DIVERSYS: DIVErse Rejuvenation SYStem

Miguel Garcia, Nuno Neves, and Alysson Bessani

Universidade de Lisboa, Faculdade Ciências
mhenriques@lasige.di.fc.ul.pt,
{neves,bessani}@di.fc.ul.pt

Abstract. Replication has been used to build intrusion-tolerant systems, which are able to tolerate a limited number intrusions before the system is compromised. An important limitation of intrusion-tolerant systems is that if the system's replicas are similar, once a flaw is discovered and exploited in one replica, then it is easy to replicate it on the other replicas, compromising the whole system. To circumvent this limitation one must find a way to make these exploits occur independently. We propose the deployment of different operating systems in order to avoid common failures, making a system correct unless $f + 1$ replicas are compromised. However, if enough time is given to the adversary, then eventually $f + 1$ different replicas will suffer an intrusion. Hence, to reduce the size of this time window, we introduce diversity on recoveries, where the system will replace the faulty replicas with fresh and different ones (therefore, cleaning their faulty state) as the adversary compromises the replicas. The remaining challenge is to manage the recoveries without violating the availability of the system. Our contribution is to assess the risk on replicated systems to trigger recoveries.

Keywords: Diversity, Intrusion Tolerance, Rejuvenation, Operating Systems, Vulnerabilities.

1 Introduction

Despite the effort to verify and test software, it is extremely hard to produce source code that has no flaws. When the software is deployed these flaws can become intentional faults (malicious) or accidental faults. In the presence of faults the system can behave abnormally, i.e., stop providing the expected service.

One of the most efficient and transparent ways to deal with faults is to tolerate them. Replicated systems have been used to solve this problem in the last decades. Server redundancy (called replicas) allows the system to be able to tolerate faults in some of the nodes and keep the availability of the service. These systems follow the state machine replication abstraction to guarantee that every replica executes the same steps. A key building block of fault-tolerant systems is Byzantine fault-tolerant (BFT) protocols, which guarantee the correct behavior in spite of arbitrary faults, provided that a minority (usually less than one third [1]) of the components are faulty. Works concerning these protocols tend to assume that nodes fail independently [2–5]. However, if the replicas have

the same software, once one flaw is discovered and exploited then it is easy to replicate the attack on the remaining replicas until the system is compromised. Moreover, if an accidental fault occurs then every replica will progress, by the state machine replication definition, into a faulty state.

To circumvent this limitation one must substantiate the fault independence assumption by construction. Diversity allows one to build safer replicated systems through the assumption that different components exhibit independent failure modes. Still, given enough time, the attacker will eventually compromise $f+1$ different replicas. The only way to deal with this limitation is to make the optimistic assumption that the (at most) f faulty replicas of the the system can be repaired before $f+1$ occur[1]. Moreover, every diverse replica is fully patched, clean from known vulnerabilities, although, a remaining challenge is to deal with zero-day vulnerabilities, i.e., vulnerabilities that only became known when the exploit occurs. Although zero-day vulnerabilities require that the attacker spends some time to discover a new vulnerability for each different OS, Symantec stated that 14 zero-day vulnerabilities were discovered in 2010 [7], and 8 in 2011 [8]. A powerful attacker could silently discover vulnerabilities on $f+1$ replicas, and then run the exploits at the same time, taking over the system.

Proactive recovery (PR) is a way to avoid this scenario. PR allows the system to periodically rejuvenate replicas: *i)* if the replica was silently compromised the faulty state will be cleaned; *ii)* if a replica is correct but it is being probed by the attacker then a recovery will force the attacker to restart the probing. Besides the PR, robust systems tend also to invoke recoveries if a misbehavior is detected [9]. The system is intrusion-tolerant as long as the recoveries occur faster than the $f+1^{th}$ fault. Moreover, this recovery must modify the replica in such a way that it is not trivial for an attacker to compromise it again. One way to do it is to create a diverse software configuration for the rejuvenated replica, with a (hopefully) different set of (unknown) vulnerabilities. One can find several opportunities to use diversity (for example at the operating system, application, hardware, location, and configuration). A previous work from the authors show that OS diversity might be effective against common mode faults at the OS level [10].

We propose a intrusion-tolerant system support called DIVERSYS (DIVErsity Recovery SYStem), which enables a system' replicas to fail independently through automatic management of diverse configurations and recoveries. Any f replicas can fail within a bounded evaluation condition, i.e., between recoveries, first to clean faulty states and then to stop attacks in progress (i.e., the attacker probing the system). We want to present a new way to manage recoveries: timely loose, without unavailability periods, adaptable and configurable by the administrator. We propose a metric to evaluate the replicated system, the *Risk level*, which is calculated based on the risk of replica's operating system being intruded. DIVERSYS uses this metric to guide decisions of replicas' proactive recovery, changing the system to keep its risk level under a predefined threshold. In the

[1] However, even with this repairing, there is a natural and slow decaying on the theoretical reliability of the recoverable system [6].

end, the objectives of DIVERSYS are to detect exploits as soon as possible, to perform fast recoveries and to install patches as soon as they are available.

2 Related work

Diversity. Design diversity was introduced in 1975 as a mechanism for software fault tolerance [11]. N-version programming is a technique to create diverse software systems [12]. The objective is to achieve fault tolerance, assuming that designs and implementations developed independently will exhibit failure diversity.

Gashi et al. [13] analyzed bug reports for four database servers (PostgreSQL, Interbase, Oracle, and Microsoft SQL Server) and verified which products were affected by each bug reported. They found a few cases of a single bug affecting more than one server, and that there were no coincident failures in more than two of the servers.

Garcia et al. [10] present a study based on the National Vulnerability Database data (NVD)[2], that shows that there is strong indication that different operating systems can exhibit vulnerability independence. They collected the NVD data and analyzed how many common vulnerabilities affected two OSes, and what is the best way to achieve vulnerability independence in a replicated system with diversity.

Rejuvenation. Software rejuvenation was proposed in the 90's by Huang et al [14]. The initial motivation was to reset the state of a server, in a client-server communication model, taking advantage of the idle time of the server to clean the state.

Castro and Liskov presented BFT-PR, a new algorithm that enhances the availability of the PBFT [15] by employing Proactive Recovery (PR) [2]. This enhancement requires that each replica has a watchdog timer that periodically yields the control to a recovery monitor, that is stored in a read-only memory. This watchdog allows the system to trigger timeouts to hold the recoveries. Hence this solution requires strong synchrony assumptions.

Sousa et al. proposed an intrusion-tolerant system [9] with proactive-reactive recovery. The system detects faulty replicas and forces them to recover, without sacrificing periodic rejuvenations. The technique can only be implemented with some synchrony [16], due to the recovery trigger clocks. To overcome this limitation the authors proposed an hybrid system model: the *payload* is an *any-synchrony* subsystem, and the *wormhole* is a *synchronous* subsystem. This is a limitation which we address in this paper.

Rodrigues et al. [17] proposed BASE, a PBFT protocol extension with a filesystem storage, similar to NFS. The replicated filesystem uses proactive recovery, although the authors were optimistic on the experiments and had an imprecise approach for estimating the reboot times as 30 seconds (to simulate

[2] http://nvd.nist.gov/

a reboot), which is very fast time for an OS recovery. A recovery was started every 80 seconds in a round-robin discipline.

Distler et al. [18] and Reiser and Kapitza [19] identified several issues that make virtualization useful for proactive recovery, which allows to create an hybrid fault model system: periodic recoveries can be triggered by a service in the privileged domain, which makes the replacement of the application domain, reducing the downtime of the recovery. Although it could not assure continuous availability because the service has to be suspended for 3 seconds in each recovery.

Roeder and Scheinder [20] propose the use proactive obfuscation, whereby each replica is periodically restarted using a clean generated diverse binary. Proactive obfuscation employs semantics-preserving program transformations. There is a technical limitation in this line of research: it seems very hard to transform (re-compile) a code that is not open (e.g., Windows OSes). The rejuvenations are timely triggered by synchronized processors, built on the assumption that hardware similarity and a hybrid architecture provide better synchrony conditions.

These cited works on rejuvenation comprise some of the state-of-the-art solution on recoveries. However, a common requirement is that recoveries are timely triggered. Typically by requiring strict scheduling, design assumptions (i.e., hybrid systems with synchronous parts), and trusted components such as watchdogs to provide clock timeouts. In this work we intend to solve the recovery management problem with loosely time requirements.

3 DIVERSYS: DIVErse Rejuvenation SYStem

This section presents the architecture of DIVERSYS, a diversity and recovery management for fault-tolerant replication systems running in a virtualized environment. This system implements proactive and reactive recovery mechanisms and introduces diversity during recoveries. DIVERSYS is built on a hybrid architecture, although without requiring the support of a real-time system, as in Sousa et al [16], but instead, relying on the shared knowledge about replicas' risk to define when it is time to recover (or change) a replica.

3.1 Diversity

In order to deploy a system with diversity we use distinct operating systems, aiming for fault independence. We can find several OS families like Linux, BSD, Solaris and Windows. Each family has different distributions (e.g., Debian, Redhat and Ubuntu for the Linux family), and each distribution has several releases available on the web (e.g., Debian 6.0, Debian 5.0 and Debian 4.0).

Figures 1 and 2 show two replicated systems with 4 replicas that tolerate one fault (according to the $n \geq 3f+1$ relation [2]). In Figure 1 the replicated system has no diverse components, therefore once a replica is compromised it is easy to replicate the attack on $f+1$ replicas. In Figure 2 the replicated system has

diversity on the components, then the attacker needs more effort to compromise the $f+1$ replicas, because it is expected that they do not share vulnerable code.

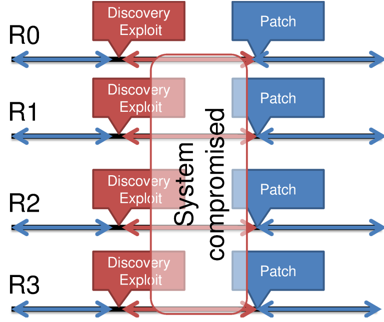

Fig. 1. Zero-day vulnerabilities in a replicated system without diversity. When a vulnerability is discovered the attacker can easily compromise the whole system reproducing the attack.

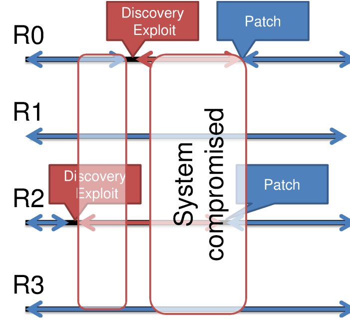

Fig. 2. Zero-day vulnerabilities in a replicated system with diversity. When a vulnerability is discovered the attacker needs to make more effort to compromise the whole system, since reproducing the attack is ineffective as vulnerabilities are not shared.

With diverse components, one is able to potentially assure fault independence, assuming that the attack's power is the same in both figures scenarios. However, the attacker will compromise $f+1$ replicas within a certain amount of time. Then we require the use of recovery mechanisms to guarantee that the system refreshes faulty replicas before $f+1$ replicas are compromised. Therefore we are reducing the time in which a replica is exposed to malicious users.

Notice that although we define DIVERSYS for OS reconfiguration, the replacement method can be employed for any software component for which diversity exists (e.g., databases, web servers, ftp servers). We focus on OS due to the existence of real-world data that can be used to feed our method.

3.2 OS recycling

In the previous section we stated that recovery mechanisms are needed, in this section we will explain how we use these mechanisms. On each recovery a diverse operating system should be loaded.

Figure 3 shows the vulnerability life-cycle of two vulnerabilities in two different OSes. *OS X* runs until the exploit *X1* is detected, and then the system replaces *OS X* with *OS Y*, which has one *vulnerability Y1* with its own life-cycle. While the system is running, a patch will be available and installed for *vulnerability X1* and *OS X* is again able to be used safely.

Our solution makes one main assumption: eventually a patch will be produced for a given vulnerability. However, the time it takes for a patch to be made available can be large. Additionally, if many other OSes are available (i.e., if

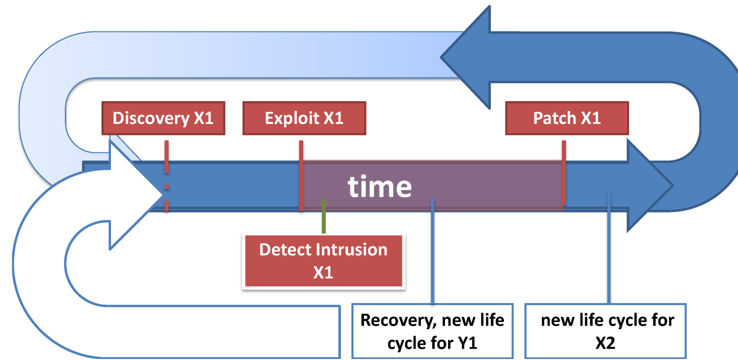

Fig. 3. Vulnerability life-cycle: In OS X first the vulnerability X1 is discovered, then X1 is exploited, the system detects the fault and starts a recovery, replacing OS X with OS Y and a new life-cycle starts for vulnerability Y1 in OS Y. Between the exploit of X1 and the patch of X1 OS X can not be used. After X1 is patched OS X can be used again.

there are many options) we do not need to recycle already used (i.e., vulnerable or/and patched) OSes.

3.3 DIVERSYS architecture

The system is composed by two main parts: the safe part and the unsafe part. Figure 4 shows the generic architecture of DIVERSYS. The architecture is built to take advantage from virtualization mechanisms [19].

Safe part. In this part the system is built to be secure and isolated. It is composed by n hypervisors, which are paired through reliable, timely and independent channels. These channels guarantee that it is possible to securely replicate information within an upper time bound. Each hypervisor has a secure connection to an OS pool, which is a virtual machine image repository where diverse OSes images are stored.

Unsafe part. This part is built to be secure, although since it is connected to the Internet we must expect threats from external agents. It is responsible to provide the service for clients. In order to detect malicious interactions in the replicas we use BFT state machine replication (e.g., [2, 22]) to guarantee that each server executes the same requests in the same order and replies correctly to the clients. This part runs in a virtual environment, the objective is to use this virtual instance as a sandbox, that can be reconfigured (by the safe part) when deemed insecure.

3.4 Risk level

This section describes the main contribution of the paper, the risk level metric and how we use it of trigger proactive recoveries of replicas. Some of the related

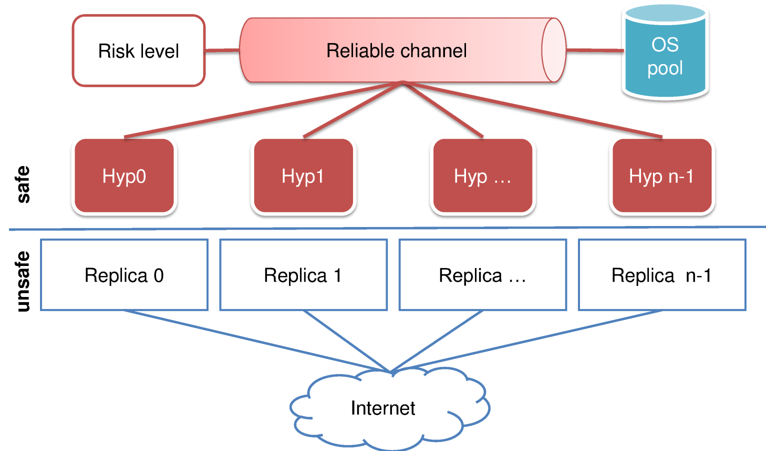

Fig. 4. DIVERSYS architecture is built on two main parts, the *unsafe part* which is exposed to potentially malicious agents, and the *safe part* which is a hypervisor that controls the *unsafe part*.

works in the literature depends on the existence of secure clocks to trigger recoveries [2, 9], which forces one to assume synchronized clocks between replicas (usually in the a secure sub-part of the system, e.g., wormhole or hypervisors). Our recovery scheduling is based on the *Risk level* of the system, therefore the system needs to be monitoring its risk level.

The main idea is that the *Risk level* should not exceed α, which is a parameter set by the administrator during system's deployment. This value is the security threshold: whenever the risk level reaches α a replica recovery is triggered. The candidate to be recovered is the replica with the major contribution to the *Risk level* of the system.

The *Risk level* is calculated based on the operating systems selected to run on each replica of the system, and the expected number of common vulnerabilities among different operating systems [10]. For instance, if the system runs four OSes with the same version then it is much likely to have a greater *Risk level* (a single vulnerability can expose all replicas). But if the system runs different OS versions, it is more likely to have a lower *Risk level*. Also, via BFT protocols or distributed IDS (Intrusion Detection System), the system can detect a replica as faulty, then the system increases the faulty OS's risk value (see Table 1), and eventually will be replaced by a new OS, which decreases the *Risk level*. In order to proactive-recover all replicas, DIVERSYS keeps increasing the degradation value, otherwise the risk level is constant until some replica is faulty. This increase rate depends on the specific OS, therefore each OS version has a degradation time.

Figure 5 shows a replicated system with OS diversity. The *Risk level* increases at every time unit, and when it reaches the α value the system performs a recovery. In order to minimize the risk level, the system must evaluate which

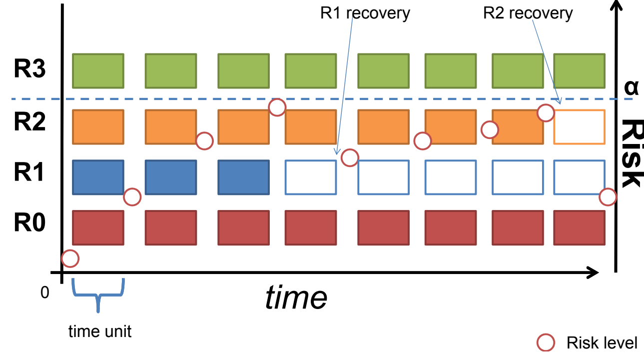

Fig. 5. Timeline of the replicated system: *Risk level* grows with time. When *risk level* achieves α, the system performs a recovery and the risk level decreases.

OSes are running, and what is the best OS to replace, and which new OS should be used. As a caution note we are not addressing the diversity selection problem, we can use an algorithm like [21] to solve this problem.

Risk level calculation. The value for the *Risk level* of a given system (based only on the OSes used by its replicas) can be calculated using three equations. To start, the *Risk level* itself is given by Equation 1, where n is the number of replicas, R_i is the ith replica's contribution, and the commonality value (CV).

$$Risklevel = (\sum_{i=0}^{n} R_i risklevel) + CV \tag{1}$$

Equation 2 is used to calculate the risk of each OS running in each replica. Where Δdt is a constant set for each OS and represents the degradation time rate, rd is the current round of each OS. Each OS running has it owns rd counting, because recoveries occur at different times.

$$R_i risklevel = \sum_{i=0}^{rd} \frac{rd}{1 - (\Delta dt - i)} \tag{2}$$

Equation 3 allows one to calculate the commonality of two OSes. The Equation returns the count of common vulnerabilities affecting R_i and R_j OSes.

$$CV = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} common(R_i, R_j) \tag{3}$$

While *Risk level* is below the α threshold the system is considered not in risk. Remember that α is defined by the administrator, and thus it can be adjusted depending on the criticality of the deployed system. During the system's execution, the algorithm only needs to calculate CV once per recovered replica.

3.5 Illustrative Results

In this section we present a simple illustration of our approach considering real data. To evaluate the *Risk level* contribution on a replicated system we selected OS vulnerabilities published by NVD between 2001 and 2012. In the experiments we consider OSes from different families: OpenBSD, FreeBSD, Debian, Ubuntu, Windows2003 and Windows2008.

The contribution of each OS is *positive* when the *risk level decreases*, and *negative* when the *risk level increases.* Analyzing Table 1 values one can see that OpenBSD or Debian have negative values, and Windows2008 has positive values, that means that Windows2008 has a more negative contribution than OpenBSD and Debian to *Risk level.* Each entry of Table 1 is the result of Equation 2 with rd from 0 to 15, and Δdt is the average number of days that takes one OS to have a vulnerability published after another.

rd	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Δdt
OpenBSD	-34	-33	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	49
FreeBSD	-33	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	48
Debian	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	43
Ubuntu	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11	41
Windows2003	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	7
Windows2008	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	7

Table 1. Degradation level of each OS, outcome from Equation 2.

Table 2 presents the results of Equation 3 for every possible set. Recall that CV is the sum of common vulnerabilities of each pair of the set (without vulnerability duplicates).

OS set	CV	**OS set**	CV
Openbsd, Freebsd, Debian, Ubuntu	46	Openbsd, Debian, Win2003, Win2008	211
Openbsd, Freebsd, Debian, Win2003	35	Openbsd, Ubuntu, Win2003, Win2008	213
Openbsd, Freebsd, Debian, Win2008	35	Freebsd, Debian, Ubuntu, Win2003	11
Openbsd, Freebsd, Ubuntu, Win2003	37	Freebsd, Debian, Ubuntu, Win2008	10
Openbsd, Freebsd, Ubuntu, Win2008	37	Freebsd, Debian, Win2003, Win2008	211
Openbsd, Freebsd, Win2003, Win2008	244	Freebsd, Ubuntu, Win2003, Win2008	211
Openbsd, Debian, Ubuntu, Win2003	13	Debian, Ubuntu, Win2003, Win2008	219
Openbsd, Debian, Ubuntu, Win2008	12		

Table 2. Commonality value (CV), which is given by sum of the common vulnerabilities between OS, the outcome from Equation 3.

The best set, according to [10], from Table 2 is: OpenBSD, Debian, Ubuntu and Windows2008, then the set CV is 12. In Table 3 we show the *Risk level* (given by Equation 1 for the selected OSes). We set the α value as -15, each round the system updates the *Risk level*. A recovery will take place in the 13th round. The candidate to be replaced with a new diverse OS is Windows2008, because it has the most negative contribution to *Risk level*. The system replaces it with Windows2003, which has its round counter as 0 and CV is then set to 13.

round	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Risk level	*-68*	*-64*	*-60*	*-56*	*-52*	*-48*	*-44*	*-40*	*-36*	*-32*	*-28*	*-24*	*-20*	*-16*	(-12)	(-8)
OpenBSD	-34	-33	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19
Debian	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13
Ubuntu	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
Windows2008	8	9	10	11	12	13	14	15	16	17	18	19	20	21	(22)	(23)
round	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1
Risk level	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*-25*	*-21*
Windows2003	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-33	-32

Table 3. *Risk level* evolution during 15 rounds (Equation 1). At the 13th the system performs a recovery, replacing Windows2008 with Windows2003.

Table 3 shows the *Risk level* contribution in action: the risk reaches α in the 13th *round*, and then a recovery is performed replacing Windows2008 with Windows2003. In the 14th *round*, the overall *Risk level* (eighth row) decreases to -25, even with each OS increasing its *Risk level*.

4 Limitations & Future work

Adding diversity to a system inevitably introduce some limitations or difficulties. The main limitation we have found in this work was to obtain data on vulnerability life-cycle as in [23]. First because Open Source Vulnerability Data Base (OSVDB)[3] is temporarily unavailable[4]. Second, most of the information available in the web is not structured, or do not contain the information we needed. Therefore it is infeasible to manually check every source of information, and cross data manually. In another issue, each OS has a different way to be patched, this problem is even more complex when applications are also considered.

Regarding DIVERSYS and the *Risk leve*, we are aware that it is impossible to guarantee that our system is (always) secure, however, our approach gives

[3] http://www.osvdb.org/

[4] http://blog.osvdb.org/2012/03/30/were-still-here-update-on-osvdb-project-data-and-exports

more confidence using several techniques that mitigate the attack's consequences. DIVERSYS uses proactive and reactive recoveries together with diversity management to avoid common failures in replicated long-lived critical systems. The recovery mechanisms within DIVERSYS are built without hard timing assumptions, contrary to some state-of-the-art approaches. We manage the recoveries based on security information about the software running on the system replicas (in this paper we focus on OSes). This information is focused on the common vulnerabilities, in order to present a set of replicas with less probability from being exploited with the same flaw, and the average in which vulnerabilities are discovered for each OS, in order to set a degradation factor to each replica.

In the future we will implement automatic mechanisms to gather vulnerability and patch information in order to automatically update the OS images in the OS pool. Moreover, we also want to integrate our approach with recent techniques for automatic patching replicas [24]. Finally, we want to validate the *Risk level* metric with further data and gain more experience on its use for real system. Our feeling is that there are still more parameters that can contribute with the risk evaluation of a system, in particular it is worth to investigate the relationship of this metric with recent work on using reliability theory to evaluate intrusion-tolerant systems [6].

Acknowledgments. This work was partially supported by the EC through project FP7-257475 (MASSIF) and by the FCT through the Multiannual (LaSIGE) and the CMU-Portugal Programmes, and the project PTDC/EIA-EIA/100894/2008 (DIVERSE).

References

1. Lamport L., Shostak R. and Pease M.: *The Byzantine Generals Problem*, ACM Transactions on Programing Languages and Systems, 1982, 4(3): 382–401.
2. Castro M. and Liskov B.: *Practical Byzantine Fault-Tolerance and Proactive Recovery*, ACM Transactions on Computer System, 2002, 20(4): 398–461.
3. Correia M., Neves N. and Veríssimo P.: *How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems*, in Proceedings of the IEEE Symposium on Reliable Distributed Systems 2004, 174–183.
4. Bessani A., Alchieri E., Correia M. and Fraga J.: *DepSpace: A Byzantine Fault-Tolerant Coordination Service*, in Proceedings of the ACM/EuroSys Conference on Computer Systems, 2008, 43(4): 163–176.
5. Moniz H., Neves N., Correia M. and Veríssimo P.: *RITAS: Services for Randomized Intrusion Tolerance*, IEEE Transactions on Dependable and Secure Computing, 2011, 8(1): 122–136.
6. Brandão L. and Bessani A.: *On the Reliability and Availability of Replicated and Rejuvenating Systems Under Stealth Attacks and Intrusions*, Journal of the Brazilian Computer Society, Springer London, 2012, 18(1): 61–80.
7. Symantec Internet Security Threat Report, Vol. 16, published: April 2011. `http://msisac.cisecurity.org/resources/reports/documents/SymantecInternetSecurityThreatReport2010.pdf`.

8. Symantec Internet Security Threat Report, Vol. 17 published: April 2012. `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf`.
9. Sousa P., Bessani A., Correia M., Neves N. and Verissímo P.: *Highly Available intrusion-tolerant Services with Proactive-Reactive Recovery*, IEEE Transactions on Parallel and Distributed Systems, 2010, 21(4): 452–465.
10. Garcia M., Bessani A., Gashi I., Neves N. and Obelheiro R.: *OS Diversity for Intrusion Tolerance: Myth or Reality?*, in Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, 2011, 383–394.
11. Randell B.: *System Structure for Software Fault Tolerance*, ACM SIGPLAN Notices, 1975, 10(6): 437–449.
12. Avizienis A. and Chen L.: *On the Implementation of N-Version Programming for Software Fault Tolerance During Execution*, Software Engineering, IEEE Transactions, 1985, 11(12): 1491–1501.
13. Gashi I., Popov P. and Strigini L.: *Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers*, IEEE Transactions on Dependable and Secure Computing, 2007, 4(4): 280–294.
14. Huang Y. and Kintala C.: *Software Implemented Fault Tolerance: Technologies and Experience*, 1993.
15. Castro M. and Liskov B.: *Practical Byzantine Fault Tolerance*, in Proceedings of the Symposium on Operating Systems Design and Implementation, 1999.
16. Sousa P., Neves N. and Verissímo P.: *How Resilient are Distributed f Fault/intrusion-Tolerant Systems?*, in Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, 2005.
17. Rodrigues R., Castro M. and Liskov B.: *BASE: Using Abstraction to Improve Fault Tolerance*, in Proceedings of the ACM Symposium on Operating Systems, 2001.
18. Distler T., Kapitza R. and Reiser H.: *Efficient State Transfer for Hypervisor-Based Proactive Recovery*, in Proceedings of the Workshop on Recent Advances on intrusion-Tolerant Systems, 2008.
19. Reiser H. and Kapitza R.: *Fault and Intrusion Tolerance on the Basis of Virtual Machines*, Tagungsband des 1. Fachgesprch Virtualisierung, 2008.
20. Roeder T. and Schneider F.: *Proactive Obfuscation*, ACM Transactions on Computer Systems, 2010, 28(2): 4:1–4:54.
21. Garcia M., Bessani A. and Neves N. *Diverse OS Rejuvenation for Intrusion Tolerance*, in poster session of the IEEE/IFIP International Conference on Dependable Systems and Networks, 2011.
22. Sousa J., Branco B. Bessani A. and Pasin M.: *Desempenho e Escalabilidade de uma Biblioteca de Replicação de Máquina de Estados Tolerante a Faltas Bizantinas*. in Simpósio de Informática INFORUM, 2011.
23. Frei S., May M., Fiedler U. and Plattner B.: *Large-scale Vulnerability Analysis*, in Proceedings of the SIGCOMM workshop on Large-scale attack defense, 2006, 131–138.
24. Costa M., Crowcroft J., Castro M., Rowstron A., Zhou L., Zhang L. and Barham P.: *Vigilante: End-to-end containment of Internet Worm Epidemics*, ACM Transaction on Computer Systems, 2008, 26(4).