UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS

DIVERSE INTRUSION-TOLERANT SYSTEMS

Doutoramento em Informática

Miguel Garcia Tavares Henriques

Tese orientada por:
Prof. Doutor Alysson Neves Bessani
e co-orientada pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Documento especialmente elaborado para a obtenção do grau de doutor

2018

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS

DIVERSE INTRUSION-TOLERANT SYSTEMS

Doutoramento em Informática

Miguel Garcia Tavares Henriques

Tese orientada por:
Prof. Doutor Alysson Neves Bessani
e pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Júri Presidente:

- Nome do Presidente

Vogais:

- Nome do Vogal
- Nome do Vogal
- Nome do Vogal
- Nome do Vogal

2018

Documento especialmente elaborado para a obtenção do grau de doutor

Este trabalho foi financiado pela Fundação para a Ciência e Tecnologia (FCT) através da Bolsa Individual de Doutormento SFRH/BD/84375/2012 e através dos projectos da Comissão Europeia FP7-257475 (MASSIF) e H2020-700692 (DiSIEM).

Abstract

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

Resumo

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

Resumo Estendido

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

Acknowledgements

To my grandmother.

Contents

List of Figures

List of Tables

Introduction

1

1.1 Context and Motivation

Intrusion Tolerance. Byzantine Fault Tolerance (BFT)[1] is a well-established area of research that aims to guarantee the safety of replicated systems even in the presence of some faulty nodes that arbitrarily deviate from their specification. In a nutshell, BFT protocols guarantee that replicas agree on the order of the message execution, and thus, the replicas work as a replicated state machine. The BFT safety holds even if a subpart of the replicas is faulty (typically f out of n), thus remaining replicas, assumed as correct, execute correctly. Although not explicit, this assumption leverages on the strict condition that nodes (must) fail independently. Otherwise, compromising f replicas is virtually the same as compromising $f+1$.

BFT was first proposed in 1980 by Pease *et al.* [146], in the following years a few works were made on this subject [105, 155]. However, it only as awaken the distributed systems research community to its relevancy in 1999 due to Castro and Liskov's Practical BFT [29]. In the last twenty years of active research on BFT replication, there were made great advances on the performance (e.g., [9, 15, 113]), use of resources (e.g., [16, 122, 183, 190, 197]), and robustness (e.g., [5, 20, 38]) of BFT systems. However, BFT in general and these works in particular, assume, either implicitly or explicitly, that replicas fail independently. This assumption implies that it takes more time to compromise the system (i.e., $f+1$ replicas) if the nodes do not share weaknesses. Nevertheless, a few works rely on some orthogonal diversity mechanism (e.g., [10, 160]) to avoid these common weaknesses, or rule out the possibility of malicious failures from their system models. Moreover, a few works have implemented and experimented with such diversity mechanisms (e.g., [5, 31, 160]), but in a very limited way.

In practice, a few dependable applications take advantage of diversity mechanisms. Nevertheless, in these cases, diversity is applied *intuitively*. For example, in avionics engineering, a few aircraft solutions embraced the diversity of components to avoid accidental failures [196]. Moreover, in 2014, the U.S. Navy developed the Resilient Hull, Mechanical, and Electrical Security (RHIMES) which introduced diversity through a *slightly different implementation* for each programmable logic controller [64]. Another example of a critical

[1] In this thesis we interchange Byzantine fault tolerance (*noun*) with Byzantine fault-tolerant (*adjective*) using the BFT acronym for both.

system that naturally adopted diversity are the Domain Name System (DNS) root servers. According to Lars-Johan Liman (chair of the Root Server System Advisory Committee) each operator has its own configurations, as "*it allows for great operational diversity*" [139]. The goal is to avoid that a single software bug cannot bring down all servers. The blockchain growing technology implements a BFT network in a large and decentralized way. Therefore, it shares the same limitations of the enumerated BFT works, i.e., the network nodes may share common weakness. There is already evidence that Bitcoin [98] suffer from this problem. Moreover, Ethereum stats show that there are a few Operating Systems (OSes) among the nodes [57, 58] which can deter attacks from exploiting common mode failures.

Despite the adoption of (naive) diversity solutions in practical systems, other variables must be considered to make systems dependable and secure. For example, even considering an initial set of n diverse replicas, long-running services need to be cleaned from possible failures and intrusions. A few works on the proactive recovery of BFT systems [30, 53, 147, 160, 170] periodically restart their replicas to clean undetected faulty states introduced by a stealth attacker. However, these works (i) assume replicas failure independence, (ii) do not provide proof to support their diversity decisions or (iii) use limited solutions (e.g., memory randomization). Therefore, replicas keep the same weaknesses unless careful diversity choices are introduced on recoveries.

Finally, contrary to the already established maturity of BFT solutions, no one debated how to apply diversity in a dependable way (i.e., which software configurations are more dependable) and when should these configurations be changed (e.g., recovering replicas with different software).

Vulnerabilities. The reason why BFT solutions need to make stronger assumptions on the replicas failure independence is due to the existence of replicas' vulnerabilities. Vulnerabilities are weaknesses in the code that can be exploited by a malicious agent to compromise some security properties (i.e., integrity, confidentiality, and availability). Zero-day vulnerabilities are a particular type of vulnerabilities, which are especially critical as their disclosure occurs only when the attack is already (being) done. On the contrary, if a vulnerability is discovered by a vendor or by a third-party software analyst, the vendor releases a patch as soon as possible. Moreover, such vulnerabilities are traded on the black market at high prices [3, 176]. For example, big companies and organizations are promoting bug bounty programs as an attempt to find zero-day vulnerabilities before the hackers (e.g., Google Vulnerability Reward Program [76], Facebook Bug Bounty Program [59], Microsoft Bounty [125], and DARPA Cyber Grand Challenge (2016) [48]).

Applying this information to the BFT context, an attacker can compromise a replica once it finds vulnerabilities in it. If the replicas have the same code and configuration, they share the same vulnerabilities, once the attacker discovers a zero-day vulnerability, he or she will eventually exploit it on $f+1$ replicas. The rationale behind diversity on BFT is that different replicas do not share the same vulnerabilities. Therefore, to discover and exploit vulnerabilities to compromise $f+1$ replicas is more difficult than to compromise just f replicas.

1.2 Objectives and Contributions

In the past, several researchers developed and improved techniques that made intrusion tolerance a cornerstone of security and dependability. Nevertheless, a few problems were left open and still lack effective answers. This thesis aims to push forward intrusion tolerance to more accurate and practical grounds. The main contributions of this thesis are summarized into the following points, along with the related publications.

1.2.1 Evidence for Implementing Diversity

The results achieved during the author's MSc thesis encouraged us to extend the work on the diversity of Off-the-Shelf (OTS) OSes [68]. This first step towards the design and development of dependable BFT systems is to assess how accurate were the implicit or explicit assumptions on diversity (e.g., [1, 18, 30, 31, 37, 41, 103, 113, 131, 197]). In that previous work, we studied the vulnerability data from the National Vulnerability Database (NVD) [134] the most complete vulnerability database available to answer to the question: *What are the dependability gains from using diverse OSes on a replicated intrusion-tolerant system?* Nevertheless, we have made some extensions to that work, which are presented in this thesis. In particular, we devised three manual strategies for selecting diverse software components to minimize the incidence of common vulnerabilities in replicated systems. Moreover, we observed that using different OS releases of the same OS is enough to warrant its adoption as a more straightforward and manageable configuration for replicated systems. The described contributions (together with the ones published during the MSc [68]) are reported in the following publication:

1. **Analysis of Operating System Diversity for Intrusion Tolerance**, Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro, in *Software: Practice and Experience, 2014* [69].

1.2.2 Applying Diversity on BFT Systems

In our first attempt to validate diversity, we presented substantial evidence to claim that using different OSes guarantees failure independence to some extent. However, this preliminary analysis suffer from some limitations as other works using NVD database (e.g., [4, 26, 66, 78, 80, 164]). It is possible to find vulnerabilities in NVD that were not reported to all affected OSes. We have found these missing vulnerabilities reported on other sources (e.g., the vendors' security advisories). Therefore, some of the results of NVD-based studies may provide less accurate conclusions about selecting different software based on common vulnerabilities. Moreover, NVD lacks some relevant data concerning exploits and patches that are relevant when considering vulnerabilities life-cycle.

An alternative (or complementary) way to OTS diversity is to generate diversity through automatic mechanisms (e.g., [5, 160]). However, their produced vulnerability independence is difficult to substantiate [25, 166].

Finally, the existent systems that implement time-triggered recoveries assume that it takes the same time/effort to compromise each replica, by assuming that vulnerabilities are all the same. This assumption is unrealistic, especially when the replicas' diversity is considered [136]. Therefore, tailored methods are required to evaluate the risk of a replicated system becoming vulnerable (i.e., $f+1$ replicas suffer from the same weaknesses).

In this thesis, we address these problems from both a theoretical and practical perspective. We address the problems of *finding evidence for supporting diversity*, *manage diversity in a dependable way*, and *support diversity practical mechanisms*. First, we suppress the limitations of NVD using clustering techniques, that group similar vulnerabilities, which potentially can be affected by (variations of) the same exploit. Moreover, we add additional Open Source Intelligence (OSINT) data sources to build a complete knowledge base about the possible vulnerabilities, exploits, and patches related to the systems of interest. The clusters and the collected data are used to assess the risk of a BFT system becoming compromised by the existence of common vulnerabilities. Once the risk increases, the system replaces the potentially vulnerable replica by another one, to maximize the failure independence of the replicated service. The solution continuously collects data from the online sources and monitors the risk of the BFT in such a way that removes the human from the loop.

We have implemented these contributions in LAZARUS, the first system that automatically changes the attack surface of a BFT system in a dependable way. LAZARUS continuously collects security data from several OSINT feeds to build a vulnerability knowledge base. This data is used to create clusters of similar vulnerabilities. These clusters and other collected attributes are used to analyze the risk of the BFT system becoming compromised by replacing replicas before risk increases too much. The replica's replacement is made automatically, while a new replica is deployed in the BFT group, the replaced node is put on quarantine and updated with the available patches, to be re-used later. These mechanisms were implemented to be fully automated and transparent to BFT systems. Moreover, the current implementation of LAZARUS manages 17 OS versions, supporting the BFT replication of a set of representative applications. The replicas run in Virtual Machines

(VMs), allowing provisioning mechanisms to configure them. We conducted two sets of experiments: The first one demonstrates that LAZARUS risk management can prevent a group of replicas from sharing vulnerabilities over time; The second one, reveals the potential negative impact that virtualization and diversity can have on performance. In this experiment, we evaluated LAZARUS with three BFT use case applications: (1) a Key-Value Store application, (2) SIEVEQ an application-like firewall (see next section), and (3) a BFT ordering service for the Hyperledger Fabric blockchain platform. The overall results show that if naive configurations are avoided, BFT applications in diverse configurations can perform close to our homogeneous bare metal setup.

The contributions of this work resulted in the following publications:

2. **DIVERSYS: DIVErse Rejuvenation SYStem**, Miguel Garcia, Nuno Neves, and Alysson Bessani in the *Simpósio Nacional de Informática (INFORUM), 2012 [71].*

3. **Towards an Execution Environment for Intrusion-Tolerant Systems**, Miguel Garcia, Alysson Bessani, and Nuno Neves, Poster session in the *European Conference on Computer Systems (EuroSys), 2016* [70].

4. **LAZARUS: Automatic Management of Diversity in BFT Systems**, Miguel Garcia, Alysson Bessani, and Nuno Neves – *Submitted for publication*.

1.2.3 BFT Multi-layer Resiliency

LAZARUS implements mechanisms that provide security and dependability for replicated systems. However, not all systems need to be BFT-replicated nor managed with such advanced mechanisms. Some of the most critical applications that can benefit from such management are firewalls. Firewalls are used as the primary protection against external threats, controlling the traffic that flows in and out of a network. Typically, they decide if a packet should go through (or be dropped) based on the analysis of its contents. Most generic firewall solutions suffer from two inherent problems: First, they have vulnerabilities as any other system, and as a consequence, they can also be the target of advanced attacks. For example, NVD [134] shows that there have been many security issues in commonly used firewalls. NVD's reports present the following numbers of security issues between 2010 and 2018:[2] 205 for the Cisco Adaptive Security Appliance; 123 in Juniper Networks solutions; and 50 related to iptables/netfilter. Common protection solutions often have been the target of malicious actions as part of a wider scale attack (e.g., anti-virus software [32], Intrusion Detection System (IDS) [6] or firewalls [35, 36, 102, 174]). Second, firewalls are typically a single point of failure, which means that when they crash, the ability of the protected system

[2] To the date of Jul 12^{th} 2018.

to communicate may be compromised, at least momentarily. Therefore, ensuring the correct operation of the firewall under a wide range of failure scenarios becomes imperative.

The last contribution of this thesis addresses the previously described problems with a new protection system, called SIEVEQ. This system mixes the firewall paradigm with a message queue service. In the last decade, several significant advances occurred in the development of intrusion-tolerant systems. However, to the best of our knowledge, very few works proposed intrusion-tolerant protection devices, such as firewalls. Performance reasons might explain this, as BFT replication protocols are usually associated with significant overheads and limited scalability. Additionally, achieving complete transparency to the rest of the system can be challenging to reconcile with the objective of having fast message filtering under attack. In SIEVEQ, we explore a different design for replicated protection devices, where we trade some transparency on senders and receivers for a more efficient and resilient firewall solution. In particular, we propose an architecture in which critical services and devices can only be accessed through a message queue where application-level filtering is implemented. It is assumed that these services have a limited number of senders, which can be appropriately configured to ensure that only they are authorized to communicate through SIEVEQ. The solution has a fault- and intrusion-tolerant architecture that applies filtering operations in two stages, acting like a sieve. The first stage, called *pre-filtering*, performs lightweight checks, making it efficient to detect and discard malicious messages from external adversaries. In particular, messages are only allowed to go through if they come from a pre-defined set of authenticated senders. Denial-of-Service (DoS) traffic from external sources is immediately dropped, preventing those messages from overloading the next stage. The second stage, named *filtering*, enforces more refined application level policies, which can require the inspection of some message fields or need the enforcement of specific ordering rules.

SIEVEQ was evaluated in different scenarios. The results show that it is much more resilient to DoS attacks and various kinds of intrusions than existing replicated-firewall approaches. We also evaluated SIEVEQ considering the protection of a Security Information and Event Management (SIEM) system. The test environment emulated the setup of the 2012 Summer Olympic Games, where the same sort of security events was generated and transmitted across the network. The experiments demonstrate that SIEVEQ can handle a workload up to sixteen times higher than the observed load in the 2012 Summer Olympic Games, without noticeable degradation in performance.

The contributions of this work resulted in the following publications:

5. **An Intrusion-Tolerant Firewall Design for Protecting SIEM Systems**, Miguel Garcia, Nuno Neves, and Alysson Bessani, in the *Workshop on Systems Resilience in*

conjunction with the IEEE/IFIP International Conference on Dependable Systems and Networks, 2013* [72].

6. **SieveQ: A Layered BFT Protection System for Critical Services**, Miguel Garcia, Nuno Neves, and Alysson Bessani, in *IEEE Transactions on Dependable and Secure Computing, 2018* [73].

Additional contributions. A collaboration with a colleague resulted in a work on Supervisory Control and Data Acquisition (SCADA) system enhanced with BFT techniques. We documented the challenges of building such a system from a "traditional" non-BFT solution. This effort resulted in a prototype, implemented by the colleague, that integrates the Eclipse NeoSCADA [56] and the BFT-SMaRt [20] open-source projects. Although this contribution is out of the scope of the thesis, it is easy to envision the integration of this prototype with Lazarus.

7. **On the Challenges of Building a BFT SCADA**, André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves, in *Proceedings of the International Conference on Dependable Systems and Networks, 2018* [141].

1.2.4 Thesis Statement

We summarize our findings in the following thesis statement:

It is possible to build dependable BFT replicated systems by minimizing the number of replicas' common vulnerabilities through software's diversity. Additionally, it is possible to continuously manage these systems while monitoring OSINT data and deciding when replicas should be diversified as deploying the most dependable configurations.

1.3 Thesis Overview

Chapter 2: Background and Related Work. This chapter provides a background overview of intrusion tolerance. It also presents the most relevant related works on the different areas of intrusion tolerance. In particular, it is focused on its main areas: Byzantine Fault Tolerance, replica rejuvenations, and diversity. Then, it identifies several works that implemented diversity guided by *intuition*, then it covers the works that analyzed vulnerabilities, and finally, presents some solutions that manage intrusion-tolerant systems.

Chapter 3: A Study on Common OS Vulnerabilities. The preliminary evidence that diversity could improve systems' dependability was published before this thesis. However, we have made a few additional contributions which are presented in this chapter. The extended work provides more substantial evidence to support the diversity adoption. Moreover, it shows that with advised choices one can build dependable systems. Although some solutions are (now recognized by us as) naive, they were a significant contribution to push us towards building dependable systems with diversity.

Chapter 4: Diversity Management for BFT Systems. This chapter describes the solution's design that we propose to solve a few of the existing open problems of intrusion tolerance. In particular, we show how to solve data source limitations that affected several works in the past. Moreover, we present a new metric that is used by an algorithm to minimize the risk of BFT replicas becoming vulnerable by common mode failures. In the end, we evaluate the efficacy of our metric and algorithm by executing the algorithm against different configuration strategies that have been used in related works.

Chapter 5: LAZARUS. This chapter presents the LAZARUS implementation. It details the different implementation aspects that we address in Chapter 4. Moreover, we present an extensive performance evaluation of LAZARUS managed systems. In particular, we evaluate the performance of three real BFT applications, one of which is presented in detail in Chapter 6.

Chapter 6: SIEVEQ: A Use Case. This chapter presents SIEVEQ, a firewall-like replicated application that implements a new architecture paradigm. The SIEVEQ architecture provides additional resilient mechanisms when compared with state-of-the-art solutions. To conclude the chapter, we present an evaluation showing the performance and resilience of SIEVEQ when attacked.

Chapter 7: Conclusion and Future Research Directions. This chapter summarizes the results that we achieved within this thesis. Additionally, we identify and briefly describe the possible directions for future research.

2 Background and Related Work

This chapter presents the reasons why intrusion tolerance is needed, and then it describes the most relevant related works in the different areas within this topic. It covers mainly the state-of-the-art of BFT replication, the techniques to rejuvenate replicas, and the benefits of using software diversity on distributed and replicated systems. It also presents a few practical attempts to build systems that comprise these techniques. Finally, it describes some of the works on vulnerability analysis and how such analysis can be used to improve classical intrusion-tolerant systems.

2.1 The Need for Intrusion Tolerance

A realistic way to provide security is to build mechanisms that protect potentially vulnerable systems from attackers. This is particularly important when considering critical systems that attract highly motivated attackers. Here, the focus is on designing mechanisms to make systems safer and resilient despite their number of vulnerabilities and the attacker's power. In the following, we make an overview of the different approaches that can be combined to deal with the challenge of providing security and dependability [12] for critical systems.

2.1.1 Vulnerability Prevention

One of the primary techniques to reduce the attackers' success is to avoid that software vulnerabilities persist throughout the development phases until the software is deployed in production. One way to do that is to detect and remove vulnerabilities during the development and testing stages. Here, we distinguish six different approaches that attempt to prevent software vulnerabilities.

First, the most basic approach is manual analysis, typically developers (named testers) or automatic tools perform a set of tests that validate program inputs and outputs. Often, in this type of analysis, the tester needs to check the results manually. The accuracy of the solution depends on how well the inputs cover all the behavior of the software and if the tester detects the output violations after (a description of this process is presented [186]).

Model checking is another approach, which abstracts the software code and formally verifies the program invariants. However, the most common techniques typically over-simplify the protocols to make them formally verifiable or just verify one or a few versions [33, 109, 137]. Moreover, this process is time-consuming, and prohibitively expensive for most companies, which makes it difficult to scale to complex systems [75]. For example, an OS kernel was formally verified [109], but contrary to the Linux kernel that has more than 20 million lines of code [121] this only has 10k lines of code.

One way to find pieces of code that could harm the correctness of the software is to resort to static vulnerability analyzers. This type of analysis is typically limited as it looks for small parts of the code that alone may not cause major alarm. Additionally, it is exhaustive and depends on the availability of the source code. On the contrary, in the software's reliability area, there are significant advances in automatic and efficient error detection [193]. A particular technique used in security that tackles both coverage and code complexity is taint analysis. In this technique, any program variable that can be modified by a user is seen as a vulnerability trigger. Then, when it is accessed, it becomes tainted for further inspection. The main limitations of taint analysis are that vulnerabilities are detected only for the execution paths that have been explored by the tester, and it is limited to call/return functions being challenging to cover shared memory or global variables [195].

Another conventional solution to explore software bugs is fuzzing. Fuzzing is a dynamic technique, as it runs against executing software, and its success in finding vulnerabilities or bugs results from a good set of input test cases. These can be built manually and tuned for a particular application, or randomly generated, for the latter they are likely to fail on triggering more complex vulnerabilities. The main limitation of fuzzing is the time it takes trying multiple execution paths [67].

There are also solutions that combine some of the previously described techniques like taint analysis and fuzzing. For example, concolic testing is a technique that performs symbolic execution along with a particular execution path [106]. Therefore, it is more accurate than fuzzing but it can, depending on the program size, exponentially explore execution paths. Some of these techniques can be combined to take the best of some of the approaches described before [173].

Finally, some works proposed workarounds that intended to minimize the window of vulnerability between the vulnerability disclosure and the patch release [79, 93]. Typically, these solutions have two phases: first, the detection phase where they look for vulnerabilities in the source code, and a second one, where they instrument the code with a vulnerability workaround. Although they may have good coverage of the vulnerabilities, there are a few caveats on applying such solutions. For example, some workarounds may crash the application upon the vulnerability activation, or they may disable the main functions that clients use in the software [93] compromising the availability.

Although these techniques are steps towards more robust software, they all have limitations on coverage, either they fail to cover the whole code surface, or they fail to detect more complex and unknown vulnerabilities. Moreover, these techniques are especially useful to use before the vulnerable system is in production.

2.1.2 Fault Detection and Removal

Most of the previously described techniques work offline and are not complete. Therefore, unknown vulnerabilities may arise when the system is already online, allowing the system to be compromised. Then, we need more complex approaches to cope with the existence of vulnerabilities that can be exploited. A possible solution is to detect intrusions at runtime and then trigger a recovery mechanism to clear the resulting effects.

Since some attacks employ a combination of different vulnerabilities (e.g., [60]), which isolated would be harmless, it is hard to detect them when the system is in production. We identify three problems on fault detection and removal: First, perfect intrusion detectors are dependent of the application semantics [55], and therefore, stealth attacks may remain active for long intervals; Moreover, typically detection is based on attack signatures or anomaly detection [167], therefore unknown attacks will not be detected so accurately. Second, the service can experience some periods of unavailability while the service is recovering [188]; Finally, recovering a system might clean its faults, but the system remains vulnerable. Therefore, depending on the flow, it could be easy for an attacker to repeat the same procedure to compromise the system every time it recovers.

2.1.3 Fault Tolerance and Masking

The previously described approaches assume that vulnerability removal can be achieved or that it is possible to detect all the attacks. However, these assumptions are unrealistic, and it is not advisable to trust the security and dependability of a sole approach, as it creates a single point-of-failure [182]. Thus, once the system becomes compromised the whole infrastructure becomes exposed to more attacks.

The established way to build a system with tolerance and masking properties is to base its implementation on a group of replicas, which execute equal commands in the same order. Primary-backup replication (i.e., $1+1$ replicas) would suffice if only crash faults are considered. If one of the replicas stops executing, the other can replace it and still deliver the service correctly. However, if arbitrary faults are considered, this form of replication is not enough because compromised replicas could return arbitrary outputs to the clients, impeding the correct answer to be delivered.

State Machine Replication (SMR) [115] is an approach that has been employed to ensure fault tolerance [162] of fundamental services in modern internet-scale infrastructures (e.g., [28, 40, 94]). SMR is achieved in distributed systems that run an agreement protocol that guarantees that all the replica nodes (i) start from the same state, (ii) process an equal sequence of messages, and (iii) execute the same state transitions. These properties guarantee that a service runs in a similar manner in all replicas, and therefore, they all produce the same outputs. However, to address malicious (or sometimes called Byzantine) faults one needs to complement SMR with intrusion tolerance techniques [182].

Intrusion tolerance was first proposed by Fraga and Powell [62] as a solution to address faults without compromising the security of a system. More formally, we adopt the following definition:

Definition 1. *"An intrusion-tolerant system is a replicated system in which a malicious adversary needs to compromise more than f out-of n components in less than T time units to make it fail."* [17]

It is necessary to employ BFT SMR, a particular case of SMR, to build a system capable of operating correctly even in the presence of compromised replicas. Since no single replica can be trusted completely, the system's correctness comes from the majority of correct nodes. For example, to tolerate a single fault (i.e., $f = 1$) the system must have four replicas, where $n \geq 3f + 1$ [30]. Although BFT protocols provide safety to a bound of f faulty nodes, with sufficient time (i.e., greater than T), an adversary could eventually compromise $f + 1$ nodes. Then, additional mechanisms are needed to clean the replicas' faulty state. For instance, to increase the time T that takes to compromise $f + 1$ replicas, one could employ periodic recoveries [30, 53, 147, 160, 170]. However, if a recovered node remains vulnerable to the same attack, the time to compromise $f + 1$ replicas becomes smaller as the attacker already knows how to exploit the existing vulnerabilities. For this reason, several authors have built their systems assuming that nodes fail independently due to some mechanism that provides failure independence (e.g., [18, 30, 170, 183]). Hence, to avoid reintroducing replicas with the same vulnerabilities in the system, the recovery should change the replica's code somehow. Finally, the recovery procedure needs to be managed carefully because otherwise, the decisions that are made with no criteria could lead to a decrease of T. If bad decisions are made, e.g., when $f + 1$ replicas with the same shared weaknesses are deployed, an attacker can replicate the same attack on these replicas to compromise the whole system. Otherwise, an attacker needs to create $f + 1$ different exploits and apply them to $f + 1$ replicas before T.

In the following sections, we present and describe several works on specific areas of intrusion tolerance, covering the various issues that we have just mentioned.

2.2 Byzantine Fault Tolerance

Castro and Liskov's PBFT [29] was the first practical BFT replicated system, and it was initially proposed as a solution to handle faults of both accidental and malicious nature. The correctness of a BFT service comes from the existence of a quorum of correct nodes capable of reaching consensus on the (total) order of messages to be delivered to the replicas. PBFT implements a SMR protocol that guarantees safety for up to $\lfloor \frac{n-1}{3} \rfloor$ faulty replicas out of a total of n. To tolerate a single replica failure, this sort of system typically must have four replicas. This property holds even in asynchronous systems such as the internet.

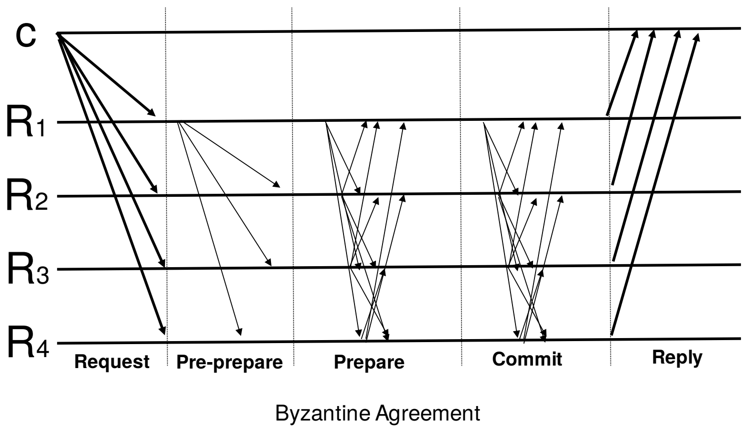


Figure 2.1 – Byzantine Fault Tolerance protocol overview, where *C* represents a client of a replicated service enumerated by replicas *Ri*.

PBFT implemented a SMR, therefore, it guarantees that each correct replica executes the same commands, in the same order, and then it produces the same output. In Figure 2.1, we present an overview of the PBFT protocol that can be summarized as follows: A client *(c)* sends a *request* to all the replicas *(R1-R4)* to execute some service or operation. Then, the leader replica has to assign a sequence number to the request and multicasts a *pre-prepare* message to the other replicas. If the replicas agree with the leader, they send a *prepare* message to each other. At this phase of the protocol, every correct replica agrees on the ordering of the messages. Next, every replica multicasts a *commit* message. When correct replicas receive the *commit* messages from a quorum, then they execute the request deterministically. In the end, they *reply* to the client, which waits for $f+1$ equal responses to guarantee that the correct response is delivered.

PBFT good performance encouraged the use of BFT in common systems, and to develop optimizations to improve BFT protocols. Since the PBFT proposal, many solutions have been dedicated to improve BFT protocols in a different manner. Table 2.1 gives an overview of a few of them.

ZYZZYVA [113]	Introduced speculation to avoid the expensive three-phase commit before processing the requests. This might introduce some inconsistency in the state of the replicas when speculation fails and then it needs the help from the client to fix the problem.
AARDVARK [38]	Shifted the paradigm to a new design that improves the performance under faulty scenarios trading some performance on the normal case. The decision was to design a system that prefers safety over performance on gracious executions. Some counter-intuitive decisions on the architecture favored the overall execution on both gracious and fault scenarios.
UPRIGHT [37]	It provided a straightforward way to support crash fault-tolerant systems, without having to run as many replicas. It combined the protocol of ZYZZYVA and the mechanisms of AARDVARK. It did not require cryptographic signatures as it uses a Message Authentication Codes (MACs) matrix instead.
RITAS [131]	It allows an alternative SMR implementation based on randomized protocols. More precisely, it uses a local coin toss algorithm to reach aggreament among the nodes. This work demonstrated that the *acclaimed inefficiency* of randomized protocols is not verified in practice.
PRIME [5]	It introduced a new *correctness criterion* that derives from bounded delays during a normal execution. They developed a protocol to detect malicious leaders that delay the execution to the limit of the (defined) time out. This protocol ensures that the (correct) non-leader replicas to bound the accepted *delay* of the leader based on messages they receive from the clients.
MINBFT [183]	Reduced the number n from $3f+1$ to $2f+1$ by leveraging on trusted components. These components are deemed as trusted as it hardware is simple and thus verified. Additionally, this solution reduced one communication step compared to other BFT protocols.
BFT-SMART [20]	It was developed to be a modular and multicore-aware. It also supports replica reconfiguration, as it allows that replicas leave and join the replica group. In this process, the joined replicas received the state from the correct replicas in order to be up to date. It also has a flexible programming interface that simplifies its use.
COP [15]	It is one of the most recent BFT implementations, having reached 2.4 million operations per second. This was achieved mostly due to BFT architecture changes, like the creation of *consensus instances* which are assigned to a client or a group of clients. These (separated) *consensus instances* are executed in parallel increasing the throughput of the system.
ABSTRACT [9]	It subsumes some of the works described before in a platform that works as a modular composite of BFT solutions. The main idea is to make use of the several optimizations made by each solution to solve particular problems.

Table 2.1 – Brief overview of some relevant BFT works.

Relationship with this thesis. In this thesis, we propose a control plane for BFT systems. The goal is to guarantee that replicas execute correctly while tolerating malicious failures in a subset of them. The implementation of such BFT protocol is a complex task, namely if it supports mechanisms for state transfer and reconfiguration while ensuring good performance. Therefore, we prefer to rely on existent libraries than to build a new one. [1]

2.3 Replica Rejuvenation

Software rejuvenation was proposed in the 90's [91, 92] as a proactive approach to prevent component performance degradation and failures due to software aging. This solution

[1] Nevertheless, for some contributions (see Chapter 6), we needed to perform some modifications on the chosen library.

was implemented using three components: a watchdog process (`watchd`), a checkpoint library (`libft`) and a replication mechanism (`REPL`). A primary component executes the application and also runs the `watchd` to monitor the application crashes and hangs. The backup component keeps its application inactive and observes the primary node in order to be able to substitute it. Additionally, there is a routine (provided by `libft`) that periodically makes checkpoints and logs messages. These checkpoints are replicated with `REPL` in the backup node. When the primary node crashes or hangs, it is restarted, and if needed the backup takes his place on the execution.

Some years later, proactive recovery was also adopted on BFT systems by Castro and Liskov [30]. In order to support long-running services, the aim is to rejuvenate replicas periodically to eliminate the effects that an attacker could have caused on a faulty node. This mechanism allows that the system remains correct as long as the adversary controls up to f replicas before a recovery. Moreover, PBFT introduced additional mechanisms to guarantee that SMR properties are maintained even with recoveries. In particular, a *state transfer* protocol, which allows that recovered replicas fetch a correct and up to date state from the other (correct) replicas. Additional assumptions were needed to guarantee the PBFT liveness and safety during the recoveries: (i) each replica contains a trusted chip to store its private key, and that can sign and decrypt messages without revealing this key; (ii) the replicas' public keys are stored in a read-only memory; and (iii) a watchdog timer is used to avoid human interaction to restart replicas. The watchdog hands the execution to the recovery monitor, which cannot be interrupted.

An alternative way to implement BFT systems with SMR is to resort to quorum systems. Zhou *et al.* [202] presented COCA, a fault-tolerant online certification authority to be deployed both in a local area network and on the internet. Proactive recovery is also employed to rejuvenate replicas' state including the capacity to refresh the private keys of the replicas. The adoption of quorum systems does not require timing assumptions and, contrary to PBFT, COCA did not rely on trusted components. However, these options may compromise the safety (i.e., f faulty nodes in-between recoveries) of asynchronous systems as shown by Sousa *et al.* [172]. More precisely, Sousa *et al.* demonstrated the need for hybrid-systems that require some level of synchrony to guarantee safety and liveness of proactive recovery systems.

Virtualization was later identified, by Reiser and Kapitza [154] and Distler *et al.* [54], as a useful mechanism to implement proactive recovery. They proposed an architecture, named VM-FIT, that was divided into two parts: an untrusted domain and a trusted domain. The intrusion-tolerant replicated system executes in the untrusted domain where replicas run in a separate VM. VM-FIT executes in a trusted domain, i.e., in the Virtual Machine Manager (VMM). Virtualization provides isolation between the untrusted and the trusted domains. Therefore, the trusted domain can trigger recoveries in a synchronous manner. Moreover, virtualization reduces the service's downtime during recoveries and makes the state transfer between replicas more efficient. The authors implemented this system using the Xen hypervisor. The trusted domain is placed in the hypervisor `Dom0`, and the replicas run in the untrusted VMs, `DomUs`.

Sousa *et al.* [170] dealt with the presence of silent faults improving recovery mechanisms and introducing the Proactive-Reactive Recovery Wormhole (PRRW). It accelerated the rejuvenation process by detecting the faulty replicas behavior and forcing them to recover without sacrificing periodic rejuvenations. This type of technique can only be implemented with synchrony assumptions as the rejuvenations are time triggered [171]. To address this need, the authors proposed a hybrid system model: the payload was an any-synchrony subsystem where the replicas execute, and the wormhole was a synchronous subsystem. The authors implemented this approach using the Xen hypervisor as the wormhole. Zhao *et al.* [201] improved PRRW with an algorithm to schedule the rejuvenations triggered by monitoring the network and CPU/memory usage.

Relationship with this thesis. BFT replication with proactive recovery represents one of the cornerstones of intrusion tolerance. Proactive recovery allows the system to reduce the replicas' vulnerability window by cleaning the faulty states, impeding the attacker to be successful at compromising more than f replicas simultaneously. Nevertheless, BFT replication with proactive recovery guarantees the system's correctness while replicas recover before $f+1$ replicas become faulty. All works on *safe* proactive recovery require the use of a trusted local component on each replica to trigger the periodic recoveries [30, 53, 147, 160, 170]. Some of these works employed hardware timers while others resorted to virtualization to separate the execution domains. In our proposal, we adopt an architecture inspired on [54] and [170]. More importantly, most of these solutions assumed that replicas fail independently without addressing that issue, which is a topic central to our research. The following section reviews diversity as a way to overcome failure independence difficulties.

2.4 Diversity

In all works presented before, the correctness of the BFT system is ensured under the assumption that replicas fail independently. Therefore, it is assumed implicitly (or explicitly) that there is some mechanism that creates different vulnerable surfaces, which would increase substantially the effort to exploit each replica. One practical way to achieve this is to build diverse replicas.

Diversity can be implemented in different manners [51, 116], which may differ in the mean and on the amount of diversity generated, but the goal is the same: to create different attack surfaces (details in the survey [14]). In particular, it decreases the chances of finding a vulnerability that compromises $f+1$ replicas at the same time [30]. In other words, diversity would contribute to avoiding common vulnerabilities among replicas. We present three different approaches to attain diversity: (i) N-version programming consists of designing and/or implementing different versions of the same specification; (ii) Automatic diversity covers implementing different memory organization schemes and compiling distinct binary executables (from the same source); and (iii) OTS diversity comprises the use of different products with similar functionality, but taking advantage of the various implementations that are already available.

2.4.1 N-version Programming

The first work on diversity was published in 1975 by Randell *et al.* [152]. They introduced the idea of using different software designs and implementations as a way to achieve reliability. The idea was to deploy additional and different replicas along the primary replica, and once the replicas have a mismatch result, a spare replica follows up as primary.

Chen and Avizienis [11, 34] defined the notion of N-version programming with the goal of achieving software reliability in redundant systems. The principle was to resort to *N* teams that would develop *N* versions of the same software specification that would be semantically equivalent. Then, the N-version programs would execute and compare the outputs. If the outputs were equivalent, then the result was accepted. These works opened research avenues in the area of security and reliability. In any case, Knight and Leveson [110] made an empirical study that concludes that N-version must be employed with particular care, as it is it may not provide increased reliability guarantees.

2.4.2 Automatic Diversity

The idea of having *N* different teams to develop *N* software versions was not attractive from several perspectives (including the cost), and soon appeared automatic approaches for introducing diversity. Forrest *et al.* [61] suggested randomized program transformations to introduce application diversity. They have made modifications to `gcc`, the GNU C Compiler, in such a way that at compiling time random padding is inserted into each stack frame. The diverse versions are generated therefore during the source code compilation, and these works aim to make the intruder's work more difficult when exploiting buffer overflow vulnerabilities. More recently, a few other works propose the usage of diversity-based compilers to build different executables [112, 147, 160, 189].

Hosek and Cadar [89] proposed VARAN, an N-version execution framework. VARAN is composed of a leader replica and its followers that share a memory ring where they access in parallel. These components are launched by a coordinator that orchestrates and setups the execution environment. Each follower runs a binary that has system calls rewritten differently. This optimization plays a significant performance improvement when compared with other solutions that rewrite the whole program. Another major improvement is the shared ring memory as it allows concurrent access by multiple producers and consumers. Nevertheless, the results show that some system calls when executed in VARAN can introduce overheads (e.g., of 36% for `close` and 240% for `open`) when compared with a native setup. The authors also scale out the number of followers, which, as expected, increases the overhead as they are added. Although VARAN aims for reliability, some principles could be adopted in the security domain after solving some challenges, namely the use of the same address space that would allow return-oriented programming attacks.

BUNSHIN by Xu *et al.* [192] works as classic N-version systems, where the N variants receive the input, execute it, and then a voter checks for divergences in the output. However, BUNSHIN does not create real *N* versions, in fact, it splits the program into parts, and each part (of each variant) executes some security check. In this way, they can reduce the overheads of typical N-variant solutions as the code is practically the same while creating different checks that should capture faulty behavior. Most of the overhead introduced is due to the variants synchronization before the output voting. Nevertheless, the security of the systems holds on the assumption that an attacker cannot easily overpass the sanity checks. Moreover, it is even more challenging to overpass them in all variants, which make sense for more straightforward attacks.

A different approach based on memory address obfuscation (i.e., Address Space Layout Randomization (ASLR)) was proposed by Bhatkar *et al.* [21]. Their solution transformed object files and executables at the link- and load-time, without kernel or compiler modifications. The goal is to ensure that an attack that compromises one target will not succeed on the other targets. Each time the program is executed, its virtual addresses and data are randomized, and therefore the attacker needs to find new ways to exploit memory errors like buffer overflows. However, recent works have shown that solutions like ASLR are still vulnerable to attacks [25, 97, 166, 181].

As it was observed in a survey by Larsen *et al.* [116], it is quite difficult to measure the efficacy of automatic techniques. For example, entropy analysis may consider two versions of a binary as high entropy but, they could be equally vulnerable to a specific attack. Another way to evaluate these solutions is to test them against real attacks, but this would raise coverage issues.

2.4.3 Off-the-shelf Diversity

The previous approaches generate diversity before the software's distribution. OTS diversity, by the other hand, does not need pre-distribution of diverse mechanisms. It relies on the existence of different software components that are ready to be used. There are plenty of products that provide the same functionality and were developed by distinct vendors. In other words, Components Off-the-Shelf (COTS) diversity is like opportunistic N-version programming.

2.4.3.1 Studies

There are a few works that study the diversity of COTS as a way to achieve failure independence. Han *et al.* [80] made a systematic analysis of the effectiveness of using OTS diversity to improve the system's security. First, the authors tried to find if there were software substitutes to provide the same functionality. Then, they determined if the OTS software shared the same vulnerabilities and if so, if the same vulnerability could be exploited with the same attack. In this study, they analyzed more than 6k vulnerabilities from NVD. The results showed that 98.5% of the vulnerable software had substitutes. Moreover, the majority of them did not have the same vulnerabilities or could not be compromised with the same exploit code. It is not expected that a single exploit works in different OSes because each one has a different memory scheme and file system. Even between releases from the same OS, the low-level functions sometimes change across the versions. The study also concluded that 22.5% of the vulnerabilities were present in multiple software components. However, only 7.1% from those vulnerabilities were present in software that offers the same service.

Gashi *et al.* [74] made an experimental evaluation of the benefits of adopting different Structured Query Language (SQL) databases. The authors analyzed bug reports of four database servers (PostgreSQL, Interbase, Oracle, and Microsoft SQL Server) and determined which products were affected by each bug reported. They have found few cases of a single bug compromising more than one server. In fact, there were no coincident failures in more than two of the servers. The conclusion was that OTS database servers' diversity is an effective mean to improve the system's reliability. However, the authors recognize the need for SQL translators to increase the interoperability between servers in the replicated scenario.

In the same line of the Gashi *et al.* , Garcia *et al.* [68] studied the vulnerabilities shared among OTS OSes. Similar to [80], this study was carried out taking as input the vulnerability feeds from NVD. However, this work was only focused on OSes and the dataset comprised 11-years of vulnerability reports. Their goal was to find to what extent different OSes had common vulnerabilities. To do that, they analyzed 2270 vulnerabilities entries manually and classified them into different categories. Then, the authors defined three types of servers: (i) a server that contained most the packages/applications available (Fat server); (ii) a server that did not contain unnecessary applications to a particular service (Thin server); and (iii) a thin server but with physically controlled access (Isolated thin server). They assumed that the third setting was the most advisable for critical systems because additional care is taken to install and setup its software. For each configuration, they compared all the common vulnerabilities in pairs of OSes. As expected, in the Isolated thin server, the number of common vulnerabilities was considerably less than in the other configurations. There was only one vulnerability that was shared among six OSes, two that were shared among five OSes, and 130 that are shared between two OSes. They went further in the study and looked for common vulnerabilities in different versions of the same OS. Even if few OSes are employed, it is possible to achieve vulnerability independence just with different versions. The authors have found evidence that suggests that using OS diversity in a replicated system can improve its dependability.

2.4.3.2 Network diversity

Diversity assignment was also used as a solution to increase the network's resiliency (in particular its connectivity). Newell *et al.* [140] presented their solution as a diversity assignment problem, that although it is an NP-hard to solve, the authors showed that for medium-sized random network graphs, it was feasible to find solutions within an acceptable time. They used mixed-integer programming for medium-sized networks and for larger networks they proposed a fast greedy approach. For the first one, it was possible to achieve optimality, and for the latter an approximation of the optimal. One of the results showed that random diversity assignment is far worse than a criteria assignment (as the one they proposed). Their results showed improvements of diversity on the connectivity (i.e., resiliency). However, their models did not use realistic data, as the authors assumed that the expectancy values for the probability of node compromise were known and independent for each replica.

In the same line, Zhang *et al.* [200] proposed a metric to evaluate the resilience of private networks. The idea was to define what is the least attacking effort to compromise some critical assets of the network (i.e., the work is focused on distributed yet not replicated systems). Therefore, they measured the weakest path to the target that compromises the main asset. The authors proposed three metrics which were evaluated in a simulated environment. None of these metrics used vulnerability data to correlate the existence of common weaknesses, and on the contrary, the metrics used data about the topology, services installed, etc. In the end, the authors made a simulated evaluation of generated graph networks and evaluated their metric against these graphs. The results show that increasing the diversity on the nodes reduced the worm propagation on the network.

Relationship with this thesis. We presented three different techniques that support diversity as a fault tolerance mechanism. Classical N-version programming is the most costly since it needs different developers and additional programs to verify if the automatic translations work. On the contrary, OTS and some automatic diversity (like randomization) are almost for free. The first one can be obtained from components that are ready to be used. The second one is automatic and already supported in most OSes. The study findings are a good sign that diversity can improve a system's dependability. However, diversity still has some gaps that must be addressed to make it a useful building block of intrusion tolerance. For example, how does one measure diversity as a contribution to increasing failure independence?

2.5 Practical Systems with Diversity

2.5.1 Diversity on Fault/Intrusion Detection

Despite the existence of several studies on vulnerabilities, a few works implemented diversity in their systems without evidence to support their diversity decisions. Totel *et al.* [179] proposed an IDS based on design diversity. The authors described an architecture that uses a set of replicated COTS servers, a proxy, and an IDS. The proxy is responsible for forwarding the requests from the client to every COTS server. When the servers reply, the proxy sends the responses to the IDS to be analyzed. The IDS compares the responses from the COTS servers, and if it detects some differences, an alarm is raised. Next, the proxy votes the responses and replies to the clients. The authors developed an algorithm to detect intrusions and tested their solutions against Snort [165] and WebStat [185]. The results showed that using a diverse-COTS service (e.g., HTTP) allows the IDS to deliver fewer alarms without missing the intrusions.

Bairavasundaram and Sundararaman [13] implemented EnvyFS, a file system that uses N-version file systems to guarantee the reliability of stored data. Their solution is able to tolerate file system mistakes (e.g., crash and integrity errors). To keep the system simple, the authors used a virtual layer that abstract the specific file systems underneath, therefore some challenges arise from the different implementation details. Another challenge was to avoid the need for multiple storages as it uses N variants of file systems. They solved this issue by resorting to only one storage layer and the different file systems in between. Each file system executes the operations, and then a voter collects the majority of outputs to the storage layer. The results showed that EnvyFS is able to improve the reliability by leveraging on multiple file systems. However, EnvyFS pays a performance penalty due to waiting for a majority of file systems responses. In most of the times, it took the sum of the time of the three file systems used.

Diversity has also been adopted to build a malware detector. Xu and Kim [191] introduced PlatPal to analyze the behavioral discrepancies of malicious document files on different platforms (e.g.,Windows or Macintosh). PlatPal loads a file in both systems and monitors the execution traces or crashes. In the end, both systems compare the outputs, and an attack is signaled if divergences are observed. The authors focused on a particular application, the pdf Adobe Acrobat Reader. Therefore, besides the internal discrepancies, the correct executions should be very similar. There are some limitations, namely, it failed to detect attacks that needed some human interaction. Although PLATPAL was proposed to detect malicious payloads on files, it vouched (indirectly) for diversity as it showed that an attacker needed to deploy multiple payloads to compromise a system with the same attack (for the majority of the malware samples used).

2.5.2 Diversity on BFT

A few BFT works already adopted practical diversity mechanisms. One of the first proposals was BASE [31], an extension of PBFT [29] that explored opportunistic OTS diversity in BFT replication. The system provided an abstraction layer for running diverse service codebases on top of the PBFT library. The key issue addressed by BASE was how to deal with different representations of the replica's state, allowing a replica that recovers from a failure to rebuild its state from other replicas. BASE was evaluated considering four different OSes and their native Network File System (NFS) implementations: Linux, OpenBSD, Solaris, and FreeBSD. The results showed that performance varied significantly when diversity was considered. Nevertheless, BASE did not address the problem of selecting replicas nor the reconfiguration of the replica group.

Roeder and Schneider [160] proposed a solution that combined an automatic diversity with rejuvenations, calling it Proactive Obfuscation (PO). The idea was to change the application and library code periodically, while preserving the original semantics by employing program transformations, i.e., system call obfuscation reordering, memory randomization, and functions return checks (through an IBM `gcc` patch that inserts and checks a random value after the functions return). Each replica generated its own obfuscated executable from a read-only device containing the "master code" based on time-triggered epochs that a controller initiates in a secure way through a trusted component. All obfuscation mechanisms were implemented and evaluated on OpenBSD 4.0, and their results showed that PO adds little extra overhead to the non-PO execution.

A similar approach by Platania *et al.* [147] adopted a compiler-based diversity for the PRIME BFT library [5] using the MultiCompiler tool [87]. This compiler was used to create diverse binaries from the same source code through randomization and padding techniques. The authors also proposed a theoretical rejuvenation model that received as input: the probability of a replica being correct over a year (c); the number of rejuvenations per day across the whole system (r); the number of replicas (n); and the system's lifetime. Although operators can control only r and n, the authors suggested (but did not show how) that c would be estimated using OSINT from the internet (CERT alerts, bug reports, and other historical data). The authors implemented their solution in two settings, including a virtualized environment provided by the Xen Hypervisor. In this setting, each replica executed in a Linux VM, the recovery watchdog runs in a trusted domain of the same machine, and the proactive-recovery controller runs in another physical machine.

Relationship with this thesis. Diversity is one of the building blocks of intrusion tolerance, alongside with BFT and rejuvenations. We presented a few works that already used in some manner diversity in intrusion-tolerant systems. Although we do not discard the possibility of employing other diversity techniques, such as randomization, we are more interested in COTS diversity. This type of diversity allows us to use data that is freely available to estimate a risk value that measures how vulnerable is each component.

Previous works are still limited to the extent that are none or few concerns on how to create diversity to avoid common failures. They implement diversity assuming (unrealistically) complete fault independence. For example, different OTS OSes can share the same weaknesses due to some shared libraries or kernel code. There is a need to understand how diversity can be efficiently employed in a replicated system to make fault independence sound. Moreover, further work is still necessary to create automatic mechanisms that abstract diversity management for the administrators.

2.6 Vulnerability Analysis

In another line of research, a few works were devoted to analyzing vulnerabilities life-cycle and evolution. Several studies analyzed several data sources that provide data about vulnerabilities, exploits, and patches. In the following, we present several studies on the usage of vulnerability data to measure and manage the security of computer systems.

2.6.1 Vulnerability Life-cycle

The different stages of vulnerability life-cycle were described by Jumratjaroenvanit and Teng-Amnuay [99]. The authors' goal was to understand how the various life-cycle dates can be useful to estimate the probability of an attack. The methodology was to collect and analyze data from public data sources on the discovery-, disclosure-, exploit-date and exploit-, and patch-availability. Moreover, they considered five life-cycle types: (i) Zero-Day Attack (ZDA), which typically is done by a black-hat, and is characterized by equal dates of disclosure and exploit; (ii) Pseudo Zero-Day Attack (PZDA), which is a ZDA but with a patch already available, but not applied; (iii) Potential Pseudo Zero-Day Attack (PPZDA) is a PZDA but does not matter if the patch is available or not; (iv) Potential for Attack (POA) is a zero-day vulnerability. One of the results showed that the language safety level was the less influential factor on the number of accounted vulnerabilities. Thus, assumed safety languages do not impact so much on the number of vulnerabilities as expected. The study also demonstrated that two weeks of work were enough to have a 50% chance of finding a zero-day vulnerability. In some cases, 53 hours were enough to find one vulnerability with more than 95% of probability.

Frei *et al.* [66] made a quantitative analysis of 27k vulnerabilities disclosed in the last decade. The authors proposed a model that explains the key factors in a vulnerability life-cycle. All the dates associated with the vulnerability life-cycle were described in detail before the analysis. A few public vulnerability databases (e.g., NVD and Open Sourced Vulnerability Database (OSVDB)) were used to collect the different data attributes. The results showed that the number of patches increased after the vulnerability's disclosure. One interesting result was the *the gap of insecurity* measurement, where the patch availability was compared against the exploit availability. The data showed that exploits are always ahead of patches. In some cases, after 30 days of the disclosure, the exploits outnumbered the number of patches by 90%.

A study conducted by Shahzad *et al.* [164] analyzed vulnerability data between 1988 to 2011 from three sources: NVD, OSVDB, and the dataset from Frei and colleagues [65]. They collected vulnerabilities that affected popular applications like Internet Explorer, Firefox, and Chrome, and popular OSes such as Windows, Mac OS X, Solaris, and several Linux distributions. From their dataset, 2.8% of the vulnerabilities have an exploit released before their public disclosure. More dramatic is the percentage of exploits that are released on the day of the vulnerability disclosure, which was in the order of 88.2%. It was observed that 9.7% of the vulnerabilities had exploits released after their disclosure. Microsoft and Apple had more exploits before the vulnerability disclosure than the other software providers. This may primarily be because hackers find it more rewarding to exploit these products due to their broader market capitalization. To what concerns patching, the authors argue that 10% of the vulnerabilities had a patch before their disclosure, and 62% had a patch on its disclosure day. There was a considerable number of vulnerabilities (28%) that were disclosed only after a patch was made available. Another conclusion was that it is easier to exploit open-source code vulnerabilities than in closed-source – this conclusion may seem obvious, but no one had shown it previously with statistical tests.

Zero-day vulnerabilities between 2008 to 2011 were part of a systematic study by Bilge and Dumitras [22]. The main dataset comes from the Worldlwide Intelligence Network Environmnet (WINE) developed by Symantec, which samples and aggregates data from several hosts running Symantec products (e.g., Norton Antivirus). The WINE data was correlated with OSVDB and Symantec's Threat Explorer for attack/exploit information, allowing the measurement of the duration of zero-day attacks. They conclude that a typical attack can last ten months. Moreover, they found that the vulnerabilities exploited in Stuxnet were exploited in a different attack two years before. One of the main conclusions was that the disclosure of zero-day vulnerabilities increases the risk, up to five orders of magnitude, of users being attacked.

Holm [86] made an analysis of malware alarms to verify if there is a statistical distribution to model the number of intrusions or the time to compromise a computer system. The author collected information from different software configurations from 2009 to 2012 from an IT company. Each computer was equipped with anti-malware software, then upon malware detection related logging data is sent to a central database. The results showed that the Pareto distribution is the best to model the time of the first compromise. Another result was that the more the system is intruded, the more vulnerable it becomes. The author presented a few hypotheses to justify this, for example, once a system was compromised it can no longer be trusted. Another more plausible hypothesis is that security awareness only increased temporarily as a result of the intrusion, and then it is again neglected. Automatic attacks caused most of the malware considered in this study. This probably happens due to the type of company. Conclusions may change if malware come due to targeted attacks as they require a different strategy to penetrate the system.

2.6.2 Metrics and Risk Management

Poolsappasit *et al.* [148] described a security risk assessment method that incorporated the cause-effect relationships of the network states and the likelihood of exploiting such relationships. To do that, they measured the organizations' security risk using the Common Vulnerability Scoring System (CVSS) metric. The authors generated a map of the network as an attack graph and collected its vulnerabilities and exposures. This information helps the administrators to have an overview of the network weaknesses and the associate a cost to the assets. Then the administrator is able to build the countermeasures can be taken to avoid such weaknesses. The countermeasures are indicated by the system administrator that will link them to the attributes model. Then, the administrator assigns the damage costs to every attribute. Finally, a genetic algorithm evolves the network for states where the cost of losing assets is minimal. Their empirical results show the costs of being compromised can be minimized by assessing the security of the system and applying the most cost-efficient countermeasures. Nevertheless, and besides the manual setup that the system administrator needs to perform, an evaluation on the time the system reaction to detect attacks is missing.

The patch deployment process of ten popular client applications was analyzed over a five-year period by Nappa *et al.* [133]. The authors have found that there was a strong correlation between the vulnerability disclosure and the beginning of the patching process among vendors. It occurred within seven days for 77% of the studied vulnerabilities. A *survival analysis* (inspired in studies that measure the mortality rates associated with diseases) was performed to measure the probability that a vulnerable host remains vulnerable beyond a specific time. The experimental results showed that real-world WINE exploits still compromised 50% of the vulnerable hosts (after vulnerabilities disclosure). The median percentage of hosts that have been patched before the exploit was released was at most 14%. Together with the number of shared vulnerable code with different patching delays, these values motivate attackers to re-use already known exploits or to create patch-based exploits [27]. The authors also showed evidence to support the natural intuition that silent or automatic updates reduce the survivability of vulnerabilities.

Machine-learning techniques were employed by Bozorgi *et al.* [26] to classify vulnerabilities and predict future exploits. Their results showed that their trained classifiers outperform current exploitability measures like CVSS exploitability subscore. They have built a dataset based on two public databases, the Mitre' Common Vulnerabilities and Exposures (CVE) (which is a part of NVD) and the OSVDB. The bag-of-words machine learning technique was utilized to extract the textual attributes that were more relevant to analyze. They also resorted to Support Vector Machines (SVM) to train the model and conduct offline and online experiments. In the offline setting, which they trained and evaluated data in a static fashion, they achieved an accuracy of 90% of correct predictions. In the online setting, where they emulate a live execution with past data, the results showed that after a small period the model stabilizes showing a false negative rate of 9% and a false positive rate of 5%. Here, they were able to make predictions of up to two days before the exploit was released with an accuracy of 75-80%. The main result, for the interest of this thesis, was the comparison between their model and the CVSS as an indicator of how much a vulnerability was likely to be exploited. The CVSS assigned high scores to vulnerabilities that did not become exploited.

An in-depth analysis using NVD, Exploit-DB, SYM, and EKITS conducted by Allodi and Massacci [4] *breaks down* the CVSS on its several attributes. From this analysis the authors concluded that the CVSS exploitability subscore *resembled more a constant than a variable*, thus not having a real impact on the CVSS overall score. In this study, they measured how CVSS influences the *risk factor* of the study cases and assessed the effectiveness of treatment over a randomized sample of subjects. A result that is worthy of note was that fixing vulnerabilities based on their CVSS was statistically equivalent to picking a random vulnerability to fix from the security relevance standpoint. On the contrary, the authors suggest that it was more critical to fix vulnerabilities that appear in the black market of vulnerabilities (this was re-evaluated in a recent study by the same authors [3]). The results indicated that the inclusion of black market information as a *risk factor* could increase the risk reduction by up to 80%. However, the authors also pointed out that using the EKITS data source may be a threat to the validity of the study, as it is unstructured data and therefore it requires manual analysis.

Using an alternative data source, Sabottke *et al.* [161] presented a study with Twitter data (e.g., specific words, the number of retweets, and information about the users posting these messages) to find information about exploits. A quantitative and qualitative exploration was carried out over the vulnerability-related information disseminated on Twitter. A Twitter-based exploit detector was developed to provide an adequate response in a such short time frame. This allows the security community to foresee the exploit activity before the information reaches the *de facto* disclosure data sources, like NVD and ExploitDB. However, to complement and strengthen their information results, the paper also used sources like NVD, OSVDB, Exploit-DB, Microsoft Security Advisories, and Symantec WINE. Real-world exploits were distinguished from the proof-of-concept exploits, as the latter are by-products of the disclosure process. The real-world exploits are typically not known until a critical zero-day attack occurs. The results showed that with their proposal there was a reasonable level of confidence about the predictions. For example, the SVM classifier set to a precision of 25% could detect the Heartbleed exploits within 10 minutes of their first appearance on Twitter. They also showed that organizations like NVD overestimate the severity of some vulnerabilities (with CVSS) that never become in fact exploited.

Gorbenko *et al.* [78] resorted to the CVE and NVD databases to study vulnerability life-cycles of different OSes. The authors made a particular effort to analyze the relationship between vulnerability discovery and fix, more precisely the time vendors took to patch the vulnerable software. Moreover, they also studied common vulnerabilities among different OSes. The results show that between 2012 and 2016 most of OSes were patched for every vulnerability that was disclosed in that period (the only exception was Ubuntu Server 12.04). Another interesting result was related to *forever-day vulnerability* (i.e., vulnerabilities that are publicly disclosed but never patched). They showed that, for the analyzed period, Ubuntu never had a single day free from vulnerabilities, and Windows and Red Hat had only 12 and 10, respectively. However, the most important results were the implications of CVSS on the *days-of-gray-risk* (i.e., the number of days it takes for a vendor to release a patch). It was observed that there was no obvious implication on the level of CVSS severity and the time it took for a vendor to develop a patch.

Other works, not focused on replicated systems dependability, worked on malware prediction. In particular, RiskTeller [23], predicts the risk of a system becoming infected by some malware. In this work, the authors conducted a study that accounted for 600k machines belonging to 18 organizations. They collected the log data from all machines together with external data (e.g., from NVD). They defined a dataset with 89 features, such as patching behavior, temporal patterns, application categories, and past threat history. RiskTeller was evaluated by using machine learning models to learn and classify the labeled software of each machine. The results show that it can make predictions on the risk of a system being attacked with an accuracy of 96%.

Relationship with this thesis. We presented several works that carried out the risk management in a way to prevent exploits or to take action upon vulnerability detection. There is a lot of relevant information freely available to address the security and dependability of software. In a replicated context this information is even more relevant. First, to react upon vulnerability/exploit disclosures, and second to select what configurations are less vulnerable to common weaknesses. We want to explore the free available data to improve the dependability and security of intrusion-tolerant systems by ensuring the assumption of failure independence with evidence supported by relevant and sound information. More precisely, we are focused on assuring that no more than f replicas become compromised within time T.

2.7 Management of Vulnerable Systems

In the previous sections, we presented a few approaches to support the recovery and diversity of replicas, several of which in BFT replicated systems. In this section, we focus on some works that already combine, even if in a limited way, the idea of diversity and rejuvenations. We review a few works that introduce the idea of risk management (see [198] for a survey).

Reynolds *et al.* [159] presented the HACQIT fault-tolerant architecture. It was build to assure fault tolerance by resorting on backup replicas on separate LANs and allowing only authorized clients through a Virtual Private Network (VPN). Moreover, HACQIT employed redundancy and diversity to detect and mask errors in the system components. The same input is sent to identical nodes, and if the output is different, then some node is considered faulty. Thus it is possible to vote outputs, and the result sent by the majority is considered correct. Moreover, it used a forensic agent that analyzed the logs of failed nodes to prevent bad requests in the future. Finally, it implemented a recovery mechanism that was triggered by the native security auditing of Windows NT/2000. Unfortunately, no experimental evaluation was presented. Nevertheless, HACQIT was one of the first systems to employ all these mechanisms together.

Wang *et al.* [187] proposed an architecture, named SITAR, that aims to build intrusion-tolerant systems with diversity and dynamic reconfigurations. SITAR defends a set of potentially vulnerable COTS servers. The system was protected by an intrusion-tolerant multilayer architecture that used redundancy and security protocols to guarantee that all nodes agree on the decisions. It is composed of three main components: the proxy servers, the acceptor monitors, and the ballot monitors. The proxies receive the requests and audit them based on the current threat level, and then they propagate the messages to the COTS servers. On the way back, the acceptance monitors validate the response and run an agreement protocol to select a final output from the COTS servers. There was also an adaptive reconfiguration module that monitors the other modules by receiving heartbeats and intrusion triggers to take action on removing compromised modules and starting new ones. Noteworthy that no results on diversity were shown.

The same type of solution was applied on cluster deployments. Arsenault *et al.* [8] presented Self-Cleaning Intrusion Tolerance (SCIT), a centralized architecture in which the goal was to recover the elements of a cluster without harming the availability of the overall service. Recovering the cluster's nodes allows for resetting the state and avoiding huge windows of vulnerability. Nevertheless, the SCIT service itself can be harmed. Therefore, the authors leveraged on trusted hardware to offer additional guarantees to the system, namely predictable and continuous operation regardless of the presence of attacks.

Junqueira *et al.* [101] proposed *informed replication* as an approach to design distributed systems that can survive catastrophes. It focused on the diversity of the components to avoid common pathogens that could compromise the entire system. A new system model was adopted to map different attributes of a node. These attributes represented characteristics that should be distinct on each node to avoid common failures. While modeling the system, some biases were introduced. For instance, the authors considered that two different services that used the same port were affected by the same vulnerability, or that the same service running with two different ports were not vulnerable. A heuristic was proposed to build configurations in linear time. The goal was to distribute the OSes and applications in different configurations to minimize the number of common attributes that could be compromised. Two metrics were utilized to select OSes and applications: selecting both randomly and to choose them based on their popularity. The results showed that their solution mitigated the number of pathogens that could compromise an entire configuration as its attributes were selected in order to maximize their difference. The heuristic guaranteed 99% chances of data survival on an attack of single- and double-exploit pathogens, while needing only three and five copies respectively.

More focused on reliability, Zhai *et al.* [199] proposed a system that looks for failure independence on cloud nodes. The system collected and audited data from the nodes to evaluate their independence by identifying potential correlated failures. Then, the different configuration dependencies were analyzed and ranked in risk groups as a way to see how the different dependencies made the system fail. The main limitation of this work was that it depended on the will of cloud providers to share private data to be analyzed by external agents. The authors evaluated the efficacy of their solution on finding the minimal risk groups (with fewer dependencies) on sampling data to make faster decisions while losing some accuracy. The work's main focus was the reliability of cloud systems, thus no evaluation was done on the vulnerabilities of such systems.

Seondong *et al.* [85] described a system that used data from NVD to select the best configuration of diverse OSes and web servers. The algorithm minimized the CVSS of shared vulnerabilities among the various software components. Although their experiments explored different configurations of software stacks, the decisions were based on the CVSS. All the experiments were done in a simulator (i.e., CSIM 20) which made it difficult to understand the performance of the solution in the real world. Moreover, the resilience of the system was evaluated against DoS attacks that affect particular vulnerabilities in the software. Depending on the type of the DoS, it can cause a performance decrease without exploiting any vulnerability. This, however, does not shed light on the ability of the system to prevent $f+1$ replicas from being compromised. In any case, this is the work that most resembles one of the main contributions of this thesis.

2.7.0.1 Moving Target Defenses

An emerging area, called Moving Target Defense (MTD), re-defines some of the already existing solutions with the goal of achieving a continuous change of the attack surfaces. MTD employs techniques like VM migration, different types of diversity, etc. Some authors have made an effort to formalize this sub-area of intrusion tolerance [203].

Okhravi *et al.* [142] presented TALENT, a framework for live migration of critical infrastructures across heterogeneous platforms. The idea was to create a moving target that make advanced targeted attacks harder to succeed. TALENT implemented OS-level virtualization with containers, which allowed the system to migrate the OS between machines periodically or upon detection of malicious activity. The virtualization was implemented with `OpenVZ` and `LXC` for Linux, `Virtuozzo` for Windows, and `Jail` for FreeBSD. The network was also virtualized, more precisely, a second layer of virtualization was used to migrate the IP address from one container to another. Even an established `ssh` session was preserved during the migration. Additionally, the state of the application also had to be migrated by employing a checkpointing technique. When all the programs were checkpointed, the state was saved and then was migrated by mirroring the file system. TALENT also had a sort of risk assessment, called operation assessment, which monitored and adjusted the diverse components using vulnerability information. However, there were almost no details on how the risk was measured and on how to adapt to the threats efficiently.

Hong and Kim [88] classified MTD into three categories, shuffle, diversity and redundancy, and assessed the effectiveness of each one. The three categories were integrated in HARM and it was evaluated how each technique could improve the security of distributed systems. The solution was built on top of VMs, as each one hosting a two to three OSes in it. The three techniques were evaluated separately, and only two OSes were considered in the diversity evaluation. The results show that deploying random diversity did not improve the risk level of the system. In an ideal scenario each host would have at least one VM with a diverse OS. On the other hand, having diverse OS across hosts decreased the connectivity, in case of some OS became compromised. However, these sort of results was not surprising as only two OSes were considered. On the contrary, with more OSes the system would maintain the level of connectivity while assuring the dependability of the hosts.

Relationship with this thesis. We presented a few control systems that manage diverse nodes that are potentially vulnerable and exploitable. Some of the works presented in other sections already combined a few of these mechanisms (e.g., Sousa *et al.* [170] introduce reactive recovery upon detection of faulty behavior). However, only a few have implemented or discussed the real implications of using diversity (with the exceptions like BASE [31] that implemented OTS diversity and assessed its performance overhead). Our thesis studies diversity in two domains: *(i)* how it can improve the dependability of BFT systems and *(ii)* what are the real overheads of implementing diversity on BFT systems.

2.8 Final Remarks

In this chapter, we described the most relevant works that are related to this thesis. We have made background revision of the intrusion tolerance area, and then we described the different topics that support intrusion-tolerant systems. Our main contributions are on BFT systems management, in particular, how to create diverse replicas that effectively are independent and when the replicas need to be recovered. Although we could adopt solutions from artificial diversity, we are more interested in employing OTS diversity. The adoption of such diversity technique allow us to use data to make decisions on which diverse options work better together.

3 A Study on Common OS Vulnerabilities

In this chapter, we present three strategies to select different OSes to be part of a BFT replicated system. These strategies leverage on the data that is available on the vulnerability database developed in a previous work. Contrary to automatic diversity, OTS diversity can be supported by data evidence to understand the correlation between vulnerabilities and products. Therefore, we focus on the diversity of OSes (not only the kernel but the whole product) for three fundamental reasons: (1) by far, most of the replica's code is the OS; (2) such size and importance, make OSes a valuable target, with new vulnerabilities and exploits being discovered every day; and (3) there are many options of OSes that can be used. The promising results confirm the intuition that applying OS diversity implies vulnerability independence to some extent.

3.1 Methodology

We describe the adopted methodology in this section, in particular, we focus on how the dataset was selected, processed, and analyzed.

3.1.1 Data Source

We have analyzed OS vulnerability data from the NVD database [134]. According to National Institute of Standards and Technology (NIST) a vulnerability is defined as follows:

Definition 2. *"[...] A vulnerability is a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source."* [135]

NIST's NVD is the authoritative data source for disclosure of vulnerabilities and associated information. NVD aggregates vulnerability reports from more than 70 security companies, advisory groups, and organizations, thus being the most extensive vulnerability database on the web. All data is made available as Extensible Markup Language (XML) data feeds, containing the reported vulnerabilities on a given period. Each NVD vulnerability receives a unique identifier, in the format CVE-*YEAR-NUMBER*, and a short description provided by the CVE [130]. The Common Platform Enumeration (CPE) [129] provides the list of products affected by the vulnerability and the date of the vulnerability publication. Additionally, the CVSS [42] calculates the vulnerability severity considering several attributes, such as the attack vector, privileges required, exploitability score, and the security properties compromised by the vulnerability (i.e., integrity, confidentiality, or availability).

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

3.1.2 Data Selection

Despite the vast amount of information about each vulnerability available in NVD, for this study, we are only interested in the name, publication date, summary (description), type of exploit (local or remote), and list of affected products. We have collected vulnerabilities reported for 64 CPE [129]. Each one of these describes a system, i.e., a stack of software/hardware components in which the vulnerability may be exploited. These CPE were filtered, resulting in the following information that was stored in our database:

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

In our previous work [68] we manually analysed those 64 CPE and grouped them in 11 OS distributions: *OpenBSD, NetBSD, FreeBSD, OpenSolaris, Solaris, Debian, Ubuntu, RedHat,*[1] *Windows2000, Windows2003* and *Windows2008*. These distributions cover the most used *OS products* of the BSD, Solaris, Linux and Windows families.

The use of an SQL database brought us 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 associating release times and family names to each affected OS distribution. Second, it allows us to modify the CVE fields to correct problems. For instance, one of the problems with NVD is that the same product is occasionally registered with distinct names in different entries; For instance, (*debian_linux*, *debian*) and (*linux*, *debian*) are two (product, vendor) pairs we have found for the Debian Linux distribution. Other users of NVD data feeds previously observed this same problem [158]. Finally, an SQL database is much more convenient to work with than parsing the feeds on demand.

[1] *RedHat* comprises the “old” RedHat Linux (discontinued in 2003) and the newer RedHat Enterprise Linux (RHEL).

OS	Valid	Unknown	Unspecified	Disputed	Duplicate
OpenBSD	153	1	1	1	0
NetBSD	143	0	1	2	0
FreeBSD	279	0	0	2	0
OpenSolaris	31	0	52	0	0
Solaris	426	40	145	0	3
Debian	213	3	1	0	0
Ubuntu	90	2	1	0	0
RedHat	444	13	8	1	1
Windows2000	495	7	28	5	5
Windows2003	56	5	34	3	5
Windows2008	334	0	8	0	3
#distinct vulns.	2270	63	210	8	12

Table 3.1 – Distribution of OS vulnerabilities in NVD.

3.1.3 Filtering the Data

The following steps were taken during the work [68] prior to this thesis, however, for the extended contributions we had to re-do some of the steps to keep the database updated. From both works, we selected 2563 vulnerabilities from a total of more than 44,000 vulnerabilities published by NVD. These vulnerabilities are the ones classified as OS-level vulnerabilities ("/o" in their CPE) for the OSes 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 precisely 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 that the vulnerability exists, and *Duplicate*, used for vulnerabilities in which the *summary* points to a duplicate entry or duplicate entry suspicion. Due to the uncertainty that surrounds these vulnerabilities, we decided to exclude them from the study as well.

Table 3.1 shows the distribution of these vulnerabilities across the analyzed OSes, together with the total number of valid vulnerabilities.

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 OSes and are counted only once. 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. We should remark that this manual filtering was necessary to increase the confidence that only valid vulnerabilities were used in the study.

3.2 OS Diversity Study

In the context of this thesis, we are focused on the dependability of replicated systems. The starting point of this analysis are vulnerabilities that were shared in OS pairs. In this study, we consider two possible server machine setups offering an increasingly more secure platform and we cover the vulnerabilities in the period of 1994 to 2011. This accommodates the case where system administrators create differentiated OS installations, which contain more or less vulnerabilities depending on the services and applications available in the server. Of course, other setups could be used, but we decided to concentrate on these configurations because they are quite generic and they lead to results that can be directly obtained from the NVD data. The setups are the following:

- **Fat Server:** The server contains most of the software packages for a given OS, and consequently, it can be used to run various kinds of applications by locally or remotely connected users. This server has potentially all vulnerabilities that were reported for the corresponding OS; Therefore, it provides an upper bound estimate on the number of vulnerabilities that can be exploited.

- **Isolated Thin Server:** This configuration corresponds to a platform that does not contain any applications (except for the replicated service). The server has a decreased security risk because the attack vectors related to applications have been mostly eliminated. Moreover, it is placed in a room physically protected from illegal accesses, with remote logins disabled, and therefore it can only be compromised by receiving malicious packets from the network. In this setting, we only consider remotely exploitable vulnerabilities (those with "Network" or "Adjacent Network" values in their CVSS_ACCESS_VECTOR field) that are not classified as *Applications*. Thus, this configuration provides a lower bound estimate on the number of common vulnerabilities that can be exploited.

In practice, when considering the second configuration, at least one application (e.g., name service or a distributed file service) would be deployed in an intrusion-tolerant system, and the vulnerabilities of this application would add up to the flaws that we will report. In any case, since we expect that most of the complexity is in the OS, we anticipate that a single application will have a small contribution to the overall number of vulnerabilities.

Intrusion-tolerant systems are usually built using four or more replicated servers [30]. Therefore, it is useful to understand if there are many vulnerabilities that involve groups of OSes larger than two – if this is the case, then it will be hard to find OS configurations for the various servers that do not share vulnerabilities, and a goal of using OTS diversity will be difficult to be achieved in practice. Figure 3.1 portrays the number of common vulnerabilities that exist simultaneously in increasingly more extensive sets of OSes (with

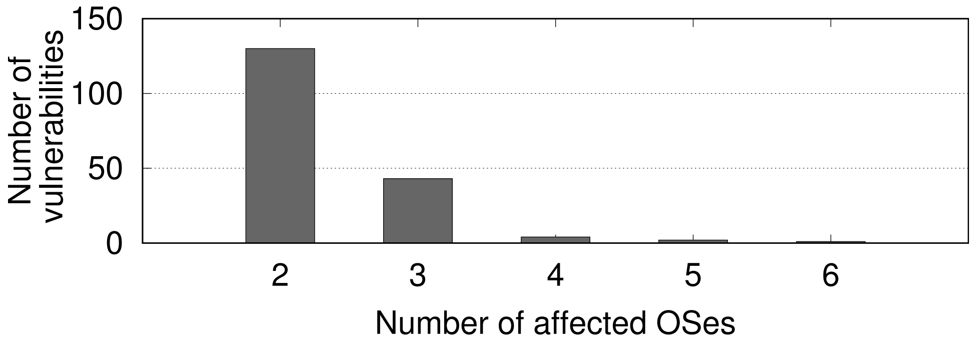


Figure 3.1 – Common vulnerabilities for *n* different OSes (with Isolated Thin Servers).

Isolated Thin Servers). The graph shows a rapid decrease in the shared vulnerabilities as the number of OSes grows. The largest group that was affected by the same vulnerability had six OSes, and this occurred only for a single flaw, and there were also two vulnerabilities in sets with five OSes. Vulnerabilities in two and three OSes usually occur in systems from the same family, whose common ancestry implies the reuse of more significant portions of the code base.

Table 3.2 describes in more detail the vulnerabilities shared by larger groups (four to six) of OSes. The first three vulnerabilities have a considerable impact because they allow a remote adversary to run arbitrary commands on the local system using a high-privilege account. They occurred either in widespread login services (telnet and rlogin) or in a primary system function, and consequently, several products from the BSD family were affected, as well as Solaris. The NVD entry for the CVE-2001-0554 vulnerability also had external references to the Debian and RedHat websites, which could indicate that these systems might suffer from a similar (or the same) problem. Vulnerability CVE-2008-1447 occurs in a large number of systems because it results from a bug in the BIND implementation of the DNS. Since BIND is a highly popular service, more OSes could potentially be affected. A closer look at the corresponding NVD entry reveals external references to the sites of OpenBSD, NetBSD, and FreeBSD, indicating that they could be vulnerable in case this software is being used. This sort of vulnerability confirms that from an intrusion-tolerance perspective it is unwise to run the same server software everywhere, and that diverse implementations must be selected (in this case for recursive DNS servers).

The remaining three vulnerabilities are related to the protocol stack of TCP/IP. All of them affect system availability, allowing different forms of DoS attacks. CVE-2008-4609 is the vulnerability that affects more OSes according to the NVD database, because TCP/IP stack code is often reused across OSes.

Overall, the above results look encouraging because over a significant period (around 18 years) there are very few vulnerabilities that appear in many OSes. A good portion of them are in the TCP/IP stack implementation, which is probably the most shared software

CVE	#affected OS	Description
CVE-1999-0046	4	Buffer Overflow of *rlogin* allows admin access. Affects: NetBSD, FreeBSD, Solaris and Debian.
CVE-2001-0554	4	Buffer overflow in telnetd (telnet daemon) allows remote attackers to execute arbitrary commands. Affects: OpenBSD, NetBSD, FreeBSD and Solaris.
CVE-2003-0466	4	Off-by-one error in the Kernel function *fb_realpath()* allows admin access. Affects: OpenBSD, NetBSD, FreeBSD and Solaris.
CVE-2005-0356	4	TCP implementations allow a denial of service (DoS) via spoofed packets with large timer values, when used with Protection Against Wrapped Sequence Numbers. Affects: OpenBSD, FreeBSD, Windows2000 and Windows2003.
CVE-2008-1447	5	The BIND 8 and 9 implementation of the DNS protocol allow attackers to spoof DNS traffic via a birthday attack to conduct cache poisoning against recursive resolvers. Affects: Debian, Ubuntu, RedHat, Windows2000 and Windows2003.
CVE-2001-1244	5	TCP implementations allow a DoS by setting a maximum segment size very small to force the generation of more packets, amplifying network traffic and CPU consumption. Affects: OpenBSD, NetBSD, FreeBSD, Solaris and Windows2000.
CVE-2008-4609	6	TCP implementation allows a DoS via multiple vectors that manipulate TCP state table. Affects: OpenBSD, NetBSD, FreeBSD, Windows2000, Windows2003 and Windows2008.

Table 3.2 – Vulnerabilities that affect more than four OSes.

component across OSes, but they only impact on the availability of the system, leaving confidentiality and integrity of data unharmed.

3.3 Strategies for OS Diversity Selection

The preliminary analysis presented in the previous section indicates that it is possible to find that some OSes share fewer vulnerabilities than others. Therefore, in this section we present three alternative strategies to select OSes, based on the built dataset, to decrease the chance of common vulnerabilities on replicated systems. We start the section by first describing an approach we called the Common Vulnerability Indicator (CVI), which is used by one of the strategies. For each strategy, we then give example sets of OSes that exhibit a reasonable level of diversity, and perform an evaluation based on the collected data. Finally, we conclude the section with an analysis of potential diversity between releases of the same OSes (concentrating on the OSes that the three strategies picked as the best configuration).

3.3.1 Common Vulnerability Indicator

In our previous work [68] we have found that the number of common vulnerabilities that are observed between different OSes varies over time. Thus, in order to be able to evaluate which OS pairs are more (or less) likely to experience common flaws while taking into account the timing of vulnerability disclosures, we developed a new metric, called the *CVI*. This indicator is calculated for a given year y, based on the vulnerabilities that were shared by OSes A and B over a period of *tspan* previous years. CVI is built to ensure the following desirable properties:

1. $CVI_y(A,B) = 0$ if A and B have no common vulnerabilities in the *tspan* interval before year y;

2. $CVI_y(A,B) < CVI_y(C,D)$ if A and B shared less vulnerabilities than C and D in each year of the considered period;

3. $CVI_y(A,B) < CVI_y(C,D)$ if A and B had N common vulnerabilities in the distant past while C and D had N shared vulnerabilities more recently;

4. $CVI_{y_2}(A,B) < CVI_{y_1}(A,B)$, with $y_1 < y_2$, if the number of common vulnerabilities for A and B has decreased over the years.

Therefore, CVI is useful for comparison purposes, allowing the identification of OS pairs that have a smaller number of common flaws, while considering the instant when vulnerabilities were found. This last point is particularly crucial because OSes are continually evolving, potentially getting more (or less) diverse, and consequently, one should take into consideration the time dimension when selecting OS configurations (e.g., OS pairs that have had less common flaws recently are likely better candidates). CVI is computed as follows:

First, a weighting factor α_i is defined for each year $i \in \{y - tspan + 1, ..., y-2, y-1, y\}$ as defined in Equation 3.1:

$$\alpha_i = 1 - \frac{y-i}{tspan} \tag{3.1}$$

	Fat Server			Isolated Thin Server		
OS	2009	2010	2011	2009	2010	2011
OpenBSD-NetBSD	17.1	13.8	14.8	8.4	7.1	6.9
OpenBSD-FreeBSD	22.2	18.0	18.1	14.1	11.6	10.3
OpenBSD-Solaris	4.0	3.1	3.3	1.8	1.4	0.9
OpenBSD-Debian	1.7	1.5	1.4	0.0	0.0	0.0
OpenBSD-RedHat	5.0	4.1	3.2	1.6	1.4	1.1
OpenBSD-Win2000	1.8	1.5	-	1.8	1.5	-
NetBSD-FreeBSD	19.7	18.3	18.8	11.0	9.3	8.6
NetBSD-Solaris	4.9	4.0	4.2	1.5	1.1	0.7
NetBSD-Debian	1.4	0.9	0.5	0.6	0.4	0.2
NetBSD-RedHat	1.8	1.1	0.5	0.6	0.4	0.2
NetBSD-Win2000	1.6	1.4	-	1.6	1.4	-
FreeBSD-Solaris	6.5	5.4	6.3	2.3	1.7	1.2
FreeBSD-Debian	1.3	0.8	0.5	0.0	0.0	0.0
FreeBSD-RedHat	5.6	4.4	3.3	1.7	1.4	1.1
FreeBSD-Win2000	2.3	1.9	-	2.3	1.9	-
Solaris-Debian	1.0	0.8	0.6	0.0	0.0	0.0
Solaris-RedHat	5.3	6.5	5.5	1.0	0.7	0.5
Solaris-Win2000	5.5	5.7	-	1.0	0.7	-
Debian-RedHat	26.1	20.5	15.7	3.2	2.1	1.4
Debian-Win2000	0.9	0.8	-	0.9	0.8	-
RedHat-Win2000	1.8	1.6	-	0.9	0.8	-

Table 3.3 – CVI for the years 2009 to 2011, with *tspan* of 10 years.

Then, CVI is obtained using the number of vulnerabilities $v_i(A,B)$ that appeared in both OSes A and B for every year i from the start of the time span up to reference year y (see Equation 3.2.

$$CVI_y(A,B) = \sum_{i=y-tspan+1}^{y} \alpha_i \cdot v_i(A,B) \tag{3.2}$$

Table 3.3 presents CVI values for the years 2009 to 2011. We have excluded from this analysis OSes with vulnerability information missing for more than one year over the period, to avoid using incomplete data in the calculation of the indicators. Therefore, the following OSes were not considered in Table 3.3: Windows2003, Windows2008, Ubuntu, and OpenSolaris. The CVI values are computed for a *tspan* of 10 years to reflect a reasonable history. It is possible to see that, except for one system (Solaris with FreeBSD/RedHat/Windows2000 in the Fat Server), in all remaining cases CVI shows a decreasing trend. Several of the OSes that have evolved over a considerable period are having less reported vulnerabilities, and this causes a decline in shared vulnerabilities in recent years. For some of the OS pairs the drop in the CVI value is quite significant, becoming almost one-third of the 2009 value (NetBSD with Debian or RedHat). In the Isolated Thin Server case, there are three pairs with $CVI(A,B) = 0$, which shows that they have no shared vulnerabilities during the past 12 years, and are thus particularly good candidates to include in intrusion-tolerant configurations. These systems are Debian with either OpenBSD or FreeBSD or Solaris.

From the CVI values in Table 3.3, it is apparent that OpenBSD and RedHat have become more diverse, in recent years, than RedHat and Solaris, and to make it advisable, from a diversity standpoint, to choose the former OS pair over the latter.

3.3.2 Building Replicated Systems with Diversity

Indicators like CVI can be useful to devise strategies to select diverse sets of OSes. These strategies utilize the data analyzed earlier as a basis to make decisions, by assuming that the information reported by NVD on vulnerabilities can be correlated to the amount of diversity among OSes. Of course, there are some caveats associated with this approach, but the alternatives can be even harder to put in practice, especially if one wants to consider a large number of OSes. For example, for closed systems (e.g., Windows) it is challenging to determine the level of sharing of OS components, and therefore, diversity estimations based on the code cannot be performed. Moreover, even if this estimate could be obtained, there is the risk that it does not reflect the number of vulnerabilities that occur simultaneously in several OSes (e.g., two distinct implementations of the same flawed algorithm are vulnerable).

The first strategy, called Common Vulnerability Count Strategy (CVCst), is based on raw data collected over a considerable interval, and it is the most straightforward approach for selecting OS pairs. It should be used when one wants to treat all vulnerabilities, regardless of the time which they were reported, as equally important to make choices. The second strategy, Common Vulnerability Indicator Strategy (CVIst), uses the CVI described in the previous section to select OS pairs taking into account the incidence of common vulnerabilities over the years. It is indicated when one wants to give greater importance to more recent vulnerabilities. The third strategy, Inter-Reporting Times Strategy (IRTst), follows a different approach from the previous ones, focusing not so much on common vulnerabilities directly, but on the frequency in which vulnerabilities appear in the two OSes. If one wants to give more importance to the time interval between successive reports of common vulnerabilities, this is the best strategy. Since this last criterion complements the previous two, one could explore strategies CVCst and CVIst in conjunction with IRTst.

For every strategy, we present examples of OS sets for the Fat Server and Isolated Thin Server configurations. Fat Server configurations can be pessimistic in the sense that they may account for common vulnerabilities in applications that are not present in the servers, while Isolated Thin Server configurations reflect more accurately the expected setup of dedicated servers in a replicated system. An intrusion-tolerant system usually requires $3f+1$ replicas to tolerate f intrusions (e.g., [30]). Therefore, we will focus on sets with four OSes to deploy a hypothetical replicated system with four replicas, which allows one fault to be tolerated.

One should take into account that Ubuntu and Windows2008 were first released in 2004 and 2008 respectively, so the data for these two OSes was collected for a smaller number of years. Windows2000 is presented in the tables because there are published vulnerabilities until 2010, although it has been gradually replaced by Windows2003 and Windows2008 in the organizations. Consequently, we do not use Windows2000 when choosing the OS sets. We have excluded OpenSolaris from the study because there is data available for only a limited period. In each strategy and configuration, we present two sets: *setCon* is more conservative, since it does not contain Ubuntu and Windows2008; and *setUpdt* is more up-to-date because it can include Ubuntu and Windows2008. When looking at these two sets, one should keep in mind that setCon is selected from a group of OSes for which there is a significant amount of NVD data, which contributes for higher confidence on the result. On the other hand, setUpdt uses OSes with different amounts of NVD data, which can cause small levels of inaccuracy when making comparisons (e.g., in the CVCst, any OS pair featuring Windows2008 has zero common vulnerabilities until 2007). One way to address this would be only to consider vulnerabilities that appear later than 2007 when choosing setUpdt. We opted not to follow this approach because it has the drawback of discarding too much data.

3.3.3 Common Vulnerability Count (CVCst)

The results from the previous section give a strong indication that it should be possible to choose groups of OSes with a few common vulnerabilities over reasonable intervals of time. However, we would like to understand if the data from the NVD database is effective at suggesting these groups of OSes. To address this point, we divided the data into two subsets: the *history period*, comprising the data for the interval between 2000 to 2009, and the *observed period*, from 2010 to 2011. The objective is to employ the historical period to pick the sets of OSes to use on the replicated system (as if the choice was made at the beginning of 2010). Then we use the data for the observed period to verify if these choices would have been adequate, i.e., if they have a small (preferably the smallest) number of common vulnerabilities in this period.

CVCst makes decisions based directly on the empirical data for the number of common vulnerabilities across all OS pairs. This data is displayed in Table 3.4 for OSes with a Fat Server configuration. Numbers to the right and above the diagonal line represent the history period, while numbers to the left and below the line stand for the observed period. For example, the entry corresponding to OpenBSD-RedHat to the right of the diagonal line has the number 10, which means that these OSes shared 10 vulnerabilities between 2000 and 2009. The equivalent entry, but to the left of the diagonal line, is 0 because they had no

	OpenBSD	NetBSD	FreeBSD	Solaris	Debian	Ubuntu	RedHat	Win2000	Win2003	Win2008	2000-2009
OpenBSD	-	33	43	9	2	3	10	3	2	1	
NetBSD	4	-	36	9	4	0	6	3	2	1	
FreeBSD	4	6	-	12	4	2	12	4	3	1	
Solaris	1	1	2	-	2	2	8	8	7	0	
Debian	0	0	0	0	-	14	52	1	1	0	
Ubuntu	0	0	0	0	0	-	27	1	1	0	
RedHat	0	0	0	2	0	0	-	2	2	0	
Win2000	0	0	0	1	0	0	0	-	216	42	
Win2003	0	0	0	1	0	0	0	49	-	53	
Win2008	0	0	0	1	0	0	0	38	229	-	
	2010-2011										

Table 3.4 – History/observed period for CVCst with Fat Servers.

common flaws reported in 2010 and 2011. As expected, OS pairs from the same family had the highest counts of common vulnerabilities. The only case where there were more vulnerabilities in the observed period than the historical period is for the Windows2008–Windows2003 pair, which is explained by the recent release date of Windows2008. It is interesting to notice, however, that most pairs had zero common vulnerabilities in the observed period.

The strategy for building sets of OSes is based on a simple cost function. Given any potential OS pair A and B that could be added to the set, one can perform a lookup in Table 3.4 to determine the pair's number of common vulnerabilities in the historical period. This number corresponds to the cost of adding this OS pair to the group. Similarly, when including a third OS C, it is possible to find in the table the entries for A–C and B–C and take their sum as the cost of integrating C in the group. When building a set with *n* OSes, the total cost is the addition of each individual cost for all combinations of OS pairs. The sets that lead to smaller values of total cost are considered the best choices for deployment in the replicated system. Accordingly, based on the table, the best groups of four OSes are:

- setCon = {*OpenBSD, Solaris, Debian, Windows2003*}, with a total cost of 23;
- setUpdt = {*NetBSD, Solaris, Ubuntu, Windows2008*}, with a total cost of 12.

One, however, should keep in mind that sometimes the total cost may be only an approximation of the actual number of shared vulnerabilities among the OSes in the set, as specific vulnerabilities might be counted more than once. This will likely not be a problem since overcounting vulnerabilities provides a conservative estimate, or an estimate worse than reality. For example, setUpdt only has 11 shared vulnerabilities for a total cost of 12, since

	OpenBSD	NetBSD	FreeBSD	Solaris	Debian	Ubuntu	RedHat	Win2000	Win2003	Win2008	
OpenBSD	-	13	26	5	0	0	3	3	3	1	2000-2009
NetBSD	1	-	18	4	2	0	2	3	1	1	
FreeBSD	1	1	-	6	0	0	3	4	2	1	
Solaris	0	0	0	-	0	0	2	2	1	0	
Debian	0	0	0	0	-	2	8	1	1	0	
Ubuntu	0	0	0	0	0	-	1	1	1	0	
RedHat	0	0	0	0	0	0	-	1	1	0	
Win2000	0	0	0	0	0	0	0	-	81	13	
Win2003	0	0	0	0	0	0	0	4	-	14	
Win2008	0	0	0	0	0	0	0	3	26	-	
	2010-2011										

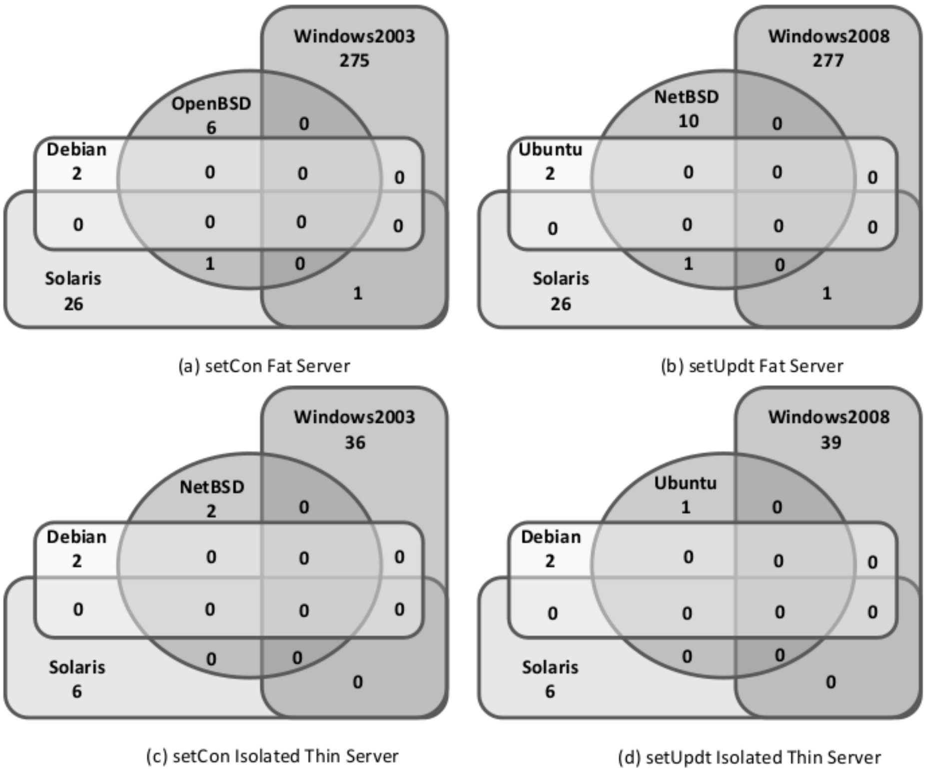
Table 3.5 – History/observed period for strategy CVCst with Isolated Thin Servers.

one of the vulnerabilities appears in three of the OSes (and is therefore included in two table entries).

Next we can check to what extent our choice of the best group of four OSes that we would pick from the historical period (2000–2009), as prescribed by Common Vulnerability Count (CVC) cost calculation, remains consistent with the choice of the best group of four OSes from the observed period (2010–2011). We see that both setCon and setUpdt have only two shared vulnerabilities. They are not the best sets in the observed period (2010–2011), since there are groups of OSes with zero common vulnerabilities in the observed period (e.g., by replacing Solaris with RedHat), though they do exhibit a high level of diversity. A graphical representation of the sets as Venn diagrams is available in Figures 3.2(a) and 3.2(b). Below the OS name is the total number of vulnerabilities during the observed period, and the number inside each intersection shows the count of common flaws for the corresponding OSes. For example, in setCon with Fat Servers (Figure 3.2(a)) there is one vulnerability that appears both on Solaris and OpenBSD and another on Solaris and Windows2003.

Table 3.5 presents the common vulnerabilities with the Isolated Thin Server configuration data. This configuration represents a class of servers that have a dedicated function and are protected against physical intruders. The same approach can be applied as above: first, we choose the best pairs based on the historical period to build a set of four OSes; next, we evaluate the sets by comparing the results with the values for the observed period. The history period provides two candidate sets:

- setCon = {*NetBSD, Solaris, Debian, Windows2003*}, with a total cost of 9;
- setUpdt = {*Solaris, Debian, Ubuntu, Windows2008*}, with a total cost of 2.



(a) setCon Fat Server (b) setUpdt Fat Server

(c) setCon Isolated Thin Server (d) setUpdt Isolated Thin Server

Figure 3.2 – Venn diagrams for vulnerabilities in setCon and setUpdt for strategies CVCst in Fat Server and Isolated Thin Server configurations.

In the observed period, setCon and setUpdt have no common vulnerabilities, showing that the strategy would have chosen sufficiently diverse groups of OSes. A graphical representation of the sets is displayed in Figures 3.2(c) and 3.2(d).

3.3.4 Common Vulnerability Indicator Strategy (CVIst)

This strategy employs the CVI value, defined at the beginning of this section, to make decisions about including/excluding particular OSes. Therefore, besides taking advantage of the available data on total counts of shared vulnerabilities, it also uses the information on how these numbers have evolved through the years.

CVIst is applied by executing the following method, which is based on minimizing a cost function. For a given year and time span, the CVI value is calculated for each of the OS pairs. Typically, one should use the most recent year for which there is available data. The time span should cover a reasonable interval so that the indicator reflects the trend of discovered vulnerabilities. In some cases, however, one may have to resort to smaller time spans due to lack of data, namely with OSes released recently. In this case, the indicator will give a higher weight to the vulnerabilities reported in the last year. In CVIst the cost of creating a group with two OSes A and B is $CVI(A,B)$. Extending this idea to a group of n OSes, the total cost becomes the sum of the individual CVI for all combinations of OS pairs. In

	OpenBSD	NetBSD	FreeBSD	Solaris	Debian	Ubuntu	RedHat	Win2000	Win2003	Win2008
OpenBSD	-	17.1	22.2	4.0	1.7	2.5	5.0	1.8	1.5	0.9
NetBSD	4	-	19.7	4.9	1.4	0.0	1.8	1.6	1.5	0.9
FreeBSD	4	6	-	6.5	1.3	1.3	5.6	2.3	2.0	0.9
Solaris	1	1	2	-	1.0	1.5	5.3	5.5	5.6	0.0
Debian	0	0	0	0	-	10.5	26.1	0.9	0.9	0.0
Ubuntu	0	0	0	0	0	-	18.5	0.9	0.9	0.0
RedHat	0	0	0	2	0	0	-	1.4	1.8	0.0
Win2000	0	0	0	1	0	0	0	-	163.6	39.5
Win2003	0	0	0	1	0	0	0	49	-	0.0
Win2008	0	0	0	1	0	0	0	38	229	-

2010-2011

$CVI_{2009}(A,B)$

Table 3.6 – History/observed period for CVIst with Fat Servers.

order to choose the best groups, the strategy searches for sets of OSes that together have the smallest total cost.

To evaluate this strategy, we split the time into two intervals as we did for CVCst. Table 3.6 presents in the cells for the history period the $CVI_{2009}(A,B)$ for a time span of 10 years in a Fat Server configuration.[2] The cells at left and bottom of the diagonal line correspond to the observed period, and they count as before the number of shared vulnerabilities in 2010 and 2011. After applying CVIst, the following sets are the best with four OSes:

- setCon = {*OpenBSD, Solaris, Debian, Windows2003*}, with a total cost of 14.7;
- setUpdt = {*NetBSD, Solaris, Ubuntu, Windows2008*}, with a total cost of 7.3.

To verify if CVI is a good indicator for selecting diverse sets, we can look at the number of common vulnerabilities in the observed period (2010–2011). By analyzing Table 3.6, it is possible to see that setCon and setUpdt remain good sets, each one with two shared flaws. As before with CVCst, one can find better sets in observed period, where no common vulnerabilities appear for several pairs, for example by replacing Solaris with RedHat. The Venn diagrams for these two sets are shown in Figures 3.3(a) and 3.3(b).

Table 3.7 provides the data for applying CVIst in Isolated Thin Server configurations. It is possible to observe that CVI values have significantly decreased when compared to the previous table. The best groups of four OSes in the historical period are:

- setCon = {*NetBSD, Solaris, Debian, Windows2003*}, with a total cost of 4.5;

[2] In some cases, we had to use smaller time spans due to the more recent release date of the OSes (e.g., Ubuntu and Windows 2008). When this happened, the CVI value was calculated using the maximum time span that is allowed by the available data.

	OpenBSD	NetBSD	FreeBSD	Solaris	Debian	Ubuntu	RedHat	Win2000	Win2003	Win2008
OpenBSD	-	8.4	14.1	1.8	0.0	0.0	1.6	1.8	1.8	0.9
NetBSD	1	-	11.0	1.5	0.6	0.0	0.6	1.6	0.9	0.9
FreeBSD	1	1	-	2.3	0.0	0.0	1.7	2.3	1.5	0.9
Solaris	0	0	0	-	0.0	0.0	1.0	1.0	0.6	0.0
Debian	0	0	0	0	-	1.9	3.2	0.9	0.9	0.0
Ubuntu	0	0	0	0	0	-	0.9	0.9	0.9	0.0
RedHat	0	0	0	0	0	0	-	0.9	0.9	0.0
Win2000	0	0	0	0	0	0	0	-	58.7	12.5
Win2003	0	0	0	0	0	0	0	4	-	13.5
Win2008	0	0	0	0	0	0	0	3	26	-

$CVI_{2009}(A,B)$

2010-2011

Table 3.7 – History/observed period for CVIst with Isolated Thin Servers.

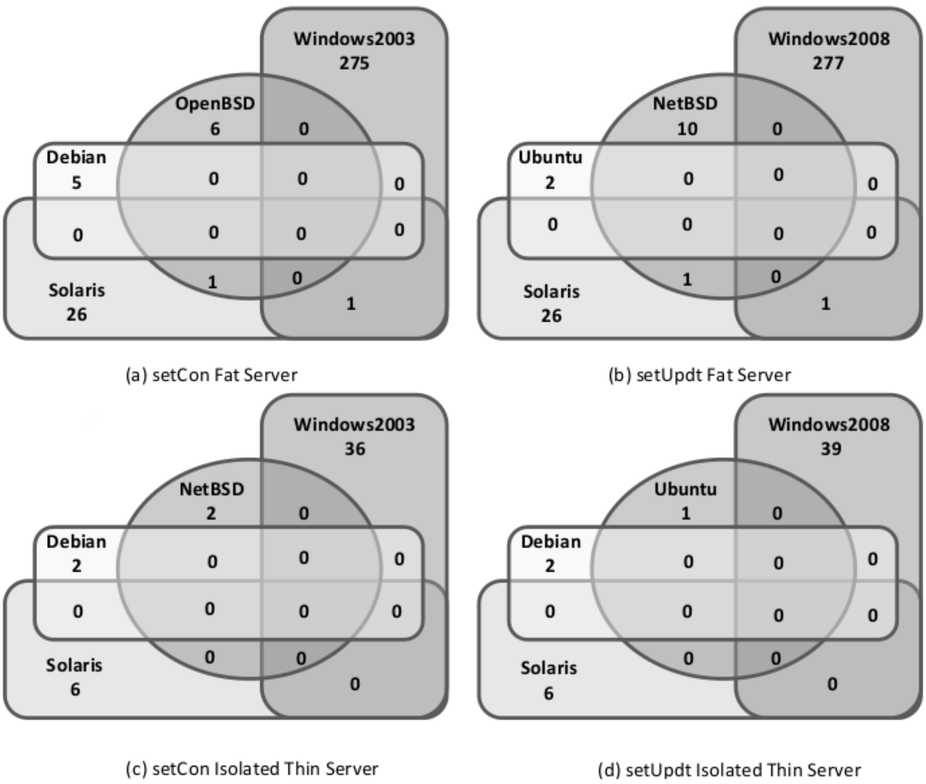


Figure 3.3 – Venn diagrams for vulnerabilities in setCon and setUpdt for strategies CVIst in Fat Server and Isolated Thin Server configurations.

- setUpdt = {*Solaris, Debian, Ubuntu, Windows2008*}, with a total cost of 1.9.

By checking the data for the observed period, one can see that both sets do not share a single vulnerability, which indicates that the strategy would have made a good selection of OSes (see also the Venn diagrams in Figures 3.3(c) and 3.3(d)).

3.3.5 Inter-Reporting Times Strategy (IRTst)

This strategy is mainly concerned with the *IRT*, i.e., the number of days between successive reports of common vulnerabilities in different OS pairs, rather than vulnerability counts. The assumption underlying this strategy is that lower inter-reporting times suggest a greater similarity between OSes, and thus it would be advisable, from a diversity standpoint, to select OSes with higher IRT.

Table 3.8 presents the number of vulnerabilities for each pair of OSes in five IRT intervals, from 0 to 10000 days. The values in the table were obtained in the following manner: first, for a given OS pair A and B, we collected the dates for common vulnerabilities; next, we calculated the IRT in days of every two consecutive vulnerabilities; and then, we counted the number of vulnerabilities that were within each interval. The table is organized such that on the top are the OSes without common vulnerabilities, which are then followed by the ones that had larger IRT values. Therefore, each horizontal line separates groups of OS pairs that have positive IRT values in the same leftmost column, starting from the rightmost column (i.e., with the longer IRT). From a diversity perspective, it is interesting to notice that in the table there are 29% of the pairs that do not have two consecutive vulnerabilities, and that 11% only have consecutive vulnerabilities from 1000 days on (last column).

The IRTst strategy allows the selection of OSes with longer IRT. This criteria is essential if one wants to deploy a system that has a short lifetime, e.g., a batching process, which ideally would only be in operation between the discovery of common vulnerabilities. IRTst tries first to select OS pair with zero common vulnerabilities; when this is not possible, it chooses the next pair whose common vulnerabilities appear in the rightmost columns. By inspecting the table, we can see that the best two sets of four OSes are:

- setCon = {*OpenBSD, Solaris, Debian, Windows2003*};
- setUpdt = {*NetBSD, Solaris, Ubuntu, Windows2008*}.

Table 3.9 presents the IRT for an Isolated Thin Server configuration. Since each OS pair has less common vulnerabilities, this often translates to larger IRT. The percentage of lines with zero IRT in all intervals is higher, 51%, but remained the same for the OS pairs that share vulnerabilities with longer IRT (11%). When applying the strategy to this table, the best sets of OSes are:

- setCon = {*OpenBSD, Solaris, Debian, Windows2003*};
- setUpdt = {*OpenBSD, Debian, Ubuntu, Windows2008*}.

	0 ≤ IRT ≤ 1	1 <IRT ≤ 10	10 <IRT ≤ 100	100 <IRT ≤ 1000	1000 <IRT ≤ 10000
OpenBSD-Win2008	0	0	0	0	0
NetBSD-Ubuntu	0	0	0	0	0
NetBSD-Win2003	0	0	0	0	0
NetBSD-Win2008	0	0	0	0	0
FreeBSD-Win2008	0	0	0	0	0
Solaris-Win2008	0	0	0	0	0
Debian-Win2000	0	0	0	0	0
Debian-Win2003	0	0	0	0	0
Debian-Win2008	0	0	0	0	0
Ubuntu-Win2000	0	0	0	0	0
Ubuntu-Win2003	0	0	0	0	0
Ubuntu-Win2008	0	0	0	0	0
RedHat-Win2008	0	0	0	0	0
OpenBSD-Win2003	0	0	0	0	1
FreeBSD-Win2003	0	0	0	0	1
Solaris-Debian	0	0	0	0	1
RedHat-Win2000	0	0	0	0	1
OpenBSD-Win2000	0	0	0	0	2
OpenBSD-Debian	0	0	0	1	0
Solaris-Ubuntu	0	0	0	1	0
NetBSD-Win2000	0	0	0	1	1
FreeBSD-Win2000	0	0	0	2	1
FreeBSD-Ubuntu	0	0	1	0	0
RedHat-Win2003	0	0	1	0	0
OpenBSD-Solaris	0	0	3	4	2
FreeBSD-Solaris	0	0	5	8	0
FreeBSD-Debian	0	1	0	2	0
OpenBSD-Ubuntu	1	0	0	1	0
NetBSD-Debian	1	0	0	1	0
NetBSD-Solaris	1	0	3	4	1
Solaris-Win2003	1	1	1	4	0
Solaris-Win2000	1	1	1	4	1
NetBSD-RedHat	2	0	0	3	0
Solaris-RedHat	2	0	0	7	0
FreeBSD-RedHat	2	1	2	5	1
Debian-Ubuntu	3	2	3	5	0
OpenBSD-RedHat	4	0	1	4	0
NetBSD-FreeBSD	4	3	20	14	0
OpenBSD-NetBSD	7	3	14	12	0
Ubuntu-RedHat	11	2	7	6	0
OpenBSD-FreeBSD	11	5	16	14	0
Debian-RedHat	22	3	18	8	0
Win2000-Win2008	54	7	15	3	0
Win2000-Win2003	167	29	66	2	0
Win2003-Win2008	222	16	41	2	0

Table 3.8 – Number of two consecutive vulnerabilities occurring in each IRT period, between 2000 and 2011 (with Fat Servers).

3.3.6 Comparing the three strategies

In one hand, the three strategies explore distinct characteristics of the data to pick the OSes to be deployed. Qualitatively they differ in the method of selection, and potentially the result of applying them to our data could lead to distinct sets being chosen. On the other hand, all of them try to find OSes that when placed together in the same system have a low probability of experiencing common vulnerabilities in the future. Therefore, if there is a small collection of OS sets with this property, then all strategies should elect one of these sets as the best choice. This is precisely what we observed with the NVD data, and consequently, the selected best sets are not too different from each other. Only when one needs to find many

	0 ≤ IRT ≤ 1	1 <IRT ≤ 10	10 <IRT ≤ 100	100 <IRT ≤ 1000	1000 <IRT ≤ 10000
OpenBSD-Debian	0	0	0	0	0
OpenBSD-Ubuntu	0	0	0	0	0
OpenBSD-Win2008	0	0	0	0	0
NetBSD-Ubuntu	0	0	0	0	0
NetBSD-Win2003	0	0	0	0	0
NetBSD-Win2008	0	0	0	0	0
FreeBSD-Debian	0	0	0	0	0
FreeBSD-Ubuntu	0	0	0	0	0
FreeBSD-Win2008	0	0	0	0	0
Solaris-Debian	0	0	0	0	0
Solaris-Ubuntu	0	0	0	0	0
Solaris-Win2003	0	0	0	0	0
Solaris-Win2008	0	0	0	0	0
Debian-Win2000	0	0	0	0	0
Debian-Win2003	0	0	0	0	0
Debian-Win2008	0	0	0	0	0
Ubuntu-RedHat	0	0	0	0	0
Ubuntu-Win2000	0	0	0	0	0
Ubuntu-Win2003	0	0	0	0	0
Ubuntu-Win2008	0	0	0	0	0
RedHat-Win2000	0	0	0	0	0
RedHat-Win2003	0	0	0	0	0
RedHat-Win2008	0	0	0	0	0
OpenBSD-Win2003	0	0	0	0	1
FreeBSD-Win2003	0	0	0	0	1
Solaris-RedHat	0	0	0	0	1
Solaris-Win2000	0	0	0	0	1
OpenBSD-Win2000	0	0	0	0	2
Debian-Ubuntu	0	0	0	1	0
NetBSD-Win2000	0	0	0	1	1
FreeBSD-Win2000	0	0	0	2	1
NetBSD-Solaris	0	0	1	2	0
OpenBSD-Solaris	0	0	1	3	0
FreeBSD-Solaris	0	0	1	4	0
NetBSD-Debian	1	0	0	0	0
NetBSD-RedHat	1	0	0	0	0
Debian-RedHat	1	0	1	3	2
OpenBSD-RedHat	2	0	0	0	0
FreeBSD-RedHat	2	0	0	0	0
OpenBSD-NetBSD	2	0	3	7	1
NetBSD-FreeBSD	2	0	6	9	1
OpenBSD-FreeBSD	6	0	9	10	1
Win2000-Win2008	8	1	3	3	0
Win2003-Win2008	16	2	19	2	0
Win2000-Win2003	35	8	36	5	0

Table 3.9 – Number of two consecutive vulnerabilities occurring in each IRT period, between 2000 and 2011 (Isolated Thin Servers).

OS sets with reasonable levels of diversity, such as with the implementation of proactive recovery mechanisms in intrusion-tolerant systems [30, 170], then the distinctions among the strategies start to become apparent.

The OS sets that resulted from the execution of the strategies achieved reasonable levels of diversity when evaluated in the observed period (years 2010 and 2011). As expected from our analysis, several of the best sets contained an OS from each of the families. The exception to this rule occurred with the Linux family, wherein a few cases it had two representatives (Debian and Ubuntu) because these OSes had very few vulnerabilities reported in the last three years. Even though the BSD family also had a small number of recent vulnerabilities, since these occasionally affected more OSes, the strategies opted for including just one of the BSD OSes.

In the next section, we will look into OS versions as a way to increase diversity. For this study, we will use the set that was most selected by the different strategies: {*OpenBSD, Debian, Solaris, Windows2003*} (4 out of 12 choices, considering both Fat Server and Isolated Thin Server configurations).

3.3.7 Exploring Diversity Across OS Releases

If one wants to build systems capable of tolerating a few intrusions, our results show that it is possible to select OSes 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 BFT algorithms that trade off performance for extra replicas (e.g., [1, 113, 163]). The number of available OSes is limited, and consequently, one rapidly runs out of different OSes (e.g., 13 distinct OSes are needed to tolerate $f = 4$ faults in a $3f+1$ system). However, our experiments are relatively pessimistic in the sense that they are based on long periods of time, and no distinctions are made between OS releases.

Newer releases of an OS can contain essential code changes, and therefore, old vulnerabilities may disappear and/or new vulnerabilities may be introduced. As a result, if we consider (OS, release) pairs, one may augment the number of different systems that do not share vulnerabilities.

3.3.7.1 Sets with Four Diverse OSes

The main goal of this thesis is to develop dependable BFT systems. Therefore, one first step towards that goal is to explore the diversity of a particular *4-diverse* set in the Isolated Thin Server configuration. In particular, we analyze in more detail the vulnerabilities for the set {*OpenBSD, Debian, Solaris, Windows2003*} across their releases between 2000 and 2011. Despite the year of the release, since some vulnerabilities can be inherited from older versions of the code, we will include all vulnerabilities no matter the published date (i.e., even flaws discovered before 2000).

From all releases available for our 4-version replicated system,[1] we looked at the major releases that had non-zero vulnerabilities. Since OpenBSD follows a fixed six-month release

[1] OpenBSD 2.7, OpenBSD 2.8, OpenBSD 2.9, OpenBSD 3.0, OpenBSD 3.1, OpenBSD 3.2, OpenBSD 3.3, OpenBSD 3.4, OpenBSD 3.5, OpenBSD 3.6, OpenBSD 3.7, OpenBSD 3.8 OpenBSD 3.9, OpenBSD 4.0, OpenBSD 4.1, OpenBSD 4.2, OpenBSD 4.3, OpenBSD 4.4, Solaris 10.0, Solaris 11.0, Solaris 8.0, Solaris 8.1, Solaris 8.2, Solaris 9.0, Solaris 9.1, Debian 2.2, Debian 2.3, Debian 3.0, Debian 3.1, Debian 4.0, Debian 6.0, Debian 6.2, and Windows 2003.

OS Versions	Total
Solaris 8.0-Solaris 9.0	21
Solaris 9.0-Solaris 10.0	8
Solaris 8.0-Solaris 10.0	7
OpenBSD 3.2-OpenBSD 3.8	4
OpenBSD 3.2-Solaris 9.0	2
OpenBSD 3.2-Windows2003	2
OpenBSD 3.2-Solaris 8.0	1
OpenBSD 3.8-Windows2003	1
Solaris 8.0-Windows2003	1
Solaris 9.0-Windows2003	1
Solaris 10.0-Windows2003	1
Solaris 10.0-Solaris 11.0	1
Solaris 8.0-Debian 2.2	1
Debian 4.0-Windows2003	1

Table 3.10 – Common vulnerabilities between OS releases.

cycle, in this case, we selected one version every three years, which is reasonable given our 12-year time span (considering all 18 releases would require 154 entries in the table). Therefore, the OS releases that are taken into the study are: OpenBSD 5.0, OpenBSD 4.4, OpenBSD 3.8, OpenBSD 3.2, Solaris 8.0, Solaris 9.0, Solaris 10.0, Solaris 11.0, Debian 2.2, Debian 3.0, Debian 4.0, Debian 5.0, Debian 6.0 and Windows2003.

Table 3.10 shows the number of common vulnerabilities for each pair of OS-release (pairs with zero values are not displayed). The first observation is that there are many pairs of releases within this set of 4 OSes that appear to be free of common flaws. Second, these shared bugs occur more often between releases of the same OS. This is anticipated because more code is re-used within the same OS. Additionally, one can notice that releases of the same OS typically have less shared vulnerabilities when comparing older and newer versions. This is particularly evident for the OpenBSD and Debian releases. This result is quite promising because it supports our thesis that one should be able to explore diversity across releases as a way to increase the number of available candidates for the construction of the diverse OS sets.

3.4 Decisions About Deploying Diversity

We have underscored that these results are only *prima facie* evidence for the usefulness of diversity. On average, we would expect our estimates to be conservative as we analyzed aggregated vulnerabilities across releases: common vulnerabilities could be much smaller in a "specific set" of diverse OS releases. However, there are limitations on what can be claimed from the analysis of the NVD data alone without further manual analysis (other than what we have done, e.g., developing/finding and running exploit scripts on every OS for each vulnerability). A better analysis would be obtained if the NVD vulnerability reports were combined with the exploit reports (including exploiting counts), and even better if they also had indications about the users' usage profile. Moreover, we have seen that NVD has some limitations, as we have found that some of the vulnerabilities were not reported in the NVD, but several security advisory websites have reported those vulnerabilities. Additionally, there are partial exploit reports available from other sites (e.g., [158]).

Given these limitations, *how can individual user organizations decide whether diversity is a suitable option for them, with their specific requirements and usage profiles?* The cost is reasonably easy to assess: costs of the software products, the required middleware (if any), added the complexity of management, difficulties with client applications that require vendor-specific features, hardware costs, run-time cost of the synchronization and consistency enforcing mechanisms, and possibly more complex recovery after some failures. The gains in improved security (from some tolerance to zero-day vulnerabilities and easier recovery from some exploits, set against possible extra vulnerabilities due to the increased complexity of the system) are difficult to predict except empirically. In any case, it seems that automated diversity management, described in the next chapter, is needed.

3.5 Final Remmarks

In this chapter, we presented results from an analysis of 18 years (1994–2011) of vulnerability reports from the NVD database. It is worth to notice that the main manual work of analyzing the 2270 vulnerabilities of eleven OS distributions was done before this thesis [68]. In this thesis, we developed three strategies to deploy diverse sets, some of which the ideas will be used in Chapter 4. In particular, the CVIst, as it introduces the notion of time evolution, which is fundamental for the following contributions. In this first chapter of contributions we intended to reinforce the answer to the main questions that drove our research for this thesis: *what are the potential security gains that could be attained from using diverse OSes in a replicated intrusion-tolerant system?*

4 Diversity Management for BFT Systems

This chapter presents the design of a control plane, named LAZARUS, to manage the diversity in BFT services. The design improves the work presented in the previous chapter with new data and clustering techniques, introduces a new metric to evaluate vulnerabilities severity, and propose a strategy to minimize the risk of having common failure modes in replicated systems. In the end, we validate this proposal against other strategies.

4.1 Overview

LAZARUS is a control plane solution that automatically changes the attack surface of a BFT system in a dependable way. LAZARUS continuously collects security data from OSINT feeds on the internet to build a knowledge base about the possible vulnerabilities, exploits, and patches related to the systems of interest. This data is used to create clusters of similar vulnerabilities, which potentially can be affected by (variations of) the same exploit. These clusters and other collected attributes are used to analyze the risk of the BFT system becoming compromised due to common vulnerabilities. Once the risk increases, the control plane replaces a potentially vulnerable replica by another one, trying to maximize the failure independence of the replicated service. Then, the replaced node is put on quarantine and updated with the available patches, to be re-used later.

In a nutshell, as displayed in Figure 4.1, LAZARUS provides a distributed operating system for BFT-replicated services. The system manages a set of nodes that run *unmodified replicas* (encapsulated in VMs or containers) in its execution plane. Each node must have a small Logical Trusted Unit (LTU) that allows the activation and deactivation of replicas as demanded by the LAZARUS *Controller*, in the control plane. The controller decides which software should run at any given time by monitoring the vulnerabilities that potentially exist in the pool of replicas, aiming to minimize the risk of having several nodes being compromised by the same attack.

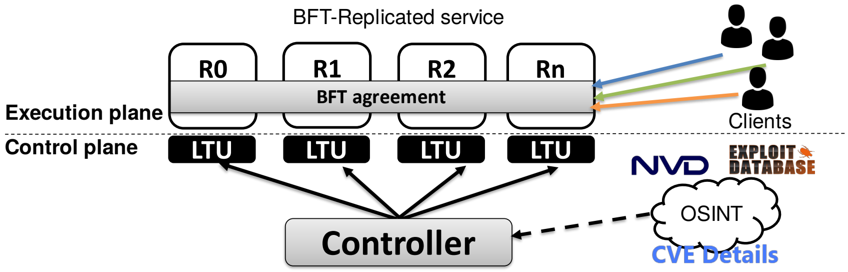


Figure 4.1 – LAZARUS overview.

4.2 System Model

LAZARUS system model shares some similarity with previous works on the proactive recovery of BFT systems [30, 147, 160, 170]. More specifically, we consider a *hybrid system model* composed of two planes (see Figure 4.1) with different properties and assumptions:

- **Execution Plane:** This plane is composed of replica processes that can be subject to Byzantine failures. Therefore, a Byzantine replica can try to mislead the other replicas or the clients. These replicas communicate through an asynchronous network that can delay, drop or modify messages, just like most BFT system models [9, 20, 30, 113]. This plane hosts n replicas from which at most f can be compromised at any given moment. We consider the typical scenario in which $n = 3f + 1$ [9, 20, 30, 113, 131], but the exact resiliency bound is ortogonal to our design.

- **Control Plane:** In this plane, we assume that each component can only fail by crashing. Each *node* hosting processes contains a LTU, and there is a logically-centralized controller to reconfigure the system, just like what has been used in several previous works on proactive recovery (e.g., [147, 160, 170]). The failures of such components do not compromise the liveness and safety of the service as long as the control plane is recovered before f replicas fail. For simplicity, in this chapter, we address this component as a logical-centralized controller, which requires stronger assumptions. However, in Chapter 5 we introduce and discuss a BFT control plane version.

Besides the execution and control planes, we assume the existence of two types of external components: (1) clients of the replicated service, which can be subject to Byzantine failures; (2) OSINT sources (e.g., NVD, ExploitDB) that cannot be subverted and controlled by the adversary. In practice, this assumption led us to consider only well-established and authenticated data sources. Dealing with untrusted sources is an active area of research in the threat intelligence community (e.g., [123, 161]), which we consider out of the scope of this thesis.

4.3 Diversity of Replicas

For our purposes, each *replica* is composed of a stack of software, including an OS (kernel plus other software contained in an OS distribution), execution support (e.g., Java Virtual Machine (JVM), Databases Management Systems (DBMS)), a BFT library, and the service that is provided by the system. The set of n replicas is called a *System configuration*.

The fault independence of the replicas is improved when different OTS components are employed in the software stack [51]. For example, it has been shown that using distinct Databases (DBs) [74], OSes [72], and filesystems [13, 31], can yield important benefits in terms of fault independence. In addition, automatic techniques, like randomization/obfuscation of OSes [160] and applications [189], could enhance diversity. Although LAZARUS can also exploit these automatic techniques, we center our attention on diverse OTS components. In particular, LAZARUS monitors the disclosed vulnerabilities of all elements of the replicas' software stacks to assess which of them may contain common vulnerabilities. As in Chapter 3, we focus on OS diversity here. Moreover, we do not explicitly consider the diversity of the BFT library (i.e., the protocol implementation) or the service code implemented on top of it. Four facts justify this decision: (1) N-version programming is too costly for this [11]; (2) there have been some works showing that such protocol implementations can be generated from formally verified specifications [84, 151]; (3) the relatively small size of such components (e.g., a key-value store on top of BFT-SMaRt has less than 15k lines of code [20]) make them relatively simple to test and assess with some confidence [118, 124]; and (4) there are no reported vulnerabilities about these systems to support our study. Notice that, although we do not explicitly consider the diversity of BFT libraries, nothing prevents LAZARUS from monitoring them (when several alternatives become available). Additionally, as a pragmatic approach, we could employ automatic diversity techniques in this layer [147, 160].

4.4 Diversity-aware Reconfigurations

One of the main contributions of this thesis is the vulnerability evaluation method used to assess the risk of having replicas with shared vulnerabilities. This section details this method.

4.4.1 Finding Common Vulnerabilities

Previous studies on diversity solely count the number of shared vulnerabilities among different OSes, assuming that less common vulnerabilities imply a smaller probability of compromising $f+1$ OSes [69]. Although this intuition may seem acceptable, in practice

CVE (affected OS)	Description
CVE-2014-0157 (Opensuse 13)	Cross-site scripting (XSS) vulnerability in the Horizon Orchestration dashboard in OpenStack Dashboard (aka Horizon) 2013.2 before 2013.2.4 and icehouse before icehouse-rc2 allows remote attackers to inject arbitrary web script or HTML via the description field of a Heat template.
CVE-2015-3988 (Solaris 11.2)	Multiple XSS vulnerabilities in OpenStack Dashboard (Horizon) 2015.1.0 allow remote authenticated users to inject arbitrary web script or HTML via the metadata to a (1) Glance image, (2) Nova flavor or (3) Host Aggregate.
CVE-2016-4428 (Debian 8.0)	XSS vulnerability in OpenStack Dashboard (Horizon) 8.0.1 and earlier and 9.0.0 through 9.0.1 allows remote authenticated users to inject arbitrary web script or HTML by injecting an AngularJS template in a dashboard form.

Table 4.1 – Similar vulnerabilities affecting different OSes.

it underestimates the number of shared vulnerabilities due to imprecisions in the data sources. For example, Table 4.1 shows three vulnerabilities, affecting three different OSes at distinct dates. At first glance, one may consider that these OSes do not share vulnerabilities. However, a careful inspection of the descriptions shows that they are very similar. Moreover, we checked this resemblance by searching for additional information on security websites, and we found out that CVE-2016-4428, for example, also affects Solaris [144].

Even with these imperfections, NVD is still the best data source for vulnerabilities. Therefore, we exploit its curated data feeds to obtain the unstructured information present in the vulnerability text descriptions and use this information to find similar weaknesses. A usual way to find similarity in unstructured data is to use clustering algorithms [96]. Clustering is the process of aggregating related elements into groups, named clusters, and is one of the most popular unsupervised machine learning techniques. We apply this technique to build clusters of similar vulnerabilities (see Section 5.1.2 for details), even if the data feed reports that they affect different products. For example, the vulnerabilities in Table 4.1 will be placed in the same cluster as there is some resemblance among the descriptions, and they can potentially be activated by (variations of) the same exploit.

It is worth to remark that by using clusters to find similar vulnerabilities, we conservatively increase the chances of capturing shared weaknesses contributing to the score of a pair of replicas.

4.4.2 Measuring Risk

Once the set of common vulnerabilities is found, LAZARUS needs to assign a score to each one of these vulnerabilities based on their severity.

Rating	CVSS Score
NONE	0.0
LOW	0.1-3.9
MEDIUM	4.0-6.9
HIGH	7.0-8.9
CRITICAL	9.0-10.0

Table 4.2 – Qualititative CVSS severity rating scale.

As presented in Chapter 3, each vulnerability in NVD has associated a few CVSS severity scores and metrics (two versions are currently offered, v2 [43] and v3 [44]). The scores provide a way to unify several vulnerability attributes in a value reflecting various aspects that impact security. The value can also be translated into a qualitative representation to assist on the vulnerability management process. For example, the CVSS base score categorizes a vulnerability in a severity scale as presented in Table 4.2.

There are some differneces between CVSS v2 and v3. In particular, the latter introduces extra metrics that allow more granularity when assessing the vulnerability characteristics (e.g., it includes *User Interaction*, specifying whether the exploitation of a given vulnerability requires human intervention).

However, CVSS has some limitations that can make it inappropriate for managing the risk associated with a replicated system: (1) it was shown that there is no correlation between the CVSS exploitability score and the availability of exploits in the wild for the vulnerability [26]; (2) CVSS does not provide information about the date when a vulnerability starts to be exploited and when the patch becomes ready; and (3) CVSS does not account for the vulnerability age, which means that severity remains the same over the years [65]; therefore, a very old vulnerability can end up being considered as critical as a recent one, even though for the former there has been plenty of time to update the component and/or the defenses. Additionally, both CVSS specifications support temporal metrics, i.e., Temporal Metric Group. However, this metric group is made to be *updated* by security teams of each asset. Therefore, it is not provided in the NVD feeds. Moreover, the specification shows that CVSS does not account for increasing the base score once there is information about existent exploits. For example, the score of a vulnerability with a known exploit (i.e., Exploit Code Maturity as High) has the same score as a vulnerability with no information about an exploit (i.e., Exploit Code Maturity as Not defined).

Given these shortcomings, we propose a more refined metric that is adapted to measure the risk of a BFT system configuration having replicas with shared vulnerabilities. The metric is mostly focused on differentiating vulnerabilities by their current *potential exploitability*, aiming to surpass the limitations identified above. In this process, (1) and (2) were addressed by using additional OSINT sources that provide information about exploits and dates. We collect data from sources like Exploit-DB [46] for exploits, CVE-details [158] and

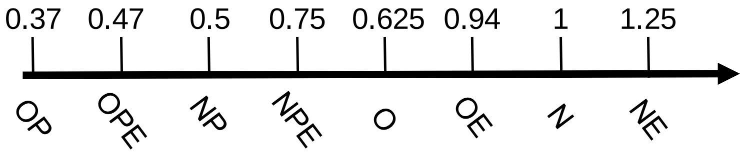


Figure 4.2 – Scoring system of vulnerabilities based on age, patch, and exploit.

vendor websites to get details about the patches, such as Ubuntu Security Notices [180], Debian Security Tracker [47], Solaris [143], Redhat [153], FreeBSD [63], and Microsoft Security Advisories and Bulletins [126]. In fact, often vendor sites also give additional product versions affected by the vulnerability, thus improving the accuracy of the analysis. Limitation (3) is settled by decreasing the criticality of a vulnerability gradually through time.

Our novel metric uses four factors that together contribute to the overall score. The starting factor is the CVSS v3 core score, as it is a reasonable basis that takes into consideration several attributes of the vulnerability. The other three factors adjust the score taking into account the age and the availability of a patch and an exploit. The rationale is to rank vulnerabilities according to their possible exploitation at a given moment in time. The resulting scale is represented in Figure 4.2. The worst scenario (higher severity score) corresponds to a vulnerability that is new (N) (i.e., recently published), for which there is an exploit already being distributed (E), and that is not yet patched – called NE. The best scenario (lowest score) is when a vulnerability is old (O), there is a patch (P), and apparently, no viable exploit has been crafted – named OP. Between these two extremes, several cases of vulnerabilities are considered and their score calculated accordingly.

The metric is defined in Equation 4.1. It is a multiplication of the CVSS v3 core score (as specified in [44]) with the three adjusting factors. The first is *Oldness*, which causes criticality to decrease over time. It is harmonized by the *Oldness Threshold* and the interval[1] that has passed since the vulnerability publication (Equation 4.2). In addition, this factor is bounded by a minimum value that impedes it from reaching zero (which would cause the vulnerability to be left unnoticed). The second is *Patched*, which reduces the severity by half when a patch is available (Equation 4.3; $v_i.patched$ is a on/off flag). Finally, the *Exploited* factor grows severity by a quarter when an exploit is made available (Equation 4.4; $v_i.exploited$ is again a on/off flag). The constants in these equations were defined to ensure the aggregated modifiers corresponds to Figure 4.2. Overall, the range of our metric is from 0.0 to 12.5 (similar to CVSS score $[0.0 - 10.0]$).

Figure 4.3 displays our score and CVSS v3 for three example vulnerabilities. Case (a) corresponds to an NE, i.e., a vulnerability that is new and has no patch yet, but an exploit

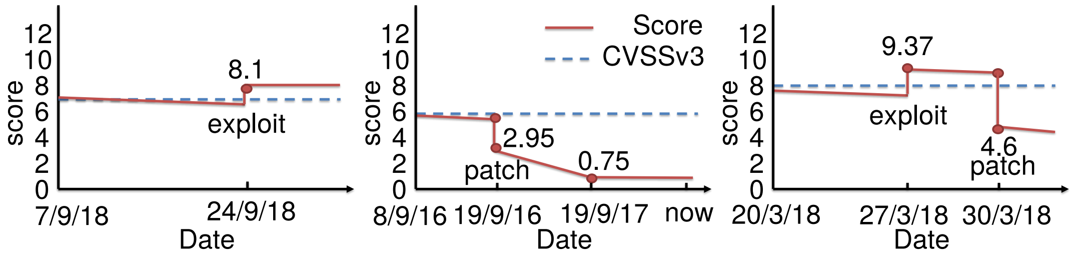
[1] *Oldness Threshold* is set to 365 in our calculations; *now()* and $v_i.published\_date$ return the current day and the day when the vulnerability was published, respectively.

$$\mathrm{score}(\mathrm{v}_i) = \mathrm{CVSSv3}(\mathrm{v}_i) \times \mathrm{oldness}(\mathrm{v}_i) \times \mathrm{patched}(\mathrm{v}_i) \times \mathrm{exploited}(\mathrm{v}_i) \tag{4.1}$$

$$\mathrm{oldness}(\mathrm{v}_i) = \max\left((1 - 0.25 \times \frac{(\mathrm{now}() - \mathrm{v}_i.\mathrm{published\_date})}{\mathrm{Oldness\ Threshold}}), 0.75\right) \tag{4.2}$$

$$\mathrm{patched}(\mathrm{v}_i) = 0.5^{\mathrm{v}_i.\mathrm{patched}} \tag{4.3}$$

$$\mathrm{exploited}(\mathrm{v}_i) = 1.25^{\mathrm{v}_i.\mathrm{exploited}} \tag{4.4}$$



(a) CVE-2018-8303. **(b)** CVE-2016-7180. **(c)** CVE-2018-8012.

Figure 4.3 – Examples of *score(v_i)* for three vulnerabilities.

was made available a few days after publication. Our score starts by decaying slowly but then there is a jump on severity when the exploit is published. Case (b) presents the best scenario (i.e., OP), where the vulnerability is *old* and a patch was eventually distributed (and no exploit is available). Here, the severity decreases once there is a patch and over time, losing its relevance from a security perspective. Finally, case (c) illustrates a vulnerability that has an exploit a few days after publishing and then a patch is also created (i.e., NPE). First, the score is raised once the exploit starts to be distributed, next it decreases three days later after the patch is released, and then continues to decay with the passage of time.

4.4.3 Measuring Configurations Risk

After discovering common vulnerabilities and assigning a score to them, the third step of our method is to calculate the *risk of a set of replicas* to be compromised by the same vulnerability in the current threat landscape.

The risk associated with a *System configuration* (`CONFIG`) with n replicas is given by Equation 4.5. It sums up the score (Equation 4.1) of the vulnerabilities that would allow an attack to compromise simultaneously a pair of replicas $r_i, r_j \in$ `CONFIG`. More precisely, the vulnerabilities in $\mathscr{V}(r_i, r_j)$ aggregate: (i) the vulnerabilities that affect the software running in both replicas as listed in NVD (and other OSINT data); and (ii) groups of vulnerabilities that are placed in the same cluster and affect each replica in the pair.

$$risk(\texttt{CONFIG}) = \sum_{r_i,r_j \in \texttt{CONFIG}} \sum_{v \in \mathscr{V}(r_i,r_j)} score(v) \tag{4.5}$$

This metric penalizes configurations that include replica pairs containing more common weaknesses, as this is an indication that they are less fault independent. In addition, the level of penalty is kept proportional to the severity of these vulnerabilities as observed at the time of the calculation. Thus, for example, replicas that share weaknesses only in the distant past are considered less risky to run in a configuration than replicas that together had recently highly exploitable vulnerabilities.

4.4.4 Selecting Configurations

The last step of our method is to periodically assess the risk of a deployed configuration and its future replacement of replicas according to the evaluation of the metric of the previous section.

The bootstrap of the monitoring algorithm is simple. The first replica is selected randomly from the available candidates (`POOL`). Then, it selects the next n elements by minimizing the risk of the `CONFIG` in construction (using Equation 4.5). After the initialization, the `CONFIG` is periodically monitored and reconfigured if needed, as detailed in Algorithm 1. This is done by periodically evaluating the risk of the current `CONFIG`. If the risk exceeds a predefined *threshold*, a mechanism is triggered to replace replicas and reduce the overall risk. When this happens, the algorithm will pick a new replica from `POOL` to join `CONFIG`. In addition, it removes the replica that will reduce the risk. The removed replica is set aside in a special group (`QUARANTINE`) to impede re-selection. While in `QUARANTINE`, the replicas wait for patches before they re-join `POOL` and be ready to be chosen again. Additionally, the algorithm ensures that all replicas that are running will eventually be replaced, despite their overall score. Otherwise, the algorithm favors configurations that have lower scores by keeping them online for longer periods, allowing an attacker to gain time to discover new vulnerabilities. To avoid this problem, the algorithm looks for the unpatched vulnerabilities with higher score affecting singles replicas. This approach is conservative in the sense that it penalizes replicas expecting that such vulnerabilities can be unknown as common to other replicas.

Function *Monitor()* (line 5) is called on each monitoring round (e.g., at midnight every day) to evaluate the configuration that is executing. If the risk of `CONFIG` is greater or equal than a certain *threshold* (line 6), the algorithm will assess which replica should be

Algorithm 1: Replica Set Reconfiguration

```
1  CONFIG: set replicas in the System configuration;
2  n: number of replicas in CONFIG;
3  POOL: set with the available replicas (not in use);
4  QUARANTINE: set of quarantine replicas;
5  Function Monitor()
6      if risk(CONFIG) ≥ threshold then
7          cadidates_list ← ⊥;
8          COMBINATIONS = (n choose n−1)CONFIG;
9          foreach r in POOL do
10             foreach COMB in COMBINATIONS do
11                 CONFIG’ ← COMB ∪ {r};
12                 score ← risk(CONFIG’);
13                 cadidates_list.add(⟨CONFIG’, score⟩);
14         RAND_CONFIG ← rand(cadidates_list) ;
15         updateSets(RAND_CONFIG);
16     else
17         toRemove ← ⊥;
18         maxScore ← HIGH;
19         foreach r in CONFIG do
20             avgScore ← scoreAVG(r);
21             if avgScore ≥ maxScore then
22                 toRemove ← r;
23                 maxScore ← avgScore;
24         if toRemove ≠ ⊥ then
25             cadidates_list ← ⊥;
26             foreach r′ in POOL do
27                 CONFIG’ ← (CONFIG \ {toRemove}) ∪ {r’};
28                 score ← risk(CONFIG’);
29                 cadidates_list.add(⟨CONFIG’, score⟩);
30             RAND_CONFIG ← rand(cadidates_list) ;
31             updateSets(RAND_CONFIG);
32     foreach r in QUARANTINE do
33         if isPatched(r) = TRUE then
34             QUARANTINE ← QUARANTINE \ {r};
35             POOL ← POOL ∪ {r};
36 Function updateSets(RAND_CONFIG)
37     toRemove ← x ∈ (CONFIG \ RAND_CONFIG);
38     toJoin ← y ∈ (RAND_CONFIG \ CONFIG);
39     QUARANTINE ← QUARANTINE ∪ {toRemove};
40     CONFIG ← (CONFIG \ {toRemove}) ∪ {toJoin};
```

replaced. The first step is to initialize two variables, the candidates list (line 7) and all possible combinations of $n-1$ out of n elements of `CONFIG` (line 8). Then, each element `r` in `POOL` (line 9) is tested as a potential substitute, i.e., as the n^{th} element that would complete each of the combinations `COMB` with $n-1$ replicas (line 10). Next, we define a `CONFIG`’ as the union of `COMB` and r (line 11). The risk of `CONFIG`’ is calculated (line 12) and added to a list that stores the tuple ⟨`CONFIG`’, `score`⟩ (line 13). At the end of the these nested loops, we have a list with all the possible combinations of `CONFIG` and `POOL` together with their risk. Then, the function `rand` selects a configuration randomly from the list of candidates (line 14). Although it is not present in the algorithm, it only selects configurations which are below the *threshold* score. Then the algorithm updates all the sets (line 15) using the function `updateSets` (lines 36-40). To decide which element needs to be removed from `CONFIG` it makes the difference between the current `CONFIG` and RAND_CONFIG (line 37) and the contrary to select the element to join the CONFIG (line 38). Then, toRemove is added to QUARANTINE (line 39), removed from CONFIG, and the new element is added to the CONFIG (line 40).

If the risk of CONFIG is smaller than the *threshold* (line 16) we need to assess if the CONFIG needs a reconfiguration. In this scenario, we start by initializing the variable toRemove as empty (line 17) and maxScore with the value of the CVSS score rating HIGH (line 18). For each element of CONFIG (line 19) the algorithm calculates the average score of the vulnerabilities affecting r using Equation 4.1 (line 20). Then, if the average vulnerability score is equals or greater than HIGH (or CRITICAL) (line 21), the variable toRemove is set to the element r (line 22) and the value of maxScore with the avgScore (line 23). At the end of the loop, the algorithm knows which is the element r from CONFIG that has vulnerabilities with a highest average score. If toRemove is empty then algorithm proceeds to line 32. Otherwise, the algorithm will select a new replica (line 24). First, the candidate_list is set as empty (line 25). Next, for each element r' in POOL (line 26) the algorithm tests a CONFIG' with r' instead of toRemove (line 27). Then, the score for this configuration is calculated (line 28) and together with the CONFIG' is stored in a list of configuration candidates (line 29). In the end, one of the candidate configurations is randomly selected (lines 30-35).

Finally, the POOL and QUARANTINE sets are updated. Each QUARANTINE element (line 32) is checked if it is fully patched (line 33). In the affirmative case, the element is removed from QUARANTINE and added to the POOL again (lines 34-35).

For the sake of simplicity we do not present two corner cases where a *system admin* may need to intervene: if the POOL runs out of elements or *rand()* cannot return a configuration that minimizes the risk of the system. One can take at least two decisions to guarantee the availability under some risk of becoming compromised: (1) increase the *threshold* or (2) reset the POOL by removing the elements from QUARANTINE with fewer unpatched vulnerabilities. Both cases are not likely to occur, but if they happen there is not much we can make to solve as we depend on the availability of replicas in POOL and patch availability to clean QUARANTINE from time to time.

4.5 Evaluation

This section evaluates how LAZARUS performs on the selection of dependable replica configurations. As discussed in Section 4.3, we focus our experimental evaluation solely on the OS diversity. In particular, we considered 21 OS versions to be deployed on four replicas with the following distributions: OpenBSD, FreeBSD, Solaris, Windows, Ubuntu, Debian, Fedora, and Redhat.

In these experiments, we emulate live executions of the system by dividing the collected data into two periods: (i) a *learning phase* covering all vulnerability data between *2014-1-1* and the beginning of the *execution phase*, which is used to setup the *Risk manager*'s algorithm; and (ii) an *execution phase* period that goes from January to August of 2018. This last period is divided into monthly intervals (JAN-AUG) allowing for eight independent tests. For example, to run the experiment in May the *learning phase* starts on *2014-1-1* and ends on *2018-4-30*, then the *execution phase* starts on the May 1^{st} until the end of the month.

The goal is to create a knowledge base in the *learning phase* that is used to assess LAZARUS' choices during each interval of the *execution phase*. A run starts on the first day of an interval and then progresses through each day of the interval until the end. Every day, we check if the currently executing replica set could be compromised by an attack exploring the vulnerabilities that were published in that month. We take the most pessimistic approach, which is to say that we consider the system to be broken if a single vulnerability comes out affecting at least two OSes that would be executing at that time and do not have a patch for it.

Four additional strategies, inspired by previous works, were defined to be compared with LAZARUS (Section 4.4.4):

- **Equal:** all the replicas use the same randomly-selected OS during the whole execution. This corresponds to the scenario where most past BFT systems have been implemented and evaluated (e.g., [5, 9, 15, 16, 20, 38, 113, 122, 183, 197]). Here, compromising a replica would mean an opportunity to intrude the remaining ones.

- **Random:** a configuration of n OSes is randomly selected, and at the beginning of each day, a new OS is randomly picked to replace an existing one. This solution represents a system with proactive recovery and diversity, but with no informed strategy for choosing the next *System configuration*.

- **Common:** This strategy is the straw man solution to prevent the existence of shared vulnerabilities among OSes. This strategy minimizes the number of common vulnerabilities for each set and was introduced in previous vulnerability studies [68].

- **CVSS v3:** This strategy is very similar to ours as it tries different combinations to find the best configuration that minimizes the sum of CVSS v3 score.

4.5.1 Diversity vs Vulnerabilities

We evaluate how each strategy can prevent the replicated system from being compromised. Each strategy is analyzed over 1000 runs throughout the execution phase in monthly slots.

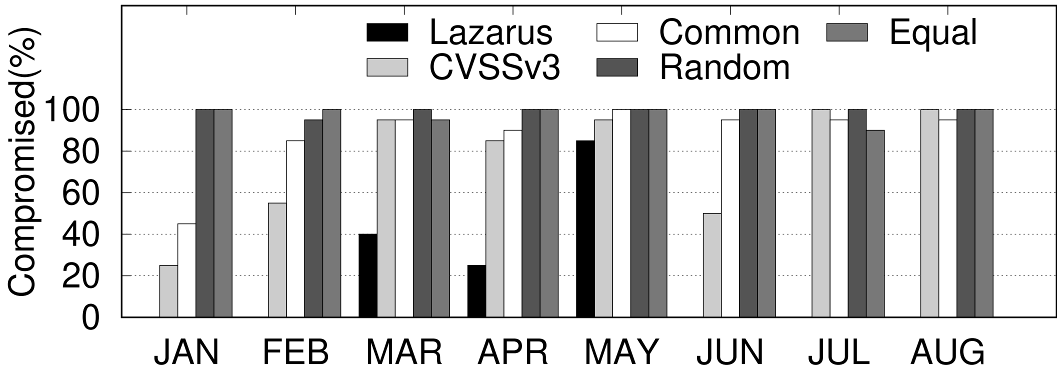


Figure 4.4 – Compromised system runs over eight months.

Different runs are initiated with distinct random number generator seeds, resulting in potentially different OS selections over the time slot. On each day, we check if there is a vulnerability affecting more than one replica in the current *System configuration*, and in the affirmative case the execution is stopped.

Results: Figure 4.4 compares the percentage of compromised runs of all strategies. Each bar represents the percentage of runs that did not terminate successfully (lower is better). In all three periods, LAZARUS presents the best results. The *Random* and *Equal* strategies perform worse because eventually, they pick a group of OSes with common vulnerabilities as they do not make decisions with specific criteria. Although some criteria guide the other strategies, most of them present a majority of executions compromised during the experiments. This result provides evidence for the claim that LAZARUS improves the dependability, reducing the probability that $f+1$ OSes eventually become compromised. Nevertheless, the results from May deserved a more careful inspection. We have identified some CVEs that make it very difficult to survive to common vulnerabilities even using the LAZARUS strategy. For example, CVE-2018-1125, CVE-2018-8897, and CVE-2018-8897 that not only affect a few Ubuntu releases but also affect Debian releases simultaneously. There are also a set of vulnerabilities affecting several Windows releases (e.g., CVE-2018-8134 and CVE-2018-0959). We also found a vulnerability (i.e., CVE-2018-1111) that affects few Fedora releases and one Redhat release that was repeatedly compromising the executions. We have made a more careful inspection of the dispersion of vulnerabilities across these months. Although May is not the month with most vulnerabilities, it has on average 3.05 OSes affected by a vulnerability, and it is the highest. This result was already observed in our previous work (see Figure 3.1, in Chapter 3) where the number of shared vulnerabilities decreases drastically between three affected OSes to four: almost 50 vulnerabilities affecting three OSes and less than ten affects four OSes.

One of the outcomes of this experiment is that, contrary to the intuition, not guided strategies will eventually create unsafe configurations. Therefore, it is paramount to have selection strategies like the ones we use in LAZARUS.

WannaCry [120]	*On Friday, May 12, 2017, the world was alarmed to discover a widespread ransomware attack that hit organizations in more than 100 countries. Based on a vulnerability in Windows' SMB protocol (nicknamed EternalBlue), discovered by the NSA and leaked by Shadow Brokers.*
StackClash [24]	*In its 2017 malware forecast, SophosLabs warned that attackers would increasingly target Linux. The flaw, discovered by researchers at Qualys, is in the memory management of several operating systems and affects Linux, OpenBSD, NetBSD, FreeBSD and Solaris.*
Petya [45]	*Another month, another global ransomware attack. [...] Just as it seemed that the threat of WannaCry has dissipated, organisations around the world are finding themselves under siege from a new threat. This cyberattack first hit targets in Ukraine, including its central bank, main international airport and even the Chernobyl nuclear facility before quickly spreading around the globe, infecting organisations across Europe, North America and even Australia. A day after the incident began, at least 2,000 attacks have been recored across at least 64 countries.*

Table 4.3 – Notable attacks during 2017.

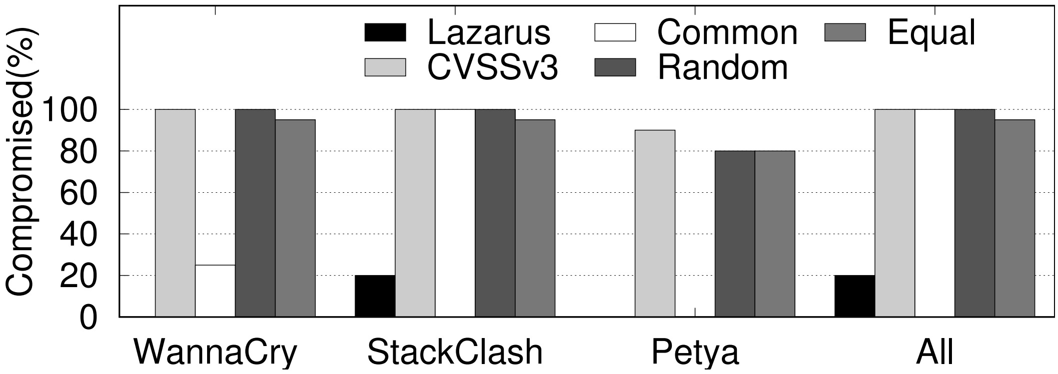


Figure 4.5 – Compromised runs with notable attacks.

4.5.2 Diversity vs Attacks

This experiment evaluates the strategies when facing notable attacks/vulnerabilities that appeared in the last year. Each attack potentially exploits several flaws, some of which affecting different OSes. The attacks were selected by searching the security news sites for high impact problems, most of them related to more than one CVE. As some of the CVEs include applications, we added more vulnerabilities to the database for this purpose. We have considered the attacks described in Table 4.3:

Since some of these attacks might have been prepared months before the vulnerabilities are publicly disclosed, we augmented the execution phase to the full eight months. Therefore, we set *learning phase* to begin *2014-1-1* and to end on *2017-12-31*. The *execution phase* period that starts on 2018 January and finishes on August 2018. As before, the strategies are executed over 1000 runs.

Results: Figure 4.5 shows the percentage of compromised runs for each attack and all attacks put together. LAZARUS is the best at handling the various scenarios, with almost no compromised executions. The StackClash is the most destructive attack as it is the one affecting more OSes. Therefore, decisions guided by criteria that aim to avoid common vulnerabilities may also fail. Nevertheless, the results show that such strategies do improve the resilience to attacks.

4.6 Final Remarks

LAZARUS addresses the long-standing open problem of evaluating, selecting, and managing the diversity of a BFT system to make it resilient to malicious adversaries. This chapter shows an approach to select the best replicas to run together given the current threat landscape and evaluate the proposed strategy. In the next chapter, we discuss how this strategy can be implemented and the overhead of using OS diversity.

LAZARUS

5

This chapter presents LAZARUS iplementation, in its current version it manages 17 OS versions, supporting the BFT replication of a set of representative applications. The replicas run in VMs, allowing provisioning mechanisms to configure them in a fully automated way. We conducted an experimental evaluation that reveals the potential negative impact that virtualization and diversity can have on performance. However, we also show that if naive configurations are avoided, BFT applications in diverse configurations can perform close to our homogeneous bare metal setup.

5.1 Implementation

This section details the implementation of each component of LAZARUS. It also presents other aspects and decisions taken in on building the prototype.

5.1.1 Control Plane

Figure 5.1 shows LAZARUS control plane with its four main modules, described below.

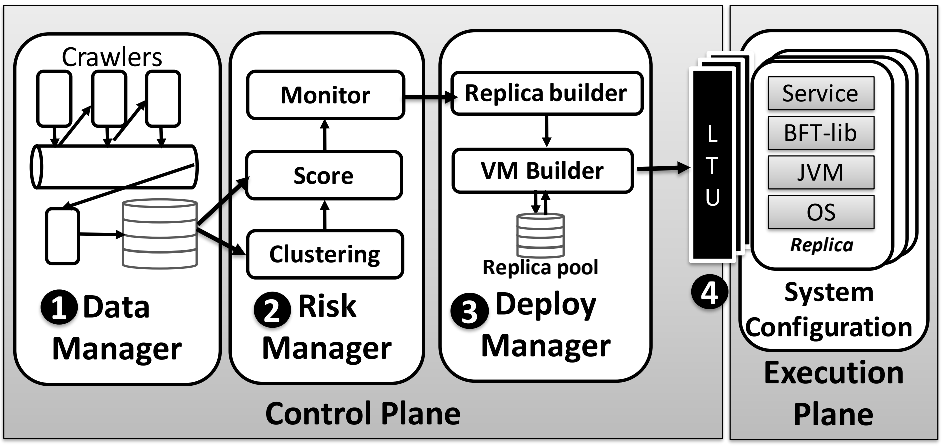


Figure 5.1 – LAZARUS architecture.

➊ ***Data manager.*** LAZARUS needs to know the software stack of each available replica to be able to look for vulnerabilities in this software. The list of software products is provided following the CPE Dictionary [129], which is also used by NVD. For each software, the administrator can indicate the time interval (in years) during which data should be obtained from NVD' feeds.

The *Data manager* parses the NVD feeds considering only the vulnerabilities that affect the chosen products. The processing is carried out with several threads cooperatively assembling as much data as possible about each vulnerability – a queue is populated with requests pertaining a particular vulnerability, and other threads will look for related data in additional OSINT sources. Typically, the other sources are not as well structured as NVD, and therefore they have to be handled with specialized HTML parsers that we have developed. Currently, the prototype supports eight other sources, namely Exploit DB [46], CVE-details [158], Ubuntu [180], Debian [47], Redhat [153], Solaris [143], FreeBSD [63], and Microsoft [126]. As previously mentioned, such sources provide complementary data, like additional affected products versions not mentioned in NVD.

The collected data is stored in a relational database (MySQL). For each vulnerability we keep its CVE identifier, the published date, the products it affects, its text description, the CVSS attributes, exploit and patching dates.

❷ ***Risk manager.*** This component finds out when it is necessary to replace the currently running group of *replicas* and discovers an alternative configuration that decreases the risk. As explained in Section 4.4, the risk is computed using score values that require two kinds of data: the information about the vulnerabilities, which is collected by the *Data manager*; and the vulnerability clusters. A vulnerability cluster is a set of vulnerabilities that are related accordingly to their description (see the following section for details). The *Risk manager* is also responsible for monitoring the replicated system and trigger reconfigurations when needed (this procedure was described in Algorithm 1).

❸ ***Deploy manager.*** This component automates the setup and execution of the diverse replicas. It creates and deploys the *replicas* in the execution environment implementing the decisions of the *Risk manager*, i.e., it dictates when and which *replicas* leave and join the system. We developed a replica builder on top of Vagrant [82]. It is responsible for downloading, installing, and configuring the *replicas*. Moreover, it performs replica maintenance, where *replicas* in `QUARANTINE` are booted to carry out automatic software updates (i.e., patching).

❹ **LTUs.** Each node that hosts a replica has a Vagrant daemon (see details in next section) running on its trusted domain. This component is isolated from the internet and communicates only with the LAZARUS controller through Transport Layer Security (TLS) channels.

5.1.2 Clustering

Clustering is the process of aggregating elements into similar groups, named clusters. For example, two elements from the same cluster have a higher probability of being similar than two elements from different clusters. We apply this technique to build clusters of similar vulnerabilities. One of the benefits of applying clustering techniques to vulnerability-data is that the algorithm does not need prior knowledge about the data. It is the process alone that discovers the hidden knowledge in the data. Each cluster is used as a hint that similar vulnerabilities are likely to be activated through the same or similar exploit. We considered the vulnerability description and published date to find these similarities. In the end, it is expected that the clusters have a minimal number of elements that represent the same or similar vulnerabilities.

A few steps are carried out to create the vulnerability clusters. First, the vulnerability description needs to be transformed into a vector, where a numerical value is associated with the most relevant words This operation entails, for example, converting all words to a canonical form and calculating their frequency (less frequent words are given higher weights). Then, the clustering algorithm is applied to build the clusters. We used the open-source machine learning library Weka [156] to prepare the data and build the clusters. This process is detailed in the following.

1) Data representation We transform the data that is stored in a database into a format that is readable by the clustering algorithm. Since we are using Weka, the data must be represented as Attribute-Relation File Format (ARFF). Basically, this is a CSV file with some meta information, see Listing 5.1.

```
@RELATION vulnerabilities
@ATTRIBUTE cve string
@ATTRIBUTE description string
@ATTRIBUTE published_date date "yyyy-MM-dd"
@DATA
CVE-2017-3301, 'vulnerability in the solaris component of oracle sun
    systems products suite subcomponent kernel the supported version that
     is [...] attacks of this vulnerability can result in unauthorized
    update insert or delete access to some of solaris accessible data
    cvss v base score integrity impacts', '2017-01-27'
...more
```

Listing 5.1 – ARFF file describing a vulnerability.

This file contains all the vulnerabilities entries in the database. As we are interested in the vulnerability similarities, we select only the CVE identifier, the text, that contains the most relevant and nonstructured information, and the published date. These attributes add meaning and temporal reference to the clustering algorithm.

2) Data preparation In general, machine learning algorithms do not handle raw data, then the data needs to be prepared: First, we transform the CVE string into a number, basically, for each CVE, there is a real number that identifies each CVE unequivocally. This attribute is transformed in numeric values using *StringToNominal* filter. Each CVE identifier is mapped to a numeric value. Second, we transform the published date to a number format that Weka can handle using the *NumericToNominal* filter. Third, the text description must be transformed into a vector, the vector will represent the frequencies of each word in the whole document of strings. We used the *StringToWordVector* to transform a text string into a vector of word weights. These weights represent the relevance of each word in the document. This filter contains the following parameters:

- **TF- and IDF-Transform**, both set to true, TF-IDF stands for term frequency-inverse document frequency. This is a statistical measure used to evaluate how relevant a word is to a document in a collection. The relevance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. For example, words that appear in all vulnerability descriptions, are less relevant than the ones that appear more in few vulnerabilities.

- **lowerCaseTokens**, convert all the words to lower case.

- **minTermFreq**, set to -1, this will preserve any word despite their occurrences in the document.

- **normalizeDocLength**, set to normalize all data counting a word with its actual value from tf-idf no matter how small or longer the document is.

- **stopwords**, we define a stop word list, then words from this list are discarded. We begin with a generic English stop word list containing pronouns, articles, etc.

- **tokenizer**, set to *WordTokenizer*, this filter will remove special characters from the text, there is a default set of characters, but we added a few more.

- **wordsTopKeep**, is the number of words that will be kept to make the clusters. We kept 200 as it was the number of words that represent better the lexical of vulnerabilities after removing the *stopwords*.

Our goal is to build clusters in such a way that similar vulnerabilities, even if they affect different products, are put together in the same cluster. For example, recall the vulnerabilities in Table 4.1, which can be put in the same cluster since they are very similar. Some of the parameters listed above needed some tuning to achieve our goals. For example, before the tuning, some of the clusters were representing types of vulnerabilities, e.g., buffer overflow, cross-site scripting, etc. Since we are not interested in that type of clusters we have refined our *stopword list* to reduce the description vocabulary to contain only what matters for us. This was done by iterating and checking the most used words for clustering (this can be seen in Weka's intermediary output). Then, we added the most relevant from the 200 words that were deviating from our goal. In the end, we have a stop word list without the security-vocabulary noise.

When the pre-filtering ends, Weka presents a file very similar to the previous ARFF file with the difference that the features are the most relevant words in the corpus. Moreover, each instance contains a value of relevance for each word.

3) Making clusters K-means is an unsupervised machine learning algorithm that groups data in K clusters [96]. The *K-means* has two important parameters: the number of clusters to be formed and the *distance function*, to calculate the distance between objects, the *Euclidean Distance* is the most adequate for this type of data; We used the elbow method [178] to decide $k = 220$; The *K-means* computes the distance from each data entry to the cluster center (randomly selected in the first round). Then, it assigns each data entry to a cluster based on the minimum distance (i.e., Euclidean distance) to each cluster center. Then, it computes the centroid, that is the average of each data attribute using only the members of each cluster. Calculate the distance from each data entry to the recent centroids. If there is no modification (i.e., re-arrangement of the elements in the cluster), then the clusters are complete, or it recalculates the distance that best fits the elements. When the K-means finishes the execution, we take the cluster assignments of each vulnerability. The assignments will be added to a new ARFF file, similar to the first one but with a new attribute that is the cluster name (e.g., cluster1, cluster2, etc).

5.1.3 Replica Virtualization and Provision

VMs can be used to implement replicated systems, leveraging on the isolation between the untrusted and the trusted domains [52, 53, 147, 170]. Recovery triggering can be initiated from the isolated domain in a synchronous manner, reducing the downtime of the service during the reconfigurations. In our implementation, we resort to the Vagrant [82] provisioning tool to do fast deployment of ready-to-use OSes and applications on VMs. Vagrant supports several virtualization providers, e.g., VMware and Docker. From the available alternatives, we chose VirtualBox [157] because it offers more diversity opportunities, i.e., it supports a more extensive set of different guests OSes.

Setup. The box configuration is defined in a configuration file, named *Vagrantfile*, consider the example in Listing 5.2. In this file, it is possible to set several options of the box, to name a few: the box name, the number of CPUs, the amount of memory RAM, the IP, the type of network, sync folders between the host and the box, etc. Additionally, it is possible to pass some VM-specific parameters, e.g., some CPU/mother board flags such as enabling VT-x technology – consider the `modifyvm` fields in Listing 5.2 (lines 6-14). It is possible to select different VM providers (e.g., libvirt, VMware, VirtualBox, Parallels, Docker, etc), we rely on VirtualBox since it is the one with more diversity opportunities in the Vagrant Cloud [83]. Vagrant Cloud is a website that offers a plethora of different VMs.

```
1 Vagrant.configure(2) do |config|
2  config.vm.box = "geerlingguy/ubuntu1604"
3  config.ssh.insert_key = false
4  config.vm.provider "virtualbox" do |v|
5   v.customize ["modifyvm", :id, "--cpus", 4]
6   v.customize ["modifyvm", :id, "--memory", 22000]
7   v.customize ["modifyvm", :id, "--cpuexecutioncap", 100]
8   v.customize ["modifyvm", :id, "--ioapic", "on"]
9   v.customize ["modifyvm", :id, "--hwvirtex", "on"]
10  v.customize ["modifyvm", :id, "--nestedpaging", "on"]
11  v.customize ["modifyvm", :id, "--pae", "on"]
12  v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
13  v.customize ["modifyvm", :id, "--natdnsproxy1", "on"]
14 end
15 config.vm.network "public_network", ip:"192.168.2.50", bridge:"em1"
16 config.vm.provision :shell, path: "run_debian.sh", privileged: true
17 end
```

Listing 5.2 – Ubuntu 16.04 Vagrantfile

Download. One of the fields of the *Vagrantfile* is the boxname, the `box` field is the key that is used to choose which box will be downloaded, each key is unique for each box. There are plenty of OSes and versions ready-to-use, and the same OS/version can have different “manufacturers" – some of which are made available by the vendor itself.

Deploy and provision. Vagrant supports complex provisions mechanisms, such as Chef or Puppet, but shell script was sufficient for us. This is set in the *Vagrantfile* `provision` field, one can describe the provision steps inline or in an external file and link it to the *Vagrantfile*. Then, when the OS is booting, after the basic setup, the OSes will execute the provision script. In this script, we program which software will be downloaded and run in the VMs and what configurations are needed. We have developed shell scripts for each OSes, some of which share the same script as Debian and Ubuntu. Each OSes, especially the ones from different OSes families (e.g., Solaris and BSD) have different commands to execute the same instructions. Therefore, we had to developed different scripts for each OS family. These scritps install software like Java 8, `wget`, `unzip`, and any additional software that one wants to install/run after the OSes boot.

Command and Control. There are some simple commands to boot and halt a box, e.g., `vagrant up` and `vagrant halt`. Vagrant also provides an ssh command (`vagrant` ssh) that allows the host to connect to the box. Our manager component makes the bridge between the *Risk manager* and the execution environment. We developed an API on top of Vagrant, another level of abstraction made to manage replicated systems.

5.1.4 BFT replication

Although there have been relevant research on BFT protocols over the last twenty years, there are few open-source replication libraries that implement them. LAZARUS can use any of those libraries, as long as they support replica set reconfigurations. More specifically, to manage the *replicas*, we need the ability to add first a new *replica* to the set and then remove the old *replica* to be quarantined. Therefore, we employ BFT-SMaRt [20], a stable BFT library that provides reconfigurations on the *replicas* set. Reconfigurations, in this case, subsumes to the addition of a new replica and the removal of the old one. The protocol we used can be briefly described as follows: *(i)* Once a new replica joins the group, it queries the other replicas about their last finished consensus instance; *(ii)* If the replica finds the others also did not execute any instance, no state transfer is required. Otherwise, the replica requests the state correspondent to the consensus instance it was given in *(i)*. *(iii)* The other replicas verify if they have the state requested (i.e., a checkpoint and a log up to the specified consensus instance). If they do, they reply with the respective checkpoint and log – only one replica sends a full checkpoint and the other $2f$ replicas send a hash of the checkpoint. Otherwise, they reply to the new replica with a void state. *(iv)* If the replica receives $2f+1$ matching replies for the requested state, it installs the received checkpoint and applies the operations in the log. If the $2f+1$ replies include a void state, the replica requests the state up to the consensus instance preceding the one being executed (thus beginning a new iteration of the protocol). This is repeated until the other replicas finally have a state to reply with.

5.1.5 Descentralized LAZARUS Version

So far we have been describing LAZARUS control plane as a logically centralized component. In this section, we present a replicated and intrusion-tolerant version of LAZARUS. This architecture envisions a set of servers that run LAZARUS in a coordinated manner. Therefore, these servers can manage the replicas in the *Execution Plane* as before, and also manage their self. This version introduces some significant modifications to the system model and architecture which are explained below.

System Model. This LAZARUS versions shares some similarities with the centralized version. For instance, the *Execution Plane* remains practically unmodified. On the contrary, the *Control Plane* suffers some changes. The new *Control Plane* is composed of replica processes, named *Controller replicas*, which can be subject to Byzantine failures. Therefore, a Byzantine *Controller replica* can try to mislead the other replicas. This plane hosts n replicas from which at most f can be compromised at any given moment. Each replica has a LTU co-allocated that is assumed as trusted. This component actions cannot deviate from their correct intention, this is especially important for the replicas recovery.

Architecture. We have designed a new architecture for the replicated LAZARUS, an overview of the architecture is shown in the Figure 5.2. In this new architecture, we have two instances of SMR. The first one runs a specific service in the *Execution Plane*, as it was already presented in Chapter 4. The second one runs in the *Control Plane*, that was assumed as a trusted and centralized component before. In this version, LAZARUS is replicated as a BFT SMR. *Controller Replicas* are coordinated to manage the *Service Replicas* in the other plane. Both, types of replicas have a LTU co-allocated with each replica. Each LTU runs a BFT library client (i.e., BFT-SMART client). These clients communicate with the library server that runs on *Controller Replicas*.

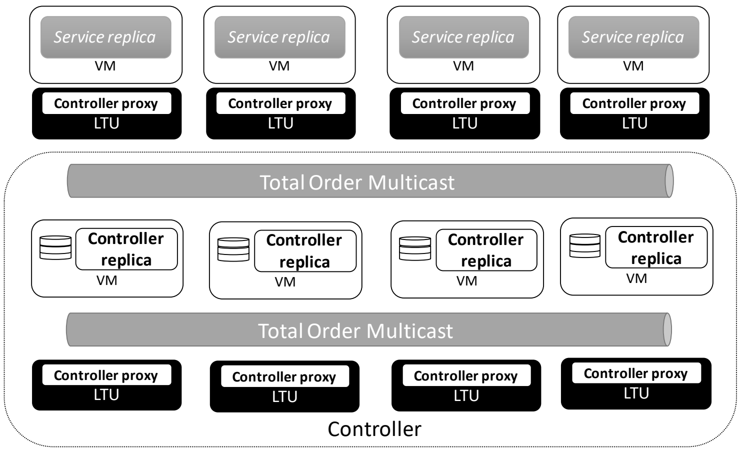


Figure 5.2 – Distributed LAZARUS architecture.

In the following we identify some of the challenges of building a BFT-LAZARUS and which solutions should be taken on each challenge.

Recovery Protocol. Contrary to the centralized LAZARUS, which decisions to reconfigure are made alone by the trusted controller, the replicated version needs a recovery protocol to ensure that a replica is reconfigured only and only if the reconfiguration was issued by correct controller replicas. We resorted to BFT-SMART to build such recovery protocol. The communication between LTUs and *Controller Replicas* is made via BFT-SMART and it is solely to trigger recoveries both on *Controller Replicas* and *Service Replicas*. Each LTU runs a *Controller Proxy* that can invoke *Controller Replicas* commands in a coordinated way, and it receives messages from *Controller Replicas*. It has to wait for $f+1$ matching messages as to know that the message is correct. Therefore, the *Controller Proxy* is ready to trigger a recovery on a *Service Replica* or on a *Controller Replica*. Contrary to the usual request-response of SMR, here, the *Controller Proxy* receives "responses" without a prior request.

Replica Synchronization Protocol. In the centralized version, LAZARUS takes decisions alone. For example, it decides when it is time to contact OSINT sources to look for new data, or when it is the time to start recoveries. On the contrary, in a SMR setting the replicas must be coordinated to execute the same command *at the same time*. One way to do that is to assume that replicas have synchronous clocks, and then, all the requests are synchronized. However, we want to make the few timely assumptions as possible. Moreover, by using the internet to communicate with OSINT sources, LAZARUS communication is subject to different levels of delays. Therefore, we propose a solution that uses a logical timeout protocol to (logically) synchronize replicas. This protocol was proposed by Kirsch *et al.* [108] to coordinate replicas to poll requests to a sensor in the context of SCADA systems. The authors named this protocol as Logical Time Out protocol, and it can be briefly described as: All *Controller Replicas* run an algorithm that periodically sends a SYNC message. This message is used to inform other replicas of the current timeouts that a replica has. Each replica will set their own timeouts by setting up a timeout $\mathtt{T}_x$ with the local time c_i and the local expiration time d to a replica i. Then, it communicates to the others the $\mathtt{T}_x$ sending the c_i and d within a SYNC message. This allows every replica to update their own list of timeouts, and each replica sets the expiration time as $c_i + d$ for $\mathtt{T}_x$. When a SYNC message from i says that the local clock is greater than $c_i + d$, then the timeout $\mathtt{T}_x$ was expired. When a logical timeout is triggered concerning $f + 1$ different replicas the timeout $\mathtt{T}_x$ is delivered to the application. This way, it is guaranteed that replicas will trigger some action only when $f + 1$ replicas had $\mathtt{T}_x$ expired.

Distributed Random Generation Protocol. As it was explained in Chapter 4, LAZARUS algorithm selects random configurations to prevent attackers from guessing which will be the next configuration. In the centralized version of LAZARUS, it is simple to select a random configuration. However, distributed random number generation is a challenge on distributed systems. In particular, to determine a random value among the replicas without prior knowledge of anyone and guaranteeing that random value is not biased, i.e., that a malicious replica is not influencing the final value. We adopted the Syta *et al.* [177] solution named RANDSHARE, as it guarantees unbiasability, unpredictability, and availability on random number generation. The protocol was developed in such a way that after a protocol barrier, honest peers will eventually output the previously decided value despite the adversary's behavior. The protocol uses secret sharing and polynomial commitments to create shares of a local random number and distribute them among the peers. With the polynomial commitment, a peer is able to verify if the share is valid without revealing information of the secret. When all their verifications are made, each peer informs the others about the validations, then any peer knows all the validations done by the others. After this stage, all nodes can recover the secrets via Lagrange interpolation. Finally, the random number is generated through a `xor` operation with the different recovered secrets.

Distributed Storage. Another modification that we have identified is the storage location. In a centralized way, the storage is a single database that is accessed by the controller. A strawman solution would use full replication of the pool and each *Controller Proxy* would wait for $f+1$ responses (the actual replica image and a cryptographic hash of the image to verify). However, this solution is not resource efficient as it requires that all replica images be replicated on all *Controller Replicas*. A more sophisticated solution would take advantage of erasure codes with secret sharing. This approach would minimize the use of resources, as each *Controller Replica* would store only parts of the complete image. Moreover, it would simplify the recovery of these replicas as they would only need to recover parts of the original image instead of the complete image. Nevertheless, this approach implies additional complexity on the *Controller Proxy* as it needs to rebuild the blocks and verify them.

Deterministic Image Patching. One of the main problems that the replicated solution brings is to guarantee that the images are loaded in a deterministic way. SMR requires that the replicas start from the same state and apply the same operations to the state in such a way that the replicas reply with the same answers to the clients. Therefore, it is a challenge to create VM images that have exactly equal binaries. A primary approach would be to guarantee that in a setup phase all the *Controller replicas* have a copy of the VM images distributed in a secure way. However, once a replica is patched in the background it becomes difficult to guarantee that the images remain equal across the *Controller replicas*. Therefore, once a *Controller proxy* that is responsible to load a VM (in the *Execution plane*) cannot vote which image is correct as they are different among the (even correct) *Controller replicas* due to the patches.

5.2 Evaluation

In this section, we evaluate how LAZARUS affects the performance of diverse replicated systems. First, we run the BFT-SMaRt microbenchmarks in our virtualized environment using 17 OS versions to understand how the performance of a BFT protocol varies with different OSes, and how they compare with the performance of a homogeneous bare metal setup. Second, we use the same benchmarks to measure the performance of specific diverse setups. Third, we analyze the performance of the system along LAZARUS-managed reconfigurations. Fourth, we measure the time that each OS takes to boot. Finally, we evaluate the performance of three BFT services running in the LAZARUS infrastructure.

These experiments were conducted in a cluster of Dell PowerEdge R410 machines, where each one has 32 GB of memory and two quad-core 2.27 GHz Intel Xeon E5520 processor with hyper-threading, i.e., supporting 16 hardware threads on each node. The machines communicate through a gigabit Ethernet network. Each server runs Ubuntu Linux 14.04 LTS (3.13.0-32-generic Kernel) and VirtualBox 5.1.28, for supporting the execution of VMs with different OSes. Additionally, Vagrant 2.0.0 was used as the provisioning tool to automate the deployment process. In all experiments, we configure BFT-SMaRt v1.1 with four replicas ($f = 1$), one replica per physical machine.

ID	Name	Cores	JVM	Mem.
UB14	Ubuntu 14.04	4	Java Oracle 1.8.0_144	15GB
UB16	Ubuntu 16.04	4	Java Oracle 1.8.0_144	15GB
UB17	Ubuntu 17.04	4	Java Oracle 1.8.0_144	15GB
OS42	OpenSuse 42.1	4	Openjdk 1.8.0_141	15GB
FE24	Fedora 24	4	Openjdk 1.8.0_141	15GB
FE25	Fedora 25	4	Openjdk 1.8.0_141	15GB
FE26	Fedora 26	4	Openjdk 1.8.0_141	15GB
DE7	Debian 7	4	Java Oracle 1.8.0_151	15GB
DE8	Debian 8	4	Openjdk 1.8.0_131	15GB
W10	Windows 10	4	Java Oracle 1.8.0_151	1GB
WS12	Win. Server 2012	4	Java Oracle 1.8.0_151	1GB
FB10	FreeBSD 10	4	Openjdk 1.8.0_144	15GB
FB11	FreeBSD 11	4	Openjdk 1.8.0_144	15GB
SO10	Solaris 10	1	Java Oracle 1.8.0_141	15GB
SO11	Solaris 11	1	Java Oracle 1.8.0_05	15GB
OB60	OpenBSD 6.0	1	Openjdk 1.8.0_72	1GB
OB61	OpenBSD 6.1	1	Openjdk 1.8.0_121	1GB

Table 5.1 – The different OSes used in the experiments and the configurations of their VMs and JVMs.

Table 5.1 lists the 17 OS versions used in the experiments and the number of cores used by their corresponding VMs. These values correspond to the maximum number of CPUs supported by VirtualBox with that particular OS. The table also shows the JVM used in each OS, and the amount of memory supported by each of these VMs. Given these limitations we setup our environment to establish a fair baseline by configuring an *homogeneous* Bare Metal (BM) environment that uses only four cores of the physical machine.

5.2.1 Homogeneous Replicas Throughput

We start by running the BFT-SMaRt microbenchmark using the *same OS version* in all replicas. The microbenchmark considers an empty service that receives and replies variable size payloads, and is commonly used to evaluate BFT state-machine replication protocols (e.g., [15, 16, 20, 29, 122]). Here, we consider the $0/0$ and $1024/1024$ workloads, i.e., 0 and 1024 bytes requests/response, respectively. The experiments employ up to 1400 client processes spread on seven machines to create the workload.

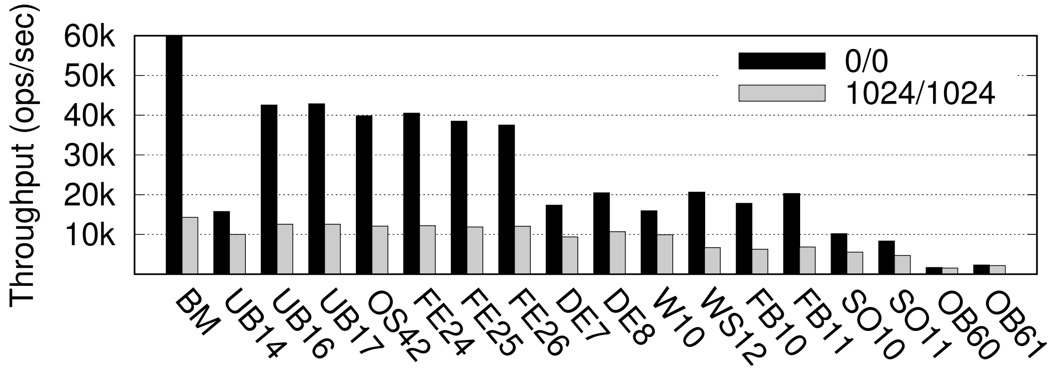


Figure 5.3 – Microbenchmark for 0/0 and 1024/1024 (request/replying) for homogeneous OSes configurations.

Results: Figure 5.3 shows the throughput of each OS running the benchmark for both loads. To establish a baseline, we executed the benchmark in our bare metal Ubuntu, without LAZARUS virtualization environment.

The results show that there are some significant differences between running the system on top of different OSes. This difference is more significant for the $0/0$ workload as it is much more CPU intensive than the $1024/1024$ workload. Ubuntu, OpenSuse, and Fedora OSes are well supported by our virtualization environment and achieved a throughput around $40k$ and $10k$ for the $0/0$ and $1024/1024$ workloads, which corresponds to approx. 66% and 75% of the bare metal results, respectively. For Debian, Windows, and FreeBSD, the results are much worse for the CPU intensive $0/0$ workloads but close to the previous group for $1024/1024$. Finally, single core VMs running Solaris and OpenBSD reached no more than 3000 ops/sec with both workloads.

These results show that the virtualization platform limitations on supporting different OSes, strongly limits the performance of specific OSes in our testbed.

5.2.2 Diverse Replicas Throughput

The previous results show the performance of BFT-SMaRt when running on top of different OSes, but with all replicas running in the same environment. In this experiment, we evaluate three diverse sets of four replicas, one with the fastest OSes (UB17, UB16, FE24, and OS42), another with one replica of each OS family (UB16, W10, SO10, and OB61), and a last one with the slowest OSes (OB60, OB61, SO10, and SO11). The idea is to set an upper and lower bound on all possible diverse sets throughput.

Results: Figure 5.4 shows that throughput drops from 39k to 6k for the $0/0$ workload (65% and 10% of the bare metal performance), and from 11.5k to 2.5k for the $1024/1024$

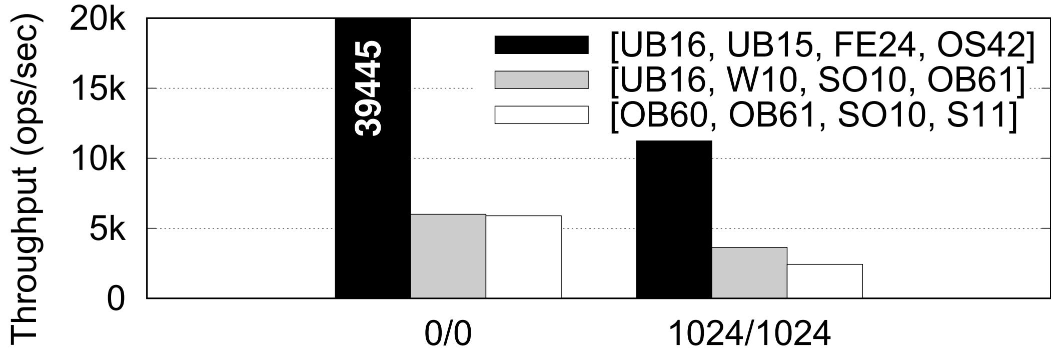


Figure 5.4 – Microbenchmark for 0/0 and 1024/1024 (request/reply) for three diverse OS configurations.

workload (82% and 18% of the bare metal performance). When comparing these two sets with the non-diverse sets of Figure 5.3, the fastest set is in 7^{th}, and the slowest set is in 16^{th}. It is worth to stress that the slowest set is composed of OSes that only support a single CPU – due to the VirtualBox limitations – therefore the low performance is somewhat expected. The set with OSes from different families is very close to the slowest set, as two of the replicas use single-CPU OSes, and BFT-SMaRt always makes progress in the speed of the 3rd fastest replica (a Solaris VM), since its Byzantine quorum needs three replicas for ordering requests. These results show that running LAZARUS with current virtualization technology results in a significant performance variation, depending on the configurations selected by the system. This opens interesting avenues for future work on protocols that consider such performance diversity, as will be discussed in Chapter 7.

5.2.3 Performance During Replicas' Reconfiguration

Another relevant feature of LAZARUS is to adapt the replicas over time. LAZARUS leverages on the BFT-SMaRt reconfiguration protocol to add and remove replicas while BFT state is maintained correct and up to date.

In this experiment, we want to compare the replicas reconfiguration in the homogeneous BM environment with the LAZARUS environment (the initial OS configuration is DE8, OS42, FE26, and SO11). Our goal is to evaluate the two different environments with the same experimental setup to measure the overhead of LAZARUS-induced reconfigurations. We execute this experiment with an in-memory Key-Value Storage (KVS) service that comes with the BFT-SMaRT library. The experiment was conducted with a Yahoo! Cloud Serving Benchmark (YCSB) [39] workload of 50% of reads and 50% of writes, with values of 1024 bytes associated with small numeric keys, varying the state size of the KVS. Since the virtualization environment limits the performance (as shown in the previous experiments), we used a workload that best fits both environments. Then we setup the benchmark to send

≈ 4000 operations per second over a state of 500MBs. We selected a time frame of 200 seconds to show the performance during a reconfiguration.

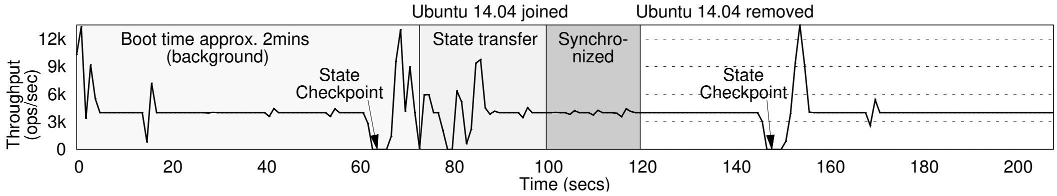


Figure 5.5 – Bare Metal reconfiguration KVS performance on a 50/50 YCSB workload, 1kB-values and with ≈ 500MBs state size.

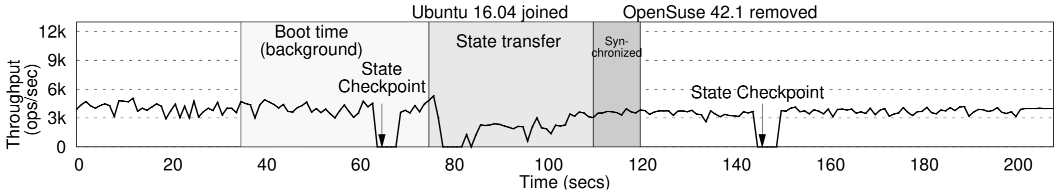


Figure 5.6 – LAZARUS reconfiguration KVS performance on a 50/50 YCSB workload, 1kB-values and with ≈ 500MBs state size..

Results: Figure 5.5 and Figure 5.6 show the replicas throughput measured with YCSB benchmark which runs in the client side. In both figures, there are two types of performance drops. The first one, named State Checkpoint, is the period in which the BFT-SMaRt replicas trim their operation logs. During these periods there are no messages processed. The second ones, composed of the state transfer and the synchronization, were already identified, and mitigated in previous works [19].[1]

There are some differences between both executions, in particular, the throughput variation and the time each phase takes to finish. Both experiments were made to represent the same behavior, therefore the different actions start approximately at the same time. For example, in both cases, the State Checkpoint is made almost at the same time as it is triggered by the number of processed messages on each replica.

In the BM experiment (Figure 5.5) the state checkpoint occurs around the second 60. After a few seconds, the new replica (i.e., Ubuntu 14.04) joins the system, and a state transfer begins. It takes almost 30secs to get the state and then it takes approx. 20secs to execute the log and reach with the other replicas state. Only then one of the *old* replicas is removed. The booting time for Ubuntu 14.04 was more than 2mins in our testbed.[2]

In the LAZARUS setting experiment (Figure 5.6), it is possible to see a little variance in the throughput and a significant impact on the throughput during the reconfiguration. However,

[1]We employ the standard checkpoint and state transfer protocols of BFT-SMaRt and not the one introduced in [19] as the developers of the system pointed them as more stable.

[2]This time is due to special verification tests that machines in a cluster typically do before booting.

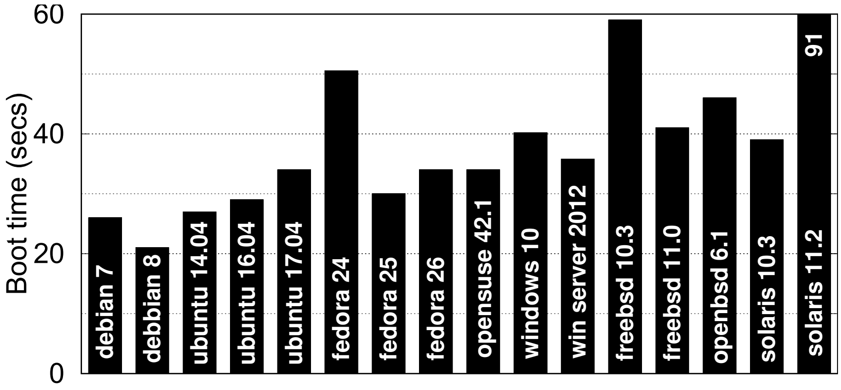


Figure 5.7 – OSes boot times (seconds).

the throughput and the time it takes to finish the state reconfiguration is close to the previous experiment. Finally, the booting takes approx. 40secs.

Additionally, we have conducted a more straightforward experiment to measure the boot time of the OSes supported by LAZARUS prototype. Figure 5.7 shows the average boot time of 20 executions boot for each OS. As can be seen, most OSes take less than 40secs to boot, with exceptions of Fedora 24 (50secs), FreeBSD 10.3 (59secs), and Solaris 11.2 (91secs). In any case, these results together with the reconfiguration times of the previous section show that LAZARUS can react to a new threat in less than two minutes (in the worst case).

5.2.4 Application Benchmarks

Our last set of experiments aims to measure the throughput of three existing BFT services built on top of BFT-SMaRt when running in LAZARUS. The considered applications and workloads are:

- *KVS* is the same BFT-SMaRt application employed in Section 5.2.3. It represents a consistent non-relational database that stores data in memory, similarly to a coordination service (an evaluation scenario used in many recent papers on BFT [16, 122]). In this evaluation, we employ the YCSB 50%/50% read/write workload with values of 4k bytes.

- SIEVEQ [73] is a BFT message queue service that can also be used as an application-level firewall, filtering messages that do not comply with a pre-configured security policy. Its architecture, based on several filtering layers, reduces the costs of filtering invalid messages in a BFT-replicated state machine. In our evaluation, we consider all the layers were running on the same four physical machines as the diverse BFT-SMaRt replicas (under different OSes). The workload imposed on the system is composed of messages of 1k bytes.

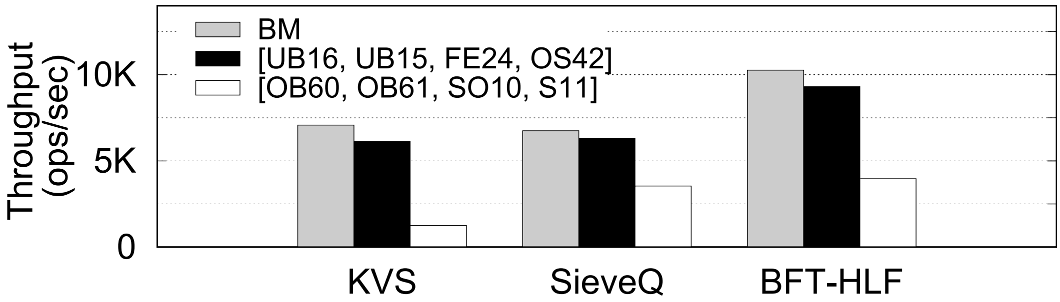


Figure 5.8 – Different BFT applications running in the bare metal, fastest and slowest OS configurations.

- *BFT ordering for Hyperledger Fabric* [169] is the first BFT ordering service for Fabric [7]. Fabric is an extensible blockchain platform designed for business applications beyond the basic digital coin. The ordering service is the core of Fabric, being responsible for ordering and grouping issued transactions in signed blocks that form the blockchain. In our evaluation, we consider transactions of 1k bytes, blocks of 10 transactions and a single block receiver.

As in Section 5.2.2, we run the applications on the fastest and slowest diverse replica sets and compare them with the results obtained in bare metal.

Results: Figure 5.8 shows the peak sustained throughput of the applications. The KVS results show a throughput of 6.1k and 1.2k ops/sec, for the fastest and slowest configurations, respectively. This corresponds to 86% and 18% of the 7.1k ops/sec achieved on bare metal.

The SIEVEQ results show a smaller performance loss when compared with bare metal results. More specifically, SIEVEQ in the fastest replica set reaches 94% of the throughput achieved on the bare metal. Even with the slowest set, the system achieved 53% of the throughput of bare metal. This smaller loss happens due to the layered architecture of SIEVEQ, in which most of the message validations happen before the message reaches the BFT replicated state machine (which is the only layer managed by LAZARUS).

The Fabric ordering service results show that running the application on LAZARUS virtualization infrastructure lead to 91% (fastest set) to 39% (slowest set) of the throughput achieved on bare metal.

Nonetheless, even the slowest configurations would not be a significant bottleneck if one takes into consideration the current performance of Fabric [169]

5.3 Final Remarks

In this chapter, we focused on the implementation details of LAZARUS and its evaluation as practical systems. We described each component and how all together make LAZARUS works. Then, we evaluated the performance of using LAZARUS solution, i.e., what is the overhead of running diverse OSes in a virtualized environment. Some of the limitations that we have identified in the evaluation are discussed in Chapter 7.

SIEVEQ: A Use Case

6

In the previous chapter, we presented a control plane for BFT systems, named LAZARUS. Although we have already introduced SIEVEQ in the evaluation of LAZARUS, we will detail SIEVEQ contributions in this chapter. SIEVEQ a message queue service that protects and regulates the access to critical systems, in a way similar to an application-level firewall. The solution provides fault and intrusion tolerance by employing an architecture based on two filtering layers, enabling efficient removal of invalid messages at early stages and decreasing the costs associated with BFT replication of previous solutions.

6.1 Introduction

Firewalls are one of the primary protection mechanisms against external threats, controlling the traffic that flows in and out of a network. Typically, they decide if a packet should go through (or be dropped) based on the analysis of its contents. Over the years, this analysis has been performed at different levels of the Open Systems Interconnection (OSI) model (see [104] for a comprehensive survey), but the most sophisticated rules are based on the inspection of application data included in the packets. State-of-the-art solutions for application-level firewalls include network appliances from several vendors, such as Juniper [100], Palo Alto [145], and Dell [49]. Such appliances are strategically placed on the network borders, and therefore, the security of the whole infrastructure relies on them. A skilled attacker, however, may find weaknesses to compromise the firewall's detection/prevention capabilities. When this happens, critical services under the protection of such devices may be affected, as in the SCADA systems targeted in the Stuxnet [60] or Dragonfly [175] attacks.

Most generic firewall solutions suffer from two inherent problems: First, they have vulnerabilities as any other system, and as a consequence, they can also be the target of advanced attacks. For example, the NVD [134] shows that there have been many security issues in commonly used firewalls. NVD's reports present the following numbers of security issues between 2010 and 2015: 157 for the Cisco Adaptive Security Appliance; 109 in Juniper Networks solutions; 29 for the Sonicwall firewall; and 24 related to `iptables`/`netfilter`. Common protection solutions often have been the target of malicious actions as part of a wider scale attack (e.g., anti-virus software [32], IDS [6] or firewalls [35, 36, 102, 174]). Second, firewalls are typically a single point of failure, which means that when they crash,

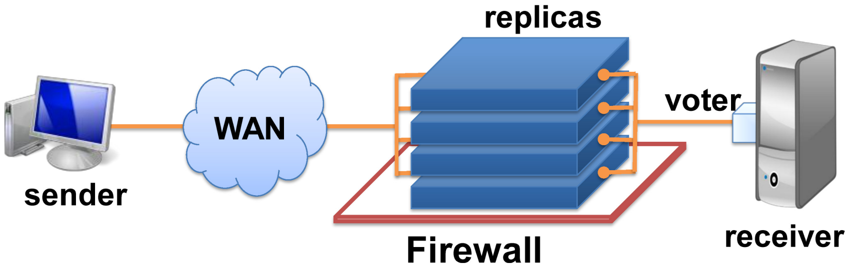


Figure 6.1 – Architecture of a state-of-the-art replicated firewall.

the ability of the protected system to communicate may be compromised, at least momentarily. Therefore, ensuring the correct operation of the firewall under a wide range of failure scenarios becomes imperative. To tolerate faults, one typically resorts to the replication of the components.

In this chapter, we propose a new protection system called SIEVEQ that mixes the firewall paradigm with a message queue service, with the goal of improving the state-of-the-art approaches under accidental failures and/or attacks. The solution has a fault- and intrusion-tolerant architecture that applies filtering operations in two stages acting like a sieve. The first stage, called *pre-filtering*, performs lightweight checks, making it efficient to detect and discard malicious messages from external adversaries. In particular, messages are only allowed to go through if they come from a pre-defined set of authenticated senders. DoS traffic from external sources is immediately dropped, preventing those messages from overloading the next stage. The second stage, named *filtering*, enforces more refined application level policies, which can require the inspection of some message fields or need the enforcement of specific ordering rules.

6.2 Intrusion-Tolerant Firewalls

In the last decade, several significant advances occurred in the development of intrusion-tolerant systems. However, to the best of our knowledge, very few works proposed intrusion-tolerant protection devices, such as firewalls. Performance reasons might explain this, as BFT replication protocols are usually associated with significant overheads and limited scalability. Additionally, achieving complete transparency to the rest of the system can be challenging to reconcile with the objective of having fast message filtering under attack.

Figure 6.1 shows an implementation of an intrusion-tolerant firewall, illustrating existing works in this area [160, 170]. In this design, a sender transmits the messages through the network (e.g., the internet) towards the receiver. As packets reach the firewall, they are disseminated to the replicas. Each replica applies the same filtering rules to decide whether the messages are acceptable. Invalid messages are discarded (and eventually logged).

Messages deemed valid are conveyed to the receiver together with a proof of validity, which demonstrates that a sufficiently large quorum of replicas agrees on their validity. The proof of validity is checked by a voter module at the receiver, before delivering the messages to the receiving application. Thus, if a compromised replica produces a message with malicious content, it will be eliminated as it lacks the necessary proof of validity or it is in conflict with the messages transmitted by the other correct replicas.

Although this architecture has interesting characteristics, such as an increased failure resilience, it suffers from some fundamental limitations:

1. The dissemination of a message to all replicas can be detrimental to the proper operation of the firewall. For example, a traffic replicator device (e.g., hub) can be placed at the entry of the firewall to reproduce all messages [160, 170] transparently. An obvious consequence of this approach is that malicious messages from an external attacker are also replicated, and therefore, all replicas have to spend the same effort to process them. As a consequence, the attack is amplified by the replicator device. Alternatively, a leader replica could receive the traffic and then disseminate the messages to the others [160]. The drawback is that the leader becomes a natural bottleneck, especially when under attack (instead of dispersing the attack load over all replicas [5]).

2. The support for stateful firewall filtering requires that all correct replicas process messages in the same order [162]. As a consequence, to ensure an agreement in a common sequence of messages, replicas need to continuously run a BFT consensus protocol [30] to establish message ordering. A significant amount of work can be wasted with malicious messages since all messages have to be agreed. This is particularly relevant because a consensus protocol consumes both computational and network resources.

3. The creation and check of the proof of validity can be a complex task. For example, one approach requires a trusted component to be deployed in the replicas to generate a MAC as a proof that a message is valid [170]. The component only returns the MAC when a quorum of replicas accepted the message. Another solution uses threshold cryptography to ensure that every replica can individually produce a partial signature (that corresponds to a part of the proof) [160]. To recreate the full proof, the voter needs to wait for the arrival of a quorum of partial proofs. When building a firewall, it would be useful if a more straightforward approach could be employed, with no need for specialized trusted components or expensive threshold cryptography.

These drawbacks can have a significant impact on the firewall performance depending on the considered setting. For example, a typical DoS attack can create a substantial decrease

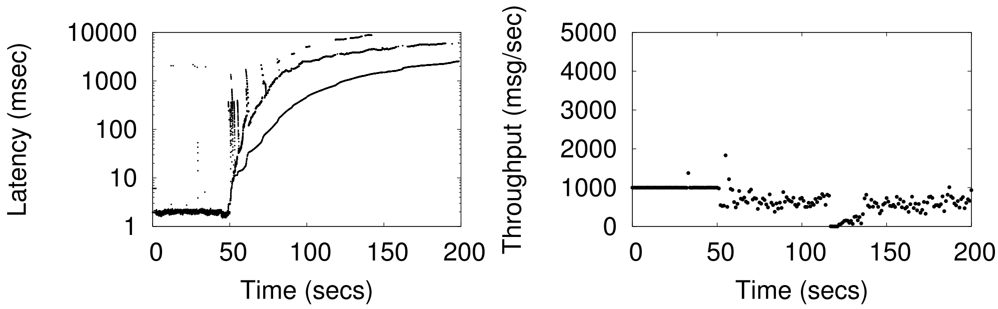


Figure 6.2 – Effect of a DoS attack (initiated at second 50) on the latency and throughput of the intrusion-tolerant firewall architecture displayed in Figure 6.1.

in the throughput and several orders of magnitude growth in the latency of message delivery. To illustrate this behavior we implemented the architecture of Figure 6.1 and launched a DoS attack on this system (see Section 6.6 for a description of the setup and environment). The results show that the system performance is significantly affected by the attack (see Figure 6.2).

In SIEVEQ, we explore a different design for replicated protection devices, where we trade some transparency on senders and receivers for a more efficient and resilient firewall solution. In particular, we propose an architecture in which critical services and devices can only be accessed through a message queue and implement the application-level filtering in this queue. It is assumed that these services have a limited number of senders, which can be appropriately configured to ensure that only they are authorized to communicate through SIEVEQ.

6.3 Overview of SIEVEQ

Typical resilient firewall designs are based on primary-backup replication, and consequently, they are able to tolerate only crash failures. Therefore, more elaborated failure modes may allow an adversary to penetrate the protected network.

Some organizations deal with crashes (or DoS attacks) by resorting to several firewalls to support multiple entry points. This solution is helpful to address some (accidental) failures but is incapable of dealing with an intrusion in a firewall. In this case, the adversary gains access to the internal network, enabling an escalation of the attack, which at that stage can only be stopped if other protection mechanisms are in place.

SIEVEQ provides a message queue abstraction for critical services, applying various filtering rules to determine if messages are allowed to go through. SIEVEQ is not a conventional firewall, and we do not claim that it should replace existing firewalls in all deployment scenarios. We are focusing on service- or information-critical systems that require a high level of protection, and therefore, justify the implementation of advanced replication mechanisms. The system we propose is able to deliver messages while guaranteeing authenticity, integrity, and availability. As a consequence, and in contrast to conventional firewalls, we lose transparency on senders and receivers, since they are aware of the SIEVEQ's end-points. The rest of the section explains how we address some of the mentioned issues and introduces the main design choices and the architecture of SIEVEQ.

6.3.1 Design Principles

Our solution was guided by the following principles:

- *Application-level filtering*: support sophisticated firewall filtering rules that take advantage of application knowledge. SIEVEQ implements this sort of rules by maintaining state about the existing flows, and this state has to be consistently replicated using a BFT protocol.

- *Performance*: address the most probable attack scenarios with highly efficient approaches, and as early as possible in the filtering stages; Reduce communication costs with external senders, as these messages may have to travel over high latency links (e.g., do not require message multicasts).

- *Resilience*: tolerate a broad range of failure scenarios, including malicious external/internal attackers, compromised authenticated senders, and intrusions in a subset of the SIEVEQ components; Prevent malicious external traffic from reaching the internal network by requiring explicit message authentication.

6.3.2 SIEVEQ Architecture

A fundamental difference between the SIEVEQ architecture and the other replicated firewall designs (see Figure 6.1), is the separation of filtering in several stages. The rationale for this change is to gain flexibility in the filtering operations while ensuring better performance under attack, retaining the ability to tolerate intrusions. As observed previously, despite the significant improvements in state-of-the-art BFT implementations, there is an inherent trade-off between the benefits of BFT replication and its performance, namely due to the need to disseminate (and eventually authenticate) all messages at the replicas, which includes both valid and invalid messages.

Figure 6.3 presents the architecture of SIEVEQ. In this architecture, message processing starts with a first filtering layer that implements a message authentication mechanism and is

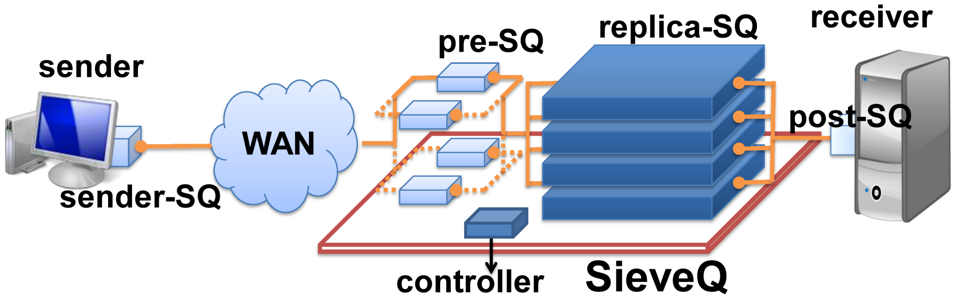


Figure 6.3 – SIEVEQ layered architecture.

responsible for discarding most of the malicious traffic efficiently. This layer is based on a set of *pre-SQ* modules, each of them in charge of the communications with a subgroup of senders. During a typical operation, a sender only interacts with its own *pre-SQ*. The assignment of a *sender-SQ* to a *pre-SQ* is done during the channel setup. Initially, a sender connects with one of a few statically-configured *pre-SQs*. If a *pre-SQ* becomes overloaded, it will request the creation of more *pre-SQs* (see details in Section 6.4.3.2) and/or hand of the new *sender-SQ* channel to another node.

The messages are sent to the second filtering layer by the *pre-SQ* to perform a more detailed inspection, which can take advantage of state information kept from previous messages and application-related rules. This layer is implemented by a group of *replica-SQs* acting together as a BFT replicated state machine. They receive all accepted messages from the *pre-SQs* and process them in the same order, which guarantees that every *replica-SQ* reaches the same decision (discard or accept a message). However, if f replicas are faulty, their output could be different. Consequently, as long as more than two-thirds of the *replica-SQs* are correct, the right decision is taken by the *post-SQ* by performing a message voting.

In the following, we describe each module of SIEVEQ presented in Figure 6.3:

Sender-SQ. The sender nodes cooperate with SIEVEQ to secure the messages by deploying a *sender-SQ* module locally. Its primary role is to secure the messages and assist in the detection of some intruded SIEVEQ components. The module can be implemented inside the sender's OS (e.g., as a kernel module or a specialized device driver) or as a library to be linked with the applications. The decision to have this module corresponds to a trade-off in our design, where we are willing to lose some transparency to improve the system's resilience.

Pre-SQ. These are the SIEVEQ front-end, and although it only performs stateless filtering to improve efficiency, it can deter the most common attacks. *Pre-SQ* modules discard invalid messages, while the approved ones are forwarded to the *replica-SQ* using a *Byzantine Total Ordered Multicast (TOM)* protocol [20]. The *pre-SQs* can be deployed, for instance, as VMs.

The effect of this layer is a significant reduction in the communication and computational overhead caused by malicious packets at the *replica-SQ*.

Replica-SQ. These components implement a replicated filtering service that tolerates Byzantine faults. The actual filtering rules can be more or less complex depending on the needs of the critical service. The TOM ensures that *replica-SQs* receive the messages in the same order. Consequently, identical rules are applied across the *replica-SQs*, and therefore, the same decision should be reached on the validity of messages. Each *replica-SQ* individually transmits approved messages to the final receiver (the others are dropped). Overall, this layer allows for sophisticated filtering as the replicas are stateful.

Post-SQ. This module runs on the receiver side, and it is responsible for the delivery of messages to the application. A *post-SQ* carries out a voting operation on the arriving data because a *replica-SQ* might be intruded and corrupt messages. It delivers a message to the application only after receiving the same approved message from a quorum of *replica-SQs*. From a deployment perspective, this module can be implemented in the OS or as a library, as in the sender.

Controller. This module is a trusted component of SIEVEQ that runs with high privilege. It takes input from the *replica-SQs* to decide on the creation or destruction of *pre-SQs* if some misbehavior is observed. Depending on the actual SIEVEQ implementation, it can be developed in different ways. This sort of component was used in previous works, and it can be implemented both in a centralized [147, 160] or distributed [170] way. In the same way, we have presented two possible solutions for the LAZARUS controller in Chapter 5.

6.3.3 Resilience Mechanisms

The SIEVEQ architecture is built to tolerate both faults with an accidental nature (e.g., crashes) and caused by malicious actions (e.g., a vulnerability is exploited, and a specific module is compromised). To be conservative, we assume that all failed components are controlled by a single entity, which will make them act together in the worst possible manner to defeat the correctness of the system. Therefore, failed components can, for instance, stop sending messages, produce erroneous information, or try to delay the system. SIEVEQ performs several mitigation actions to guarantee a valid operation (as long as the number of faults is within the assumed bounds, see Section 6.3.4).

The most common attack scenario occurs when an external adversary attempts to attack a system that is being protected by SIEVEQ. He can deploy many nodes, whose aim is to delay the communications or bypass SIEVEQ protection and reach the internal network. SIEVEQ addresses these attacks by discarding unauthenticated or corrupted messages with minimal effort at the *pre-SQ* filtering stage. As with any other firewall, if a DoS attack completely overloads the incoming channels, SIEVEQ cannot handle or react to the attack. The network needs to include other defense mechanisms to deal with this sort of problem [128].

As (authenticated) senders might be spread over many (outside) networks, it is advisable to consider a second scenario where an adversary is capable of taking control of some of these nodes. In this case, we assume that the adversary gains access to all data stored locally, including the *sender-SQ* keys. Thus, he will be able to generate traffic that is correctly authenticated, allowing these messages to go through the first filtering step. The messages are however still checked against the application related rules (namely, the ones defined in the *replica-SQ*), which can cause most malicious traffic to be dropped (e.g., a pre-defined *sender-SQ* can only send messages to a particular *post-SQ* accordingly to a specific application protocol). If the messages follow all the rules, the firewall has to forward them because they are indistinguishable from any other valid messages.

A third scenario occurs when the adversary is able to cause an intrusion in SIEVEQ and compromises a few of the *pre-SQs* and/or *replica-SQs*. When this happens, these components can act in an erroneous (Byzantine) way. However, unlike with an intruded *sender-SQ*, malicious *pre-SQs* cannot generate fully authenticated messages, since they lack all the required keys. They can still perform DoS attacks on the *replica-SQs*, e.g., by transmitting many messages, but this strategy creates obvious misbehavior allowing discovery immediately. Malicious *pre-SQs* are detected with the assistance of correct *sender-SQ* and *replica-SQs*, eventually leading to their substitution.

Replica-SQs modules are much harder to exploit because they do not face the external network. However, if they end up being intruded, *replica-SQs* can produce arbitrary traffic to the internal network. *Post-SQ* addresses this issue by carrying out a voting step, which excludes these messages. Moreover, an alarm is generated and sent to the *controller*.

The SIEVEQ architecture does not attempt to recover from intrusions in the *controller* and *post-SQ*. The first is assumed to be trusted, as it is deployed in a separated administrative domain and is to be used only in a few particular operations. Moreover, its simplicity allows the audit of its code and ensures correctness with a high level of confidence. The *post-SQ* already runs in the internal network, and therefore, SIEVEQ can not preclude its misbehavior.

6.3.4 System and Threat Model

The system is composed of a (potentially) large number of external nodes, called *senders*, some internal nodes, called *receivers*, and SIEVEQ nodes. Senders run *sender-SQ* modules to be able to transmit packets through the SIEVEQ, while receivers receive validated messages by using *post-SQ* modules.

Communications can experience accidental faults or attacks. Thus, packets might be lost, delayed, reordered or corrupted, but we assume that if messages are retransmitted, eventually they will be correctly received by SIEVEQ. The fault model also assumes that *sender-SQ*, *pre-SQ*, and *replica-SQ* nodes can suffer from arbitrary (Byzantine) faults. When this happens, failed nodes may perform actions that deviate from their specification, including colluding against the system. However, at most f_{ps} *pre-SQs* from a total of $N_{ps} = f_{ps} + k$ (with $k > 1$), and f_{rs} *replica-SQ* from a total of $N_{rs} = 3f_{rs} + 1$ may fail. Redundant components should fail independently by employing diversity techniques such as the ones described in Chapter 4. A component that is unable to communicate is also considered faulty because from a practical perspective it is indistinguishable from a crashed module.

The cryptographic operations used in the SIEVEQ protocol are assumed to be secure, and therefore, they cannot be subverted by an adversary. Consequently, traditional properties of digital signatures, MACs, and hash functions will hold as long as the associated keys are kept safe. The deployment of SIEVEQ requires a key distribution scheme to create shared keys between the *sender-SQ* and the *pre-SQ* and to periodically re-issue private-public key pairs for the *sender-SQ*. We assume that the key distribution scheme is similar to solutions that already address this sort of problem (e.g., [81]). If required, the key distribution infrastructure could also be made intrusion-tolerant (e.g., [114, 202]).

6.3.5 Resilience Mechanisms

The SIEVEQ architecture is built to tolerate both faults with an accidental nature (e.g., crashes) and caused by malicious actions (e.g., a vulnerability is exploited, and a specific module is compromised). To be conservative, we assume that all failed components are controlled by a single entity, which will make them act together in the worst possible manner to defeat the correctness of the system. Therefore, failed components can stop sending messages, produce erroneous information, or try to delay the system. SIEVEQ performs several mitigation actions to guarantee a valid operation (as long as the number of faults is within the assumed bounds, see Section 6.3.4).

The most common attack scenario occurs when an external adversary attempts to attack a system that is being protected by SIEVEQ. He can deploy many nodes, whose aim is to delay the communications or bypass SIEVEQ protection and reach the internal network. SIEVEQ addresses these attacks by discarding unauthenticated or corrupted messages with minimal effort at the *pre-SQ* filtering stage. As with any other firewall, if a DoS attack completely overloads the incoming channels, SIEVEQ cannot handle or react to the attack. The network needs to include other defense mechanisms to deal with this sort of problem [128].

As (authenticated) senders might be spread over many (outside) networks, it is advisable to consider a second scenario where an adversary is capable of taking control of some of these nodes. In this case, we assume that the adversary gains access to all data stored locally, including the *sender-SQ* keys. Thus, he will be able to generate traffic that is correctly authenticated, allowing these messages to go through the first filtering step. Then, the messages are still checked against the application related rules (namely, the ones defined in the *replica-SQ*), which can cause most malicious traffic to be dropped (e.g., a pre-defined *sender-SQ* can only send messages to a particular *post-SQ* accordingly to a specific application protocol). If the messages follow all the rules, the firewall has to forward them because they are indistinguishable from any other valid messages.

A third scenario occurs when the adversary can cause an intrusion in SIEVEQ and compromises a few of the *pre-SQs* and/or *replica-SQs*. When this happens, these components can act in an erroneous (Byzantine) way. However, unlike with an intruded *sender-SQ*, malicious *pre-SQs* cannot generate fully authenticated messages, since they lack all the required keys. They can still perform DoS attacks on the *replica-SQs*, e.g., by transmitting many messages, but this strategy creates apparent misbehavior allowing immediate discovery. Malicious *pre-SQs* are detected with the assistance of correct *sender-SQ* and *replica-SQs*, eventually leading to their substitution.

Replica-SQs modules are much harder to exploit because they do not face the external network. However, if they end up being intruded, *replica-SQs* can produce arbitrary traffic to the internal network. *Post-SQ* addresses this issue by carrying out a voting step, which excludes these messages. Moreover, an alarm is generated and sent to the *controller*.

The SIEVEQ architecture does not attempt to recover from intrusions in the *controller* and *post-SQ*. The *post-SQ* already runs in the internal network, and therefore, SIEVEQ can not preclude its misbehavior. In this chapter, for the sake of simplicity we are considering the *controller* as trusted. However, in Chapter 5 we discuss alternatives with weaker assumptions. Moreover, its simplicity allows the audit of its code and ensures correctness with a high level of confidence.

6.4 SIEVEQ Protocol

This section details the SIEVEQ protocol and service properties. We conclude the section with an analysis of the behavior of the system under different kinds of attacks and component failures and highlight how the countermeasures integrated into our design mitigate such threats.

6.4.1 Properties

SIEVEQ protocol guarantees the following three properties for messages transmitted from a sender to a receiver:

Compliance. *If a message, transmitted by a correct sender, is delivered to a correct receiver then the message is in accordance with the security policy of* SIEVEQ.

Validity. *If a correct receiver delivers a message* `Msg.DATA`*, then the message was transmitted by* `Msg.sender`*.*

Liveness. *If a correct sender sends a message, then the message eventually will be delivered to the correct receiver.*

These properties require SIEVEQ to behave in a way similar to most firewalls while offering a few extra guarantees. Only external messages that are approved by the policies defined in SIEVEQ can reach the receivers, and the rest should be dropped (Compliance). *Post-SQ* can use the message field `Msg.sender` to find who transmitted the message contents (`Msg.DATA`), and accordingly decide if the message should be delivered to the receiver application (Validity). Progress is also ensured, as correct senders eventually can transmit their messages (Liveness).

Besides these functional properties (related to message filtering), SIEVEQ also ensures a *resilience* property related to the detection and recovery of components of the system that exhibit faulty behavior:

Resilience. *Every component exhibiting observable faulty behavior will be eventually removed or recovered.*

In the following, we present the mechanisms for implementing the SIEVEQ functionalities, i.e., the mechanisms for satisfying the three functional properties stated above. The mechanisms for ensuring resilience will be detailed after that.

6.4.2 Message Transmission

***Sender-SQ* processing** This module gets a buffer with the `DATA` to be transmitted to a particular application placed behind SIEVEQ. The buffer needs to be encapsulated in a message with some extra information required for protection (see (6.1), below): we add a *sender-SQ* identifier S_i and a sequence number sn that is incremented on each message. This information is needed to prevent replay attacks, either from the network or from a compromised *pre-SQ*. Some information is also added to protect the integrity and authenticate the message. A signature sgn_{Si} is performed over the message contents, and a MAC

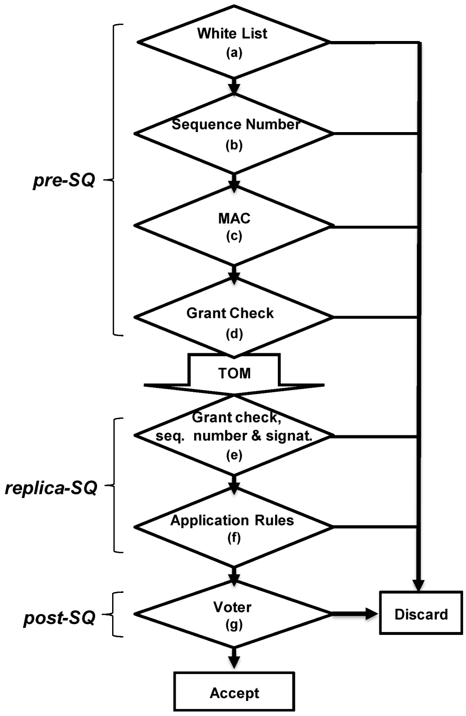


Figure 6.4 – Filtering stages at the SIEVEQ.

mac_{ski} is computed using a shared key established with the *pre-SQ*. This MAC serves as an optimization to speed up checks [37].

$$\texttt{Msg} = \langle S_i, sn, \texttt{DATA}, sgn_{Si} \rangle_{mac_{ski}} \tag{6.1}$$

After constructing the message, it is sent to the *pre-SQ* assigned to the *sender-SQ*, and a timer is started. If this timer expires before the *sender-SQ* receives an acknowledgment message, it re-sends the `Msg` to another *pre-SQ*.

***Pre-SQ* filtering** The *pre-SQ* determines if an arriving message `Msg` should be forwarded or discarded (stages (a)–(d) in Figure 6.4). It applies the following checks to make this decision:

(a) **White list:** each *pre-SQ* maintains a list of the nodes that are allowed to transmit messages (i.e., which were authorized by the system administrator). Messages coming from other nodes are dropped. This check is based on the address of the message sender, and therefore it serves as an efficient first test, but it is vulnerable to spoofing.

(b) **Sequence number:** finds out if a message with sequence number *sn* from *sender-SQ* S_i was seen before. In the affirmative case, the message is discarded to prevent replay attacks. Messages are also dropped if their *sn* is much higher than the largest sequence number ever observed from that *sender-SQ* (a sliding window of acceptable sequence numbers is used).

(c) **MAC test:** MAC mac_{ski} is verified to authenticate the message contents, including to find out if the expected S_i is the sender. If the check is invalid, the message is dropped, and the sequence number information is updated to forget that this message was ever received (the update carried out in the previous step needs to be undone).

(d) **Grant check:** each *pre-SQ* controls the amount of traffic that a *sender-SQ* is transmitting. Messages that fall outside the allocated amount are dropped to ensure that all senders get a fair share of the available bandwidth (and to avoid DoS attacks by compromised senders). The amount of traffic that is allowed to each *sender-SQ* is adjusted dynamically based on the available and consumed resources.

Finally, the *pre-SQ* invokes the TOM primitive to forward the message to every correct *replica-SQ*.

***Replica-SQ* filtering** When a correct *replica-SQ* delivers a message to be filtered, the following checks are applied:

(e) **Grant check, sequence number, and signature:** since a *pre-SQ* may have been intruded and may collude with malicious *sender-SQ*, extra checks are required on the amount of forwarded traffic and the integrity of the message. The grant check and the test on the sequence number are similar to the ones performed by the *pre-SQ*, and the signature ensures that all *replica-SQs* reach the same decision regarding the validity of the message content.

(f) **Application-level rules:** apply the application-defined filtering rules to determine if the message is compliant with the security policy of the firewall.

Although uncommon, *replica-SQs* may receive messages in a different order from what is defined in their sequence numbers. As a consequence, the *replica-SQ*'s application-level rules may drop some of the out-of-order messages, which later on will have to be re-transmitted by the *sender-SQ*. For example, if messages `A` and `B` should appear in this sequence but are re-ordered, then the rules may consider `B` invalid and then accept `A`. At some point, the *sender-SQ* would consider `B` as lost, and re-transmit it.[1]

To address this issue, each *replica-SQ* enqueues messages with a sequence number greater than the expected for a while, as long as they do not exceed a threshold above the last

[1] It is important to remark that, independently of any violation on the order of delivery of sender messages, all correct *replica-SQs* receive the messages in the same order.

processed sequence number. These messages are processed when either: 1) the missing messages with smaller sequence numbers arrive, and then they are all tested in order, or 2) the *replica-SQ* gives up on waiting and checks the enqueued messages. This last decision is made after processing a pre-determined number of other messages.

Valid messages are encapsulated in a new format (see (6.2), below) and are sent to their receivers. Basically, *replica-SQ* RS_j substitutes the signature with a new MAC, mac_{skj}. This MAC is created with a shared key between the *replica-SQ* and the *post-SQ*.

$$\texttt{Msg}' = \langle S_i, sn, \texttt{DATA}, RS_j \rangle_{mac_{skj}} \tag{6.2}$$

***Post-SQ* processing** *Post-SQ* accumulates the messages that arrive from *replica-SQs* until enough evidence is collected to allow their delivery.

(g) *vote:* A message can be delivered to the application when it is received from $f_{rs}+1$ *replica-SQs*.

Algorithm 2 presents the *post-SQ* voting protocol. The *post-SQ* starts by checking if the message contains a valid MAC (lines 5 and 6). Then, it finds out if the message carries an acceptable sequence number before storing it in a `WaitingQuorom` set (lines 7 and 8). Notice that a different set is used for each sender S_i and sn pair. Messages with sequence numbers already delivered or higher than a threshold (`snThreshold`) are discarded.

The expected sequence number is stored in the auxiliary variable k (line 9). Next, the *post-SQ* tries to find a message with this sequence number and with at least $f_{rs}+1$ votes (by searching the corresponding set and using function `equalMsg` — line 10). For each different `DATA` value that may exist in the set, the `equalMsg` function counts the number of times its appears, and returns the largest count.[2] Next, while there are `Msg`'s with a $f_{rs}+1$ quorum, *post-SQ* delivers `Msg`'s in order (line 10-13). The function `mostVotedMsg` returns the message with the `DATA` value with most votes (line 11). Then, the `deliver` function delivers `DATA` to the application on the receiver and deletes the `WaitingQuorom` set. Finally, the `snExpect` is incremented (line 12) and the k index is updated (line 13). This allows *post-SQ* to deliver messages in order while there are messages with the expected sequence number in its buffer.

[2]Notice that all correct *replica-SQs* transmit messages with the same `DATA` value and that malicious replicas can send at most f_{rs} arbitrary `DATA` values. Therefore, eventually there will be a `DATA` value with at least $f_{rs}+1$ votes because there are at least $2f_{rs}+1$ correct *replica-SQs*.

Algorithm 2: *post-SQ* protocol

Init : *executed only once*
`snExpect`$_{Si}$, $k \leftarrow 0$;
`WaitingQuorom`$_{Si,sn} \leftarrow \perp$;
Function *Filter* (`Msg’`)
    if (`verifyMAC(Msg’)` = *FALSE*) **then**
        `return` *errorMAC*;
    if (`snExpect`$_{Si} \leq$ `Msg’`.*sn* $<$ (`snExpect`$_{Si}$ + `snThreshold`)) **then**
        `WaitingQuorom`$_{Si,sn} \leftarrow$ `WaitingQuorom`$_{Si,sn} \cup \{$`Msg`$'\}$;
    $k \leftarrow$ `snExpect`$_{Si}$;
    while (`equalMsg(WaitingQuorom`$_{Si,k}$`)` $\geq f_{rs}+1$) **do**
        `deliver(mostVotedMsg(WaitingQuorom`$_{Si,k}$`))` ;
        `snExpect`$_{Si} \leftarrow$ `snExpect`$_{Si}+1$;
        $k \leftarrow$ `snExpect`$_{Si}$;

6.4.2.1 Correctness Argument

In the following, we show that SIEVEQ design satisfies the three functional properties described in Section 6.4.1. The proofs work under the assumptions of our system and threat model (defined in Section 6.3.4).

Validity. *If a correct receiver delivers a message* `Msg.DATA`*, then the message was transmitted by* `Msg.sender`*.*

Proof. Assume that a receiver R_j gets a message content `Msg.DATA` with the `Msg.sender` $=$ `Msg.`S_i. For this to happen, the *post-SQ* of R_j waited for the arrival of at least $f_{rs}+1$ correctly authenticated messages[3] with equal `Msg.`S_i, `Msg.`*sn*, and `Msg.DATA`. Therefore, at least $f_{rs}+1$ *replica-SQs* received `Msg` signed by S_i, and verified the signature sgn_{Si} as valid. This indicates that `Msg.DATA` was not modified by the network or by any *pre-SQ*. Only a sender `Msg.`S_i (correct or not) can create and authenticate a `Msg` with its signature. Therefore, S_i is the `Msg.sender`, i.e., the creator of `Msg.DATA`. □

Compliance. *If a message, transmitted by a correct sender, is delivered to a correct receiver then the message is in accordance with the security policy of* SIEVEQ.

Proof. The proof of the *Compliance* property is very similar to the *Validity* proof. The difference is that, we also know that if $f_{rs}+1$ *replica-SQs* sent `Msg` to a *post-SQ*, then `Msg`

[3]Note that *replica-SQs* send `Msg`$'$ (defined in (6.2)) instead of `Msg` (defined in (6.1)). Both are similar, but `Msg`’ has a MAC instead of a signature to authenticate its contents with *post-SQ*. For the sake of simplicity we will only use the `Msg` notation in the rest of the proof.

was verified against the security policy in at least one correct *replica-SQ*, which approved it. □

Liveness. *If a correct sender sends a message, then the message eventually will be delivered to the correct receiver.*

Proof. For the sake of simplicity, our proof only considers nodes that drop or delay messages. Attacks to the integrity will make the message be discarded (i.e., dropped).

Assume that a sender S_i transmits a message `Msg` with the sequence number *sn* to a *pre-SQ* PS_u. Then S_i sets a timer $timer_{sn}$ for `Msg`.*sn*. If PS_u is correct, after it receives a message, it re-sends `Msg` via TOM to the *replica-SQs*. At least $f_{rs}+1$ correct *replica-SQs* will send `Msg` to the *post-SQ*, which will deliver `Msg.DATA` to the receiver R_j. If the PS_u is faulty, i.e., drops or delays messages, $timer_{sn}$ in S_i will eventually expire. When this happens, S_i re-transmits `Msg` to another *pre-SQ*. Eventually, this process will make some correct *pre-SQ* forward `Msg` to the *replica-SQs* using the TOM primitive, which will cause `Msg.DATA` to be delivered to R_j. □

6.4.3 Addressing Component Failures

In this section, we discuss the implications of failures in the different components of the SIEVEQ, and how they are handled for ensuring the *Resilience* property stated in Section 6.4.1.

In the Byzantine model, every failed component can behave arbitrarily, intentionally or accidentally. Therefore, the SIEVEQ design incorporates mechanisms that are resilient to different failure scenarios. Given the architecture of Figure 6.3, one has to address faults in authenticated *sender-SQs*, *pre-SQs* and *replica-SQs*, as they are the main components subject to Byzantine failures in our model. *Post-SQ* is not considered in this section as it is co-located with the receiver. Therefore, we can not make further assumptions about it. In this chapter, we assume the *controller* as trusted for simplicity. In Chapter 5 we presented an intrusion-tolerant *controller*

In the following, we try to focus on complex scenarios whereas faulty components send syntactically-valid messages. Therefore avoiding cases that could be easily detected and recovered by existing network monitoring and protection tools (e.g., it is easy to discover that a *pre-SQ* is sending messages to another *pre-SQ*, something our protocol does not allow).

Since *pre-SQs* are directly exposed to the external network, there is a higher risk of them being compromised. Then, to keep the SIEVEQ operational, it is required that failed *pre-SQs* be identified and recovered. We leverage from the *replica-SQ* setup to perform failure detection, and then use the *controller* to restart erroneous *pre-SQs*. Replacing these components is almost trivial because they are stateless.

Replica-SQs execute as a BFT replicated state machine, processing messages in the same order and producing identical results. Consequently, *replica-SQs* faults can be tolerated by employing a voting technique on the *post-SQ* that selects results supported by a sufficiently large quorum (as explained above, an output with at least $f_{rs}+1$ votes). Below, we discuss in more detail a few failure scenarios.

6.4.3.1 Faulty *Sender-SQ*

A faulty *sender-SQ* is authorized to communicate while suffering from some arbitrary problem (e.g., intrusion). Therefore, it can produce correctly authenticated messages to attack the firewall. In some scenarios, it is possible to discard these messages. For example, if the SIEVEQ receives a correctly signed message with a sequence number is higher than what was expected, then it can be easily detected and eliminated (checks *(b)* and *(e)* in Figure 6.4).

A more demanding scenario occurs when a *sender-SQ* transmits faster than the allowed rate (verification (d) of Figure 6.4). In this case, some defense action has to be carried out, as these attacks can lead SIEVEQ to waste resources. To be conservative, we decided to follow a simple procedure to protect the firewall: SIEVEQ maintains a counter per sender that is incremented whenever new evidence of failure can be attributed to it, e.g., the faulty *sender-SQ* is overloading the system with invalid messages or if it is sending messages to *pre-SQs* which it was not assigned. When the counter reaches a pre-defined value, the *sender-SQ* is disallowed from communicating with SIEVEQ by temporarily removing it from the whitelist and adding it to a quarantine list (failing verification (a) of Figure 6.4) and by giving a warning to the system administrator. *This ensures that faulty sender-SQs are eventually removed from the system.*

Excluded *sender-SQs* may regain access to the service later on because the counter is periodically decreased (when the counter falls below a certain threshold, the *sender-SQ* is moved back into the whitelist). Additionally, the administrator is free to update the white/quarantine lists. For instance, he may choose to manually add a faulty sender to the quarantine list or even deploy policies to do that under certain conditions (e.g., if the same sender is in the quarantine list for a certain number of times).

6.4.3.2 Faulty *Pre-SQ*

Addressing failures in *pre-SQs* is difficult because these components may look as compromised even when they are correct. Notably, when a *pre-SQ* is under a DoS attack, messages can start to be dropped due to buffer exhaustion, and this is indistinguishable from malicious behavior in which messages are selectively discarded. In the same way, a failed signature check at a *replica-SQ* indicates that either the *pre-SQ* is faulty (it is tampering/generating invalid messages) or that a *sender-SQ* is misbehaving (recall that a *pre-SQ* verifies the message MAC, but not its signature). Finally, a *replica-SQ* may also detect problems if it observes a sudden increase in the arrival of messages (above the grant check), which could indicate a DoS attack by a malicious *pre-SQ* (maybe colluding with a compromised *sender-SQ*). This kind of ambiguity precludes accurate failure detection, and consequently, we aim to provide a mechanism that allows SIEVEQ to recover from end-to-end problems and continue to deliver a correct service.

An initial step to deal with these complex failure scenarios is to make *pre-SQs* evaluate their own state. This is done by analyzing the amount of arriving traffic and by observing if it could overload the *pre-SQ*. The analysis can be done by measuring the inter-arrival times of messages over a specified period. If those intervals are small (on average), there is a chance that the *pre-SQ* is working at its full capacity or is even overloaded. When this happens, the *pre-SQ* broadcasts (using the TOM primitive) a `WARNREQ` message to the *replica-SQs*, so that they may take some action to solve the problem (see below).

When the *pre-SQ* is faulty, the *sender-SQ* and *replica-SQs* need to detect it together. SIEVEQ provides a procedure to find how many messages are being discarded on *pre-SQ*:

1. Periodically, the *sender-SQ* sends a special `ACKREQ` request to *replica-SQs*, in which it indicates the sequence number of the last message that was sent (plus a signature and a MAC). This request is first sent to the preferred *pre-SQ*, but if no answer is received within some time window, and then it is forwarded to another *pre-SQ*. The waiting period is adjusted in each retransmission by doubling its value.

2. When the *replica-SQs* receive the request, the included sequence number together with local information is used to find how many messages are missing. The local information is the set of sequence numbers of the messages that were correctly delivered since the last `ACKREQ`.

3. Based on the number of missing messages, the *replica-SQs* transmit through the same *pre-SQ* a response `ACKRES` to the sender, where they state the observed failure rate and other control information (plus a signature). *Replica-SQs* may also perform some recovery action if the failure rate is too high.

Additionally, if a *replica-SQ* detects that a signature is invalid or that it is receiving more messages than the expected (check (e) in Figure 6.4), it suspects the *pre-SQ* that sent the messages and takes some action.

Once the problem is detected, the *replica-SQs* should attempt to fix the erroneous behavior by employing one of three possible remediation actions, depending on the extent of the perceived failures:

- *Redistribute load:* if a *pre-SQ* has sent a warning about its load, or a high failure rate related with this component is observed, the first course of action is to move some of the messages flows from the problematic component to other *pre-SQ*. This is achieved by specifying, in the `ACKRES` response to a *sender-SQ*, the identifier of a new *pre-SQ* that should be contacted. At that point, the *sender-SQ* is expected to connect to the indicated *pre-SQ* and begin sending its traffic through it.

- *Increase the pre-SQs capacity:* if the existing *pre-SQs* are unable to process the current load, then the SIEVEQ needs to create more *pre-SQs* (depending on the available hardware resources). To do that, *replica-SQs* contact the *controller* informing that an extra *pre-SQ* should be started. When the controller receives $f_{rs}+1$ messages, it performs the necessary steps to launch the new *pre-SQ* (which are dependent on the deployment environment). The new *pre-SQ* begins with a few startup operations, which include the creation of a communication endpoint, and then it uses the TOM channel to inform the *replica-SQ* that it is ready to accept messages from *sender-SQs*.

- *Kill the pre-SQ:* when there is a significant level of suspicion on a *pre-SQ*, the safest course of action is for *replica-SQs* to ask the controller to destroy it. Moreover, if the load on the firewall is perceived as having decreased substantially, the *replica-SQs* select the oldest *pre-SQ* for elimination, allowing future aging problems to be addressed. The controller carries out the needed actions when it gets $f_{rs}+1$ of such requests (once again, which depend on how SIEVEQ is deployed). The affected *sender-SQs* will be informed about the *pre-SQ* replacement through the `ACKREQ` mechanism, i.e., they will eventually use another *pre-SQ* to send a request, and get the information about their newly assigned *pre-SQ* in the response. Moreover, the new *pre-SQ* is informed about the expected sequence number for each *sender-SQ*. This information is stored by *replica-SQs*, which contrary to *pre-SQs* are stateful.

Together, these mechanisms ensure that *a faulty pre-SQ affecting the* SIEVEQ *performance will be eventually removed from the system.*

6.4.3.3 Faulty *Replica-SQ*

A *post-SQ* only delivers a message if it receives $f_{rs}+1$ matching approvals for this message. Given the number of *replica-SQs*, this quorum is achievable for a correct message even if up to f_{rs} *replica-SQs* are faulty. However, a faulty *replica-SQ* can create a large number of messages addressed to other components of the system, effectively causing a DoS attack. Therefore, we need countermeasures to disallow a faulty *replica-SQ* to degrade the performance of the system in a similar way as illustrated in Figure 6.2. For example, a faulty *replica-SQ* can send an unexpected amount of messages to a *pre-SQ*, making it slower and triggering suspicions that may lead it to be killed. Similarly, it can attack other *replica-SQs* to make the system slower. Finally, a faulty *replica-SQ* can also overload the *post-SQ* with invalid messages.

Each of these attacks requires a different detection and recovery strategy:

- When a *pre-SQ* is being attacked it complains to the controller, which first requests the *pre-SQ* replacement (assuming it might be compromised) and increases a suspect counter against the *replica-SQ*. If $f_{ps}+1$ *pre-SQs* also complain about the same *replica-SQ*, the controller starts an *individual recovery* in this *replica-SQ* (see below).

- If more than f_{rs} other *replica-SQs* are being attacked, they will probably become slower. In this case, it is possible to detect such attacks if $f_{rs}+1$ *replica-SQs* complain about a single *replica-SQ*. This would be possible because target *replica-SQs* would observe unjustifiable high traffic coming from a single replica and because such spontaneously generated messages would be deemed invalid. When this attack is detected, the controller starts an individual *replica-SQ* recovery.

- If only up to f_{rs} other *replica-SQs* are being attacked it is still possible to make the system slower without being detected by the previous mechanism. This happens because $N_{rs}-f_{rs}$ replicas must participate in the TOM protocol [20], and the target f_{rs} plus the attacker intersect this quorum in a least one component. This intersecting component will define the pace of the messages coming, which typically will be slow. We can mitigate this attack as each *replica-SQ* periodically informs the *post-SQ* about its throughput (number of processed messages per second), piggybacking this value in some approved messages submitted for voting. The *post-SQ* verifies that there are *replica-SQs* presenting throughputs lower than expected and initiates a *recovery round* on all the *replica-SQs* (as described below).

- When the *post-SQ* is being attacked it detects the abnormal behavior. Then, it requests the individual recovery of the compromised replica.

The previous mitigation mechanisms suggest two kinds of recovery actions. First, an individual recovery, where the machine is rebooted with clean code (we addressed this in Chapter 4) and then is reintegrated in the system. Second, a recovery round is used when a performance degradation attack is detected, but there is no certainty about which *replica-SQ* was compromised.

The integration of these reactive recovery mechanisms into LAZARUS' risk management can be assessed by drastically increasing the individual risk of each replica (e.g., similarly to the *age* value of reach replica). Then, LAZARUS proceeds with the recovery of the faulty replica.

6.5 Implementation

We implemented a prototype of SIEVEQ following the specification of the previous section to validate our design. The *sender-SQ* and *post-SQ* were developed as libraries that are linked with the sender and receiver applications. The libraries offer an interface similar to the TCP sockets, to simplify the integration and minimize changes both in the client and server applications. A sender can provide a buffer to be transmitted, and the receiver can indicate a buffer where the received data is to be stored. Internally, the *sender-SQ* library adds a MAC and a signature to each message. The MAC is created using Hash-based Message Authentication Code (HMAC) with the Secure Hash Algorithm (SHA)-256 hash function and a 256-bit key, and the signature employs Rivest–Shamir–Adleman (RSA) with 512-bit keys. Messages are authenticated at the *post-SQ* also using HMAC. The RSA keys were kept relatively small for performance reasons. However, since they should be updated periodically (e.g., every few hours), this precludes all practical brute force attacks [2].

The *pre-SQs* operate as separate processes receiving the messages and forwarding them to the *replica-SQs*. We resorted to BFT-SMART, a SMR library [20], to implement the TOM and manage the *replica-SQs*. BFT-SMART, like most SMR systems (e.g., PBFT [30], ZYZZYVA [113], PRIME [5]) follows a client-server model, where a client transmits request messages to a group of server replicas and then receives the output messages with the results of some computation. We had to modify BFT-SMART because this model does not fit well with the message flow of SIEVEQ. Therefore, we have decoupled the client-server model into a client-server-client model. The client sends messages (but does not wait for responses as in the traditional SMR), and the server forwards them (after the validation) to another client, which is the last receiver. A reverse procedure is carried out for the traffic originating from the receiver. Furthermore, in BFT-SMART, the replicated servers are typically designed with a single-thread to process requests. To improve performance, we also modified the system to allow CPU-costly operations (like a signature verification) to occur concurrently with the rest of the checks performed by *replica-SQs*.

In the prototype, a *pre-SQ* can be replaced on two occasions: first, voluntarily by asking for a substitution to the *replica-SQs*, when it is flooded with unauthorized messages (DoS attack); and second, when a *replica-SQ* detects message corruptions by a *pre-SQ*. In both situations, the *replica-SQs* make a request to the *controller*, which will replace a *pre-SQ* instance. Notice that the *controller* has to wait for $f_{rs} + 1$ messages, requesting a *pre-SQ* replacement, to ensure that a faulty *replica-SQ* cannot force the recovery of a correct *pre-SQ*. The *replica-SQs* are recovered when the collected information indicates that some component might be attacking other components. The information is collected and sent to the trusted controller to evaluate and decide if the replicas are making progress as expected. The information allows the system to suspect on $f_{rs} + 1$ *replica-SQs* and recover them (there is no need to recover all the *replica-SQs*).

Since BFT-SMaRt is programmed in Java, we decided to use the same language to develop the various SieveQ components. If the sender and receiver applications are coded in other languages, they can still be supported by implementing specific *sender-SQ* and *post-SQ* libraries.

6.6 Evaluation

In this section, we present the evaluation of SieveQ under different network and attack conditions. We present the results of four types of experiments. In the first one, we evaluate the latency for different *sender-SQ* workloads, assessing the performance of SieveQ in the absence of failures. The second experiment assesses the effect of filtering rules complexity on the performance of the system. In the third experiment, we assess the throughput in three scenarios: *i)* the normal case, *ii)* a DoS attack without countermeasures; and *iii)* a DoS with all the resilience mechanisms enabled (in fact we considered two DoS attacks: external, from a malicious *sender-SQ*, and internal, from a compromised *replica-SQ*). The last experiment considers the capability of SieveQ to safeguard a SIEM system under a similar workload as the one observed in the 2012 Summer Olympic Games.

6.6.1 Testbed Setup

Figure 6.5 illustrates the testbed, showing how the various SieveQ components were deployed in the machines. We consider one *sender-SQ* and one *post-SQ* deployed in different physical nodes, and an additional host acted as a malicious external adversary. The *controller* and *pre-SQs* were located in the same physical machine for convenience but in different VMs. Four *replica-SQs* were placed in distinct physical nodes. Every machine had two Quad-core Intel Xeon 2.27 GHz CPUs, with 32 GB of memory, and a Broadcom NetXtreme II Gigabit network card. All the machines were connected by a 1Gbps switched network and run Ubuntu 10.04 64-bit LTS (kernel 2.6.32-server) and Java 7 (1.7.0_67).

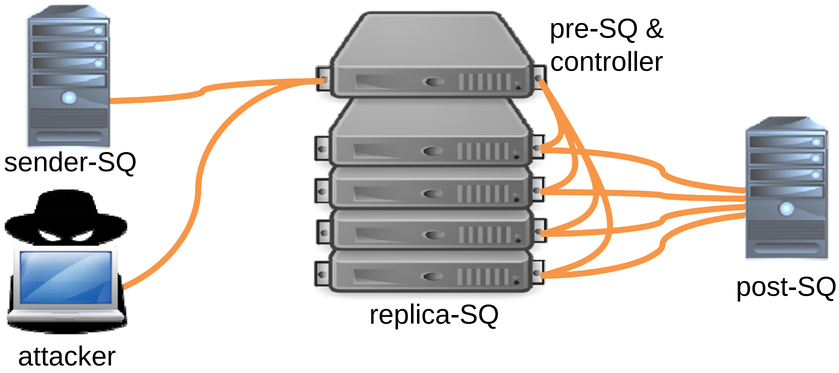


Figure 6.5 – The SIEVEQ testbed architecture used in the experiments.

6.6.2 Methodology

SIEVEQ acts as a highly resilient protection device, receiving messages on one side and forwarding them to the other side. Therefore, the performance of SIEVEQ is assessed with latency/throughput measurements that can be attained under different network loads. In the experiments, the sender transmits data at a constant rate, i.e., 100 to 10000 messages per second, with three different message payload sizes, i.e., 100 bytes, 500 bytes, and 1k bytes. We used the Guava library [77] to control the message sending rate.

Latency measures the time it takes to transmit a message from *sender-SQ* until it is delivered to the application on the *post-SQ*. The following procedure was employed to compute the latency: *sender-SQ* obtains the local time before transmitting a message. When the message arrives and is ready to be delivered to the receiver application (after voting), the *post-SQ* returns an acknowledgment over a dedicated User Datagram Protocol (UDP) channel. The *Sender-SQ* gets the current time again when the acknowledgment arrives. The latency of a message is the elapsed time calculated at *sender-SQ* (receive time minus send time) subtracted by the average time it takes to transmit a UDP message from *post-SQ* to *sender-SQ*.

Throughput gives a measure of the number of messages per second that can be processed by SIEVEQ. It was calculated at the *post-SQ* using a counter. This counter is incremented every time a message is delivered to the receiver application, and the counter is reset to zero after one second. Consequently, the server can calculate the number of messages delivered every second. The throughput is computed as the average value of the individual measurements collected over a period of time (in our case, 5 minutes after the steady state was reached).

In some experiments, we wanted to assess the behavior of SIEVEQ under a DoS attack. The attack was made using `PyLoris` [150], a tool built to exploit vulnerabilities on TCP connection handling. The tool implements the Slowloris attack method, which opens many TCP connections and keeps them open. The tool allows the user to define parameters like

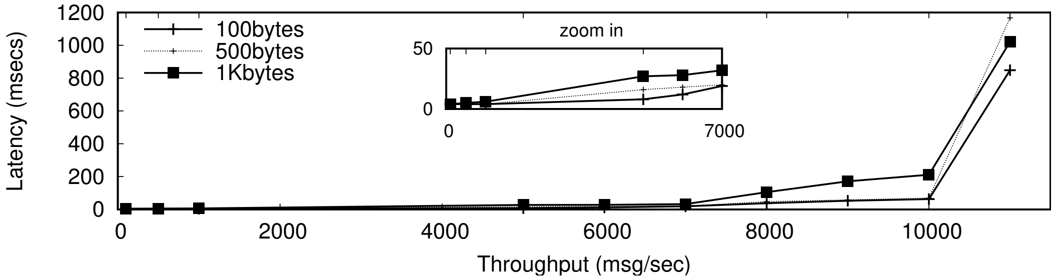


Figure 6.6 – SIEVEQ latency for each workload (message size and transmission rate).

Payload size (*bytes*)	**Non-optimized** (*msg/sec*)	**Optimized** (*msg/sec*)
100	1360	10682
500	1320	10618
1000	1311	10332

Table 6.1 – Maximum load induced by the *sender-SQ* library with various message sizes

group size of attack threads, the maximum number of connections, and the time interval between connections among others. In our setup, `PyLoris` was configured to perform an unlimited number of connections, with 0.1 milliseconds between each connection.

In all experiments, measurements were taken only after the JVM was warmed-up, and the disks were not used (all data is kept in memory).

6.6.3 Performance in Failure-free Executions

This experiment measures the latency of SIEVEQ with several message sizes and distinct message transmission rates. It demonstrates the overall performance of SIEVEQ in different scenarios, gradually stressing the *post-SQ* side as the workload is slowly increased. Measurements were collected after the system reached a steady state. The experiments were repeated 10 times for every workload, and the average result is reported.

Figure 6.6 shows how the latency is affected by the transmission rate and message size. As expected, when the system becomes increasingly loaded, the latency grows proportionally because resources have to be shared among the various messages. The latency increase is approximately linear for the messages with 100 and 500 bytes until the throughput reaches $10k$ messages per second. The messages with $1k$ bytes have a linear increase on latency until the throughput reaches approximately $7k$ messages per second, and then it has a higher increase as more load is put on the system. This means that very high workloads can only be supported if applications have some tolerance to network delays. Overall, the performance degrades gracefully when varying the message payload sizes, for rates under $7k$ messages per second.

We performed a more detailed analysis of the overheads introduced by the various components of SIEVEQ. We observed that *sender-SQ* performs the most expensive operations, which is interesting because it shows that our design offloads part of the effort to the edges, reducing bottlenecks. The most significant overheads were caused by the tasks associated with securing the message payload (which are the most costly operations in the system). Several optimizations were made to mitigate the performance penalties during the message serialization (e.g., the creation of the signature), including the use of parallelization to take advantage of the multicore architecture (as done, for example, in [108]). Table 6.1 shows the gain of using the optimized version of the *sender-SQ* library. Similar optimizations were employed in other components.

6.6.4 Effect of Filtering Rules Complexity

The results presented in the previous section do not consider any complex filtering rule set. This section presents experiments with a fully loaded system and application-level filtering rules with different complexity at the *replica-SQs*.

Given the diverse requirements imposed by application-level firewalls, we decided to approximate the complexity of the filtering rules by considering a variable number of string matchings on processed messages. It is well recognized that the most expensive aspect of message filtering is exactly finding (or not) specific strings in the packet contents (besides crypto verifications, which were included in all our experiments). The high cost of running such algorithms leads for instance to several implementations in Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) to improve performance in firewalls and IDS (e.g., [117, 132]).

We used a classical algorithm for string matching (Knuth–Morris–Pratt (KMP) [111]) at the *replica-SQs* to implement the filtering rules. This algorithm is employed in IDSs [149] and firewalls like `iptables` [138]. The algorithm employs a pre-computed table to execute string matching in $O(n+k)$ complexity, where n is the string length, and k is the pattern length.

We measured the latency of the system by varying the message size from 100 to 1000 bytes and the string pattern size from 5 to 20 bytes. Both the message content and the strings were randomly generated. The experiments were performed with the maximum throughput of 10000 messages per second, as identified in the previous section. Figure 6.7 shows the latency of SIEVEQ without message filtering (Baseline) and string matching (KMP) of different sizes. In the last group of experiments, where the message size is 1000 bytes, and the pattern is 20 bytes, the latency increases by 50%. In other cases, sometimes higher overheads were observed, e.g., with 500 bytes messages and 20 bytes pattern the overhead is approximately 200%. This result is expected as the string matching is a slow operation

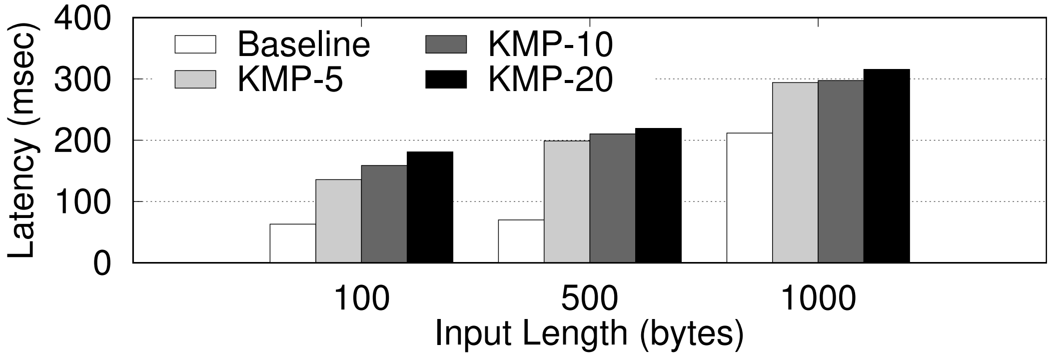


Figure 6.7 – Comparison of the SIEVEQ's latency between the baseline and adding the filtering rules.

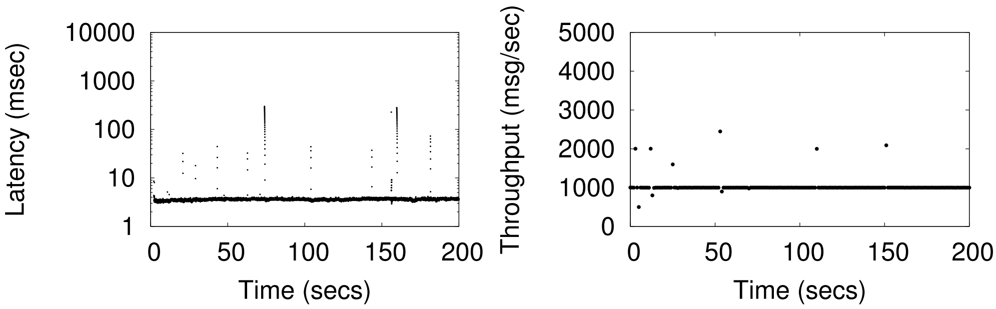
and it needs to be performed in the critical path of message processing. Implementations with hardware support (as mentioned before) could be integrated with SIEVEQ to reduce these delays significantly.

6.6.5 SIEVEQ Under Attack

Our next set of experiments aims to evaluate the system under different attack scenarios. Here, the *sender-SQ* creates a steady load of 1000 500-byte messages per second. We measured the latency and throughput of the system in three conditions: failure-free operation, a malicious external and internal DoS attack, and a malicious attack with remediation mechanisms. Before presenting the results, we need to stress that these experiments must be compared with the results displayed in Figure 6.2, which were obtained in the same way but with a different architecture (see Figure 6.1).

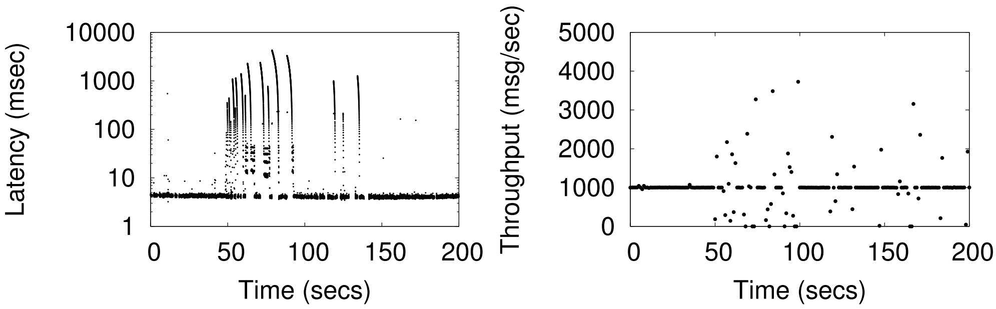
Figure 6.8a and Figure 6.8b show the latency and throughput, in the failure-free scenario. One can observe that latency stays on average around 3.4 milliseconds. The throughput is approximately constant during the whole period. It is possible to observe some momentary spikes in the latency and throughput, which happens due to Java garbage collector and a queuing effect from the SMR.

The behavior of the SIEVEQ during a DoS attack is displayed in Figure 6.9a and Figure 6.9b. In this scenario, we have disabled the SIEVEQ capability of replacing *pre-SQs* at runtime. The attack consists in stressing the TCP socket interface of the *pre-SQs* by creating many TCP connections, which consumes network bandwidth and wastes resources at system and application levels. When the attack is started, it executes for 50 seconds. The latency graph displays a reasonable impact regarding an increase in the delays for message delivery. In some cases, the latency is not too affected, but in others, there is a drastic delay, with some messages taking more than 3 seconds. The attack also has consequences on the throughput as it is possible to observe an oscillation between 0 to 4000 messages per second (which



(a) Baseline latency. **(b)** Baseline throughput.

Figure 6.8 – Performance of SIEVEQ in fault-free executions.



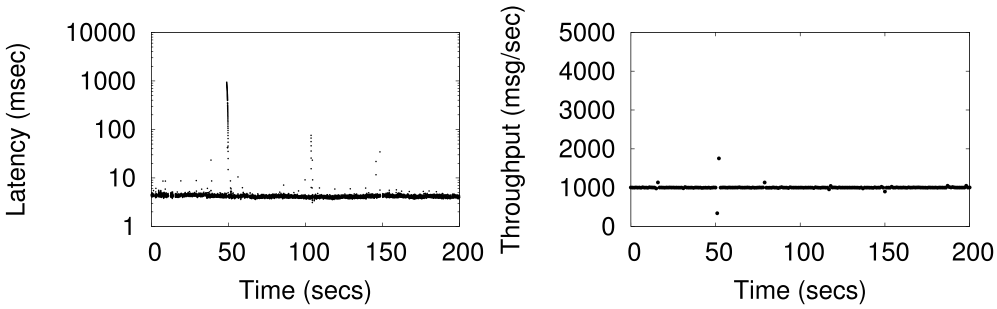
(a) DoS w/ no recovery. **(b)** DoS w/ no recovery.

Figure 6.9 – Performance of SIEVEQ under DoS attack conditions.

correspond to the situation when the *post-SQ* processes a batch of messages that have been accumulated).

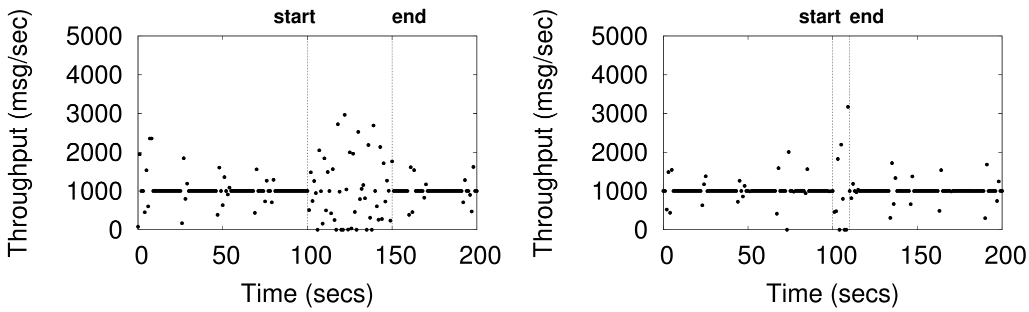
Figure 6.10a and Figure 6.10b show the latency and throughput when a similar DoS was carried out, but in this case the SIEVEQ replaced the *pre-SQ* under attack with a new *pre-SQ*. When the *pre-SQ* finds out that it is being overloaded with messages coming from non-authorized senders, it asks for a replacement. After that, the controller replaces the faulty *pre-SQ*, and the existing *sender-SQs* are contacted to migrate their connections. As the figures show, the impact of the attack is minimized, since only a few messages are delayed, and throughput is only affected momentarily while the *pre-SQ* is switched. Once the new *pre-SQ* takes over, the messages lost during the switching period are retransmitted and delivered. In practice, the attack becomes ineffective because, although it continues to consume network bandwidth, there is no longer a *pre-SQ* to process the malicious messages. An adversary could increase the attack sophistication and try to find a new *pre-SQ* target. However, even in this case, the attack has a limited effect because during an interval of time (while there is a search for a fresh target) the system can make progress.

Figure 6.11a and Figure 6.11b shows the SIEVEQ throughput when an internal attack is carried out by a compromised *replica-SQ*. The experiment was made with a *replica-SQ* ($r1$) launching a DoS to another *replica-SQ* ($r2$). The attack consists in overloading a *replica-SQ*



(a) DoS w/ recovery. **(b)** DoS w/ recovery.

Figure 6.10 – Performance of SIEVEQ under DoS attack conditions with recovery.



(a) Internal DoS execution without recovery actions. **(b)** Internal DoS execution with recovery actions.

Figure 6.11 – Performance of SIEVEQ under internal DoS attack conditions without and with recovery.

with *state transfer* requests, which are the most demanding request a replica can receive in BFT-SMART [19]. Figure 6.11a shows the impact on the SIEVEQ throughput during an attack lasting 50 seconds, without any recovery capability on the system. As can be seen, the performance of the system is severely disrupted during the attack. Figure 6.11b shows the same attack but with the detection and recovery mechanism described in Section 6.4.3.3. The *post-SQ* detects the problem by noticing that $f_{rs}+1$ *replica-SQs* are sending fewer messages than the others and then requests a recovery. When the *replica-SQ* is recovered, it requests the state from the other replicas, and then after applying the new state, the replica resumes the normal execution (end line in the figure). In the experiment of Figure 6.11b we show a case in which the faulty *replica-SQ* is the first to be recovered. It could happen that SIEVEQ recovered f_{rs} *replica-SQs* before the faulty one. This would take $(f_{rs}+1)\times 3$ seconds (in our setup) before the system resumes the normal execution.

6.6.6 SIEVEQ to Protect a SIEM System

SIEM systems offer various capabilities for the collection and analysis of security events and information in networked infrastructures [127]. Organizations are employing these systems as a way to help with the monitoring and analysis of their infrastructures. They

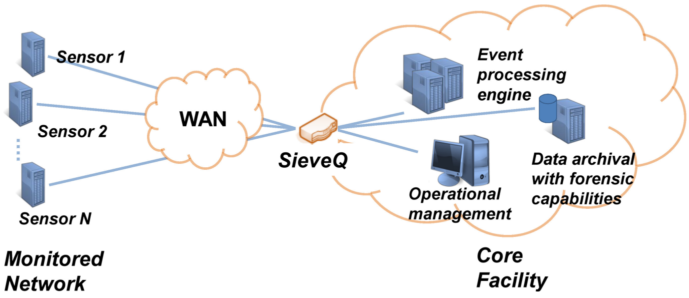


Figure 6.12 – Overview of a SIEM architecture, showing some of the core facility subsystems protected by the SIEVEQ.

integrate an extensive range of security and network capabilities, which allow the correlation of thousands of events and the reporting of attacks and intrusions in near real-time.

A SIEM operates by collecting data from the monitored network and applications through a group of sensors, which then forward the events towards a correlation engine at the core facility. The engine performs an analysis of the stream of events and generates alarms and other information for post-processing by other SIEM components. Examples of such components are an archival subsystem for the storage of data needed to support forensic investigations, or a communication subsystem to send alarms to the system administrators.

As part of the MASSIF European project [184], we have implemented a resilient SIEM system, where SIEVEQ was used to protect the access to the core facility (in Figure 6.12). In the SIEVEQ architecture, the sensors had the *sender-SQ* while the *post-SQ* was placed in the correlation engine. Additionally, we had access to an anonymized trace with the security events collected during the 2012 Olympic Games. Each event corresponds basically to a string describing some observed problem by a sensor. The strings of text had lengths varying between a minimum of 551 bytes and a maximum of 2132 bytes, with an average length of 1990 bytes (and a standard deviation of 420 bytes). Based on this log, we built a sensor emulator that generates traffic at a pre-defined rate. Basically, when it is time to produce a new event, the emulator selects an event from the trace and feeds it to the *sender-SQ*.

During the 2012 Olympic Games, the workload was approximately 11 million events per day, i.e., around 127 events per second. Figure 6.13 shows the latency imposed by SIEVEQ for this workload, and when it is scaled up from 2 to 16 times more. As can be observed, the latency is in the order of 4 milliseconds for the emulated scenario. Even with the highest load (16 times), the observed values had a latency below 70 milliseconds. This means that SIEVEQ could potentially deliver on average 176 million events per day, which is more than enough to accommodate the expected growth in the number of events for the next Olympic Games[4].

[4] In 2016 the Rio's Olympic Games the number of events per second was approximated 174, source: `https://diginomica.com/2016/11/24/securing-the-olympics-lessons-for-enterprise-cyber-security/`

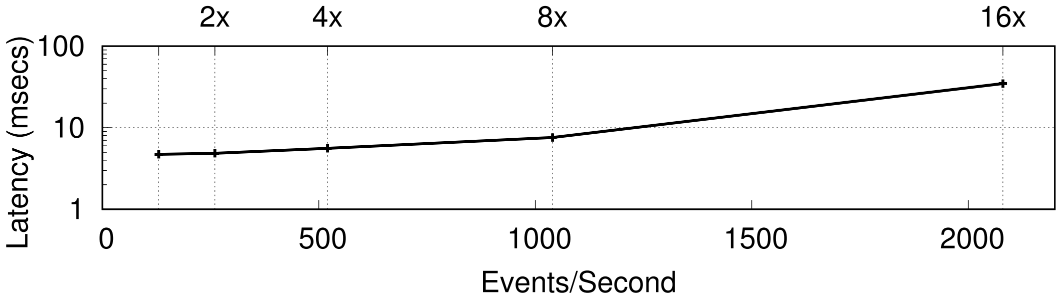


Figure 6.13 – SIEVEQ latency for the 2012 Summer Olympic Games scenario throughput requirement (127 events per second) and how the SIEVEQ scale, we doubled on each experiment the number of events.

6.7 Discussion

SIEVEQ differs from standard firewalls like `iptables` [138] that do not need client- or server-side code modifications. However, our system requires these modifications to ensure end-to-end message integrity and tolerance of compromised components. Complete transparency would be hard to achieve mainly due to the use of voting. Nonetheless, it is worth stressing that SIEVEQ is to be used as an additional protection device in critical systems, not as a substitute to regular L3 firewalls. SIEVEQ provides a protection similar to an application-level/L7 firewall, as one can implement arbitrary rules on the *replica-SQ* module.

State machine replication is a well-known approach for replication [162]. In this technique, every replica is required to process requests in a deterministic way. This requirement traditionally implies in two limitations: (1) replicas cannot use their local clock during request processing, and (2) all requests are executed sequentially. The first limitation can affect the capacity of SIEVEQ to process rules that use time. We remove this limitation by making use of the timestamps generated by the leader replica and agreed upon on each consensus, as proposed in PBFT [30] and implemented in BFT-SMART [20]. The second limitation can constraint the performance of the system, especially when CPU-costly operations such as signature verifications are executed. One of the optimizations we implemented in SIEVEQ was to add multi-threading support in the BFT-SMART replicas. More precisely, the signature verification is done by a pool of threads that either accept or discard messages. Once accepted, a message is added to a processing queue following the order established by the total order multicast protocol. A single thread consumes messages from this queue, verifies them against the security policy, updates the firewall state (if needed) and forwards them to their destinations, without violating the determinism requirement.

6.8 Final Remarks

We presented SIEVEQ, a new intrusion-tolerant protection system for critical services, such as Industrial Control Systems (ICS) and SIEM systems. Our system exports a message queue interface which is used by senders and receivers to interact in a regulated way. The main improvement of the SIEVEQ architecture, when compared with previous systems, is the separation of message filtering in several components that carry on verifications progressively more costly and complex. This allows the proposed system to be more efficient than the state-of-the-art replicated firewalls under attack. SIEVEQ also includes several resilience mechanisms that allow the creation, removal, and recovery of components in a dynamic way, to respond to evolving threats against the system effectively. Experimental results show that such resilience mechanisms can significantly reduce the effects of DoS attacks against the system.

7 Conclusion and Future Research Directions

7.1 Conclusions

The correctness of BFT systems is tied to the assumption that replicas fail independently. Otherwise, once a replica is compromised, the next $f+1$ replicas would be compromised within virtually the same time. In this thesis, we addressed the long-standing open problem of many works over the past 20 years of BFT replication research: evaluate, select, and manage the failure independence (through diversity) of BFT replicas. That said, we developed a control plane that maximizes the failure independence of BFT replicated nodes practically and effectively to make them more resilient to malicious adversaries.

The first step to achieve this objective was, following the line of prior work to this thesis, to validate the diversity hypothesis. To this end, we leverage on the study using NVD data about the shared vulnerabilities among OSes and conducted an analysis more focused on the deciding which OSes provide dependable configuration sets for replicated systems. In particular, we presented several strategies to choose the most diverse OSes. These strategies comprise different approaches to the data depending on whether one (i) considers all common vulnerabilities as being of equal importance; (ii) place greater emphasis on more recent common vulnerabilities; or (iii) is primarily interested in the common vulnerabilities being reported less frequently in calendar time. All strategies delivered the same best combination of OSes, and this emphasizes the importance of carefully select OSes that minimize the number of common weaknesses.

Despite the good results that we have achieved in the previous contribution, we have identified some problems that lead us to pursue further analysis. Namely, our and a few other works that used NVD as a primary data source were missing essential data. In particular, NVD does not report all products that are affected by a particular vulnerability. We addressed this issue by using clustering techniques to group similar vulnerabilities. Moreover, we added other data sources with other relevant data (that NVD does not provide) to enrich our data. We devised a new metric that uses the vulnerability attributes provided by NVD and CVSS, and other attributes added from additional OSINT sources. This metric is used by an algorithm that builds and reconfigures replica sets on BFT systems. The algorithm decides when and which replica should be replaced by estimating the risk of the BFT system being vulnerable and potentially compromised. The results show that our algorithm supports better decisions when selecting OSes to run in the replicated system. For example, for the evaluated time it *compromised rate* varies between 2%-5% of the total of executions while a random selection varies between 98%-99%.

Besides the lack of proof for supporting diversity, most of the works did not address diversity in a practical manner. We implemented LAZARUS to manage BFT systems according to security data available online that is then analyzed to trigger replicas' recoveries. We conducted an extensive evaluation of the costs of using real OS diversity in BFT systems. In the experiments, we have found a significant limitation on the virtualization technology, forcing us to limit the evaluation setup by reconfiguring the bare metal machines (e.g., limit the number of CPUs). Therefore, we manage to make a fair comparison (for some of the cases) between a bare metal configuration with a virtualized (yet limited) solution that accommodates LAZARUS. Although most of the results show a non-negligible overhead, we have shown that for OSes that did not suffer virtualization limitations the overhead is minimal. In these cases, the performance of the replicated system managed by LAZARUS is 86%-94% of the bare metal, which runs a replicated non-virtualized homogenous OS configuration.

One of the BFT systems used in the LAZARUS evaluation, is SIEVEQ. It is a multi-layer firewall-like application that was designed to absorb most of the external and internal attacks with additional resilient mechanisms. Firewalls represent one of the most critical roles in infrastructures, as they control which packets go in and out of the network. The main improvement of the SIEVEQ multi-layered architecture, when compared with previous systems, is the separation of message filtering in several components that carry on verifications progressively more costly and complex. This allows the proposed system to be more efficient than the state-of-the-art replicated firewalls under attack. SIEVEQ also includes several resilience mechanisms that allow the creation, removal, and recovery of components in a dynamic way, to respond to evolving threats against the system adequately. Experimental results show that such resilience mechanisms can significantly reduce the effects of DoS attacks against the system.

7.2 Future Work

The results of this thesis open many avenues for future work on this research topic, which we address in the following topics:

Integration of prevention techniques: In Chapter 2, we have briefly described a few prevention techniques. Although we did not address them in this thesis, here, we initiate the discussion on how to integrate such techniques in LAZARUS. The LAZARUS replicas have two states were they are dormant when they are in quarantine or before they are deployed in the execution plane. Thus, this time could be used to apply a battery of automatic procedures (e.g., vulnerability detectors) that would (1) detect additional weaknesses, which could be reported to NVD, and (2) remediate these weaknesses with automatic mechanisms like automatic patching [93]. Moreover, although limited to open source products, it could use vulnerable code clone detectors, that would indicate potentially shared vulnerabilities [107, 194]. Such mechanisms would decrease the vulnerability surface, especially if common vulnerabilities are detected. Although we are most concern with more complex attacks (e.g., Advanced Persistent Threats (APTs)), using automatic attacks [90] could activate some common vulnerabilities before the replicas deployment. These would trigger the algorithm to adjust the risk associated with such replicas avoiding their selection.

Extending the LAZARUS source types and sensors: LAZARUS monitors five security data feeds on the internet to collect data about vulnerabilities, exploits, and patches for the OSes it manages. However, it could be extended to monitor other indicators of compromise (e.g., Internet Protocol (IP) blacklists) extracted from a much richer set of sources [119, 161]. Similarly, LAZARUS could additionally use the outputs of IDSs to assess the BFT system behavior and trigger replica's reconfigurations in case of need. Finally, and exploring a different domain it could be interesting to mine black market data sources to discover new threats that could compromise several replicas [4]. However, exploring this path would require natural language processing techniques, and thus which is an entirely different area of research.

Virtualization technology: LAZARUS paid a performance penalty due to the limitations of the virtualization platform we used (VirtualBox). VirtualBox was selected because it is the platform where we could run more OSes. Therefore its use enabled LAZARUS to support 17 different OSes for running BFT systems. It would be great to have a VM technology capable of supporting all existing OSes without the resource limitations we experienced. The idea of replacing VirtualBox with an alternative solution would reduce the opportunities available to create diversity. However, it could be interesting to use different VMMs by taking the most of each one. For example, OSes that have CPU limitations on VirtualBox would run in the alternative VMM (e.g., Xen). Although this solution seems promising, as it also creates another level of diversity, by the knowledge we have so far, the OSes that have limitations in VirtualBox have fewer compatibilities with other VMMs like Xen.

Diversity-aware replication: In the same line, the evaluation of BFT-SMART on top of LAZARUS shows that different replica set configurations can impact on the performance of applications, mostly due to the performance heterogeneity of the different OSes. It would be interesting to consider protocols in which this heterogeneity is taken into account. For example, the leader could be allocated in the fastest replica, or weighted-replication protocols such as WHEAT [168] could be used to assign higher weights to the replicas running in faster replicas. These approaches would reduce the impact of running OSes that suffer from resource limitations.

Trusted components: Our prototype implements the LTU as a trusted component isolated from the rest of the replica in a VM, as many works on hybrid BFT [53, 147, 160, 170, 183]. The recent popularization of trusted computing technologies such as Intel SGX [95], and its use for implementing efficient BFT replication [16], open interesting possibilities for using novel hardware to support services like LAZARUS on bare metal. An alternative solution would be to leverage on the existent rack-server firmware (e.g., Dell's iDRAC [50]) to implement LAZARUS in bare metal. Nevertheless, the controller firmware will be the local trusted component. Such solution would also improve LAZARUS performance as it would elimnate the virtualization limitations.

Acronyms

APTs Advanced Persistent Threats. 117

ARFF Attribute-Relation File Format. 70, 72

ASLR Address Space Layout Randomization. 16, 17

BFT Byzantine Fault Tolerance. 1, 2, 4–7, 11–15, 20, 21, 26, 29, 31, 49, 53–56, 58, 62, 65, 67, 73, 74, 76, 80–85, 87, 88, 99, 115–117

BM Bare Metal. 76

COTS Components Off-the-Shelf. 17, 19, 21, 27

CPE Common Platform Enumeration. 31–33, 68

CVC Common Vulnerability Count. 41

CVCst Common Vulnerability Count Strategy. 39, 40, 43, 44

CVE Common Vulnerabilities and Exposures. 24, 26, 31, 32, 64, 68, 71

CVI Common Vulnerability Indicator. 36–39, 42–44

CVIst Common Vulnerability Indicator Strategy. 39, 43, 51

CVSS Common Vulnerability Scoring System. 22, 24–26, 28, 31, 58, 59, 68, 115

DBMS Databases Management Systems. 56

DNS Domain Name System. 1, 2, 35

DoS Denial-of-Service. 5, 28, 35, 64, 84–86, 89–92, 95, 100, 102, 104, 105, 108–110, 113, 116

FPGA Field-Programmable Gate Array. 107

GPU Graphics Processing Unit. 107

HMAC Hash-based Message Authentication Code. 103

ICS Industrial Control Systems. 113

IDS Intrusion Detection System. 5, 19, 83, 107, 117

IP Internet Protocol. 35, 36, 117

IRT Inter-Reporting Times. xxii, 45–47

IRTst Inter-Reporting Times Strategy. 39, 46

JVM Java Virtual Machine. 56, 76, 106

KMP Knuth–Morris–Pratt. 107

KVS Key-Value Storage. 54, 79, 81

LTU Logical Trusted Unit. 54, 55, 73, 74

MAC Message Authentication Code. 12, 85, 91, 93–97, 100, 103

MTD Moving Target Defense. 28, 29

NFS Network File System. 20

NIST National Institute of Standards and Technology. 31

NVD National Vulnerability Database. 2, 3, 5, 18, 22, 24–26, 28, 31–36, 39, 40, 47, 50, 51, 55, 57, 58, 68, 83, 115, 117

OS Operating System. 2–4, 9, 18–23, 26–29, 31–51, 53, 54, 56, 57, 62–64, 67, 69, 73, 76–80, 88, 89, 115–118

OSI Open Systems Interconnection. 83

OSINT Open Source Intelligence. 3, 4, 21, 53, 55, 58, 68, 74, 115

OSVDB Open Sourced Vulnerability Database. 22–25

OTS Off-the-Shelf. 2, 3, 16–21, 29, 30, 32, 35, 56

PO Proactive Obfuscation. 20, 21

POA Potential for Attack. 22

PPZDA Potential Pseudo Zero-Day Attack. 22

PRRW Proactive-Reactive Recovery Wormhole. 15

PZDA Pseudo Zero-Day Attack. 22

RSA Rivest–Shamir–Adleman. 103

SCADA Supervisory Control and Data Acquisition. 6, 74, 83

SCIT Self-Cleaning Intrusion Tolerance. 27

SHA Secure Hash Algorithm. 103

SIEM Security Information and Event Management. 104, 111, 113

SMR State Machine Replication. 11, 12, 14, 73, 74, 103, 108

SQL Structured Query Language. 17, 18, 32, 33

SVM Support Vector Machines. 25

TCP Transmission Control Protocol. 35, 36, 103, 105, 106, 108

TLS Transport Layer Security. 70

TOM Total Ordered Multicast. 88, 89, 95, 98, 100–103

UDP User Datagram Protocol. 105

VM Virtual Machine. 4, 14, 15, 21, 28, 29, 53, 54, 56, 67–69, 73, 76, 78, 79, 88, 117

VMM Virtual Machine Manager. 14, 117

VPN Virtual Private Network. 26

WINE Worldlwide Intelligence Network Environmnet. 23

XML Extensible Markup Language. 31, 32

XSS Cross-site scripting. 57

YCSB Yahoo! Cloud Serving Benchmark. 79, 80

ZDA Zero-Day Attack. 22

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*, Oct. 2005.

[2] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2015.

[3] L. Allodi. Economic Factors of Vulnerability Trade and Exploitation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2017.

[4] L. Allodi and F. Massacci. Comparing Vulnerability Severity and Exploits Using Case-Control Studies. *ACM Transactions on Information and System Security*, 17(1):1–20, Aug. 2014.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine Replication Under Attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, Dec. 2011.

[6] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., 1st edition, 2001.

[7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the ACM European Conference on Computer Systems*, May 2018.

[8] D. Arsenault, A. Sood, and Y. Huang. Secure, Resilient Computing Clusters: Self-Cleansing Intrusion Tolerance with Hardware Enforced Security (SCIT/HES). In *Proceedings of the International Conference on Availability, Reliability and Security*, Apr. 2007.

[9] P. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems*, 32(4):1–45, Jan. 2015.

[10] A. Avizienis. The Methodology of N-version Programming. *Software Fault Tolerance*, 3:23–46, Jan. 1995.

[11] A. Avizienis and L. Chen. On the Implementation of N-Version Programming for Software Fault Tolerance During Execution. In *Proceedings of the Annual International Computer Software and Applications Conference*, Nov. 1977.

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

[13] L. Bairavasundaram, S. Sundararaman, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Tolerating File-system Mistakes with EnvyFS. In *Proceedings of USENIX Annual Technical Conference*, June 2009.

[14] B. Baudry and M. Monperrus. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Computer Surveys*, 48(1):1–26, Sept. 2015.

[15] J. Behl, T. Distler, and R. Kapitza. Consensus-Oriented Parallelization: How to Earn Your First Million. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, Dec. 2015.

[16] J. Behl, T. Distler, and R. Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the ACM European Conference on Computer Systems*, Apr. 2017.

[17] A. Bessani. From Byzantine Fault Tolerance to Intrusion Tolerance (a position paper). In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2011.

[18] 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*, Apr. 2008.

[19] A. Bessani, M. Santos, J. Félix, N. Neves, and M. Correia. On the Efficiency of Durable State Machine Replication. In *Proceedings of the USENIX Annual Technical Conference*, June 2013.

[20] A. Bessani, J. Sousa, and E. Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014.

[21] E. Bhatkar, D. Duvarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the USENIX Security Symposium*, June 2003.

[22] L. Bilge and T. Dumitras. Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2012.

[23] L. Bilge, Y. Han, and M. Dell'Amico. RiskTeller: Predicting the Risk of Cyber Incidents. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2017.

[24] Bill Brenner. Stack Clash Linux vulnerability: you need to patch now. `https://nakedsecurity.sophos.com/2017/06/20/stack-clash-linux-vulnerability-you-need-to-patch-now/`. 2017-6-20.

[25] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking Blind. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2014.

[26] M. Bozorgi, L. Saul, S. Savage, and G. Voelker. Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, Jan. 2010.

[27] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[28] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. Mckelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, M. Mcnett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2011.

[29] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Oct. 1999.

[30] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[31] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, Aug. 2003.

[32] J. Chauhan and R. Roy. Is Firewall and Antivirus Hacker's Best Friend? `http://www.ivizsecurity.com/blog/wp-content/uploads/2011/02/Vulnerability-Trends-in-Security-Products-upto-Year-2010.pdf`, Jan. 2011.

[33] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2015.

[34] L. Chen and A. Avizienis. N-version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Proceedings of the IEEE Symposium on Fault-Tolerant Computing*, 1978.

[35] Cisco. Multiple Vulnerabilities in Cisco Firewall Services Module. `http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20121010-fwsm`. 2012-10-10.

[36] Cisco. Multiple Vulnerabilities in Firewall Services Module. `http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20070214-fwsm`. 2007-2-14.

[37] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright Cluster Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2009.

[38] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the USENIX Symposium on Networked Design and Implementation*, Apr. 2009.

[39] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*, June 2010.

[40] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems*, 31(3):261–264, Aug. 2013.

[41] 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*, Oct. 2004.

[42] CVSS. Common Vulnerability Scoring System. `https://www.first.org/cvss/`. Accessed: 2018-4-26.

[43] CVSSv2. Common Vulnerability Scoring System. `https://www.first.org/cvss/v2/guide`. Accessed: 2018-4-26.

[44] CVSSv3. Common Vulnerability Scoring System. `https://www.first.org/cvss/specification-document`. Accessed: 2018-4-26.

[45] Danny Palmer. Petya ransomware attack: What it is, and why this is happening again. `https://www.zdnet.com/article/petya-ransomware-attack-what-it-is-and-why-this-is-happening-again/`. 2017-6-28.

[46] E. Database. Offensive Security's Exploit Database Archive. `https://www.exploit-db.com/`. Accessed: 2018-4-26.

[47] Debian. Debian Security Tracker. `https://security-tracker.debian.org/tracker/`. Accessed: 2018-4-26.

[48] Defense Advanced Research Projects Agency. The Cyber Grand Challenge. `http://archive.darpa.mil/cybergrandchallenge/`. Accessed: 2018-4-26.

[49] Dell. Sonicwall. `http://www.sonicwall.com/us/en/products/network-security.html`. Accessed: 2018-4-26.

[50] Dell. The Integrated Dell Remote Access Controller. `https://www.dell.com/support/contents/us/en/04/article/product-support/self-support-knowledgebase/enterprise-resource-center/systemsmanagement/idrac`. Accessed: 2018-4-26.

[51] Y. Deswarte, K. Kanoun, and J. Laprie. Diversity Against Accidental and Deliberate Faults. In *Proceedings of the Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions*, July 1998.

[52] F. Dettoni, L. Lung, M. Correia, and A. Luiz. Byzantine Fault-tolerant Sate Mmachine Replication with Twin Virtual Machines. In *Proceedings of the IEEE Symposium on Computers and Communications*, Mar. 2013.

[53] T. Distler, R. Kapitza, I. Popov, H. Reiser, and W. Schröder-Preikschat. SPARE: Replicas on Hold. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2011.

[54] T. Distler, R. Kapitza, and H. Reiser. Efficient State Transfer for Hypervisor-based Proactive Recovery. In *Proceedings of the Workshop on Recent Advances on Intrusiton-tolerant Systems*, Apr. 2008.

[55] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings of the European Dependable Computing Conference on Dependable Computing*, Sept. 1999.

[56] Eclipse. NeoSCADA. `https://www.eclipse.org/eclipsescada/`. Accessed: 2018-4-26.

[57] Ethernodes.org. The Ethereum Nodes Explorer. `https://www.ethernodes.org`. Accessed: 2018-4-26.

[58] Ethstats. Ethereum Stats. `https://ethstats.net/`. Accessed: 2018-4-26.

[59] Facebook. Facebook Bug Bounty Program. `https://www.facebook.com/whitehat`. 2018-9-12.

[60] N. Falliere. Exploring Stuxnet's PLC Infection Process. `http://www.symantec.com/connect/blogs/exploring-stuxnet-s-plc-infection-process`, 2010.

[61] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of Workshop on Hot Topics in Operating Systems*, May 1997.

[62] J. Fraga and D. Powell. A Fault-and Intrusion-tolerant File System. In *Proceedings of the International Conference on Computer Security*, Sept. 1985.

[63] FreeBSD. Freebsd advisories. https://www.freebsd.org/security/advisories.html. Accessed: 2018-4-26.

[64] B. Freeman. A New Defense for Navy Ships: Protection from Cyber Attacks. https://www.onr.navy.mil/en/Media-Center/Press-Releases/2015/RHIMES-Cyber-Attack-Protection.aspx. 2015-9-17.

[65] S. Frei, M. May, U. Fiedler, and B. Plattner. Large-scale Vulnerability Analysis. In *Proceedings of the SIGCOMM Workshop on Large-scale Attack Defense*, Sept. 2006.

[66] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. Modeling the Security Ecosystem - The Dynamics of (In)Security. In *Procedings of the Workshop on the Economics of Information Security and Privacy*, July 2010.

[67] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2018.

[68] 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*, June 2011.

[69] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. Analysis of Operating System Diversity for Intrusion Tolerance. *Software: Practice and Experience*, 44(6):735–770, June 2014.

[70] M. Garcia, A. Bessani, and N. Neves. Towards an Execution Environment for Intrusion-Tolerant Systems. In *Poster session in Proceedings of the ACM European Conference on Computer Systems*, June 2016.

[71] M. Garcia, N. Neves, and A. Bessani. DIVERSYS: DIVErse Rejuvenation SYStem. In *Simpósio Nacional de Informática (INFORUM)*, Sept. 2012.

[72] M. Garcia, N. Neves, and A. Bessani. An Intrusion-Tolerant Firewall Design for Protecting SIEM Systems. In *Proceedings of the Workshop on Systems Resilience*, June 2013.

[73] M. Garcia, N. Neves, and A. Bessani. SieveQ: A Layered BFT Protection System for Critical Services. *IEEE Transactions on Dependable and Secure Computing*, 15(3):511–525, May 2018.

[74] 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):280–294, Oct 2007.

[75] C. Giuffrida, L. Cavallaro, and A. Tanenbaum. Practical Automated Vulnerability Monitoring Using Program State Invariants. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2013.

[76] Google. Google Vulnerability Reward Program Rules. `https://www.google.com/about/appsecurity/reward-program/index.html`. Accessed: 2018-4-26.

[77] Google. Guava. `https://code.google.com/p/guava-libraries/`, Accessed: 2018-4-26.

[78] A. Gorbenko, A. Romanovsky, O. Tarasyuk, and O. Biloborodov. Experience Report: Study of Vulnerabilities of Enterprise Operating Systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, Oct. 2017.

[79] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan. 2012.

[80] J. Han, D. Gao, and R. Deng. On the Effectiveness of Software Diversity: A Systematic Study on Real-World Vulnerabilities. In *Proceedings of the International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, July 2009.

[81] D. Harkins and D. Carrel. The Internet Key Exchange. Requests for Comments (RFC), Nov. 1998.

[82] HashiCorp. Vagrant. `https://www.vagrantup.com/`. Accessed: 2018-4-26.

[83] HashiCorp. Vagrant Cloud. `https://app.vagrantup.com/boxes/search`. Accessed: 2018-4-26.

[84] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2015.

[85] S. Heo, S. Lee, B. Jang, and H. Yoon. Designing and Implementing a Diversity Policy for Intrusion-Tolerant Systems. *IEICE Transactions on Information and Systems*, E100.D(1):118–129, Jan. 2017.

[86] H. Holm. A Large-Scale Study of the Time Required to Compromise a Computer System. *IEEE Transactions on Dependable and Secure Computing*, 11(1):2–15, Jan. 2014.

[87] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided Automated Software Diversity. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, Feb. 2013.

[88] J. Hong and D.Kim. Assessing the Effectiveness of Moving Target Defenses Using Security Models. *IEEE Transactions on Dependable and Secure Computing*, 13(2):163–177, Mar. 2016.

[89] P. Hosek and C. Cadar. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2015.

[90] H. Hu, Z. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[91] Y. Huang and C. Kintala. Software Implemented Fault Tolerance: Technologies and Experience. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, June 1993.

[92] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, June 1995.

[93] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proceedings of the IEEE Symposium on Security and Privacy*, Aug. 2016.

[94] P. Hunt, M. Konar, F. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of USENIX Annual Technical Conference*, June 2010.

[95] Intel. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. `https://software.intel.com/sites/default/files/`

managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf. Accessed: 2018-4-26.

[96] A. Jain. Data Clustering: 50 Years Beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, June 2010.

[97] Y. Jang, S. Lee, and T. Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016.

[98] Jordan Pearson. A Major Bug In Bitcoin Software Could Have Crashed the Currency. https://motherboard.vice.com/amp/en_us/article/qvakp3/a-major-bug-in-bitcoin-software-could-have-crashed-the-currency? 2018-9-19.

[99] A. Jumratjaroenvanit and Y. Teng-amnuay. Probability of Attack Based on System Vulnerability Life Cycle. In *Proceedings on the International Symposium on Electronic Commerce and Security*, Aug. 2008.

[100] Juniper Networks. Juniper Networks Security. http://www.juniper.net/us/en/products-services/security/. Accessed: 2018-4-26.

[101] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. Voelker. Surviving Internet Catastrophes. In *Proceedings of USENIX Annual Technical Conference*, Apr. 2005.

[102] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen. Analysis of Vulnerabilities in Internet Firewalls. *Computers & Security*, 22(3):214–232, Apr. 2003.

[103] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the ACM European Conference on Computer Systems*, Apr. 2012.

[104] A. Keromytis and V. Prevelakis. *Designing Firewalls: A Survey*, chapter 3, pages 33–49. John Wiley & Sons, Inc., June 2006.

[105] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering protocols for securing group communication. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, Jan. 1998.

[106] S. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of USENIX Annual Technical Conference*, July 2017.

[107] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2017.

[108] J. Kirsch, S. Goose, Y. Amir, D. Wei, and P. Skare. Survivable SCADA Via Intrusion-Tolerant Replication. *IEEE Transactions on Smart Grid*, 5(1):60–70, Jan. 2014.

[109] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal Verification of an OS Kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2009.

[110] J. Knight and N. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, Jan. 1986.

[111] D. Knuth, J. Morris, and V. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, Aug. 1977.

[112] H. Koo, Y. Chen, L. Lu, V. Kemerlis, and M. Polychronakis. Compiler-assisted Code Randomization. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2018.

[113] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems*, 27(4):1–39, Jan. 2010.

[114] D. Kreutz, A. Bessani, E. Feitosa, and H. Cunha. Towards Secure and Dependable Authentication and Authorization Infrastructures. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing*, Nov. 2014.

[115] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, July 1978.

[116] P. Larsen, S. Brunthaler, and M. Franz. Automatic Software Diversity. *IEEE Security & Privacy*, 13(2):30–37, Mar 2015.

[117] C. Lee, Y. Lin, and Y. Chen. A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection. *PLoS ONE*, 10(10):1–22, Oct. 2015.

[118] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru. Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, July 2014.

[119] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah. Acing the IOC Game: Toward Automatic Discovery and Analysis of Open-source Cyber Threat Intelligence. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016.

[120] Limor Kessem. WannaCry Ransomware Spreads Across the Globe, Makes Organizations Wanna Cry About Microsoft Vulnerability. `https://securityintelligence.com/wannacry-ransomware-spreads-across-the-globe-makes-organizations-wanna-cry-about-microsoft-vulnerability/`. 2017-5-14.

[121] Linux Counter. Lines of code of the Linux Kernel Versions. `https://www.linuxcounter.net/statistics/kernel`. Accessed: 2018-4-26.

[122] S. Liu, P. Viotti, C. Cachin, V. Quema, and M. Vukolic. XFT: Practical Fault Tolerance Beyond Crashes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2016.

[123] Y. Liu, A. Sarabi, J. Zhang, P. Naghizadeh, M. Karir, M. Bailey, and M. Liu. Cloudy with a Chance of Breach: Forecasting Cyber Security Incidents. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[124] R. Martins, R. Gandhi, P. Narasimhan, S. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo. Experiences with Fault-Injection in a Byzantine Fault-Tolerant Protocol. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, Dec. 2013.

[125] Microsoft. Microsoft Bug Bounty Program. `https://technet.microsoft.com/en-us/library/dn425036.aspx`. Accessed: 2018-4-26.

[126] Microsoft. Microsoft Security Advisories and Bulletins. `https://technet.microsoft.com/en-us/library/security/`. Accessed: 2018-4-26.

[127] D. Miller, Z. Payton, A. Harper, C. Blask, and S. VanDyke. *Security Information and Event Management (SIEM) Implementation*. McGraw-Hill Education, Oct. 2010.

[128] A. Mishra, B. Gupta, and R. Joshi. A Comparative Study of Distributed Denial of Service Attacks, Intrusion Tolerance and Mitigation Techniques. In *Proceedings of the Intelligence and Security Informatics Conference*, Sept. 2011.

[129] Mitre. Common Platform Enumeration. `http://cpe.mitre.org/`. Accessed: 2018-4-26.

[130] Mitre. CVE Terminology. `http://cve.mitre.org/about/terminology.html`. Accessed: 2018-4-26.

[131] H. Moniz, N. Neves, M. Correia, and P. Verissimo. RITAS: Services for Randomized Intrusion Tolerance. *IEEE Transactions on Dependable and Secure Computing*, 8(1):122–136, Dec. 2011.

[132] J. Moscola, J. Lockwood, R. Loui, and M. Pachos. Implementation of a Content-scanning Module for an Internet Firewall. In *Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2003.

[133] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[134] National Institute of Standards and Technology. National Vulnerability Database. `http://nvd.nist.gov/`. Accessed: 2018-4-26.

[135] National Institute of Standards and Technology. Guide for Conducting Risk Assessments. `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf`, Sept. 2012.

[136] K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitraş. Some Vulnerabilities Are Different Than Others. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, Sept. 2014.

[137] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2017.

[138] Netfilter. The netfilter.org project. `http://www.netfilter.org/`. Accessed: 2018-4-26.

[139] Netnod's Lars-Johan Liman. DNS root server FAQ. https://www.netnod.se/dns/dns-root-server-faq. Accessed: 2018-4-26.

[140] A. Newell, D. Obenshain, T. Tantillo, C. Nita-Rotaru, and Y. Amir. Increasing Network Resiliency by Optimally Assigning Diverse Variants to Routing Nodes. *IEEE Transactions on Dependable and Secure Computing*, 12(6):602–614, Nov 2015.

[141] A. Nogueira, M. Garcia, A. Bessani, and N. Neves. On the Challenges of Building a BFT SCADA. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2018.

[142] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding Focus in the Blur of Moving-Target Techniques. *IEEE Security & Privacy*, 12(2):16–26, Mar 2014.

[143] Oracle. Map of cve to advisory/alert. https://www.oracle.com/technetwork/topics/security/public-vuln-to-advisory-mapping-093627.html. Accessed: 2018-4-26.

[144] Oracle Solaris. Oracle Solaris Third Party Bulletin. https://www.oracle.com/technetwork/topics/security/bulletinjul2016-3090568.html. Accessed: 2018-4-26.

[145] Palo Alto. Palo Alto Networks. http://www.paloaltonetworks.com/products/platforms/PA-5000_Series.html. Accessed: 2018-4-26.

[146] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the Association for Computing Machinery*, 27(2):228–234, Apr. 1980.

[147] M. Platania, D. Obenshain, T. Tantillo, R. Sharma, and Y. Amir. Towards a Practical Survivable Intrusion Tolerant Replication System. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, Oct. 2014.

[148] N. Poolsappasit, R. Dewri, and I. Ray. Dynamic Security Risk Management Using Bayesian Attack Graphs. *IEEE Transactions on Dependable and Secure Computing*, 9(1):61–74, Jan. 2012.

[149] K. Prabha and S. Sukumaran. Improved Single Keyword Pattern Matching Algorithm for Intrusion Detection System. *International Journal of Computer Applications*, 90(9):26–30, Mar. 2014.

[150] Pyloris. Pyloris. `https://sourceforge.net/projects/pyloris/`, Accessed: 2018-4-26.

[151] V. Rahli, I. Vukotic, M. Volp, and P. Verissimo. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Proceedings of the European Symposium on Programming*, Apr. 2018.

[152] B. Randell. System Structure for Software Fault Tolerance. In *Proceedings of the International Conference on Reliable Software*, Apr. 1975.

[153] RedHat. RedHat CVE Database. `https://access.redhat.com/security/cve/`. Accessed: 2018-4-26.

[154] H. Reiser and R. Kapitza. VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology. In *Proceedings of the Workshop on Recent Advances on Intrusiton-tolerant Systems*, May 2007.

[155] M. K. Reiter. The rampart toolkit for building high-integrity services. In *Proceedings of the International Workshop on Theory and Practice in Distributed Systems*, Sept. 1995.

[156] Machine Learning Group at the University of Waikato. WEKA. `http://www.cs.waikato.ac.nz/ml/weka/`. Accessed: 2018-4-26.

[157] Oracle. VirtualBox. `https://www.virtualbox.org/`. Accessed: 2018-4-26.

[158] CVE Details. The Ultimate Security Vulnerability Datasource. `http://www.cvedetails.com/`. Accessed: 2018-4-26.

[159] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt. The Design and Implementation of an Intrusion Tolerant System. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2002.

[160] T. Roeder and F. Schneider. Proactive Obfuscation. *ACM Transactions on Computer Systems*, 28(2):1–54, July 2010.

[161] C. Sabottke, O. Suciu, and T. Dumitras. Vulnerability Disclosure in the Age of Social Media: Exploiting Twitter for Predicting Real-World Exploits. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[162] F. Schneider. Implementing Faul-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[163] 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*, June 2010.

[164] M. Shahzad, M. Shafiq, and A. Liu. A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles. In *Proceedings of the International Conference on Software Engineering*, June 2012.

[165] Snort. Network Intrusion Detection & Prevention System. `https://www.snort.org/`. Accessed: 2018-4-26.

[166] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2013.

[167] R. Sommer and V. Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[168] J. Sousa and A. Bessani. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, Sept. 2015.

[169] J. Sousa, A. Bessani, and M. Vukolić. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2018.

[170] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Verissimo. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, Apr. 2010.

[171] P. Sousa, N. Neves, and P. Verissimo. How Resilient are Distributed f Fault/Intrusion-tolerant Systems? In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2005.

[172] P. Sousa, N. Neves, and P. Verissimo. Hidden Problems of Asynchronous Proactive Recovery. In *Proceedings of the Workshop on Hot Topics in System Dependability*, June 2007.

[173] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the USENIX Network and Distributed System Security Symposium*, Feb. 2013.

[174] S. Surisetty and S. Kumar. Is McAfee SecurityCenter/Firewall Software Providing Complete Security for Your Computer? In *Proceedings of the International Conference on Digital Society*, Feb. 2010.

[175] Symantec. Dragonfly: Cyberespionage Attacks Against Energy Suppliers. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/Dragonfly_Threat_Against_Western_Energy_Suppliers.pdf`. 2014-7-7.

[176] Symantec. Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf, 2017.

[177] E. Syta, P. Jovanovic, E. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. Fischer, and B. Ford. Scalable Bias-Resistant Distributed Randomness. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2017.

[178] R. Thorndike. Who Belongs in the Family. *Psychometrika*, 18(4):267–276, 1953.

[179] E. Totel, F. Majorczyk, and L. Mé. COTS Diversity Based Intrusion Detection and Application to Web Servers. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection*, Sept. 2005.

[180] Ubuntu. Ubuntu Security Notices. `https://usn.ubuntu.com/usn/`. Accessed: 2018-4-26.

[181] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2017.

[182] P. Veríssimo, N. Neves, and M. Correia. *Intrusion-Tolerant Architectures: Concepts and Design*, chapter 1. Springer, 2003.

[183] G. Veronese, M. Correia, A. Bessani, L. Lung, and P. Verissimo. Efficient Byzantine Fault-Tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Nov. 2013.

[184] V. Vianello, V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, R. Torres, R. Diaz, and E. Prieto. A Scalable SIEM Correlation Engine and Its Application to the Olympic Games IT Infrastructure. In *Proceedings of the International Conference on Availability, Reliability and Security*, Sept. 2013.

[185] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 2003.

[186] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2018.

[187] F. Wang and R. Upppalli. SITAR: A Scalable Intrusion-tolerant Architecture for Distributed Services - a technology summary. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Apr. 2003.

[188] T. Wilfredo. Software Fault Tolerance: A Tutorial. Technical report, 2000.

[189] D. Williams-King, G. Gobieski, K. Williams-King, J. Blake, X. Yuan, P. Colp, M. Zheng, V. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and Deployable Continuous Code Re-randomization. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[190] 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*, Apr. 2011.

[191] M. Xu and T. Kim. PlatPal: Detecting Malicious Documents with Platform Diversity. In *Proceedings of the USENIX Security Symposium*, 2017.

[192] M. Xu, K. Lu, T. Kim, and W. Lee. Bunshin: Compositing Security Mechanisms through Diversification. In *Proceedings of USENIX Annual Technical Conference*, July 2017.

[193] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2016.

[194] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings*

of the ACM SIGSAC Conference on Computer and Communications Security, Oct. 2017.

[195] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[196] Y. Yeh. Unique Dependability Issues for Commercial Airplane Fly by Wire Systems. In *IFIP World Computer Congress: Building the Information Society*, 2004.

[197] 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*, Oct. 2003.

[198] E. Yuan, N. Esfahani, and S. Malek. A Systematic Survey of Self-Protecting Software Systems. *ACM Transactions on Autonous Adaptative Systems*, 8(4):1–41, Jan. 2014.

[199] E. Zhai, R. Chen, D. Wolinsky, and B. Ford. Heading off Correlated Failures Through Independence-as-a-service. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2014.

[200] M. Zhang, L. Wang, S. Jajodia, A. Singhal, and M. Albanese. Network Diversity: A Security Metric for Evaluating the Resilience of Networks Against Zero-Day Attacks. *IEEE Transactions on Information Forensics and Security*, 11(5):1071–1086, Jan. 2016.

[201] F. Zhao, M. Li, W. Qiang, H. Jin, D. Zou, and Q. Zhang. Proactive Recovery Approach for Intrusion Tolerance with Dynamic Configuration of Physical and Virtual Replicas. *Security and Communication Networks*, 5(10):1169–1180, Mar. 2012.

[202] L. Zhou, F. Schneider, and R. Renesse. COCA: A Secure Distributed Online Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.

[203] R. Zhuang, S. DeLoach, and X. Ou. Towards a Theory of Moving Target Defense. In *Proceedings of the ACM Workshop on Moving Target Defense*, Nov. 2014.