UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

Diversity Management in Intrusion Tolerant Systems

Miguel Garcia Tavares Henriques

MESTRADO EM INFORMÁTICA

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

Diversity Management in Intrusion Tolerant Systems

Miguel Garcia Tavares Henriques

DISSERTAÇÃO

Trabalho orientado pelo Prof. Doutor Alysson Neves Bessani
e co-orientado pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

MESTRADO EM INFORMÁTICA

2011

Acknowledgments

First, I would like to thank my advisors, Professor Alysson Bessani and Professor Nuno Neves. It was in my second year of graduation in a particular lecture from Professor Nuno that I got interested in security, and later on again during the Security course by Professor Alysson. At the end of that year, Professor Alysson invited me to join Navigators. Second, I would like to thank to Professor Paulo Sousa for advising me in my first two years as a junior researcher in Navigators. Without their support and guidance it would be very difficult to achieve what I have done so far. It is a honor, and also a responsibility, to work with researchers of such high-quality and I am in their debt for their teachings and for being my role models.

I would also like to thank the important collaboration of Professors Ilir Gashi and Rafael Obelheiro, – a great part of this thesis, due to their work and commitment, results from our joint work. I also thank Ilir for the confidence he passed to me at my first international conference presentation.

To all my colleagues I encountered during my graduation, MSc, and in the LaSIGE research group, specially to my friend and colleague Bruno for all the work we have done together in the graduation and master years, it was a war and we won! To Tiago and João, my informal advisors and friends, thank you for your wisdom, experience, advise and time spent with my insecurities.

I must thank my sister, brother and parents for all the patience and unconditional support. To my uncles and cousins for the motivation, inspiration and for making this journey possible. To Zé and Dulce for being my second family.

To all the people and “things” that made me who I am.

Resumo

Uma aplicação importante dos protocolos de tolerância a faltas arbitrárias (ou Bizantinas) é a construção de sistemas tolerantes a intrusões, que são capazes de funcionar correctamente mesmo que alguns dos seus componentes sejam comprometidos. Estes sistemas são concretizados através da replicação de componentes e da utilização de protocolos capazes de tolerar faltas arbitrárias, quer na rede como num subconjunto de réplicas. Os protocolos garantem um comportamento correcto ainda que exista uma minoria (normalmente menor do que um terço) de componentes controlados por um adversário malicioso. Para cumprir esta condição os componentes do sistema têm de apresentar independência nas falhas. No entanto, quando estamos no contexto da segurança de sistemas, temos de admitir a possibilidade de ocorrerem ataques simultâneos contra várias réplicas. Se os vários componentes tiverem as mesmas vulnerabilidades, então podem ser comprometidos com um só ataque, o que destrói o propósito de se construir sistemas tolerantes a intrusões. Com o objectivo de reduzir a probabilidade de existirem vulnerabilidades comuns pretendemos utilizar diversidade: cada componente usa software distinto que fornece as mesmas funcionalidades, com a expectativa de que as diferenças vão reduzir o número de vulnerabilidades semelhantes.

Reconhecemos também que isoladamente a tolerância a faltas arbitrárias tem algumas limitações uma vez que consideramos faltas maliciosas: uma das limitações mais importantes é que dado tempo suficiente o adversário pode comprometer $f + 1$ réplicas, e então violar a hipótese de que no máximo f componentes podem sofrer uma falta, levando os recursos do sistema à exaustão.

Uma forma de lidar com esta limitação consiste em renovar periodicamente as réplicas, uma técnica denominada por Recuperação Proactiva. O propósito das recuperações é limpar o estado do sistema, reiniciando a replica com código disponível em armazenamento apenas com permissões de leitura (ex: CD-ROM) e validar/obter o estado de outro componente (que seja correcto). Num sistema tolerante a intrusões com recuperação proactiva o intervalo temporal que o adversário tem para comprometer $f + 1$ réplicas passa a ser uma pequena *janela de vulnerabilidade*, que compreende o tempo de recuperação do sistema todo. Apesar dos benefícios que as recuperações periódicas oferecem em termos de fiabilidade persiste a seguinte dificuldade: as vulnerabilidades exploradas nas execuções anteriores da replica podem ainda ser exploradas depois da recuperação. Esta

limitação permite facilmente a um atacante criar um *script* que automaticamente compromete novamente a replica logo a seguir à sua recuperação - pois as vulnerabilidades não são apagadas mas sim a falta (i.e., o estado incorrecto no componente devido à intrusão).

Com o objectivo de melhorar o sistema introduzimos diversidade nas recuperações, mais precisamente em componentes *off-the-shelf (OTS)*. Hoje em dia praticamente todo o software desenvolvido é baseado neste tipo de componentes, como por exemplo sistemas operativos (SO) e gestores de bases de dados. Isto deve-se principalmente à complexidade do desenvolvimento destes componentes em conjugação com os benefícios relacionados com o baixo custo[1], a instalação rápida e a variedade de opções disponíveis. No entanto, a maior parte dos componentes OTS não foram desenhados com segurança como prioridade, o que significa que em todos eles existem vulnerabilidades que podem ser maliciosamente exploradas.

Por vezes, sistemas supostamente seguros são comprometidos através de uma componente critica na sua infraestrutura. Por outro lado, dada a quantidade de oferta das componentes OTS, utilizar diversidade nestes componentes é menos complexo e tem um menor custo do que desenvolver várias componentes de software diferentes. Um bom exemplo disto é o caso dos SO: as organizações na verdade preferem um sistema operativo OTS do que construir o seu próprio SO. Dada a variedade de sistemas operativos disponíveis e a criticidade do papel desempenhado por estes em qualquer computador, a diversidade ao nível dos SO pode ser uma forma razoável de garantir segurança contra vulnerabilidades comuns com um baixo custo adicional.

O foco nas vulnerabilidades comuns é um aspecto importante deste trabalho. Visto que a tolerância intrusões é aplicada em sistemas críticos, é seguro afirmar que no sistema operativo vai ser assegurada a máxima segurança, aplicando todos os *patches* disponíveis. No entanto, mesmo com sistemas actualizados, o sistema pode ser comprometido através de vulnerabilidades que ainda não foram descobertas pelos programadores (vulnerabilidades de dia zero), visto que os *patches* aparecem normalmente depois da vulnerabilidade ser anunciada. Se uma vulnerabilidade de dia zero afectar o sistema existe uma janela de oportunidade para o atacante causar uma intrusão.

A questão principal que tratamos na primeira parte desta tese é: *Quais são os ganhos de se aplicar diversidade de SO num sistema tolerante a intrusões replicado?* Para responder a esta questão, recolhemos e seleccionámos dados sobre vulnerabilidades do *NIST National Vulnerability Database (NVD)* entre 1994 e 2010 para 11 sistemas operativos. Os dados do NVD relativamente aos SO são consideráveis, o que nos permite tirar algumas conclusões. Cada vulnerabilidade presente no NVD contém (entre outras coisas) informação sobre que produtos são afectados pela vulnerabilidade. Recolhemos estes dados e verificámos quantas vulnerabilidades afectam mais do que um sistema operativo. Constatámos que este número é relativamente pequeno para a maior parte de pares

[1]Alguns destes componentes podem ser de código aberto e gratuitos.

de sistemas operativos. Este estudo foi depois estendido a um número maior de SO, com conclusões semelhantes para esses conjuntos. Estes resultados sugerem que existem ganhos de segurança que podem ser alcançados recorrendo à utilização de sistemas operativos diferentes num sistema replicado.

Como nota de cautela, não pretendemos afirmar que estes resultados são uma prova final sobre a ausência de vulnerabilidades comuns (embora sejam bastante promissores). Um dos principais problemas encontrados é que os relatórios focam-se nas vulnerabilidades e não em quantas intrusões ou *exploits* ocorreram para cada vulnerabilidade; isto faz com que a avaliação, em termos de segurança, seja mais difícil.

A segunda parte da tese propõe uma arquitectura que explora a diversidade disponível nos sistemas operativos juntamente com mecanismos de recuperação proactiva. O objectivo principal é mudar a configuração das réplicas de forma a alterar o conjunto de vulnerabilidades após uma recuperação. Desenvolvemos também um algoritmo que selecciona entre os candidatos o melhor sistema operativo para ser usado numa recuperação, assegurando o maior nível de diversidade possível entre as réplicas que se encontram em execução.

As contribuições principais deste trabalho podem ser descritas em:

1. Um estudo que consistiu na classificação manual das vulnerabilidades que afectam 11 sistemas operativos em categorias: *drivers*, *kernel*, *software* de sistema e aplicações; comparação entre pares de sistemas operativos, e construção de conjuntos de sistemas operativos para sistemas replicados com foco na diversidade.

2. Uma discussão sobre as limitações e oportunidades oferecidas pelos dados disponíveis no NVD para avaliar sistemas seguros e confiáveis;

3. Desenvolvimento de uma arquitectura que combina a diversidade descrita anteriormente com mecanismos de recuperação proactiva, baseada num algoritmo que oferece a melhor combinação de sistemas operativos para o sistema.

O trabalho descrito na tese resultou nas seguintes publicações:

- M. Garcia, A. Bessani, I. Gashi, N. Neves, R. Obelheiro, *"OS Diversity for Intrusion Tolerance: Myth or Reality?"*, in Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, Hong Kong, June 2011;

- M. Garcia, A. Bessani, N. Neves, *"Diverse OS Rejuvenation for Intrusion Tolerance"*, Poster paper in Supplement of the IEEE/IFIP International Conference on Dependable Systems and Networks, Hong Kong, June 2011.

Palavras-chave: Diversidade, Vulnerabilidades, Sistemas Operativos, Tolerância a Intrusões, Recuperção Proactiva.

Abstract

One of the key benefits of using intrusion-tolerant systems is the possibility of ensuring correct behavior in the presence of attacks and intrusions. These security gains are directly dependent on the components exhibiting failure diversity. To what extent failure diversity is observed in practical deployment depends on how diverse are the components that constitute the system. In this thesis we present a study with operating systems (OS) vulnerability reports from the NIST National Vulnerability Database. We have analyzed the vulnerabilities of 11 different OS over a period of roughly 15 years, to check how many of these vulnerabilities occur in more than one OS. We found this number to be low for several combinations of OS. Hence, our analysis provides a strong indication that building a system with diverse OS may be a useful technique to improve its intrusion tolerance capabilities. However, even with diversity the attacker eventually will find vulnerabilities in all OS replicas. To mitigate/eliminate this problem we introduce diverse proactive recovery on the replicas. Proactive recovery is a 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 malicious user, it allows the removal of intrusions from the compromised replicas. We propose that after each recovery a replica starts to run a different software. The selection of the new replica configuration is a non-trivial problem, as we will explain, since we would like to maximize the diversity of the system under the constraint of the available configurations.

Keywords: Diversity, Vulnerabilities, Operating Systems, Intrusion Tolerance, Proactive Recovery.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we motivate the work of the thesis, and we present the main goals and most important achievements. In the end of the chapter, we analyze the planning presented on the preliminary report and the actual task accomplishment, and we also describe the organization of the rest of the document.

1.1 Motivation

One important application of Byzantine fault-tolerant protocols is to build intrusion-tolerant systems, which are able to keep functioning correctly even if some of their parts are compromised. Such protocols guarantee correct behavior in spite of arbitrary faults provided that a minority (usually less than one third [30]) of the components are faulty (for an overview of the area see [63]). To respect this condition, system components need to exhibit failure diversity. However, when security is considered, the possibility of simultaneous attacks against several components cannot be dismissed. If multiple components exhibit the same vulnerabilities, they can be compromised by a single attack, which defeats the whole purpose of building an intrusion-tolerant system in the first place. To reduce the probability of common faults, *diversity* can be employed: each component uses different software to perform the same functions, with the expectation that the differences will reduce the occurrence of common vulnerabilities. This is an orthogonal aspect that affects all works on Byzantine fault-tolerant replication (e.g., [1, 8, 11, 13, 41, 56, 67]).

We also recognize in BFT replication alone some limitations once malicious faults are considered: One of the most important limitations is that given sufficient time, an adversary might be able to compromise $f + 1$ replicas and then break the assumption that at most f replicas are faulty, exhausting the resources of the system [60].

A way to deal with this limitation is to employ periodic rejuvenations of replicas [11, 58, 60], a technique commonly called proactive recovery (PR). The rationale of these rejuvenations is to clean the state of the system, which is performed typically in the following way: 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 [23] 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.

In order to address this problem, we introduce diversity on the rejuvenations, more precisely on the off-the-shelf (OTS) components that most software systems built today rely on, such as operating systems and database management systems. This common usage is mostly due to the sheer complexity of such components, coupled with benefits such as the perceived lower costs from their use (some of the components may be open-source and/or freely available), faster deployment and the multitude of available options. Most OTS software, however, have not been designed with security as their top priority, which means that they all have their share of security flaws that can be exploited. At times, supposedly secure systems are compromised not due to vulnerabilities in application software but in a more surreptitious manner, by compromising some other component in their software infrastructure (e.g., the operating system). On the other hand, given the ready availability of OTS software, leveraging OTS components to implement diversity is less complex and more cost-effective than actually developing variants of software. One of the prime examples is the operating system (OS): realistically, people will resort to an OTS operating system rather than build their own. Given the variety of operating systems available and the critical role played by the OS in any system, diversity at the OS level can be a reasonable way of providing good security against common vulnerabilities at little extra cost.

The focus on common vulnerabilities is an important distinctive of this thesis. Since intrusion tolerance is usually applied to critical systems, it is safe to assume that maximum care will be exercised in protecting system components, including applying all security patches available. However, even an up-to-date system can be compromised through an undisclosed vulnerability (using a zero-day exploit), since patches usually only appear *after* a vulnerability has been publicized. If such a vulnerability affects several components, there is a window of opportunity for compromising many or all of them at the same time.

The main question we address in this thesis is: *What are the gains of applying OS diversity on a replicated intrusion-tolerant system?* To answer this question, we have collected vulnerability data from the NIST National Vulnerability Database (NVD) [44] reported in the period between 1994 and 2010 for 11 operating systems. We focus our study on operating systems for several reasons: they offer a good opportunity for diversity, many intrusions exploit OS vulnerabilities, and the number of OS-related vulnerability reports in the NVD is sufficiently large to give meaningful results. Each vulnerability report in the NVD database contains (amongst other things) information about which products are affected by the vulnerability. We collected this data and checked how many vulnerabilities affect more than one operating system. We found this number to be relatively low for most of the operating systems pairs. This study was then extended to a greater number of OS, with similar conclusions for selected sets. These results suggest that security gains may be achieved if diverse operating systems are employed in replicated systems.

In the second part of this thesis we propose an architecture to exploit the opportunistic diversity available from OTS components 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 [58] with a *configuration selector* able to choose system configurations for recovering replicas preserving some expected fault independence between them.

1.2 Contributions and Publications

The main contributions of this thesis can be summarized as:

1. A hand-made classification of the vulnerabilities that affect 11 operating systems in drivers, kernel, system software and applications, over a period of approximately 15 years.

2. A study of how many common vulnerabilities appear for several pairs of operating systems divided in four families (BSD, Solaris, Linux and Windows) that capture different users preferences. That study shows that for several pairs of OS there was a very small number of common flaws over the considered period. This result gives evidence that with an appropriate selection of OS running on the replicas of an intrusion tolerant system, it is possible to avoid shared vulnerabilities. The practical implications of this conclusion can be significant because it demonstrates that there can be security gains by employing diversity.

3. Design an architecture that uses proactive recovery mechanisms, introducing diversity on rejuvenations. The diversity is generated with an algorithm that selects the configurations to minimize the number of common vulnerabilities across the running replicas.

From the described work resulted two papers:

- M. Garcia, A. Bessani, I. Gashi, N. Neves, R. Obelheiro, *"OS Diversity for Intrusion Tolerance: Myth or Reality?"*, in Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, Hong Kong, June 2011;
- M. Garcia, A. Bessani, N. Neves, *"Diverse OS Rejuvenation for Intrusion Tolerance"*, Poster paper in Supplement of the IEEE/IFIP International Conference on Dependable Systems and Networks, Hong Kong, June 2011.

1.3 Planning

In this section we made an analysis of the planning presented in the preliminary report and the actual task accomplishment. There was some deviations from the original planning due to the papers listed in the previous section, it is important to notice that, at the first planning was only supposed to write one paper, but we end up writing another paper to a poster submission in the same conference. There was also a delay on writing the dissertation. These delays total had a cost of two additional months of work.

1.4 Document Structure

In Chapter 2 we make an overview of several related works in different topics such as: the different approaches on diversity in computer systems, studies using vulnerability reports, proactive recovery and diverse proactive recovery. In Chapter 3 we present a study based on vulnerability reports, where we describe the data source and the data filtering, and then analyze the data in different ways. In the Chapter 4 we design an architecture for a replicated system using diverse rejuvenation. We also provide an algorithm to select the diverse components to use in the system based on the data analyzed in the previous chapter, and we also describe an evaluation of this algorithm. The thesis is concluded in the Chapter 5, where we make an overview of the work done, summarizing the overall contributions and the limitations.

Chapter 2

Related Work

This chapter presents the most relevant related work to the research area addressed in the thesis, which includes the following subjects: diversity applied to increase the dependability and security of the system, studies about vulnerability reports and proactive-reactive recovery mechanisms (including some Byzantine fault tolerant protocols).

2.1 Diversity Studies

Design diversity is a classical mechanism for fault tolerance, and it was first introduced in the 1970s [49]. N-version programming is a technique for creating diverse software components, which was also proposed in those early years [7]. The main idea behind this technique is to use *n* different implementations of the same component, programmed by *n* different teams, ideally using distinct languages and methodologies. The objective is to achieve fault tolerance, assuming that designs and implementations developed independently will exhibit failure diversity. The authors described a practical experiment, where they hired 30 programmers to develop a program in PL/1[1], from three different specifications: PDL, OBJ and English (natural language). Some of the results of the study were the following: of the 30 programmers only 18 delivered the work, and from the 18 programs seven were based on OBJ, five in PDL and six in English; a set of 100 valid input transactions was made to test the programs, and 11 of the 18 programs aborted due to these inputs. Therefore, as the authors referred, it is a difficult process to go from different specifications to different implementations, and consequently it is hard to create the *n* versions of the same program. They also claimed that using different programmers does not avoid that each person makes the same mistakes, causing the same flaws.

Liburd [31] discusses if whether or not N-version programming has enough dependability gains to warrant the time, additional human resources and costs. In his thesis, he developed a N-version program voter, SAVE, and he concluded that N-version programming added dependability to a system. Knight and Leveson found that the assumption of

[1]Programming Language One (PL/1) is an IBM programing language developed in 1960.

independence of failures in N-version programs failed statistically [27]. In this paper, they made a practical experiment with students to develop different software, resulting on 27 different programs. Each of these programs was then subjected to one million randomly-generated test cases. The 27 programs ranged in length from 327 to 1004 lines of code. They concluded, using a probabilistic model to evaluate fault independence, that the results indicated that the model had to be rejected with 99% confidence level. This paper was very controversial, even with the all reservations made on the conclusions, such that the results mainly apply to their experiment. In 1990 they published another paper [26], to respond to all criticism made on the previous work.

In the late 80's Joseph and Avizienis [25] proposed the usage of N-version programming to identify computer viruses. They recognized that Program Flow Monitors (PFM) have some limitations. A PFM is used to monitor the concurrent system-level error detection, while comparing the dynamic characteristics of the program behavior with the expected behavior. Then they proposed some extensions to circumvent the limitations. Their solution protects the program in run-time, enables the detection of physical derived errors and certain design faults. The PFM extension is also virus proof become its components are hardwired or ROM based. On the evaluation, as a potential alternative to formal verification, they proposed a 3 version C-language compiler. The compiler generates 3 program outputs, and it is expected that a virus would not infect all three versions simultaneously. Then all the 3 versions are executed concurrently. Periodically they run a consensus on the intermediate and final results to check for the local state, on the temporary output and on the actions that manipulate files. As long as the majority produce the correct result, the design faults on the other versions are detected and masked. This thesis work, however, does not focus so much on generating diversity to detect the presence of viruses but on using diverse components.

Later, Forrest and colleagues applied notions from biologic systems to computer security and argued that diversity is an important natural mechanism to reduce the effects of attacks [17, 22]. The authors suggested some mechanisms to provide diversity that are similar to N-version programing, like: add or delete non functional code, introduce some functions that are not used but that will make the binary different; re-ordering code, depending on the compiler this can be more simple or difficult to achieve; and memory layout, to change layout by randomization or padding techniques.

In [42], Nagy et al. made a study on N-version programing to detect zero day vulnerabilities. They implemented a simple auction site using three different replicated server programs running on a virtual machine environment: Debian Linux with Apache, PHP and MySQL; Solaris 9 with Tomcat, JSP and Postgres; and Windows 2000 with IIS, ASP and MS SQL. Each query is logged on the backend of each server. When a mismatch between the backend server outputs occurs, they mark the response assuming that something went wrong. The results from their work shows that N-version programing can be used to provide detection of zero day vulnerabilities but also to enhance the intrusion tolerance of the system.

Taxonomies of diversity techniques for improving security have been introduced in [15, 45]. Deswarte recognizes several types of diversity: at the level of users or operators, in the human-computer interfaces, at the application software level, at the execution level and at the hardware and operating system level. Obelheiro et al. presented two interesting definitions: *Axis of diversity*, in one system we can vary several components such as applications, operating system, hardware, OTS software, etc; and *Degree of diversity*, the number of choices available for a specific axis of diversity. They stated the important role of the OS in any system and also the cost of using OS diversity due to different setups and deployment. In this thesis, we take a particular focus on OS diversity, and make an empirical evaluation to support the vulnerability independence claim. This sort of evaluation is mostly lacking in the previous studies.

An experimental study on the benefits of adopting diversity of SQL database servers is presented by Gashi et al. in [21]. The authors analyzed bug reports for four database servers and verified which products were affected by each reported bug (the focus of their study was on overall dependability, not specifically on security). 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. Their conclusion is that diversity of off-the-shelf database servers is an effective mean of improving system reliability. They found some limitations on the work, called by the authors as "dialect-specific" – for example, if the bug that is being explored relies on a specific operand/semantic of the SQL language that one database manager brand offers, it is impossible to verify those bugs in other providers that do not implement/offer this operand. Also, due to this same reason, it is more difficult to employ application diversity, because of the semantic and operational differences. This results in a replicated solution that limits the services available to the lowest common denominator. Some of the limitations of our data set (see Section 5.2.1) prevent us from making the same type of study with NVD data.

Given the criticality of operating systems, there are many papers that study the distribution of bugs and vulnerabilities in OS code. Miller et al. [37, 38] analyzed how commands and services in different UNIX variants dealt with random input and found out that between 25% and 50% of them (depending on the study) would crash or hang. Chou et al. [12] used compiler extensions to perform static analysis of the Linux and OpenBSD kernels. Their study shows that device drivers exhibit more flaws than the rest of the kernel, and that some types of bugs in the Linux kernel take an average of 1.8 years before being fixed.

Ozment and Schechter [48] studied how OpenBSD security evolved over time, using data from OpenBSD security advisories and the project's source code repository. They conclude that many vulnerabilities are still found in legacy code, that bugs in security-related code are more likely to cause vulnerabilities, and that the rate of vulnerability reports for OpenBSD is decreasing over time. There are also some interesting numbers in the paper, such that after 7 years 61% of the lines of code are still the same between version 2.3 and 3.7 (14 versions released between them). We discuss these numbers in Section 3.2.1.

Anbalagan and Vouk [5] analyzed vulnerabilities in Bugzilla and Fedora Linux and found out that 34% of the vulnerabilities are exploited before being disclosed. The authors distinguish two vulnerability types, one called “voluntary exploited” and the other “unvoluntary exploited”. On average they claim that 24% of the vulnerabilities are from “voluntary” type. They also pointed, from a universe of 43710 vulnerabilities (retrieved from OSVDB[2]), that only 3.4% of the voluntary exploits are immediately disclosed after being discovered, 0.1% are exploited and disclosed after a certain period, and 0.2% are exploited after disclosure but before the fix patch. Although none of these papers attempted to analyze the occurrence of common vulnerabilities across different OS, the numbers are interesting and complement our conclusions.

A comparison of the robustness of 15 different POSIX-based operating systems is presented in [28]. This study was based on fault injection: combinations of valid and invalid parameters were supplied to often-used system calls and C library functions, and the effects of this on reliability (e.g., system crash, process hang/crash, wrong or no error code returned) was observed. The authors found out some commonalities among the studied systems, especially with respect to the common mode failures of C library functions. However, from the available data it is impossible to conclude whether there were specific bugs that affected more than one system (the work only shows how many failures were observed for each system call in several degrees of severity). Still, their evidence indicates that, from a reliability standpoint, using different operating systems reduces the number of common failure modes.

2.2 Studies Using Vulnerability Reports

Some vulnerability discovery models, which attempted to forecast the amount of vulnerabilities found in software, have been proposed [3, 6, 52]. Alhazmi and Malayia [4] investigate how well these models fit with vulnerability data from the NVD, and conclude that the vulnerability discovery process follows the same S-shaped curve of “traditional” software reliability growth models[3] [35]. This conclusion is disputed in [55], where it is claimed that the number of vulnerabilities disclosed in the NVD grows linearly with time (this contrast might be due to methodological differences). These studies cross-validate our idea of using the NVD as a source of vulnerability data. However, they are

[2]http://osvdb.org/

[3]These models measure all the defects found in a system, and not only those affecting security.

more concerned in modeling how many vulnerabilities are found in specific software over its lifetime [4] and if there are significant differences between open- and closed-source software [55]. Our focus is on assessing the degree of independence between different operating systems.

Littlewood and colleagues [33] survey a number of issues in software diversity modeling, presenting models for assessing the reliability of systems that adopt diversity. The discussed models aim to provide a measure of the reliability of a system as a function of the demands presented to the system and how these demands influence the correctness of the behavior of the system. These parameters are, for the most part, expressed as probability distributions. Some of these ideas have later been extended to the security domain as well [34]. They show that, although diversity does not provide complete failure independence (since design faults are correlated to some extent), it is an effective means of increasing overall system reliability. They also discuss a number of caveats regarding software diversity modeling. It would be desirable to use these models in our context, but this is currently unfeasible, since we lack sufficiently detailed data (operational profiles and vulnerability exploitation rates) to apply them.

2.3 Proactive Recovery

Software rejuvenation was proposed in the 90's by Huang et al [23, 24]. 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. Huang and colleagues developed NT-Swift [23], a software component that can be attached to Windows NT to improve the dependability capabilities. NT-Swift has a component that can be installed on clients and servers. If NT-Swift detects a failure it restarts the application or even, if necessary, restarts the machine This approach has a limitation, the downtime of the service.

Castro and Liskov [10] presented PBFT, an algorithm for state machine replication that offers safety if $\lfloor (n-1)/3 \rfloor$ out of a total of n replicas are faulty. The service may be unable to reply during a denial-of-service attack, but it guarantees that the client will receive replies when the attack ends. PBFT uses symmetric cryptography to reduce the time consumption, and it needs only one message round trip to execute read-only operations, and two message rounds to execute read-write operations. PBFT also referred to diversity as an enhancement of the system, although the authors do not present results to testify that. Later on, the same authors presented BFT-PR, a new algorithm that enhances the availability of the system by employing Proactive Recovery (PR) [11]. Additional assumptions require trusted physical components, for hardware cryptography and watchdogs timers, such that they are impossible to compromise. To ensure that no more than one replica recovers at the same time, the authors define a formula: $T_v = 2T_k + T_r$, in which T_v is the window of vulnerability (the amount of time that an adversary has to compromise more than f replicas), T_k is the maximum key refreshment period and T_r is the maximum time between a replica fails and recovers. To evaluate this algorithm, they developed a BFT version of NFS implemented in Linux with Ext2fs. They present some interesting results, with the Andrew100 and Andrew500 benchmarks[4], showing that a recovery takes 42.59 and 143.68 seconds respectively.

Sousa et al. proposed a new approach to intrusion tolerant systems [58] that periodically rejuvenates the replicas, to remove the effects of malicious attacks/faults. The basic idea is to perform rejuvenations sufficiently often, in order to make the attackers unable to compromise enough replicas to bring the whole system down. The system fails only if $f + 1$ replicas are compromised between rejuvenations. One of the contributions of this work is the *proactive-reactive recovery*. If one replica is faulty it can disturb the behavior of the other *n-1* replicas, and there is nothing that a correct replica can do to avoid this. With proactive-reactive recovery the rejuvenation process can be accelerated by detecting the faulty replicas and forcing them to recover, without sacrificing periodic rejuvenations. The technique can only be implemented with some synchrony [60], due to the recovery trigger clocks. To overcome this limitation the authors proposed an hybrid system model: the *payload* is a *any-synchrony* subsystem, and the *wormhole* is a *synchronous* subsystem.

In this work, Sousa and colleagues also made an experiment with a CIS (CRUTIAL Information Switch). The CIS is a distributed firewall [9] in which at most f replicas can suffer *Byzantine failures* in a given recovery period, and also at most *k* replicas can recover at same time. In this experiment, each *wormhole* subsystem is connected through a dedicated and secure control channel to the *payload* subsystem. A wormhole has a high precision clock to synchronize the *payload* recoveries. There is also a point-to-point timed reliable channel connecting to the other wormholes. The authors named this architecture as Proactive-Reactive Recovery Wormhole. It offers a service that can be called by the payload whenever there is a suspicion (or detection) of incorrect behavior by the other replicas. The interface to this service is through the two functions: *W_suspect(j)* for crash suspicions, since it is impossible to know if a replica really crashed or if is only slow; and *W_detect(j)* if the *BFT protocols* running on the payload replicas detected incorrect messages from some replica. If $f + 1$ replicas detected j as faulty the recovery of j occurs immediately. If $f + 1$ replicas suspect j as faulty, the recovery must be coordinated with periodic recoveries to guarantee a minimum of replicas to ensure system availability. The quorum of $f + 1$ is needed, in terms of suspicions or detections, to avoid recoveries triggered by faulty replicas. In order to schedule recoveries without harming the availability of the whole system, Sousa et al. designed an algorithm that runs in the wormhole part [58]. The algorithm is based on temporal slots that are allocated based on two variables, T_P the maximum time interval between consecutive recoveries, and T_D

[4]http://www.usenix.org/event/usenix01/full_papers/kroeger/kroeger_html/node12.html

defines the worst case scenario of execution of a recovery.

The authors made an experiment to evaluate the CIS where they used 1.7 GB Fedora OS images and the Xen virtualization solution [66]. They setup the system as $n = 4$, $f = 1$ and $k = 1$, and run three experiments: *Recovery Performance*, the maximum time to complete a recovery was 146 seconds; *Latency and throughput under a DoS attack from the WAN*, was found that there is no throughput loss with a reasonably loaded network; *Throughput under a DoS attack from a compromised replica*, wich shows that proactive-reactive recovery provides a much better solution than proactive recovery alone.

2.4 Diverse Proactive Recovery

Sousa et al. stated the need for diversity in time (i.e., changing replicas on recoveries) [59]. Their 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.

Rodrigues et al. [53] proposed BASE, a PBFT protocol extension, that uses off-the-shelf service components. This allows the utilization of different implementations to provide the same service, with the expectation of reducing the probability of common failures. BASE defines an abstract state to be shared among the replicas. The authors claim that with abstraction is possible to hide the distinct implementations of the services, exploiting opportunistic N-version programing with off-the-shelf software. This also enhances the PBFT because, in the original version, it cannot tolerate deterministic software errors that make all the replicas fail with the same input. The authors made an experiment with a file-system storage, similar to NFS, using off-the-shelf products. The replicated file-system uses proactive recovery, and in each recovery a replica requests the abstract state from the correct replicas and converts the abstract state to a concrete state. Due to the abstraction functions, each replica can run distinct software since it provides the same service. In the experiment diversity was employed: *n* clients and one server replica runs Linux, the other server replicas run OpenBSD 2.8, Solaris 8 and FreeBSD 4.0. Each OS has its own NFS implementation. There is one point that we must highlight in the experiment, the authors were optimistic and had an imprecise approach for estimating the reboot times as 30 seconds (during proactive recovery), which is a very low number for an OS recovery. A recovery was started every 80 seconds in a round-robin discipline, and in the best case it took 6 minutes with Andrew100, and in the worst case, it took 17 minutes with Andrew500 benchmark. This means that the system will work correctly as long as less than $\frac{1}{3}$ of the replicas fail in 6 or 17 minutes - the vulnerability window. They also present a comparison between heterogeneous replicas and the measuring of the computation elapsed time: OpenBSD takes 1599.1 seconds, Solaris 1009.2 seconds, FreeBSD 848.4 seconds and Linux 338.3 seconds. In our work we are not interested so much in the values but in the relative differences between them.

Distler et al. [16, 51] identified several issues that make virtualization useful for proactive recovery, such as the isolation between application domains and the privileged system domains, which allows the creation of 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. Additionally, it can be used to efficiently make state transfer between replicas, and could be employed to simplify the usage of diversity on replicas. In those two articles the authors describe the VM-FIT prototype, which is a replicated system that uses virtual machines to make an hybrid model with proactive recovery. In the second work they introduced diversity on replicas. The authors used the Xen virtualization solution, and on *Dom0* (privileged) they run Ubuntu and on *DomU* used Debian, NetBSD and OpenSolaris, based on the intuition that diversity could improve security. VM-FIT achieved some interesting results, although it could not assure continuous availability because the service had to be suspended for 3 seconds in each recovery.

In a recent work, Roeder and Scheinder [54] propose the use proactive obfuscation, whereby each replica is periodically restarted using a clean generated diverse binary. The authors implemented two prototypes: 1) a distributed firewall based on pf[5] (packet filter) in OpenBSD; and 2) a distributed storage service. Proactive obfuscation employs semantics-preserving program transformations, and it can be used in: reordering and stack padding, system call reordering, instruction set randomization, heap and data randomization (e.g., a buffer overflow attack depends on stack layout, and therefore using entropy on the stack will crash the program and not allowing the attacker to take control). Although it is not clear how they perform the obfuscations, because it is a non-trivial problem, there are several requirements that have to be satisfied such as the semantic must be the same in every obfuscated replica. It is also hard to measure the resulting fault independence and how can one estimate the number of different obfuscated versions that can be generated. In any case, 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 operating systems). This work does not evaluate possible options and neither considers changing OTS components 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.

2.5 Final Remarks

This section makes an overview of some of the most relevant research in different areas related to our work. Some of the research has direct contributions to the thesis, but there is no previous work that evaluated the OS diversity based on vulnerability reports.

[5]http://www.openbsd.org/faq/pf/filter.html

Chapter 3

Vulnerability Study

This chapter explains the methodology adopted in our study, with a particular focus on how the data set (i.e., the operating system (OS) vulnerabilities) was selected and on how the data was filtered and processed. The chapter includes an analysis on several aspects of the data, such as: vulnerability distribution by OS part, temporal distribution of vulnerabilities, comparison between pairs of OS, selection of sets from 4 different OS, and a comparison between pairs of OS releases. In the end of the chapter we summarize the results. Parts of this chapter appeared in [19].

3.1 Data Source

The OS vulnerability data was obtained from a public database, namely from the National Vulnerability Database (NVD) [44]. NVD uses the *Common Vulnerability Enumeration (CVE)* definition of vulnerability [14], which is presented below.

Definition 1 (CVE Vulnerability) *An information security "vulnerability" is a mistake in software that can be directly used by a hacker to gain access to a system or network.*

CVE considers a mistake a vulnerability if it allows an attacker to use it to violate a reasonable security policy for that system (this excludes entirely "open" security policies in which all users are trusted, or where there is no consideration of risk to the system).

For CVE, a vulnerability is a state in a computing system (or set of systems) that either:

- *allows an attacker to execute commands as another user;*
- *allows an attacker to access data that is contrary to the specified access restrictions for that data*
- *allows an attacker to pose as another entity*
- *allows an attacker to conduct a denial of service*

NVD aggregates vulnerability reports from more than 70 security companies, forums, advisory groups and organizations[1], being thus the most complete vulnerability database on the web. All data is made available as XML files containing the reported vulnerabilities on a given period, called *data feeds*. We analyze feeds from 2002 to 2010. The 2002 feed includes information about vulnerabilities that were reported between 1994 and 2002[2].

Each NVD data feed contains a list of reported vulnerabilities sorted by its date of publication on a given period. For each vulnerability, called *entry* in the NVD parlance, interesting information is provided such as an unique name for the entry, in the format CVE-*YEAR-NUMBER* (line 1 of Listing 3.1); the list of products (with version numbers) affected by the vulnerability (lines 3-6 of Listing 3.1); the date of the vulnerability publication (line 10 of Listing 3.1); the CVSS score[3] (see Section 4.2.1), that is calculated[4] based on the security attributes; the security attribute(s) that are affected when the vulnerability is exploited on a system (lines 17-23 and line 26 of Listing 3.1) and the description of the vulnerability (lines 29 of Listing 3.1). An entry contains other fields, which were not represented in Listing 3.1 to simplify the presentation.

We developed a Java program that collects, parses the XML data feeds and inserts the processed data into an SQL database, deployed with a custom schema to do the aggregation of vulnerabilities by affected products and versions.

Data selection

Despite the large amount of information about each vulnerability available in NVD, for the purposes of this study, we are only interested in the name, publication date, CVSS score, summary (description), type of exploit (local or remote) and the list of affected configurations. We have collected vulnerabilities reported for 64 Common Platform Enumerations (CPEs) [39]. Each one of these describes a system, i.e., a stack of software/hardware components in which the vulnerability may be exploited. These CPEs were filtered, resulting in the following information that was stored in our database:

- **Part:** NVD separates this in Hardware, Operating System and Application. For the purpose of this study we choose only enumerations marked as Operating System;
- **Product:** The product name of the platform;
- **Vendor:** Name of the supplier or vendor of the product platform.

[1]See the complete list at `http://cve.mitre.org/compatible/alerts_announcements.html`.

[2]This chapter performs an analysis on the data feed that contained vulnerabilities until September 30th 2010. For the next chapter, this database had to be updated with the feeds until February 2011, in order to validate the algorithm described in Section 4.2.2 with the most recent OS releases.

[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 [36].

[4]See the equation here: http://www.first.org/cvss/cvss-guide.html

Listing 3.1: NVD XML feed entry example

```
<entry id="CVE-2005-0004">
  <vuln:vulnerable-configuration id="http://nvd.nist.gov">
    <cpe-lang:fact-ref name="cpe:/o:debian:debian_linux:3.0::alpha" />
    <cpe-lang:fact-ref name="cpe:/o:redhat:fedora_core:core_1.0" />
    <cpe-lang:fact-ref name="cpe:/o:redhat:linux:7.3::i386" />
    <cpe-lang:fact-ref name="cpe:/o:redhat:linux:9.0::i386" />
  </vuln:vulnerable-configuration>
  <vuln:cve-id>CVE-2005-0004</vuln:cve-id>
  <vuln:published-datetime>
  2005-04-14T00:00:00.000-04:00
 </vuln:published-datetime>
  <vuln:last-modified-datetime>
  2008-09-10T15:34:44.570-04:00
 </vuln:last-modified-datetime>
  <vuln:cvss>
    <cvss:score>4.6</cvss:score>
    <cvss:access-vector>LOCAL</cvss:access-vector>
    <cvss:access-complexity>LOW</cvss:access-complexity>
    <cvss:authentication>NONE</cvss:authentication>
    <cvss:confidentiality-impact>PARTIAL</cvss:confidentiality-impact>
    <cvss:integrity-impact>PARTIAL</cvss:integrity-impact>
    <cvss:availability-impact>PARTIAL</cvss:availability-impact>
    <cvss:source>http://nvd.nist.gov</cvss:source>
  </vuln:cvss>
 <vuln:security-protection>
  ALLOWS_USER_ACCESS
 </vuln:security-protection>
 <vuln:summary>
  The mysqlaccess script in MySQL 4.0.23 and earlier [...] attack on
     temporary files.
 </vuln:summary>
</entry>
```

Those 64 CPEs were, by manual analysis[5], clustered in 11 OS distributions: *OpenBSD, NetBSD, FreeBSD, OpenSolaris, Solaris, Debian, Ubuntu, RedHat*[6], *Windows 2000, Windows 2003* and *Windows 2008*. These distributions cover the mostly used *server OS products* of the families: BSD, Solaris, Linux and Windows.

[5]To do the analysis we developed a *php* program that query the database, the response is a form that provide the information needed to classify the vulnerability. Then the program updates the vulnerability state.

[6]*RedHat* comprises the “old” Red Hat Linux (discontinued in 2003) and the more recent Red Hat Enterprise Linux (RHEL).

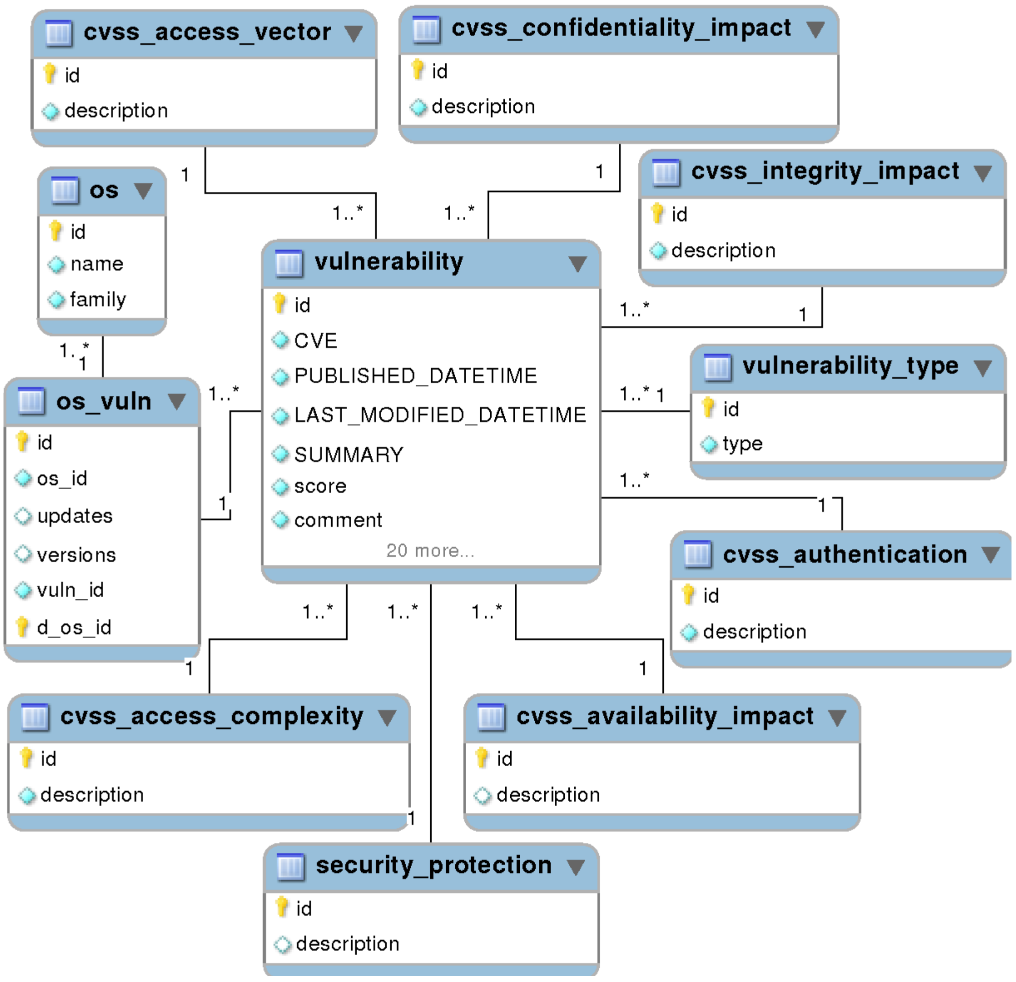


Figure 3.1: Simplified SQL schema of the database used to store and analyze the NVD data.

The schema of the resulting database is displayed in Figure 3.1. The tables with prefix *cvss*[7], *vulnerability_type* and *security_protection* are employed to optimize the database. The most important tables are:

- *cvss_access_vector*: stores how the vulnerability is exploited: Local (local access), Adjacent Network (domain access) and Network (the attacker does not need to be on the local network or local access to exploit the vulnerability);
- *cvss_access_complexity*: stores the complexity required to exploit the vulnerability, divided in three basic values: High, Medium and Low.
- *cvss_authentication*: stores the number of times that the attacker must authenticate to exploit the vulnerability: Multiple, Single and None;

[7]http://www.first.org/cvss/cvss-guide.html

- *cvss_confidentiality_impact*: stores the description impact on confidentiality: None, Partial and Complete. Confidentiality refers to limiting information access and disclosure to only authorized users;
- *cvss_integrity_impact*: stores the description impact on integrity: None, Partial and Complete. Integrity refers to the trustworthiness and guaranteed veracity of information;
- *cvss_availability_impact*: stores the description impact on availability: None, Partial and Complete. Availability refers to the accessibility of information resources;
- *security_protection*: stores what kind of access the attacker gains after the exploit: Admin access, User access and Other access;
- *vulnerability*: stores some information about a vulnerability (name, publication date, etc.);
- *vulnerability_type*: stores the vulnerability type assigned by us (see Section 3.1.2);
- *os*: stores the operating systems platforms of interest in this study;
- *os_vuln*: stores the relationship between vulnerabilities and operating systems, and their affected versions.

The use of an SQL database brings at least three benefits when compared with analyzing the data directly from the XML feeds. First, it allows us to *enrich the data set* by hand, for example, by assigning to each vulnerability information regarding its type (see Section 3.1.2), and also by associating release times and family names to each affected OS distribution. Second, it allows us to modify the CVE fields to correct problems. For example, one of the problems with NVD is that the same product is registered with distinct names for different entries. For example, (“$debian\_linux$”, “$debian$”) and (“$linux$”, “$debian$”) are two (product,vendor) pairs we have found for the Debian Linux distribution. This same problem was observed previously by other users of NVD data feeds [46]. Finally, an SQL database is much more convenient to work with than parsing the feeds on demand.

3.1.1 Filtering the Data

From the more than 44000 vulnerabilities published by NVD at the time of the study, we selected 2120 vulnerabilities. These vulnerabilities are the ones classified as OS-level vulnerabilities (“/o” on its CPE) for the operating systems under consideration.

When manually inspecting the data set, we discovered and removed vulnerabilities that contained tags in their descriptions such as *Unknown* and *Unspecified.* These correspond to vulnerabilities for which NVD does not know exactly where they occur or why they exist (however, they are usually included in the NVD database because they were mentioned in some patch released by a vendor). We also found few vulnerabilities flagged as ***DISPUTED***, meaning that product vendors disagree with the vulnerability existence. Due to the uncertainty that surrounds these vulnerabilities, we decided to exclude them from the study. Table 3.1 shows the distribution of these vulnerabilities on the analyzed OS, together with the total number of valid vulnerabilities.

OS	**Valid**	**Unknown**	**Unspecified**	**Disputed**
OpenBSD	142	1	1	1
NetBSD	126	0	1	2
FreeBSD	258	0	0	2
OpenSolaris	31	0	40	0
Solaris	400	39	109	0
Debian	201	3	1	0
Ubuntu	87	2	1	0
RedHat	369	12	8	1
Win2000	481	7	27	5
Win2003	343	4	30	3
Win2008	118	0	3	0
# distinct vuln.	1887	60	165	8

Table 3.1: Distribution of OS vulnerabilities in NVD.

An important observation about Table 3.1 is that the columns do not add up to the number of distinct vulnerabilities (last row of the table) because some vulnerabilities are shared among OS. Notice that about 60% of the removed vulnerabilities affected Solaris and OpenSolaris. Moreover, these two systems are the only ones that have more than 10% of its vulnerabilities removed: Solaris has 71 vulnerabilities and 56.3% of them are "invalid"; OpenSolaris has 548 vulnerabilities and 27.7% are "invalid"; OpenBSD, NetBSD, FreeBSD and Debian have only 2.1%, 2.3%, 0.8% and 2.0% respectively of invalid vulnerabilities. This result for the last set of OS might point to a better community support. We should remark that this manual filtering was necessary to increase the confidence that only valid vulnerabilities were used in the study.

3.1.2 Distribution of Vulnerabilities by OS Parts

For NVD, an operating system is not only the kernel, but the complete product that is distributed for installation. Therefore an *operating system product* is composed by the kernel, several drivers, optional modules, system software and applications. So, besides knowing how many vulnerabilities affect different operating system products, it is also important to understand what part or module of these systems is compromised by the vulnerability. Since NVD does not provide any information other than the vulnerability description, we inspected manually each of the 1887 entries and classified them in one of four categories: *Driver, Kernel, System Software* and *Application*. The rationale for this classification is the following:

- **Kernel:** vulnerabilities that affect the TCP/IP stack and other network protocols whose implementation is OS-dependent, file systems, process and task management, core libraries and vulnerabilities derived from processors architectures;
- **Driver:** vulnerabilities that affect drivers for wireless/wired network cards, video/-graphic cards, web cams, audio cards, Universal Plug and Play devices, etc;
- **System Software:** vulnerabilities that affect the majority of the software that is necessary to provide common operating system functionalities such as login, shells and basic daemons. We account just for software that comes by default with the distribution (although sometimes it is possible to uninstall these components without affecting the main OS operation);
- **Application:** vulnerabilities in software products that come with the operating system but that are not needed for basic operations, and in some cases require specific installation: database management systems, messenger clients, text editors and processors, web/email/FTP clients and servers, music/video players, programming languages (compilers and virtual machines), antivirus, Kerberos/LDAP software, games, etc.

The classification above facilitates the analyses of which parts of the operating systems may suffer most from common vulnerabilities, which would influence the architectural decisions of how one designs a diverse system.

3.2 OS Diversity Evolution

This section presents the results of the study. In particular, it presents an overall analysis of the counts of vulnerabilities for each OS component class, and shows how many vulnerabilities affect each OS pair. The section also provides empirical evidence to demonstrate that there are security gains in using diverse OS when deploying an intrusion-tolerant system.

3.2.1 Distribution of OS Vulnerabilities

This section presents two vulnerabilities distributions, a distribution based on the classification described in the last section, and a temporal distribution for the four OS families.

Vulnerability classification

The descriptions of 1887 vulnerabilities were examined, and then they were assigned to one of the OS component classes presented in the previous section. Table 3.2 summarizes the result of this analysis.

OS	**Driver**	**Kernel**	**Sys. Soft.**	**App.**	**Total**
OpenBSD	2	*75*	33	32	142
NetBSD	9	*59*	32	26	126
FreeBSD	4	*147*	54	53	258
OpenSolaris	0	*15*	9	7	31
Solaris	2	*156*	114	128	400
Debian	1	24	34	*142*	201
Ubuntu	2	22	8	*55*	87
RedHat	5	89	93	*182*	369
Windows 2000	3	143	132	*203*	481
Windows 2003	1	95	71	*176*	343
Windows 2008	0	42	14	*62*	118
% Total	1.4%	35.5%	23.2%	39.9%	

Table 3.2: Vulnerabilities per OS component class.

The table shows that with the exception of *Drivers*, all OS distributions have a reasonable number of vulnerabilities in each class. In the BSD and Solaris OS families, vulnerabilities appear in higher numbers in the *Kernel* part, while in the Linux and Windows families, the *Applications* vulnerabilities are more prevalent. This can be explained by noticing that Windows and Linux distributions usually contain a larger set of pre-installed applications, when compared to more stripped down products like BSD family OS. Therefore, there is a tendency to include more applications in platforms based on these OS, causing more vulnerabilities of this type to appear in the statistics.

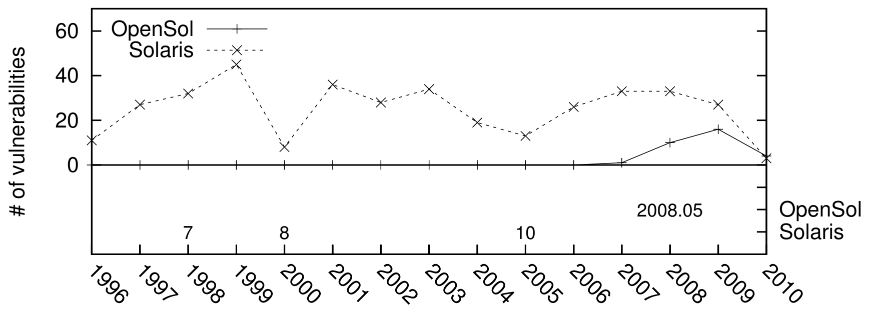
The last row of the table presents the percentage of each class on the total data set. One can observe that most vulnerabilities occur in the Application and Kernel components, which is then followed by the System Software group of utility programs. It is interesting to notice that Drivers account for a very small percentage of the published OS vulnerabilities. This observation seems to contradict previous studies showing that drivers are the main contributor of crashes [18], and it is somewhat surprising given that drivers usually account for a large percentage of the OS code [12]. One, however, should keep in mind that crash-inducing bugs do not necessarily translate into vulnerabilities, since they might not be exploitable by an adversary (e.g., because the conditions to activate the fault might be extremely hard to force). On the other hand, large and complex codes are a typical breeding ground for programming flaws, and we may experience a rise in driver vulnerabilities in the future.

Temporal distribution of the vulnerabilities

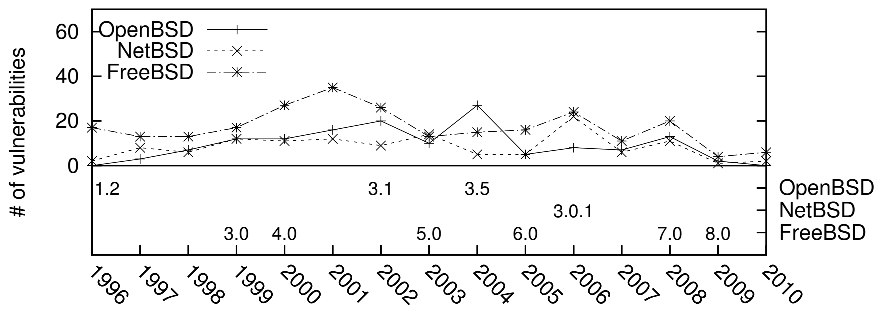
Figure 3.2 presents the number of vulnerabilities announced per OS for each year, while organizing in separate graphs the OS families. The figure also includes the dates of some of the major releases of the OS. Certain OS like Windows 2008 and Ubuntu have several years with zero vulnerabilities because their first distribution is relatively recent.

The graphs lead to some interesting observations. First, it is possible to notice a strong correlation among the peaks and valleys of both the Windows and Linux families, and somewhat to a lesser extent in the BSD family. This could mean that some vulnerabilities might be shared across the family members (see next section for a better discussion). Second, some OS families have less vulnerabilities being reported in the recent past (last 5 years) when compared with the more distant past. This is true both for the BSD and Linux families, which could indicate that the systems are becoming more stable, but also that the employed development process imposes stronger requirements on the quality of the software.

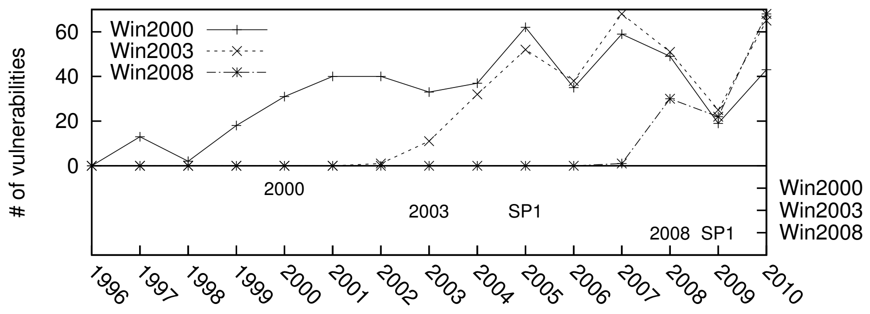
Finally, it is also important to compare the vulnerability dates and the year of the first OS release. NVD classifies vulnerabilities when they are first discovered, and then lists the OS that might be compromised by their exploitation. Therefore, it was possible to find Windows 2000 in seven entries earlier than 1999, sharing vulnerabilities with Windows NT. This confirms that Windows 2000 was built with some of the code of Windows NT, but apparently it seems that this code was not fixed from all already known vulnerabilities. We found three cases in other OS versions where a vulnerability was reported much earlier than the corresponding release. After examining the NVD entry, we were able to exclude them as errors in the database, and therefore, they are not shown in the graphs.



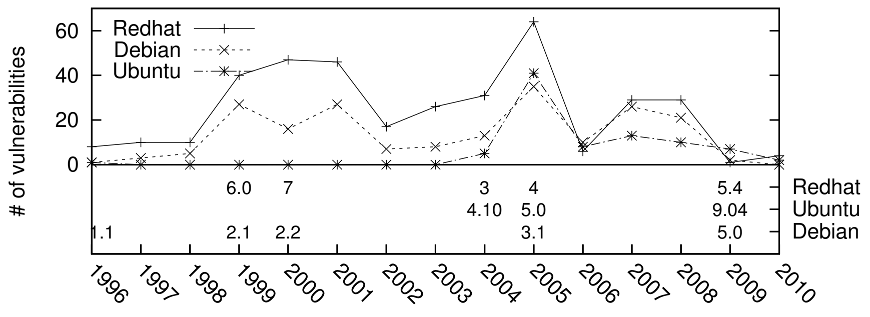
(a) Solaris family.



(b) BSD family.



(c) Windows server family.



(d) Linux family.

Figure 3.2: Temporal distribution of vulnerability publication data for four operating system families.

3.2.2 Common Vulnerabilities

Table 3.3 shows the common vulnerabilities that were found in every combination of OS pairs over the period of 1994 to (Sept) 2010. The columns with *v(A)* and *v(B)* show the total number of vulnerabilities collected for OS A and B respectively, whereas *v(AB)* is the count of vulnerabilities that affect both the A and B systems. Three filters were applied to the data set: *All* corresponds to all vulnerabilities, representing the raw data; *No Applications* removes from the data set the vulnerabilities classified as *Applications* (see Section 3.1.2); *No Applications and No Local:* same as the previous filter but only considers remotely exploitable vulnerabilities (vulnerabilities with "Network" or "Adjacent Network" values in their CVSS_ACCESS_VECTOR field). The aim of the first filter is to characterize a platform with a reasonable number of installed applications (called a *Fat Server*). The second filter captures only the fundamental OS vulnerabilities, and it basically corresponds to a server platform that, to decrease security risks, is stripped of all applications with the exception of the offered single service (called a *Thin Server*). The third filter represents a similar configuration, but where the machine is physically protected from illegal access and therefore it can only be remotely attacked (called an *Isolated Thin Server*).

The number of shared vulnerabilities between two OS is substantially reduced when compared to the overall set of vulnerabilities. Even considering a Fat Server configuration, it is possible to find out OS pairs that do not have common flaws (e.g., NetBSD-Ubuntu). As expected, OS from the same family are affected by more common vulnerabilities due to the software components and applications that are reused (e.g., Debian-RedHat or Windows2000-Windows2003). The use of an Isolated Thin Server, when compared with a Fat Server, has a strong impact on the security of the platform because it decreases the number of common vulnerabilities by 56% on average. This means that a significant portion of common vulnerabilities are local (i.e., cannot be exploited remotely) or come from applications that are available on both operating systems.

Table 3.4 shows which part of the OS is affected by common vulnerabilities in an Isolated Thin Server configuration, considering only the OS pairs with non-zero common vulnerabilities. The fact that there are many common *Kernel* and *System Software* vulnerabilities between Windows 2000 and 2003 indicates that the latter inherits considerable parts of the OS from its predecessor. This same trend is also observed between Windows 2008 and Windows 2003/Windows 2000, although to a less extent. Interestingly, no single vulnerable driver is present in all products, which can be explained by the very few faulty drivers that are reported.

The second family of OS with more common vulnerabilities is BSD, which also reutilizes several components of the operating system. A somewhat surprising result is the fact that most Linux distributions have much less common vulnerabilities than we anticipated. We inspected manually the vulnerabilities in order to find an explanation, and

Operating Systems	**All**			**No Applications**			**No App. and No Local**		
Pairs (A-B)	$v(A)$	$v(B)$	$v(AB)$	$v(A)$	$v(B)$	$v(AB)$	$v(A)$	$v(B)$	$v(AB)$
OpenBSD-NetBSD	142	126	40	110	100	32	60	41	16
OpenBSD-FreeBSD		258	53		205	48		87	32
OpenBSD-OpenSolaris		31	1		24	1		6	0
OpenBSD-Solaris		400	12		272	10		103	6
OpenBSD-Debian		201	2		59	2		25	0
OpenBSD-Ubuntu		87	3		32	1		10	0
OpenBSD-RedHat		369	10		187	5		58	4
OpenBSD-Windows2000		481	3		278	3		178	3
OpenBSD-Windows2003		343	2		167	2		109	2
OpenBSD-Windows2008		118	1		56	1		26	1
NetBSD-FreeBSD	126	258	49	100	205	39	41	87	24
NetBSD-OpenSolaris		31	0		24	0		6	0
NetBSD-Solaris		400	15		272	12		103	8
Netbsd-Debian		201	3		59	2		25	2
NetBSD-Ubuntu		87	0		32	0		10	0
NetBSD-RedHat		369	7		187	4		58	2
NetBSD-Windows2000		481	3		278	3		178	3
NetBSD-Windows2003		343	1		167	1		109	1
NetBSD-Windows2008		118	1		56	1		26	1
FreeBSD-OpenSolaris	258	31	0	205	24	0	87	6	0
FreeBSD-Solaris		400	21		272	15		103	8
FreeBSD-Debian		201	7		59	4		25	1
FreeBSD-Ubuntu		87	3		32	3		10	0
FreeBSD-RedHat		369	20		187	13		58	5
FreeBSD-Windows2000		481	4		278	4		178	4
FreeBSD-Windows2003		343	2		167	2		109	2
FreeBSD-Windows2008		118	1		56	1		26	1
OpenSolaris-Solaris	31	400	27	24	272	22	6	103	6
OpenSolaris-Debian		201	1		59	1		25	0
OpenSolaris-Ubuntu		87	1		32	1		10	0
OpenSolaris-RedHat		369	1		187	1		58	0
OpenSolaris-Windows2000		481	0		278	0		178	0
OpenSolaris-Windows2003		343	0		167	0		109	0
OpenSolaris-Windows2008		118	0		56	0		26	0
Solaris-Debian	400	201	4	272	59	4	103	25	2
Solaris-Ubuntu		87	2		32	2		10	0
Solaris-RedHat		369	13		187	8		58	4
Solaris-Windows2000		481	9		278	3		178	3
Solaris-Windows2003		343	7		167	1		109	1
Solaris-Windows2008		118	0		56	0		26	0
Debian-Ubuntu	201	87	12	59	32	6	25	10	2
Debian-RedHat		369	61		187	26		58	11
Debian-Windows2000		481	1		278	1		178	1
Debian-Windows2003		343	0		167	0		109	0
Debian-Windows2008		118	0		56	0		26	0
Ubuntu-RedHat	87	369	25	32	187	8	10	58	1
Ubuntu-Windows2000		481	1		278	1		178	1
Ubuntu-Windows2003		343	0		167	0		109	0
Ubuntu-Windows2008		118	0		56	0		26	0
RedHat-Windows2000	369	481	2	187	278	1	58	178	1
RedHat-Windows2003		343	1		167	0		109	0
RedHat-Windows2008		118	0		56	0		26	0
Windows2000-Windows2003	481	343	253	278	167	116	178	109	81
Windows2000-Windows2008		118	70		56	27		26	14
Windows2003-Windows2008	343	118	95	167	56	39	109	26	18

Table 3.3: Vulnerabilities (1994 to (Sept.) 2010): *All* - all vulnerabilities; *No Application* - no application vulnerabilities; *No App. and No Local* - no application vulnerabilities and only remotely-exploitable vulnerabilities.

we discovered that Linux distributions customize both their kernels and system software, and thus the vulnerabilities are less common. Another interesting point about OS from the Linux family is that they have an almost negligible number of driver vulnerabilities (see Table 3.2), and none of them appears in more than one OS.

OS Pairs	*Driver*	*Kernel*	*System Software*	*Total*
Win2000-Win2003	0	40	41	81
OpenBSD-FreeBSD	1	14	17	32
NetBSD-FreeBSD	2	13	9	24
Win2003-Win2008	0	10	8	18
OpenBSD-NetBSD	1	8	7	16
Win2000-Win2008	0	8	6	14
Debian-RedHat	0	5	6	11
FreeBSD-Solaris	0	5	3	8
NetBSD-Solaris	0	4	4	8
OpenBSD-Solaris	0	5	1	6
OpenSolaris-Solaris	0	3	3	6
FreeBSD-RedHat	0	1	4	5
FreeBSD-Win2000	1	3	0	4
OpenBSD-RedHat	0	1	3	4
Solaris-RedHat	0	3	1	4
NetBSD-Win2000	1	2	0	3
OpenBSD-Win2000	0	3	0	3
Solaris-Win2000	0	3	0	3
Solaris-Debian	0	1	1	2
OpenBSD-Win2003	0	2	0	2
FreeBSD-Win2003	0	2	0	2
Debian-Ubuntu	0	0	2	2
NetBSD-Debian	0	0	2	2
NetBSD-RedHat	0	0	2	2
NetBSD-Win2003	0	1	0	1
NetBSD-Win2008	0	1	0	1
OpenBSD-Win2008	0	1	0	1
FreeBSD-Win2008	0	1	0	1
Solaris-Win2003	0	1	0	1
FreeBSD-Debian	0	0	1	1
Debian-Win2000	0	0	1	1
Ubuntu-RedHat	0	0	1	1
Ubuntu-Win2000	0	0	1	1
RedHat-Win2000	0	0	1	1

Table 3.4: Common vulnerabilities on Isolated Thin Servers.

We extended the study of common vulnerabilities to higher numbers of OS. When we created combinations of three OS, we found that there are still 285 vulnerabilities that could compromise the systems (in general these three systems are from the same family).

This number is reduced to 102 and 9 vulnerabilities, respectively, in groups of four and five OS. There are only two vulnerabilities shared by six OS (with identifiers CVE-2008-1447 and CVE-2007-5365), and one vulnerability that appears in nine OS (with identifier CVE-2008-4609). The first two cases correspond to protocols that are implemented in a similar manner in various systems, namely DNS and DHCP, and the last one to a well-known denial of service problem in the TCP design.

3.2.3 Selecting the OS for the Replicas

Recall that when building an intrusion-tolerant replicated system, one would like to pick a group of OS for the servers that share a minimum number of vulnerabilities (ideally zero). This selection ensures that the adversary spends more effort and time to break the system, since he or she has to compromise each replica separately[8]. Typical intrusion-tolerant systems require at least $3f + 1$ replicas to tolerate f faults (e.g., [8, 11, 41]), and in some specific services they might only need $2f + 1$ or more replicas (e.g., [13, 67]).

	OpenBSD	NetBSD	FreeBSD	Solaris	Debian	RedHat	Win2000	Win2003	
OpenBSD		9	25	6	0	4	2	1	1994-2005
NetBSD	7		15	8	2	2	2	0	
FreeBSD	7	9		8	1	5	3	1	
Solaris	0	0	0		2	3	3	1	
Debian	0	0	0	0		10	0	0	
RedHat	0	0	0	1	1		0	0	
Win2000	1	1	1	0	1	1		35	
Win2003	1	1	1	0	0	0	46		
	2006-2010								

Table 3.5: History/observed period results for Isolated Thin Servers.

The results from the previous section give a strong indication that it should be possible to choose groups of OS with few common vulnerabilities over long intervals of time. However, we would like to understand if the data from the NVD database is effective at suggesting these groups of OS. To address this point, we divided the data in two subsets: the *history period* comprising the data from the interval 1994 to 2005 (2/3 of the valid vulnerabilities), and the *observed period* for the years between 2006 and 2010 (1/3 of the valid vulnerabilities). The objective is to employ the history period to pick groups of OS to deploy in an hypothetical intrusion-tolerant system (e.g., BFS [11] or DepSpace [8]), and then use the data on the observed period to verify if the number of shared vulnerabilities is as small as expected. Table 3.5 presents the result of the analysis for groups of Isolated Thin Servers. The experiment does not consider Ubuntu, OpenSolaris and Windows 2008 due to the lack of meaningful data during the history period. In the table, values above the diagonal line and to the right correspond to common vulnerabilities in pairs of OS during

[8]This is valid under the assumption that the cost to compromise each OS is non-negligible and approximately the same.

the history period. Values to the left and below the diagonal line represent the observed period results.

For the *base case* consider that one wants to tolerate a single intrusion, i.e., $f = 1$, in a set of four identical (non-diverse) replicas (e.g., because one wants to keep administrative tasks simple). The best strategy for this scenario would be to pick the OS with the least vulnerabilities during the history period. Debian would be the best choice because it only had 16 vulnerabilities that could be remotely exploited either in the drivers, kernel or system programs. Over the observed period, this system would have 9 shared vulnerabilities (i.e., the ones that were reported for Debian between 2006 and 2010) that could compromise the four replicas of the hypothetical system (see Figure 3.3).

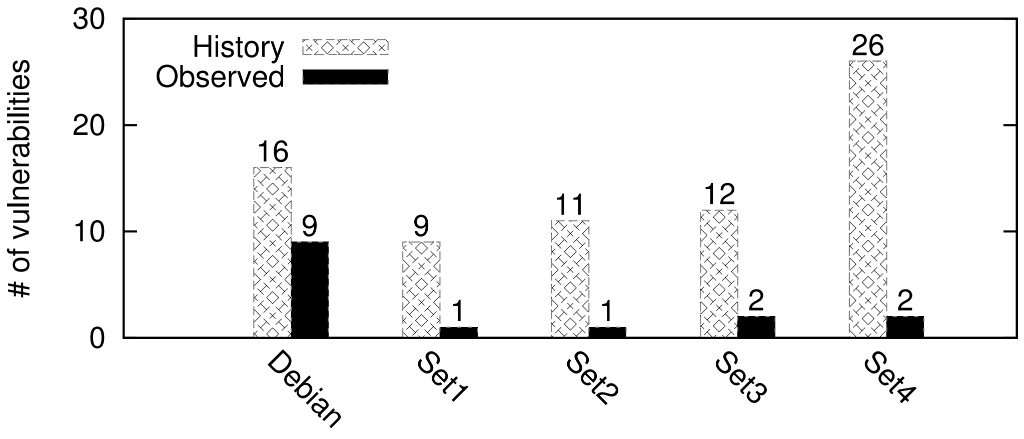


Figure 3.3: Several configurations of OS: Debian - only Debian; *Set1* is {Win2003, Solaris, Debian, OpenBSD}; *Set2* is {Win2003, Solaris, Debian, NetBSD}; *Set3* is {Win2003, Solaris, RedHat, NetBSD}; *Set4* is {OpenBSD, NetBSD, Debian, Redhat}.

If one had chosen to employ the "most diverse" operating system group based on what was reported on the history period, then the selected OS would be Set1 of Figure 3.3, which is composed by {Windows 2003, Solaris, Debian and OpenBSD}. During the observed period, this set would only have one vulnerability affecting two of the replicas – OpenBSD and Windows 2003. Alternatively, if we had chosen the second "most diverse" configuration, where NetBSD would substitute OpenBSD, then one would add 3 extra common vulnerabilities during the history period. However, during the observed period, one would still only have a single common vulnerability (between OpenBSD and Windows 2003). Therefore, in both configurations of the intrusion-tolerant system, the number of common vulnerabilities would be extremely small, and lower than in the base case.

The results also point out that one can deploy an intrusion-tolerant system with few common vulnerabilities, which is based only on Linux distributions and BSD flavors (Set4 in Figure 3.3). Since these four OS can be managed in a relatively similar way, this type of configuration can be extremely useful for organizations that need to operate with tight budgets – for instance, it would not be necessary to hire personnel that knows how to administer Solaris or Windows machines.

Table 3.5 shows that it is possible to build a set of six operating systems with few vulnerabilities: two from the BSD family (OpenBSD and NetBSD), one from the Windows family (Windows 2003), the two Linux (Debian and RedHat) and Solaris. By adding one extra operating system, either FreeBSD or OpenSolaris (which only had 6 common vulnerabilities with Solaris in the observed period), we would have seven options available, making it possible to deploy diverse systems up to $f = 2$ and $f = 3$, for $3f+1$ and $2f+1$ replicas, respectively.

3.2.4 Exploring Diversity Across OS Releases

The results from the previous section are encouraging if one wants to build systems capable of tolerating a few intrusions, since it is possible to select OS for the replicas with a small collection of common vulnerabilities. It is hard, however, to support critical services that need to remain correct with higher numbers of compromised replicas or to use some Byzantine fault-tolerant algorithms that trade performance by extra replicas (e.g., [1, 56]).

The number of available operating systems is limited, and consequently, one rapidly runs out of different OS (e.g., it is necessary 13 distinct OS to tolerate $f = 4$ in a $3f + 1$ system). On the other hand, our experiments are relatively pessimistic in the sense that they are based in long periods of time and no distinctions are made between OS releases. Newer releases of an OS can contain important code changes, and therefore, current vulnerabilities may not appear in previous versions. As a result, if we consider (OS, release) pairs, one may increase the number of different systems that do not share vulnerabilities.

As mentioned in Chapter 2, Ozment and Schechter [48] made a study on OpenBSD code sharing. They observed that after 7 years only 61% of the code remained the same (between the two versions of OpenBSD 2.3 and OpenBSD 3.7). At first glance, this observations support the idea that even with different releases of one OS, it is possible to achieve diversity. Crossing these results with Figure 3.2(b) and the distribution dates of the two releases (respectively, 19th May 1998 and 18th May 2005), one can understand that 61% of the shared code should not create important problems. This part of the code was revised along seven years and was already patched, and therefore, it should contain less vulnerabilities.

This provides a good indication that exploring diversity across OS releases might offer a potential security gains. We looked for security advisories (or trackers) available in the various OS websites to determine if they correlate the vulnerabilities patched in each release with the information in NVD. This correlation was found in a meaningful way in four of the OS under study: NetBSD[9], Debian[10], Ubuntu[11], and RedHat[12]. From all combinations of pairs of these OS in an Isolated Thin Server configuration, the pair with highest number of common vulnerabilities is Debian-RedHat (see Tables 3.3 and 3.4). Table 3.6 presents the number of common vulnerabilities for three releases of Debian and RedHat, spread along the following years: Debian2.1, 1999; Debian3.0, 2002; Debian4.0, 2007; RedHat6.2*, 2000[13]; RedHat4.0, 2005; RedHat5.0, 2007. One can observe that even though Debian-RedHat shared a total eleven vulnerabilities, the (OS, release) pairs are mostly without common flaws, both in the case of the same OS but distinct releases (left side of the table) and between different operating systems (right side of the table). These same kind of benefits were also reported in a previous work related with non-security bugs for database management systems [21].

OS Versions	Total	**OS Versions**	Total
Debian2.1-Debian3.0	0	Debian3.0-RedHat6.2*	0
Debian2.1-Debian4.0	0	Debian3.0-RedHat4.0	0
Debian3.0-Debian4.0	1	Debian3.0-RedHat5.0	0
RedHat6.2*-RedHat4.0	0	Debian4.0-RedHat6.2*	0
RedHat6.2*-RedHat5.0	0	Debian4.0-RedHat4.0	1
RedHat4.0-RedHat5.0	1	Debian4.0-RedHat5.0	1
		Debian2.1-RedHat6.2*	0
		Debian2.1-RedHat4.0	0
		Debian2.1-RedHat5.0	0

Table 3.6: Common vulnerabilities between OS releases.

3.3 Final Remarks

The main goal of Vulnerability study presented in this chapter was to understand if the data available in on-line databases provides sufficient information to determine if diversity across operating systems ensures vulnerability independence. To made this study, we used the National Vulnerability Database as a source of the data, and then we selected and filtered some of the information available on the whole data set. Various types of analysis were performed on the resulting data set. We found that there is a strong suggestion of vulnerability independence across different operating systems.

The main findings of the study can be summarized as:

[9]http://www.netbsd.org/support/security/release.html

[10]http://security-tracker.debian.org/tracker/

[11]http://people.canonical.com/ ubuntu-security/cve/

[12]https://www.redhat.com/security/data/cve/

[13]As a cautious note, RedHat had change the version naming, that is why RedHat 6.2 was released in 2000 and RedHat 4.0 appears only after.

1. The number of common vulnerabilities on the studied OS pairs was reduced by 56% on average if the application and locally-exploitable vulnerabilities are removed;
2. More than 50% of the 55 OS pairs that were studied have at most one non-application, remotely exploitable common vulnerability;
3. The top-3 diverse setups for a four-replica system (tolerating a single failure in typical intrusion-tolerant systems) are: {Windows 2003, Solaris, Debian and OpenBSD}, {Windows 2003, Solaris, Debian and NetBSD} and {Windows 2003, Solaris, RedHat and NetBSD};
4. A preliminary analysis of the diversity among different versions of Debian and RedHat distributions suggests that there are possible setups with the same OS that have a disjoint set of vulnerabilities.
5. There are two vulnerabilities from 2007 and 2008 that affect six OS, and one vulnerability from 2008 that affected nine OS;
6. Driver vulnerabilities accounts only for a very small set (less than 1.5%) of all reported OS vulnerabilities.

In the next chapter we will use these results to propose a replicated system architecture that uses diverse rejuvenation, to select distinct operating system during the life-time of the system. We will also describe an algorithm that maximizes the diversity on the replicas based on the data analyzed on this chapter.

Chapter 4

Diverse Proactive Recovery

This chapter proposes a replicated system that takes advantage of the results presented in Chapter 3 to build proactive-recovery mechanisms. First we give an overview of intrusion tolerance and present the system architecture. Then, we describe and evaluate an algorithm to select OS configurations in a diverse replicated system. Parts of this chapter appeared in [20].

4.1 System Architecture

As discussed in previous chapters, 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 4.1). 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 (usually 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 [58]).

The architecture of the rejuvenation service is depicted in the part above the line of Figure 4.1. Virtualization is used to divide each replica in two logical components, where

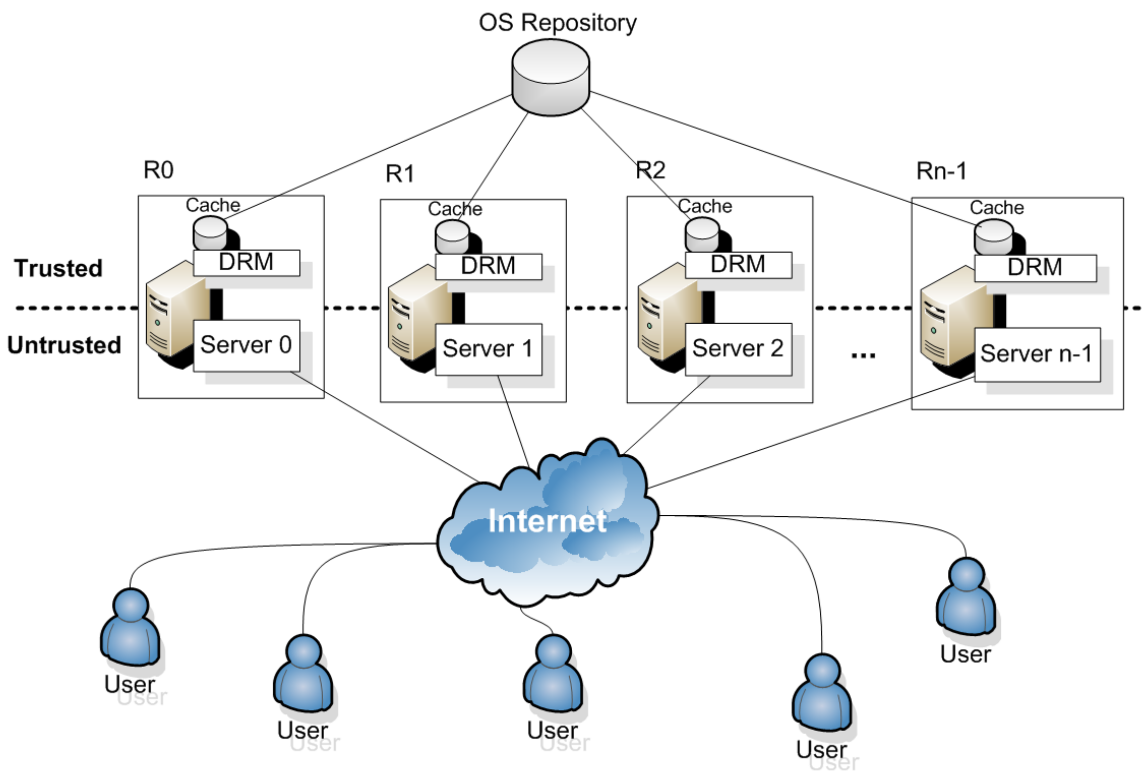


Figure 4.1: Architecture of the rejuvenation system.

the server software is run in a separate virtual machine and the *diversity rejuvenation module* (DRM) is executed in the trusted part of a virtual machine[1]. 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.

Replica rejuvenation can 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 after in the next rejuvenation.

[1]For example, Domain 0 in Xen [66].

- 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. These configurations are stored in a virtual machine image (i.e., a file) that can be loaded 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.

4.2 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 the discussion and make it well grounded on the available results about diverse configurations, we describe the technique using diverse operating systems. However it can be easily extended to deal with configurations composed by a stack of software components.

4.2.1 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[2]. 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.

[2]Notice that we are working under the assumption that finding and exploiting new (or zero-day) vulnerabilities in mature software takes some time.

P3 All diverse OS configurations available in the configuration repository[3] for selection are picked by the algorithm with a reasonable probability.

P4 The algorithm is executed individually by each DRM of the replicas.

As explained, P1 cannot be ensured with absolute certainty. However, the study described in the previous chapter shows a strong empirical evidence for: 1) it is possible to find OS pairs that have had no (or only a few) common vulnerabilities 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. For each OS version pair we can obtain the list of shared vulnerabilities and the CVSS score of each vulnerability.

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 small 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).

[3]The repository has a subset of all available configurations containing the operating systems that match some performance or dependability criteria.

4.2.2 Selection Algorithm

Algorithm 2 is executed 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 VUL_SCORE 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 than this value if there are no alternatives).

The algorithm starts by doing some global initializations (Lines 1-3). The number of rejuvenations $rejCount$ is set to -1 to indicate that 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 4.1, using a round robin policy, at every t time unit, 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 VUL_SCORE (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 of $getScore(OSConf[j], cand)$ the value of $OSConf[j]$ can be $*$. In this case, the $getScore()$ function should return 0, which causes $cand$ to be an acceptable candidate.

The reader should notice that the algorithm is designed in such a way that is always possible 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.

Algorithm 2: A diversity selector algorithm

```
Initialization:
1  $rejCount = -1$;
2  $OSConf = (*, *, *, ..., *)$;
3  $initRandom(seed)$;

initSelectCandidate():
4  $firstC = getRandom()$ mod $size(OSTable)$;
5  $nextC = 0$;
6  $score = VUL\_SCORE$;

selectCandidate():
7  $cand = getOSTable((firstC + nextC)$ mod $size(OSTable))$;
8  if $((nextC > 0)$ and $(nextC$ mod $size(OSTable) = 0))$ then
9      $score = score + \alpha$;
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 $(getScore(OSCon[j], cand) > score)$ then
22             $done = true$;
23         end
24         $j = j + 1$;
25     until $((\neg done)$ and $(j < n))$ ;
26     if $(\neg done)$ then
27         $OSConf[i] = cand$;
28         return $cand$;
29     end
30 end
```

4.3 Evaluation

This section presents some experimental results with an implementation of the diverse proactive recovery system. From the four different OS families, 15 distributions[4] were deployed and configured in a virtual machine environment, in this case *VirtualBox*[5]. Due to drivers or aging limitations we could not install more OS.

The experiments can be divided in two independent evaluations: algorithm evaluation, where we analyse the diverse selection algorithm; and an experimental measurement of the duration of the recovery process for different operating systems.

4.3.1 Experimental Setup

In the first evaluation, we implemented the diversity selector algorithm in Java and them evaluated its capabilities for creating diverse OS sets. This implementation uses as input a table derived from the database designed and enriched from the NVD data set, as described in the previous chapter.

The second evaluation has as objective the measurement of the interval of time that it takes to rejuvenate an OS configuration. The following steps had to be performed to carry out these experiments: First, it was necessary to prepare the virtual machines environment, which in our case was based on *VirtualBox*. Next, a maximum number of OS was downloaded from the official repositories, and then they were installed as guest OS in distinct virtual machines. This task ended up being time consuming due to the several differences between some of the OS, which required for instance specific configurations to make them run in our network environment. Second, it was necessary to develop two programs to control the experiment and take measurements. One, *host*, that ran in the virtual machine host (or hypervisor), and the other, *vm*, that ran in the virtual machine guest. This program was written in the C language and the *vm* part had to be ported to all OS versions.

The experiments were carried out in a Dell Optilex 755 with a Intel Core Due CPU 2.83 Ghz and 3.2 Gb memory RAM. Since the prototype runs in a virtual machine environment, each virtual machine had their own specifications: 1 CPU and 512 MB of memory, and the disk size varies due to the installed OS requirements.

4.3.2 Algorithm Evaluation

In this first experiment we intend to understand how the diversity selection algorithm degrades, i.e, for how long can it provide diversity sets while at the same time minimizing repeating the sets. This experiment uses 15 different OS configurations. With a simple

[4]Windows 2003; Redhat 4.5; Debian 4.0, 5.0 and 6.0; OpenBSD 4.6, 4.7 and 4.8; FreeBSD 7.2, 7.3, 7.4, 8.0 and 8.1; Solaris 10 and Solaris 11 Express.

[5]http://www.virtualbox.org/

combination calculation, these configurations allow for 1365 different possible sets (without repeating and the order is not important). Notice, however, that these combinations can only be used if one changes in each recovery the whole set of OS configurations in all replicas. In our case, the system managing the diversity needs to modify a single OS per recovery, which has an impact on the numbers.

In the evaluation, we run the selection algorithm 20 times for each number of rounds, beginning with 100. This means that function $\mathit{findNextConfiguration}()$ is executed 100 times, to generate different OS sequences with 20 different initial seeds. Figure 4.2 shows the results of the algorithm execution. As an example, the first column of the graph has the following meaning: when the algorithm is run 100 times there are in average 14 sets of 4 replicas that are repeated.

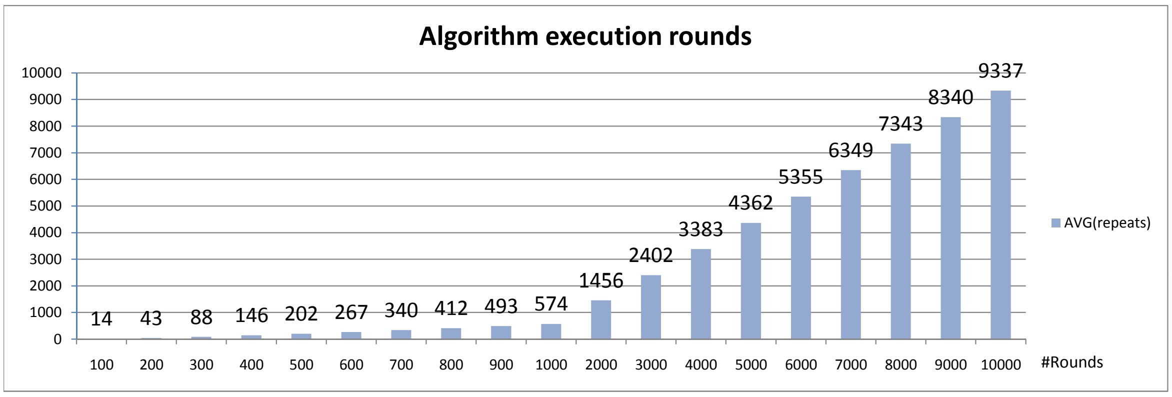


Figure 4.2: The degradation evolution of the algorithm.

When the algorithm executes 700 rounds, the occurrence of repeating sets is always approximately half of the rounds. From 700 to 10000 rounds, the repeating sets starts to get closer to the number of executed rounds. This result is expected since we have a limited number of combinations without having to re-use a previously selected set.

Taking this into account, there is a trade-off between the life-time duration of a system and the security gains from using diversity. For short periods of time more diversity can be offered by the selector algorithm. For longer periods of execution, the algorithm is required to repeat some sets, and therefore, the system becomes more vulnerable to attacks, under the assumption that with longer runtimes the attacker has more time to discovery zero-day vulnerabilities. For example looking at the 600 rounds value on the x-axis and considering that each replica recovers once per hour, we can maintain the system with only 267 repeated sets for 25 days.

4.3.3 Recovery Evaluation

In this experiment we intend to evaluate the time (in seconds) spent in recoveries, which are composed of two tasks: disk copy and OS boot. As in previous works [51, 59], virtual machines were used to create the trusted and untrusted execution environments (see Figure 4.1). A program was developed to measure the time of cloning a disk and the booting time, and then the program was ported to all OS running on the virtual machines. The first task, cloning a disk, uses the *VBoxManager*[6] function to copy disks. This function allows one to clone a virtual hard drive that can be loaded in a new instance of a virtual machine with a distinct unique identifier (UUID).

The second task, the booting time, corresponds to the duration between the virtual machine start instant and the boot level where the ported program executes. This varies from OS to OS, for example Linux OS has six init levels, each one has a specific function. In our case the program is run at the fifth level with the */etc/rc.d* scripts. When the program starts to execute, it sends an UDP message to the host to inform that the reboot was completed. Then, the host starts a new experiment by deleting the current virtual machine and loading a new one, and so on until, in this particular experiment, 20 recoveries are executed.

Tables 4.1, 4.2, 4.3, 4.4 and 4.5 present the measurements for Debian-5.0, FreeBSD-7.3, Windows 2003, Solaris 11 Express and Redhat-5.6. We choose one operating system from each family because it seems reasonable to assume that there are not so many differences on the booting time of the similar OS.

Debian-5.0		
	Clone	**Boot**
mean	21.80s	31.90s
stddev	0.50s	1.38s
max	22.64s	34.24s
min	20.92s	29.87s

Table 4.1: Disk size: 1.02 GB.

FreeBSD-7.3		
	Clone	**Boot**
mean	96.036s	41.14s
stddev	2.24s	0.70s
max	100.58s	41.83s
min	92.52s	38.73s

Table 4.2: Disk size: 2.2 GB.

RedHat-5.6		
	Clone	**Boot**
mean	98.36s	109.87s
stddev	3.31s	21.18s
max	94.77s	150.58s
min	20.52s	88.39s

Table 4.3: Disk size: 2.2 GB.

Solaris 11 Express		
	Clone	**Boot**
mean	134.67	111.88
stddev	1.59s	6.04s
max	138.32s	124.68s
min	132.46s	104.22s

Table 4.4: Disk size: 3.3 GB.

Windows2003		
	Clone	**Boot**
mean	427.94s	46.15s
stddev	1.38s	0.38s
max	430.41s	47.39s
min	424.37s	45.25s

Table 4.5: Disk size: 10.5 GB.

[6]http://www.virtualbox.org/manual/ch08.html

The results show that, as expected, the time to clone an operating system image is directly dependent of its size. Besides that, one can see that Solaris and Redhat takes more than twice the time to boot than Windows, FreeBSD and Debian.

4.4 Final Remarks

In this chapter we presented an intrusion-tolerant replicated system architecture that uses operating system diversity on rejuvenations. To guarantee the maximum diversity across the replicas we made an algorithm that selects the best operating systems for the *n* replicas among the pool of available configurations. This algorithm uses the data that we analyzed in the previous chapter. The chapter also provided an evaluation of the algorithm, together with experiments on the amount of time that it takes to rejuvenate an operating system.

Chapter 5

Conclusion

This chapter makes a discussion of the whole work to summarize the contributions of the thesis. Then, we present some of the limitations of NVD and how they might have influenced the results of Chapter 3. Next we describe the limitations of the diverse recovery architecture. We complete the thesis by proposing few ideas for the future work.

5.1 Summary of Results

One way to decrease the probability of common vulnerabilities on the replicas of intrusion-tolerant systems is by using diverse OTS software components. In this thesis we analyzed the likelihood of common vulnerabilities on an important class of OTS components used in intrusion-tolerant systems: operating systems. We analyzed more than 15 years of vulnerability reports from NVD totaling 1887 vulnerabilities of eleven operating system distributions. The results suggest a substantial security gain by using diverse operating systems for intrusion tolerance. Of course, our analysis has several limitations. Some of these limitations are discussed in detail in the next section, and we explain what additional data, analysis and clarifications may be needed to increase our confidence about the claims on the benefits of diversity. Despite these limitations, we argue that on average our estimates may be seen as conservative as we analyzed aggregated vulnerabilities across releases – hence common vulnerabilities could be smaller in a "specific set" of diverse OS releases. A better analysis would be obtained if the NVD vulnerability reports were combined with the exploit reports (including exploit counts), and even better if they also had indications about the users' usage profile. However, vendors are often wary of sharing such detailed dependability and security information with their customers. There are partial exploit reports available from other sites (e.g., [46]), but they are incomplete and a significant amount of manual analysis is required to match the vulnerabilities with exploits for each operating system.

The thesis also presents a system that adds proactive recovery to a diverse replicated system, by introducing diversity in OS configurations after each rejuvenation. The architecture can be briefly described as a n replica system in which each node has a DRM (Diverse Recovery Module) and a server that provides a service to the users (e.g., firewall or a database). When one DRM$_i$ needs to rejuvenate some server$_i$, it gets from a secure OS repository a new and different OS image to load on the server$_i$. We have created an algorithm to maximize the diversity on each rejuvenation given a set of n running replicas, increasing the difficulty for the adversary to exploit $f+1$ out-of n replicas, which would compromise the system availability. This algorithm uses random selection to avoid adversary predictions about the next OS to be deployed, but it runs deterministically on each replica to avoid the communication between DRMs.

5.2 Limitations of the Work

In this section we present some of the limitations of the developed work.

5.2.1 Limitations of NVD and its Implications on the Study

The numbers we have presented in Chapter 3 are intriguing and point to a potential for serious security gains from assembling an intrusion-tolerant system using different operating systems, but they are not a definitive evidence. Even though the NVD is arguably *the* most complete and referenced database for security vulnerabilities and it is regularly updated with contributions from several sources, there are several uncertainties that remain about the data, which limit the claims we can make about the benefits of diversity to increase security. Ozment [47] points out some problems with the NVD (chronological inconsistency, inclusion, separation of events and documentation); for our purposes, the first two and the last one are the most relevant. “Chronological inconsistency” means that the NVD data has inherent inaccuracies about the dates when vulnerabilities were discovered and when the vulnerable code was released, which not only complicates reasoning about the lifetime of vulnerabilities but also affects the versions that are vulnerable (for instance, sometimes obsolete versions of a product are vulnerable but are not listed in the NVD as such). “Inclusion” refers to the fact that not all vulnerabilities are included in the NVD, only those with a CVE number; as CVE and NVD have gained traction, this has become less of an issue. Finally, there is little documentation about the NVD, and, in the past, the meaning of some fields has occasionally changed without prior notice, which might make comparisons less meaningful. In what follows, we will discuss some other limitations and the implications that they have on the claims we can make about the benefits of diversity:

1. The NVD does not provide “reproducible scripts” or exploits – probably wisely – which would allow one to check whether the vulnerability can be exploited. Therefore, relying solely on the data available in the NVD, it is not possible to confirm

that a reported vulnerability is actually exploitable.
Implication: The lack of exploitability information makes it harder to adequately assess the risk posed by a vulnerability. Caution forces us to consider that all vulnerabilities are exploitable, and must be remediated in due time, a strategy that has obvious implications both in terms of cost and in terms of complexity of management.

2. When a vulnerability is reported for more than one operating system, it is not clear whether the reporter has checked that it has been confirmed to exist in the OS, or it is just an indication that the vulnerability may exist in each of the operating systems listed.
Implication: The implications of the previous item apply here as well. Additionally, we have the implication that we cannot claim with certainty whether our estimates of the benefits of diversity, given earlier in the thesis, are conservative or optimistic. If a vulnerability has been reported for operating systems A and B but in fact only exists in A, then our estimates are conservative. On the other hand, if the vulnerability has been reported for operating systems A and B only, but in fact it exists additionally in operating systems C and D, then our estimates are optimistic.

3. Although more than 70 organizations (including many important OS vendors) use CVE to identify vulnerabilities, it is not clear if all products are equally represented in the NVD. Another related issue is that the vulnerability reporting process is inherently biased, both in timing and in coverage, although not necessarily in an intentional manner. For instance, when a new class of vulnerabilities is discovered or disseminated, there is often a surge of new reports involving this class, as it has happened with format string bugs [43] and integer overflows [2]. Finally, not all targets are given the same attention by vulnerability researchers. Software with smaller user bases tend to attract less scrutiny than popular ones, vulnerabilities with higher impact usually receive more attention, and there is even the case when specific vendors are targeted for some reason. As an example, when Oracle claimed their database was "unbreakable" only to have several vulnerabilities disclosed within 24 hours [32], and the rise in exploitation of Adobe software in the last 15–20 months [29].
Implication: With any analysis of bug or vulnerability reports from an open database, there is uncertainty about how many of the vulnerabilities are actually reported. This fraction is certainly less than 100%. If all vulnerabilities had the same probability of being reported, the ratio between our predicted vulnerability counts for AB (mAB – those that affect both products A and B) and A or B (mA or mB – those that affect only one of the products) would still be the ratio mAB/mA or mAB/mB respectively. But, in fact, we do not know whether the vulnerabilities

of some operating systems are less likely to be reported in NVD than others (or conversely). It is not clear if the vulnerabilities of some operating systems are reported to the vendors only (or some other vulnerability database) and do not appear in NVD. This again has implications about the claims that we can make about the benefits of diversity, as data entries may be missing which overestimate the benefits of diversity for some products.

5.2.2 DRM Limitations

The ideas outlined in Chapter 4 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 have to be addressed:

1. At startup we must have all virtual machine images with the different OS configurations 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.

2. The $OSTable$ construction is based on results from empirical studies such as [19, 21], which do 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.

3. Defining VUL_SCORE and α is still an open problem highly dependent of results of the $OSTable$.

5.3 Future Work

The work developed during the thesis is a first step towards the design and development of diverse intrusion tolerant systems. The main idea is to build a system that will offer some already well known and accepted solutions, such as BFT protocols, state transfer and replica rejuvenation, and then the novel contribution will be an implementation that integrates different levels of diversity. At the end we expect to make a practical evaluation that proves that this architecture is better than a non-diverse solution.

Below, there are ideas for the future work:

- Study and analyze some recent resource-efficient replication models and verify if the DRM architecture can be integrated on them [65].

- Integrate the architecture presented in the Chapter 4 with the BFT-Smart library[1].
- Make a deeper analysis on diversity among different releases of the same OS.
- Explore other opportunistic diversity possibilities besides different OS configurations. For example, we would like to experiment memory layout randomization and code obfuscation to increase heterogeneity across the replicas and explore other taxonomies of diversity within the OS [45].
- Further research is needed on how to do automatic patching in the stored OS configurations [64].

[1]http://code.google.com/p/bft-smart/

Bibliography

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2005.

[2] D. Ahmad. The rising threat of vulnerabilities due to integer errors. *IEEE Security & Privacy*, 1(4), 2003.

[3] O. Alhazmi and Y. Malayia. Quantitative vulnerability assessment of systems software. In *Proceedings of the Annual Reliability and Maintainability Symposium*, 2005.

[4] O. Alhazmi and Y. Malayia. Application of vulnerability discovery models to major operating systems. *IEEE Transactions on Reliability*, 57(1), 2008.

[5] P. Anbalagan and M. Vouk. Towards a unifying approach in understanding security problems. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, 2009.

[6] R. Anderson. Security in open versus closed systems—the dance of Boltzmann, Coase and Moore. In *Conference on Open Source Software: Economics, Law and Policy*, 2002.

[7] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the IEEE Computer Software and Applications Conference*, 1977.

[8] A. Bessani, E. Alchieri, M. Correia, and J. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the ACM European Conference on Computer Systems*, 2008.

[9] A. Bessani, P. Sousa, M. Correia, N. Neves, and P. Verissímo. The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy*, 6(6), 2008.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the the USENIX Symposium on Operating Systems Design and Implementation*, 1999.

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

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2001.

[13] M. Correia, N. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2004.

[14] CVE terminology. `http://cve.mitre.org/about/terminology.html`.

[15] Y. Deswarte, K. Kanoun, and J. Laprie. Diversity against accidental and deliberate faults. In *Computer Security, Dependability, and Assurance: From Needs to Solutions*, 1998.

[16] T. Distler, R. Kapitza, and H. Reiser. Efficient state transfer for hypervisor-based proactive recovery. In *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems*, 2008.

[17] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 1997.

[18] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the Large Installation System Administration Conference*, 2006.

[19] M. Garcia, A. Bessani, I. Gashi, N. 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*, 2011.

[20] M. Garcia, A. Bessani, and N. Neves. Diverse os rejuvenation for intrusion tolerance. Poster in the IEEE/IFIP International Conference on Dependable Systems and Networks, 2011.

[21] 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*, 4(4), 2007.

[22] S. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4), 2000.

[23] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In *Proceedings of the IEEE Proceedings of International Symposium on Fault-Tolerant Computing*, 1993.

[24] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: analysis, module and applications. In *Proceedings of the IEEE Proceedings of International Symposium on Fault-Tolerant Computing*, 1995.

[25] M. Joseph and A. Avizienis. A fault-tolerant approach to computer viruses. In *IEEE Security & Privacy*, 1988.

[26] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions Software Engineering*, 12, 1986.

[27] J. Knight and N. Leveson. A reply to the criticisms of the Knight & Leveson experiment. *ACM SIGSOFT Software Engineering Notes*, 15, 1990.

[28] P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. In *Proceedings of the IEEE Proceedings of International Symposium on Fault-Tolerant Computing*, 1999.

[29] McAfee Labs. 2010 threat predictions. Whitepaper, 2009. Available from `http://www.mcafee.com/us/local_content/white_papers/7985rpt_labs_threat_predict_1209_v2.pdf`.

[30] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3), 1982.

[31] Liburd and S. Denise. An n-version electronic voting system. Master's thesis, Massachusetts Institute of Technology, 2004.

[32] D. Litchfield. Hackproofing Oracle Application Server. Whitepaper, NGSSoftware Insight, 2002.

[33] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: A review. *ACM Computing Surveys*, 33(2), 2001.

[34] B. Littlewood and L. Strigini. Redundancy and diversity in security. In *Proceedings of the European Symposium on Research in Computer Security*. 2004.

[35] M. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1995.

[36] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6), 2006.

[37] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[38] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. CS-TR 1995–1268, University of Wisconsin-Madison, 1995.

[39] MITRE. Common platform enumeration. `http://cpe.mitre.org/`.

[40] H. Moniz, N. Neves, M. Correia, and P. Veríssimo. Randomized intrusion-tolerant asynchronous services. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2006.

[41] H. Moniz, N. Neves, M. Correia, and P. Verissímo. RITAS: Services for randomized intrusion tolerance. *IEEE Transactions on Dependable and Secure Computing*, 8(1), 2011.

[42] L. Nagy, R. Ford, and W. Allen. N-version programming for the detection of zero-day exploits. CS 06–04, Florida Institute Technologies, 2006.

[43] T. Newsham. Format string attacks. Technical report, Guardent, Inc., 2000. Available from `http://www.thenewsh.com/~newsham/format-string-attacks.pdf`.

[44] National Vulnerability Database. `http://nvd.nist.gov/`.

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

[46] S. Özkan. CVE details website. `http://www.cvedetails.com/`.

[47] A. Ozment. *Vulnerability Discovery & Software Security*. PhD thesis, University of Cambridge, 2007.

[48] A. Ozment and S. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the USENIX Security Symposium*, 2006.

[49] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2), 1975.

[50] B. Randell. System structure for software fault tolerance. *ACM Special Interest Group on Programming Languages Notices*, 10(6), 1975.

[51] H. Reiser and R. Kapitza. Fault and intrusion tolerance on the basis of virtual machines. In *Tagungsband des 1. Fachgesprch Virtualisierung*, 2008.

[52] E. Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1), 2005.

[53] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2001.

[54] T. Roeder and F. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems*, 28(2), 2010.

[55] G. Schryen. Security of open source and closed source software: An empirical comparison of published vulnerabilities. In *Proceedings of the ACM Americas Conference on Information Systems*, 2009.

[56] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast Byzantine replication in presence of unresponsive replicas. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010.

[57] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Verissímo. Resilient intrusion tolerance through proactive and reactive recovery. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing*, 2007.

[58] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Verissímo. Highly available intrusion-tolerant services with proactive-reactive recovery. *Proceedings of the IEEE Transactions on Parallel and Distributed Systems*, 21(4), 2010.

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

[60] P. Sousa, N. Neves, and P. Verissímo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2005.

[61] P. Sousa, N. Neves, and P. Veríssimo. Proactive resilience through architectural hybridization. In *Proceedings of the ACM Symposium on Applied Computing*, 2006.

[62] P. Sousa, N. Neves, and P. Verissímo. Hidden problems of asynchronous proactive recovery. In *Proceedings of the Workshop on Hot Topics in System Dependability*, 2007.

[63] P. Verissímo, N. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. Springer, 2003.

[64] M. Vojnovic and A. Ganesh. On the effectiveness of automatic patching. In *Proceedings of the ACM Workshop on Rapid Malcode*, 2005.

[65] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the ACM European Conference on Computer Systems*, 2011.

[66] Xen. http://www.xen.org/, 2011.

[67] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement form execution for Byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.