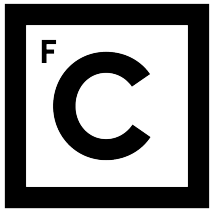
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS



Ciências
ULisboa

# 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

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).

2018

# Acknowledgement

*To my grandmother.*

# Abstract (PT)

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

# Abstract (EN)

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

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Motivation

In the last decade, we have witnessed a significant number of notable cyber attacks from data leakages to ransomware. The root cause of such attacks is two-fold: (1) the malicious intention of their perpetrators and (2) the existence of software vulnerabilities. The later, according to the National Institute of Standards and Technology (NIST), are:

**Definition 1.** *"[...] A weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source."* [93]

The increasing complexity and interconnectivity of software systems is a breeding ground for vulnerabilities, as it is virtually impossible for any company to cover all its software and find its weaknesses. Therefore, most software is shipped to production containing unknown vulnerabilities. Even if, most of the times, the developers insert these vulnerabilities unintentionally.

Typically, vulnerabilities can be discovered by vendors, third-party software analysts, or hackers. When a hacker discovers a vulnerability, often its discloser is made at the same time of the attack while a patch may not be yet available. This type of vulnerability is called a zero-day vulnerability, which can be activated by a zero-day attack. In this case, without a solution to fix the vulnerability, the software users are vulnerable without knowing it before the attack happens. Therefore, a system is vulnerable until a patch for the vulnerability is developed and applied. Moreover, such vulnerabilities are traded on the black market as a highly paid service [123]. 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. For example, big companies and organizations are promoting bug bounty programs to try to find zero-day vulnerabilities before the hackers (e.g., Google Vulnerability Reward Program, [1] Facebook Bug Bounty Program, [2] Microsoft Bounty, [3], DARPA VET, [4] and DARPA Cyber Grand Challenge [5]). Additionally, the traditional detection techniques based on signature or anomaly detection (e.g., anti-virus and Intrusion Detection System (IDS) [115]) may fail to

[1] https://www.google.com/about/appsecurity/reward-program/index.html
[2] https://www.facebook.com/whitehat
[3] https://technet.microsoft.com/en-us/library/dn425036.aspx
[4] https://www.darpa.mil/program/vetting-commodity-it-software-and-firmware
[5] https://www.darpa.mil/program/cyber-grand-challenge

detect new attacks as they do not have the (new) attack signatures yet. Moreover, in some cases these vulnerabilities can still be a problem after their public disclosure [19], as they can be activated even if a patch is already developed by not yet applied.

Although vulnerabilities are everywhere, it does not mean that all systems will eventually be attacked. Therefore, the risk of a system being attacked is a combination of the value of the assets, i.e., the system that one is trying to protect, and the threat level, i.e., the strength of what one is trying to defend against. For example, critical systems, such as Industrial Control Systems (ICS), have a higher risk of being attacked. Typically, threatening these systems results in a higher profit for the attacker due to its economic impact. In the last decade, a few notable attacks targeted ICS like Supervisory Control and Data Acquisition (SCADA) (e.g., Stuxnet attack [40]). Moreover, the continuous integration of field networks with corporate and enterprise networks make ICS more exposed to the plethora of attacks plaguing Internet-based systems [106].

It is a challenge to build secure and dependable systems when highly motivated attackers only need to exploit a few vulnerabilities to compromise the whole system, while the defenders need to cover virtually all the unknown vulnerabilities. Thus, there is a need to design and develop solutions that deal with the abundant presence of vulnerabilities, their unpredictability, and their potential exploitation. In the case of the latter occurs, still provide resilient mechanisms to mitigate the attack's impact on security and dependability.

## 1.2 Background

It is unrealistic to make prior assumptions on the attacker's power rather than assuming that he or she is a powerful attacker. Therefore, researchers and developers have been putting all the efforts on the vulnerability side by trying to reduce the probability of the attacker's success. 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 for critical systems.

### Prevention

One of the primary techniques is to try to avoid that vulnerabilities persist in the code throughout all the software development phases until it is in production. One way to do that is to detect and remove vulnerabilities during the development and testing stages. There are three main categories: static, dynamic, and concolic analysis. A few examples of static analysis are source code verification and validation. However, the most common techniques typically over-simplify the protocols to make them formally verifiable. Moreover, this

process consumes much time, and it is expensive for most of the companies making this method difficult to scale to complex systems [51].

Fuzzing is one dynamic technique and its success in finding vulnerabilities or bugs outcomes from a good set of input test cases. These can be built manually and tuned for a particular application, or randomly generated and then they are likely to fail on triggering more complex vulnerabilities. Another dynamic technique, which is used in information security, is taint analysis. In this technique, any 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 limitation of taint analysis is that the vulnerabilities are detected only for the execution paths that have been explored by the user/tester.

Finally, concolic testing is a technique that performs symbolic execution along with a concrete execution path. Therefore, it is more accurate than fuzzing, but it tends to succumb to path explosion. Some of these techniques can be combined to take the best of some of the approaches (e.g., using fuzzing with symbolic execution [120]).

## Detection and removal

A more complex approach tries to cope with the existence of vulnerabilities that can be exploited. This approach works when an intrusion is detected, and then a recovery mechanism is triggered to clear its effects. Since some attacks use a combination of different vulnerabilities, which isolated would be harmless, it is hard to detect them when the system is already in production. This approach presents three problems: First, there are no perfect intrusion detectors, and then, stealth attacks may not be detected; Second, the service can experience some periods of unavailability while the service is recovering; Last, recovering a system might clean its faults, but the system remains vulnerable. Therefore, it is easy for an attacker to repeat the same procedure to compromise the system every time it recovers.

## Tolerance and masking

The previously described approaches assume that vulnerability prevention is complete to some extent or that is possible to detect all the attacks. First, these assumptions are unrealistic, and it is not advisable to trust the security and dependability of a system in a single component, as it creates a single point-of-failure. Thus, once the system becomes compromised the whole infrastructure becomes open to more attacks. The established way to provide tolerance and masking properties for a system is to distribute it to a set of replicas, which execute the same 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 crashes, the other can replace it and deliver the requests correctly. However, if arbitrary faults are

considered, primary-backup replication is not enough because compromised replicas could deliver arbitrary outputs, impeding a client to decide which output is correct.

State Machine Replication (SMR) [77] is an active approach that has been employed to ensure fault tolerance [112] of fundamental services in modern internet-scale infrastructures (e.g., [22, 32, 64]). 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) execute the same sequence of messages, and (iii) execute the same state transitions. These properties guarantee that a service runs deterministically in the distributed replicas. However, SMR does not provide tolerance for malicious (Byzantine) faults. Therefore, one must resort to intrusion tolerance techniques [127] to address these more complex type of faults. There are different techniques to build a replicated intrusion-tolerant system. More formally, we adopt the following intrusion tolerance definition:

**Definition 2.** *"A replicated 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."* [15]

It is necessary to employ Byzantine fault-tolerant (BFT) SMR, a particular case of SMR, to build a system capable of operating correctly even in the presence of some compromised nodes. Since no single replica can be trusted completely, the correctness of the system comes from the majority of correct nodes. For instance, to tolerate a single arbitrary fault, the system must have four replicas ($n = 4$ and $f = 1$ in the $3f + 1$ fault model) [23]. In Figure 1.1 we present an overview of an BFT protocol. Consider a client *(c)* that issues requests to the replicas *(R1-R4)* which run a Byzantine Agreement among the replicas. In a nutshell, the replicas exchange several messages in a few number of rounds until they agree in the order of the requests to be processed (for more details of the internal steps of the protocol [24]).

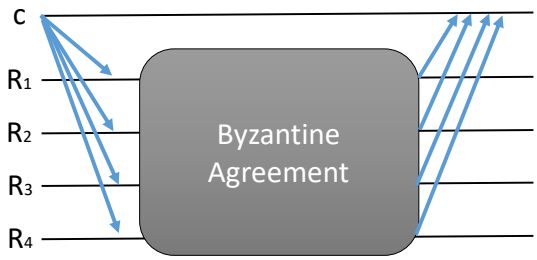



**Figure 1.1** – Byzantine Fault Tolerance protocol overview.

Although BFT protocols provide safety to a bound of $f$ faulty nodes, with sufficient time $T$ an adversary eventually compromises $f + 1$ nodes. Then, additional mechanisms are needed to clean the faulty state (e.g., from time to time the nodes are recovered [24]). However, if a recovered node remains vulnerable to the same attack, the time to compromise $f + 1$ becomes smaller as the attacker already knows how to exploit its vulnerabilities. For this

reason, several authors built their models assuming that the nodes fail independently due to some mechanism that provides failure independence (e.g., [24, 118, 128]).

## 1.3 Problem Statement

In the last twenty years of BFT replication research, few efforts were made to justify or support the assumption that nodes fail independently. On contrary, there were great advances on the performance (e.g., [9, 13, 75]), use of resources (e.g., [14, 83, 128, 135]), and robustness (e.g., [6, 17, 30]) of BFT systems. These works assume, either implicitly or explicitly, that replicas fail independently, relying on some orthogonal mechanism (e.g., [26, 110]) to remove common weaknesses, or rule out the possibility of malicious failures from their system models. A few works have implemented and experimented with such mechanisms [6, 109, 110], but in a very limited way. Nonetheless, in practice, diversity is a fundamental building block of dependable services in avionics [134], military systems [43], and even in recent blockchain platforms such as Ethereum[6] – three essential applications of BFT.

A few works consider the diversity of replicas as a way to achieve such failure independence, however, it is mostly taken for granted. For example, by using memory randomization techniques [110] or different Operating Systems (OSs) [69, 109] without providing evidence for such independence. Moreover, it has been shown that memory randomization does not suffice to impede common failures to occurring [20, 114], and that although diversity promotes fault independence to some extent, it does not avoid utterly different OSes from sharing vulnerabilities [47].

Even considering an initial set of $n$ diverse replicas, supporting the fault independence assumption, long-running services need to be cleaned from possible failures and intrusions. A few works on the proactive recovery of BFT systems [24, 38, 101, 110, 118] periodically restart their replicas to clean undetected faulty states introduced by a stealth attacker. However, a common limitation of these works is that they assume that these weaknesses will be cleaned after the recovery. In practice, this will not happen unless the replica changes its software (e.g., via the previously described techniques) after its recovery.

## 1.4 Identified Problems

We have identified a few long-standing open problems on how BFT systems handle malicious threats. Some of the problems are related to how BFT replicated systems handle different

[6] https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/

types of attacks, and the others (most of them) are related to evaluating, selecting, and managing the diversity of BFT systems:

1. **Traffic dissemination:** Typical BFT protocols use one of the two following approaches to disseminate messages to the replicas: (1) traffic replicator before the replicas or (2) a leader is responsible to dessiminate the messages to the other replicas. The dissemination of a message to all replicas can be detrimental to the proper operation of the replicated service. For example, a traffic replicator device (e.g., hub) can be placed at the entry of the system to transparently reproduce all messages [110, 118]. The effect is an attack amplification caused by the replicator device. Alternatively, a leader replica could receive the traffic and then disseminate the messages to the others [6]. The drawback is that the leader becomes a natural bottleneck, especially when under attack (instead of dispersing the attack load over all replicas).

2. **Finding evidence for supporting diversity:** The current solutions that vouch for diversity as a way to guarantee failure independence, either lack of supporting evidence, or use evidence that is limited to some extent. Therefore, some of the results may provide false conclusions. We identify a need for finding *accurate* sources for supporting sound evidence of the benefits diversity as a dependable mechanism.

3. **Dependably manage diversity:** A few works use automatic and artificial diversity (e.g., [6, 110]). However, they lack evidence to support the failure of independence through diversity. Moreover, some studies show that these techniques fail to provide real diversity [20, 114]. Additionally, the existent systems that implement time-triggered recoveries assume that it takes the same time to compromise each replica. This assumption is unrealistic, especially when the diversity of replicas is considered. Therefore, it is required tailored methods to evaluate the risk of a replicated system becoming compromised.

4. **Supporting diversity mechanism:** The few works have implemented and experimented with such mechanisms [6, 109, 110]. However, despite the lack of evidence for supporting the diversity claim, they lack mechanisms that can make practical BFT systems using diversity in a continuous mode operation (i.e., with recoveries). Thus, reducing the costs typically associated with the management of such complex systems which deemed them as practical.

5. **Untrusted (distributed) controller:** Solutions that implement recovery mechanisms are dependent on controller components that react (i.e., reactively or proactively) to some alarm or variable to trigger a recovery. This type of component has been assumed to be trusted for the most existent works (e.g., [101, 110, 118]). Moreover, they can be centralized or distributed, and in the second case it is assumed that

the controller nodes are synchronous, i.e., they execute as a single node. We have identified a few problems on designing a solution for intrusion-tolerant (distributed) controllers. Although some of these problems result directly from the solutions that we have built to the previous problems, the others are more generic, and then, they can solve the existent controllers' limitations. These problems are related to the decisions that the (distributed) controller must take in a secure way, guaranteeing that a malicious controller does not damage the integrity of the system. For example, the controller nodes need to be proactive in taking some actions without harming SMR properties. Some of these decisions require secure distributed random number generation, or to take "synchronous" decisions (e.g., update a replicated database at the same time). Another problem that must be addressed is the way that data is stored among the controllers, which in the existent solutions is always centralized or distributed and trusted.

## 1.5 Summary of the Contributions

In this thesis, we present the contributions that we propose to solve the previously enumerated problems.

We design a new architecture to solve the *dissemination problem*, and the envisioned solution applies filtering operations in two stages acting like a sieve. We introduced a first pre-filtering stage that performs lightweight checks, making it efficient to detect and discard malicious messages from external adversaries. This effort resulted in a BFT system named 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. SIEVEQ was experimentally evaluated in different scenarios, and the results show that it is much more resilient to Denial-of-Service (DoS) attacks and various kinds of intrusions than existing replicated-firewall approaches. The contributions of this work resulted in the following publications:

1. *An Intrusion-Tolerant Firewall Design for Protecting SIEM Systems, Miguel Garcia, Nuno Neves, Alysson Bessani, in the Workshop on Systems Resilience in conjunction with the IEEE/IFIP International Conference on Dependable Systems and Networks, 2013* [48].

2. 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* [49].

We address the problems of *finding evidence for supporting diversity*, *manage diversity in a dependable way*, and *supporting diversity mechanism* in the thesis main contribution. We use different 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. Moreover, this data is used to create clusters of similar vulnerabilities, which potentially can be affected by (variations of) the same exploit. These clusters and the other collected data are used to assess the risk of the BFT system becoming compromised due to 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 developed these contributions in a solution named LAZARUS, and it is the first system that manages BFT replicated systems (e.g., SIEVEQ) in a dependable and automatic way. LAZARUS experimental evaluation shows that its strategy reduces the number of executions where the system becomes compromised and that our prototype supports the execution of full-fledged BFT systems in diverse configurations with 17 OS versions, reaching a performance close to a homogeneous bare metal setup.

The contributions of this work resulted in the following publications:

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* [45].

4. LAZARUS*: Automatic Management of Diversity in BFT Systems, Miguel Garcia, Alysson Bessani, and Nuno Neves* [].

☞ RE-DO

In the previous contribution, we presented a controller that was logically-centralized. In the following publication we present BATON, a generic BFT-controller for BFT systems that eliminates some of LAZARUS' assumptions:

5. BATON *Miguel Garcia, Alysson Bessani, and Nuno Neves* [].

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 and deploying the most dependable configurations.*

Although these are not directly related to this thesis, we present the following publications:

The results achieved during the MSc thesis encouraged us to extend the work on Off-the-Shelf (OTS) OSes common vulnerabilities. In this extension, 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 are enough to warrant its adoption as a more straightforward, less complicated, more manageable configuration for replicated systems.

6. *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* [47].

The following work describes a SCADA system enhanced with BFT techniques. We document the challenges of building such system from a "traditional" non-BFT solution. This effort resulted in a prototype that integrates the Eclipse NeoSCADA and the BFT-SMaRt open-source projects. This solution could be managed by 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* [97].

Table 1.1 presents how we map the different identified problems into publications, chapters, and solution tools.

| Identified Problems | Publication | Chapter | Solution tool |
|---|---|---|---|
| 1. | (2), (3) | | SIEVEQ |
| 2. 3. 4. | (1), (4), (5) | | LAZARUS |
| 5. | (6) | | BATON |

**Table 1.1** – Overview of the mapping of open problems, published contributions, and chapters.

## 1.6 Document Structure

# 2 Literature Review

## 2.1 Byzantine Fault Tolerance

Intrusion tolerance was first proposed by Fraga and Powell in 1985 [42] as a solution to address faults without compromising the security of a system. Almost 15 years after, BFT replication became the most common solution to implement intrusion tolerance systems. Castro and Liskov's PBFT [23] was the first practical BFT protocol. PBFT implements a SMR protocol that guarantees both liveness and safety for $\lfloor \frac{n-1}{3} \rfloor$ out of a total of n replicas. These properties hold even in asynchronous systems such as the internet. PBFT implements a SMR, therefore, it must guarantees that each replica executes the same commands, in the same order, and then it produces the same output. The protocol can be briefly described as follows: a client sends a message to all the replicas. Then, the leader replica has to assign a sequence number to the request and multicast 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. A commit message is sent by every replica. When a replica receives the commit message from a quorum it executes the message. In the end, it replies to the client. Every replica shares a key with each other and with clients. These keys are used to authenticate messages with a Message Authentication Code (MAC). The messages that are multicast by the clients are authenticated with a vector of MACs. Then, each replica verifies its own MAC. The authors implemented BFS, a Byzantine fault tolerant Filesystem, to validate this BFT library. The results show that when the workload increases the throughput and latency is nearly the same of a non-replicated system. This level of performance encouraged the use of BFT in common systems, and to develop optimizations to improve BFT protocols. Since the PBFT proposal, some work has been dedicated to improve BFT protocols (see Table 2.1):

**Summary.** Intrusion tolerance has an important role in our work, namely BFT-replication. We propose a system that enhances intrusion tolerance by adding mechanisms that work on top of a BFT replicated system. We need BFT protocols to guarantee that replicas execute in the same order and to be able to tolerate some malicious failures in a subset of the replicas. The implementation of such BFT protocol is a complex task, particularly if we need to include mechanisms for state transfer, reconfiguration, and guarantee a good performance. We prefer to rely on existent libraries than to build a new one.

| | |
|---|---|
| Zyzzyva [75] | Introduces 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 the client needs to help replicas to fix the servers' inconsistency. |
| Upright [29] | It provides a straightforward way to also add BFT to crash fault tolerant systems (CFT); |
| Aardvark [30] | Shifts the paradigm to a new design. This design improves the performance under faulty scenarios trading some performance on the normal case; |
| BFT-SMaRt [17] | It is modular and multicore-aware. Supports replica reconfiguration and has a flexible programming interface; |
| COP [13] | It is the most recent BFT implementation that reached 2.4 million operations per second. This was achieved mostly due to BFT architecture changes. |

**Table 2.1** – Brief overview of the most relevant BFT works.

## 2.2 Rejuvenations

Software rejuvenation was proposed in the 90's [62, 63] as a proactive approach to prevent performance degradation and failures due to software aging. The first proposals were based on periodical clean-up of the aging effects by restarting some parts of the software. This mechanism would postpone failures and restore the performance. This solution was implemented using three components: a watchdog process (`watchd`), a checkpointing library (`libft`) and a replication mechanism (`REPL`). A primary node executes the application while a backup node has the application inactive. The primary also runs the `watchd` to monitor the application crashes and hangs. In the backup, there is another watchdog watching the primary node. There is a routine (provided by `libft`) that periodically makes checkpoints and logging. 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. PBFT-PR [24] and COCA [138] introduced proactive recoveries in BFT replicated systems. Proactive Recovery is a mechanism that periodically rejuvenates the replicas to clean potential faults or stealth attacks. This mechanism allows that an adversary controls some replicas, usually $f$ , during a time window, where the system is vulnerable. Castro and Liskov presented PBFT-PR [24], a BFT replication library that does proactive recovery (PR). PBFT-PR assumes that replicas will frequently recover. Additional assumptions are needed to guarantee the PBFT-PR's liveness and safety during the recoveries: (i) Each replica contains a trusted chip to store its private key, and it can sign and decrypt messages without revealing the key; (ii) The replicas' public keys are stored in a BIOS read-only memory, which needs physical access to modify the BIOS; And last, (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. Zhou *et al.* presented COCA [138], a fault-tolerant and secure online certification authority that has been built and deployed both in a local area network and on the internet. COCA uses proactive signature sharing to ensure that when replicas fail and recover there is no key-leakage. Contrary to PBFT-PR it does not rely on trusted components, but it may need an administrator to refresh the COCA server keys. Distler *et al.* identified virtualization as

a useful mechanism to implement proactive recovery in VM-FIT [39, 105]. The authors assume that an intrusion-tolerant replicated system executes in an untrusted domain, running in Virtual Machine (VM). VM-FIT executes in a trusted domain, i.e., the Virtual Machine Manager (VMM). Virtualization allows the isolation between the untrusted and the trusted domains. Therefore, it can trigger recoveries from the trusted domain in a synchronous manner. Moreover, virtualization reduces downtime of the service during the recovery and makes the state transfer between replicas more efficient. The authors implemented this system using the Xen hypervisor to get the two domains: the trusted domain is the Xen Dom0, and the replicas run in the untrusted domains, DomUs.

Sousa *et al.* improved the state-of-the-art recovery algorithms by introducing Proactive-Reactive Recovery Wormhole (PRRW) [118]. This technique removes the effects of faults "immediately". PRRW accelerates 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 recoveries are time triggered [119]. To address this need the authors proposed an hybrid system model: the payload is an any-synchrony subsystem, and the wormhole is a synchronous subsystem. The authors implemented this system using the Xen hypervisor as the wormhole. Zhao *et al.* [137] improved [118] with an algorithm to schedule the rejuvenations that is adaptable to a constant monitoring of the network and CPU/memory performance.

**Summary.** BFT replication with proactive recovery represents one of the cornerstones of intrusion tolerance. PR allows the system to reduce the vulnerability window of the replicated system drastically. BFT replication with PR guarantees the system's correctness while the replicas recover faster than $f + 1$ replicas become faulty. One limitation of PR is that there must be a trusted element somewhere to implement this mechanism. Some works use hardware timers while others resort to virtualization to separate the execution domains. In our proposal, we will adopt an architecture similar to [39] and [118]. However, our solution will implement a different (from [118]) algorithm to proactively trigger recoveries.

## 2.3 Diversity

In some of the works presented before, the correctness is ensured under the assumption that replicas fail independently. Often the authors assume that fault independence is achieved using diversity [24, 118]. Diversity can be implemented in different ways [98**?** ], which may differ in the mean and the amount of diversity generated, but the goal is the same: to create different attack surfaces. The goal is to make the discovery of common vulnerabilities more difficult. We present at least three different ways to generate diversity: (i) N-version programming consists in design/implementation of different program binaries (from the same or different source code); (ii) Randomization implements different memory schemes;

and (iii) Off-the-shelf diversity comprises the use of different products with the same functionality but taking advantage of implementations that are already available.

**N-version programming** N-version programming is a technique used to create diverse software components [10, 26, 73]. The main idea behind this mechanism is to develop N different implementations of the same specification. These implementations can be implemented by N different developers or using translators to N different programming languages.

**Artificial diversity.** Forrest [41] suggested randomized program transformations to introduce application diversity. They have made modifications on the gcc, a C compiler, in such a way that during the compiling time gcc inserts random padding into each stack frame. Linger presented stochastic diversification tool to swap code structures [11, 78]. The diverse versions are generated during the source code compilation. The main idea of these works is to make the intruder's work more difficult when exploiting buffer overflow vulnerabilities. Bhatkar *et al.* [18] proposed a solution based on memory address obfuscation (i.e., Address Space Layout Randomization (ASLR)). Their solution transforms object files and executables at the link-time and load-time, without kernel or compiler modifications. The goal is to ensure that an attack that succeeds in 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, a recent work proved that solutions like ASLR are vulnerable to attacks [20]. A few works propose the usage of compilers that generate different executables [101, 110, 133].

**Off-the-shelf diversity.** The two previous solutions generate diversity before the software's distribution. OTS diversity 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 different free products that provide the same functionality and were developed by different vendors. In other words, Components Off-the-Shelf (COTS) diversity is like an opportunistic N-version programming. Totel *et al.* proposed an IDS based on design diversity [125]. 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. Then the proxy votes the responses and replies to the clients. The authors developed an algorithm to detect intrusions and tested their solutions against Snort [113] and WebStat [130]. The results show that using diverse-COTS service (e.g., http) allows the IDS to deliver fewer alarms without missing the intrusions. Gashi *et al.* made an experimental evaluation of the benefits of adopting diversity of SQL database servers [50]. The authors analyzed bug reports of four database servers (Post-greSQL, Interbase, Oracle, and Microsoft SQL Server) and verified which products were affected by each bug reported.

They have found few cases of a single bug affecting more than one server. In fact, there were no coincident failures in more than two of the servers. The conclusion is that OTS database servers' diversity is an effective mean to improve system's reliability. However, the authors recognize the need for SQL translators, to increase the interoperability between servers in the replicated scenario. Han *et al.* . made a systematic analysis of the effectiveness of using OTS diversity to improve system's security [55]. First, the authors found if there were OTS 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, more than 6k 1 vulnerabilities were analyzed, which were published in the National Vulnerability Database (NVD) on the year 2007. The results showed that 98.5% of the vulnerable software have substitutes. Moreover, the majority of them either did not had 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 different filesystems. Even between versions from the same OS, the low-level functions change across the versions. The study also concluded that 22.5% of the vulnerabilities were present in multiple software. However, only 7.1% from those vulnerabilities were present in software that offers the same service. The study findings are a good sign that diversity can improve a system's dependability. In a previous work, we studied OTS OSes diversity [47]. Similar to [55] this study was carried out taking vulnerability feeds from NVD. However, our work was only focused on OSes, and the data collected comprised 11-year of vulnerability reports. Our goal was to find to what extent different OSes shared common vulnerabilities. In this study, we analyzed 2270 vulnerabilities entries manually, where they were classified in different categories. Then, we defined three types of servers: (i) a server that contains most the packages/applications available (Fat server); (ii) a server that does not contain unnecessary applications to a certain service (Thin server); and (iii) a thin server but with physically controlled access (Isolated hin server). We assumed that the third setting it is the most advisable for critical systems, since additional care is taken to install and setup its configuration. For each configuration, we compared all the common vulnerabilities in pairs of OSes. As expected, in the Isolated thin server, the number of common vulnerabilities was considerable less than in the other configurations. There was only 1 vulnerability that was shared among six OSes, 2 that were shared among five OSes, and 130 that are shared between two OSes. We went further in the study and looked for common vulnerabilities in different versions of the same OS. We have found evidence that suggests that using OS diversity in a replicated system can improve its dependability. Even if few OSes are employed, it is possible to achieve vulnerability independence just with different versions.

Larsen *et al.* made a study with several examples on how to apply diversity [78, 79]. The paper explains in detail what to diversify, from single instructions to an entire program. They also explain the three types of security impacts when diversity is applied: entropy analysis, attack-specific code analysis, and logical argument. However, they also point that there is no consensus on how to evaluate the efficacy of diversity.

**Summary.** We presented three different techniques that support diversity as a fault-tolerant mechanism. N-version programming is the most costly, since it needs different developers or additional programs to verify if the automatic translation works. On the contrary, OTS and randomization diversity are almost for free. The first one can be obtained from free sources that are ready to be used, and the second one is automatic and already supported in most OSes. Diversity still has some gaps that must be addressed to make it an effective building block of intrusion tolerance. For example, how to measure diversity in such a way that failure independence can be guaranteed?

## 2.4 Intrusion-tolerant Firewalls

☞ **NOTE**: Improve– make it more embeded

Over the past years, there has been an important amount of research in the development of systems that are intrusion tolerant. However, only very little work was devoted to design intrusion-tolerant firewall-like devices. Performance reasons might explain this, as BFT replication protocols are usually associated with reasonable overheads and limited scalability.

Sousa et al. [118] proposed the first replicated intrusion-tolerant system that implements proactive-reactive recovery. The paper presented a firewall for critical services, named CRUTIAL Information Switch (CIS), which was integrated with a trusted component and works under the assumption of an hybrid synchrony model.

Roeder and Schneider [110] proposed a replicated intrusion-tolerant device that introduces diversity through software obfuscation techniques on each rejuvenation. The authors main focus was in supporting diversity, a technique that could be integrated into our work.

There are several approaches to defend systems from DoS/Distributed Denial-of-Service (DDoS) (see a survey in [136]). For example, Walfish et al., [131] proposed a solution based on active response to DDoS. Basically, upon the detection of the attack, the server requests the client to send more data. The idea is that by increasing the load, the network congestion management mechanisms will make the channels be used in a more fair way among the correct and incorrect clients. Unfortunately, this solution is not resource efficient because it overloads the channels for the benefit of the client. Jia *et al.* , proposed a solution based on traffic redirection. It works on cloud-based services to solve DDoS. Upon the detection of a DDoS the system redirects the correct client to non-attacked servers [67]. If the network support is available, this kind of technique could be used together with SIEVEQ to ensure *sender-SQs* will always be able to reach a correct *pre-SQ*.

**Summary** SIEVEQ shares a few similarities with CIS, but there are two fundamental differences. The CIS needs traffic replicators, one for in-bound and other for out-bound messages, and therefore, an external DoS will be replicated to all replicas; another difference is that the CIS is a stateless firewall.

In the last decade, several important 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.

## 2.5 Security Metrics

Jumratjaroenvanit and Teng-amnuay [68] described vulnerability life-cycles presenting their different stages. The authors goal was to understand how different dates can be useful to estimate the probability of an attack. The authors methodology was to collect and analyze data on the discovery, disclosure, and exploit dates, exploit and patch availability, all from public data sources. They used this information to define five life cycle types: (i) Zero-Day Attack (ZDA), which typically is a done by a black-hat and is defined by the date of disclosure and exploit being the same; (ii) Pseudo Zero-Day Attack (PZDA), 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), it is a zero-day vulnerability. The most influential variables were the code that was scrutinized by a the tester with access to the source code, and the code that was analyzed with some static tool before. On the contrary, results showed that language safety was the less influent factor. The main result of the study was that two weeks of work were enough to have a 50% chance of finding a zero-day vulnerability. Sometimes, 53 hours were enough to find one vulnerability with more than 95% of probability.

Okhravi *et al.* presented TALENT a framework for live migration of critical infrastructures across heterogeneous platforms [99, 100]. The idea was to create a moving target that difficult the success of advanced targeted attacks. TALENT implements OS level virtualization with containers, which allows 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 filesystem. The filesystem synchronization took 98.7% of the migration time. The authors decided to focus on optimizing the filesystem synchronization. In the optimized version, the filesystem synchronizes in periodic intervals by sending the differences to the destination. Therefore, the migration was made seamless to the application. TALENT also had a risk assessment, called operation assessment, which monitored and adjusted the diverse components using vulnerability information. However, there was almost no details on how the risk was measured and on how to adapt to the threats in an efficient manner.

Guo and Bhattacharya addressed the problem of BFT replicas fault independence [54]. The authors proposed different replicas configurations using OTS OSes to increase the diversity among the replicas. There was a set of different configurations that could build a replicated system. Each configuration was composed of different OS. The authors proposed a formalization of the virtual replica diversification problem. They used a game-theory approach to find an optimal diversification strategy for the defender's side. The authors validated their model with the NVD data used in [46]. Wang *et al.* proposed a different approach to measure the risk [132]. The study explored how many zero-day vulnerabilities were required to compromise a network. They assumed that zero-day vulnerabilities typically do not occur at the same time, however the solution did not depend on any statistical model. The reasoning behind the proposal was novel, e.g., on the method to find the complexity of multiple zero-day vulnerabilities in a N-node network.

Newell *et al.* [95] presented a solution to assign diversity variants among network nodes to increase the network's resiliency.

Sabote *et al.* [111] presented a study that used non-common vulnerability data sources, i.e., Twitter data (e.g., specific words, the number of retweets, and replies, information about the users posting these messages), to find exploit data. The authors made a quantitative and qualitative exploration of the vulnerability-related information disseminated on Twitter. They developed a Twitter-based exploit detector 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 strength their information results they also used sources like NVD, Open Sourced Vulnerability Database (OSVDB), Exploit-DB, Microsoft Security Advisories and Symantec WINE. The authors distinguished the real-world exploits from the proof-of-concept exploits, as the latter are by products of the disclosure process. The real-world exploits typically are not known until a critical zero-day attack occurs. The authors used Support Vector Machines (SVM) to classify exploits in the social media and tested how robust it was to attacks, i.e., if a powerful attack could manipulate the information on Twitter with a false account and false data. The results showed that with their proposal they could be confident with the predictions. For example, the SVM classifier set to a precision of 25% could detect the Heartbleed exploits within 10 minutes of its first appearance on

Twitter. They also showed that organizations like NVD overestimate the severity of some vulnerabilities that never become in fact exploited. However, their work did not focus on the discovery of the zero-day vulnerabilities, which is relevant for our research.

**Summary.** We presented several works that carry out for risk assessment in a way to prevent exploits or to take action upon vulnerability detection. This is the last building block that intrusion tolerance is missing. There is a lot of relevant free information 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 weakness. 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 data.

## 2.6 Systems Management

☞ **NOTE:** Change title

In the last two decades there were a number of BFT protocols and systems deemed "practical" (e.g., [9, 13, 14, 24, 75, 83, 128, 135]). Most of these systems either ignore the issue of fault independence or simply assume it is solved in some way (e.g., N-version programming [26] or OTS diversity [47, 50]). In principle, LAZARUS can support the execution of all these systems/protocols, as long as they support, or are extended to support, replica group reconfigurations, just like BFT-SMaRt [17]. In this section, we discuss the few previous works that address the issue of diversity in BFT systems.

BASE [109] is an extension of PBFT [23] that explores opportunistic OTS diversity in BFT replication. The system provides an abstraction layer for running diverse service codebases on top of the PBFT library. The key issue addressed by BASE is 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, from 16 years ago, show the same trends we observed: performance varies significantly when diversity is considered. Differently from LAZARUS, BASE does not address the selection of replicas or the reconfiguration of a replica group.

In order to support long-running services, Castro and Liskov [24] introduced the notion of proactive recovery for BFT services. The objective is to rejuvenate replicas periodically to remove stealth attackers and support the execution of long-running services. All works on *safe* proactive recovery consider the use of trusted local component on each replica

to trigger the periodic recoveries [24, 38, 101, 110, 118]. A common weakness of most of these works is that they do not support diversity, therefore a compromised replica can be attacked immediately after its recovery by exploiting the same vulnerability. The two noticeable exceptions are discussed in the following.

Roeder and Schneider [110] propose an intrusion tolerance technique called Proactive Obfuscation (PO) for supporting a different type of diversity from LAZARUS. The idea is, instead of changing OSes or other off-the-shelf component of the replica, PO changes the application and library code periodically preserving the original semantics using 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 generates its own obfuscated executable from a read-only device containing the "master code" based on time-triggered epochs that a controller triggers in a secure way through a trusted component similar to LAZARUS' Logical Trusted Unit (LTU). All obfuscation mechanisms were implemented and evaluated on OpenBSD 4.0, and their results show that PO adds a little extra overhead to the non-PO execution.

Similarly to PO, Platania *et al.* [101] proposed a compiler-based diversity for the Prime BFT protocol [6] using the MultiCompiler tool [60]. This compiler creates diverse binaries from the same source code through randomization and padding techniques. The authors also proposed a theoretical rejuvenation model that receives 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 suggest (but does not show how) that $c$ can 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 executes 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, in an architecture similar to ours.

**Summary.** Diversity is becoming one of the building blocks of intrusion tolerance, alongside with BFT and rejuvenations. We presented few works that already used somehow diversity in intrusion-tolerant systems. Some of the works employed opportunistic OTS components to create diversity among the replicas. We do not discard the possibility of using other diversity techniques, such as randomization. However, in our work we are interested in OTS diversity because it is possible to estimate the vulnerability of each component. These works are still limited to that extent, i.e., there are none or few concerns on how to create diversity to avoid common failures. Most of the works implement diversity assuming complete fault independence, but that is unrealistic. 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 the

failure independence sound. Moreover, further work is still necessary to create automatic mechanisms that abstract diversity management for the administrators.

Both PO and the work from Platania *et al.* improve the diversity of applications and replication libraries,[1] but not on OSes and other OTS components used by replicas. Therefore, LAZARUS complements these services, as well as any proactive recovery system,[2] by assessing and monitoring the potential risk of common-mode failures in a replicated service, and selecting appropriate replacements as the threat landscape evolves.

LAZARUS exploits the opportunistic OTS diversity derived from the existence of many version for OSes, Databases (DBs), Web Servers, Java Virtual Machine (JVM), etc. Several studies show that different versions of these components significantly increase chances of avoiding common-mode failures [47, 50, 53]. Although we focus only on OSes in our LAZARUS prototype, the same techniques can be used to evaluate and deploy full diverse software stacks in the replicas.

## 2.7 Moving Target Defenses

A few works on *Moving Target Defense* have goals similar to ours. However, most of these works do not focus on replicated system nor use real data to generate different configurations. Hong and Kim [61] is an exception, as it proposes a formal security model for Shuffle, Diversity, and Redundancy techniques. Although this work considers OS diversity, a few unrealistic assumptions were made, e.g., it assumes that any OS requires the same amount of time to compromise and that there are no common vulnerabilities among different OSes. Moreover, their model and experiments were made with only two OSes and a handful of vulnerabilities for each of them.

[1] It should be noted, however, that recent studies have been shown that the benefits of these techniques are limited [20, 114].
[2] In particular, the ones that support proactive and reactive recoveries [118].

# 3 Overview and Preliminaries

In this chapter, we present an overview of the whole thesis components, we present the system model, and some definitions.

## 3.1 Overview

In the end of this chapter, it should be clear how the different contributions are connected and how we materialize them in software systems.

In a nutshell, as displayed in Figure 3.1, LAZARUS provides a distributed operating system for BFT-replicated services. The system manages in its execution plane a set of nodes that can run *unmodified replicas* (encapsulated in VMs or containers). Each node must have a small 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 existing vulnerabilities in the pool of replicas, aiming to minimize the risk of having several nodes being compromised by the same attack.

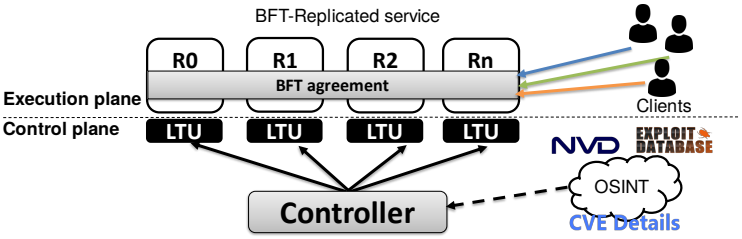



**Figure 3.1** – LAZARUS overview.

## 3.2 System Model

LAZARUS system model shares some similarity with previous works on the proactive recovery of BFT systems (e.g., [24, 101, 110, 118]). More specifically, we consider a *hybrid system model* composed of two planes with different properties and assumptions:

- **Execution Plane:** This plane is componsed 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, 17, 23, 75]. This plane hosts $n$ replicas from which at most $f$ can be compromised at any given moment. In this paper, we consider the typical scenario in which $n = 3f + 1$ [9, 24, 75].

- **Control Plane:** For simplicity, we will address this component as a logical-centralized controller, which requires stronger assumptions. However, in Chapter **??** we introduce BATON which requires weaker assumptions like the Execution 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., [101, 110, 118]). 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.

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 can not be subverted and controlled by the adversary. In practice, this assumption lead us 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., [84, 111]), which we consider out of scope for this paper.

## 3.3 Execution Plane

The Execution Plane can accomodate any replicated system that already levareges on the existence of a controller node (e.g., [49, 101, 110, 118]) or any BFT system that would benefit from the LAZARUS assistence (e.g., [117]). However, a few requriements must be fulfilled, the Execution Plane is supported by virtulization therefore the BFT must run in a VM. Additionally, the BFT library must provide replicas configuration in order to allow replicas' replacement. In this thesis we evaluate LAZARUS with three instances of differente Execution Planes: BFT ordering for Hyperledger Fabric, a Key-Value Storage (KVS), and SIEVEQ. The latter, is one of the contributions of the thesis (see Chapter 4 and Chapter **??**, therefore detail its description.

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.

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 into 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.

Different fault tolerance mechanisms are employed at the two stages. Pre-filtering is implemented by a dynamic group of nodes named *pre-SQs*. *Pre-SQs* can be the target of various kinds of attacks and eventually may be intruded because they face the external network. Therefore, we take the conservative approach of assuming that *pre-SQs* can fail in an arbitrary (or Byzantine) way, meaning that they may crash or start to act maliciously. When a failed *pre-SQ* is detected, it is simply replaced by a new one that is clean from errors. Since SIEVEQ needs to support different message loads, e.g., due to additional senders, *pre-SQs* can be created dynamically to amplify the aggregated processing capabilities (within the constraints of the hardware). The filtering stage is performed by a group of *replica-SQ* components, which execute as a replicated state machine [112]. *Replica-SQs* may also fail in an arbitrary way, and therefore, we employ an intrusion-tolerant replication protocol that ensures correct operation in the presence of Byzantine faults.

Our solution was guided by the following design 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.

## 3.4 Control Plane

LAZARUS is the first control plane 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. Once the risk increases, LAZARUS replaces the potentially vulnerable replica by another one, trying to maximize the failure independence. Then, 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, removing the human from the loop.

The current implementation of LAZARUS 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. We conducted two sets of experiments, one demonstrates that LAZARUS risk management can prevent a group of replicas from sharing vulnerabilities over time; the other, 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 actually perform close to our homogeneous bare metal setup.

### 3.4.1 BFT-Control Plane

☞ **NOTE**: add details later

## 3.5 Diversity of Replicas

☞ **NOTE**: citar survey [12]

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., 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*.

It is possible to improve the *replicas* fault independence by resorting to different OTS components in the software stack [36]. For example, it has been shown that using distinct OSes [47], filesystems [11, 109], and databases [50], can yield important benefits in terms of fault independence. In addition, automatic techniques could enhance diversity, like randomization/obfuscation of OSes [110] and applications [133].

Although LAZARUS can exploit automatic techniques, in this paper we center our attention on diverse OTS components. In particular, LAZARUS monitors the disclosed vulnerabilities of all elements of the software stacks of the replicas to assess which of them may contain common vulnerabilities.

However, in the experimental evaluation, 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 two last factors are particularly important to enrich the validity of our analysis.

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 [10]; (2) there have been some works showing that such protocol implementations can be generated from formally verified specifications [59, 104]; (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 [17]) make them relatively simple to test and assess with some confidence [81, 85]; and (4) there are no reported vulnerabilities about these system 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 [101, 110].

# 4 Traffic Dissemination

## 4.1 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 [71] 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 [1], Palo Alto [2], and Dell [3]. 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 [40] or Dragonfly [122] 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 [96] 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 [25], IDS [7] or firewalls [27, 28, 70, 121]). Second, firewalls are typically a single point of failure, which means that when they crash, 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

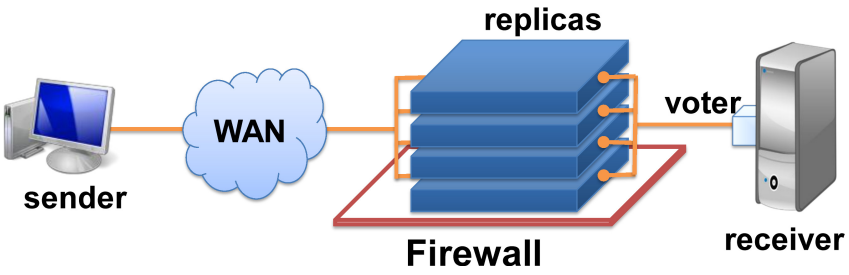



**Figure 4.1** – Architecture of a state-of-the-art replicated firewall.

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.

## 4.2 Intrusion-Tolerant Firewalls

In the last decade, several important 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 4.1 shows an implementation of an intrusion-tolerant firewall, illustrating existing works in this area [110, 118]. 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 [110, 118] 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. This result on attack amplification caused by the replicator device. Alternatively, a leader replica could receive the traffic and then disseminate the messages to the others [110]. The drawback is that the leader becomes a natural bottleneck, especially when under attack (instead of dispersing the attack load over all replicas [6]).

2. The support for stateful firewall filtering requires that all correct replicas process messages in the same order [112]. As a consequence, to ensure an agreement in a common sequence of messages, replicas need to continuously run a BFT consensus protocol [24] 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 [118]. 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) [110]. 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 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 4.1 and launched a DoS attack on this system (see Section 4.5 for a description of the setup and environment). The results show that the system performance is significantly affected by the attack (see Figure 4.2).

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

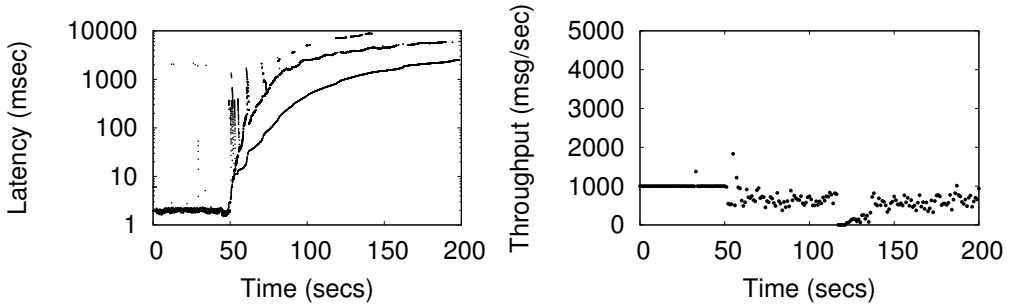



**Figure 4.2** – Effect of a DoS attack (initiated at second 50) on the latency and throughput of the intrusion-tolerant firewall architecture displayed in Figure 4.1.

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.

### 4.2.1 SIEVEQ Architecture

A fundamental difference of the SIEVEQ architecture, when compared with other replicated firewall designs (see Figure 4.1), 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 4.3 presents the architecture of SIEVEQ. In this architecture, message processing starts with a first filtering layer that implements a message authentication mechanism and is 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 4.3.3) 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

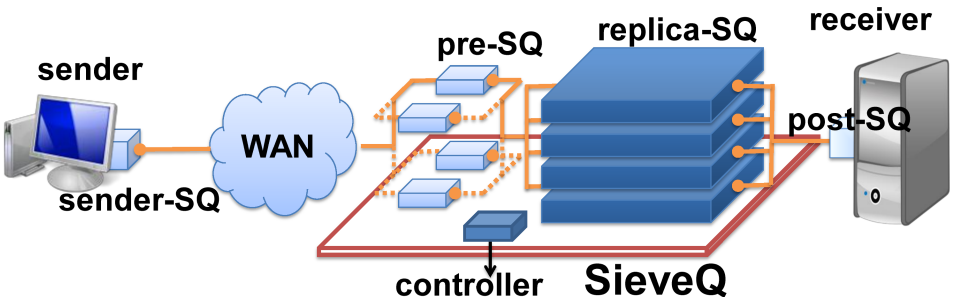



**Figure 4.3** – SIEVEQ layered architecture.

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 Figure 4.3:

☞ **NOTE:** TOM deve ser explicado na intro

- *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 [17]. 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 [101, 110] or distributed [118] way. In Chapter **??**, we present a sophisticated implementation of this controller component and in the Chapter **??** we present a intrusion-tolerant *controller* version.

### 4.2.2 System and Threat Model

The system is composed of a (potentially) large number of external nodes, called *senders*, some internal nodes, called *receivers*, and LAZARUS nodes. Senders run *sender-SQ* modules to be able to transmit packets through the LAZARUS, 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 **??**. 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., [56]). If required, the key distribution infrastructure could also be made intrusion-tolerant [76, 138].

### 4.2.3 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 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 4.2.2).

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 either 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 [89].

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 the *controller* 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. Nevertheless, in Chapter **??** we present a intrusion-tolerant *controller*.

## 4.3 SIEVEQ Protocol

This section details the SIEVEQ protocol and the 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 in our design mitigate such threats.

### 4.3.1 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 in Section 4.3.3.

### 4.3.2 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 (4.1), 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 $mac_{ski}$ is computed using a shared key established with the *pre-SQ*. This MAC serves as an optimization to speed up checks [29].

$$\texttt{Msg} = \langle S_i, sn, \texttt{DATA}, sgn_{Si} \rangle_{mac_{ski}} \tag{4.1}$$

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 acknowledgement message, it re-sends the Msg to another *pre-SQ*.

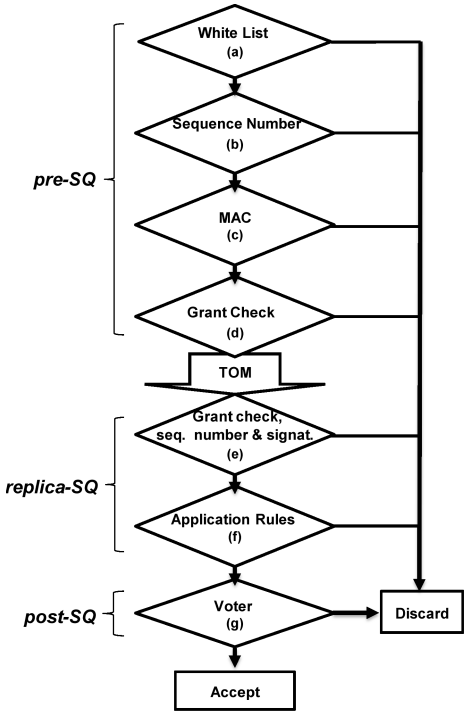



**Figure 4.4** – Filtering stages at the SIEVEQ.

### *Pre-SQ* filtering

The *pre-SQ* determines if an arriving message `Msg` should be forwarded or discarded (stages (a)–(d) in Figure 4.4). 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 finding out that the expected $S_i$ was 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 on 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, it can happen that *replica-SQs* 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 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.

[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.

Valid messages are encapsulated in a new format (see (4.2), 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*.

$$\texttt{Msg}' = \langle S_i, sn, \texttt{DATA}, RS_j \rangle_{mac_{skj}} \tag{4.2}$$

### *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 1 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 1:** *post-SQ* protocol

```
1  Init : executed only once
2  snExpect_Si, k ← 0;
3  WaitingQuorom_Si,sn ←⊥;
4  Function Filter (Msg')
5      if ( verifyMAC(Msg') = FALSE ) then
6          return errorMAC;
7      if (snExpect_Si ≤ Msg'.sn < (snExpect_Si + snThreshold)) then
8          WaitingQuorom_Si,sn ← WaitingQuorom_Si,sn ∪ {Msg'};
9      k ← snExpect_Si;
10     while ( equalMsg(WaitingQuorom_Si,k) ≥ f_rs + 1 ) do
11         deliver(mostVotedMsg(WaitingQuorom_Si,k)) ;
12         snExpect_Si ← snExpect_Si + 1 ;
13         k ← snExpect_Si;
```

### Correctness Argument

In the following, we show that SIEVEQ design satisfies the three functional properties described in Section 4.3.1. The proofs work under the assumptions of our system and threat model (Section 4.2.2).

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

[3]Note that *replica-SQs* send Msg′ (defined in (4.2)) instead of Msg (defined in (4.1)). Both are similar, but Msg' has a MAC instead of a signature to authenticate its contents with *post-SQ* (see Section 4.3.2). For the sake of simplicity we will only use the Msg notation in the rest of the proof.

**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 $R_j$. □

### 4.3.3 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 4.3.1.

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 4.3, 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. Other components of our architecture, such as the *controller* and *post-SQ* are not considered in this section as they are either assumed to be trusted or co-located with the receiver. However, in Chapter **??** we introduce a intrusion-tolerant *controller* version.

In the following, we try to focus on complex scenarios in 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* are 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.

### 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 4.4).

A more demanding scenario occurs when a *sender-SQ* transmits faster than the allowed rate (verification (d) of Figure 4.4). 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 4.4) 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).

### 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 exact 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 are 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 4.4), 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 eventual 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*.

### 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 4.2. 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 [17], 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 address this in Chapter **??**) 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.

## 4.4 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 Transmission Control Protocol (TCP) sockets, to simplify the integration and minimize changes in the 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 [4].

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 [17], to implement the TOM and manage the *replica-SQs*. BFT-SMaRt, like most SMR systems (e.g., PBFT [24], Zyzzyva [75], Prime [6]) 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.

## 4.5 Evaluation

This section focuses on 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 mechanisms defined in Section 4.3.3 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 Security Information and Event Management (SIEM) system under a similar workload as the one observed in the 2012 Summer Olympic Games.

### 4.5.1 Testbed Setup

Figure 4.5 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).

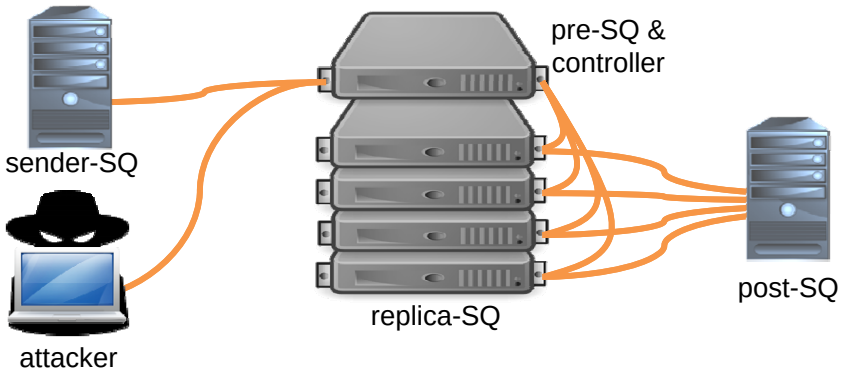



**Figure 4.5** – The SIEVEQ testbed architecture used in the experiments.

### 4.5.2 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 [52] 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 in 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 [103], 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 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).

### 4.5.3 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 4.6 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 important overheads were caused by the tasks associated with securing the message payload (which in fact are the most costly operations in the system). Several optimizations were made to mitigate the performance penalties during the message serialization (e.g., creation of the signature), including the use of parallelization to take advantage of the multicore architecture (as done, for example, in [72]). Table 4.1 shows the gain of using the optimized version of the *sender-SQ* library. Similar optimizations were employed in other components.

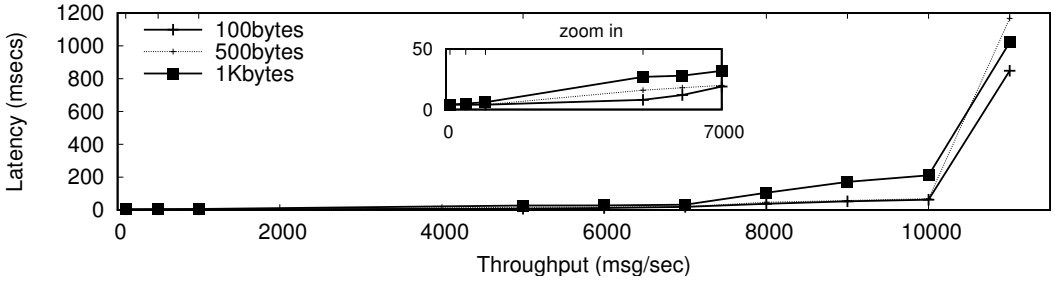



**Figure 4.6** – SIEVEQ latency for each workload (message size and transmission rate).

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

**Table 4.1** – Maximum load induced by the *sender-SQ* library with various message sizes

### 4.5.4 Effect of Filtering Rules Complexity

The results presented in the previous section do not consider any kind of 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 costly 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 FPGAs and GPUs to improve performance in firewalls and IDS (e.g., [80, 92]).

We used a classical algorithm for string matching (Knuth–Morris–Pratt (KMP) [74]) at the *replica-SQs* to implement the filtering rules. This algorithm is employed in IDSs [102] and firewalls like iptables [94]. 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 4.7 shows the latency of SIEVEQ without message filtering (Baseline) and string matching (KMP) with 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

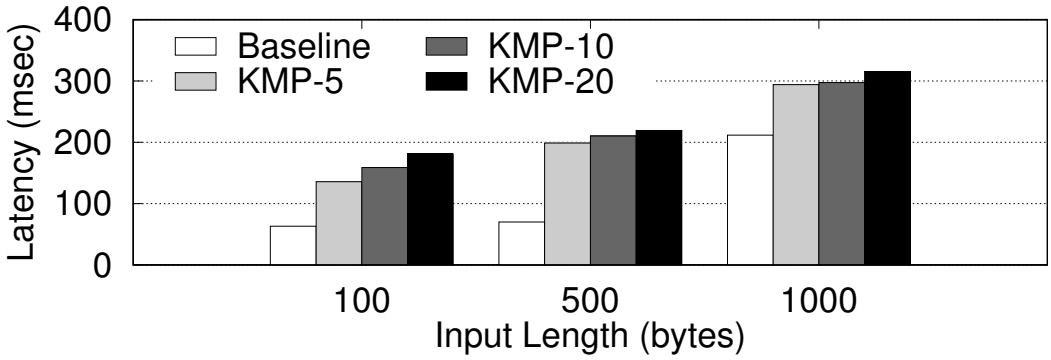



**Figure 4.7** – Comparison of the SIEVEQ's latency between the baseline and adding the filtering rules.

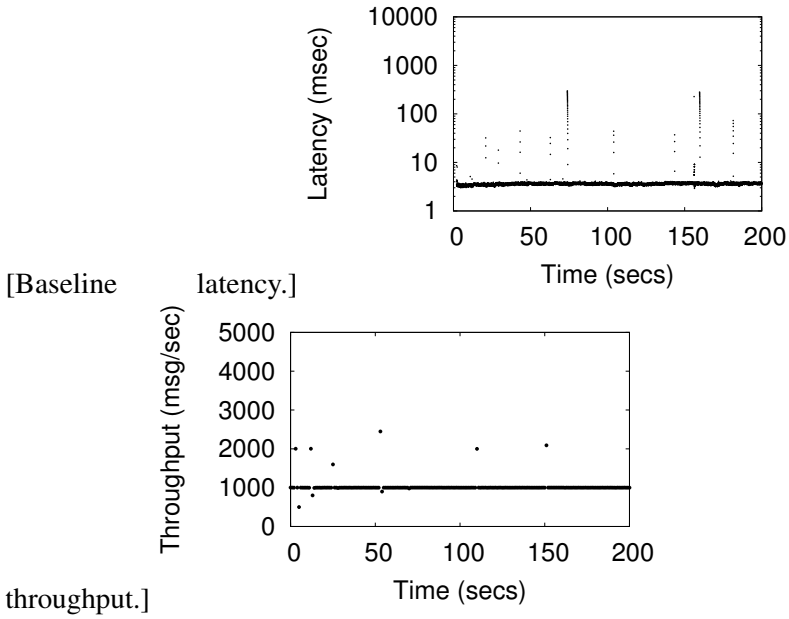
overheads were observed, e.g., with 500 bytes messages and 20 bytes pattern the overhead is approximately $200\%$. This result is expected as the string matching is a slow operation and it needs to be performed in the critical path of message processing. Implementations with hardware support (as mentioned before) could be integrated with SIEVEQ to reduce these delays significantly.

### 4.5.5 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. The results are reported in Figure 4.11. Before presenting the results, we need to stress that these experiments must be compared with the results displayed in Figure 4.2, which were obtained in the same way but with a different architecture (see Figure 4.1).

Figure **??** and Figure **??** 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 **??** and Figure **??**. 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 in terms of 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



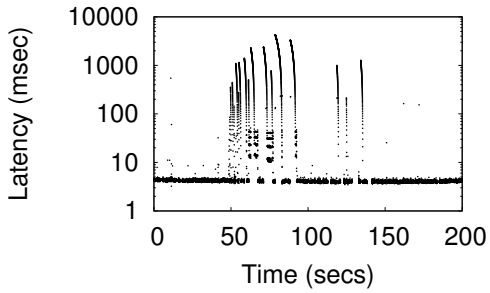

[Baseline latency.] [Baseline throughput.]

**Figure 4.8** – Performance of SIEVEQ under different attack conditions.

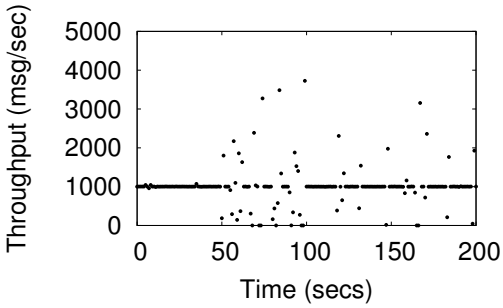
messages taking more than 3 seconds. The attack also has consequences on the throughput as it is possible to observe an oscillation between 0 to 4000 messages per second (which correspond to the situation when the *post-SQ* processes a batch of messages that have been accumulated).

Figure **??** and Figure **??** 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 limited effect because during an interval of time (while there is a search for a fresh target) the system can make progress.

Figure **??** and Figure **??** 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* with *state transfer* requests, which are the most demanding request a replica can receive in BFT-SMaRt [16]. Figure **??** 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




[DoS w/ no recovery.]




[DoS w/ no recovery.]

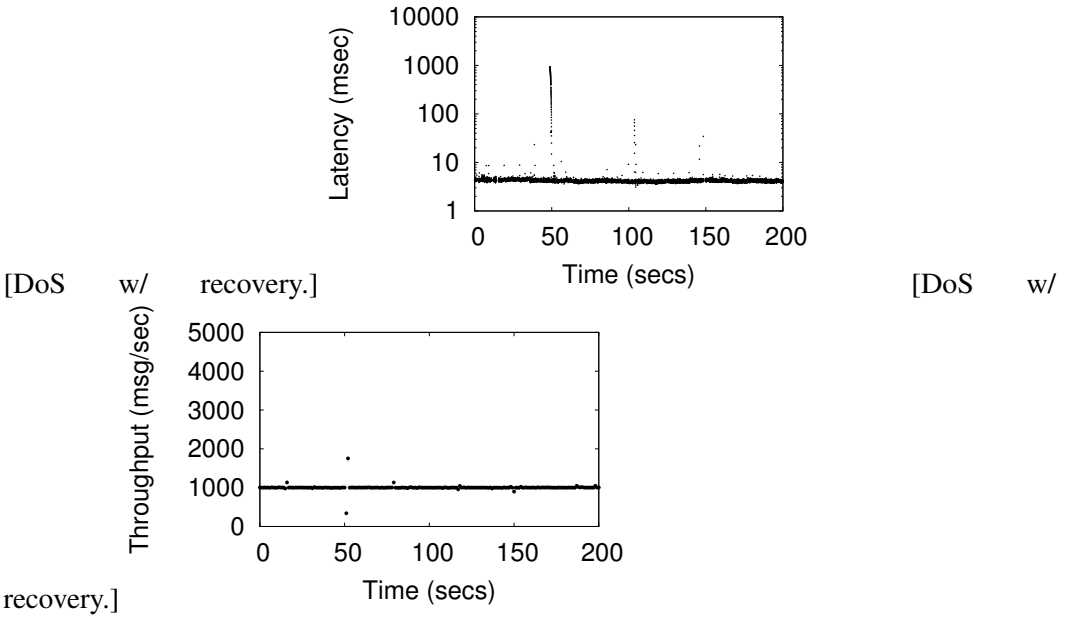
**Figure 4.9** – Performance of SIEVEQ under different attack conditions.

performance of the system is severely disrupted during the attack. Figure **??** shows the same attack but with the detection and recovery mechanism described in Section 4.3.3. The *post-SQ* detects the problem by noticing that $f_{rs} + 1$ *replica-SQs* are sending less 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 **??** 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.

### 4.5.6 Use Case: SIEVEQ to Protect a SIEM System

SIEM systems offer various capabilities for the collection and analysis of security events and information in networked infrastructures [88]. These systems are being employed by organizations as a way to help with the monitoring and analysis of their infrastructures. They integrate a large 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.

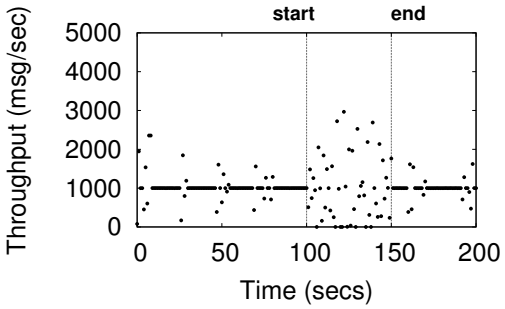



[DoS w/ recovery.] [DoS w/ recovery.]

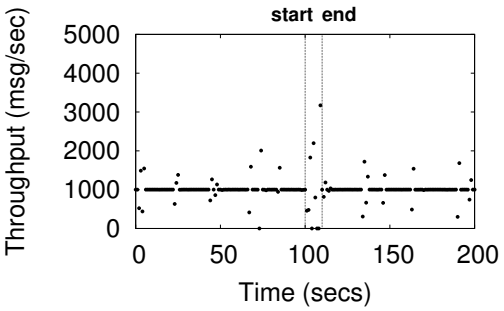
**Figure 4.10** – Performance of SIEVEQ under different attack conditions.

As part of the MASSIF European project [129], we have implemented a resilient SIEM system, where SIEVEQ was used to protect the access to the core facility (in Figure 4.12). 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 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 4.13 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.




[Internal DoS execution without recovery actions.]




[Internal DoS execution with recovery actions.]

**Figure 4.11** – Performance of SIEVEQ under different attack conditions.

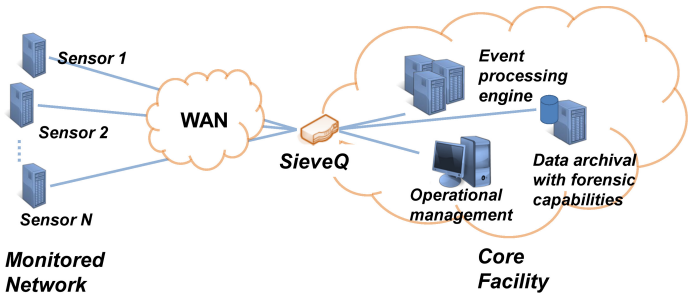



**Figure 4.12** – Overview of a SIEM architecture, showing some of the core facility subsystems protected by the SIEVEQ.

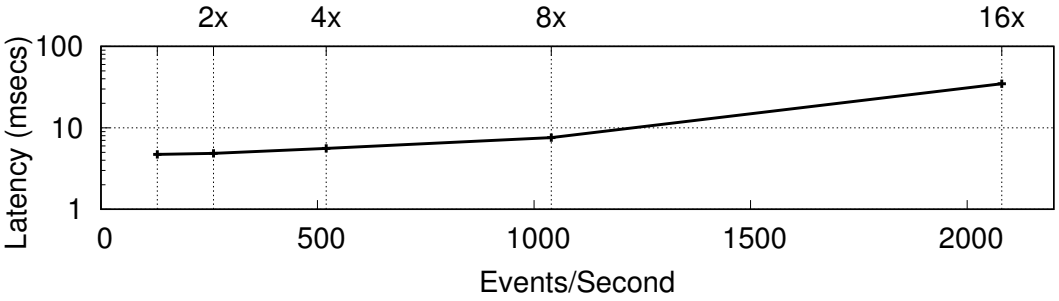



**Figure 4.13** – SIEVEQ latency for the 2012 Summer Olympic Games scenario throughput requirement (127 events per second) and how the SIEVEQ scale, we doubled on each experiment the number of events.

## 4.6 Discussion

SIEVEQ differs from standard firewalls like iptables [94] 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 normal 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 [112]. 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 [24] and implemented in BFT-SMaRt [17]. 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.

## 4.7 Final Remarks

We presented SIEVEQ, a new intrusion-tolerant protection system for critical services, such as 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.

# 5 Finding evidence for supporting and manage diversity

☞ **NOTE**: We need to clarify that LAZARUS is proactive and SIEVEQ is reactive, they can be used together. E.g., when there is a problem the controller increases the risk

## 5.1 Introduction

Practical BFT replication was initially proposed as a solution to handle Byzantine faults of both accidental and malicious nature [23]. 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. For instance, to tolerate a single replica failure, the system typically must have four replicas [9, 24, 75]. This model only works if nodes fail independently, otherwise, once an attacker discovers a vulnerability in one node, it is most likely that the remaining nodes suffer from the same weakness.

In the last twenty years of BFT replication research, few efforts were made to justify or support this assumption. However, there were great advances on the performance (e.g., [9, 13, 75]), use of resources (e.g., [14, 83, 128, 135]), and robustness (e.g., [6, 17, 30]) of BFT systems. These works assume, either implicitly or explicitly, that replicas fail independently, relying on some orthogonal mechanism (e.g., [26, 110]) to remove common weaknesses, or rule out the possibility of malicious failures from their system models. A few works have implemented and experimented such mechanisms [6, 109, 110], but in a very limited way. Nonetheless, in practice, diversity is a fundamental building block of dependable services in avionics [134], military systems [43], and even in recent blockchain platforms such as Ethereum[1] – three essential applications of BFT.

For the few works that do consider the diversity of replicas, the absence of common-mode failures is mostly taken for granted. For example, by using memory randomization techniques [110] or different OSes [69, 109], it is assumed that such failures will not exist without providing evidence for it. In fact, researchers have argued that randomization techniques do not suffice to create fault independence [20, 114]. In addition, although the

[1] https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/

use of distinct OSes promotes fault independence to some extent, *per se* it is not enough to preclude vulnerability sharing among diverse OSes [47].

Even if there was an initial diverse set of $n$ replicas that would have fault independence, long-running services eventually will need to be cleaned from possible failures and intrusions. Proactive recovery of BFT systems [24, 38, 101, 110, 118] periodically restarts the replicas to remove undetected faulty states introduced by a stealth attacker. However, a common limitation is that these works assume that the weaknesses will be eliminated after the recovery. In practice, this does not happen unless the replica code changes after its recovery.

This paper presents LAZARUS, the first system 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. Once the risk increases, LAZARUS replaces the potentially vulnerable replica by another one, trying to maximize the failure independence. Then, 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, removing the human from the loop.

The current implementation of LAZARUS 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. We conducted two sets of experiments, one demonstrates that LAZARUS risk management can prevent a group of replicas from sharing vulnerabilities over time; the other, 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 actually perform close to our homogeneous bare metal setup.

In summary, we make the following contributions:

1. LAZARUS, a control plane that monitors OSINT data and manages the BFT service replicas, selecting and reconfiguring the system to always run the "most diverse" set of replicas at any given time (Sections **??** and 6.1);

2. A method for assessing the risk of a group of replicas being compromised based on the security news feeds available on the internet. The method overcomes limitations from works that use NVD data for managing the replicas vulnerability independence (Section 5.2);

3. An evaluation of our risk management method based on real historical vulnerability data showing its effectiveness in keeping a group of replicas safe from common vulnerabilities (Section 5.4);

4. An extensive evaluation of LAZARUS prototype using 17 OS versions, a BFT replication library, and some BFT applications (i.e., a KVS, an application-level firewall/message queuing service, and a blockchain service) showing the costs of supporting diversity in BFT systems (Section 6.2).

## 5.2 Diversity-aware Reconfigurations

The core of LAZARUS is the vulnerability evaluation method used to assess the risk of having replicas with shared vulnerabilities. This section details this method.

### 5.2.1 Finding Common Vulnerabilities

NIST's NVD [96] is the authoritative data source for disclosure of vulnerabilities and associated information [86]. 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 and a short description provided by the Common Vulnerabilities and Exposures (CVE) [91]. The Common Platform Enumeration (CPE) [90] provides the list of products affected by the vulnerability and the date of the vulnerability publication. The Common Vulnerability Scoring System (CVSS) [33] 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).

Previous studies on diversity solely count the number of shared vulnerabilities among different OSes, assuming that less common vulnerabilities implies a smaller probability of compromising $f + 1$ OSes [47]. Although this intuition may seem acceptable, in practice it underestimates the number of shared vulnerabilities due to imprecisions in the data sources. For example, Table 5.1 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 web sites, and we found out that CVE-2016-4428, for example, also affects Solaris.[2]

[2] https://www.oracle.com/technetwork/topics/security/bulletinjul2016-3090568.html

| 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 5.1** – Similar vulnerabilities affecting different OSes.

Even with these imperfections, NVD is still the best data source for vulnerabilities. Therefore, we exploit its curated data feeds for obtaining 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 [66]. 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 6.1.2 for details), even if the data feed reports that they affect different products. For example, the vulnerabilities in Table 5.1 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.

### 5.2.2 Measuring risk

As discussed before, each vulnerability in NVD has an associated CVSS severity score. Therefore, a straw man solution for measuring risk would be to sum the CVSS scores of all common vulnerabilities in the software stack of two replicas to get an estimate of how dangerous are their shared weaknesses. However, CVSS has some limitations that make it unsuitable for managing the risk of replicated systems: (1) In practice, it has been shown that there is no correlation between the CVSS exploitability score and the existence of real exploits for the vulnerability [21]; (2) CVSS does not provide information about vulnerabilities exploiting and patching times; (3) CVSS does not account for the vulnerability age, which means that severity remains the same over the years [44]; and (4) some studies show that CVSS may overestimate severity [111], as for example larger scores do not correspond to higher prices in the vulnerabilities' black markets [5].

Given these limitations, we derive a novel, more refined, metric to measure the risk of a BFT system being affected by common vulnerabilities. In our particular context, we are mostly

interested in capturing information that relates to the window of exposure that vulnerabilities have, mainly when they are correlated among *replicas*. Therefore, we developed a risk metric that aims to overpass the identified limitations. We solved (1) and (2) by using additional OSINT sources that provide information about the exploit and patch dates. Since NVD does not provide this information, we collect more data from other OSINT sources like Exploit-DB [34] for exploits, patching information from CVE-details [37], and additional vendor websites, such as Ubuntu Security Notices [126], Debian Security Tracker [35], and Microsoft Security Advisories and Bulletins [87] (which also give additional product versions affected by the vulnerability). We solve (3) using the vulnerability published date to calculate its age. Finally, we only use the CVSS attributes that concern to integrity and availability, the properties traditionally related with BFT replication (4).

$$risk(sc) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} common(rc_i, rc_j) \tag{5.1}$$

$$common(rc_i, rc_j) = \sum_{v_k \in \mathcal{V}_{i,j}} score(v_k) \tag{5.2}$$

$$score(v_k) = (A + I + exp(v_k)) \times tdist(v_k) \tag{5.3}$$

$$exp(v_k) = \begin{cases} max(DP - DE, 0) + 1 & v_k \text{ exploited, patched} \\ DE & v_k \text{ exploited, not patched} \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

Our metric considers all this information to measure the risk of a set of $n$ replicas having active shared vulnerabilities. More specifically, it works as an indicator of how fault-independent is a *System configuration*. Equation 5.1 shows the risk of a *System configuration* as the sum of the different *replica* pairs' score. This score is calculated based on the set of vulnerabilities $\mathcal{V}_{i,j}$ that affects both $r_i$ and $r_j$ or that are present in a cluster containing vulnerabilities affecting both *replicas* (Equation 5.2). Finally, we calculate the score of each vulnerability in $\mathcal{V}_{i,j}$ (Equation 5.3). We assign a *dynamic score* to each vulnerability, considering the referred attributes: *(i)* we take two CVSS attributes to capture the extent to which a vulnerability $v_k$ affects availability ($A$) and integrity ($I$); *(ii)* we account for the number of days the vulnerability was exposed with $exp$, i.e., there was an exploit and no patch available. This is calculated considering the number of days to patch ($DP$) and to exploit ($DE$) $v_k$ (Equation 5.4); *(iii)* and we use an amortization function to reflect the fact that older vulnerabilities have are less likely to harm the system ($tdist(v_k) \in [0, 1]$).

### 5.2.3 Selecting Configurations

We use the risk metric to choose the *replicas* that should be included in the *System configuration*. This is done by periodically evaluating the risk of the current *System configuration*. If the risk exceeds a pre-defined threshold, a mechanism is triggered to replace replicas and reduce the overall risk. First, it decides which *replica* (`r`) should be removed and put in a quarantine set (`QUARANTINE`). Then, it selects (one of) the best candidate(s) replicas from all the available candidates (`POOL`) to make the substitution. When the replacement takes place, the resulting *System configuration* (`CONFIG`) has lower risk than the previous one. Additionally, we ensure that removed *replicas* can sometime later re-enter the system, and the ones that are in the system, despite their overall score, are eventually replaced. Therefore, each replica `r` in `CONFIG` has an *age* value that is incremented. On the contrary, each removed replica `r` in `QUARANTINE`, has a *healing* value that is decremented.

This procedure is detailed in Algorithm 2. The *Monitor* function is called on each monitoring round (e.g., on every hour). Consider a `CONFIG` that is already running with risk$= \alpha$. First, the algorithm increments the *age* of each `r` in `CONFIG` (lines 6-7). Then, it verifies if the risk of `CONFIG` (Equation 5.1) does not exceed the predefined $threshold$ (line 8). In the affirmative case, a *replica* replacement is started. First, some local variables are initialized (lines 9-10). Second, it randomly gets pairs $\langle i, j \rangle$ from `CONFIG` (line 11) and saves some of them that augment the risk in `MAXIMALS` (Equation 5.2) (lines 12-14). Third, the algorithm picks the older replica (i.e., the one that is in `CONFIG` for more time) of all selected pairs (lines 15-18). This replica is removed from `CONFIG` (line 19) and added to `QUARANTINE` (line 21). The *healing* value is initialized with a value, different for each *replica*, based on historical data about the time it takes for a patch to be published for this software (line 20). The algorithm calls a function that selects a new replica to join `CONFIG` (line 22). Finally, it decrements the *healing* of each `r` in `QUARANTINE` (line 24). When such value reaches zero, `r` is removed from `QUARANTINE` and added to `POOL` (lines 25-28).

Function *Find_new_config()* (line 22) solves the following optimization problem:

$$\begin{array}{ll} \underline{\text{min}} & \textit{risk}(\texttt{CONFIG} \cup \{r\}) \\ \underline{\text{subject to}} & r \in \texttt{POOL} \end{array}$$

where `CONFIG` is the set of $n - 1$ replicas that will stay in the system and $r$ is the new replica (which we have to find) among the ones in `POOL`. The twist in our case is that we avoid deterministic solutions to increase the difficulty of an adversary guessing the next configurations. Therefore, we developed a simple heuristic that finds the $k$ best replicas in `POOL` (e.g., $k = 3$) and randomly picks one of them to be added to `CONFIG`. The heuristic is quite simple: we just calculate the risk of a configuration with each candidate replica from `POOL`, and choose one of the $k$ replicas that induce lower risk.

**Algorithm 2:** Replica Set Reconfiguration

```
1  CONFIG: set replicas in the System configuration;
2  POOL: set with the available replicas (not in use);
3  MAXIMALS: set of candidate replicas to remove;
4  QUARANTINE: set of quarantine replicas;
5  Function Monitor ()
6      foreach r in CONFIG do
7          increment(r.age);
8      if risk(CONFIG) > threshold then
9          maxScore, maxAge ← 0;
10         toRemove ← ⊥;
11         foreach ⟨i, j⟩ in CONFIG do
12             if common(i,j) ≥ maxScore then
13                 MAXIMALS ← MAXIMALS ∪ {⟨i, j⟩};
14                 maxScore ← common(i,j);
15         foreach ⟨i, j⟩ in MAXIMALS do
16             if older(i,j) ≥ maxAge then
17                 toRemove ← older(i,j);
18                 maxAge ← toRemove.age;
19         CONFIG ← CONFIG \ { toRemove };
20         toRemove.healing ← init_healing(toRemove);
21         QUARANTINE ← QUARANTINE ∪ { toRemove };
22         CONFIG ← Find_new_config(POOL, CONFIG);
23     foreach r in QUARANTINE do
24         decrement(r.healing);
25         if r.healing = 0 then
26             QUARANTINE ← QUARANTINE \ {r};
27             r.age ← 0;
28             POOL ← POOL ∪ {r};
```

Although better heuristics can be developed, this brute-force method works in LAZARUS because we do not expect POOL (our solution space) to be large. In addition, this function is only called if the risk exceeds the threshold.

## 5.3 LAZARUS Implementation

This section details the implementation of each component of LAZARUS. It also briefly presents other aspects of our prototype.

### 5.3.1 Control Plane

Figure 6.1 shows LAZARUS control plane with its four main modules, described below.

❶ ***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

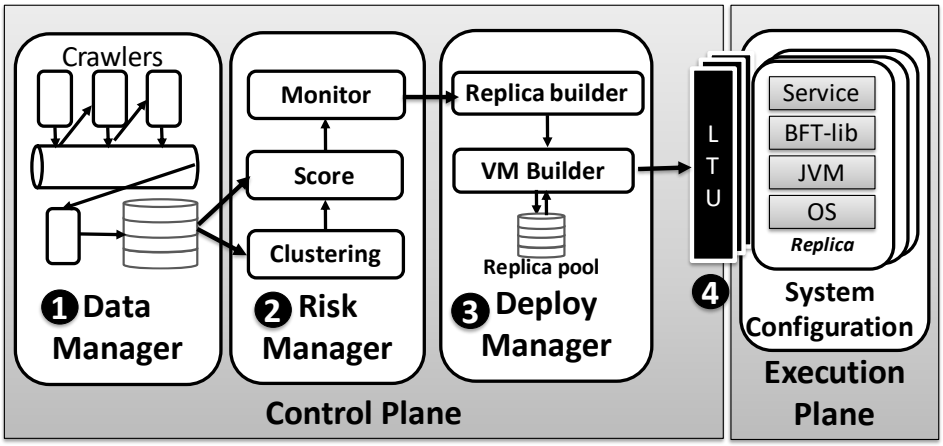



**Figure 5.1** – LAZARUS architecture.

following the CPE Dictionary [90], 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 assemblying 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 five other sources, namely Exploit DB, CVE-details, Ubuntu, Debian, and Microsoft. 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, some CVSS attributes (e.g., availability and integrity); and 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 5.2.2, 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 Section 6.1.2 for details). The *Risk manager* also runs Algorithm 2 to monitor the replicated system and trigger reconfigurations.

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 [57]. 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).

**Setup.** The box configuration is defined in a configuration file, named *Vagrantfile*, consider the example in Listing 6.1. 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 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 6.1 (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 [58]. 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 5.1** – Windows Server 2016 Vagrantfile

✍ **TO-DO**: Add all fields that are mentioned, then add lines to the text

**Download.** One of the fields of the *Vagranfile* 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 official.

**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 *Vagranfile* `provision` field, one can describe the provision steps inline or in an external file and link it to the

*Vagranfile*. 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. Although different, all scripts were meant to do the same thing: First the script installs the software that is missing in the box – this is not true for all OSes – 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.

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.

### 5.3.2 Additional Details

**Vulnerability Clustering.** 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 (up to 200 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 K-means algorithm is applied to build the clusters [66], where the number of clusters to be formed is determined by the elbow method [124]. We used the open-source machine learning library Weka [107] to build the clusters.

### 5.3.3 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 just represent the same or similar vulnerabilities.

In order to apply a clustering technique to our data, we need to follow some steps:

**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 [107], the data must be represented as Attribute-Relation File Format (ARFF). Basically, this is a CSV file with some meta information, see Listing 6.2.

```
1 @RELATION vulnerabilities
2 @ATTRIBUTE cve string
3 @ATTRIBUTE description string
4 @ATTRIBUTE published_date date "yyyy-MM-dd"
5 @DATA
6 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'
7 ... more
```

**Listing 5.2** – 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.
- **stopwords**, we define a stop word list, then words from this list are discarded. We begin with a general 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 way that similar vulnerabilities, even if they affect different products, are put together in the same cluster. For example, recall the vulnerabilities in Table 5.1, 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. We have done this by iterating the process and checking the most used words (top 200 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 [3] 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. And 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. The *K-means* has two important parameters: the number of clusters to be

[3] Stop word list is available: here

formed, we set to 200 clusters. We used the elbow method [124] to decide $k = 200$; and the *distance function*, to calculate the distance between objects, the *Euclidean Distance* is the most adequate for this type of data; 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).

Sometimes the resulting clusters include vulnerabilities that are unrelated. Therefore, we use the Jaccard index (J-index) to measure the similarity between the vulnerabilities within the cluster. We calculate the J-index of each element, and then the average J-index of the cluster. The clusters with a smaller average J-index (below a certain threshold) are considered ill-formed. In this case, we select the vulnerabilities with lower J-index and move them to another cluster that would result in a better J-index. If no such cluster exists, then we create a new one with the "orphan" vulnerability.

**BFT replication.** Although there has 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 [17], a stable BFT library that provides reconfigurations on the *replicas* set.

**Replica Virtualization.** VMs can be used to implement replicated systems, leveraging on the isolation between the untrusted and the trusted domains [38, 101, 118]. 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 [57] 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 [108] because it offers more diversity opportunities, i.e., it supports a more extensive set of different guests OSes.

## 5.4 Evaluation of Replica Set Risk

This section evaluates how LAZARUS performs on the selection of dependable *replica* configurations. As discussed in Section **??**, we focus our experimental evaluation solely on the OS diversity.

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 *2010-1-1* and *2017-9-29*, which is used to setup the *Risk manager*'s algorithm; and (ii) an *execution phase* composed of the period between *2017-10-1* and *2018-3-30*. This last period is divided into three intervals of two months (OUT-NOV, DEC-JAN, and FEB-MAR), allowing for three independent tests. 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 released on that day. We take the most pessimist approach, which is to say that we consider the system to be broken if a vulnerability comes out that affects at least two OSes that would be executing at that time.

Three additional strategies, inspired by previous works, were defined to be compared with LAZARUS (Section 5.2.3):

- **Equal:** all the replicas use the same randomly-selected OS during the whole execution. This strategy corresponds to the scenario where most past BFT systems have been implemented and evaluated (e.g., [6, 9, 13, 14, 17, 30, 75, 83, 128, 135]). Here, compromising a replica would mean an opportunity to intrude the remaining ones.

- **Static:** a configuration of $n$ different OSes is randomly selected, and there are no changes during the whole execution. This corresponds to a diverse BFT system without reconfigurations (e.g., [109]).

- **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*.

The experiments consider a pool of 38 OS versions to be deployed on four replicas. At the beginning of the execution phase, the OSes are assumed to be fully patched.

### 5.4.1 Diversity vs Vulnerabilities

We evaluate how each strategy can prevent the replicated system from being compromised. Each strategy is analyzed over 5000 runs throughout the execution phase in two-month slots. 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.

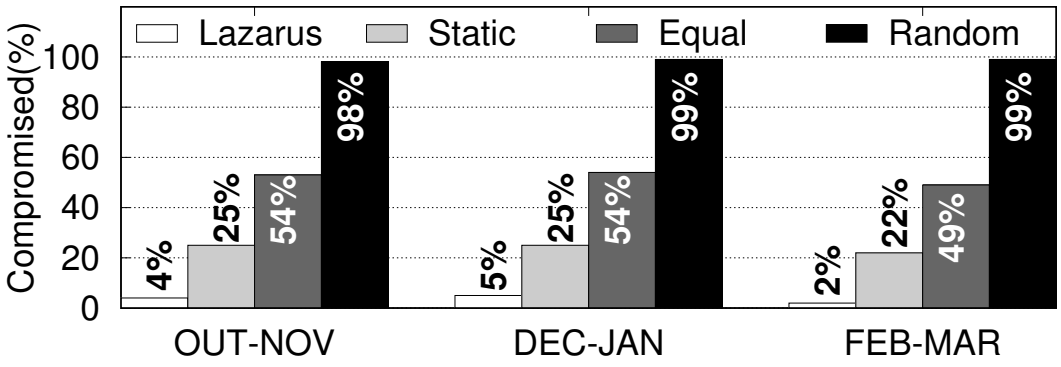



**Figure 5.2** – Compromised system runs over 2 month slots.

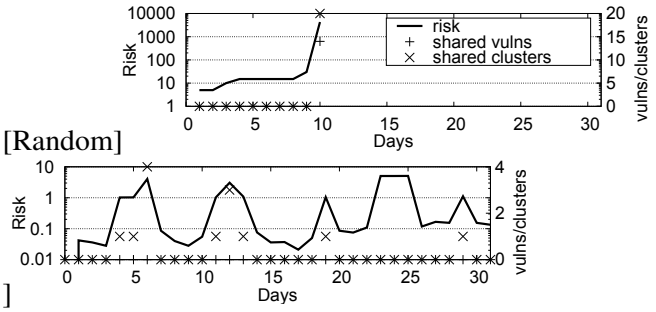
**Results:** Figure 5.2 compares the percentage of compromised runs of all strategies. Each bar represents the percentage of runs that did not terminate successfully (lower is better). In all three periods, LAZARUS presents the best results. The *Random* strategy performs worse because eventually, it picks a group of OSes with common vulnerabilities. This result provides evidence for the claim that LAZARUS improves the dependability, reducing the probability that $f + 1$ OSes eventually become compromised. Interestingly, and contrary to intuition, changing OSes every day with no criteria will always create unsafe configurations. Therefore, it is paramount to have selection strategies like the ones we use in LAZARUS.

☞ **NOTE:** Add all the months we already have

### 5.4.2 Risk evaluation

In order to better understand how LAZARUS performed, we isolated one of the 5000 runs to observe the risk evolution over time. We picked the *Random* and LAZARUS strategies for this analysis, with results displayed in Figure 5.3. The graphs present the evolution of the common vulnerabilities, the common clusters, and our risk metric for both schemes. Notice that two OSes might appear in the same cluster but with no mutual flaw as clusters can include many distinct vulnerabilities.

**Results:** As shown in Figure **??**, *Random* survives only for 10 days. The number of shared clusters and vulnerabilities remains small for the first days. Then, there is a replica




[Random] [LAZARUS




]

**Figure 5.3** – Execution phase for Random and LAZARUS OS configuration strategies (log scale).

| |
|---|
| **Samba:** *On February 2, 2017, security researchers published details about a zero-day vulnerability in Server Message Block (SMB) of Windows, affecting several versions such as 8.1, 10, Server 2012 R2, and Server 2016. Could cause a DoS condition when a client accesses a malicious SMB.*<br>**CVES:** CVE-2017-0016 |
| **Wanna Cry:** *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.*<br>**CVES:** CVE-2017-0143, CVE-2017-0144, CVE-2017-0145, CVE-2017-0146, CVE-2017-0147, CVE-2017-0148 |
| **PowerShell:** *Security feature bypass vulnerabilities in Device Guard that could allow an attacker to inject malicious code into a Windows PowerShell session.*<br>**CVES:** CVE-2017-0219, CVE-2017-0173, CVE-2017-0215, CVE-2017-0216, CVE-2017-0218 |
| **Stackclash:** *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.*<br>**CVES:** CVE-2017-1000365, CVE-2017-1000366, CVE-2017-1000367, CVE-2017-1000369, CVE-2017-1000370, CVE-2017-1000370, CVE-2017-1000371, CVE-2017-1000372, CVE-2017-1000373, CVE-2017-1000374, CVE-2017-1000375, CVE-2017-1000376, CVE-2017-1000379, CVE-2017-1083, CVE-2017-1084, CVE-2017-3629, CVE-2017-3630, CVE-2017-3631 |

**Table 5.2** – Notable attacks during 2017.

replacement that adds to the configuration an OS that has common vulnerabilities with the others.

LAZARUS survives until the end of the experiment, as the risk is continually managed to keep the system safe. Figure **??** shows that shared clusters sometimes increase, at the same pace as the risk. But then, the next reconfigurations are carried out with the goal of decreasing the risk. Notice that the risk value is always under 1 for LAZARUS, and in the *Random* is mostly above 10.

### 5.4.3 Diversity vs Attacks

This experiment evaluates the strategies when facing notable attacks/vulnerabilities that appeared in 2017. 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

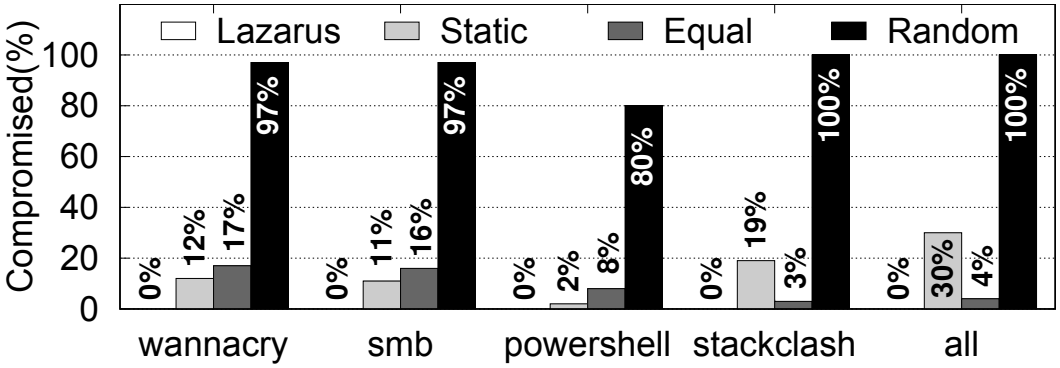



**Figure 5.4** – Compromised runs with notable attacks.

applications, we added more vulnerabilities to the database for this purpose. Table 5.2 lists the attacks and related CVEs: Samba,[4] WannaCry,[5] Powershell,[6] and Stackclash.[7]

Since some of these attacks might have been prepared months before the vulnerabilities are publicly disclosed, we augmented the execution phase to the full six months. As before, the strategies are analyzed over 5000 runs.

**Results:** Figure 5.4 shows the percentage of compromised runs for each attack and all attacks put together. LAZARUS is clearly the best at handling the various scenarios, with no compromised executions. *Random* is the worse, as it does not use any criteria to select the OSes. Both *Equal* and *Static* may perform not so bad as they are static, i.e., the OSes selected by random chance might end up not being exploitable until the end of the run.

## 5.5 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. Our work focuses on two fundamental issues: how to select the best replicas to run together given the current threat landscape, and what is the performance overhead of running a diverse BFT system in practice.

[4] https://www.secureworks.com/blog/attacking-windows-smb-zero-day-vulnerability
[5] https://securityintelligence.com/wannacry-ransomware-spreads-across-the-globe-makes-organizations-wanna-cry-about-microsoft-vulnerability/
[6] http://blog.talosintelligence.com/2017/06/ms-tuesday.html
[7] https://nakedsecurity.sophos.com/2017/06/20/stack-clash-linux-vulnerability-you-need-to-patch-now/

# 6 Supporting Diversity Mechanisms

This chapter presents LAZARUS iplementation, these mechanisms were implemented to be fully automated, removing the human from the loop. The current implementation of LAZARUS 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. We conducted two sets of experiments, one demonstrates that LAZARUS risk management can prevent a group of replicas from sharing vulnerabilities over time; the other, 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 actually perform close to our homogeneous bare metal setup.

## 6.1 LAZARUS Implementation

This section details the implementation of each component of LAZARUS. It also briefly presents other aspects of our prototype.

### 6.1.1 Control Plane

Figure 6.1 shows LAZARUS control plane with its four main modules, described below.

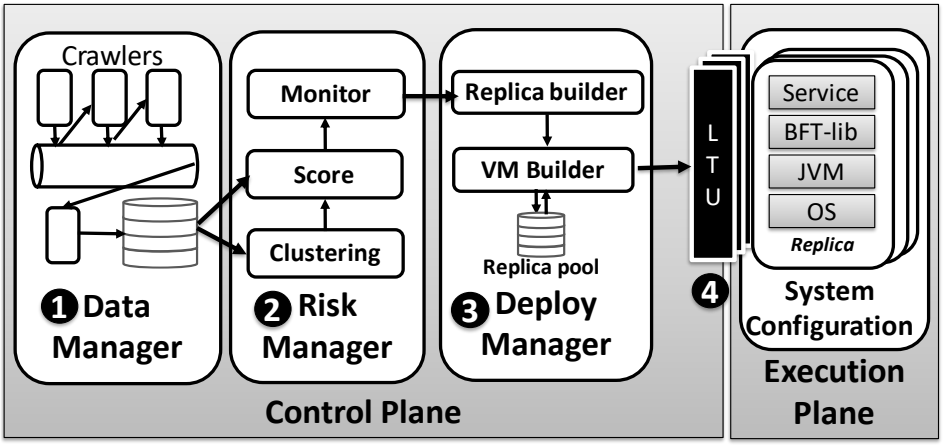



**Figure 6.1** – LAZARUS architecture.

❶ ***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 [90], 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 assemblying 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 five other sources, namely Exploit DB, CVE-details, Ubuntu, Debian, and Microsoft. 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, some CVSS attributes (e.g., availability and integrity); and 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 5.2.2, 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 Section 6.1.2 for details). The *Risk manager* also runs Algorithm 2 to monitor the replicated system and trigger reconfigurations.

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 [57]. 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).

**Setup.** The box configuration is defined in a configuration file, named *Vagrantfile*, consider the example in Listing 6.1. 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 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 6.1 (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 [58]. Vagrant Cloud is a website that offers a plethora of different VMs.

```
Vagrant.configure(2) do |config|
 config.vm.box = "geerlingguy/ubuntu1604"
 config.ssh.insert_key = false
 config.vm.provider "virtualbox" do |v|
  v.customize ["modifyvm", :id, "--cpus", 4]
  v.customize ["modifyvm", :id, "--memory", 22000]
  v.customize ["modifyvm", :id, "--cpuexecutioncap", 100]
  v.customize ["modifyvm", :id, "--ioapic", "on"]
  v.customize ["modifyvm", :id, "--hwvirtex", "on"]
  v.customize ["modifyvm", :id, "--nestedpaging", "on"]
  v.customize ["modifyvm", :id, "--pae", "on"]
  v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
  v.customize ["modifyvm", :id, "--natdnsproxy1", "on"]
 end
 config.vm.network "public_network", ip:"192.168.2.50", bridge:"em1"
 config.vm.provision :shell, path: "run_debian.sh", privileged: true
end
```

**Listing 6.1** – Windows Server 2016 Vagrantfile

✍ **TO-DO**: Add all fields that are mentioned, then add lines to the text

**Download.** One of the fields of the *Vagranfile* 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 official.

**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 *Vagranfile* `provision` field, one can describe the provision steps inline or in an external file and link it to the *Vagranfile*. 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. Although different, all scripts were meant to do the same thing: First the script installs the software that is missing in the box – this is not true for all OSes – 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.

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 TLS channels.

### 6.1.2 Additional Details

**Vulnerability Clustering.** 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 (up to 200 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 K-means algorithm is applied to build the clusters [66], where the number of clusters to be formed is determined by the elbow method [124]. We used the open-source machine learning library Weka [107] to build the clusters.

### 6.1.3 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 just represent the same or similar vulnerabilities.

In order to apply a clustering technique to our data, we need to follow some steps:

**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 [107], the data must be represented as ARFF. Basically, this is a CSV file with some meta information, see Listing 6.2.

```
1 @RELATION vulnerabilities
2 @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 6.2** – 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.

- **stopwords**, we define a stop word list, then words from this list are discarded. We begin with a general 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 way that similar vulnerabilities, even if they affect different products, are put together in the same cluster. For example, recall the vulnerabilities in Table 5.1, 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. We have done this by iterating the process and checking the most used words (top 200 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 [1] 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. And 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. The *K-means* has two important parameters: the number of clusters to be formed, we set to 200 clusters. We used the elbow method [124] to decide $k = 200$; and the *distance function*, to calculate the distance between objects, the *Euclidean Distance* is the most adequate for this type of data; 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).

[1] Stop word list is available: here

Sometimes the resulting clusters include vulnerabilities that are unrelated. Therefore, we use the Jaccard index (J-index) to measure the similarity between the vulnerabilities within the cluster. We calculate the J-index of each element, and then the average J-index of the cluster. The clusters with a smaller average J-index (below a certain threshold) are considered ill-formed. In this case, we select the vulnerabilities with lower J-index and move them to another cluster that would result in a better J-index. If no such cluster exists, then we create a new one with the "orphan" vulnerability.

**BFT replication.** Although there has 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 [17], a stable BFT library that provides reconfigurations on the *replicas* set.

**Replica Virtualization.** VMs can be used to implement replicated systems, leveraging on the isolation between the untrusted and the trusted domains [38, 101, 118]. 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 [57] 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 [108] because it offers more diversity opportunities, i.e., it supports a more extensive set of different guests OSes.

## 6.2 Performance 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

| 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 6.1** – The different OSes used in the experiments and the configurations of their VMs and JVMs.

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.

Table 6.1 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.

### 6.2.1 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., [13, 14, 17, 23, 83]). 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.

**Results:** Figure 6.2 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.

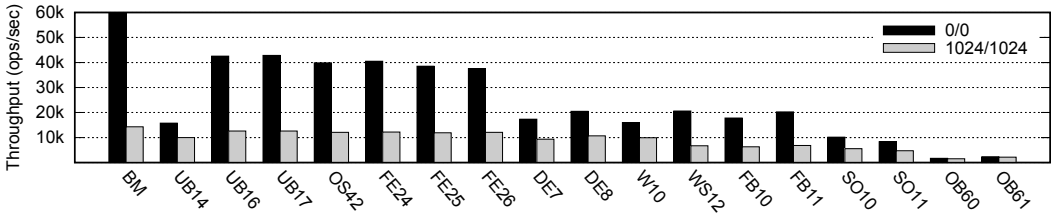



**Figure 6.2** – Microbenchmark for 0/0 and 1024/1024 (request/replying) for homogeneous OSes configurations.

### 6.2.2 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 6.3 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 $1024/1024$ workload ($82\%$ and $18\%$ of the bare metal performance). When comparing these two sets with the non-diverse sets of Figure 6.2, 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

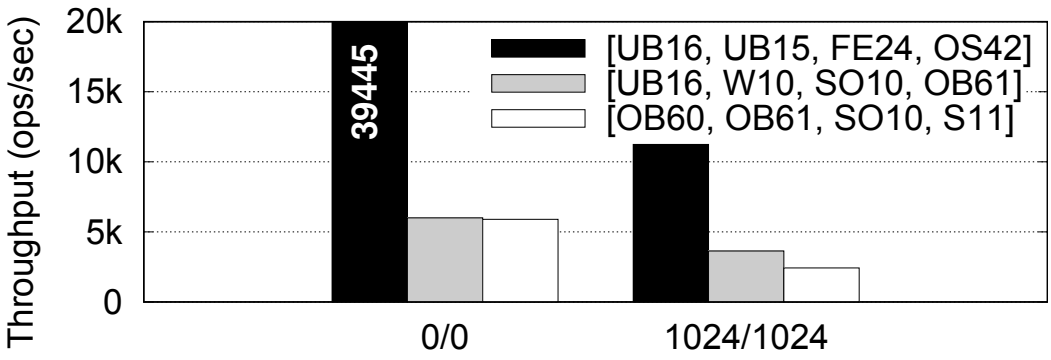



**Figure 6.3** – Microbenchmark for 0/0 and 1024/1024 (request/reply) for three diverse OS configurations.

selected by the system. This opens interesting avenues for future work on protocols that consider such performance diversity, as will be discussed in Section **??**.

### 6.2.3 Performance During Reconfiguration

In this experiment, we show how LAZARUS-triggered reconfigurations affect the replicas' performance. Reconfigurations, in this case, subsumes to the addition of a new replica and the removal of the old one, using the BFT-SMaRt reconfiguration protocols [**?** ]. We execute this experiment with an in-memory KVS service that comes with the BFT-SMaRT library.

The experiment was conducted with a Yahoo! Cloud Serving Benchmark (YCSB) [31] 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. We run this experiment for 400 seconds, and reconfigure the set of replicas with a period of 100 seconds. The initial OS configuration is the fastest OS set (i.e., UB17, UB16, FE24, OS42).

[KVS state size ≈ 30MBs.]

[KVS state size ≈ 60MBs.]

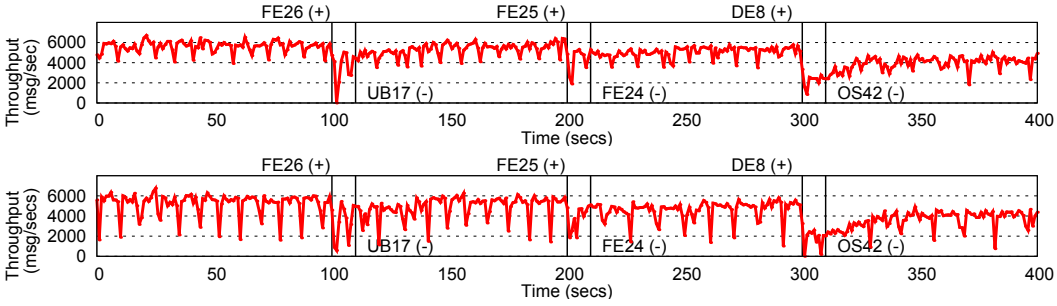



**Figure 6.4** – KVS performance with LAZARUS-triggered reconfigurations on a 50/50 YCSB workload and 1kB-values.

**Results:** Figure 6.4 shows two types of performance drops. The first type happens at every 1000 writes and is due to the state checkpoints used to trim the operation logs. The second

type is more severe and less frequent, and happens during reconfigurations, mostly due to the state transfer to the new replica joining the system. These types of performance perturbations, which become more severe with bigger states, were already identified, and mitigated in previous works [16].[2]

The figure shows that the reconfigurations take around 10 seconds for these setups, and did not change significantly with states of approx. 30 and 60 MBs. Regarding diversity, the figure shows no noticeable disruption when changing Ubuntu 17 to Fedora 26 and Fedora 25 to Fedora 24, but the replacement of a Debian 8 by an OpenSuse 42 replica significantly decreases the performance, which is consistent with the results from Figure 6.2.

### 6.2.4 Booting time

Before triggering a reconfiguration, LAZARUS needs to start the new VM, boot the OSes, and launch the BFT-SMaRt replica. Therefore, the total time to replace a "risky" replica roughly comprises the time required to boot the new OS plus the reconfiguration time (discussed in the previous section). This experiment measures the boot time of the OSes supported by LAZARUS prototype.

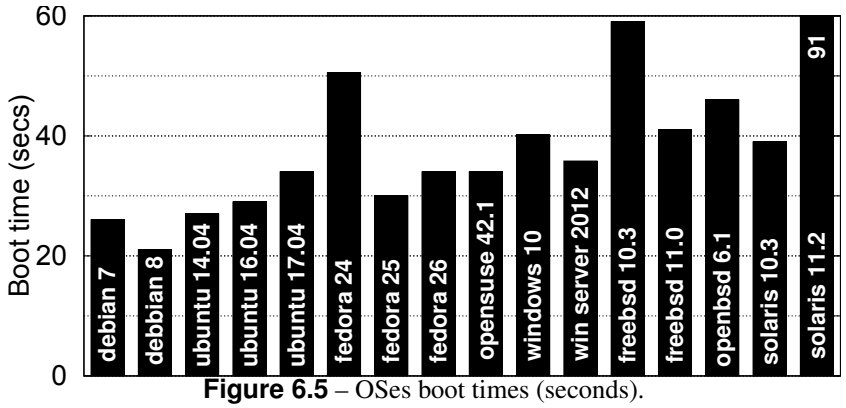



**Figure 6.5** – OSes boot times (seconds).

**Results:** Figure 6.5 shows the average boot time of 20 executions boot for each OS. As can be seen, most OSes take less than 40s to boot, with exceptions of Fedora 24 (50s), FreeBSD 10.3 (59s), and Solaris 11.2 (91s). 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).

[2] We employ the standard checkpoint and state transfer protocols of BFT-SMaRt and not the one introduced in [16] as the developers of the system pointed them as more stable.

### 6.2.5 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 6.2.3. 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 [14, 83]). In this evaluation, we employ the YCSB $50\%/50\%$ read/write workload with values of 4k bytes.

- SIEVEQ [49] 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 to the system is composed of messages of 1k bytes.

- *BFT ordering for Hyperledger Fabric* [117] is the first BFT ordering service for Fabric [8]. 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 6.2.2, we run the applications on the fastest and slowest diverse replica sets and compare them with the results obtained in bare metal.

**Results:** Figure 6.6 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).

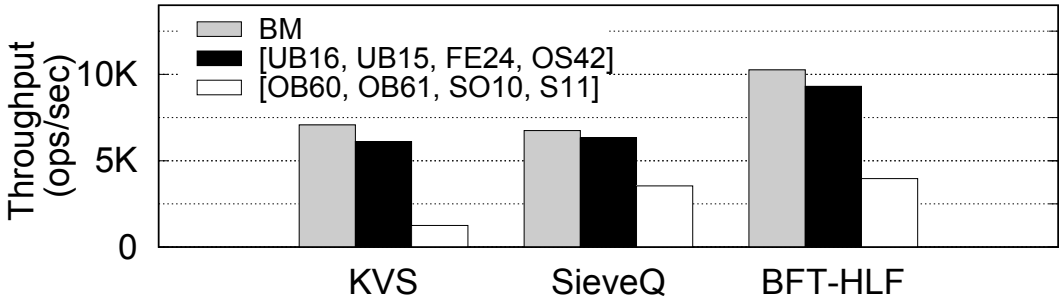



**Figure 6.6** – Different BFT applications running in the bare metal, fastest and slowest OS configurations.

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 [117]

## 6.3 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. Our work focuses on two fundamental issues: how to select the best replicas to run together given the current threat landscape, and what is the performance overhead of running a diverse BFT system in practice.

# 7 Untrusted (distributed) controller

As in previous works [101, 110], our current design for SIEVEQ and LAZARUS considers a centralized trusted control plane that analyze OSINT and orchestrate replica group reconfigurations. It would be desirable to have a distributed version of such control plane, not only for improving its dependability, but also to support the existence of multi-domain applications, such as blockchain platforms.

# 8 Conclusion

## 8.1 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. Our work focuses on two fundamental issues: how to select the best replicas to run together given the current threat landscape, and what is the performance overhead of running a diverse BFT system in practice.

## 8.2 Future work

Our promising results open many avenues for future work in this topic.

**Integration with other OSINT and IDSs:** LAZARUS monitors only five security data feeds on the internet looking for vulnerabilities, exploits, and patches in the OSes it manages, but it could be extended to monitor other indicators of compromise (e.g., IP black lists) extracted from much richer set of sources [82, 111]. Similarly, LAZARUS can be extended to additionally use the outputs of IDSs to assess the BFT system behaviour and trigger replica reconfigurations in case of need.

**Diversity-aware replication:** The evaluation of BFT-SMaRt on top of LAZARUS shows that different replica group configurations can have a huge 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 [116] could be used to assign higher weights to the replicas running in faster replicas.

**Trusted components:** Our prototype implements the LTU as a trusted component isolated from the rest of the replica through virtualization, just like many previous works on hybrid BFT [38, 101, 110, 118, 128]. The recent popularization of trusted computing technologies such as Intel SGX [65], and its use for implementing efficient BFT replication [14], open

interesting possibilities for using novel hardware to support services like LAZARUS on bare metal.

# Acronyms

**ARFF** Attribute-Relation File Format. 57–59

**ASLR** Address Space Layout Randomization. 12

**BFT** Byzantine fault-tolerant. 4–7, 9–11, 14–19, 21–25, 41, 47–51, 59, 71, 74, 75, 79–82

**BM** Bare Metal. 75

**CIS** CRUTIAL Information Switch. 14, 15

**COTS** Components Off-the-Shelf. 13

**CPE** Common Platform Enumeration. 49, 53

**CVE** Common Vulnerabilities and Exposures. 49, 54, 57, 73

**CVSS** Common Vulnerability Scoring System. 49–51, 54

**DB** Database. 19

**DBMS** Databases Management Systems. 24

**DDoS** Distributed Denial-of-Service. 15

**DoS** Denial-of-Service. 15, 22, 28, 30, 34, 37, 41–43, 45, 46, 61, 65, 66, 74

**HMAC** Hash-based Message Authentication Code. 45

**ICS** Industrial Control Systems. 2, 46

**IDS** Intrusion Detection System. 13, 27, 64, 81

**JVM** Java Virtual Machine. 19, 24, 63, 75

**KMP** Knuth–Morris–Pratt. 64

**KVS** Key-Value Storage. 22, 49, 77, 80

**LTU** Logical Trusted Unit. 18, 21, 22

**MAC** Message Authentication Code. 9, 30, 35–38, 41, 42, 45

**NFS** Network File System. 18

**NIST** National Institute of Standards and Technology. 1, 49

**NVD** National Vulnerability Database. 7, 13, 16, 17, 22, 27, 49, 50, 53, 54

**OS** Operating System. 5, 6, 13, 16, 18–20, 24, 25, 32, 44, 47–49, 55, 56, 59, 71–78, 81

**OSI** Open Systems Interconnection. 27

**OSINT** Open Source Intelligence. 7, 19, 22, 24, 48, 50, 51, 54, 81

**OSVDB** Open Sourced Vulnerability Database. 17

**OTS** Off-the-Shelf. 5, 6, 12–14, 18, 19, 24, 34

**PO** Proactive Obfuscation. 18, 19

**POA** Potential for Attack. 16

**PPZDA** Potential Pseudo Zero-Day Attack. 15

**PRRW** Proactive-Reactive Recovery Wormhole. 11

**PZDA** Pseudo Zero-Day Attack. 15, 16

**RSA** Rivest–Shamir–Adleman. 45

**SCADA** Supervisory Control and Data Acquisition. 2, 27

**SHA** Secure Hash Algorithm. 45

**SIEM** Security Information and Event Management. 28, 46, 61, 67, 68

**SMR** State Machine Replication. 3, 4, 9, 45, 65

**SVM** Support Vector Machines. 17

**TCP** Transmission Control Protocol. 45, 62, 66

**TLS** Transport Layer Security. 56

**TOM** Total Ordered Multicast. 32, 40, 42–45

**UDP** User Datagram Protocol. 62

**VM** Virtual Machine. 11, 19, 21, 22, 24, 48, 55, 56, 59, 75, 77, 78

**VMM** Virtual Machine Manager. 11

**XML** Extensible Markup Language. 49

**XSS** Cross-site scripting. 50

**YCSB** Yahoo! Cloud Serving Benchmark. 77, 79

**ZDA** Zero-Day Attack. 15

# Bibliography


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

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

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

[4] 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. Technical report, Weakdh, 05 2015.

[5] 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.

[6] 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, 2011.

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

[8] 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. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the ACM European Conference on Computer Systems*, 2018.

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

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

[11] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tolerating File-system Mistakes with EnvyFS. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2009.

[12] B. Baudry and M. Monperrus. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Computer Surveys*, 48(1):16:1–16:26, Sept. 2015.

[13] J. Behl, T. Distler, and R. Kapitza. Consensus-Oriented Parallelization: How to Earn Your First Million. In *Proceedings of the Middleware Conference*, 2015.

[14] 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*, 2017.

[15] 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*, pages 15–18, June 2011.

[16] 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.

[17] 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*, 2014.

[18] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits.

[19] 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 Conference on Computer and Communications Security*, pages 833–844. ACM, 2012.

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

[21] 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*, 2010.

[22] B. Calder et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2011.

[23] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186, 1999.

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

[25] J. Chauhan and R. Roy. Is Firewall and Antivirus Hacker's Best Friend?, Jan 2011.

[26] L. Chen and A. Avizienis. N-version programming: A fault tolerance approach to reliabiltiy of software operation. In *International Symposium on Fault-Tolerant Computing*, Jun 1995.

[27] Cisco. Multiple vulnerabilities in firewall services module, Feb. 2007.

[28] Cisco. Multiple vulnerabilities in Cisco firewall services module, Oct. 2012.

[29] 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*, 2009.

[30] 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 Systems Design and Implementation*, 2009.

[31] B. F. 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*, 2010.

[32] J. Corbett et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems*, 31(3), 2013.

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

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

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

[36] 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*, 1998.

[37] C. Details. CVE Details Website. http://www.cvedetails.com/, 2018. Accessed: 2018-4-26.

[38] 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*, 2011.

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

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

[41] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of Workshop on Hot Topics in Operating Systems*, pages 67–72, May 1997.

[42] J. Fraga and D. Powell. A fault-and intrusion-tolerant file system. In *Proceedings of the International Conference on Computer Security*, volume 203, pages 2–s2, 1985.

[43] 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, September 2015. Accessed: 2017-10-17.

[44] S. Frei, M. May, U. Fiedler, and B. Plattner. 6large-scale vulnerability analysis.

[45] M. Garcia and A. B. amd N. Neves. In *Proceedings of the ACM European Conference on Computer Systems*.

[46] 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*, pages 383–394, June 2011.

[47] 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, 2014.

[48] M. Garcia, N. Neves, and A. Bessani. An intrusion-tolerant firewall design for protecting SIEM systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (Workshop)*, pages 1–7, June 2013.

[49] 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, 2018.

[50] 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.

[51] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum. Practical automated vulnerability monitoring using program state invariants. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, June 2013.

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

[53] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and A. Romanovsky. Using diversity in cloud-based deployment environment to avoid intrusions. In *Proceedings of the International Conference on Software Engineering for Resilient Systems*, 2011.

[54] M. Guo and P. Bhattacharya. Diverse Virtual Replicas for Improving Intrusion Tolerance in Cloud. In *Proceedings of the Annual Cyber and Information Security Research Conference*. ACM, 2014.

[55] J. Han, D. Gao, and R. H. Deng. *On the Effectiveness of Software Diversity: A Systematic Study on Real-World Vulnerabilities*, pages 127–146. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[56] D. Harkins and D. Carrel. The Internet Key Exchange, 1998.

[57] HashiCorp. Vagrant. https://www.vagrantup.com/, 2017. Accessed: 2018-4-26.

[58] HashiCorp. Vagrant Cloud. https://app.vagrantup.com/boxes/search, 2017. Accessed: 2018-4-26.

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

[60] 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*, 2013.

[61] J. Hong and D.Kim. *IEEE Transactions on Dependable and Secure Computing*.

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

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

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

[65] 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, 2018. Accessed: 2018-4-26.

[66] A. K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010.

[67] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and W. Powell. Catch Me If You Can: A Cloud-Enabled DDoS Defense. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[68] 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*, 2008.

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

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

[71] A. D. Keromytis and V. Prevelakis. Designing Firewalls: A Survey. In C. Douligeris, editor, *Network Security: Current Status and Future Directions*, chapter 3, pages 33–49. John Wiley & Sons, Inc., 2006.

[72] 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.

[73] J. C. Knight and N. G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, 12(1), Jan. 1986.

[74] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

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

[76] 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*, pages 43–52, Nov 2014.

[77] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions Programming Languages Systems*, 6(2):254–280, Apr. 1984.

[78] P. Larsen, S. Brunthaler, and M. Franz. Security through Diversity: Are We There Yet? *IEEE Security & Privacy*, 12(2):28–35, Mar 2014.

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

[80] C.-L. Lee, Y.-S. Lin, and Y.-C. Chen. A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection. *PLoS ONE*, 10(10), 10 2015.

[81] 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*, 2014.

[82] 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*, 2016.

[83] 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*, 2016.

[84] 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*, 2015.

[85] 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*, 2013.

[86] F. Massacci and V. H. Nguyen. Which is the Right Source for Vulnerability Studies?: An Empirical Analysis on Mozilla Firefox. In *Proceedings of the International Workshop on Security Measurements and Metrics*, 2010.

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

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

[89] 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*, 2011.

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

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

[92] 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*, 2003.

[93] National Institute of Standards and Technology. *Guide for Conducting Risk Assessments*. 2012.

[94] Netfilter. The netfilter.org project. Accessed: 2018-4-26.

[95] 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.

[96] NIST. National Vulnerability Database. http://nvd.nist.gov/, 2018. Accessed: 2018-4-26.

[97] 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*, 2018.

[98] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? Technical Report TR 06–15, Department of Informatics.

[99] H. Okhravi, A. Comella, E. Robinson, and J. Haines. Creating a cyber moving target for critical infrastructure applications using platform diversity. *International Journal of Critical Infrastructure Protection*, 5(1):30–39, Mar. 2012.

[100] 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.

[101] 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*, pages 242–252, Oct 2014.

[102] K. Prabha and S. Sukumaran. Improved single keyword pattern matching algorithm for intrusion detection system. *International Journal of Computer Applications*, 90(9):26–30, March 2014.

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

[104] 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*, 2018.

[105] H. P. Reiser and R. Kapitza. VM-FIT: supporting intrusion tolerance with virtualisation technology. In *Proceedings of the Workshop on Recent Advances on Intrusiton-tolerant Systems*, 2007.

[106] Kaspersky Lab. The State of Industrial Cybersecurity 2017. https://go.kaspersky.com/rs/802-IJN-240/images/ICS

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

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

[109] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 15–28, 2001.

[110] T. Roeder and F. B. Schneider. Proactive Obfuscation. *ACM Transactions on Computer Systems*, 28(2):4:1–4:54, July 2010.

[111] 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*, 2015.

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

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

[114] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. 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*, 2013.

[115] 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*, pages 305–316, May 2010.

[116] 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.

[117] 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*, 2018.

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

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

[120] 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 *USENIX Network and Distributed System Security Symposium*, page 21–24, February 2013.

[121] 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*, 2010.

[122] 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`, July 2014.

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

[124] R. L. Thorndike. Who belongs in the family. *Psychometrika*, pages 267–276, 1953.

[125] 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*, 2005.

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

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

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

[129] 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. 2013.

[130] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer. 6a stateful intrusion detection system for world-wide web servers.

[131] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS Defense by Offense. 28(1), Aug. 2010.

[132] L. Wang, S. Jajodia, A. Singhal, P. Cheng, and S. Noel. k-Zero Day Safety: A Network Security Metric for Measuring the Risk of Unknown Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(1):30–44, Jan 2014.

[133] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. 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.

[134] Y. Yeh. Unique dependability issues for commercial airplane fly by wire systems. In *IFIP World Computer Congress: Building the Information Society*, 2004.

[135] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement form Execution for Byzantine Fault Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[136] S. Zargar, J. Joshi, and D. Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys Tutorials*, 15(4), 2013.

[137] 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.

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