maximum number of f faulty nodes under the asynchronous model is dangerous. Given that an asynchronous system may have a potentially long execution time, there is no way of guar-

anteeing exhaustion-safety, i.e., that no more than f faults will occur, specially in malicious environments [13]. Therefore, we can rephrase the above statement and say that the asynchronous model of computation is not sufficiently powerful to build *any* type of (exhaustion-safe) fault-tolerant distributed protocols and applications.

To achieve exhaustion-safety, the goal is to guarantee that the assumed number of faults is never violated. In this context, proactive recovery seems to be a very interesting approach [11]. The aim of proactive recovery is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/faults. If the rejuvenation is performed frequently often, then an adversary is unable to corrupt enough resources to break the system. Proactive recovery has been suggested in several contexts. For instance, it can be used to refresh cryptographic keys in order to prevent the disclosure of too many secrets [8, 16, 1, 15, 10]. It may also be utilized to restore the system code from a secure source to eliminate potential transformations carried out by an adversary [11, 3].

Therefore, proactive recovery has the potential to support the construction of exhaustion-safe intrusion-tolerant distributed systems. But, in order to achieve this, proactive recovery needs to be architected under a model sufficiently strong that allows regular rejuvenation of the system. In fact, proactive recovery protocols (like FDs) typically require stronger environment assumptions (e.g., synchrony, security) than the rest of the system (i.e., the part that is proactively recovered).

This paper proposes proactive resilience – a new and more resilient approach to proactive recovery based on architectural hybridization [14]. We argue that the proactive recovery subsystem should be constructed in order to assure a synchronous and secure behavior, whereas the rest of the system may even be asynchronous.

We describe a generic Proactive Resilience Model (PRM), which proposes to model the proactive recovery subsystem as an abstract component – the Proactive Recovery Wormhole (PRW). The PRW may have many instantiations depending on the application proactive recovery needs. Then, a design methodology under the PRM is presented and shown to be a way of building exhaustion-safe systems. Finally, we use this methodology to build an exhaustion-safe distributed f intrusion-tolerant secret sharing system, which makes use of a specific instantiation of the PRW targetting the secret sharing scenario [12].

2. AN ARCHITECTURAL HYBRID MODEL FOR PROACTIVE RECOVERY

The main difficulty with proactive recovery is not the concept but its implementation – this mechanism is useful to periodically rejuvenate components and remove the effects of malicious attacks/failures, as long as it has timeliness guarantees. In fact, the rest of the system may even be completely asynchronous – only the proactive recovery mechanism needs synchronous execution.

This type of requirement make us believe that one of the possible approaches to dependably use proactive recovery, is to execute it in the context of a wormhole: a subsystem capable of providing a small set of services, with good properties (e.g., timeliness, security) that are otherwise not available in the rest of the system [14]. Wormholes must

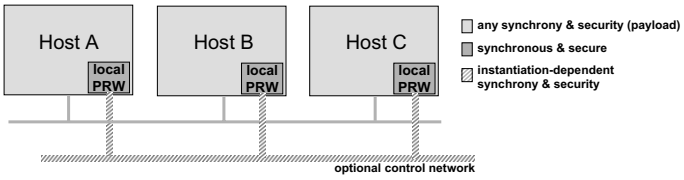

Figure 1: The architecture of a system with a PRW.

be kept small and simple to ensure the feasibility of building them with the expect trustworthy behavior. Moreover, their construction must be carefully planned to guarantee that they have access to all required resources when needed.

We propose the Proactive Resilience Model (PRM) as a more resilient approach to proactive recovery based on a wormhole-enhanced architecturally hybrid distributed system model. The PRM defines that the architecture of a system enhanced with proactive recovery should be hybrid, i.e., divided in two parts: the proactive recovery subsystem and the payload system, the latter being proactively recovered by the former. Each of these two parts should be built under different timing assumptions and fault models.

The payload system executes the "normal" applications. Thus, the payload synchrony and fault model entirely depends on the applications executing in this part of the system. For instance, the payload may operate under an asynchronous Byzantine environment.

The proactive recovery subsystem executes the proactive recovery protocols that rejuvenate the applications running in the payload part. This subsystem is more demanding, by definition, in terms of timing and fault assumptions, but some of these assumptions depend on the specific proactive recovery protocol, which can be of many types. Thus, we chose to model the proactive recovery subsystem as an abstract component which allows many instantiations.

2.1 The Proactive Recovery Wormhole

The Proactive Recovery Wormhole (PRW) is an abstract secure real-time distributed component that aims to execute proactive recovery procedures.

The architecture of a system with a PRW is suggested in Figure 1. An architecture with a PRW has a local module in some hosts, called the *local PRW*. Depending on the instantiation, these modules may or may not be interconnected by a *control network*. This set up of local PRWs optionally interconnected by the control network is collectively called *the* PRW. The PRW is used to execute proactive recovery procedures of applications running between participants in the hosts concerned, on any usual distributed system architecture (e.g., the Internet). We call the latter the *payload system and network*, to differentiate from the PRW part.

Conceptually, a local PRW should be considered to be a module inside a host, and separated from the OS. In practice, this conceptual separation between the local PRW and the OS can be achieved in several ways: (1) the local PRW can be implemented in a separate, tamper-proof hardware module (e.g., PC board) and so the separation is physical; (2) the local PRW can be implemented on the native hardware, with a virtual separation and shielding implemented in software, between the former and the OS processes.

The local PRWs are assumed to be fail-silent (they fail by crashing). Every local PRW preserves, by construction, the

following property:

P1 There exists a known upper bound $T^{local}_{exec_{max}}$ on the processing delays.

As mentioned, a PRW instantiation may or may not have a control network. For instance, if a proactive recovery procedure only requires local information, then the control network is expendable. Even when the control network is required, its characteristics will depend on the specific requirements of the proactive recovery procedure.

The PRW offers a single service, defined as follows:

Definition 1. Given any function F, with a calculated worst case execution time of T_{Xmax}, an execution interval T_D, and a time interval (period) T_P, satisfying $T_{Xmax} < T_D < T_P$, then F is triggered by the PRW **periodic timely execution service** at real time instants t_i (the i-th triggering occurs at instant t_i), with $T_D < t_i - t_{i-1} \leq T_P$, and F terminates within T_D from $t_i, \forall i$.

In short, the PRW has the ability to periodically execute well-defined functions in known bounded time. Moreover, the PRW allows the definition of a set of fail-safe measures to be triggered in certain situations. For instance, these fail-safe measures may shutdown the system if the *periodic timely execution* service fails to satisfy its specification.

A triple $\langle D, \langle F, T_P, T_D\rangle, S\rangle$ defines a PRW instantiation, such that:

- D represents the set of *data* which is proactively recovered in all nodes;
- $\langle F, T_P, T_D\rangle$ represents the *function* F which is periodically triggered with period T_P and timely executed within T_D of each triggering, through the *periodic timely execution* service, in all nodes. F makes operations over the data defined in D;
- S represents the set of (optional) *self-checking* mechanisms, which have the goal of guaranteeing a fail-safe behaviour of all the nodes.

2.2 Building Exhaustion-Safe Intrusion-Tolerant Systems

In order to build an exhaustion-safe distributed f intrusion-tolerant system, one has to guarantee that no more than f (accidental or malicious) faults occur during system execution. If the system maximum execution time is known, then one may choose a sufficient high f – by endowing the system with sufficient nodes – so that exhaustion never occurs. However, if the system has an unbounded execution time, we have a problem – it is not possible to estimate how many nodes will be needed to avoid exhaustion. One possible approach to solve this problem is to use the Proactive Resilience Model – enhance the system with a PRW in order that nodes are periodically and timely rejuvenated. Notice that this approach may even be applied in systems with a known bound on execution time when there is the need of minimizing the number of used nodes.

We propose a design methodology to build exhaustion-safe distributed f intrusion-tolerant systems, under the Proactive Resilience Model. The methodology has 3 steps.

1. Define the data D to rejuvenate, the rejuvenation procedure F, and calculate F's worst case execution time (T_{Xmax}). Then, define the execution interval T_D (greater than T_{Xmax}), and the periodicity T_P (greater than T_D). Finally, define the actions S to be performed if F is not executed with the required periodicity and execution time.
2. Build a PRW instantiation $\langle D, \langle F, T_P, T_D\rangle, S\rangle$.
   - Notice that T_P and T_D may be increased if necessary. This will only impact the required fault-tolerance degree, as explained in step 3.
3. Define the degree f_{safe} of fault-tolerance, such that, the minimum time necessary ($T_{exhaust_{min}}$) for $f_{safe}+1$ faults to be produced satisfies the condition $T_{exhaust_{min}} > T_P + T_D$.

Given that at most f_{safe} faults are produced during any two consecutive rejuvenations, it is guaranteed that no more than f_{safe} faults will ever be produced at the same time during the entire execution of the system.

3. THE PROACTIVE SECRET SHARING WORMHOLE

Secret sharing schemes protect the confidentiality and integrity of secrets by distributing them over different locations. A secret sharing scheme transforms a secret s into n shares $s_1, s_2, ..., s_n$ which are distributed to n share-holders. In this way, the adversary has to attack multiple share-holders in order to learn or to destroy the secret. For instance, in a $(k+1, n)$-threshold scheme, an adversary needs to compromise more than k share-holders to learn the secret, and corrupt at least $n-k$ shares in order to destroy the same secret.

Various secret sharing schemes have been developed to satisfy different requirements. In this paper we use Shamir's scheme [12] to implement a $(k+1, n)$-threshold scheme: given an integer valued secret s, pick a prime q which is bigger than both s and n. Randomly choose $a_1, a_2, ..., a_k$ from $[0, q[$ and set polynomial $f(x) = (s + a_1 x + a_2 x^2 + ... + a_k x^k)$ mod q. For $i = 1, 2, ..., n$, set the share $s_i = f(i)$. The reconstruction of the secret can be done by having $k+1$ participants providing their shares and using polynomial interpolation to compute s.

In many applications, a secret s may be required to be held in a secret-sharing manner by n share-holders for a long time. If at most k share-holders are corrupted throughout the entire lifetime of the secret, any $(k+1, n)$-threshold scheme can be used. In certain environments, however, gradual break-ins into a subset of locations over a long period of time may be feasible for the adversary. If more than k share-holders are corrupted, s may be stolen. An obvious defense is to periodically refresh s, but this is not possible when s corresponds to inherently long-lived information (e.g., cryptographic root keys, legal documents).

Thus, what is actually required to protect the secrecy of the information is to be able to periodically renew the shares without changing the secret. Proactive secret sharing (PSS) was introduced in [8] in this context. In PSS, the lifetime of a secret is divided into multiple periods and shares are renewed periodically. In this way, corrupted shares will not accumulate over the entire lifetime of the secret since they are checked and corrected at the end of the period during which they have occurred. A $(k+1, n)$ proactive threshold scheme guarantees that the secret is not disclosed and can be recovered as long as at most k share-holders are corrupted during each period, while every share-holder may be corrupted multiple times in several periods.

The Proactive Secret Sharing Wormhole (PSSW) is an instantiation of the PRW presented in Section 2.1. The PSSW

Algorithm 1 refresh() for each node $P_i, i \in \{1...n\}$

```
1: for m = 1 to k do
2:    δ_im ← generate_random_number([0, q[)
   {{δ_im}_{m∈{1...k}} defines the polynomial δ_i(z) = δ_i1 z^1 +
   δ_i2 z^2 + ... + δ_ik z^k}

   {send δ_i(j) to each correct server P_j}
3: for j = 1 to n do
4:    if j ≠ i and pfd(P_j) = correct then
5:       u_ij ← δ_i(j) mod q
6:       for l = 1 to Od + 1 do
7:          send u_ij to P_j

   {receive δ_j(i) from each correct server P_j}
8: for j = 1 to n do
9:    if j ≠ i and pfd(P_j) = correct then
10:      receive u_ji from P_j

   {calculate the new share and erase the old one}
11: share ← share + (u_1i + u_2i + ... + u_ni)(mod q)
12: erase_old_share()
```

targets distributed systems which are based on secret sharing and the goal of the PSSW is to periodically rejuvenate the secret share of each system node.

The PSSW is defined by $\langle D_{PSSW}, F_{PSSW}, S_{PSSW} \rangle$, such that:

- $D_{PSSW} =$ { `share` }, where `share` is the secret share to be periodically refreshed.
- $F_{PSSW} =$ $\langle$ `refresh`, $T_P, T_D \rangle$, where the `refresh()` function is presented as Algorithm 1, and is based on the share renewal scheme of [8]. Each process $P_i, i \in \{1..n\}$, executes Algorithm 1 at the beginning of the time periods defined by T_P.
- $S_{PSSW} =$ {shutdown if share is not periodically and timely refreshed, as specified by T_P and T_D}.

THEOREM 1. *If all local PSSWs follow Algorithm 1, then:*

Bounded execution time *There is an upper bound $T_{exec_{max}}$ on the difference between the real time instant when the first local PSSW starts executing the algorithm and the real time instant when the last local PSSW finishes executing it.*

Robustness *After all local PSSWs finish the algorithm execution, the new computed shares correspond to the initial secret (i.e., any subset of $k+1$ of the new shares interpolate to the initial secret).*

Secrecy *An adversary that at any time knows no more than k shares learns nothing about the secret.*

PROOF. Robustness and Secrecy are proved in [8].

Bounded execution time: Due to space limitations, we just give a brief intuition. Given that Algorithm 1 is executed in the context of the PSSW, there is a bound on local processing delays (Prop. P1). Moreover, by deploying an appropriated control network, it is also possible to guarantee a bound on network delivery delays. In such an environment, it is straightforward to implement a perfect failure detector pfd, such that, $pfd(P_i) = correct$ iff P_i is not crashed, $\forall P_i$. Therefore, $T_{exec_{max}}$ exists and can be calculated by combining the bounds on local processing and network delivery delays. □

We now apply the methodology presented in the previous section, to build an exhaustion-safe distributed intrusion-tolerant secret sharing system:

1. $D = \{share\}$, $F = refresh()$, $T_D = c_d T_{exec_{max}}$, $T_P = c_p T_{exec_{max}}$, and c_d and c_p are constants with $c_p > c_d > 1$, and $S = S_{PSSW}$.
2. Build the PSSW with the parameters defined in step 1.
3. k is chosen in order that $(c_p + c_d) T_{exec_{max}} <$ time needed to compromise $k+1$ shares.

Notice that in the secret sharing scenario, the degree of fault-tolerance (f_{safe}) is represented by k. In the following section, we present an example of how this methodology can be applied in practice.

3.1 Experimental Results

We have implemented an experimental prototype of the PSSW using RTAI [5], an operating system with real-time capabilities, and a switched Fast-Ethernet control network. The feasibility of achieving timeliness guarantees using this type of operating system and network is discussed in [2]. RTAI allows the construction of an architecturally hybrid execution environment, with the PSSW executing as a set of real-time tasks, and the normal applications executing at Linux user-level.

This section presents the results of a few experiments that we have conducted using this prototype, with the goal of observing the execution time of the `refresh()` function presented in Algorithm 1. Our experimental infrastructure was composed by 500 MHz PIII based PCs running RTAI, and interconnected by a 3COM SuperStack II Baseline switch. The experiences presented below used 32-bit shares. We are currently conducting more experiments with larger shares.

In the first experience we set $k = 1$ and tested configurations from 2 to 6 machines. The goal was to evaluate the overhead introduced when the number of machines increases. The results (mean, maximum and minimum execution time) are presented in Table 1, and are based on 65535 periodic executions per configuration. The main conclusion is that the mean execution time increases with the number of machines used. This was expected given that more machines generate more messages and thus greater network and processing delays. However, the maximum execution time remains quite stable independently of the number of machines. This is very important and consolidates our conjecture that there exists an upper bound $T_{exec_{max}}$ on the execution time. Moreover, these measurements also allow us to conclude that one could trigger a rejuvenation every 2 seconds with a maximum overhead of less than 2% (given that $T_{exec_{max}} < 30ms$, we could set $T_D = 40ms$ and $T_P = 2000ms$). An adversary would have to obtain $k+1 = 2$ shares in less than 2.1 seconds ($\approx T_P + T_D$) in order to reconstruct the protected secret.

The next step was to evaluate the impact of increasing k. Therefore, in the second experience, we used 6 machines to test the behaviour of the system with k varying between 1 and 5. The results are presented in Table 2, and are also based on 65535 periodic executions per configuration. We see that there is a slight increment on the mean and maximum execution when k increases. This happens because the size of k only impacts on the processing delay. With such results, and setting $k = 5$, one can extend the previous conclusions and say that an adversary would have to

$\#machines$	T_{exec} (msec)		
	mean	max	min
2	10.1	20.0	10.0
3	13.6	20.4	10.0
4	15.2	20.8	10.2
5	16.0	20.9	12.1
6	16.7	20.8	11.6

Table 1: Performance of the PSSW with $k = 1$.

k	T_{exec} (msec)		
	mean	max	min
1	16.7	20.8	11.6
2	16.9	21.1	11.2
3	17.1	21.2	10.7
4	17.2	21.8	10.7
5	17.3	21.9	10.8

Table 2: Performance of the PSSW with 6 machines.

obtain 6 shares in less than 2.1 seconds in order to discover the secret. However, so much resilience may raise problems in terms of system performance. The overhead of periodically rejuvenating the system with intervals of 2 seconds is still less than 2%, but the same small window of vulnerability that limits malicious actions, also limits the actions of normal system applications. They need to do something useful in less than 2 seconds or they will never manage to use a consistent set of shares. So, there is a tradeoff between resilience and performance, which should be taken into account during system design.

All these results are useful for the system architect to calculate the appropriate degree of fault-tolerance k. For instance, consider that an adversary A takes n seconds to discover n shares. Following the methodology presented in section 2.2, and if we set $T_P = 2sec$ and $T_D = 40ms$, the degree k of fault-tolerance should be such that the time needed to discover $k+1$ shares ($(k+1)sec$) is greater than $T_P + T_D$ ($2.04sec$). So, we conclude that $k = 2$ suffices to tolerate A's intrusions.

4. CONCLUSIONS

Recently, we showed that it is not possible to build dependable intrusion-tolerant distributed systems under the asynchronous model, because they do not guarantee exhaustion-safety. Based on this finding and on the fact that proactive recovery protocols typically require stronger environment assumptions than the rest of the system, we proposed *proactive resilience* as a novel approach to proactive recovery that is based on an architectural hybrid distributed system model: the proactive recovery protocols are executed through a subsystem with "better" properties than the rest of the system.

The Proactive Resilience Model (PRM) was presented and we showed that there exists a design methodology under the PRM allowing to build exhaustion-safe systems. This methodology was applied to the secret sharing scenario in order to derive an exhaustion-safe distributed intrusion-tolerant secret sharing system.

We presented some experimental results that confirm our conjectures. Our experimental secret sharing prototype is intrusion-tolerant and, with a degree of fault-tolerance $k \geq 2$, it can tolerate, with an overhead of less than 2%, any number of intrusions produced at the maximum rate of 1 intrusion per second, over the lifetime of the system.

5. ACKNOWLEDGMENTS

This work was partially supported by the EC, through project IST-2004-27513 (CRUTIAL), and by the FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE) and project POSC/EIA/60334/2004 (RITAS).

6. REFERENCES

[1] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proc. of the Conf. on Computer and Comm. Security*, pages 88–97, 2002.

[2] A. Casimiro, P. Martins, and P. Veríssimo. How to build a Timely Computing Base using Real-Time Linux. In *Proc. of the Int. Workshop on Factory Comm. Systems*, pages 127–134, Sept. 2000.

[3] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Comp. Sys.*, 20(4):398–461, Nov. 2002.

[4] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. of the ACM*, 43(2):225–267, Mar. 1996.

[5] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper. In *Real-Time Linux Workshop*, Nov. 2000.

[6] F. Cristian and C. Fetzer. The timed asynchronous system model. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, pages 140–149, 1998.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. of the ACM*, 32(2):374–382, Apr. 1985.

[8] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proc. of the Annual Int. Crypto. Conf. on Advances in Crypto.*, pages 339–352, 1995.

[9] N. Lynch. *Distributed Algorithms*. 1996.

[10] M. A. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Trans. on Dep. and Sec. Comp.*, 1(1):34–47, Jan–Mar 2004.

[11] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (ext. abstract). In *Proc. of the Annual Symp. on Princ. of Dist. Comp.*, pages 51–59, 1991.

[12] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[13] P. Sousa, N. F. Neves, and P. Veríssimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proc. of the Int. Conf. on Dependable Systems and Networks*, pages 98–107, June 2005.

[14] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Dir. in Dist. Computing*, volume 2584 of *LNCS*, pages 108–113. 2003.

[15] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Trans. on Comp. Sys.*, 20(4):329–368, Nov. 2002.

[16] L. Zhou, F. B. Schneider, and R. van Renesse. Proactive secret sharing in asynchronous systems. Technical Report 2002-1877, Cornell Univ., Oct. 2002.