Diverse OS Rejuvenation for Intrusion Tolerance

Miguel Garcia, Alysson Bessani and Nuno Neves
LaSIGE University of Lisbon, Faculty of Sciences – Lisbon, Portugal

***Abstract*—Proactive recovery is technique that periodically rejuvenates the components of a replicated system. When used in the context of intrusion-tolerant systems, in which faulty replicas may be under control of some adversary, it allows the removal of intrusions from the compromised replicas. However, since the set of vulnerabilities remains the same, the adversary can take advantage of the previously acquired knowledge and rapidly exploit them to take over the system. To address this problem, we propose that after each recovery a replica starts to run a different (or diverse) software. As we will explain, the selection of the new replica configuration is a non-trivial problem, since we would like to to maximize the diversity of the system under the constraint of the available configurations.**

***Keywords*-Diversity, Vulnerabilities, Operating Systems, Intrusion Tolerance, Proactive Recovery.**

I. Introduction

Byzantine fault-tolerant (BFT) replication is an active research area that produced a number of results in the last 12 years. The proposed algorithms usually guarantee correct operation if at most f out-of n replicas deviates from their expected behavior arbitrarily.

BFT replication is considered to be a fundamental component of *intrusion-tolerant systems* [1], since arbitrary faults can model malicious behavior in replicas attacked and intruded by an active adversary.

Despite that, BFT replication alone has strong limitations once malicious faults are considered. One of the most important limitations is that given sufficient time, an adversary that was able to compromise $f+1$ replicas and then break the assumption that at most f replicas are faulty, exhausting the resources of the system [2].

A way to deal with this limitation is to employ periodic rejuvenations of replicas [2]–[4], a technique commonly called proactive recovery (PR). The rationale of these rejuvenations is to clean the state of the system, reboot the machine with code from a read-only storage (e.g., a CD-ROM) and validate/fetch the service state from other (correct) replicas. An intrusion-tolerant system with proactive recovery decreases the time an adversary has to compromise $f+1$ replicas from the complete system lifetime to a small *window of vulnerability* comprising approximately the period to rejuvenate the whole system.

Although periodic rejuvenations bring benefits in terms of reliability [5] it has a the following problem: the vulnerabilities exploited on previous incarnations of the replica may still be exploitable after the recovery. This limitation makes it very easy for a smart adversary to create a script to automatically compromise the replica again just after the rejuvenation.

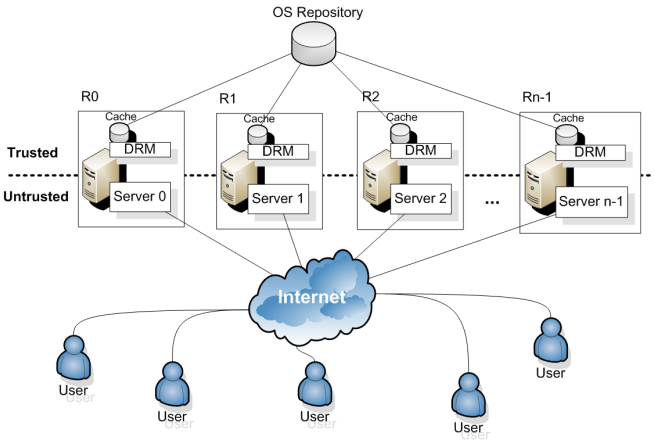

Figure 1. Architecture of the rejuvenation system.

This paper proposes an architecture to exploit the opportunistic diversity available from COTS (Component of the Shelf) software such as operating systems, database management systems, virtual machines and cryptographic libraries. The main objective is to change the configuration of a replica in order to modify its *vulnerability set* after the recovery. In particular we extend the PRRW (Proactive-Reactive Recovery Wormhole) architecture [4] with a *configuration selector* able to choose system configurations for recovering replicas preserving some expected fault independence between them.

The problem we address here was also considered in two previous works. Sousa et al. stated the need for diversity in time (i.e., changing replicas after a recovery) in a previous paper [6]. This work also suggested possible sources of diversity, however, no method for the selection of the new configurations was proposed and neither concrete results were presented. A more recent work proposes the use of program transformations for generating different application binaries after a recovery [7]. This work does not evaluate possible options and neither considers changing COTS software on a recovery, and thus can be seen as a technique for improving diversity of the same software component, being thus complementary to what we are proposing here.

II. System architecture

An Intrusion-tolerant (IT) system is typically composed by n replicated servers $0,...,n-1$ that implement a given service, for example, a file system or a database. Users

contact the replicas following the rules of the service – they send requests to one or all servers, and then select one of the returned responses (see below the line part of Figure 1). Servers keep their state consistent by running a replication protocol that is able to tolerate Byzantine failures. The system maintains a correct behavior even if there is an undetermined number of malicious users and/or if an attacker controls up to f replicas (with $n \geq 3f+1$).

The aim of the diversity rejuvenation service is to ensure that this last invariant continues to be valid throughout the lifetime of the system. It basically employs two mechanisms. First, replicas run diverse software to guarantee that vulnerabilities are not shared. If this is true, then the adversary would need to spend a considerable time to compromise each replica, since previously found exploits cannot be re-used to create intrusions in further servers. Second, periodically each replica is rejuvenated with a new diverse software, removing the effects of some prior intrusion, and therefore making the adversary start over. In order to ensure the availability of the IT service, rejuvenations occur in a round-robin fashion every T time units (i.e., at time (t_0+kT) starts the rejuvenation of replica i, with $i = (k \ mod \ n)$ and t_0 is the instant when the system was initialized).

The architecture of the rejuvenation service is depicted in the part above the line of Figure 1. Virtualization is used to divide each replica in two logical components, where the server software is run in a separate virtual machine and the *diversity rejuvenation module* (DRM) is executed in the hypervisor. This setup provides an acceptable level of protection for the DRM because the hypervisor is isolated from the virtual machines. Therefore, if an adversary manages to exploit a vulnerability in the OS supporting the server execution, he or she will not be able to propagate the intrusion to the hypervisor and affect the correctness of the DRM. Additionally, replica rejuvenation can also be performed in an effective manner by carrying out the following steps:

- DRM starts a new virtual machine with a diverse *OS configuration* stored in the local cache. This virtual machine runs in parallel with the current server replica.
- A new server is initiated in the virtual machine by running the necessary setup operations, which might include contacting the other server replicas to obtain an updated state of the IT service.
- The virtual machine of the current server is shutdown and discarded, and the new server takes the place of the old one.
- DRM runs a selection algorithm (see next section) to find out which OS configuration should be run in the next rejuvenation.
- DRM fetches from the *configuration repository* the chosen OS configuration and stores it in the cache. This occurs in the background, while the server is processing the user requests.

An OS configuration basically contains the OS, plus other auxiliary programs, and a server. They are stored in a virtual machine disk (i.e., a file) that can be run by the virtualization solution. System administrators typically create these configurations, which should only contain fully patched software without any known vulnerabilities, and save them in a secure repository. The access to this repository is protected by employing a separate LAN (as represented in the figure) or by using cryptographic mechanisms to safeguard the communications.

III. Diverse rejuvenation

In this section we present a solution for the problem of selecting diverse configurations. To our knowledge, this problem has been mainly overlooked in the past, and it corresponds to the decision of which configuration should be run in a replica, given the already running configurations and a suitable set of candidate diverse configurations. To simplify our discussion and make it well grounded on the available results about diverse configurations, we describe our technique using diverse operating systems configurations, but it can be easily extended to deal with configurations composed by a stack of software components.

A. Rational for the solution

Intuitively, the selection algorithm should pick from the available alternatives the *best* OS configuration, in the sense that it should not have common vulnerabilities with the already running replicas. This would considerably delay the adversary to compromise more than f replicas[1]. This solution however suffers from one difficulty – given two fully patched configurations, one does not know if they share some vulnerability (which might be discovered in the future). Therefore, when designing the selection algorithm, we should attempt to fulfill the following prepositions:

- P1 The new selected OS configuration does not share vulnerabilities with the configurations already executing in the other replicas.
- P2 Given the group of configurations currently running, the adversary can not predict the configurations that will be selected in the future.
- P3 All diverse OS configurations available on the configuration repository[2] for selection are picked by the algorithm with a reasonable probability.
- P4 The algorithm is run individually by each DRM of the replicas.

As explained, P1 cannot be ensured with absolute certainty. However, we have found in a recent study about OS diversity [8] strong empirical evidence for: 1) it is possible to find OS pairs that have had no (or only a

[1]Notice that we are working under the assumption that finding and exploiting new vulnerabilities in mature software takes some time.

[2]Which is a subset of all available configurations containing the operating systems that match some performance or dependability criteria.

few) vulnerabilities in common in the past; and 2) if OS pairs share few vulnerabilities in the past, then with high probability no (or very few) common vulnerabilities are found in the future. This study was based on vulnerability data from the NVD database [9] over a period of 15 years, and it allowed us to collect information about vulnerabilities that are present in more than one OS version. For each OS version pair we can obtain the list of shared vulnerabilities and the CVSS score of each vulnerability[3]. There are studies that cross-validate our idea, also based on data from NVD (see [11], [12] for more details). Therefore, by combining this data we can calculate a rough criteria for deciding if two OS configurations share vulnerabilities: $score(OS_A, OS_B) = \sum_{v \in \mathcal{V}_{A,B}} CVSSscore_v$, where $v \in \mathcal{V}_{A,B}$ is the set of past common vulnerabilities of OS_A and OS_B, and $CVSSscore_v$ is the score of a vulnerability v.

Preposition P2 is necessary to address the following attack – to increase the available time to find vulnerabilities, the adversary predicts a system configuration that will be used some time from now (e.g., in a month); then, he or she starts to attack the corresponding OS versions, so that when this configuration is eventually installed, more than f replicas can be corrupted in a limited amount of time. Since we only have a limited number of OS configurations, our aim should be to make the prediction as hard as possible. This means that selecting an OS configuration from the available ones should entail some level of randomness, even if this implies choosing a system configuration that has a somewhat higher score among some of the executing replicas.

Some OS pairs share much less vulnerabilities than others, and therefore, there is the risk that some of the available OS configurations are never selected. To address this problem, the algorithm should enforce P3. The last preposition is useful because it simplifies the implementation, since this allows the DRM to determine which OS configurations are (and will be) used in replicas without having to communicate. This requires that the algorithm executes in a deterministic way (after some potential initial random setup step).

B. Selection algorithm

Algorithm 1 is run individually by each replica DRM, and it provides a solution to the diversity selection problem fulfilling the above four prepositions. When the system is initialized, every DRM receives an equal random *seed* value and a copy of table *OSTable* containing a description of the OS configurations stored at the repository. Among other things, this table has for each OS configuration pair the *score* of vulnerability (as discussed above). The system administrator also indicates in a *vulScore* configuration variable, what he or she considers as an acceptable maximal score value between any two OS configurations that are run in the system (sometimes the algorithm may need to select configurations higher this value if there are no alternatives).

[3]The Common Vulnerability Scoring System (CVSS) score provides an indication of the impact of a vulnerability in a system, and it takes into consideration aspects like ease of exploitation and the impact on the integrity/confidentiality/availability [10].

Algorithm 1: Diverse: A Diversity selector algorithm

```
   Initialization:
1  rejCount = −1;
2  OSConf = (∗,∗,∗,...∗);
3  initRandom(seed);

   initSelectCandidate():
4  firstC = (getRandom() mod size(OSTable));
5  nextC = 0;
6  score = vulScore;

   selectCandidate():
7  cand = getOSTable((firstC + nextC) mod size(OSTable));
8  if ((nextC > 0) and (nextC mod size(OSTable) = 0)) then
9  |  score = score + α;
10 end
11 nextC = nextC + 1;
12 return cand;

   findNextConfiguration():
13 rejCount = rejCount + 1;
14 i = rejCount mod n;
15 initSelectCandidate();
16 while (true) do
17 |  done = false;
18 |  j = 0;
19 |  cand = selectCandidate();
20 |  repeat
21 |  |  if (getScore(OSConf(j), cand) > score) then
22 |  |  |  done = true;
23 |  |  end
24 |  |  j = j + 1;
25 |  until ((¬done) and (j < n)) ;
26 |  if (¬done) then
27 |  |  OSConf(i) = cand;
28 |  |  return cand;
29 |  end
30 end
```

The algorithm starts by doing some global initializations (Lines 1-3). The number of rejuvenations *rejCount* is set to -1 to indicate the no rejuvenation has occurred, and the local random number generator is initialized with the global *seed*. *OSconf* contains the current OS configuration that is used at each replica (numbered between 0 and $n-1$), and it is started with some undefined value $*$.

As explained in Section II, in round-robin and in every T time units, one of the replicas is rejuvenated with a new OS configuration. Function *findNextConfiguration*() is called to determine which OS configuration should be used in that replica. It is also called n times during the system startup, to find out the initial configuration of each replica. The function begins by incrementing the rejuvenation count and by determining which replica will be rejuvenated (Lines 13 and 14). Then, it calls *initSelectCandidate*(), which randomly

finds the index on the *OSTable* of the first candidate OS configuration (*size*(*OSTable*) gives the number of elements of the table) and sets the score level *score* as *vulScore* (Lines 4-6). Next, function *findNextConfiguration*() enters in a loop, where it picks a new candidate OS configuration (Line 19), and then checks if this candidate has few shared vulnerabilities with the already running replicas (i.e., the score between any pair of the candidate and running replicas should be less than *score*) (Lines 16-28). This procedure prevents the selection of the same OS configuration that is currently running, if *OSTable* is setup in such a way that $getScore(OS_A, OS_A) = \infty$ (Line 21) for any configuration OS_A. In the first n executions $getScore(OSConf(j), cand)$ returns 0 because $OSConf(j) = *$, which causes *cand* to be the selected OS for replica j.

The reader should notice that the algorithm is designed in such a way that is possible always to find a new candidate OS configuration. First, it tries all available candidate configurations that are different from the ones currently running (Line 21 and function *selectCandidate*()) for a given score level *score*. If no configuration is found acceptable, then it increases the *score* by an α constant (Line 9) and the whole process is repeated.

IV. Conclusion and Future Work

The ideas outlined in previous sections comprise the current solution to solve a long lasting problem of diversity configuration on an intrusion tolerant system. Although we believe the configuration selector can solve the problem of changing the vulnerability set of a distributed system during its execution time, our approach still has a number of limitations that we are currently addressing:

- At startup we must have all virtual machine images with the different OSs already created. This complicates the deployment and update of system software. In particular, it may be difficult or costly to manage and apply patches on this large base of installed software.
- The *OSTable* construction is based on results from empirical studies such as [8], [13], which does not prove that the software does not have common vulnerabilities/bugs, but gives some evidence pointing in this direction. However, given the inherent complexity of these studies, defining such table for components not yet analyzed may be a complex and error-prone task.
- Defining *vulScore* and α is still an open problem highly dependent of results of the *OSTable*.
- We also want to explore other opportunistic diversity possibilities besides different OS configurations. For example, the case of memory layout randomization and code obfuscation to increase heterogeneity across the replicas using the same software stack and explore other taxonomies of diversity within the OS [14].

References

[1] P. Verissimo, N. F. Neves, and M. P. Correia, "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems*, ser. LNCS, 2003, vol. 2677.

[2] P. Sousa, N. F. Neves, and P. Verissimo, "How resilient are distributed *f* fault/intrusion-tolerant systems?" in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2005.

[3] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, 2002.

[4] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, 2010.

[5] Y. Huang and C. M. R. Kintala, "Software implemented fault tolerance: Technologies and experience," in *Proceedings of the IEEE International Symposium on Fault Tolerant Computing*, Jun. 1993.

[6] P. Sousa, A. N. Bessani, and R. R. Obelheiro, "The FOREVER service for fault/intrusion removal," in *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems*, Apr. 2008.

[7] T. Roeder and F. Schneider, "Proactive obfuscation," *ACM Transactions on Computer Systems*, vol. 28, Jul 2010.

[8] M. Garcia, A. Bessani, I. Gashi, N. F. Neves, and R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality?" in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun 2011.

[9] "National Vulnerability Database," http://nvd.nist.gov/.

[10] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system," *IEEE Security & Privacy*, vol. 4, no. 6, Nov–Dec 2006.

[11] O. H. Alhazmi and Y. K. Malayia, "Application of vulnerability discovery models to major operating systems," *IEEE Transactions on Reliability*, vol. 57, no. 1, Mar. 2008.

[12] G. Schryen, "Security of open source and closed source software: An empirical comparison of published vulnerabilities," in *Proceedings of the Americas Conference on Information System*, Aug. 2009.

[13] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, Oct.

[14] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia, "How practical are intrusion-tolerant distributed systems?" Department of Informatics, University of Lisbon, DI/FCUL TR 06–15, Sep 2006.