Proceedings of the First Workshop on Recent Advances on Intrusion-Tolerant Systems

Miguel Correia, Nuno Ferreira Neves (editors)

DI–FCUL TR–07–05

March 2007

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

WRAITS 2007

Workshop on Recent Advances on Intrusion-Tolerant Systems

March 23 2007
Lisboa, Portugal

Miguel Correia
Nuno Ferreira Neves (Eds.)

Miguel Correia
Nuno Ferreira Neves (Eds.)

Recent Advances on Intrusion-Tolerant Systems

Proceedings of the
1st Workshop on Recent Advances on Intrusion-Tolerant Systems

In conjunction with the European Conference on Computer Systems
EuroSys 2007

March 23 2007
Lisboa, Portugal

Editors

Miguel Correia
Nuno Ferreira Neves

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande
1749-016 Lisboa, Portugal

Preface

The First Workshop on Recent Advances on Intrusion-Tolerant Systems aims to bring together researchers in the related areas of Intrusion Tolerance, Distributed Trust, Survivability, Byzantine Fault Tolerance, and Resilience. These areas have the purpose of enhancing the Dependability and Security of computer systems by tolerating both malicious faults (attacks, intrusions) and accidental faults.

The workshop will be specially interested in "intrusion-tolerant systems": how to build them? How to evaluate and test their dependability and security? What systems need to be intrusion-tolerant? It will provide a forum for researchers in these areas to present recent results, discuss open problems that still need research, the steps that need to be taken for intrusion-tolerant systems to be deployed in practice, and the target application domains for intrusion tolerance.

The program consists of 7 papers, and one keynote speech by Professor William H. Sanders, from University of Illinois at Urbana-Champaign, whose presence in the workshop we thank.

We are very grateful to the members of the Program Committee for their work with reviewing the papers, helping us to assemble a very good program. It was a pleasure to collaborate with such a remarkable set of specialists. We are also thankful to all the authors who submitted papers for this first edition of the workshop.

Miguel Correia
Nuno Ferreira Neves

Program Committee

Cristina Nita-Rotaru, Purdue University, US
David Powell, LAAS, France
Felix Freiling, University Mannheim, Germany
HariGovind Ramasamy, IBM Zurich, Switzerland
Joni Fraga, Federal University Santa Catarina, Brazil
Klaus Kursawe, Philips Research Labs, Germany
Lau Cheuk Lung, Pontifícia Universidade Católica Paraná, Brazil
Lorenzo Alvisi, University Texas-Austin, US
Paulo Veríssimo, University Lisboa, Portugal
Piotr Zielinski, Cambridge University, UK
Priya Narasimhan, Carnegie-Mellon University, US
Roberto Baldoni, University Roma, Italy
Rodrigo Rodrigues, Technical University Lisboa, Portugal

Table of Contents

Automatic Recovery from Failures and Attacks Using Bounded Partially Observable Markov Decision Processes

Keynote Speech

William H. Sanders
Donald Biggar Willett Professor of Engineering
Director, Information Trust Institute
University of Illinois at Urbana-Champaign

ABSTRACT

Providing high availability with acceptable cost is becoming increasingly critical in a wide variety of application domains. Automatic system monitoring, followed by diagnosis and recovery, has the potential to provide the required low-cost solution, high-availability solution that is needed in those domains. However, automating recovery is difficult in practical settings because of low coverage, poor localization abilities, false positives, and/or false negatives exhibited by many commonly used monitoring techniques. Furthermore, multiple recovery actions are often applicable even when the fault or attack can be correctly diagnosed, and the correct choice is not always obvious. We present a holistic model-based approach that overcomes these challenges and enables automatic recovery in distributed systems. To do so, it uses theoretically sound techniques to provide controllers that choose good, if not optimal, recovery actions according to a user defined optimization criteria. The automatic recovery problem is framed as an undiscounted accumulated reward maximization problem on a structurally restricted class of partially observable Markov decision processes (POMDPs) that we call R-POMDPs. The resulting recovery controller can be proven to always terminate in a finite amount of time, to probabilistically not terminate before recovery is finished, and to generate recovery actions that are on the average no more expensive than promised. Simulation-based experimental results on a realistic e-commerce system demonstrate that the proposed bounds can be rapidly improved iteratively, and the resulting controller convincingly outperforms a controller that uses heuristics instead of bounds.

Design and Implementation of an Intrusion-Tolerant Tuple Space

Alysson Neves Bessani† Eduardo Pelison Alchieri† Joni da Silva Fraga† Lau Cheuk Lung§
† Departamento de Automação e Sistemas – Universidade Federal de Santa Catarina – Brazil
§ Programa de Pós-Graduação em Informática Aplicada – Pontifícia Universidade Católica do Paraná – Brazil

ABSTRACT

The tuple space coordination model is one of the most interesting communication models for open distributed systems due to its space and time decoupling and its synchronization power. Several works have tried to improve the dependability of tuple spaces. Some have made tuple spaces fault-tolerant while others have focused on security. However, many practical applications in the Internet require both these dimensions. This paper describes the design and implementation of DEPSPACE, a dependable communication infrastructure based on the tuple space coordination model. DEPSPACE is dependable in a strong sense of the word: it is secure, fault-tolerant and intrusion-tolerant, i.e. it behaves as expected even if some of the machines that implement it are successfully attacked. Moreover, it is a policy-enforced augmented tuple space, a shared memory object that we have recently proven to be universal, i.e., capable of implementing any other shared memory object.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

Keywords

Intrusion Tolerance, Tuple Space, Replication, Confidentiality

1. INTRODUCTION

The *generative* (or *tuple space*) *coordination* model, originally introduced in the LINDA programming language [8], relies on a shared memory object called a *tuple space* to support coordination between distributed processes. Tuple spaces can support communication that is decoupled in time – processes do not have to be active at the same time – and space – processes do not need to know each others locations or addresses [5], providing some level of synchronization at the same time. The operations supported by a tuple space are essentially the insertion, reading and removal of tuples, i.e., of finite sequences of values.

Previous works on fault-tolerant and secure tuple spaces (e.g., [2, 10, 4]) have a narrow focus in two senses: they consider only simple faults (crashes) or simple attacks (invalid access); and they are about *either* fault tolerance *or* security. The present paper goes one step further by investigating the implementation of *secure and fault-tolerant tuple spaces*. The solution is inspired on a current trend in dependability that applies fault tolerance concepts and mechanisms in the domain of security, *intrusion tolerance* [7, 16]. The proposed tuple space is not centralized but implemented by a set of tuple space servers. This set of tuple spaces forms a tuple space that is *dependable*, meaning that it enforces the attributes of reliability, availability, integrity and confidentiality [1], despite the occurrence of arbitrary faults, like attacks and intrusions in some servers.

The implementation of a dependable tuple space with the abovementioned attributes presents some interesting challenges. Our design is based on the classical *state machine replication* approach [13, 6]. However, this approach does not guarantee the confidentiality of the data stored in the servers; quite on the contrary, replicating data in several servers is usually considered to reduce the confidentiality since the potential attacker has more servers where to attempt to read the data, instead of just one. Therefore, combining the state machine approach with confidentiality is a non-trivial challenge that has to be addressed. A second challenge is intrinsically related to the tuple space model. Tuple spaces resemble associative memories: when a process wants to read a tuple, it provides a template and the tuple space returns a tuple that "matches" the template. This match operation involves comparing data in the tuple with data in the template, but how can this comparison be possible if tuples are encrypted to guarantee confidentiality? In this paper we present DEPSPACE, a system that addresses these challenges using a particular kind of secret sharing scheme together with cryptographic hash functions in such a way that it guarantees that a tuple stored in the system will have its content revealed only to authorized parties.

The design of a dependable tuple space is not a merely academic exercise. The tuple space system presented in this paper might be useful in several practical application domains, like the following. *(i.) Ad hoc networks* are an important current trend in computer science. It has been shown that tuple spaces can be a powerful solution to coordinate activities in those environments, and systems like LIME [12] already explore this paradigm. *(ii.) Mobile agents* are programs that migrate from node to node in the network, usually to gather data or to perform computations close to the data source. The interaction of these agents is usually complex due to the lack of fixed location. Tuple spaces are an obvious solution to support this communication since they provide time and space decoupling [5]. *(iii.) Grid computing* involves using resources in large numbers of computers to perform complex computations. These computations are decoupled both in space and time so a tuple space would be a

EuroSys WRAITS'07 March 23, 2007, Lisbon, Portugal

good solution to coordinate the tasks performed.

Algorithms based on a tuple space with the properties of DEPSPACE are well suited for coordination of non-trusted processes in practical dynamic systems. Instead of trying to compute some distributed coordination task using a complete dynamic model (like, for instance, the one proposed in [11]), we pursue a more pragmatic approach where a tuple space is deployed on a fixed and small set of servers and is used by a unknown, dynamic and unreliable set of processes that need to coordinate themselves. An example of scenario were this kind of system can be deployed are peer-to-peer systems and infrastructured wireless networks.

The paper has two main contributions. The first is the presentation of the dependable and intrusion-tolerant tuple space. This design involves a non-trivial combination of security and fault tolerance mechanisms: state machine replication, space and tuple level access control, and cryptography. To the best of our knowledge this is the first work to implement Byzantine state machine replication for tuple spaces and to integrate this technique with a confidentiality scheme. The second contribution is the first practical assessment of the performance of an intrusion-tolerant scheme that provides data confidentiality even when there are intrusions in some of the servers. We are not aware of any other practical assessment of such a scheme in the literature.

2. DEFINING A DEPENDABLE TUPLE SPACE

A *tuple space* can be seen as a shared memory object that provides operations for storing and retrieving ordered data sets called *tuples*. A tuple t with all its fields defined is called an *entry*, and can be inserted in the tuple space using the $out(t)$ operation. A tuple in the space is read using the operation $rd(\bar{t})$, where $\bar{t}$ is a *template*, i.e. a special tuple in which some of the fields can be wild-cards or formal fields. The operation $rd(\bar{t})$ returns any tuple in the space that *matches* the template, i.e. any tuple with the same number of fields and with the field values equal to all corresponding defined values in $\bar{t}$. A tuple can be read *and* removed from the space using the $in(\bar{t})$ operation. The in and rd operations are blocking. Non-blocking versions, inp and rdp, are also usually provided [8].

The tuple space implemented in this paper provide another operation usually not considered by most tuple space works: $cas(\bar{t}, t)$ (conditional atomic swap) [2, 15, 3]. This operation works like an indivisible execution of the code: **if** $\neg rdp(\bar{t})$ **then** $out(t)$. The operation inserts t in the space iff $rdp(\bar{t})$ does not return any tuple, i.e., if there is no tuple in the space that matches $\bar{t}$. The cas operation is important mainly because a tuple space that supports it is capable of solving the consensus problem [15], which is a building block for solving many important distributed synchronization problems like atomic commit, total order multicast, leader election and fault-tolerant mutual exclusion.

A tuple space is dependable if it satisfies the *dependability attributes* [1]. Like in many other systems, some of these attributes do not apply or are orthogonal to the core of the design (e.g. safety and maintainability). The relevant attributes in this case are: *reliability* (the operations on the tuple space have to behave according to their specification), *availability* (the tuple space has to be ready to execute the operations requested), *integrity* (no improper alteration of the tuple space can occur), and *confidentiality* (the content of tuple fields cannot be disclosed to unauthorized parties).

The difficulty of guaranteeing these attributes comes from the occurrence of *faults*, either due to accidental causes (e.g., a software bug that crashes a server) or malicious causes (e.g., an attacker that modifies some tuples in a server). Since it is difficult to model the behavior of a malicious adversary, intrusion tolerant systems mostly assumes the most generic class of faults – arbitrary or Byzantine faults – so the solution we propose and describe in the next section is quite generic in terms of the faults it handles.

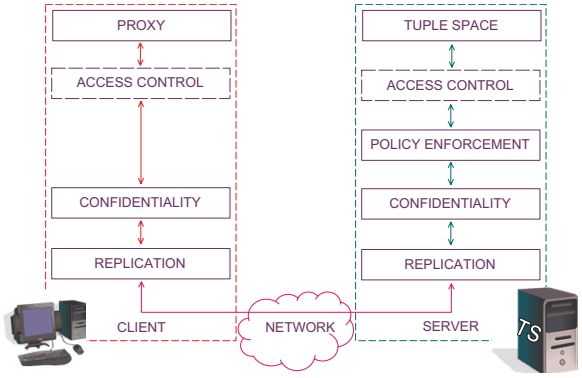


Figure 1: DEPSPACE architecture

3. BUILDING A DEPENDABLE TUPLE SPACE

This section presents the design of DEPSPACE. We begin with a model of the underlying system and the basic assumptions of our design, then delve into the general layered architecture and finally into each layer.

3.1 Underlying Assumptions

The system is composed by an infinite set of *clients* which interact with a set of n *servers* that implement the dependable tuple space with the properties introduced in the previous section. At most f servers and an unbounded number of clients can suffer Byzantine failures, i.e. they can deviate arbitrarily from their specification. We assume *fault independence* for servers, i.e. the failures of the servers are uncorrelated. This assumption can be substantiated in practice using diversity.

All communication between clients and servers is made over *reliable authenticated point-to-point channels*. These channels can be implemented using TCP and some cryptographic mechanism such as MACs (Message Authentication Codes) with session keys.

The dependable tuple space does not require any explicit time assumption, however, since it is based on the *state machine replication* model [13], it requires a total order multicast primitive. We implement this primitive using the BYZANTINE PAXOS protocol [6, 17], which only ensures liveness if the system eventually becomes synchronous.

3.2 Architecture Overview

The architecture of the dependable tuple space consists in a series of integrated layers that enforce each one of the dependability attributes stated in Section 2. Figure 1 presents the DEPSPACE architecture with all its layers.

On the top of the client-side stack is the proxy layer, which provides access to the replicated tuple space, while on the top of the server-side stack is the tuple space implementation (a local tuple space). The communication follows a scheme similar to remote procedure calls. The application interacts with the system by calling functions with the usual signatures of tuple spaces' operations: $out(t)$, $rd(\bar{t})$, ... These functions are called on the proxy. The layer below handles tuple level access control. After, there is a layer that

takes care of confidentiality (Section 3.4) and then one that handles replication (Section 3.3). The server-side is similar, except that there is a new layer to check the access policy for each operation requested. Access control and policy-enforcement are not described in this paper due to space constraints. Some aspects of these mechanisms are described in [3].

We must remark that not all of these layers must be used in every tuple space configuration. The idea is that the layers are added or removed according to the desired quality of service required for the tuple space instance.

3.3 Replication

The most basic mechanism used in DEPSPACE is *replication*: the tuple space is maintained in a set of n servers in such a way that the failure of up to f of them does not impair the reliability, availability and integrity of the system. The idea is that if some servers fail, the tuple space is still ready (availability) and the operations work correctly (reliability and integrity) because the correct replicas manage to overcome the misbehavior of the faulty replicas. A simple approach for replication is *state machine replication* [13]. This approach guarantees *linearizability* [9], which is a strong form of consistency in which all replicas appear to take the same sequence of states.

The state machine approach requires that all replicas *(i.)* start in the same state and *(ii.)* execute all requests in the same order [13]. The first point is easy to ensure, e.g. by starting the tuple space with no tuples. The second requires a fault-tolerant *total order multicast* protocol, which is the crux of the problem. The state machine approach also requires that the replicas are deterministic, i.e. that the same operation executed in the same initial state generates the same final state in every replica. This implies that a read (or removal) in different servers in the same state (i.e. with the same set of tuples) must return the same response.

The protocol for replication is very simple: the client send an operation request using total order multicast and wait for $f + 1$ replies with the same response from different servers. Since each server receives the same set of messages in the same order (due to the total order multicast), and the tuple space is deterministic, there will be always at least $n - f \geq 2f + 1$ correct servers that execute the operation and return the same reply.

3.4 Confidentiality

The enforcement of confidentiality in a replicated tuple space is not trivial. Several solutions that come to mind simply do not work or are unacceptable for the generative coordination model. One of those solutions would be to encrypt the client-server communication and let the tuple space encrypt the tuple fields with its own key(s). This is unacceptable because we assume f servers can fail maliciously, so they might decrypt the tuple fields and disclose their contents. A second solution would be to let the client that inserts a tuple to encrypt the tuple fields either with a secret key (with a symmetric cryptography algorithm like AES) or with its private key (with a public-key algorithm like RSA). The problem of this solution is that it requires all clients that might read and/or remove this tuple to know the decryption key. This contradicts the anonymity property of the generative coordination model [8], which states that clients should not need to know information about each other.

The solution we propose follows in some way the idea of letting the servers handle the confidentiality. However, instead of trusting each server to keep the confidentiality of the tuple fields, we trust a *set* of servers. The solution is based on a $(n, f + 1)$*–publicly verifiable secret sharing* scheme (PVSS) [14]. Clients play the role of the dealer of the scheme, encrypting the tuple with the public keys of each server and obtaining a set of tuple *shares*. Any tuple can be decrypted with $f + 1$ shares, therefore a collusion of malicious servers cannot disclose the contents of confidential tuple fields. A server can build a proof that the share that it is giving to the client is correct. The PVSS scheme also provides two verification functions, one for each server to verify the share it received from the dealer and other for the client to verify if the shares collected from servers are not corrupted.

The confidentiality scheme has also to handle the problem of matching (possibly encrypted) tuples with templates. To solve this problem we use a *collision-resistant hash function* $\mathsf{H}(v)$ (e.g. SHA-1) that maps an arbitrarily length input to a fixed length output (called a *hash*).

The idea is to use the hashes of the fields of a tuple as a fingerprint of the tuple, and execute the matching of tuples using the hashes of the fields of the tuple, instead of the their values. The fingerprint of a tuple is stored in each server together with its tuple share. One limitation of this scheme is that although hash functions are unidirectional, if the range of values hat a field can take is known and limited, then a brute-force attack can disclose its content. This limitation is a motivation for not using typed fields in a dependable tuple spaces. Using fingerprints and the PVSS scheme, the procedures for providing confidentiality for tuple spaces on top of state machine replication is the following:

Tuple insertion. All shares are sent encrypted together with the fingerprint of the tuple and its validity proof by the client using total order multicast. The encryption of each share s_i addressed to server p_i is made through symmetric cryptography, using the session key shared between the client and the server p_i. Notice that all servers will receive all encrypted shares, however, each server will have access only to its corresponding share, the fingerprint of the tuple and the proof generated by the PVSS algorithm. These three pieces of data are stored in the tuple space.

Tuple access. To access a tuple, the client sends the fingerprint of the template and then waits for the replies from the servers containing the same tuple fingerprint that matches the template fingerprint sent, the encrypted share of the server for this tuple and its corresponding proof of validity (produced by the server). Each share is encrypted by the servers with the session key shared between the client and the server to avoid eavesdropping on the replies. Additionally, the replies from the servers can be signed to make the client capable of cleaning invalid tuples from the space (see bellow). The client decrypts the received shares, verifies their validity, and combines $f + 1$ of them to obtain the stored tuple.

Recovery procedure. Notice that nothing prevents a malicious client to insert a tuple with a fingerprint that does not correspond to it. Consequently, after obtained a stored tuple, the client has to verify if the tuple corresponds to the fingerprint. If such correspondence does not exist, the client must clear the tuple from the space (if it is not removed yet) and reissue its operation to the space. The "tuple cleaning" is made in two steps: *(1.)* the client sends all replies received to the servers to prove that the stored tuple is invalid; and *(2.)* the servers verify if the replies are produced by the servers and, if the tuple returned does not correspond to the fingerprint, this tuple is removed from the local tuple space. Moreover, the client that inserted the invalid tuple can be put on a black list (and its further requests ignored). This ensures that a malicious client cannot insert tuples after some of its invalid insertions have been cleaned.

A key advantage of the confidentiality scheme of DEPSPACE is that most of the cryptographic processing is done at client side. This improves the scalability of the system, as will be show in Section 5.

An interesting point of our scheme is that the confidentiality layer weakens our tuple space semantics since it no longer satisfies linearizability in all situations: a malicious client can insert invalid shares in some servers and valid shares in others, so it is not possible to ensure that the same read/remove operation executed in the same state of the tuple space will have the same result: the result depends of the $n-f$ responses collected. However, DEPSPACE satisfies linearizability for all tuples that have been inserted by correct processes.

4. IMPLEMENTATION

The DEPSPACE was implemented using the Java programming language, and at present it is a simple but fully functional dependable tuple space. The Byzantine-resilient state machine replication algorithm implemented is the PAXOS AT WAR described in [17], combined with a total ordering scheme inspired by the one defined by [6][1]. Authentication was implemented using the SHA-1 algorithm for producing HMACs (providing an approximation for authenticated channels on top of Java TCP Sockets). SHA-1 was also used for computing hashes. For symmetric cryptography we employed the Triple DES algorithm while RSA with exponents of 1024 bits was used for digital signatures. All the cryptographic primitives used in the prototype were provided by the default provider of version 1.5 of JCE (Java Cryptography Extensions). The only exception was the PVSS scheme, which we implemented following the specification in [14], using algebraic groups of 192 bits.

Two main implementation optimizations are specially relevant for the system performance. The first is to try to execute rdp and rd first without total order multicast and wait for $n-f$ responses. If all of them are equal, the returned value is the result of the operation, otherwise the normal protocol operation must be executed. The second optimization is for servers to send read replies without signing them. The clients must explicitly request signed responses for an operation if they find that the read tuple is invalid. This improves the latency of the read operations since the processing cost for asymetric cryptography is still very high. Since it is expected that invalid tuples will be very rare, in most cases the read operations will not require digital signatures.

5. EXPERIMENTAL EVALUATION

This section presents an experimental evaluation of DEPSPACE. The execution environment was composed by a set of five Athlon 2.4GHz PCs with 512 Mb of memory and running Linux (kernel 2.6.12). They were connected by a 100Mbps switched Ethernet network. The Java runtime environment used was Sun's JDK 1.5.0_06.

We considered tuples with 4 fields and sizes equal to 64, 256, and 1024 bytes, running on a system with 4 servers[2]. Two cases were considered in all experiments: the complete system (with confidentiality) and the system with the confidentiality scheme deactivated. All our experiments considered fault-free executions. Figure 2 present the results.

The first experiments (Figures 2(a) to 2(d)) measured the delay perceived by the client for each one of the tuple space non-blocking operations. The client was in one of the machines and the servers in the other four. We executed each operation 1000 times and obtained the mean time discarding the 5% values with greater variance.

The results presented in the figure show that out, inp and cas have almost the same latency when the confidentiality layer is not used – the solid lines in Figures 2(a) to 2(d). This is the latency imposed by the total order multicast protocol (about 6 ms). rdp, on the other hand, is much more efficient (about 2 ms) due to the optimization presented in Section 4, which avoids running the total order multicast protocol.

The dotted lines in the graphs show the latency of the protocols when the confidentiality layer is used. In these experiments all tuples inserted and read have all their fields comparable. In fact, the number of comparable fields is not relevant since the overhead for producing a hash is negligible when compared to the overhead of the PVSS scheme. The cas operation (Figure 2(d)) has two dotted lines, one measuring the cases where a tuple is inserted and other for the cases when some tuple is read. The additional latency cost caused by the confidentiality scheme is mostly due to the client-side processing of the operations. The global cost of the confidentiality scheme is also higher for out since this is the only operation in which the shares and their proofs have to be generated. Notice that the processing cost of the cas operation when a tuple is read is approximately the cost of out plus the cost of rdp. This reflects the fact that this operation executes both tuple insertion and access confidentiality processing.

From the Figure 2, it is clear that the size of the tuple has almost no effect on the latency experienced by the protocols. This happens due to two implementation features: *(i.)* our BYZANTINE PAXOS implementation makes agreement over message hashes; and *(ii.)* the secret shared in the PVSS scheme is not the tuple, but a symmetric key used to encrypt the tuple. *(i.)* implies that it is not the entire message that it ordered by the PAXOS protocol, but only its hash (MD5 hashes always have 128 bits), consequently the message size has little effect over the agreement protocol. With feature *(ii.)* we can execute all the required PVSS cryptographic in the same, relatively small algebraic field of 192 bits, which means that the tuple size has no effect in these computations and the use of the confidentiality scheme implies almost the same overhead regardless the size of the tuple.

The second set of experiments measured the throughput of DEPSPACE. For these experiments we used a modified client process that pre-processes C requests for the operation of interest (executing the client-side processing) and send then one-by-one to the servers. We measured the time T taken to process all these requests at one of the replicas, from the moment it receives the first request to the moment it sends the response for the last one. The throughput of the system is calculated as C/T.

Figures 2(e) to 2(h) show that the system provides a high throughput with few servers. Even with larger tuples, the decrease in throughput is reasonable small, e.g. increasing the tuple size 16 times (64 to 1024 bytes) causes a decrease of about 10% in the system throughput. Therefore, the good throughput of the system is due to the low processing required at server side and the batch message ordering implemented in PAXOS protocol [6].

6. FINAL REMARKS

The paper presents a solution for the implementation of an intrusion-tolerant tuple space. The proposed architecture integrates several dependability and security mechanisms in order to enforce the required properties. This architecture was implemented in a system called DEPSPACE.

Another interesting aspect of this work is the integration of replication with confidentiality. To the best of our knowledge, this is the

[1]Our algorithm is an extension to PAXOS AT WAR to provide total order multicast. It differs from BFT [6] since we assume reliable channels instead of using checkpoints.

[2]We do not present experiments with more servers due to space constraints. However we could say that our system suffers the same scalability problems of other protocols with message complexity $O(n^2)$.

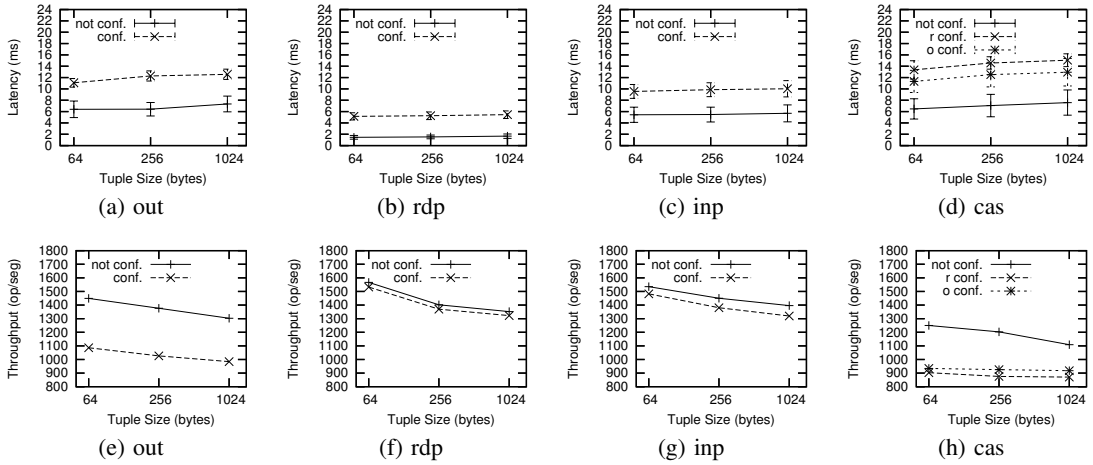


Figure 2: Latency and throughput of DEPSPACE operations considering different tuple sizes.

first paper to integrate state machine replication and confidentiality of data stored in the servers. Somewhat surprisingly, this integration is not trivial and the use of secret sharing fundamentally weakens the semantics of state machine replication in a Byzantine-prone environment (linearizability is not unconditionally ensured).

All code used in DEPSPACE is available at the *JITT (Java Intrusion Tolerance Tools)* project homepage: `http://www.das.ufsc.br/~neves/jitt`.

Acknowledgements

This work was supported by CNPq (Brazilian National Research Council) through process 550114/2005-0 and CAPES/GRICES (project TISD).

7. REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Mar. 2004.

[2] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programing in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, Mar. 1995.

[3] A. N. Bessani, M. Correia, J. da Silva Fraga, and L. C. Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, July 2006.

[4] N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro. SecSpaces: a data-driven coordination model for environments open to untrusted agents. *Elect. Notes in Theoretical Computer Science*, 68(3), 2003.

[5] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2), 2000.

[6] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4), 2002.

[7] J. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218, 1985.

[8] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, Jan. 1985.

[9] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[10] N. H. Minsky, Y. M. Minsky, and V. Ungureanu. Making tuple-spaces safe for heterogeneous distributed systems. In *Proc. of the 15th ACM Symposium on Applied Computing - SAC 2000*, 2000.

[11] A. Mostefaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. E. Abbadi. From static distributed systems to dynamic systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems - SRDS 2005*, pages 109–118, Oct. 2005.

[12] A. Murphy, G. Picco, and G.-C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3), 2006.

[13] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4), 1990.

[14] B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Proc. of the 19th International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, pages 148–164, Aug. 1999.

[15] E. J. Segall. Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing - SPDP'95*, pages 320–327, Oct. 1995.

[16] P. Verssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[17] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, June 2004.

Refined Quorum Systems

[Position Paper] *

Rachid Guerraoui
IC, EPFL
CH-1015, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Marko Vukolic
IC, EPFL
CH-1015, Lausanne, Switzerland
marko.vukolic@epfl.ch

ABSTRACT

It is considered good distributed computing practice to devise object implementations that tolerate contention, periods of asynchrony and a large number of failures, but perform fast if few failures occur, the system is synchronous and there is no contention. This paper initiates the first study of quorum systems that help design such implementations. Namely, our study of quorum systems encompasses, at the same time, the optimal resilience of distributed object implementations (just like traditional quorum systems), as well as their *optimal best-case complexity* (unlike traditional quorum systems).

We introduce the notion of a *refined quorum system* (RQS) of some set S as a set of three refined classes of subsets (quorums) of S: first class quorums are also second class quorums, which are also third class quorums. First class quorums have large intersections with all other quorums, second class quorums might have slightly smaller intersections with those of the third class, the latter simply correspond to traditional quorums. Intuitively, under uncontended and synchronous conditions, a distributed object implementation would expedite an operation if a quorum of the first class is accessed, then degrade gracefully depending on whether a quorum of the second or the third class is accessed. Moreover, we show that our RQS is, in a sense, minimal (i.e., necessary and sufficient), for optimally resilient and best-case optimal implementations of two fundamental Byzantine-resilient objects — *atomic storage* and *consensus*.

RQS are devised assuming a general adversary structure, and this basically allows algorithms relying on RQS to relax the assumption of independent process failures, often questioned in practice.

1. INTRODUCTION

Quorum systems are powerful mathematical tools to reason about distributed implementations of shared objects including read/write storage (e.g., [3, 16, 22]) and consensus [8, 18, 28]. In particular, quorum systems have been used to reason about implementations that tolerate asynchrony and are optimally resilient to process failures. Originally, a quorum system was defined as a set of subsets that intersect [9], and this notion was key to reasoning about crash-resilient asynchronous algorithms. More sophisticated forms of quorum systems have been introduced to cope with Byzantine (malicious) failures [21]: these require larger intersections among subsets (i.e., quorums) [22].

Perhaps surprisingly, most recent distributed object implementations, e.g., [1, 4, 5, 7, 10, 11, 20, 23, 26, 29] make little use of an abstract quorum notion. The absence of such a notion makes it in particular difficult to move away from a threshold-based adversary structure with the assumptions of independent and uniformly distributed failures, often questioned in practice, to a general adversary structure. The reason for this absence is, we believe, because traditional quorum notions (be they simple or Byzantine), while very useful to reason about the resilience dimension, are not adequate to capture the complexity dimension, specifically the *best-case* one. The implementations in [1,4,5,7,10,11,20,23, 26,29] were indeed devised to tolerate worst-case conditions, namely a large number of failures, arbitrarily long periods of asynchrony and contention. Motivated by practical considerations however, these implementations are also *optimistic* and geared to reduce best-case complexity, i.e., performance under situations of synchrony and no-contention, which are typically argued to be frequent in practice. As a consequence of their optimism, these implementations expedite operations in uncontended and synchronous situations, provided *"enough"* servers are accessed. Precisely capturing this very notion of *"enough"* in general terms was the motivation of this work.

This paper introduces the notion of *refined quorum systems* (RQS). In short, a refined quorum system of some set of elements S is a set of three classes of subsets (quorums) of S: first class quorums are also second class quorums, which are also third class quorums. Quorums (subsets of S) of the first class have large intersections with quorums of other classes, those of the second class might have slightly smaller intersections with those of the third class, the latter simply correspond to traditional quorums. In the context of a distributed object implementation, a set S would typically be the set of fault-prone server processes over which some object abstraction (e.g., storage or consensus) is implemented.

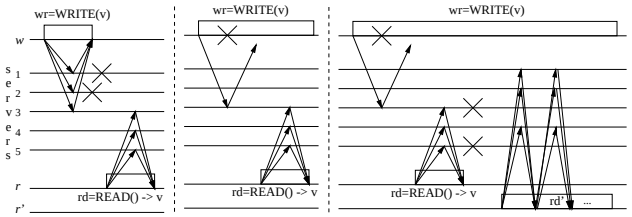
1.1 Example

To illustrate the intuition behind refined quorums, con-

*Full paper is available as a EPFL Technical Report LDP-REPORT-2007-002 (submitted for publication). We thank Hagit Attiya, Christian Cachin, Petr Kouznetsov and Eric Rupert for their very helpful comments.

sider the simple context of a crash-resilient implementation of an atomic storage over a set of server processes [3]. It is known [6] that no optimally resilient atomic storage algorithm can have both reads and writes complete in a single communication round-trip (we simply say round), even if a single writer is involved (SWMR). For instance, the classical, optimally crash-resilient solution [3] (that assumes a majority of correct processes) requires two rounds for a read.

As we discussed earlier, it is practically appealing to look into best-case complexity and ask if it is possible to expedite *both* reads and writes within a single round in a synchronous and contention-free period. Clearly, if the reader (resp. the writer) access all servers in the first round, then it can immediately return a valid response. But do we need to access all servers? How many servers actually *need* to be accessed to achieve such a *fast termination* in best-case conditions?



(a) ex2 (b) ex3 (c) ex4

Figure 1: Violation of atomicity in case the single-round operations access only 3 servers.

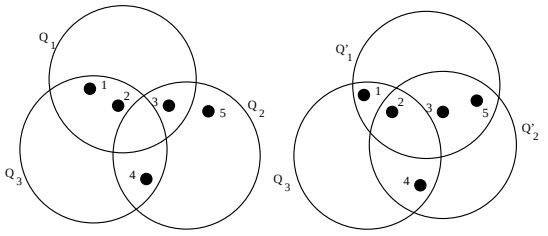
Consider the following simple example of 5 servers that implement a crash-tolerant atomic storage assuming $t = 2$ server failures (optimal resilience). We argue below that any algorithm that greedily expedites read/write operations in one round during synchronous and contention-free periods whenever $S - t = 3$ servers are accessed, violates atomicity. This is depicted through several executions of such an algorithm (Figure 1):

1. In $ex1$, the writer w invokes $wr = \mathsf{write}(v)$ and servers 4 and 5 are faulty. Then, wr writes the value v into the subset of servers $Q_1 = \{1, 2, 3\}$ and completes in a single round.

2. $Ex2$ (Fig. 1(a)) is slightly different because servers 4 and 5 are actually correct. Yet wr also completes in a single round, after writing in Q_1. Then servers 1 and 2 crash and a read rd (by the reader r) is invoked. Assuming synchrony and no contention, rd accesses server set $Q_2 = \{3, 4, 5\}$ and completes in a single round.

3. $Ex3$ (Fig. 1(b)) is similar to $ex2$ except that (1) the write is incomplete and writes only to server 3, (2) servers 1 and 2 (i.e., servers from the set $Q_2 \setminus Q_1$) are correct, but the communication between the reader and the servers from $Q_2 \setminus Q_1$ is delayed. Read rd does not distinguish $ex3$ from $ex2$ and completes in a single round, returning v.

4. Finally, consider $ex4$ (Fig. 1(c)) that extends $ex3$ by: (1) the crash of servers 3 and 5 and (2) the invocation of read rd' by a different reader r'. This reader cannot return v using $Q_3 = \{1, 2, 4\}$ regardless of how many rounds are used. Atomicity is violated.



(a) $|Q_1| = |Q_2| = 3$ (b) $|Q'_1| = |Q'_2| = 4$

Figure 2: Quorum intersections

Essentially, atomicity is violated because $Q1 \cap Q2 \cap Q3 = \emptyset$ (Figure 2(a)). On the other hand, we can devise a storage algorithm that achieves fast termination whenever 4 servers are accessed. For instance:

- A write wr completes in a single round only if it writes v to 4 servers, say $Q'_1 = \{1, 2, 3, 5\}$. A subsequent single-round read rd will also have to access at least 4 servers, say $Q'_2 = \{2, 3, 4, 5\}$ (including at least 3 servers from Q'_1). A subsequent read rd' that accesses some subset Q_3 of 3 servers will surely learn about v since there is a set $X = Q'_1 \cap Q'_2$ of (at least) 3 servers that witnessed both wr and rd, and X intersects with any set of 3 servers. This server in the intersection will inform rd' about the value written by wr.

The key to atomicity is that $Q'_1 \cap Q'_2 \cap Q_3 \neq \emptyset$. Namely, (Figure 2(b)) in a system of 5 elements, any two subsets of 4 elements intersect with any subset of 3 elements. Basically, boosting complexity requires to access subsets of servers that have larger intersections than traditional quorums. The above example is (relatively) simple because we were interested in the necessary and sufficient intersection properties considering: a) crash failures, b) threshold-based quorums and c) no graceful degradation.

The idea behind our notion of *refined quorum system* is precisely to characterize the required intersection properties in a precise and general manner. We aim at a characterization that is necessary and sufficient for optimizing the best-case complexity of various distributed object implementations, in various failure models, under various adversary structures, and also considering graceful degradation.

1.2 Contributions

Intuitively, under uncontended and synchronous conditions, a distributed object implementation would expedite an operation if a quorum of the first class is available, then degrade gracefully, depending on whether a quorum of the second or the third class is available. We argue that our quorum notion is, in a sense, complete: there is no reason for further refinement of quorums with the goal of optimizing best-case efficiency, since the properties provided by our third class quorums are anyway necessary for hindering the partitioning of the asynchronous system, which is key to any resilient distirbuted object implementation.

Our refined quorum systems are designed to handle a general adversary structure expressing situations where an adversary controls subsets of processes in a specific manner [15, 22]. As a consequence, this allows algorithms designed with such quorums to relax the assumption of independent process failures, often criticized in practice. In

the full paper [13] we illustrate the power of our notion of RQS by introducing two new atomic object implementations. Each algorithm is interesting in its own right and is, in a precise sense, the first fully optimal protocol of its kind.

- Our first object implementation is a new Byzantine-resilient asynchronous distributed storage algorithm. Such algorithms constitute an active area of research and are appealing alternatives to classical centralized storage systems based on specialized hardware [27]. The challenge when devising storage algorithms is to ensure that *reads* and *writes* have low latency in most frequent situations, while (a) tolerating the failures of a large number of base servers (typically commodity disks) as well as any number of clients that access the storage (wait-freedom [14]) and (b) ensuring strong consistency (ideally atomicity [17]). Using RQS, we present an atomic wait-free storage algorithm that combines optimal resilience with the lowest possible *read/write* latency in best-case conditions (synchrony and no-contention). Under such conditions, our algorithm expedites storage operations (*reads* and *writes*) in a single round if a first class quorum is accessed, in two rounds if a second class quorum is accessed and in three rounds otherwise. The latter case is when a third class quorum is available which is a necessary condition for resilience anyway. Our algorithm does not use any data authentication primitive, and matches the resilience and complexity lower bounds of [11, 24] when these are extended to a general adversary structure, together with a new complementary bound. Our new bound captures the best-case complexity of gracefully degrading atomic storage implementations.
- Our second algorithm implements a Byzantine-resilient consensus abstraction in the general state machine replication (SMR) framework of [18], distinguishing different process roles: *proposers* that propose values to be learned by *learners* with the mediation of *acceptors*. Our algorithm is the first to tolerate (1) any number of Byzantine failures of proposers and learners, (2) the largest possible number of acceptor failures, and (3) arbitrarily long periods of asynchrony. On the other hand, under best-case conditions, our algorithm allows a value to be learned in only two message-delays in case a first class quorum is accessed, and in three (resp., four) message delays in case a second (resp., third) class quorum is accessed. Note here that (a) learning in a single message delay is obviously impossible with multiple or potentially Byzantine proposers, and (b) the availability of a third class quorum is anyway necessary for resilience. Our algorithm matches the resilience and complexity lower bounds of [19] when these are extended to a general adversary structure, together with a new complementary bound on consensus algorithms that degrade gracefully in best-case executions. These bounds state minimal conditions under which the SMR approach can be made optimally resilient and best-case efficient. Until now, it was not clear whether the conditions of [19] were also sufficient. We show they are and we complement them.

In this position paper we first present our quorum notion and illustrate how it generalizes previous ones through examples from the literature. Then, we point out some open research directions. We postpone the detailed model as well as our algorithms and their proofs to the full paper [13].

2. REFINED QUORUM SYSTEMS

Definition of our refined quorum system is expressed in an environment including S a non-empty set of elements, and an *adversary structure* (or, simply, *adversary*) $\boldsymbol{B}$ defined as follows [15]. Let $\boldsymbol{B}$ be any set of subsets of S. $\boldsymbol{B}$ is an *adversary* (for the set S) if: $\forall B \in \boldsymbol{B}$: $B' \subseteq B \Rightarrow B' \in \boldsymbol{B}$.

Let $\boldsymbol{RQS}$ be any set of subsets of S.

Definition 1. **Refined Quorum System.** We say that $\boldsymbol{RQS}$ is a *refined quorum system* for a set S and adversary $\boldsymbol{B}$, if $\boldsymbol{RQS}$ has two subsets $\boldsymbol{QC_1} \subseteq \boldsymbol{QC_2} \subseteq \boldsymbol{RQS}$ such that the following properties hold: (every $\boldsymbol{QC_i}$ is called a *quorum class*, and elements of $\boldsymbol{QC_i}$ are called *class i elements*)

Property 1. An intersection of any two elements of $\boldsymbol{RQS}$ is not an element of $\boldsymbol{B}$, i.e.,

- $\forall Q, Q' \in \boldsymbol{RQS}$: $Q \cap Q' \notin \boldsymbol{B}$.

Property 2. The intersection of any two class 1 elements and any element of $\boldsymbol{RQS}$ is not a subset of the union of any two elements of $\boldsymbol{B}$, i.e.,

- $\forall Q_1, Q_1' \in \boldsymbol{QC_1}$, $\forall Q \in \boldsymbol{RQS}$, $\forall B_1, B_2 \in \boldsymbol{B}$: $Q_1 \cap Q_1' \cap Q \nsubseteq B_1 \cup B_2$.

Property 3. The intersection of any class 2 element Q_2 and any element Q of $\boldsymbol{RQS}$ is:
(a) not a subset of the union of any two elements of $\boldsymbol{B}$ (we say $P_{3a}(Q_2, Q)$ holds), *or*
(b) its intersection with every class 1 element[1] is not an element of $\boldsymbol{B}$ (we say $P_{3b}(Q_2, Q)$ holds), i.e.,

- $\forall Q_2 \in \boldsymbol{QC_2}$, $\forall Q \in \boldsymbol{RQS}$, $\forall B_1, B_2 \in \boldsymbol{B}$: $(Q_2 \cap Q \nsubseteq B_1 \cup B_2) \vee$ $\vee (\boldsymbol{QC_1} \neq \emptyset \wedge \forall Q_1 \in \boldsymbol{QC_1}$: $|Q_1 \cap Q_2 \cap Q| \notin \boldsymbol{B})$.

We simply call elements of a refined quorum system — *quorums*. In addition, for simplicity, we sometimes refer to any quorum that is not a class 2 quorums as a *class 3* quorum, and write $\boldsymbol{QC_3} = \boldsymbol{RQS}$. Note that class 1 quorums are also class 2 quorums, which are also class 3 quorums. Notice also that, when $\boldsymbol{QC_1} = \boldsymbol{QC_2}$, Property 2 implies Property 3. Furthermore, when $\boldsymbol{B} = \emptyset$, Property 1 implies Property 3. Therefore, Property 3 is interesting on its own only if $\boldsymbol{B} \neq \emptyset$ and $\boldsymbol{QC_1} \neq \boldsymbol{QC_2}$.

2.1 Examples

We denote by $\boldsymbol{B_k}$ a *k-bounded threshold adversary*, a special case of an adversary that contains all subsets of S with cardinality at most k (i.e., $\boldsymbol{B_k} = \{B | B \subseteq S \wedge |B| \leq k\}$). Moreover, we denote by $\boldsymbol{Q_i}$ the set of subsets of S that contains all subsets of S that contain all but at most i elements of S, i.e., $\boldsymbol{Q_i} = \{P | P \subseteq S \wedge |P| \geq |S| - i\}$.

Example 1. Figure 3 depicts a simple illustration of a RQS for an adversary $\boldsymbol{B_1}$: 4 quorums are involved. As depicted by the example, the cardinality of a quorum might not be a good indication of its class: it is the intersection with others that matters. Quorum Q_1 contains 5 elements and is a class 1 quorum, while Q' contains 6 elements yet is only an ordinary (or class 3) quorum.

[1]Assuming there is at least one class 1 element, i.e., $\boldsymbol{QC_1} \neq \emptyset$.

To see that the quorum system in Figure 3 is indeed a RQS, note that every pair of depicted sets intersects in at least 2 elements (satisfying Property 1). Q_1 intersects with every other set in at least 3 elements (satisfying Property 2, for an intersection with itself). Moreover, $P_{3a}(Q_2, Q')$ and $P_{3a}(Q_2, Q_1)$ hold (since $|Q2 \cap Q'| = 2 = |Q2 \cap Q1|$) as well as $P_{3b}(Q_2, Q)$ (since $|Q_2 \cap Q \cap Q_1| = 2$). Hence, $\boldsymbol{RQS} = \{Q, Q', Q_2, Q_1\}$ is a refined quorum system, where Q_1 (resp., Q_2) is a class 1 (resp., class 2) quorum.

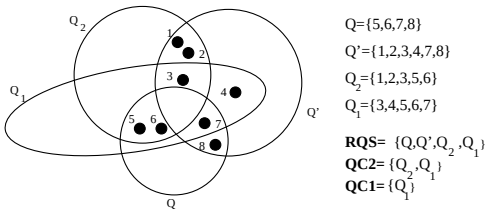


Figure 3: A simple example of a RQS

In the following, we give more illustrations of our quorum notion by explaining how it generalizes traditional ones. In the following, an adversary $\boldsymbol{B}$ for a set of processes S contains all subsets of S that can simultaneously be Byzantine. Note that, in this paper, we do not consider a process that simply crashes to be Byzantine.

Example 2. Consider the case where: (a) $\boldsymbol{B} = \emptyset$, (b) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ and (c) $\boldsymbol{RQS} = \boldsymbol{Q}_{\lfloor(|S|-1)/2\rfloor}$ (i.e., every majority subset of S is a quorum). Property 1 is trivially satisfied. So are Properties 2 and 3, since $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$. This quorum system is typically used when devising algorithms that tolerate (a minority of) crash-failures, e.g., [3, 9, 18, 25].

Example 3. Consider the case of an adversary $\boldsymbol{B}_{\lfloor|S|/3\rfloor}$, where (a) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ and (b) $\boldsymbol{RQS} = \boldsymbol{Q}_{\lfloor(|S|-1)/3\rfloor}$. In this case, each quorum contains more than two thirds of processes and satisfies Property 1. Properties 2 and 3 are also satisfied (since $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$). Such a quorum system is typically used to tolerate (up to one third of) Byzantine failures, e.g., [4, 5, 21, 24].

Example 4. The RQS for which $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ is a *disseminating quorum system* in the sense of [22]. In [22], disseminating quorum systems are used to build resilient distributed services that store authenticated (also called self-verifying) data. On the other hand, the RQS in which $\boldsymbol{QC_1} = \emptyset$ and $\boldsymbol{QC_2} = \boldsymbol{RQS}$ is a *masking quorum system* in the sense of [22]. These systems have been used to build resilient distributed services that store unauthenticated data.

So far, in examples 2-4, we considered refined quorum systems in which $\boldsymbol{QC_1} = \emptyset$. In the rest of the paper, we study the more general case where $\boldsymbol{QC_1} \neq \emptyset$. This is the case where our refined quorum systems capture both the resilience and the best-case complexity dimensions of distributed algorithms.

Example 5. Consider the case of a RQS where $\emptyset \neq \boldsymbol{QC_1} = \boldsymbol{QC_2}$. Such a RQS was implicitly used in [20], for the specific case $\boldsymbol{B} = \emptyset$, to devise a consensus algorithm that tolerates asynchronous periods and a threshold t of process (crash) failures, yet expedites decisions in best-case scenarios. In fact, although not used in the algorithm, the idea of a *fast* quorum (class 1 quorum in our terminology) was used to explain the algorithm. In the special case of an adversary $\boldsymbol{B_k}$, where (a) $\boldsymbol{RQS} = \boldsymbol{Q_t}$, and (b) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \boldsymbol{Q_q}$ ($q \leq t$), Property 2 is satisfied if $|S| > 2q + t + 2k$ and Property 1 is satisfied if $|S| > 2t + k$. These inequalities correspond to Lamport's lower bounds for "asynchronous" consensus [19]. The special case $k = q = t$ (i.e., where $\boldsymbol{QC_1} = \boldsymbol{RQS}$) corresponds to the refined quorum system implicitly used in [1, 23], also with the goal of boosting consensus latency.

Example 6. Even more interesting is the general case where $\emptyset \neq \boldsymbol{QC_1} \neq \boldsymbol{QC_2} \subseteq \boldsymbol{RQS}$ (e.g., Fig. 3), especially when $\boldsymbol{RQS}$, $\boldsymbol{QC_1}$ and the adversary are defined as in Example 4, $\boldsymbol{QC_2} = \boldsymbol{Q_r}$, and $0 \leq q < r \leq t$. In other words, each quorum contains all but at most t processes, while class 1 (resp., class 2) quorums contain all but at most q (resp., r) elements. $\boldsymbol{RQS}$ satisfies (i) Property 1 if $|S| > 2t + k$, (ii) Property 2 if $|S| > t + 2k + 2q$, and (iii) Property 3 if $|S| > t + r + k + min(k, q)$, i.e., $\boldsymbol{RQS}$ is a refined quorum system if $|S| > t + k + max(t, k + 2q, r + min(k, q))$. This quorum system was implicitly used in [7, 11], and later in [29].

3. CONCLUDING REMARKS

This paper introduces the notion of *refined quorum systems* (RQS) and argues that this is a useful notion to reason about optimally resilient and efficient distributed object implementations. We show [13] that RQS are necessary and sufficient (or, in a sense, minimal) for implementing an important class of *atomic* objects, namely atomic storage and consensus. This minimality holds when we indeed require atomicity and do not rely on authentication primitives to cope with Byzantine failures in best-case executions.

In the above context, and roughly speaking, denoting the best possible latency of an object implementation by l_1[2] (i.e., 1 round in the case of storage, or 2 message delays in the case of (Byzantine and asynchronous [19]) consensus), and by l_2 and l_3, incrementally, the next best possible latencies according to the corresponding metric, we proposed RQS-based object implementations that achieve a latency of l_i whenever a quorum of class i is available and best-case conditions (namely, synchrony and no-contention) are met. Since Property 1 of RQS (defined on class 3 quorums) is anyway necessary for any resilient implementation of distributed storage and consensus in an asynchronous environment, there is no need for refining quorums further. It might be important to notice here that the very notion of refined quorum system helps highlight the information structure of optimally resilient and best-case efficient atomic object implementations (at least those implementing the abstractions of atomic storage or consensus). Basically, these implementations inherently have three rounds together with a backup subprotocol in case of asynchrony or contention.

Our study opens at least two research directions. First, it would be interesting to carefully look into non-atomic semantics, e.g., regular or safe storage [17]. Recent results (in the threshold-based context) suggest that some (yet not all) properties of our refined quorum system are necessary and sufficient even for achieving optimal best-case complexity of weaker object implementations. Namely [2, 12] suggest that Properties 1 and 3a of RQS are necessary and sufficient for non-atomic best-case efficient storage implementations. These properties correspond to the special case

[2]This can be measured by the best possible latency in synchronous, uncontended and failure-free situations.

of RQS where $\boldsymbol{QC_1} = \emptyset$. Second, it would also be interesting to look into atomic object implementations that use data authentication in best-case executions. The lower bounds of [19], stated in the threshold-based context, suggest that Properties 1 and 2 are necessary and sufficient for best-case efficient and optimally resilient consensus implementations regardless of whether authentication is used in the best-case. These properties correspond to the special case of RQS where $\boldsymbol{QC_2} = \boldsymbol{QC_1}$. This suggests a general RQS-based framework for optimally efficient and resilient distributed objects, parameterized by the use of authentication and the desire for atomicity.

4. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the 20th ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM Press.

[2] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, USA, February 1999.

[5] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementations*, Seattle, Washington, November 2006.

[6] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing*, pages 236–245, New York, NY, USA, 2004. ACM Press.

[7] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report 200499, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2005.

[8] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[9] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.

[10] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144, 2004.

[11] R. Guerraoui, R. R. Levy, and M. Vukolić. Lucky read/write access to robust atomic storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 125–136, Washington, DC, USA, 2006. IEEE Computer Society.

[12] R. Guerraoui and M. Vukolić. How Fast Can a Very Robust Read Be? In *25th ACM Symposium on Principles of Distributed Computing*, 2006.

[13] R. Guerraoui and M. Vukolić. Refined quorum systems. Technical Report LPD-REPORT-2007-002, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, February 2007.

[14] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[15] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *Proceedings of the 16th annual ACM symposium on Principles of distributed computing*, pages 25–34, New York, NY, USA, 1997. ACM Press.

[16] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[17] L. Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, May 1986.

[18] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[19] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, Springer Verlag (LNCS), pages 22–23, 2003.

[20] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[21] L. Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[22] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[23] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[24] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325. Springer-Verlag, 2002.

[25] M. Naor and A. Wool. The load, capacity and availability of quorum systems. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 214–225, 1994.

[26] H. V. Ramasamy and C. Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 88–102, December 2005.

[27] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004.

[28] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[29] P. Zieliński. Optimistically terminating consensus. Technical Report UCAM-CL-TR-668, Cambridge University, Cambridge, UK, June 2006.

Secure Lookup without (Constrained) Flooding

Bobby Bhattacharjee*
University of Maryland
College Park, Maryland, USA

Rodrigo Rodrigues*
Instituto Superior Técnico
(Technical Univ. of Lisbon)
and INESC-ID
Lisbon, Portugal

Petr Kouznetsov
Max Planck Institute for
Software Systems
Saarbrücken, Germany

1. INTRODUCTION

We present a new protocol for secure routing in overlay networks. Our protocol exports the same functionality as regular decentralized lookup protocols [9, 10, 12, 14, 15]. Moreover, the existing routing protocols can be enriched with the security primitives we introduce. Recall that a routing protocol exports a lookup operation that, given a key in a virtual identifier space (the *id space*), locates the node (or the group of nodes) that are, in a well-defined sense, the closest to the key. The routing primitive can then be used, e.g., to implement a secure Distributed Hash Table (DHT), a popular abstraction for publishing and retrieving items in a decentralized manner.

Originally, decentralized lookup protocols assumed that nodes are cooperative (follow their specifications unless they fail by crashing) and provided the lookup operation in $O(\log(N))$ network hops where N is the number of participating nodes. There have been a number of improvements to the base schemes in the cooperative scenario, including reduction of some of the operations to an amortized constant [7].

All of these protocols, however, are susceptible to various attacks if the underlying assumption that nodes are cooperative is violated. In particular, malicious nodes may misroute or simply drop protocol messages. Even a small fraction of compromised nodes can adversely affect all routing guarantees. There have been a number of prior efforts towards securing a routing protocol. Castro et al. [2] and Fiat et al. [4] both consider systems where at most $\frac{1}{4}$ of the nodes are malicious. In order to securely forward messages, Castro et al. relies on *redundant routing*, which floods the message along multiple paths. Fiat et al.'s solution, S-Chord, groups sets of contiguous nodes into *swarms*. Nodes flood requests to every node in a swarm, which requires $O(\log^2 n)$ messages. A different attack model is considered in [8] — here, each node can be mapped to its autonomous system (AS), and adversaries are constrained to at most k. Within these ASs, there can be unlimited number of adversaries (perhaps even totaling more then $\frac{1}{4}$ of all nodes in the system). Like S-Chord, the protocol in [8], also divides the id space into a set of contiguous neighborhoods, and uses Byzantine agreement to secure each neighborhood.

In this paper, we propose a decentralized routing protocol that addresses the security issues by using novel challenge-response mechanisms and mobile proactive secret sharing. Our protocol was inspired by [8]; however, we address the general problem of an f-fraction of malicious nodes. Our system is secured using a system-wide public/private key pair, and the private key is stored in a distributed manner amongst system participants. We use threshold cryptography [6, 1, 16, 13] to ensure that no single (or small set of) node(s) has the system private key at any point in time. Since the private key is threshold distributed, some malicious nodes might hold key shares. Crucially, we show the set of shares held by bad nodes (including all shares over all time) can never be used to reconstruct the private key.

The key idea in our protocol is the notion of a *challenge* — when a (good) node receives a negative answer (e.g., when a lookup fails), it can challenge other nodes whether the routing data they provided is correct. The nodes must answer the challenge and sign the answer using the system-wide public key. Since good nodes must be involved in signing a message, a correctly answered challenge implies that the negative condition, in fact, exists in the system. This use of challenges and threshold cryptography was also inspired by previous work [11].

Our system provides several attractive features compared to prior approaches: the lookup path maintains the logarithmic complexity of original protocols, and we provide (easily) provable guarantees. In particular, we prove that (with configurably high probability) as long as challenges are correctly answered, the system state is not compromised. Furthermore, unlike previous secure routing protocols, we do not impose limits on the fraction of compromised nodes in the system.

The tradeoff of our protocol is in its computational complexity — threshold signature and key redistribution protocols are expensive, and our protocol must periodically incur a high key re-sharing cost. However, we believe this high over-

*Work done while visiting the Max Planck Institute for Software Systems

head can be reduced using more sophisticated cryptography, and the current protocol is instructive in its design and in the simplicity with which it enables global security properties to be asserted. Indeed, our goal here is not to present a fully optimized (or perhaps even a deployable) protocol; instead it is to explore a new part of the secure routing design space using first principles.

As a side contribution, we propose an interesting replication scheme that may be used (outside the context of this paper) in the design of large-scale intrusion-tolerant systems. In particular, we use a proactive threshold signature protocol in a way that provides very strong safety properties (we can set the failure threshold arbitrarily high), but may incur liveness problems if there are more than 1/3 faulty replicas. To address the liveness problems we present a takeover protocol where the system regains liveness by having some groups taking over the responsibilities of groups that have halted.

In the rest of this paper, we explain how the protocol boots and maintains its invariants through node joins, leaves, and attacks.

2. PROTOCOL

We model the system as a dynamic collection of nodes that are able to communicate with each other thorup exchanging messages.

Each node has a unique id that includes its IP address and its public key. There exists a one-way function h that maps every node (every object) to a unique point in the id space.

We assume a malicious adversary that is able to compromise a subset of nodes. We assume, however, that it is not in the power of the adversary to obtain many node ids (identities are "expensive") or to choose the positions of the nodes it controls in the id space.[1] Thus we assume incoming nodes can obtain a join certificate that can be validated by any system node. This assumption does not imply the need for an online global PKI. As long as the participants are willing to trust a certificate authority (CA), they only need to be seeded with this CA's public key. New nodes would get their ids generated at random and certified by the CA, and each node in the system could verify this new ID by verifying the CA's signature (using the CA public key they already have).

Further, we assume that at most fraction f of nodes, with ids chosen uniformly at random, can be compromised by the adversary within a bounded time period (a parameter of the system, called the *vulnerability window*).

The messages can be dropped, though the communication channels cannot produce or duplicate messages. We assume, however, that every two correct (non-compromised) nodes are able to eventually reliably communicate.

2.1 The Secure Routing Primitive

A secure key-based routing abstraction exports conventional membership operations for a new node to join and leave the system and the following operations to locate nodes in the system: *lookup*(x) and *secure-lookup*(x). Both operation return a set of nodes of size t that are, in a certain sense, close to x in the id space (we will call these the *neighbors* of x). Operation *lookup*(x) is the best-effort lookup operation exported by the conventional (insecure) routing schemes [9, 10, 12, 14, 15] and, in the normal operation case, it returns the set of neighbors of x. Secure lookup is typically invoked when the conventional lookup does not return a satisfactory result. E.g., in a typical implementation of DHT used for storing and locating self-verifiable data, a secure lookup primitive can be invoked when the nodes returned by the conventional lookup claim not to have the required data (i.e., we suspect that either the lookup instance or the node are compromised).

The *secure-lookup*(x) operation guarantees that, under the condition that the system membership eventually stabilizes and liveness properties of our system rely upon the assumption that there exist bounds on relative processing speeds and communication delays (which are however unknown to the nodes), there is a time after which all *secure-lookup*(x) operations return the same group of nodes (which are the correct neighbors of x).

2.2 Spans

We dynamically partition the id space into sub-intervals called *spans*. Initially, there is a single span in the system that consists of the entire id space, but, as we will detail later, spans can be split or merged as the system membership evolves.

For each span there is a *span committee*, a subset of the span nodes that is responsible for keeping track of the span membership (i.e., the subset of the current system members whose ids lie within the span), and for producing and disseminating a *span certificate*, which is an authenticated description of the span membership.

We envision that the span committee is chosen when the span is formed, and its composition will only change when some number of its members leaves the system, before the liveness of the committee is affected. Details of how exactly this choice is made are left for future work.

Our system is still in its design phase, and it is not clear what the precise span and span committee sizes will be (partly because this depends on the security parameters specified by the application). In Section 3 we provide a resilience analysis for different committee sizes, and we expect, in practice, the committee sizes to be between 12–25 nodes.

2.3 Threshold Cryptography

To authenticate span certificates, we use a proactive threshold signature scheme [6, 1, 16, 13], which allows spans certificates to be validated with a single, well-known public key, without relying on any particular committee member knowing the corresponding private key (since if it were faulty it could expose it).

In an (n, t) proactive threshold signature schemes, each one

[1] Note that this assumption rules out the famous Sybil attack [3]. Relaxing or validating this assumption is left for future work.

of n nodes (in this case, the committee members) holds a share of a secret, and the protocol will only generate a correct signature if t of these nodes agree on signing the same statement.

Furthermore, these protocols include a mechanism for share refreshment that produces a new set of shares from the old ones. In particular, we require a protocol where the set nodes that hold the shares can change as part of the share refreshment protocol (and for this reason we intend to use MPSS [13]).

Share refreshment is triggered when there is a change in the span committee. After the share refreshment protocol ends, the old nodes can discard their shares. This allows the protocol to work correctly (meaning informally that malicious nodes cannot produce a valid signature) provided that less than t in each set of share holders (committee members) are compromised during a window of vulnerability, which is the time interval during which these nodes are holding their shares.

2.4 System Operation

In this section we describe how the system works in the normal case, assuming, for now, a steady-state operation where there are no membership changes. This will clarify the importance of span certificates and span committees.

We explain our system in terms of the Pastry [12] protocol, where nodes maintain a leaf set (a set of neighboring nodes in the id space) and a routing table (a set of nodes in distant locations of the id space).

Nodes also maintain the current span certificate for their own span, and, for each entry in the routing table, a cached copy of the span certificate for the corresponding span.

The *lookup*(x) operation works exactly as in Pastry. Therefore it does not provide any guarantee that the answer is correct. On the contrary, the *secure-lookup*(x) operation must ensure that it returns the current set of neighbors of id x. This primitive starts by performing a normal lookup but the reply from the neighboring nodes must be accompanied by the span certificate for that span that includes id x.

Obtaining the span certificate is not enough, though, since this could be an old certificate, where a large fraction of the members had left the system or even be compromised by now. To ensure the freshness of the span certificate the client must issue a *challenge* to the span committee members. This is a random nonce that the client sends, which the span committee members must sign (with the threshold signature protocol) along with some digest of the span membership. Only the current span committee can produce such reply since old committee members would no longer hold the shares of the secret required to sign.

2.5 Join and Leave

Another crucial function of the span committee is to keep track of the changes in system membership. We now describe how the system handles nodes joining and leaving the system.

To join the system, the node must first obtain its join certificate and the address of one or more current system nodes (the bootstrap nodes) using some out-of-band mechanism.

Then it asks the bootstrap node to perform a lookup to the incoming node's id, and it also asks for the corresponding span certificate. This will enable the incoming node to find out about its leaf set and span committee. The incoming node can verify that the span committee is current in two ways: it can challenge it right now, or it can wait until the join operation succeeds to see if a new span certificate is produced containing itself.

The incoming node contacts the span committee members, asking them to join the span, and sending them its join certificate. Span committee members run the threshold signature protocol to produce a new span certificate that includes the new node. As they do this they exchange the join request among themselves, to deal with requests that did not reach all committee members. If multiple joins occur concurrently, the span committee members add all incoming nodes to the span certificate they are trying to produce.

The consequence of this operation is that the node is added to the span membership, and a new span certificate is produced and disseminated to the span members Possibly this may trigger a span split, which we describe later.

Once the joining node receives the new certificate containing itself, it does a normal Pastry join, which will update the Pastry structures. We need to deal with a bad incoming node that will run the operation on the span committee but will not contact its Pastry neighbors. To handle this case, system nodes will update their leaf sets when they realize that there is a new node in the span certificate that should be part of the respective leaf set.

Node departure.. Due to space constraints, we briefly summarize the protocol actions when a node departs (or crashes). We assume that nodes leave ungracefully, i.e. without notifying others. Instead, all nodes implement a protocol such as Rosebud [11] or PeerReview [5] to monitor and robustly detect when a neighbor has died or misbehaved. Once sufficient nodes declare a span member to be faulty, they contact the span committee and a new span certificate is computed.

2.6 Span Split and Merge

There are two system parameters that define the minimum and maximum number of elements in a span, s_{min} and s_{max}. This is important to avoid overloading span committees if the span is too populated, and to avoid creating a liveness problem if there aren't enough nodes in the span to form a committee.

After each join or leave operation concludes, nodes in the span must verify if the number of nodes in the span is still within these limits. If the number of nodes in the span is too high, the span is split in half, and two new committess are formed (one for each span). In this case the old committee must run two parallel instances of the share refreshment protocol with each one of the new committees.

If, on the other hand, the number of nodes in the span becomes too low, the span must merge with its neighboring span. In this case we can just pick one of the committees to take over the responsibility of the entire span to avoid running another share refreshment protocol.

2.7 Span Takeover

In some cases, a span committee may experience liveness problems. This could happen because there aren't enough non-faulty replicas to meet the threshold for signing statements, or because the membership turnover required a change in the composition of the committee, and the MPSS protocol requires more than 2/3 non-faulty replicas to provide liveness (even though there is no such bound for providing safety).

Our system includes a mechanism for recovering liveness in such cases called a *span takeover*. The idea is that periodically each span committee monitors the liveness of one of its adjacent span committees (e.g., in clockwise direction of the circular id space). Monitoring liveness consists of using the challenge mechanism described in Section 2.4. If a span committee detects that its neighboring span is not responding to challenges, it takes over the neighboring span. This means that the monitoring span will force a merge with its neighboring span, becoming responsible for its id interval.

Note that since we do not rely upon strong synchrony assumptions, the takeover protocol may be initiated against a live (but slow) span. This might lead to bounded periods when the ranges of two neighboring spans overlap. Such a situation is considered normal, since eventually, the spans' live nodes will not suspect each other and spans will resolve the conflict. In some cases, e.g. if a network partition persists for a long period, we may have two disjoint systems with independent routing guarantees. This is not different from what would occur in a normal routing overlay.

3. DISCUSSION

Space constraints will not permit us to present a comprehensive evaluation of either the security or the overhead of our protocol. Instead, we begin with a brief summary of overheads using the base share refreshment protocols (without optimizations). Next we present two specific attack scenarios and how our protocol is resilient — we believe the general security "theme" will be apparent from our informal analysis. Finally, we conclude this paper with a discussion of open areas of work.

3.1 Overhead

The threshold key redistribution protocol incurs $O(n^4)$ overhead, where n is the size of the span committee. Recall that in an (n, t) sharing scheme, n members get a key share, and at least t members are required to produce a valid signature. Hence, for the key to be exposed, at least t members in the committee have to be corrupt. Thus, given a fixed (f_e) expected fraction of malicious nodes, we can choose the values of n and t to make the probability of key exposure arbitrarily low.

In Table 1, we present the probabilities of key exposure for different values of f_e and n. For example, suppose 10% of the nodes are assumed to be malicious (with malicious node ids picked uniformly at random), and if the committee size is 20, then the expected number of bad nodes in the committee is 2. However, suppose we set the threshold t to be 10. Then the probability that all 10 nodes are corrupt is 0.0003 (second line in the table). These values were derived by upper bounding the mass in the tail of the PDF using the Chernoff-Hoeffding bounds.

Assuming reasonable key sizes (1024 bit DSA private keys), and the $O(n^4)$ overhead of the rekeying protocol, the total amount of data that is exchanged by a node during the rekey is approximately 8 MB (with a 20 node committee). The overhead drops to 4 MB per node if the committee size is reduced to 16.

Total shares	f_e	Probability of more than t bad nodes		
20	.1	> 8	$<$	0.0061
		> 10	$<$	0.0003
		> 12	$<$	1.01e-05
	.25	> 15	$<$	0.001
16	.05	> 6.4	$<$	0.0004
		> 7.2	$<$	8.1e-05
	.1	> 9.6	$<$	0.0001
		> 11.2	$<$	5.05e-06

Table 1: Probability of Key Exposure, assuming an expected fraction f_e of bad nodes.

3.2 Security Analysis

We consider two different attacks: the eclipse attack (in which the malicious nodes try to take over the entire neighborhood of a good node, hence isolating it from the system), and a data erasure attack (in which a malicious node pretends that data that was published in a DHT does not exist).

Eclipse Attack.. Suppose node i joins the system and i's ID is hashes to a malicious node j (i.e. j is the closest node to i in the system). Node j wants to isolate i, and provides i with a "bad" neighborhood certificate C. Note that if C is simply signed with the wrong key, i can immediately discard it. However, suppose C is signed with the correct key, but is old. Many more nodes have joined (and left) the system since C was produced, and moreover, many (or even all) of the alive nodes pointed to by C are now corrupt, but perhaps belong to different spans. If i were to use this certificate to seed its routing table, then it could be isolated (or eclipsed), since all its neighbors would always return other bad nodes as neighbors and so on. However, before accepting the certificate, i produces a nonce and challenges j to produce a valid signature that binds the nonce to C. Since the bad nodes do not have sufficient shares to sign (any statement), the challenge will fail, and i will not accept C. The only certificate that can be signed is the current valid one, which will ensure that i joins the group correctly.

Data Erasure.. Now consider a DHT implementation based on our secure routing primitive. Assume that a data item d is (successfully) published, and then i performs a lookup

for it. The lookup reaches malicious node j, which tries to convince i that the item does not exist. First, j has to convince i that j belongs to the span which covers d. If j does belong to the correct span, then it can just produce the current (valid) certificate. It is possible that j does not belong to the correct span, but has an old cached span certificate which did cover d. In both cases, when j returns a negative answer, i will issue a challenge, which j will not be able to respond to. Node i can then sequentially contact other nodes in the j's span and either get a pointer to a closer span or get the item d itself. (Note that the same argument also applies during forwarding, in case j asserts that it is not in d's span and is not aware of a closer span id).

4. SUMMARY

We have presented a new protocol for securing distributed hash tables. Our protocol does not penalize the lookup path, but does currently impose a periodic heavy overhead (due to the proactive secret sharing protocol we have used). Perhaps more importantly, the security properties of our protocol are relatively easy to verify, and a single challenge mechanism is used to secure the protocol against all forms of attacks.

5. REFERENCES

[1] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proc. 9th (ACM) conference on Computer and Communications Security*, pages 88–97. (ACM) Press, 2002.

[2] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI*, 2002.

[3] J. Douceur. The Sybil attack. In *IPTPS*, 2002.

[4] A. Fiat, J. Saia, and M. Young. Making Chord robust to Byzantine attacks. In *ESA*, 2005.

[5] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. In *HotDep*, 2006.

[6] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *ACM Conference on Computer and Communications Security*, pages 100–110, 1997.

[7] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, pages 53–65, 2002.

[8] R. Morselli. *Lookup Protocols and Techniques for Anonymity*. PhD thesis, University of Maryland, College Park, 2006.

[9] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA*, 1997.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.

[11] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. MIT LCS TR/932, Dec. 2003.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[13] D. Schultz, B. Liskov, and M. Liskov. Mobile proactive secret sharing. MIT CSAIL Research Abstract. `http://publications.csail.mit.edu/abstracts/abstracts06/das/das.html`, 2006.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[15] B. Zhao, K. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California, Berkeley, 2001.

[16] L. Zhou, F. B. Schneider, and R. van Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security*, 8(3):259–286, August 2005.

VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology

Hans P. Reiser
Departamento de Informática
Universidade de Lisboa, Portugal
hans@di.fc.ul.pt

Rüdiger Kapitza
Department of Compter Science 4
University of Erlangen-Nürnberg, Germany
kapitza@cs.fau.de

ABSTRACT

The use of virtualisation technology on modern standard PC hardware has become popular in the recent years. This paper presents the VM-FIT architecture, which uses virtualisation for realising fault and intrusion tolerant network-based services. The VM-FIT infrastructure intercepts the client–service interaction at the hypervisor level, below the guest operating system that hosts a service implementation, and distributes requests to a replica group. The hypervisor is fully isolated from the guest operating system and provides a trusted component, which is not affected by malicious intrusions into guest operating system, middleware, or service. Furthermore, the hypervisor allows the implementation of more efficient strategies for proactive recovery in order to cope with the undetectability of malicious intrusions.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; C.4 [**Performance of Systems**]: Fault tolerance

General Terms

Reliability

Keywords

Virtualisation, Byzantine fault tolerance, proactive recovery, innovative system architectures

1. INTRODUCTION

Services in distributed systems have become omnipresent, and the reliability of such services is becoming an increasingly significant issue for a growing class of applications. In the area of fault-tolerant systems, the toleration of malicious intrusions is one of the most difficult challenges. In order to enable the use of intrusion-tolerant concepts in a wide range of application domains, light-weight and efficient concepts are needed.

This paper investigates the use of virtualisation technology for the construction of fault and intrusion tolerant systems. Virtualisation provides a hypervisor component that is fully isolated from the guest operating system that hosts the actual service implementations. We identify two potential gains from virtualisation technology: First, the hypervisor is able to provide an isolated trusted component that does not have all the vulnerabilities of the guest systems hosting the actual service. Second, the hypervisor has full control over the guest systems, and thus can support an efficient proactive recovery of the guest system instances.

The paper is structured as follows. The next section discusses related work. Section 3 describes the core concepts of the VM-FIT (virtual machine – fault and intrusion tolerance) system architecture. Section 4 presents details of our current prototype implementation and discusses future work. Finally, Section 5 concludes.

2. BACKGROUND AND RELATED WORK

Frequently, software-based replication schemes are implemented in middleware systems such as fault-tolerant CORBA [12]. Most systems assume a crash-stop behaviour of nodes and cannot tolerate non-benign faults such as undetected random bit errors in memory, invalid states caused by software faults, or malicious intrusions of an attacker. Handling that kind of faults is the aim of Byzantine fault-tolerant mechanisms, such as the Castro-BFT algorithm [4] and the intrusion-tolerant SINTRA architecture [3]. One problem of intrusion-tolerant systems is that compromised components may remain undetected for extended periods of time. In the course of time, an attacker might obtain control of an increasing number of nodes, finally exceeding the maximum number of faulty nodes that the system is able to tolerate. Proactive recovery [13, 5, 16] is a technique that periodically refreshes nodes in order to remove potential intrusions; as a side effect, the refresh operation may also deploy new software versions that eliminate known vulnerabilities.

Virtualisation is an old technology that was introduced by IBM in the 1960s [8]. Systems such as Xen [1] and VMware [17] made this technology popular on standard PC hardware. Virtualisation enables the execution of multiple operating system instances simultaneously in isolated environments on a single physical machine.

While mostly being used for issues related to resource management, virtualisation can also be used for constructing fault-tolerant systems. Bressoud and Schneider [2] demonstrated the use of virtualisation for lock-stepped replication

of an application on multiple hosts. Besides such direct replication support, virtualisation can also help to encapsulate and avoid faults. The separation of system components in isolated virtual machines reduces the impact of faulty components on the remaining system [11]. Furthermore, the separation simplifies formal verification of components [18]. Using virtualisation is also popular for intrusion detection and analysis. Several systems transparently inspect a guest operating system from the hypervisor level [6, 7].

The RESH architecture [15] proposes redundant execution of a service on a single physical host using virtualisation. This approach allows the toleration of non-benign random faults such as undetected bit errors in memory, as well as N-version programming in order to tolerate software faults.

The contributions of this paper towards virtualisation-based intrusion tolerance differ from previously published approaches. Unlike strictly-coupled replication systems, we aim at replicating network-based services across heterogeneous nodes, using asynchronous communication networks. We use virtualisation to supply all nodes with a trusted entity that is fully isolated from the potentially faulty guest domain, which hosts a service together with an operating system and a middleware environment. Active replication of a service is complemented with support for proactive recovery, in order to eliminate potentially undetected intrusions into the guest domains. This way, we obtain mechanisms for proactive recovery that are more efficient than previous solutions.

3. VM-FIT ARCHITECTURE

The VM-FIT architecture provides generic support for the replication of network-based services. It transparently intercepts remote communication at the hypervisor level, and provides support for proactive recovery. We assume that the following properties hold:

- Clients use a request–reply interaction to access a remote network-based service.
- The service has deterministic behaviour, i.e., the service state and the replies sent to clients are uniquely defined by the initial state and the sequence of incoming requests.
- Byzantine faults may occur in a limited number of service replicas.

The first assumption allows the interception of client–service interaction at the network level. Together with the second assumption, the usual model for deterministic state machine replication is used. The failure model of the third assumption will be specified in more detail later on.

In the following, we adopt a terminology that is inspired by the Xen hypervisor [1]. The hypervisor is a minimal layer running at the bare hardware. On top, service instances are executed in *guest domains*, and a privileged *Domain 0* controls the creation and execution of the guest domains. In the following, we introduce the terminology of a *Domain NV* (network/voting), which handles the communication with clients and among replicas, and the voting over replica replies. The Domain NV is fully isolated from all guest domains. It may or may not be integrated into the usual Domain 0.

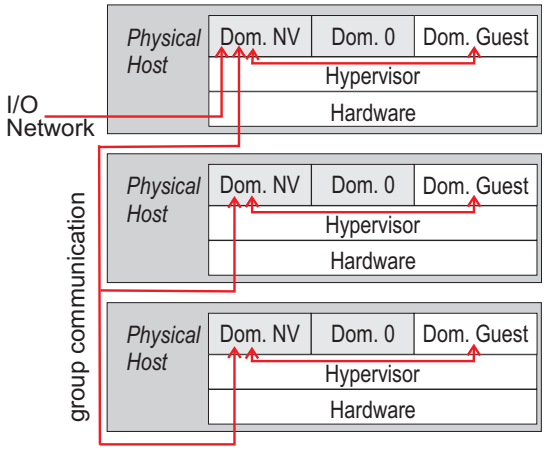


Figure 1: VM-FIT basic replication architecture

3.1 Replication Support

The basic system architecture is shown in Figure 1. The service replicas are running in isolated virtual machines in Domain Guest. The network interaction from client to the service is handled by the replication manager in the separate Domain NV. This manager intercepts the client connection and distributes all requests to the replica group using a group communication system. Each replica processes the client requests and sends a reply to the node that accepted the client connection. At this point, the replication manager selects the correct reply for the client using majority voting.

The first benefit from this architecture is a *transparent interception* of the client–service interaction, independent of the guest operating system, middleware, and service implementation. As long as the assumption of deterministic behaviour is not violated, the service replicas may be completely heterogeneous, with different operating systems, middleware, and service implementations.

The second benefit from this architecture is the support for a *composite fault model*. The architecture is composed of multiple isolated elements (Domain NV, Domain 0, Domain Guest), and each element may be subject to failures. For VM-FIT, we consider two possible variants:

- All system elements are potentially affected by Byzantine faults.
- The Domain NV (which contains the basic replication logic and network functionality, including network driver, communication stack, and group communication protocol) fails only by crash-stop of the complete host. The Domain Guest uses a Byzantine fault model, and thus can be subject to malicious intrusion. The same Byzantine model can be applied to the Domain 0, as long as the system makes sure that Domain NV is an protected, isolated entity that cannot be influenced from Domain 0.

The first variant is more powerful in terms of tolerable faults. It is able to tolerate intrusions even in the Domain NV. The drawback of this approach is that it requires more complex, Byzantine fault tolerant group communication protocols, compared to the second variant.

The second variant can be justified if the guest domain is a complex system with a legacy OS and a full-blown middleware, whereas the Domain NV executes in a fully isolated, lean domain. In the extreme case, the Domain NV could be an entity that can be subjected to full formal verification. In this case, simple, crash-stop group communication protocols can be used for distributing client requests. Thus, implementing consistency mechanisms in the privileged domain is expected to allow more efficient protocols than in systems that aim at tolerating Byzantine faults in all components of the system. Because of this aspect, we expect that our approach is best used in situations in which the first variant can be used. As a result, we obtain a hybrid system architecture that is able to use simple protocols for replication control, while still being able to tolerate malicious intrusions in the Domain Guest of a limited number of nodes.

In the case of a crash-stop Domain NV, the architecture can use majority voting for verifying client replies. In this case, the number of replicas must be $n \geq 2f + 1$ in order to tolerate up to f Byzantine faulty replica instances. In Section 4.4, we consider the execution of the group communication protocol in the non-crash-stop domains. In this case, Byzantine fault tolerant protocols are needed, which require $n \geq 3f + 1$ nodes.

3.2 Proactive Recovery

Proactive recovery increases the resilience of the replicated service, as the faulty nodes become rejuvenated periodically, and thus the upper bound of f tolerable faults is no longer required for the whole system lifetime, but only for the duration of a rejuvenation cycle. Proactive recovery requires a system component that controls the re-initialisation of the local service instance (including the operating system and middleware). For example, a tamper-proof external hardware might be used for rebooting the node from a secure code image. It is not feasible to trigger the recovery within a service replica, as a malicious intrusion might cause the replica component to ignore the required recovery.

In the VM-FIT architecture, the Domain NV can be used as a trusted entity that is able to completely re-initialise the target domain that hosts the service. For this purpose, all elements (i.e., operating systems, middleware, and service instance) need to be initialised with a "clean" state, by securely obtaining the service state from other replicas. As discussed by Sousa et al. [16], the recovery of a node has an impact on either the ability to tolerate faults or on the system availability. The VM-FIT architecture avoids the costs of using additional spare replicas for maintaining availability during recovery. Instead, it accepts the temporary unavailability during recovery, and uses the advantages of virtualisation in order to minimise this unavailability.

Unlike other approaches to proactive recovery, the hypervisor-based approach permits the initialisation of the rejuvenated replica instance concurrent to the execution of the old instance. The hypervisor is able to instantiate a second Domain Guest on the same hosts. After initialisation, the replication coordinator can shut down the old replica and trigger the activation of the new one. This way, the downtime of the service replica is minimised to the time necessary for the coordinated transition with a consistent state.

The state of the rejuvenated replica needs to be initialised on the basis of a consistent checkpoint of all replicas. As replicas may be subject to Byzantine faults and thus have an invalid state, the state transfer has to be based on an majority agreement of all replicas. For this purpose, the VM-FIT architecture is able to exploit the locality of the old replica version on the same host. The actual state is transferred locally, with a verification of its validity on the basis of checksums obtained from other replicas. Only if the local state is invalid, a remote state transfer becomes necessary. We discuss this state-transfer issue in more detail in the Section 4.3.

In summary, virtualisation allows a secure proactive recovery of service instances without additional hardware support. At the same time, it minimises downtime during recovery by creating a new replica instance in parallel to the running instance, and it enhances the state-transfer efficiency by transferring local state.

3.3 Application Areas

The proposed VM-FIT architecture can be applied to various kinds of applications, ranging from simple web applications to critical infrastructure. Applications using VM-FIT, however, have to meet the constraints defined at the start of this section.

The implementation of *new intrusion-tolerant applications* using VM-FIT is the easier variant. In this case, the developer is aware of the constraints of the service replication. Specifically, the application can make the guarantee of being deterministic, can provide interfaces for state transfer, and strictly adhere to a state machine principle.

It might, however, also be desirable to apply the VM-FIT architecture to *existing services* without or with only minimal modifications. This approach requires several steps. First, it must be verified if the application behaviour is strictly deterministic. One source of nondeterminism that is found in many applications is the support for multithreading. Handling multiple client requests with independent threads easily violates the state-machine principle by applying state changes to replicas in an inconsistent order. A further problem is the use of local address data in requests and replies. Each replica uses a different local communication address that is only known to the replication controllers in Domain NV, and in all communication with the external clients, the "outer" address of Domain NV has to be used. Because of these problems, we anticipate that the VM-FIT architecture cannot be used for transparently making existing applications intrusion tolerant. This does not mean that the architecture is useless for such existing services. As an example, in the next section we illustrate how a prototype implementation of VM-FIT can be used to replicate CORBA-based services.

4. PROTOTYPE IMPLEMENTATION

We have implemented a simple prototype of VM-FIT, which enables the transparent replication of a CORBA-based service. In the prototype, the Xen 3.0 hypervisor was used with Linux as operating system for Domain 0 and for Domain Guest. This virtualisation software has reached a high level of maturity, is available as open source, and supports a wide range of guest operating systems. This broad support is essential for the envisioned goal of providing a generic interception and replication architecture that is independent from specific operating systems.

4.1 Implementation Details

In the first prototype, no attempts towards formal verification of the trusted component have been made. Indeed, the same operating system, an off-the-shelf Linux distribution, is used for Domain 0 and Domain Guest, and Domain NV is integrated into Domain 0. As a consequence, vulnerabilities at the operating system level are currently present in both domains. We expect, however, that this is only a limitation of the early prototype. For future work, we envision two potential options. The first variant uses Xen as basic hypervisor, but—instead of Linux—uses a minimalist operating system in Domain NV. The second variant is based on the L4ka microkernel [10]. This microkernel offers virtualisation functionality. In addition, significant efforts towards a formal verification of the L4 kernel have been made by other researchers [18]. These results provide an excellent basis for a trusted entity.

A core task of the Domain NV is the provision of mechanisms for the instantiation and initialisation of service replicas. A simple description language enables the developer to describe how the privileged domain has to instantiate a local replica in a new guest virtual machine. Currently, a disk image of a Xen virtual machine with a preconfigured operating system and middleware environment is provided.

After initialisation, as well as after recovery, a state transfer from the set of replicas is required. The actual transfer is described below; a prerequisite for the transfer is the serialisation of the replica state into a byte stream. We use an application-controlled serialisation mechanism for this purpose. This means that the Domain NV can request the service in the Domain Guest to serialise its state.

4.2 Consistency Management

The replication of a CORBA services requires custom mechanisms within the Domain NV. The first reason for this is the addressing mechanisms of CORBA. In CORBA, the remote address of a service is specified as an *interoperable object reference (IOR)*. Internally, the IOR contains the TCP/IP address of the object request broker (ORB). In a VM-FIT environment, each replica will create its own IOR, using the local network address. These addresses are private IP addresses only used for communication between the virtual machines of a single physical host, and cannot be used by clients. Therefore, the Domain NV of our prototype is able to construct a new service IOR for the replicated service, and publishes this address to a public naming service.

The envisioned hypervisor-based replication architecture can be applied to various kinds of network-based distributed services. For validating the architecture, it is assumed that the service is implemented as a deterministic CORBA object. Thus, the initialisation of a service object includes the three levels operating system, middleware, and replica implementation. Using CORBA objects allow a comparison between replication support at the middleware level and at the hypervisor level.

As a basis for realising group communication on Domain NV, we currently use the AGC group communication system [14], which internally uses the Paxos algorithm for message ordering.

4.3 State Transfer and Proactive Recovery

For proactive recovery, every recovery operation requires a consistent checkpoint of a majority of replicas. This checkpoint has to be transferred to the recovering replica and verified by the majority of nodes (e.g., by checksums). Finally, the recovering replica has to be reinitialised by the provided state. In our prototype, we assume that a secure code basis for the replica is available locally, and only the data state is required to initialise the replica.

The checkpointing and state transfer are time-consuming operations. Furthermore, their duration depends on the state size. During the checkpoint operation, a service is not accessible by clients; otherwise, concurrent state-modifying operations might cause an inconsistent checkpoint. Consequently, there is a trade-off between service availability and safety gained by proactive recovery given by the recovery frequency of replicas. To reduce the unavailability of a service, while still providing the benefits offered by proactive recovery, more than one replica could be recovered at a time. However, in previous systems with dedicated hardware for triggering recovery, the number of replicas recovering in parallel is limited by the fault assumption, as every recovering replica reduces the number of available nodes in a group and, consequently, the number of tolerable faults.

The VM-FIT architecture is able to offer a parallel recovery of all replicas at a time by doing all three steps necessary for proactive recovery in parallel. If service replicas have to be recovered, every node receives a checkpoint message and determines the replica state. The Domain NV receives this state and prepares a *shadow replica domain*. This domain will later be used replace the existing local replica instance and is initialised by the state transfer operation. Thereby, the state is provided as a stream and checksums on the stream data are generate for a configurable block size. These checksums are distributed to all other nodes hosting replicas of the service. Before a certain block is used for the initialisation of the shadow replica, it has to be verified by the majority of all state-providing replicas via the checksums. If a block turns out to be faulty, it is requested from one of the nodes of the majority. After the state transfer, every replica has a fully initialised shadow replica that is a member of the replication group. In a final step, the old replicas can be safely shutdown as the shadow replicas already substitute them.

This approach reduces the downtime due to checkpointing to one checkpoint every recovery period. Furthermore, the amount of transferred data over the network is reduced as only faulty blocks have to be requested from other nodes. Finally, the state transfer is sped up in the average case as only checksums have to be transferred.

4.4 Future Work

The current prototype enables an early evaluation of the core VM-FIT architecture. In future work, we plan to provide a detailed qualitative and quantitative evaluation of the system properties. For an experimental evaluation of the performance of the system, fault injection allows to simulate the effect of non-benign faults in the system to some extent. We plan to use [9] as a platform for executing such experiments. The results will not only provide an assessment of VM-FIT, but also will also permit a comparison with other replication infrastructures, such as a fault-tolerant CORBA implementation.

Another issue of future work is the applicability of VM-FIT to other existing distributed services. The CORBA replication prototype has shown that, for existing services,

dedicated support for these infrastructures has to be integrated into the Domain NV. We will discuss what extensions will be required for other, non-CORBA applications.

An even more important issue is the trust into the Domain NV. Currently, no real mechanisms are used to substantiate this trust. In the future, it is essential to replace the complex domain NV (replication controller application on top of a full Linux instance) with a leaner alternative. In the extreme case, a complete formal verification of Domain NV might be desirable.

Our prototype current implements group communication in the privileged Domain NV. In order to minimise the amount of code in the privileged domain, the advantages and disadvantages of handling group communication here or at the application layer need to be discussed. The existing AGC architecture could be used for a hybrid approach, which provides some basic services (such as distributed consensus) in the privileged part, which acts as a trusted base, and the remaining part of the group communication protocol is handled in the guest operating systems. Such a hybrid architecture is highly promising and feasible to implement within a short time on the basis of the existing system.

5. CONCLUSIONS

In this paper, we have investigated the use of virtualisation technology for the construction of fault and intrusion tolerant systems. The VM-FIT architecture provides basic support for transparent intrusion-tolerant replication of services and for proactive recovery of replicas. In this architecture, virtualisation is used to provide a trusted component on each machine. This enables the use of more efficient protocols. Furthermore, the hypervisor can be used for supporting proactive recovery of service instances. Our current prototype enables a detailed investigation of some core issues of the VM-FIT architecture. Future work will provide a more detailed quantitative study of dependability and efficiency.

6. ACKNOWLEDGEMENTS

This work has been supported by the DAAD.

7. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proc. of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[2] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

[3] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *Intl. Conf. on Dependable Systems and Networks*, pages 167–176, 2002.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI '99: Proc. of the third Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[5] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, USA, Oct. 2000.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[7] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[8] R. P. Goldberg. Architecture of virtual machines. In *Proc. of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM Press.

[9] H. Höxer, M. Waitz, and V. Sieh. Advanced virtualization techniques for FAUmachine. In R. Spenneberg, editor, *11th International Linux System Technology Conference, Erlangen, Germany, September 7-10, 2004*, pages 1–12, 2004.

[10] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[11] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[12] O. M. G. (OMG). Common object request broker architecture: Core specification, version 3.0.2. OMG document formal/02-12-02, 2002.

[13] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *PODC '91: Proc. of the tenth annual ACM symposium on Principles of distributed computing*, pages 51–59, New York, NY, USA, 1991. ACM Press.

[14] H. P. Reiser, U. Bartlang, and F. J. Hauck. A reconfigurable system architecture for consensus-based group communication. In *Proc. of the 17th IASTED Int. Conf. on Parallel and Distributed Computing and Systems*, pages 680–686. IASTED, 2005.

[15] H. P. Reiser, F. J. Hauck, R. Kapitza, and W. Schrder-Preikschat. Hypervisor-based redundant execution on a single physical host. In *Proc. of the 6th European Dependable Computing Conf., Supplemental Volume - EDCC'06 (Oct 18-20, 2006, Coimbra, Portugal)*, pages 67–68, 2006.

[16] P. Sousa, N. F. Neves, P. Verissimo, and W. H. Sanders. Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available. In *SRDS '06: Proc. of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 71–82, Washington, DC, USA, 2006. IEEE Computer Society.

[17] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001.

[18] H. Tuch, G. Klein, and G. Heiser. Os verification — now! In M. Seltzer, editor, *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

An Intrusion-tolerant e-Voting Client System

André Zúquete
IEETA / UA
Campus Univ. de Santiago
Aveiro, Portugal
avz@det.ua.pt

Carlos Costa
IEETA / UA
Campus Univ. de Santiago
Aveiro, Portugal
ccosta@det.ua.pt

Miguel Romão
UA
Campus Univ. de Santiago
Aveiro, Portugal
a28244@alunos.det.ua.pt

ABSTRACT

The ambition of any e-voting system is to reproduce, in an electronic environment, the characteristics of physical voting systems, such as accuracy, democracy, privacy and verifiability. REVS is an Internet e-voting system based on blind signatures and designed to be robust in distributed and faulty environments. However, the execution of REVS client system, used by voters, can be tampered by intruders willing to compromise the accuracy of submitted votes or the privacy of voters. In this document we present a new, intrusion tolerant e-voting client architecture for REVS. This architecture is based on public key cryptography, smart cards and FINREAD terminal readers.

1. INTRODUCTION

Internet voting systems are appealing for several reasons, one of them being the mobility of voters. Paper-based voting systems usually require voters to go to a specific voting poll. But using the Internet, voters may contact the right electoral servers virtually from anywhere in the world. However, the hosts used by voters to express their will must be trusted. Namely, they should not steel authentication credentials of voters nor interfere destructively with the voting process.

Trusted voting clients are difficult to implement in most hosts running general-purpose operating systems, such as Windows or Linux. The complexity of these systems and the degree of freedom in their configuration makes it nearly impossible to assure the correct behavior of a local voting client application. Therefore, critical parts of client voting applications should be deployed in restricted computing environments, capable of providing a proper Trusted Computing Base (TCB).

In this paper we describe an implementation of a voting client using a TCB composed by a FINREAD terminal and a smart card. The FINREAD terminal provides protection against disclosure for user input – authentication input, voter's choices – and output – presentation of ballots and voter's choices. The smart card provides protection for voter's authentication credentials – asymmetric key pair and voter's signatures – and the authentication of the received electoral data – ballot and information about electoral servers, all signed by an electoral authority.

By using this TCB we hope to be able to build an intrusion-tolerant, client voting system formed by the TCB and an ordinary personal computer (PC). The PC makes the required bridge between the TCB and the Internet. A compromised PC should, at most, interfere with the voting protocol using Denial of Service attacks.

For implementing and testing our voting client using this TCB we used REVS (Robust Electronic Voting System [10]). REVS is a publicly available Internet voting system where client voting applications are Java applets that run on voter's hosts. While our main goal was to change only the client application, in order to use the TCB when necessary, we had also to change other components of the REVS system – part of the voting protocol and some electoral servers – to introduce the authentication of voters with asymmetric key pairs.

This paper is organized as follows. Section 2 briefly describes the architecture of REVS, its voting protocol and the weaknesses of the client system. Section 3 describes the new REVS TCB-based client architecture. Section 4 describes some implementation details, regarding both modifications in REVS and the implementation of the TCB. Finally, Section 5 concludes the paper.

2. REVS VOTING PROTOCOL

REVS is a blind-signature based voting system designed for providing secure and robust electronic voting using the Internet [10]. The REVS architecture, depicted in Fig. 1, includes a client application (Voter Module), an electoral Commissioner, and a set of replicated electoral servers – Ballot Distributors, Administrators, Anonymizers and Counters. All these servers can be arbitrarily replicated for improving load balance, availability or for preventing collusion-based frauds.

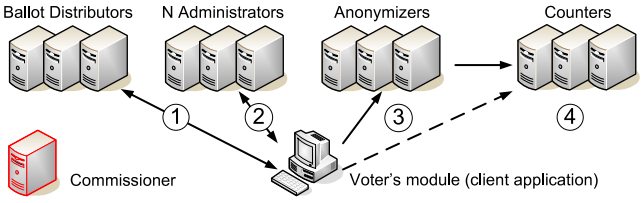


Figure 1: Architecture of REVS

WRAITS 2007 Lisboa, Portugal

Fig. 1 also presents REVS protocol steps, which are:

1 – Ballot distribution. The Voter Module contacts a Ballot Distributor to get a ballot for a given election. The Ballot Distributor returns him the requested ballot, the election's public key and the election's operational configuration (e.g. servers' keys and locations); all this data was previously signed by the Commissioner. Ballots are XML documents providing a set of rules for presenting and verifying voting options for voters.

2 – Vote signing. After getting the voter's will for the ballot, the Voter Module commits to the vote with a random bit string (bit commitment) and generates a digest of the committed vote. Then the Voter's Module generates random blinding factors, applies them to the digest and sends the results to a subset of the N Administrators for signing. Messages sent to Administrators are authenticated by voters using a username/password pair, in order to ensure that only authorized voters participate in the election. An Administrator, after receiving an authenticated, signing request, verifies if it had already signed a blind digest for the requesting voter. If not, it signs the vote and saves that signature; if it had signed before, it returns the previously saved data. After receiving the signature, the Voter's Module removes the blinding and verifies its correctness using the Administrator's public key. This process is repeated until a required number of t signatures is collected. The value of t must be higher than $N/2$, to prevent voters from getting more than one valid vote.

Each voter uses a different password for each of the N Administrators. This is automatically managed by the Voter's Module: the voter introduces only a PIN and a password and the Voter Module uses a cyclic, digest-based password generation process to generate the password for each Administrator. To prevent eavesdropping, the passwords are sent directly to Administrators using SSL secure channels with server-side authentication.

3 – Vote submission. The Voter's Module constructs the vote submission package, joining the vote, its signatures and the bit commitment and encrypting everything with the election's public key. Then he submits this package to any number of the Counters through the Anonymizers, concluding the voting protocol.

4 – Vote tallying. At the end of election Counters merge and publish all submitted packages. The Commissioner discloses the election private key and anyone is able perform the tally: decrypt the submitted packages, discard replicated votes (the ones with the same bit commitment), validate signatures on remaining votes and perform the tally with the valid votes. Missing packages can be detected and resubmitted anonymously by voters.

2.1 Weaknesses in the Voter Module

For portability, the Voter Module is an ordinary Java application. The Voter System, formed by this application and the system where it runs – Java Virtual Machine (JVM) and host system – may be tampered in many ways in order to interfere with the privacy of local voters or with the accuracy of an election. Possible malicious actions taken by the Voter System are the following:

- Provide the voter with false ballots or sending false votes instead of the ones filled by the voter (accuracy issue).
- Steel voter's credentials – identification and authentication passwords (impersonation issue).
- Use otherwise the identification of the voter and his votes (anonymity issue).

To improve the privacy of the voter and accuracy of its participation in the election, the Voter System should use a TCB for:

- Protecting the credentials – the public key of the Commissioner – used to validate the data received from Ballot Distributors.
- Protecting the input of voter's choices for ballots.
- Protecting the voter identity and authentication credentials from steeling.
- Protecting all sensitive information related with the voting protocol – bit commitment and blinding factors.

This approach requires moving part of the Voter Module into a TCB, in order to protect critical validation data (e.g., the Commissioner's public key), voter's critical input/output operations and all permanent or temporary, personal data involved in the voting process.

3. NEW TCB-BASED VOTING CLIENT SYSTEM

The present contribution was developed to improve the REVS e-voting system, namely its Voter System. As we saw before, this component raises some critical accuracy, impersonation and privacy issues when relying on ordinary, insecure hosts. Therefore, it can become the Achilles heel of the whole REVS system.

Our main goal was to develop an intrusion-tolerant voting client system by migrating part of the REVS Voter Module into a TCB, providing the required protection of private data and preventing vote tampering. For implementing the TCB we chose a trustable smart card/smart-terminal environment, namely a smart card and a FINREAD device. These two components are described in more detail in the two following sections.

Voting clients using smart cards are not new, though not frequent [3, 4, 9]. They use smart cards mainly for making computations with voters' private keys in a personal, secure computing environment. We go one step further, by using the smart card for storing other sensitive data, such as the Commissioner's public key and cryptographic values used for creating the vote. Furthermore, we protect the interaction between the voter and the smart card by means of a secure I/O terminal with enough human-interface capabilities – a FINREAD reader. To the best of our knowledge, this is the first time a FINREAD reader is used for supporting secure voting clients.

3.1 Smart Cards

Smart cards provide a properly protected support for storing sensitive, private information, such as personal details or cryptographic keys [11]. There are various ways to use this technology [7], but when correctly combined with other security technologies, such as public key cryptography (PKC) and biometrics, it strongly enforces effective access control through personal identification and/or authentication [8].

There are various types of smart cards, but the most interesting in terms of security for implementing our TCB are those that have an embedded microprocessor capable of executing strong cryptographic algorithms on the card itself, thus not requiring protected information to be moved from the card. The use of those storage and processing devices, with native cryptographic capabilities and protected by user-provided secrets, can improve the level of security to the whole system. Smart cards are an ideal solution for PKC authentication: the private key lies in a secure, tamper resistance storage, a second factor authentication (a PIN) must be introduced to unlock it and a crypto accelerator provides cryptographic hardware operations, such as asymmetric key pair generation and digital signature generation/verification. We have been working with two distinct types of smart cards proving these functionalities: Schlumberger (Axalto) Cryptoflex (16 kB) and Javacard Cyberflex egate (32 kB).

PKC-based authentication systems are more secure than password-based systems because there is no shared knowledge of secrets between transaction intervenients. The private key, used to compute an authentication signature, needs only to be known by one side of transaction – the one being authenticated. In a typical PKC authentication process, the entity being authenticated signs some data with its own private key and the authenticator validates the data and the signature using the corresponding public key.

Considering REVS, a voter proves his authenticity to Administrators by signing requests with a PIN protected, private key stored inside a smart card. Administrators verify the authenticity of a voter by checking his signatures with his public key stored in their databases. This public key must be fetched from the voter's smart card during his registration phase.

By providing a voter with a PKC digital credential, stored and protected by a smart card, we can enforce strong voter authentication and protected, authentication signatures performed inside the smart card. However, smart cards cannot evaluate the correctness of data provided for signing. Therefore, they cannot prevent the adulteration of votes by a tampered Voter System running on a PC environment (see Fig. 2); this issue is handled by using a trusted card reader.

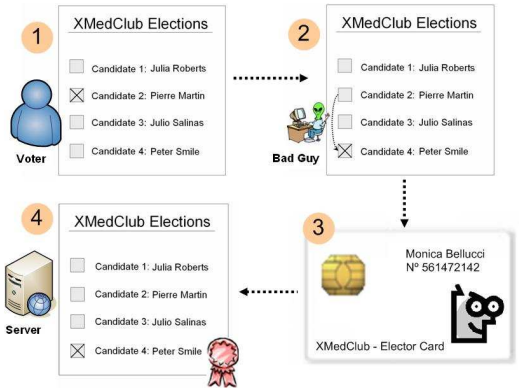


Figure 2: Vote adulteration by the Voter Module: (1) the voter fills the ballot; (2) the Voter Module tampers the data sent for smart card signing; (3) the smart card signs the provided data; and (4) a tampered vote is submitted

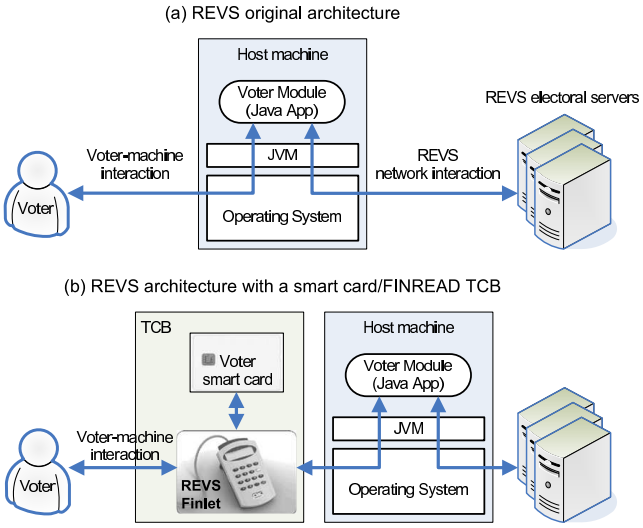


Figure 3: Evolution in the architecture of REVS for support a smart card/FINREAD TCB in the voter side

3.2 OMINEY CardMan Trust FINREAD

As previously explained, the usage of a smart card does not prevent the possibility of vote adulteration in the client PC environment. For instance, REVS Java-based client module could be cracked or maliciously cloned without being detected by the voter. Since steps 1 and 2 of Fig. 2 are actually executed in the PC environment, this represents a potential risk to the voting process.

After analyzing some possibilities to enforce a secure human-machine I/O in the client voting environment, we decided to separate this component from the Voter Module running on the PC, using instead a secure I/O device. The evaluation of several tamperproof devices conducted us to elect the FINREAD smart card terminal as a promising option.

Therefore, our TCB was built around a Java-based smart card reader, namely the Omnikey CardMan Trust FINREAD [12]. This is an ISO 7816 and EMV 3.1.1 compliant smart card reader [1], which also adopts the latest FINREAD specifications [5, 2].

The FINREAD platform adopts and extends the Java applet technology; in a FINREAD environment applets are called Finlets. FINREAD terminals are provided with internal memory (1MB) capable of hosting different software modules (Finlets) and a JVM to execute them. When a Finlet is activated in the terminal, it starts operating in secure mode, i.e. the access to the smart card is always mediated by a Finlet [12]. Otherwise, it operates in transparent mode, acting as an ordinary smart card reader.

Our FINREAD device has reduced but enough human-machine I/O capabilities. It has a small LCD display, containing 4 lines of 20 characters, and a 16 keys pin-pad for user input interaction. Finlets running on the terminal can control both sensitive transactions with the smart card and interactions with the user (voter) using the terminal's display and pin-pad. Namely, the vote options can be displayed by the terminal, the voter choices are introduced through the pin-pad, as well as the smart card PIN.

TCB	Functionalities	
Smart card	Storage	Commissioner public key Voter's authentication asymmetric key pair Per-vote temporary data (vote, bit commitment and blinding factors)
	Computing	Generation of random data (bit commitment and blinding factors) Generation of voter's authentication signatures Digest computations
FINREAD terminal	I/O	Vote presentation and input Smart card PIN input
	Storage	Per-election public keys (Administrators, election) Per-election Administrators' signatures
	Computing	Validation of data provided by Ballot Distributors (signed by the Commissioner) Blinding and unblinding of data exchanged with Administrators Validation of Administrator's blind signatures Production of the vote submission package

Table 1: Functionality provided by the smart card and FINREAD terminal

3.3 TCB Functionality

We considered that this set of components, a FINREAD terminal and a smart card, was suitable for implementing a TCB assuring the required level of security for the new REVS Voter System. The new architecture of REVS using this TCB is presented in Fig. 3. The TCB components provide for REVS the set of functionalities presented in Table 1.

The smart card is used to store sensitive data, private data or critical data for resuming the voting process latter if a fault occurs. The Commissioner's public key is sensitive data, as it is used to validate all the data received by the voter from electoral servers. The voter's asymmetric key pair, his vote, the vote's bit commitment and all blinding factors used in blind signature procedures are both private and critical. Note that this critical data must be stored in non-volatile memory, such as a smart card, to resume the voting process latter if a fault occurs. Storing this data in the smart card also allows the voter to use another FINREAD terminal to complete his voting process.

On the contrary, the FINREAD only stores temporary data that is required to perform local computations – the public keys of electoral servers and valid signatures provided by Administrators. This data can always be recovered after a fault during the voting process.

3.4 Changes in REVS protocol

The REVS voting protocol was only modified to support the PKC-based authentication of voters. In the original protocol each request sent to an Administrator was formed by an election ID, a voter ID and password and a blinded hash of the vote and bit commitment:

$$\text{old request} = \begin{array}{l}\text{election ID, voter ID, password,}\\ \textit{blinded}\,(\textit{hash}\,(\text{vote, bit commitment}))\end{array}$$

The password for each Administrator was presented by the voter in the registration phase and distributed by the Commissioner to the proper Administrator.

In the new protocol, a voter gets authenticated using an asymmetric key pair produced by his smart card. This key pair can be produced during the registration phase, if not already present in the card, and the Commissioner fetches the voter public key from the card and sends it to all Administrators. During the voting process, each request sent to an Administrator is now formed by an election ID, a voter ID, a blind hash of the vote and bit commitment and a voter signature of all this data. For preventing eavesdropping by a malicious Voter Module, both election ID and voter ID padded with a random value and encrypted with the Administrator's public key:

$$\text{new request} = \begin{array}{l}\{\text{election ID, voter ID, random}\}_{Adm\,K_{pub}},\\ \textit{blinded}\,(\textit{hash}\,(\text{vote, bit commitment})),\\ \text{voter signature}\end{array}$$

3.5 FINREAD human interface issues

Because the REVS platform is fully implemented in Java, at the beginning it may appear that it would be relatively easy to migrate selected parts of the original Voter Module into one or more FINREAD Finlets. However, this migration is more complex due to natural limitations of the FINREAD terminal. Other limitations, presented by the FINREAD JVM, will be presented in Section 4.1.

As we have already highlighted, the FINREAD device not only allowed us to prevent the PIN capture outside the smart card reader, but also to display the ballot and retrieve the voter choices. Like all blind signature based voting systems [6], REVS is ballot independent. Ballots are XML documents, which can be easily transferred to the terminal and parsed inside it for proper presentation and fulfillment.

Due to the FINREAD display limitations, the presentation of the ballot in this device must be completely changed. Furthermore, the ballots' text to present to voters must be produced differently, namely using reduced amounts of data and tacking into consideration the reduced capabilities of a FINREAD display. A complementary approach is to produce in the FINREAD terminal, from the same ballot, a rich, graphical image for presenting in the PC display. This image could facilitate the understanding of the ballot options by the voter. But for the privacy of the voter, it must never present the voter choices and the graphical image must not be easily tampered by the PC, something that may be difficult to accomplish.

4. IMPLEMENTATION

The implementation of this TCB in REVS was done in two phases. In the first phase we changed the protocol to use PKC-based authentication of voters. Namely, we (i) changed the Commissioner to register voters' public keys fetched from smart cards and to store its own public key in the smart cards, (ii) changed the Administrators to use voters' public keys for authentication and (iii) changed the

Voter Module to use a smart card to sign requests sent to Administrators. During this phase we have used ordinary smart card readers, or the FINREAD reader in transparent mode [12], and the IAIK Java Cryptographic Extension [13].

In a second phase we developed a Finlet for implementing parts of the Voter Module within the FINREAD terminal. Currently the Finlet functionality includes 3 components.

The first component receives the data retrieved from a Ballot Distributor and validates it with the Commissioner public key stored in the smart card. The vote is interpreted, presented and filled using the FINREAD display and pin-pad. A random bit commitment and several random blinding factors are generated using the smart card random generator. At this stage the vote and all the previous random values are stored in the card for fault tolerance. The smart card PIN is read using the FINREAD pin-pad and the requests for the all the Administrators are produced and signed using the smart card and the voter private key. At the end, this component returns to the Voter Module all the requests that must be sent to Administrators.

The second component receives a signature from an Administrator and validates it with the Administrator's public key received by the first component and stored in the FINREAD memory. If the signature is correct it is stored in the FINREAD memory for latter use.

The third component packs the vote, the bit commitment and the required number of valid signatures collected from the Administrators and builds the submission package. This package is then returned to the Voter Module for being submitted to Counters through Anonymizers.

The Voter Module uses the first component after getting data from a Ballot Distributor, uses the second component after getting each signature from an Administrator and uses the third component after getting $t > N/2$ valid signatures from Administrators.

Along all this process, the Voter Module that runs in the PC has no direct influence in the presentation and ballot filling out, no access to the voter's authentication credentials – smart card PIN and private key – and no direct access to the vote – the vote submission package can only be open at the end of the election. Furthermore, the identification of voters sent to Administrators, when getting their authorization signature, is encrypted by the terminal with the Administrators' public keys.

4.1 Implementation issues

The FINREAD execution environment is quite unique. Finlets are event-driven applications but there is no event loop; methods with standard names are called for handling events. There are no debugging facilities other than sending messages to the FINREAD display or to the PC.

The FINREAD JVM misses some useful functionalities that could facilitate our work: there is no support for XML parsing and for serialization. The lack of XML parsing support forced us to implement a small, simple parser within the Finlet for handling ballots. The lack of serialization support complicated the interface between Voter Module and Finlet and forced us to build differently vote submission packages and to deploy a new interface in Counters.

5. CONCLUSIONS

There are a great number of sensitive services available on the Internet and this number is expected to continue to grow in the next few years. One of these services is electronic voting, that should accomplish the desired characteristics of traditional voting systems, such as accuracy, democracy, privacy and verifiability.

Our working platform was REVS, a robust electronic voting system designed for distributed and faulty environments, namely the Internet. The Voter Module of REVS was designed for trusted hosts, thus raising privacy and accuracy concerns when considering its widespread use in any host. To handle this problem we designed a TCB, using a smart card and a card reader with human-machine interface capabilities, to enforce the trusted execution of critical parts of the Voter Module in untrusted or compromised hosts.

In this paper we described a new architecture for the REVS Voter Module using this TCB. The result is a voting client system that is intrusion tolerant. We also modified the authentication process of voters in REVS – from username/password pair into asymmetric key pairs – to benefit from the use of PKC-based authentication provided by the smart card. The final system provides a robust PKC-based authentication of voters and protects all critical actions and data of voters, during the voting process, with tamper-proof devices – smart card and FINREAD reader.

6. REFERENCES

[1] Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 4 : Interindustry commands for interchange. ISO/IEC 7816-4, 1995.

[2] OMNIKEY FINREAD SDK Manual, v1.22.3, 2005.

[3] C.-B. Breunesse, B. Jacobs, and M. Oostdijk. Voting using Java Card smart cards: A case study, Jan. 2002.

[4] S. Canard and H. Siberty. How to fit cryptographic e-voting into smart cards. In *Frontiers in Electronic Elections (FEE 2006)*, Hamburg, Germany, Sept. 2006.

[5] F. Consortium. FINREAD Technical Specifications, Parts 1-8, 2003.

[6] A. Fujioka, T. Okamoto, and K. Ohta. A Practical Secret Voting Scheme for Large Scale Elections. In *Adv. in Cryptology – AUSCRYPT '92 Proc. (LNCS 718)*, Queensland, Australia, 1992. Springer-Verlag.

[7] H. Gobioff, S. Smith, J. D. Tygar, and B. Yee. Smart Cards in Hostile Environments. In *2nd USENIX Works. on Electronic Commerce*, Oakland, USA, 1996.

[8] G. Hachez, F. Koeune, and J. Quisquater. Biometrics, Access Control, Smart Cards: a Not So Simple Combination. *Security Focus Magazine*, Oct. 2001.

[9] J.-K. Jan and C.-C. Tai. A Secure Electronic Voting Protocol with IC Cards. *Journal of Systems and Software*, 39(2), Nov. 1997.

[10] R. Joaquim, A. Zúquete, and P. Ferreira. REVS – A Robust Electronic Voting System. *IADIS Int. Journal of WWW/Internet*, 1(2), Dec. 2003.

[11] R. Marvie, M.-C. Pellegrini, O. Potonniée, and S. Jean. Value-added Services: How to Benefit from Smart Cards. In *Gemplus Developer Conf. (GDC 2000)*, Montpellier, France, June 2000.

[12] K. Schmid and H. Zeitlhofer. FINREAD Whitepaper, Rev 1.0, 2003.

[13] I. SIC. The IAIK Provider for the Java Cryptography Extension (IAIK-JCE), 2001.

Experiments on COTS Diversity as an Intrusion Detection and Tolerance Mechanism

Frédéric Majorczyk, Éric Totel, Ludovic Mé
firstname.lastname@supelec.fr

ABSTRACT

COTS (Components-Off-The-Shelf) diversity has been proposed by many recent projects to ensure intrusion detection and tolerance. However using COTS in a N-version architecture presents some drawbacks, especially in intrusion detection, which have consequences on intrusion tolerance. COTS Diversity is prone to raise many false positives (false alerts). In this article, we explain what a COTS Diversity architecture can detect and propose a masking mechanism to reduce the false positive rate. We apply this method to web servers and provide some experimental results that confirm the necessity of this mechanism.

Keywords

Intrusion tolerance, design diversity, COTS diversity, intrusion detection

1. INTRODUCTION

Design Diversity, and more especially N-version programming, are techniques used to detect and tolerate faults. They have been studied actively [1, 9] and have been used in many industrial projects. N-version programming consists in the execution of a single function by two or more elements, called versions, and the comparison of the results of the different versions to make a decision on the result. The underlying hypothesis is that the different versions used are independent from the point of view of their faults. N-version programming has been proved to provide a high coverage of fault detection [10], although some common-mode failures may still exist [8].

Recently several projects have explored the idea of using COTS (Components-Off-The-Shelf) instead of specifically developed software both in the dependability [5] and in the security field [14, 4, 3, 6, 16, 15, 2]. Indeed, developing specific software several times is very costly, while many Internet services are already implemented by COTS. In the security field, three projects, DIT [15], HACQIT [6], and BASE [2], focus on intrusion tolerance while the others [14, 4, 3] focus on intrusion detection. The intrusion tolerance property relies heavily on intrusion detection. While false negatives (missing of intrusions) have a direct impact on the intrusion tolerance, false positives (false alarms) can decrease the performance of the architecture and lead in some cases to a self denial of service (DoS) ; the availability of the system is then not ensured. A binary comparison of the outputs of the COTS can lead to a very high false positive rate. A very simple comparison algorithm can lead to many false negatives and may yield some false positives. Depending on the choice of the algorithm, it is necessary to introduce some mechanisms in the architecture to counterweight its drawbacks. We suggest here to implement a masking mechanism to reduce the false positive rate.

In this paper, we study, through the example of web servers, the false positive and false negative rates in a COTS diversity based architecture, their potential influence on the intrusion tolerance property and the effectiveness of a masking mechanism. In Section 2, we briefly present the DIT, HACQIT, and BASE projects which use COTS diversity for intrusion tolerance. Then, in Section 3, we analyse the type of differences detected by COTS diversity, and show the necessity for a masking mechanism avoiding false positives to be generated by the detection algorithm. Finally, in Section 4, we present some results with relation to intrusion detection.

2. RELATED WORK

Three recent projects have brought design diversity to the security field in order to detect and tolerate intrusions. We present here these three projects: DIT, HACQIT, and BASE.

2.1 The DIT Project

DIT (Dependable Intrusion Tolerance) [15, 16] is a project that proposes a general architecture for intrusion tolerant enterprise systems and the implementation of an intrusion-tolerant web server as a specific instance. The architecture includes functionally redundant COTS servers running on diverse operating systems and platforms, hardened intrusion-tolerant proxies that mediate client requests and verify the behaviour of servers and other proxies, and monitoring and alert management components based on the EMERALD intrusion-detection framework [11]. The architecture was next extended to consider the dynamic content issue and the problems related to on-line updating [13]. Intrusion detection relies mostly on host monitors and network intrusion detection systems, but is also enforced through the comparison of md5 hashes of the servers outputs. Once a COTS server is considered as compromised, it is reconfigured from a backup and can reinserted again in the architecture.

2.2 The HACQIT Project

HACQIT (Hierarchical Adaptive Control for QoS Intrusion Tolerance) [6] is a project that aims to provide intrusion tolerance for web servers. The architecture is composed by two COTS web servers: an IIS server running on Windows and an Apache server running on Linux. One of the servers is declared as the primary server and the other one as the backup server. Only the primary server is connected to clients. Another computer, the Out-Of-Band (OOB) computer, is in charge of forwarding each client request from the primary to the backup server, and of receiving the responses of each server. Then, it compares the responses given by each server. The comparison is based on the status code of the HTTP response. In addition, host monitors, application monitors, a network intrusion detection system like Snort [12] and an integrity tool (Tripwire [7]) are also used to detect intrusions. A mechanism of rejuvenation is used to restart possibly compromised servers in a safe state. In case of an intrusion, a sandbox mechanism is used to replay requests in order to find the sequence of malicious requests that has lead to the intrusion. These requests can then be rejected by the firewall.

2.3 The BASE Project

BASE [2] proposes to use an abstract model of the protected service to allows to use COTS and thus to reduce the cost of Byzantine fault tolerance, and to improve its ability to mask software errors. A diversified NFS service is built as an example. The abstract model, which describe the normal functioning of the diversified service, is used to normalize the outputs of the diversified implementations, and thus to mask the output differences due to specification differences or nondeterminism. This approach can be used if the service to diversify is well documented, in order to allow the definition of the abstract model. If the implementations used to build the service behave very differently, the outputs of one or several implementations will deviate significantly from what is modeled in the abstract model.

2.4 Discussion

In these two projects, the intrusion tolerance property relies to a great extend on the detection mechanism. If an intrusion occurs and is not detected, the compromised server must be considered as a byzantine process. Then it can skew all the next results of the comparison algorithm and so the intrusion tolerance property is not ensured anymore. It is then possible to compromise other servers in the architecture without being detected. There are many ways to circumvent false negatives or their effects: using a very strict comparison algorithm, adding several others IDSes, reconfiguring regularly the servers. The reconfiguration of a server is obviously required when an intrusion is detected but, as the detection mechanism can miss some intrusions, it seems essential to reconfigure the servers periodically. DIT, HACQIT and BASE implement it. DIT and HACQIT add also other IDSes to reduce the probability of a false negative.

Too many false positives can decrease the global performance of the system and even lead to a self DoS. In the HACQIT project, in case of an alert generated by a request, a window of past requests is replayed in a sandbox to establish the requests that are responsible for the alert. These requests are then rejected by the firewall. If the requests considered as malicious are in fact sound, the server will not respond to normal requests and so is partly unavailable. To resolve the problem of false positives, we introduce a masking mechanism, which allows us to use a strict comparison algorithm and thus to be able to detect and tolerate intrusions without additional IDSes. The BASE project masking mechanism avoids some false positives. Nevertheless, this require an *explicit* model of the service, which may be difficult to build, as we propose to use an *implicit* model associated to an *explicit* model limited to the definition of known differences.

3. INTRUSION TOLERANCE BY COTS DIVERSITY

We present first some drawbacks of COTS Diversity with regard to classical N-version programming. Then, we detail a general taxonomy of the differences detected by a COTS diversity based architecture.

3.1 COTS Diversity Drawbacks

Using COTS in an N-version architecture leads to some drawbacks in the detection process. There is no proof that the assumption of independent failure is verified by the COTS used. Consequently, a study of the known vulnerabilities must be performed in order to ensure that this hypothesis is verified. This has been carried out in several studies, such as [5] that shows that there are very few common failures for COTS databases, or the one of [17] that Apache and IIS servers have no common vulnerabilities.

Moreover, the specification of the COTS neither precise what are the data to be compared, nor when it has to be compared. Thus, a choice has to be made about that two points. This choice will have a major impact on the detection. On one hand, a very strict comparison algorithm can generate many false positives but no false negatives. On the other hand, a loose comparison algorithm may generate comparatively few false positives but miss several intrusions.

Finally, the specifications of the COTS used may not be known exactly and some differences may exist though the COTS are supposed to implement the same service. The comparison may then lead to output differences that are not only due to software failures but also to design or specification differences. If not handled correctly, these differences may cause many false positives. We think that this is the main issue with COTS diversity based architecture. Thus we detail in the next Section what kind of differences such an architecture will detect.

3.2 Taxonomy of Detected Differences

In N-version programming, since the different versions have the same specification, an output difference means that a fault has been activated in one of the version. That is not necessarily the case when COTS are used. COTS software have indeed not exactly the same specification. The specification of a COTS with respect to other COTS can be viewed as a common part and a specific part that differs from other variants specific parts.

Thus, the output differences that are detected are the results either of design differences that are due to design faults in the part of the program covered by the common specification, or design differences that are due to differences in the specific parts of the specifications. These later design differences are not necessarily (but can be) design faults;

Design faults can in their turn be divided in two different sets: classical design faults and vulnerabilities. Classical design faults are faults in the system that cannot lead to a violation of the security policy of the system while vulnerabilities are faults that can be exploited to breach the security of the system.

Clearly, without an additional mechanism, the comparison algorithm would detect all these kinds of differences. The output differences detected that are due to classical design faults or specification differences would actually be false positives, because they do not imply any violations of the security policy. These false positives must, of course, be eliminated. This elimination can be performed by masking the legitimate differences. The masking functions are applied to modify the request before it is processed (pre-request masking: proxy masking function) or after the request has been performed (post-request masking: rule masking mechanisms). In both cases, an off-line experimental identification of the specification differences is needed. It is not easy to evaluate theoretically the sets of differences detected and the need of masking rules since it depends on the COTS and the comparison algorithm used. We decided to evaluate them experimentally on a web server implementation, presented in the next Section.

4. APPLICATION TO WEB SERVERS

First, we present briefly the detection algorithm and the output difference masking mechanism. We expose then the results with relation to detection.

4.1 Detection Algorithm

The detection algorithm depends on the application monitored and must be developed specifically for each application considered. Here, we compare the HTTP responses from the web servers, since HTTP is obviously part of the common specification between the COTS.

The detection algorithm is composed of two phases. First, a watchdog timer provides a way to detect that a server is not able to answer to a request. All servers that have not replied are considered to be unavailable, and an alert is raised for each of them. Then, the comparison algorithm is applied on the set of answers that have been collected.

When all server responses are collected, we first try to identify if these answers are known design differences. In this case, we mask the differences by modifying some of the headers. Then, we begin the comparison process itself. As the comparison of the body can consume a lot of time and CPU, the detection algorithm

compares first the status code, then the other headers in a given order (Content-Length, Content-Type, Last-Modified), and eventually the body. If no majority can be found amongst the responses from the servers, the algorithm exits and an alert is raised. It is useless to compare the body and the other headers of the responses if the status code is not of type 2XX (i.e., the request has not been successfully processed). In this case, the response is indeed generated dynamically by the web server, and may differ from one server to the others. (If these bodies were compared, it would generate a large amount of false positives.)

4.2 Output Difference Masking

The recognition of the output differences that are not due to vulnerabilities is driven by the definition of rules. These rules define how such differences can be detected. They currently take into account several parameters, such as: a characteristic of the request (length, pattern matching, etc.), the status code, and the Content-Type headers. For example, a rule can define a relation between the outputs, e.g., between the status code of the several outputs. Another example would be to link a particular input type to its expected outputs.

It is not possible to define all differences using these rules. For example, Windows does not differentiate lower case letters from upper case letters, and thus we had a lot of behaviour differences due to this system specification difference. Thus we added a mechanism in the proxy which processes the requests to standardize them before they are sent to the servers. Thus, all web servers provide the same answers.

The output difference masking mechanism is thus divided in two parts: pre-request masking mechanisms that standardize the inputs and post-request masking mechanisms that mask the differences that are not due to errors in the servers.

4.3 Experimental Results

The objective of the tests that have been conducted is to evaluate the COTS diversity based detection mechanism in terms of both reliability and accuracy of the detection process. The reliability of the approach is its ability to detect correctly all the intrusions. The accuracy refers to its capacity to avoid false positives generation.

It is not easy to evaluate detection reliability for practical reasons (vulnerabilities affect specific versions of the servers in specific configurations). However, our previous work [14] shows that the detection mechanism is valuable since it is able to detect several intrusions launched against our web server implementation.

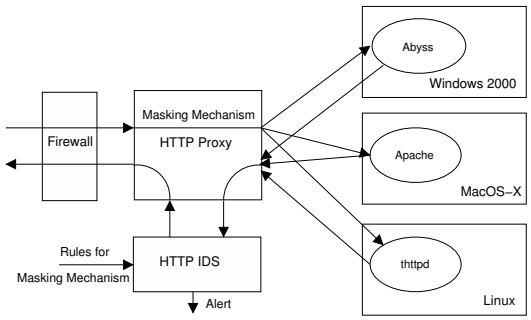


Figure 1: Architecture for the Accuracy Test

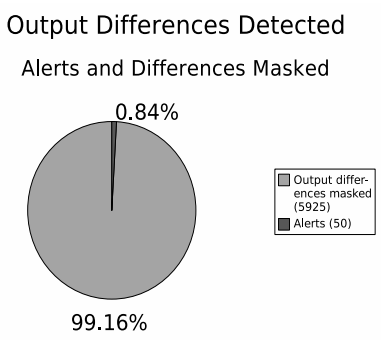


Figure 2: Analysis of the Detected Differences

In order to evaluate the detection accuracy, we set an architecture with three servers shown on Figure 1. We use two sets of HTTP requests. Both contains HTTP requests logged on the website of our campus during a week. The first one is composed of 71,596 requests and is used to write the masking rules. The second one is composed of 105,228 requests and is used to evaluate the detection mechanism. On the test set, 50 alerts are raised, which represents about 7 alerts a day. After analysis, all these alerts are false positives. Without masking mechanisms, 5975 alerts would have been raised or in other words, the comparison algorithm detects 5975 output differences. 99.16% of the output differences are then masked by masking rules.

It must be noticed that, in case of an intrusion or an attack, the localization of the server attacked is often not possible since there is no majority in the responses of the servers. A request can indeed activate a design fault (especially a vulnerability) and induce an output difference due to design differences between all COTS used. If the localization is not possible, it is necessary

to reconfigure all servers.

5. CONCLUSION

As a conclusion, we can state that the COTS Diversity approach provides a high coverage of detection (consequence of COTS diversity and hypothesis of decorrelation of vulnerabilities).

However using COTS in a intrusion tolerant architecture must be done carefully: the choice of the COTS, the comparison algorithm and the masking mechanisms have an impact on the false positive rate. By our experiments, we have shown that a high amount of output differences are actually due to design differences and not to exploits of vulnerabilities. We conclude that a masking mechanism is mandatory for COTS Diversity being effective in intrusion detection and tolerance.

6. REFERENCES

[1] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *IEEE Computer*, 17:67–80, August 1984.

[2] M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3):236–269, 2003.

[3] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, Canada, August 2006.

[4] D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, pages 63–81, Seattle, WA, September 2005.

[5] I. Gashi, P. Popov, V. Stankovic, and L. Strigini. *On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*, volume 3069 of *Lecture Notes in Computer Science*, pages 196–220. Springer-Verlag, 2004.

[6] J. E. Just, J. C. Reynolds, L. A. Clough, M. Danforth, K. N. Levitt, R. Maglich, and J. Rowe. Learning unknown attacks - a start. In A. Wespi, G. Vigna, and L. Deri, editors, *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, volume 2516 of *Lecture Notes in Computer Science*, pages 158–176, Zurich, Switzerland, October 2002. Springer.

[7] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *in Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, Fairfax, VA, November 1994.

[8] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *Software Engineering*, 12(1):96–109, 1986.

[9] M. R. Lyu and A. Avizienis. Assuring design diversity in N-version software: A design paradigm for n-version programming. *Dependable Computing and Fault-Tolerant Systems*, 6:197–218, 1992.

[10] M. R. Lyu and Y.-T. He. Improving the N-version programming process through the evolution of a design paradigm. *Transactions on Reliability*, 42(2):179–189, June 1993.

[11] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proc. of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, MD, October 1997.

[12] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX LISA'99 conference*, pages 229–238, Seattle, WA, November 1999.

[13] A. Saidane, Y. Deswarte, and V. Nicomette. An intrusion tolerant architecture for dynamic content internet servers. In P. Liu and P. Pal, editors, *Proceedings of the 2003 ACM Workshop on Survivable and Self-Regenerative Systems (SSRS-03)*, pages 110–114, Fairfax, VA, October 2003. ACM Press.

[14] E. Totel, F. Majorczyk, and L. Mé. COTS diversity based intrusion detection and application to web servers. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, pages 43–62, Seattle, WA, september 2005.

[15] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saïdi, V. Stravidou, and T. E. Uribe. An adaptive intrusion-tolerant server architecture. In *Proceedings of the 10th International Workshop on Security Protocols*, pages 158–178, Cambridge, United Kingdom, April 2002.

[16] P. E. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Sptringer-Verlag, April 2003.

[17] R. Wang, F. Wang, and G. T. Byrd. Design and implementation of acceptance monitor for building scalable intrusion tolerant system. In *Proceedings of the 10th International Conference on Computer Communications and Networks*, pages 200–5, Phoenix, AZ, October 2001.

The DPASA Survivable JBI—A High-Water Mark in Intrusion-Tolerant Systems[1]

Partha Pal
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
1-617-873-2056
ppal@bbn.com

Franklin Webber
BBN Technologies
410 W Green Street #1
Ithaca, NY 14850
1-607-877-0803
fwebber@bbn.com

Richard Schantz
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
1-617-873-3550
schantz@bbn.com

ABSTRACT
In this paper, we describe the design, development, and validation of an information system that has recently set a new high-water mark for intrusion tolerance. The system, known as the DPASA Survivable JBI, conforms to an abstract architecture for survivable systems and integrates concrete defense mechanisms for preventing intrusion and for detecting and responding to intrusions that cannot be prevented. The system has shown a high level of resistance to sustained attacks by sophisticated adversaries.

Categories and Subject Descriptors
D.4.6 [**Security and Protection**], C.2.4 [**Distributed Systems**], K.6 [**Management of Computing and Information Systems]**

General Terms
Design, Security, Experimentation.

Keywords
Survivability, Intrusion-tolerance, Survivability Architecture, Defense-enabling, Defense Mechanisms

1. INTRODUCTION
Defending an information system against cyber-attacks is an arms race that is inherently asymmetric and favors the adversary. The adversary needs only to find a single exploit, whereas the defenders need to prevent as many of these exploits from succeeding as possible. The adversary tends to find more opportunities to attack as information systems become increasingly interconnected, and new flaws and vulnerabilities are continually being discovered (even as the defenders are addressing or coping with known ones). Information systems that are part of critical infrastructure (e.g., the national power grid, banking systems) or related to national security (e.g., weapons control, missile defense) obviously face the greatest risk.

Over the past decade, a substantial investment went to developing technologies that address specific and individual aspects of the cyber threat. For instance, firewalls focused on efficiently blocking unwanted traffic, digital signatures focused on preventing modification of data in transit, while redundancy and Byzantine protocols focused on surviving corrupt or compromised application components. Evaluation of these technologies was also limited in scope focusing only on the stated goal of the technology. While this showed good technological progress in the individual problem areas, it was unclear how to integrate these "building block" technologies in a well-defended, survivable system that tolerates sustained and sophisticated attacks.

In 2002, DARPA issued a challenge to the research community as part of the OASIS Dem/Val program, which is one of the most significant undertakings in cyber-defense research in recent days. This program sought to demonstrate, by way of developing and experimenting with a survivable DoD-relevant information system that a new high-water mark in intrusion-tolerance and survivability is achievable using the currently available technologies as building blocks. The aim was to survive against sustained and fairly unrestricted attacks from a well prepared class A Red Team for over 12 hours, whereas previous (circa 2001-2002) DARPA Red Team experiments [1] have shown survival time of a defended system only in the order of 20 minutes, with various rules of engagement restricting the attackers options. From late 2002 to early 2005, the DPASA (which stands for Designing Protection and Adaptation into a Survivability Architecture) team, led by BBN, designed a survivability architecture, used it to defense-enable the undefended target system and subjected the resulting system to multiple, fairly unrestricted Red Team exercises.

Overall, the survivable system showed excellent resiliency in thwarting, bypassing or degrading service levels for surviving the attack effects. However, as expected of a high-water mark research prototype, the survivable system needed expert operators to interpret the information it captures and to use the provided response mechanisms effectively. Nevertheless, the survivable information system produced by this program is truly a significant achievement. It demonstrated that it is possible to deploy a tightly configured, highly resilient system under aggressive cost and

Workshop on Recent Advances in Intrusion Tolerant Systems'07, March 23, 2007, City, State, Country.

[1] This research was funded by DARPA under AFRL contract No. F30602-02-C-0134

scheduling constraints. It also broke new grounds in validating the survivability claims of a design and its implementation.

2. THE UNDEFENDED SYSTEM

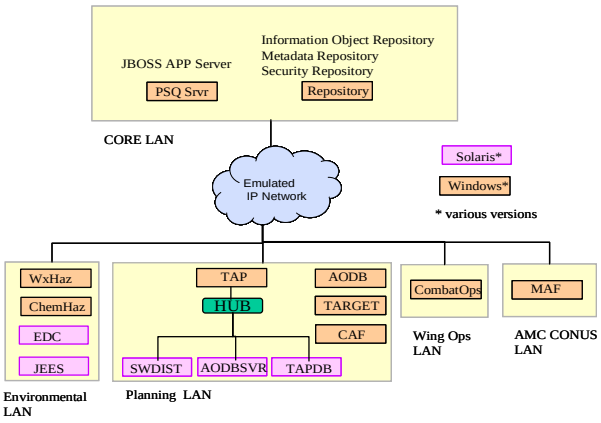


Figure 1: The Undefended JBI

The Joint Battlespace Infosphere (JBI) [2] concept, developed by the US Air Force Research Laboratory (AFRL), seeks to establish effective interaction among disparate military computer systems that must exchange information in support of various network-centric warfare activities ranging from intelligence gathering to mission planning and tactical operations. The JBI aims to achieve this goal by using a publish-subscribe framework that allows diverse computing systems to interact in a decoupled manner as long as they follow a common API for defining Information Objects (IO); and for performing publish, subscribe and query (PSQ) operations facilitated by a set of core services. A JBI instantiation is therefore a set of applications and core services that are needed to execute a specific mission. One such instantiation, simulating the execution of an Air Tasking Order (ATO) and planning of a concurrent airlift through the theater, was used as the demonstration vehicle in the OASIS Dem/Val program. This exemplar JBI integrates applications for selecting appropriate targets, monitoring environmental conditions, and creating ATOs. These applications are organized in 4 Local Area Networks (LANs, which are often referred to as enclaves) namely, the Planning LAN, The Environmental LAN, the Wing Operations LAN, and the AMC CONUS LAN after the Air Force functions they perform. This system is shown in Figure 1. The core services are organized in their own LAN. A successful mission would involve making the go/no-go decision on an ATO that may have WMD sites as targets. The factors influencing the go/no-go decision include presence of WMD sites in the ATO as targets, the predicted weather condition in the target area, presence of friendly force near by, possibility of other air traffic (such as the airlift mission) in the theater, etc.

3. HIGH-LEVEL STRATEGY

The DPASA team realized that the desired level of survivability is not achievable by any individual security tool or technique, or by a straightforward implementation of "defense in depth" — the often cited cyber-defense strategy seeking to use of multiple security techniques to mitigate the risk of one being compromised or circumvented. The following high-level strategy motivated the team to organize elements of protection, detection and adaptive response in *the survivability architecture* of the defended system:

- The architecture must create a very high barrier to entry for an attacker trying to intrude into the system, and also for the attacker who is trying to attack or move to other parts of the system after gaining access or corrupting a toehold within the system.
- The architecture must maximize the likelihood of a sensor being tripped by attacker activity by monitoring all parts of the system at all layers, perhaps in a redundant and overlapping manner.
- The architecture must support, with or without human intervention, dynamic reconfiguration of the system to recover from the damages caused by the attack, or to cope as long as possible with the attack-induced damages that cannot be repaired.

The design of the survivability architecture was further shaped by key design principles derived from decades of experience in building adaptive distributed systems integrated with defensive capabilities. A sampling of these principles appears next.

4. DESIGN PRINCIPLES

SPOF protection: It is practically impossible to eliminate (and in some cases identify) all possible single points of failure in a large and complex distributed system. But, obvious single points of failure in key parts of a system must be protected. In our case, it was clear that the PSQ server in the undefended JBI (that handles publish, subscribe and query requests from JBI clients) is absolutely necessary for continued forward progress of the mission. Therefore, the architecture ensured that no single PSQ server is an SPOF. Since we anticipated adversary trying to corrupt the PSQ service, the PSQ server was replicated 4-fold and organized in 4 *quads* (see Figure 2). With 4 PSQ servers, it was possible to tolerate Byzantine corruption in one of the PSQ servers. Spatial redundancy like server replication is not the only way to protect SPOFs. For example, the clients that issue PSQ service requests are driven by human operators, and were hard to replicate. Therefore, the architecture incorporated mechanisms to restart the clients from check-pointed state when required. Ultimately, the level of effort spent in SPOF protection is a risk-benefit tradeoff. The designer/system owners may decide that the cost of protecting a particular SPOF is too high, and assume the risk of leaving it unprotected.

Physical barriers before key assets: In addition to protecting the SPOFs, parts of the system that are responsible for key functionality and control decisions should have some elements of physical fortification. In our case the PSQ servers and their databases were put in the *operations zone*, which is protected by the *crumple zone*—a layer of proxy hosts that anyone seeking PSQ service must go through. The system management function (implement by the System Manager or SM), which controls many of the defense-mechanisms and adaptive responses, is put in the *executive zone*, which is situated behind the operations zone. The operations zone proxy of the system management function is known as Downstream Controller or DC. This made it very difficult for the intruder, coming from the outside of the system to attack and take control over the key decision-making and control capabilities implemented in the system management components.

Controlled use of diversity: Wholesale use of diversity is operationally expensive and complex. But in the DPASA architecture, we leveraged a small amount of OS diversity in the context of redundant quads and the physical layering provided by the zones in such a way that the attacker has to compromise at least two hosts running different operating systems. The general principle we recommend is that the survivability architecture should include some diversity in each access path to key assets. Apart from OS diversity, other means of introducing diversity include different processor architecture (e.g., SUN, X86), policy configuration, application and service implementations.

Robust basis of defense in depth: Ideally, the multiple obstacles an adversary must face in a survivable system (comprising of multiple protection measures, redundant detection mechanisms with overlapping scope and coverage, and adaptive responses) should be well-grounded, preferably upon some hardware or cryptographic base. To illustrate, note that the primary means of protecting an individual host from intrusion is to control network access to that host. A software-based firewall can do this, but then the defender and the defended would share the same hardware and OS, implying that flaws in the host and OS may affect the firewall, and the host may be attacked before the firewall can stop the attack, which in turn further weakens any other defense that assumes that the host is protected by the firewall. We reduced this risk by using the Autonomic Distributed Firewall (ADF)[3] NICs that can only be controlled by cryptographically authenticated policy servers, and provided a much stronger, separately situated and harder-to-circumvent protection than software based firewalls. Using a smart card to store and perform computation instead of storing the key on disk or using the host CPU to compute with it could provide similar advantages. The NIDS appliances and self-monitoring policy enforcement provided a similar robust basis for detection. The NIDS appliance is a hardened host that collects data, but does not respond. Policy violations, including changes in the policy or policy enforcement mechanisms are reported as detection events. For robust adaptive response we relied mostly on independent corroboration and digital signatures before triggering adaptive response; however the adaptive responses were mostly implemented in software (and some required human intervention).

Containment layers: Once we accept the reality that some attacks will succeed in penetrating the system, and may not be detected until their unwanted effects manifest, it becomes clear that the survivable system needs to *contain* the spread of the attack. A key architectural concept we used in this regard is known as "containment layers". Containment layers must be coordinated at the spatial level as well as the functional level. At the spatial dimension, one can think of concentric layers of increasing scope and span starting from an application process, to the host on which the process runs, to the network segment on which the host resides, and to sets of network segments and eventually spanning the entire system. However, a given functionality may be implemented by multiple processes, possibly in multiple hosts, and typically a single host runs multiple application processes. This adds the functional dimension to the containment layers. In our case, key functionality like the PSQ (i.e., the ability to address Publish, Subscribe or Query requests issued by the JBI clients) and System Management are replicated, and are made available by application level proxies that reside on the crumple zone hosts.

This view of intersecting and overlapping containment layers enabled the designers to develop key sensor and actuator mechanisms and place them in appropriate boundaries. This reinforced the overall defense by providing higher than normal access control, visibility to attacker actions and manageability of systems resources. Access control, and process and application level security policies are used to contain the attack spread from one process to other co-located processes, or from one host to others in the same network segment. Policy enforcement mechanisms, and customized sensors embedded in software applications provide indication of attempted or successful breaches of such containment boundaries. Detection or suspicion of such breaches prompt adaptive response (under defense's control) such as killing or restarting specific processes, rebooting or quarantining individual hosts, or isolating entire network segments. Note that as the container grows bigger, the level of detection and response becomes less specific—if a violation is only detected between network segments, accurate information about the compromised processes, or hosts may not be available.

Range of adaptive responses: It follows from the discussion of containment layers that the survivability architecture will exhibit a range of adaptive responses with varying scope and consequences. Clearly, they should not be treated uniformly—some could be mounted as knee-jerk responses, others should be mounted with deliberation. The designers should consider carefully the kind of decision-making and control mechanism to put between the various detectors and actuators introduced by earlier considerations. In DPASA we followed the simple rule of thumb: actions that have localized impact or can be easily reversed are mounted in a knee-jerk manner, typically driven by an if-then condition or a local table look-up. Examples include restoring a lost or corrupt file, re-sending a publish request, killing a spurious (i.e., that was not part of, or a descendant of the application processes) process on a local host, or blocking a request that is against the predefined security policy. However, killing an application process, or isolating a host or a network segment can adversely affect the system's desired functionality. Thus, such responses need to be coordinated. In DPASA, this coordination had two distinct parts: deciding when to mount such a response, and adapting the rest of the system to deal with the effect of such a response. For the former, our approach was fairly conservative—multiple independent corroborations were required for mounting such actions. For the latter, specialized protocols were developed to propagate or inform the affected components of the system.

Configuration generation from specs: Misconfiguration is often cited as the main source of security vulnerability. Use of redundancy and multiple layers of defense create more opportunity for misconfiguration. Not only policies governing the defense mechanisms must be consistent across different layers of the defense, they must be coordinated across the system among the redundant and interacting components. This risk must not be left unmanaged. In DPASA, we took the position that configurations must be auto-generated as much as possible from a single specification. We were successful in removing one key source of misconfiguration by specifying the IP address and port assignments in a file, and using that to generate all policy and configuration that needed them. Even though some policies needed to be generated manually, sourcing the IP and port assignments from a single place provided significant time and cost savings in terms of testing and debugging that was avoided

5. THE DEFENDED SYSTEM

The shortest way to describe the defended system is to highlight the enhancements resulting from the survivability architecture.

System Level Enhancements: Instead of a single JBI core enclave, the defended system has four core enclaves (quads). Each enclave runs the same functions, but consists of hosts running 3 different operating systems (Security Enhanced Linux, Windows, and Solaris). The crumple, operations, and executive zones are also depicted in Figure 2: the crumple zone has an AP host and the executive zone has an SM host in every quad. The operations zone has 5 hosts (namely, the NIDS, the PSQ, the PS, the CORR, and the DC) in each quad.

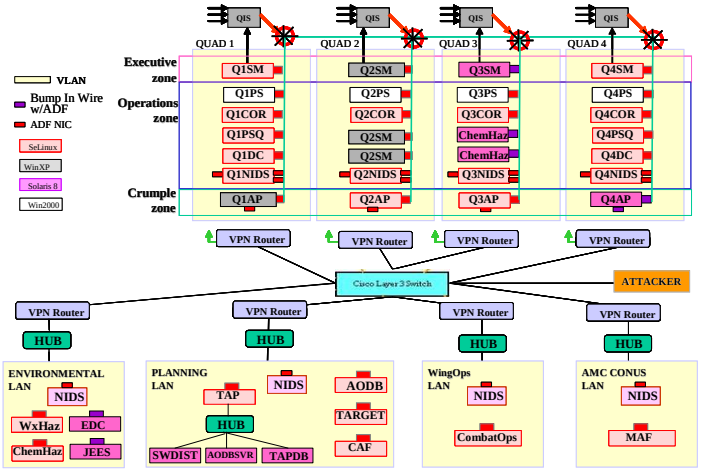


Figure 2: The Defended System

The NIDS is a network-based intrusion detection appliance, responsible for watching over network traffic within, coming into and going out of the enclave in which it resides. The PSQ host runs the main functions of the core part of the undefended system (i.e, the core JBI services including the JBoss server and the MySQL repositories). The PS host runs the policy server process that manages the ADF NICs in the system, and interfaces with the system management function in the executive zone for mounting adaptive responses involving the ADF NICs. The CORR host acts as an alert sink and runs two applications namely, the EMERALD correlator and the Advisor. The former consumes the alerts to correlate, ranks the aggregated alerts, and presents the ranked alerts in a GUI. The latter consumes the alerts to assess whether a given component can be trusted to remain a part of the system, and advises the system management function in the executive zone. It does so by interpreting an alert as an accusation made by a component about another; and considering factors such as whether the accuser has physical or logical connection or relation with the accused, whether or not the accused is accused by others, whether the accuser itself can be trusted, etc. The DC host runs the Downstream Controller (DC) application that acts as an intermediary avoiding a direct connection between the crumple and executive zones. In addition, the DC processes the heartbeats.

The AP host in the crumple zone acts as the port of entry for all traffic into a quad from the clients. For each process running in the operations zone (i.e., processes in the DC, PS, CORR and PSQ hosts) that the clients interact with, there is a corresponding proxy on the AP host performing rate-limiting and certain application-level checks (without violating the end-to-end authentication and encryption) before forwarding traffic to the core.

The SM host in the executive zone runs the System Manage (SM) application that manages defense mechanisms in clients and in its own quad, and also interfaces with the DPASA security operators.

The configuration of a client enclave is not changed except for the addition of a NIDS. The Client hosts run applications that make use of the mediated service in the operations zone. In the context of the JBI exemplar, these are applications that have specific roles in the mission. They are known as JBI clients, and interact with each other via publish, subscribe and query.

Network Level Enhancements: The defense-enabled system has a Virtual Private Network (VPN) router in front of each enclave, establishing a VPN trunk from a client enclave to each of the core enclaves. All hosts are equipped with ADF NICs, as shown by the small red rectangles. Hosts in the crumple zone have two ADF NICs since each crumple zone host has one client-facing and one core-facing interface. We worked around the lack of Solaris ADF device drivers by using ADF equipped Linux hosts as "bumps in the wire" (indicated by the small purple rectangle attached to a Solaris hosts). Each core enclave has a managed switch (HP Procurve 2524) controlling the communication between quads as well as inter-zone communication within its own quad. Each such switch is further controlled via a Quad Isolation Switch (QIS) by the 4 SMs. A Cisco 3750 was used to emulate the public IP network connecting client and core enclaves implemented as VLANs off the Cisco 3750.

Host Level Enhancements: In each host protection domain policies tightly define how a process can be started and limit it's the capabilities. In addition to SELinux (Linux hosts) and Cisco Security Agent (Solaris and Windows hosts) policy enforcement, the Java application components were further sandboxed by JVM security policies. Each host runs a component (known as LC) that periodically checks the file system integrity, CPU, memory and disk usage of the host, and acts as the aggregator and reporter of local host-based policy violations. The clients run 3 instances of LCs that monitor each other and can take over if one crashes.

New Dataflow: The undefended JBI had only one data flow (i.e., the PSQ operation). The DPASA architecture introduced 4 new types of dataflow in the defended system: 1) Heartbeats sent by each software component to the core, 2) Alerts from NIDS, HIDS, LCs and software components, 3) Coordination and command messages sent by SMs among themselves and to other components, sometime via the DCs, and 4) the status, key renewal and audit messages from the PS to ADF NICs.

With the addition of the 4 PSQ servers the new PSQ dataflow is as follows. A PSQ request first goes to the AP, and then the AP checks whether the traffic belongs to an existing session by consulting the SM via DC. If this succeeds, the request is forwarded to the PSQ server, which checks for signatures and permissions. If this succeeds, the request is disseminated to the other quads. Each quad processes the request and sends the appropriate signed ACK or response back to the client. The client completes the request if enough signed responses are received; otherwise it keeps resending the request (indefinitely). If the PSQ request is a publication, the IO is escrowed. When the escrow period is over, the IO is released for circulation. The PSQ dataflow, encapsulated within a DPASA JBI middleware implementation, uses sockets over the network and JMS between the server-side end-point and JBoss.

New Protocols: Key protocols introduced by the survivability architecture are described below.

The **registration protocol**, used when a client is ready to join the JBI, involves the SM and the client mutually authenticating each other, and is initiated by a human operator at the SM. The **alerts protocol** used for reporting suspected events works as follows. Alerts generated in a quad are sent to a "Tee" process on the CORR host of its own quad, which passes a copy of each alert to Emerald and the Advisor. Alerts generated by clients however go to each quad (through the Correlator Proxies). The **PSQ protocol** implements the basic PSQ functions used by all JBI applications. The PSQ protocol is fault-tolerant. When all four quads are participating in the protocol, JBI clients see correct behavior from the core even if any one of the PSQ servers is corrupt and behaving in an arbitrarily malicious way. If fewer servers are participating, the protocol may not tolerate corruption but it will tolerate crashes if at least two servers participate. Corrupt access proxies are tolerated as long as one proxy works correctly. Corrupt client behavior is tolerated in some cases, detected in others. A design goal of the PSQ protocol was to isolate it as much as possible to allow other domain specific service providers to be plugged into DPASA. The **TS protocol** implements a distributed fault-tolerant clock for use by both JBI clients and DPASA defenses. The 4 TS servers respond to requests for the current time. The protocol maintains a system-wide time despite corruption of any one of the TS servers. The **heartbeat protocol** is used to detect failures in the DPASA survivability components. The heartbeat messages are sent to the core where the SM on each quad uses this to display the status of the system. The SM participates in a number of protocols, key among the **SM protocols** are the SM-PS protocol to retrieve data and exert control on ADF NICs; the PSQAdmin protocol for administrative functions on the PSQ server local to the SM, i.e. quorum group management; and the PSQDB protocol for accessing the internal state of the PSQ's database, and recalling of IOs.

6. EVALUATION RESULTS

In the first round of red team exercises, conducted at the Air Force Research Laboratory (AFRL), Rome, NY (March 15-18, 2005), the survivable system was subjected to two separate 12 hour exercises. Two red teams launched a number of planned and ad hoc attacks. While the first red team caused a system slow down, the critical mission objectives were met. The second red team succeeded in disrupting the communication paths between clients provided by the commercial VPN routers but could not penetrate beyond that outer layer. The success enjoyed the red teams came at the expense of commercial products deployed at the periphery of the survivable system, and were limited to disrupting the availability of inter-enclave communication. When individual defense mechanisms were directly tested, the red teams were unable to compromise any mission requirement. The results show that the survivable JBI made it very hard for the attacker to penetrate into the system, or to cause significant damage inside it.

The second round (Nov 7-18, 2005) of adversarial testing was designed to evaluate the system's resilience when the high barrier to entry at the periphery is removed and the internal architecture and defense-mechanisms are attacked directly. The red team was augmented with developers and given total access to the system including source code, configurations, keys and passwords. The augmented red team considered a wide range of attack possibilities to violate confidentiality, integrity or availability of the system. Many of these possibilities were nullified by the design of the system, leaving a set of ~24 attack ideas, of which 19 were executed. In all the attack runs the red team was given one or more hosts within the survivable system to preposition attack code, and to launch their attacks. Of the 19 runs, the red team was able to stop the mission from completion within the stipulated time in only 4. In one of these 4 runs, the survivable system recovered to such an extent that it was able to complete its mission just 20 minutes after the stipulated time. Both automatic and human assisted responses occurred in 17 out of 19 runs. Some indication of attacker activity was reported automatically in 15 runs. No operator assistance was required in 3 runs

Overall, the distributed and embedded sensors provided excellent localization of compromises. Because of the multiple layers of cryptographic and agreement protocols the survivable system did not experience any compromise in confidentiality and integrity even with high level of access inside the system and prepositioned attack code. Even attacks that aimed to cause loss of confidentiality and integrity resulted in loss of availability only. The containment layers and physical barriers in the architecture limited the adversary's visibility into the system, and thereby limiting his ability to control it. The redundancy based protocol showed a very high level of resiliency—as long as one of the redundant components was providing some level of service, the system gracefully degraded and continued to operate.

7. CONCLUSION

As expected of a high-water mark research prototype, the survivable system needed expert operators to interpret the information it captured and to effectively mount some responses. Nevertheless, the DPASA survivable JBI is truly a pathfinder. It demonstrated that a tightly configured system with multiple and overlapping layers of defense can be designed and implemented under aggressive cost and scheduling constraints. It also broke new ground in validating the survivability properties of a system. By demonstrating that the red teams can be successfully challenged and forced to attack the system through the legitimate entry points, it provides a renewed level of confidence in the continued fight against the cyber-threat.

8. ACKNOWLEDGEMENT

The authors would like to thank Lee Badger of DARPA and Pat Hurley of AFRL for their support and valuable feedback.

9. REFERENCES

[1] Nelson, W., Farrel, W., Atighetchi, M., Clem, J., Shepard, M., and Theriault, K. APOD Experiment 2: Final Report. *BBN Technologies LLC, Technical Memorandum 1326* (Sep. 2002).

[2] AFRL JBI homepage: http://www.infospherics.org

[3] Markham, T., Meredith, L., and Payne, C. Distributed embedded firewalls with virtual private groups. In *Proceedings of the DARPA Information Survivability Conference and Exposition, Volume II* (Washington DC, April, 2003). IEEE