Intrusion Tolerant Services Through Virtualization: a Shared Memory Approach

Valdir Stumm Júnior*, Lau Cheuk Lung*, Miguel Correia‡, Joni da Silva Fraga‡, Jim Lau‡
* *Departamento de Informática e Estatística*
Universidade Federal de Santa Catarina, Florianópolis, Brazil
e-mail: {stummjr, lau.lung}@inf.ufsc.br
‡ *Faculdade de Ciências*
Universidade de Lisboa, Lisboa, Portugal
e-mail: mpc@di.fc.ul.pt
‡ *Departamento de Automação e Sistemas*
Universidade Federal de Santa Catarina, Florianópolis, Brazil
e-mail: {fraga, jim}@das.ufsc.br

Abstract

Much research aiming to design practical algorithms to support Byzantine Fault-Tolerant distributed applications has been made in recent years. These solutions are designed to make the applications resistant to successful attacks against the system, thereby making services tolerant to intrusions. Recently, some of these studies have considered the use of virtual machines for building a trusted computing environment. This paper presents SMIT (Shared Memory based Intrusion Tolerance), an architecture for Intrusion Tolerance using virtual machines that benefits from a shared memory to simplify the consensus protocol.

1. Introduction

The role that computing systems play in our society is growing in importance in the last decades. The trust and dependency over such systems have increased considerably, day after day. Given the importance of these systems to the correct operation of basic services in our everyday life, it is necessary that they behave properly even under the presence of faults, which can lead to large losses, from financial to human. Recently, operating systems faults have been frequently appeared as intrusions (e.g. viruses, trojans, etc), which are the result of a malicious attack that achieves success by exploiting one or more vulnerabilities of the system (e.g. bugs). To ensure that these systems remain available, correct and safe even in the presence of faults and vulnerabilities, is necessary the development of mechanisms to provide intrusion tolerance in these environments [1].

To ensure security to these systems, solutions for the development of fault-tolerant applications have been researched for more than twenty five years [2], [3], and in the last decade several studies with solutions for Byzantine Fault-Tolerant (BFT) State Machine replication (active replication) with practical viability were proposed [4]–[6]. The BFT protocols are, in general, replicated services in a set of machines that communicate between themselves to provide a safe and reliable service, even under the presence of a limited number f of malicious members (intruders) that behave out of the protocol specification. Such protocols are designed to allow the implementation of replicated services that are able to meet the requirements for reliability, integrity and availability, which are fundamental to achieving *dependability* [7].

Recently, some papers have proposed the use of virtualization technologies to implement intrusion-tolerant systems in a single physical machine [8], [9]. In these proposals, each replica is executed in a virtual environment, with the whole set of replicas running in the same computer. These proposals make feasible to implement the concept of services and operating systems diversity at virtual machines level [10]. An obvious advantage of such approach is the reduction of the replication cost to implement an intrusion tolerant service by the use of a single physical machine. But, the drawback of such approach is that the physical machine is a single point of failure.

It is noteworthy that despite the fact that the occur-

rence of machine failures is often due to crash faults than to Byzantine ones (or intrusions), the damages are usually much more severe in the occurrence of the latter. For instance, consider the case of a malicious intruder doing unauthorized financial operations in the servers of a bank or, an intruder doing terrorist acts in a computing system of a nuclear plant. Related works has shown that, to address the problem of malicious attacks on software, the solutions must adopt techniques of state machine replication along with design diversity techniques [11]. These proposals are based on the observation that Byzantine failures with malicious intent (intrusion) occur by the attempt to use the vulnerabilities of the part of computer system implemented in software (ie operating system, device drivers, services, applications, etc.).

This paper proposes SMIT (Shared Memory based Intrusion Tolerance), a new architecture for development of intrusion-tolerant services using virtualization technology, and an algorithm to perform services over that replicated architecture. The most important point about our paper is the application of a shared memory abstraction between the virtual machines to simplify the consensus protocol. Thus, we are able to avoid point to point message passing communication between the replicated services, what could increase the algorithm complexity to reach consensus. In this model, we also demonstrate that is possible to reduce from $N \geq 3f + 1$ to $N \geq 2f + 1$ where N is the total number of replicas to tolerate f byzantine ones, reducing the replication cost. Other papers had shown that this reduction is possible through the utilization of a safe component/wormhole [12]. Our study demonstrates that this safe component only needs to provide a simple shared memory abstraction to reach this reduction. Furthermore, the host machine does not play an active role in our proposal, because it only needs to provide a communication abstraction.

2. System Model

Our model has three kinds of systems: the *clients*, external to the virtualized environment, the *host* system, which is running over the bare hardware and can be a Virtual Machine Monitor (VMM) or an operating system which is supporting a VMM execution, and the *guests*, which are virtual systems running over the VMM. The latter systems support the execution of *service replicas*. The host system also hosts a *shared memory* block, which is used by the service replicas to exchange messages to implement a BFT protocol.

The assumptions described below are similar to the ones adopted by the related works (section 5). We assume that an attacker can take complete control over the actions of f replicated servers, but the damage that it can cause to the systems is restricted to the f violated virtual machines. In our model, the isolation must be provided by the VMM. We also use cryptographic techniques applied to the messages exchanged between replicas and clients to improve the isolation. The messages exchanged have attached a Message Authentication Code (MAC) [13] to authenticate the messages, generated using secret keys shared between the communicating pairs. We also assume that an attacker has no computational power to break the cryptographic techniques used.

Our model assumes that the host operating system can have vulnerabilities, but these can not be exploited by the virtual systems. To ensure this assumption, we trust that the VMM provides the required isolation to a safe execution of the virtual systems. This is a premise of any virtualization technology (e.g. VirtualBox, Xen, VMware, VirtualPC, etc). Besides, our model needs, by construction, that the host system is not accessible externally. This can be achieved through firewall techniques (e.g. Linux iptables) and/or disabling/removing the network drivers of the host operating system, blocking the access to the host network address and keeping the virtual machines IP addresses accessible. Thus, the attacker does not have any access to the host system. Because of this, the attacker does not even know of its existence.

In this study, we assume an asynchronous system, comprised by servers and clients. The clients are connected to the servers through a reliable point-to-point communication channel. The set of *replicas*, $S_n = \{S_0, S_1, ..., S_{n-1}\}$, also known as *service replicas*, is a set of at least n virtual systems running over a single physical host. The service replicas do not need to use the network to communicate between them, because all that communication is made by a shared memory area, which will be detailed later. Service replicas can perform two roles in the system: (*i*) primary replica, which is the responsible for defining the order in which the requests will be processed, and (*ii*) the backup replicas, which are the remaining virtual systems, which follow the order proposed by the primary to execute the requests. In our model, up to f faulty replicas (Byzantine faults [2]) are allowed, of at least *2f+1* processes. In our model, the Byzantine behavior involves stopping the execution, omission and sending inconsistent messages out of protocol specification. The *clients*, comprised by the set $C = \{C_0, C_1, ...\}$, are systems that must behave correctly and communicate with the service replicas through message passing over the network.

Software diversity techniques [10] are applied in the service replicas implementation. The main idea is that the replicas fail independently, ie, the failure of a replica does not mean the failure of others. This diversity can be implemented at the level of operating system and application (programming language and development methodologies).

Our model does not tolerate crash faults in the physical machine, related to the host, to the VMM and to the physical support. Remember that the main objective of this proposal is intrusion tolerance for the replicated service running into each VMs. A crash fault occurred in the host implies directly in the stop of virtual systems. The use of crash fault-tolerant techniques, such as active replication of the physical machine, can be applied to our model to make it also tolerant to such faults.

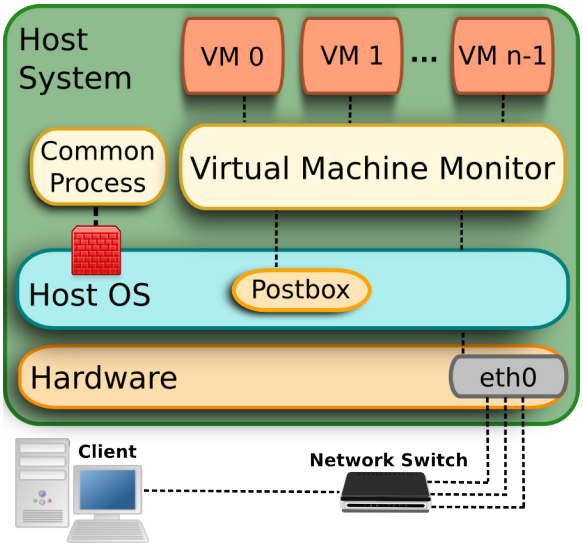

Figure 1. SMIT Architecture

2.1. The Postbox

As previously described, our architecture benefits from a shared memory area, which will also be called as *postbox* and will be used to exchange messages between the replicas. This shared memory abstraction must be provided by the host system to the guests. Any replica can store values at the postbox to be read by the other replicas. The correct replicas in the set S read the values in the exactly same order in which they were stored in the postbox. This component has a simple interface, which is composed by two methods:

- ***append(value):boolean***
- ***read():value***

The ***append*** method stores a value at the postbox, together with a replica identification (the VMM manages it, so there is no forgery). The value is stored immediately after the value written in the last append operation that has been executed. The return value is a boolean, indicating whether the operation has been successful executed or not. It is important to the VMM to employ a fair scheduling algorithm to manage postbox accesses, in order to avoid denial of service attacks. The ***read*** method returns a previously stored value. This operation always returns the next value to the last read.

Once stored at postbox, a value can not be changed, because the postbox must operate in ***append-only*** mode. It avoids the situation where a malicious replica writes a value, waits for a replica read operation, and change that value before another replica can perform its read operation. That situation could generate an inconsistency in the whole replicated service. Access and concurrency control at the postbox are a responsibility of the VMM. It is important to realize that the postbox is a finite capacity component. To avoid overflow of the shared memory, causing the shutdown of the protocol, a garbage collection mechanism is necessary to clean up postbox entries that already had their requests processed by the correct replicas.

As previously described, all the correct replicas have a homogeneous view of the postbox content. So, for any set W_i with k write operations, containing writes from the *n-th* to the *(n+k)-th* position in the postbox, all the correct replicas which made a set of k read operations from position n will get as result a set R_i of read operations, with the same content as W_i.

Figure 1 illustrates the overview of the whole architecture.

2.2. Algorithm Properties

The algorithm proposed here follows the line of State Machine Replication [3] algorithms. So, our algorithm must ensure the following properties to provide a correct service:

- **Safety**: the replicated service must behave like a centralized one.
- **Liveness**: requests sent from correct clients eventually complete, ie, the algorithm always makes progress, does not matter what happens.

In the state machine approach, it is necessary that the requests issued by clients are delivered (*agreement*) and executed in the same order (*total order*) by all the correct service replicas. In addition, the correct replicas must behave deterministically, ie, all of them starts its execution in the initial state and end in the same final state.

To ensure that the above properties match, our algorithm must be executed in an environment where no more than $\lceil \frac{(N-1)}{2} \rceil$ replicas behave incorrectly, ie, the system needs at least $N \geq 2f + 1$ replicas, where f is the total number of tolerated faulty members.

To ensure that all replicas execute the same requests in the same order, is needed a consensus protocol or its equivalent, an atomic multicast protocol. In PBFT [4], which is a protocol to support State Machine Replication, the total order and agreement aspects are defined by a leader replica through message passing, ie, the leader disseminates a sequence number proposal for a client request to the other replicas. In our algorithm, the total order is defined by the primary replica in a very simple way. That replica writes a sequence number to the request from the client in the postbox. The agreement is reached through direct communication of the correct client with all the replicas.

In our model, is possible to avoid all the communication steps that are needed in the previous agreement protocols. This is because our model has a memory area that is homogeneously viewed by all the replicas. Once the primary replica has wrote his proposal in the shared memory, the other replicas only need to read that value and execute the corresponding request.

To ensure *liveness*, our study proposes a view change protocol, triggered when the primary is suspected to be faulty.

3. Algorithm

The algorithm follows a succession of configurations called views. Each view v, the replica s is the *primary* replica iff $s = v \mod |S|$. The remaining replicas are *backups*.

In normal way of operation, only the primary could write values in the postbox and these values conform to the format $\langle$ PROP, c, t, h$\rangle$, where c is the client identification, t is the timestamp and h is the hash calculated over the message received. If other replica writes values in the postbox, that values are simply ignored by all correct replicas.

The client side of the algorithm is illustrated in Figure 2. It basically sends a multicast message *m* to all replicas of the set *S*. After sent *m*, the client waits for *f+1* replies with identical content.

```
1: multicast_send(⟨REQ, content⟩, S)
2: repeat
3:     buffer ← buffer ∪ recv()
4: until f + 1 matching replies ∈ buffer
```

Figure 2. Client request invocation task

The replica side of the protocol has two concurrent tasks. The first one (Figure 3) keeps listening for new messages and adding them to a buffer. If the replica is the primary of the current view, then it proposes the received message execution, appending the message hash to the postbox.

```
1: loop
2:     r = receive()
3:     if primary() then
4:         postbox.append(⟨PROP, c, t, hash(r)⟩)
5:     end if
6:     buffer ← buffer ∪ r
7: end loop
```

Figure 3. Request listener task

```
1: while messages in buffer do
2:     prop = postbox.read()
3:     if type(prop) == VC then
4:         buffer ← buffer ∪ prop
5:         view-change()
6:     else if type(prop) == PROP then
7:         req = buffer.search(prop) ∧ timeout
8:         if req == ∅ then
9:             postbox.append(⟨VC, view + 1⟩)
10:            view-change()
11:        else
12:            send(exec(req))
13:        end if
14:    end if
15: end while
```

Figure 4. Consensus task

The other one (Figure 4) is the task which ensures that the messages are processed in the same sequence by all the correct replicas. In this task, the replica keeps reading the postbox while there are messages in the buffer. If the message is found in the buffer before the timeout expires, the replica executes it and send the reply directly to the client. If the message read from postbox is not in the replica's buffer after the timeout (lines 7 and 8), then the replica suspects of the primary and tries to start a view-change. It will succeed only if at least *f+1* replicas are suspecting that the primary is faulty (Figure 5). Otherwise, if less than *f+1* suspects of primary correctness, the replica does not execute the view-change.

1: **if** f+1 matching VC in buffer **then**
2: view = view + 1
3: process each remaining request from view-1
4: **end if**

Figure 5. View Change Algorithm

A correct execution of the algorithm is summarized in figure 6. In the first phase (request) of the algorithm, the client *c* multicasts a request *req* to the replicas from the set *S*. The second phase is executed by the replicas to ensure the consensus in the message execution. After received the message, the primary replica *r0* writes its sequence proposal for *req* in the postbox. The backup replicas read that proposal. Once defined the sequence to *req* execution, the replicas execute it and send the reply to *c* in the third phase of the protocol.

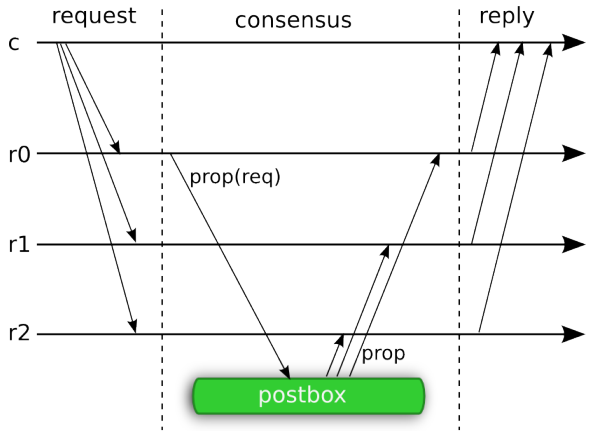

Figure 6. SMIT algorithm steps sequence

There are some important details that are not explicit in the algorithms listed above. For each message received, each backup replica starts a timer if there is not another active timer already running for another message. While there is a message waiting to be executed, there is a timer running. If the timer expires, the primary is then suspected by the replica. It avoids a primary postponing indefinitely the order proposal. A similar mechanism was proposed in PBFT [4].

Another important implicit detail about the proposed algorithms is concerned to the timeout at line 7 in Figure 4. When that timeout expires, the replica tries to start a view-change, which will succeed only if at least $f + 1$ view-change proposals have been made by different replicas for that message. Otherwise, the replica will back to the step at line 7, restarting the timeout, waiting for the delivery of the message related to the proposal made by the primary. The replica repeats those steps (waiting for the message delivery with a timeout and trying a view-change if the message is still not received) until either of the conditions below hold:

1) The message related to the proposal is finally delivered to the replica.
2) $f + 1$ different replicas are in the same situation and start a view-change.

Since our assumptions about the channel consider this as a reliable point-to-point channel, once the message is sent by the client, it will be eventually delivered to all the correct replicas.

4. Implementation

This section presents details about the prototype implementation of the algorithm and model described in the previous sections. To evaluate the prototype performance, we have used micro-benchmarks similar to those used in PBFT [4]. In these micro-benchmarks, we have executed null operations at the service replicas, ranging the argument and result size from 0 kB to 4 kB. For example, an operation named as 00 has 0 kB in messages of both argument and result. A operation 04 has a 0 kB message as argument and 4 kB message as result. We evaluate system response time and throughput, as described below.

The algorithm proposed in this paper has been implemented in Python, using the 2.6.2 version of the interpreter. We built a test environment, as described below:

- **Host system**: a Core 2 Quad CPU, with 8 GB of RAM, running Debian GNU/Linux 5.0, kernel 2.6.26-2.
- **VMM**: Sun VirtualBox 2.2.4 running over the operating system above.
- **Postbox**: a file in the host system, shared by VirtualBox to the guests through a mechanism called *Shared Folders*.
- **3 VMs**: each with 1 GB of RAM, running Ubuntu GNU/Linux 9.04 Server, kernel 2.6.28-11.
- **Clients**: running in a Core 2 Duo CPU, with 2 GB of RAM, running Ubuntu GNU/Linux 9.04 Desktop, kernel 2.6.28-15.

The clients are connected to the servers through a 100 Mb/s network switch.

4.1. Response Time

The prototype response time has been measured in our system by the clients, reading its local clock immediately before send the request and immediately after receiving the reply. For each kind of operation tested (varying the argument and result size), we have

tested 10000 requests, in two separated executions. We have created three distinct scenarios to evaluate the response time from our prototype:

1) **SMIT**: Our prototype with the service replicated over 3 VMs, each running the algorithm proposed.
2) **Single**: a single service (no replicated).
3) **Single-VM**: a single service running over a Virtual Machine.

The values in Table 1 are represented in milliseconds and correspond to the average calculated over the results obtained in each kind of execution. Our prototype incurs in a substantial increment in the response time of operations. It was expected, because our prototype offers a safe execution of the request, while the remaining two environments (*Single* and *Single-VM*) are just single executions of the service, without replication and the operations that are required to safely execute a replicated operation. Also noticeable is the fact that the reply size has more effect over the response time than the request size, because the client must receive and vote over at least *f+1* replies to complete the request execution. Then, operations that require large amounts of data as its replies, leads to larger response times. As in PBFT [4], it can be optimized by letting the client choose just one of the replicas to send the entire reply message. The other ones send only the message digest. Then, as in the normal case, the client waits for *f+1* matching replies. If the entire message received does not match the digests sent, the client requests to the replicas to send its entire messages too. Furthermore, is worth noting that the response times from the *Single-VM* scenario execution also presents a substantial increment compared to the values obtained in scenario *Single*, showing that the execution in a virtualized environment incurs in a substantial overhead in the response time.

Table 1. Response Time (milliseconds)

Operation	SMIT	Single	Single-VM
00	3.504	0.2223	0.5553
02	3.6873	0.6056	0.8930
20	3.7835	0.6037	0.8674
04	4.019	0.8049	1.0895
40	3.9631	0.8274	1.1344

4.2. Throughput

To evaluate the throughput of our prototype, we executed operations with both argument and result of 0 kB, varying from 0 to 20 the number of clients sending requests concurrently. Figure 7 shows the results obtained in this experiment. The throughput keeps growing as the number of clients is increased until reach around 600 operations per second, when it stabilizes. It can be explained by the overhead that exist in accessing the postbox in the current implementation, which is the component through which the replicas execute all the message exchange to reach consensus. As described above, postbox is implemented as a file shared by the host system to the guests, requiring the overhead generated by file system. Optimizations in such component can bring us better results related to the scalability for large number of concurrent accesses.

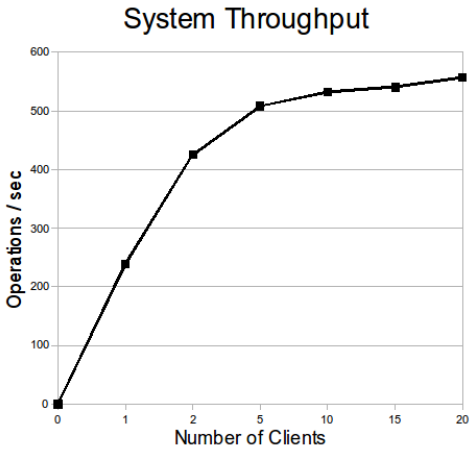

Figure 7. Throughput under different loads

5. Related Work

The concept and virtualization technologies to add Byzantine Fault Tolerance to computing systems has been recently studied. The VM-FIT architecture [9] is one of the first proposals to apply virtualization to Byzantine fault tolerance. The basic idea behind that architecture is the redundant execution of a service in many virtual machines over a single physical host. It allows a significant cost reduction to deploy a BFT system, because it avoids the requirement for hardware replication to create the service redundancy. The VM-FIT architecture requires a trusted component, which is the algorithm coordinator, being responsible to multicast the request from the clients to the service replicas and to vote over the replies from the replicas, to deliver only the correct value to the client.

There are other proposals that replicate a service through virtual machines over a single physical host. In the LBFT1 and LBFT2 [14] proposed approaches,

there is a trusted system, which is the protocol coordinator, responsible for the sequence number assignment to the message from the client. In this proposal, the coordinator is a special system, in the sense that it does not act like a service replica. The coordinator is only responsible for coordinate the requests sent by clients. Thus, both algorithms tolerates f faulty replicas using at least $2f+2$ systems, where one of them is the trusted coordinator, and the rest of them are services replica.

In SMIT, we need $2f+1$ virtual machines to tolerate f faulty replicas and the consensus is obtained with only one step of communication between the replicas.

In some active replication systems, the communication time between replicas does not grows indefinitely, even in asynchronous systems. This assumption is made, for example, in the original PBFT protocol and in other BFT protocols [4]–[6]. That assumption is required to circumvent the FLP impossibility [15]. In our case, this assumption is not too impressive, because encompasses only the communication between clients and replicas. The other ways of communication are made through the shared memory.

6. Conclusion

This paper have presented an architecture and an algorithm for BFT replication, using virtualization to deploy replication and service diversity to provide as a result an Intrusion Tolerance System less expensive than previous proposals. To evaluate the practical viability of the model, we have developed a prototype, running micro-benchmarks over it to measure the response time and the throughput obtained. The future works are focused on architectural and algorithmic improvements to support malicious clients, on the architecture implementation using another VMM (like KVM, Xen, etc.), on the possibility of create a postbox in main memory and on a distributed version, in order to also tolerate crash faults in the host system.

References

[1] J. S. Fraga and D. Powell, "A fault- and intrusion-tolerant file system," in *Proceedings of the 3rd International Conference on Computer Security*, Aug. 1985, pp. 203–218.

[2] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[4] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[5] J. Yin, J. P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 253–267, 2003.

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM Press New York, NY, USA, 2007, pp. 45–58.

[7] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on dependable and secure computing*, pp. 11–33, 2004.

[8] A. N. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Verissimo, "Intrusion-tolerant protection for critical infrastructures," *DI/FCUL TR*, pp. 07–8, 2007.

[9] H. P. Reiser and R. Kapitza, "VM-FIT: Supporting intrusion tolerance with virtualisation technology," in *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems*, 2007, pp. 18–22.

[10] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[11] M. A. Hiltunen, R. D. Schlichting, and C. A. Ugarte, "Building survivable services using redundancy and adaptation," *IEEE TRANSACTIONS ON COMPUTERS*, pp. 181–194, 2003.

[12] M. Correia, N. Neves, L. Lung, and P. Veríssimo, "Worm-IT–a wormhole-based intrusion-tolerant group communication system," *The Journal of Systems & Software*, vol. 80, no. 2, pp. 178–197, 2007.

[13] G. Tsudik, "Message authentication with one-way hash functions," *ACM SIGCOMM Computer Communication Review*, vol. 22, no. 5, pp. 29–38, 1992.

[14] B. G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance," in *Proceedings of the 2008 USENIX Annual Technical Conference*, 2008.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.