Lazarus: Automatic Management of Diversity in BFT Systems

Miguel Garcia
miguel.garcia.th@gmail.com
LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal

Alysson Bessani
anbessani@ciencias.ulisboa.pt
LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal

Nuno Neves
nfneves@ciencias.ulisboa.pt
LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal

Abstract

A long-standing promise of Byzantine Fault-Tolerant (BFT) replication is to maintain the service correctness despite the presence of malicious failures. The key challenge here is *how to ensure replicas fail independently*, i.e., avoid that a single attack compromises more than f replicas at once. The obvious answer for this is the use of *diverse replicas*, but most works in BFT simply assume such diversity without supporting mechanisms to substantiate this assumption. Lazarus is a control plane for managing the deployment and execution of diverse replicas in BFT systems. Lazarus continuously monitors the current vulnerabilities of the system replicas (reported in security feeds such as NVD and ExploitDB) and employs a metric to measure the risk of having a common weakness in the replicas set. If such risk is high, the set of replicas is reconfigured. Our evaluation shows that the devised 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.

ACM Reference Format:
Miguel Garcia, Alysson Bessani, and Nuno Neves. 2019. Lazarus: Automatic Management of Diversity in BFT Systems. In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3361525.3361550

1 Introduction

Practical Byzantine Fault-Tolerant (BFT) replication was proposed as a solution to handle Byzantine faults of both accidental and malicious nature [19]. 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 [6, 20, 46]. This approach only works as expected if nodes fail independently, otherwise, once an attacker finds a way to exploit a vulnerability in one node, it is most likely that the same attack can also be used to successfully compromise the other replicas.

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., [6, 11, 46]), use of resources (e.g., [12, 48, 77]), and robustness (e.g., [4, 15, 23]) of BFT systems. These works assume, either implicitly or explicitly, that replicas fail independently, relying on some orthogonal mechanism (e.g., [22, 66]) 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 [21, 58, 66], but in a very limited way. Nonetheless, in practice, diversity is fundamental for building dependable services in avionics [81], military systems [32], and even in blockchains (e.g., Bitcoin [41] and Ethereum [29]).

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 [66] or different OSes [21, 43], 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 [17, 68]. In addition, although the use of distinct OSes promotes fault independence to some extent, *per se* it is not enough to preclude vulnerability sharing among diverse OS distributions [34, 35].

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 [20, 28, 58, 66, 71] 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, which 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

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361550

dependable way. LAZARUS continuously collects security data from Open Source Intelligence (OSINT) feeds on the internet to build a knowledge base about the possible vulnerabilities, exploits, and patches related to the software 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 a 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, significantly reducing the burden of managing different software.

In summary, this paper makes 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 3 and 5);
2. A method for assessing the risk of a group of replicas being compromised based on the security news feeds available on the internet (Section 4);
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 6);
4. An extensive evaluation of LAZARUS prototype using 17 OS versions, a BFT replication library, and three BFT applications (including a BFT ordering service for the Hyperledger Fabric blockchain platform) showing the that it is feasible to run BFT systems with diversity for specific configurations (Section 7).

2 Revisiting BFT for Security

In its inception, practical BFT replication was motivated by the prevalence of software errors and malicious attacks [19]. However, as the time passed, BFT solutions focused more on non-malicious Byzantine failures, such as hardware defects and non-deterministic bugs. We believe one of the reasons for this change was the difficulty in ensuring correctness in the presence of malicious adversaries by relying on the BFT protocol alone. In particular, it is hard to build a practical approach that forces an *intelligent attacker* to find more than f weaknesses in different replicas to be able to compromise the system or, alternatively, that reduces the probability of *a single attack compromising more than one replica.* Additionally, known and unknown vulnerabilities must be cleaned from the system, to force the attacker to look for unknown or unpatched weaknesses. In consequence, *replicas diversity* and *recovery* can be seen as two pillars of *intrusion tolerance* [30, 76], together with BFT replication.

Despite the existence of several works addressing these issues [20, 21, 58, 66, 71], to the best of our knowledge no previous proposal showed *how to avoid the compromise of the* $f + 1^{th}$ *replica with the same vulnerability.* LAZARUS fills this gap by employing threat intelligence techniques (e.g., [2, 18, 49, 67, 79]) for managing a pool of diverse replicas supporting a BFT system. More specifically, LAZARUS constantly collects cybersecurity data from OSINT sources such as National Vulnerability Database (NVD) and ExploitDB (among others) to procure information about new weaknesses on the replicas software stack, which is used to reconfigure the set of active replicas. With this, we aim to alleviate two kinds of threats:

1. *Newly found vulnerabilities affecting one or more active replicas*, by identifying such threats in a timely manner and by replacing the affected replica(s) by other variants while applying the security updates (or patches);
2. *Zero-day vulnerabilities affecting multiple running replicas*, by leveraging historical data about exploits, patches, and vulnerabilities that compromise simultaneously several systems to infer subsets of replicas less likely to contain common vulnerabilities in the future.

For addressing (1), LAZARUS employs relatively straightforward techniques: when new critical vulnerabilities are published, alarms are generated and active replicas are removed from the system and moved to quarantine. However, addressing (2) is a significant challenge as it requires predicting the set of active replicas, among the ones available, with less risk of containing common vulnerabilities. One of the key contributions of this work is devising a method to make this prediction, taking into account the plethora of public information about vulnerabilities, exploits, and patches, and known results about vulnerability studies in the scientific literature. For example, we know that looking for past vulnerability data can give hints about which pairs of operating systems are more likely to have shared vulnerabilities in the future [35]. We also know that not all vulnerabilities have the same chance to be exploited [67], or break the same security properties, nor have the patches available at the same time from the affected vendors. We know (unsurprisingly) that patched [16] or old [33] vulnerabilities are less likely to be exploited. As a last example, it is also known that although NVD' Common Vulnerability Scoring System (CVSS) [25] consolidates important information about vulnerabilities, it is not particularly good at measuring its exploitability [1, 18].

LAZARUS takes into account all this information to maintain correctness in an evolving threat landscape, where novel vulnerabilities are continuously found, disclosed, and patched. Nevertheless, it is important to note that our proposal cannot preclude the discovery of $f + 1$ zero-day vulnerabilities in distinct replicas. However, in this case, our goal has been met as we forced the attacker to spend resources proportional to f. In addition, coercing an adversary to find $f+1$ weaknesses

to compromise a BFT system is arguably harder than finding a single vulnerability in a non-replicated system, thus bringing important security gains.

3 Overview and Preliminaries

In a nutshell, LAZARUS provides a distributed operating system for BFT-replicated services. The system manages in its execution plane a set of nodes that run *unmodified replicas*. Each node must have a small *Local Trusted Unit (LTU)* that allows the activation and deactivation of replicas as demanded by the LAZARUS *Controller*, in the control plane. The controller decides which software should run at any given time by monitoring the vulnerabilities that may exist in the replicas pool, replacing the nodes with high risk of being compromised by the same attack. Figure 1 presents this architecture.

3.1 System and Adversary Model

LAZARUS system model shares some similarity with previous works on the proactive recovery of BFT systems [20, 28, 58, 66, 71]. More precisely, we consider a *hybrid distributed system model* composed of two planes with different failure models:

- *Execution Plane:* replicas are subject to *Byzantine failures* and communications go through an asynchronous network that can delay, drop or modify messages [6, 15, 19, 46]. 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 $n = 3f + 1$.
- *Control Plane:* Each *node* hosting a replica contains an LTU, which is a fundamental trusted component required for *safe proactive recoveries* [72]. Each LTU receives power on/off commands from a *logically-centralized controller* to reconfigure their replicas. As in previous works [58, 66, 71], this controller is assumed to be trusted. Such assumption can be substantiated by running it in an isolated control network (similarly to cloud resource managers) our by building it in a trustworthy manner (tolerating Byzantine failures, as discussed in Section 5.3).

Besides the execution and control planes, we assume the existence of two types of external components: (1) clients of the replicated service, which can be subject to Byzantine failures; (2) OSINT sources (e.g., NVD, ExploitDB) that cannot be subverted and controlled by the adversary. In practice, this assumption leads us to resort only to well-established and authenticated data sources. Dealing with untrusted sources is an active area of research in the threat intelligence community (e.g., [2, 67]), which is orthogonal to this paper.

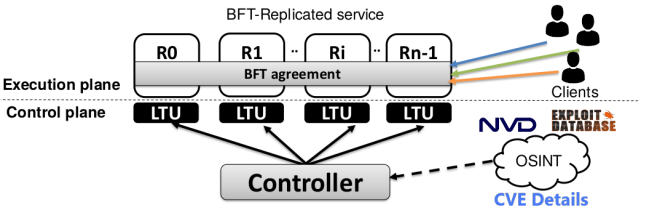


Figure 1. LAZARUS overview.

3.2 Diversity of Replicas

For our purposes, each *replica* is composed of a stack of software, including an OS (kernel plus other software contained in an OS distribution), execution support (e.g., database, JVM), a BFT library, and the service that is provided by the system. The set of n replicas is called a `CONFIG`.

The fault independence of the replicas is improved when different OTS (Off-The-Shelf) components are employed in the software stack [27]. For example, it has been shown that using distinct databases [37], OSes [34, 35], filesystems [10], and web browsers [80], can yield important benefits in terms of fault independence. In addition, automatic techniques like randomization/obfuscation of OSes [66], applications [78], and BFT libraries [58] can enhance fault independence of the replicas. Although LAZARUS replicas can benefit from these techniques, in this paper we focus on diverse OTS components. In particular, LAZARUS monitors the disclosed vulnerabilities of all replicas' software stacks to assess which of them may contain common vulnerabilities.

In the experimental evaluation, however, we focus on the diversity of OSes (*not only their kernel, but the whole OS distribution*) because: (i) by far, most of the replica's code is the OS; (ii) such size and the role played in the stack, makes the OS a significant target, with new vulnerabilities and exploits being discovered every day; and (iii) there are many alternatives to bring variety. The two last factors are particularly important to enrich the validity of our analysis. Consequently, we will not explicitly consider the diversity of the BFT library (i.e., the protocol implementation) or the service code implemented on top of it. Three arguments justify this decision: (1) N-version programming is too costly [7]; (2) the small size of such components[1] makes them relatively simpler to test and assess with some confidence [47, 50]; and (3) a few works show that such protocol implementations can be generated from formally verified specifications, resulting in no vulnerabilities at this level [39, 60]; Although we do not consider the diversity of BFT libraries, nothing prevents LAZARUS from monitoring them in a similar way as the OSes. However, there are no reported vulnerabilities about these to support our study.

[1]For example, a BFT key-value store based on BFT-SMaRt has less than 15k lines of code in total [15].

4 Diversity-aware Reconfigurations

Lazarus aims to maintain the fault independence of the running CONFIG, given the present knowledge about vulnerabilities. The core of Lazarus is the *vulnerability evaluation method* used to assess the risk of having replicas with shared vulnerabilities, and an algorithm to trigger replacements when necessary.

4.1 Finding Common Vulnerabilities

The first step of our method is to query vulnerability databases on the internet to find the common vulnerabilities that may affect the BFT system.

NIST's NVD [55] is the authoritative data source for disclosure of vulnerabilities and associated information [51]. 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 in 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) [54]. The Common Platform Enumeration (CPE) [53] provides the list with the various products affected by the vulnerability and the date of the vulnerability publication. The CVSS [25] calculates the vulnerability severity considering a few attributes like the attack vector, privileges required, exploitability score, and the security properties [8] compromised by the vulnerability.

Previous studies on diversity count the CVE entries that list multiple OSes, as they should represent vulnerabilities that are shared, assuming that less common vulnerabilities imply a smaller probability of compromising $f + 1$ replicas [34, 35]. Although this intuition may seem acceptable, in practice it underestimates the number of vulnerabilities that compromise two or more OSes due to imprecisions in the data source. For example, Table 1 shows three vulnerabilities, affecting different OSes at distinct dates. At first glance, one may consider that these OSes do not have the same vulnerability. However, a careful inspection of the descriptions shows that they are very similar. Moreover, we checked this resemblance by searching for additional information and found out that CVE-2016-4428 also affects Solaris.[2]

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 [42]. Clustering is the process of aggregating related elements into groups, named clusters, and is one of the most popular unsupervised machine learning techniques. We apply this technique to build clusters of similar vulnerabilities (see Section 5), even if the data feed reports that they affect different products. For example, the vulnerabilities in Table 1 will be placed in the same cluster due to the similarity in their descriptions, which can make them potentially be activated by (variations of) the same exploit.

CVE (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 cross-site scripting (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)	Cross-site scripting (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 1. Similar vulnerabilities affecting different OSes.

4.2 Measuring Vulnerability Severity

Once the set of common vulnerabilities is found, our method assigns a score to each vulnerability in the set.

As mentioned before, each vulnerability in NVD has associated a few CVSS severity scores and metrics [25]. The scores provide a way to marshal several vulnerability attributes in a value reflecting various aspects that impact security. The score value also has a qualitative representation to assist on the vulnerability management process, i.e., from NONE (0.0) to CRITICAL (9.0 to 10.0).

CVSS has some limitations that can make it inappropriate for managing the risk associated with a replicated system: (1) there is no correlation between the CVSS exploitability score and the availability of exploits in the wild for the vulnerability [18]; (2) CVSS does not provide information about the date when a vulnerability starts to be exploited and when the patch becomes ready; and (3) CVSS does not account for the vulnerability age, which means that severity remains the same over the years [33]; therefore, a very old vulnerability can end up being considered as critical as a recent one, even though for the former there has been plenty of time to update the component and/or the defenses.

Given these shortcomings, we propose a CVSS extension and use it to measure the risk of a BFT system configuration having replicas with shared vulnerabilities. This extension is mostly focused on differentiating vulnerabilities by their

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

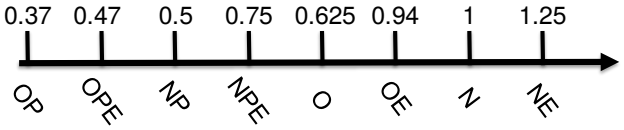


Figure 2. Modifiers of vulnerabilities scores based on age and the existence of patches and exploits.

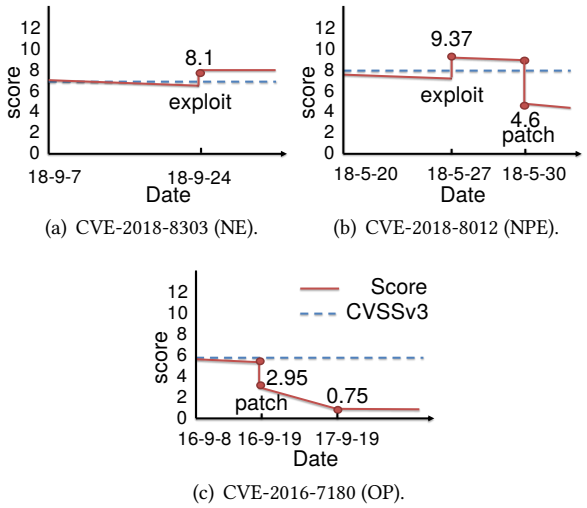
current *potential exploitability*, aiming to surpass the limitations identified above. In this process, (1) and (2) were addressed by using additional OSINT sources (e.g., other security databases and vendor sites) that provide information about exploits and dates. In fact, often vendor sites also give additional product versions compromised by the vulnerability, thus improving the accuracy of the analysis. Limitation (3) is settled by decreasing the criticality of a vulnerability gradually through time.

Our CVSS extension uses four factors that together contribute to the overall score. The starting factor is the CVSS core score, as it is a good basis that takes into consideration several attributes of the vulnerability. The other three factors adjust the score taking into account the age and the availability of patches and exploits. The rationale is to allow the ranking of vulnerabilities according to their possible exploitation at a given moment in time. The worst scenario (higher severity score) corresponds to a vulnerability that is new (N) (i.e., recently published), for which there is an exploit already being distributed (E), and that is not yet patched – called NE. The best scenario (lowest score) is when a vulnerability is old (O) and there is a patch (P), and apparently, no viable exploit has been crafted – named OP. Between the two extremes, several cases of vulnerabilities are considered, with their scores calculated accordingly (see Figure 2).

The metric is defined in Equation 1. It is a multiplication of the CVSS core score [25] by three adjusting factors. The first is *oldness*, which causes criticality to decrease over time. It is harmonized by the *oldness_threshold* and the elapsed time since the vulnerability publication[3] (Equation 2). In addition, this factor is bounded by a minimum value that impedes it from reaching zero (which would cause the vulnerability to be left unnoticed). The second is *patched*, which reduces the severity by half when a patch is available (Equation 3; $v.patched$ is an on/off flag). Finally, the *exploited* factor grows severity by a quarter when an exploit is made available (Equation 4; $v.exploited$ is again a on/off flag). The constants in these equations were defined to ensure the aggregated modifiers corresponds to Figure 2.

$$score(v) = CVSS(v) \times oldness(v) \times patched(v) \times exploited(v) \quad (1)$$

[3] *oldness_threshold* is set to 365 days in our experiments; *now* and $v.published\_date$ return the current day and the day when the vulnerability was published, respectively.



(a) CVE-2018-8303 (NE). (b) CVE-2018-8012 (NPE).

(c) CVE-2016-7180 (OP).

Figure 3. *Score* evolution for three vulnerabilities.

$$oldness(v) = max\left((1 - 0.25 \times \frac{(now - v.published\_date)}{oldness\_threshold}), 0.75\right) \quad (2)$$

$$patched(v) = 0.5^{v.patched} \quad (3)$$

$$exploited(v) = 1.25^{v.exploited} \quad (4)$$

Figure 3 displays our score and CVSS for three example vulnerabilities: (a) NE is a vulnerability that is new and has no patch yet, but an exploit was made available a few days after publication. Our score starts by decaying slowly but then there is a jump on severity when the exploit is published; (b) NPE illustrates a vulnerability that has an exploit a few days after publishing and then a patch is also created. First, the score is raised once the exploit starts to be distributed, next it decreases three days later after the patch is released, and then continues to decay over time; and (c) OP represents the best scenario, where the vulnerability is *old* and a patch was eventually distributed (and no exploit is available). Here, the severity decreases once there is a patch, losing its relevance over time from a security perspective.

4.3 Measuring Configurations Risk

The third step of our method is to calculate the *risk of a set of replicas* according to the score of the common weaknesses. The risk associated with a `CONFIG` with n replicas is given by Equation 5. It sums up the score (Equation 1) of the vulnerabilities that would allow an attack to compromise simultaneously a pair of replicas $r_i, r_j \in$ `CONFIG`. More precisely, the vulnerabilities in $\mathcal{V}(r_i, r_j)$ aggregate: (i) the vulnerabilities that affect the software running in both replicas as listed in NVD (and other OSINT sites); and (ii) groups

of vulnerabilities that are placed in the same cluster and affect each replica in the pair (as explained in Section 4.1).

$$risk(\texttt{CONFIG}) = \sum_{r_i, r_j \in \texttt{CONFIG}, i \neq j} \; \sum_{v \in \mathcal{V}(r_i, r_j)} score(v) \quad (5)$$

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

4.4 Selecting Configurations

The final step of the method is to assess the risk of a deployed configuration and, if needed, replace replicas according to our metric evaluation.

At a high-level, Lazarus executes the following procedure. If the CONFIG risk exceeds a predefined *threshold*, a mechanism is triggered to replace replicas. When this happens, the algorithm randomly picks a new replica from the available candidates (POOL) and use it to replace one of the replicas in CONFIG to minimize its associated risk. Thus, maliciously inspecting the POOL is not enough to guess the next best configuration. In addition, it removes the replica that increases the overall risk and set it aside (i.e., in QUARANTINE) to impede its re-selection. There, the replicas wait for patches before they re-join POOL and become ready to be chosen again. Moreover, the algorithm ensures that the running replicas will eventually be replaced, despite their overall score.

Algorithm 1 details this procedure. Function *Monitor()* (line 5) is called on each monitoring round (e.g., at midnight every day) to evaluate the current configuration. If the risk of CONFIG is greater or equal than a certain *threshold* (line 6), the algorithm will assess which replica should be replaced. First, it initializes two variables, the candidates list (line 7) and all possible combinations of $n-1$ out of n elements of CONFIG (line 8). Then, each element r in POOL (line 9) is tested as a potential substitute, i.e., as the n^{th} element that would complete each of the combinations COMB with $n-1$ replicas (line 10). Next, we define a CONFIG' as COMB plus r (line 11). The risk of CONFIG' is calculated (line 12) and if it is below the *threshold* (line 13), then CONFIG' is added to a list of candidate configurations (line 14). At the end of these nested loops, we have a list with all the possible combinations of CONFIG and POOL together with their risk. Then, the function rand is used to randomly selects a configuration from the list of candidates (line 15). Then the algorithm updates all the sets (line 16) using the function updateSets (lines 38-42). To decide which element needs to be removed from CONFIG it makes the difference between the current CONFIG and RAND_CONFIG (line 39) and the contrary to select the element to join the CONFIG (line 40). Then, toRemove is added to QUARANTINE (line 41), removed from CONFIG and the new element is added to CONFIG (line 42).

If the risk of CONFIG is lower than the *threshold* (line 17), the algorithm assesses if a reconfiguration is needed. In this scenario, we start by initializing the variable toRemove as empty (line 18) and maxScore with the value of the CVSS score rating HIGH (line 19). For each element r of CONFIG

Algorithm 1: Replica Set Reconfiguration

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

(line 20) the algorithm calculates the average score of the vulnerabilities affecting r using Equation 1 (line 21). Then, if the average vulnerability score is equal to or greater than HIGH (line 22), the variables toRemove and maxScore are set to r and avgScore, respectively (lines 23 and 24). At the end of the loop, the algorithm knows which is the element r from CONFIG that has vulnerabilities with a highest average score. If toRemove is empty, the algorithm proceeds to line 34. Otherwise, it selects a new replica (line 25). First, the candidate_list is set as empty (line 26). Next, for each element r' in POOL (line 27) the algorithm tests a CONFIG' with r' instead of toRemove (line 28). Then, the score for this configuration is calculated (line 29) and if it is lower than *threshold* (line 30), it is stored in the list of candidates together with CONFIG' (line 31). After that, a new configuration is randomly selected and the sets are updated (lines 32 and 33).

Finally, the algorithm checks if some quarantined replica is ready to re-join the selection pool. More precisely, each element in QUARANTINE is checked if it is fully patched. In the affirmative case, the element is removed from QUARANTINE and added to the POOL again (lines 34-37).

Algorithm 1 is subject to two unlikely corner cases where a system administrator may need to intervene: if the POOL runs out of elements or if there is no candidate configuration that keeps the system risk bellow *threshold*. In these cases, reconfigurations will not happen by default, but we propose two actions to continue reconfiguring the system: increase *threshold* or move the elements with fewer unpatched vulnerabilities from QUARANTINE to POOL.

5 Lazarus Implementation

This section details Lazarus components implementation and briefly discusses other aspects of it.

5.1 Centralized Control Plane

Figure 4 shows Lazarus control plane with its four main modules, described below.

❶ ***Data manager.*** A list of software products is provided by the CPE Dictionary [53], which is also used by NVD. Then, an administrator selects all the software that runs on each replica from this list and indicates the time interval (in years) during which data should be obtained from NVD.

The *Data manager* parses the NVD feeds considering only the vulnerabilities that affect the chosen products. The processing is carried out with several threads cooperatively assembling as much data as possible about each vulnerability – a queue is populated with requests pertaining a particular vulnerability, and other threads will look for related data in additional OSINT sources. Typically, the other sources are not as well structured as NVD and therefore we had to develop specialized HTML parsers for them. Currently, the prototype supports eight other sources, namely Exploit DB [65], CVE-details [64], Ubuntu [75], Debian [26], Redhat [61], Solaris [56], FreeBSD [31], and Microsoft [52].

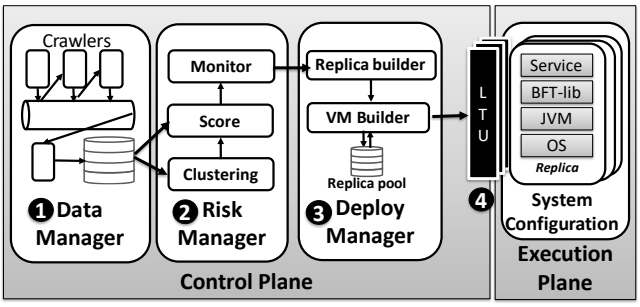


Figure 4. Lazarus architecture.

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

❷ ***Risk manager.*** This component finds out when it is necessary to replace the currently running group of *replicas* and discovers an alternative configuration that decreases the risk. As explained in Section 4.2, 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. We used the open-source machine learning library Weka [62] to build these clusters. In a first phase, 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 [42] is executed to build the clusters, where the number of clusters to be formed is determined by the elbow method [74].

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. This sort of behaviour must be initiated in a synchronous manner from a trusted domain. One way to achieve this, is to employ virtualization, leveraging on the isolation between the untrusted and the trusted domains [28, 58, 71]. Therefore, we developed a replica builder on top of the Vagrant provisioning tool [38], which allows the automatic deployment of ready-to-use OSes and applications on VMs and containers (e.g., VirtualBox, VMware, and Docker), without human intervention. We chose VirtualBox [63] as it supports a bigger set of different guests OSes.

❹ **LTUs.** Each node that hosts a replica has a Vagrant daemon running on its trusted domain. This component is isolated from the internet and communicates only with the Lazarus controller through TLS channels.

5.2 Execution Plane

Despite the amount of relevant research on BFT protocols, only a few open-source libraries exist. In theory, Lazarus can use any of these, as long as they support replica set reconfigurations. In particular, we need the ability to first add a new replica and then remove the old replica to be quarantined. Thus, we use BFT-SMaRt [15], a (reasonably) stable BFT library that supports reconfigurations on the replicas set.

5.3 Alternative Control Plane Design

As in previous works [58, 66], our implemented control plane is centralized. This design was selected for two reasons. First, it allows us to focus more on the main contributions of this work: runtime diversity assessment and performance. Second, it matches the classical scenario where a single organization deploys an *intrusion-tolerant service* [76], and even the managed (permissioned) blockchain services offered by major cloud providers (e.g., [3, 40]). In both scenarios, replica resources are already managed by a single organization that can host a Lazarus control plane.

However, such design makes the controller a single point of compromise and disallow decentralized deployments where replicas are managed by different organizations. Therefore, we outline a design for a BFT Lazarus control plane, by considering a set of *controller replicas* running on top of a BFT library (e.g., BFT-SMaRt [15]).

Essentially, there are four key issues that must be addressed to make this design work. First, LTUs cannot trust a single controller command to reconfigure, so they need to periodically poll the system (as a normal BFT client) to see if there are reconfiguration commands for them. Second, we need to guarantee that all *controller replicas* poll OSINT sources in a (logically) synchronized way to ensure they will use the same vulnerabilities information to decide reconfigurations. This can be done by using a distributed timeout protocol [45]. Third, replicas need to generate secure and distributed random numbers to be used in the Lazarus algorithm. There are efficient protocols for that (e.g., [73]). Finally, it seems difficult to build a solution for "replicated patching" as the same patch applied on the same image results on different (binary) representations, making it difficult to vote for trusted patches. To solve this, one must rely on trusted curator components (e.g., one per organization) for applying patches to the quarantined replicas images and distribute such images to the replicas trusting them.

6 Evaluation of Replica Set Risk

This section evaluates how Lazarus performs on the selection of dependable replica configurations. As discussed in Section 3.2, we focus our experimental evaluation solely on the OS diversity. In particular, we considered 21 OS versions of the following distributions: OpenBSD, FreeBSD, Solaris, Windows, Ubuntu, Debian, Fedora, and Redhat.

These experiments emulate live executions of the system by dividing the collected data into two periods. The goal is to create a knowledge base in a *learning phase* that is used to assess Lazarus' choices during an *execution phase*. (1) The *learning phase* comprises all vulnerabilities between 2014-1-1 and the beginning of the *execution phase*; and (2) the *execution phase* is divided into monthly intervals, from January to August of 2018, allowing for eight independent tests. A run starts on the first day of the *execution phase* and then progresses through each day until the end of the interval. Every day, we check if the executing replica set could be compromised by an attack exploring the vulnerabilities that were published in that month. We take the most pessimistic approach, which is to consider the system as broken if a single vulnerability comes out affecting at least $f + 1$ OSes executing at that time. Therefore, if f of the OSes already has a patch for the tested vulnerability, it is not counted as compromised. In this experiment, we consider $n = 4$, thus the system can tolerate up to $f = 1$ compromised replicas in the set.

Four additional strategies, inspired by previous works, were defined to be compared with Lazarus:

- *Equal:* all the replicas use the same randomly-selected OS during the whole execution. This corresponds to the scenario where most past BFT systems have been implemented and evaluated (e.g., [4, 6, 11, 12, 15, 19, 23, 46, 48, 77]). Here, compromising a replica would mean an opportunity to intrude the remaining ones.
- *Random:* a configuration of n OSes is randomly selected, and then a new OS is randomly picked to replace an existing one each day. This solution represents a system with proactive recovery and diversity, but with no informed strategy for choosing the next `CONFIG`.
- *Common:* this strategy is the straw man solution to prevent the existence of shared vulnerabilities among OSes. This strategy minimizes the number of common vulnerabilities for each set and was introduced in previous vulnerability studies [34, 35].
- *CVSS v3:* this strategy is very similar to ours as it tries different combinations to find the best one that minimizes the sum of CVSS v3 score.

6.1 Diversity vs Vulnerabilities

We evaluate how each strategy can prevent the replicated system from being compromised. Each strategy is analyzed

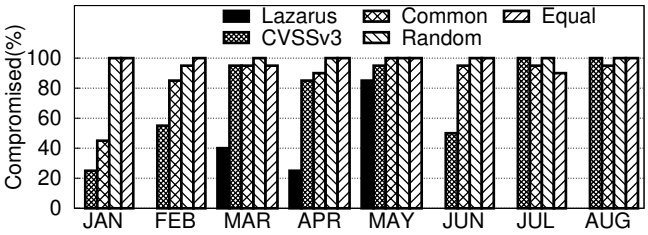


Figure 5. Compromised system runs over eight months.

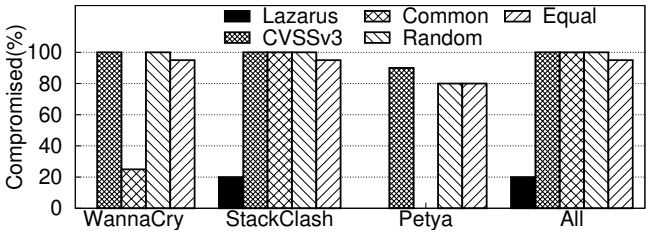


Figure 6. Compromised runs with notable attacks.

over 1000 runs throughout the execution phase in monthly slots. Different runs are initiated with distinct random number generator seeds, resulting in potentially different OS selections over time. On each day, we check if there is a vulnerability affecting more than one replica in the current `CONFIG`, and in the affirmative case the run is stopped.

Results: Figure 5 compares the percentage of compromised runs of all strategies. Each bar represents the percentage of runs that did not terminate successfully (lower is better). Lazarus presents the best results for every month. The *Random* and *Equal* strategies perform worse because eventually, they pick a group of OSes with common vulnerabilities as they do not make decisions with specific criteria. Although some criteria guide the other strategies, most of them present a majority of executions compromised during the experiments. This result provides evidence that Lazarus improves the dependability, reducing the probability that $f + 1$ OSes eventually become compromised and, contrary to intuition, shows that changing OSes every day with no criteria tend to create unsafe configurations.

Nonetheless, the results from May deserve a more careful inspection. We have identified some CVEs that make it very difficult to survive to common vulnerabilities even using the Lazarus strategy. For example, CVE-2018-1125 and CVE-2018-8897 affect a few Ubuntu and Debian releases simultaneously. There are also a set of vulnerabilities affecting several Windows releases (e.g., CVE-2018-8134 and CVE-2018-0959). We also found a vulnerability (CVE-2018-1111) that affects few Fedora releases and one Redhat release.

6.2 Diversity vs Attacks

This experiment evaluates the same strategies when facing notable attacks that appeared in 2017: WannaCry [44], Stackclash [9], and Petya [57]. Each of these attacks potentially exploits several flaws, some of which affecting different OSes. The attacks were selected by searching the security news sites for high impact problems, most of them related to more than one CVE. As some of the CVEs include applications, we added more vulnerabilities to the database for this purpose.

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

Results: Figure 6 shows the percentage of compromised runs for each attack and all attacks put together. Lazarus is the best at handling the various scenarios, with almost no compromised executions. The StackClash is the most destructive attack as it is the one affecting more OSes. Therefore, decisions guided by a criteria that aim to avoid common vulnerabilities may also fail. Nevertheless, the results show that such strategies do improve the resilience to attacks.

7 Performance Evaluation

In this section, we present some experiments conducted to evaluate Lazarus: we run the BFT-SMaRt microbenchmarks in our virtualized environment (1) to compare the performance of 17 OS versions[4] against a homogeneous bare metal setup and (2) to measure the performance of specific diverse setups; (3) we analyze the performance of the system during Lazarus-managed reconfigurations; and (4) we evaluate three BFT services running in our infrastructure.

These experiments were conducted in a cluster of Dell PowerEdge R410 machines, where each one has 32 GB of memory and two quad-core 2.27 GHz Intel Xeon E5520 processor with hyper-threading, i.e., supporting 16 hardware threads on each node. The machines communicate through a gigabit Ethernet network. Each server runs Ubuntu Linux 14.04 LTS (3.13.0-32-generic Kernel) and VirtualBox 5.1.28, for supporting the execution of VMs with different OSes. Additionally, Vagrant 2.0 was used as the provisioning tool to automate the deployment process. We configure BFT-SMaRt 1.1 with four replicas ($f = 1$), one per physical machine.

Table 2 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 and the amount of memory *supported* by VirtualBox with that particular OS. Given these limitations, we setup our environment to establish a fair baseline by configuring an *homogeneous bare metal environment* (BM) that uses only four cores of the physical machine.

[4]Here we have less OSes than in Section 6 experiments because not all of them were supported by Vagrant.

ID	Name	#Cores	Memory
UB14	Ubuntu 14.04	4	15GB
UB16	Ubuntu 16.04	4	15GB
UB17	Ubuntu 17.04	4	15GB
OS42	OpenSuse 42.1	4	15GB
FE24	Fedora 24	4	15GB
FE25	Fedora 25	4	15GB
FE26	Fedora 26	4	15GB
DE7	Debian 7	4	15GB
DE8	Debian 8	4	15GB
W10	Windows 10	4	1GB
WS12	Win. Server 2012	4	1GB
FB10	FreeBSD 10	4	1GB
FB11	FreeBSD 11	4	1GB
SO10	Solaris 10	1	1GB
SO11	Solaris 11	1	1GB
OB60	OpenBSD 6.0	1	1GB
OB61	OpenBSD 6.1	1	1GB

Table 2. The different OSes used in the experiments and the configurations of their VMs.

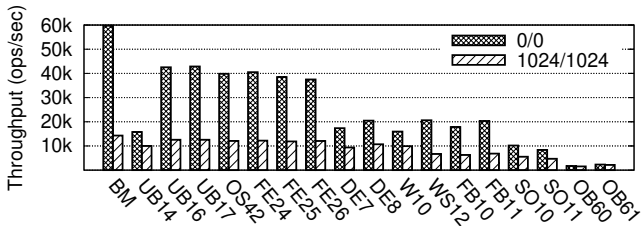


Figure 7. Microbenchmark results under 0/0 and 1024/1024 workloads for homogeneous configurations.

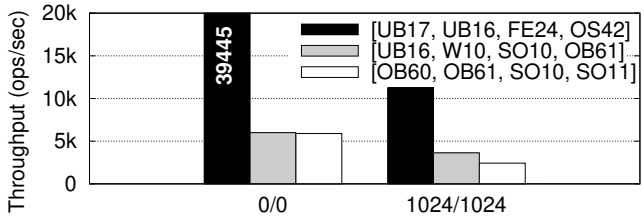


Figure 8. Microbenchmark results under 0/0 and 1024/1024 workloads for three diverse configurations.

7.1 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., [6, 11, 12, 15, 19, 46, 48]). 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 closed-loop clients spread on seven machines to create the workload.

Results: Figure 7 shows the throughput of each OS running the benchmark for both loads. To establish a baseline, we executed the benchmark in our BM Ubuntu.

The results show 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 approx. 66% and 75% of the BM throughput for the 0/0 and 1024/1024 workloads, respectively. For Debian, Windows, and FreeBSD, the results are much worse for 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 limitations of our virtualization platform for supporting different OSes strongly constrains the performance of specific OSes in our testbed.

7.2 Diverse Replicas Throughput

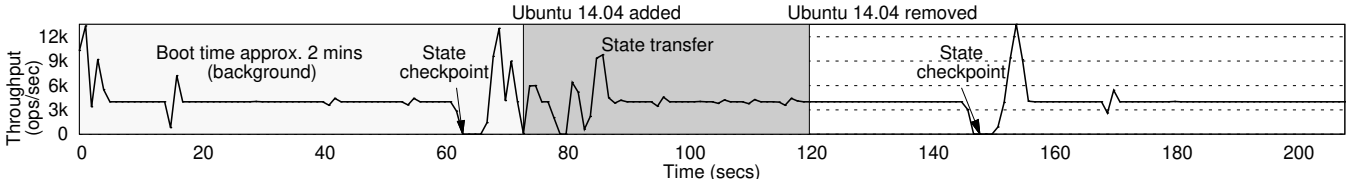
In this experiment, we evaluate three diverse sets of four BFT-SMaRt replicas, one with the fastest OSes (UB17, UB16, FE24, and OS42), another with one replica of each OS family (UB16, W10, SO10, and OB61), and a last one with the slowest OSes (OB60, OB61, SO10, and SO11). The idea is to set upper and lower bounds on our configurations throughput.

Results: Figure 8 shows that throughput drops from 39k to 6k for the 0/0 workload (65% and 10% of the BM performance), and from 11.5k to 2.5k for the 1024/1024 workload (82% and 18% of the BM performance). When comparing these two sets with the non-diverse sets of Figure 7, 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 makes progress in the speed of the 3^{rd} fastest replica (a Solaris VM) as its Byzantine quorum needs three replicas for ordering requests with $f = 1$. These results show that running Lazarus with current virtualization technology results in a significant performance variation, depending on the configurations selected by the system.

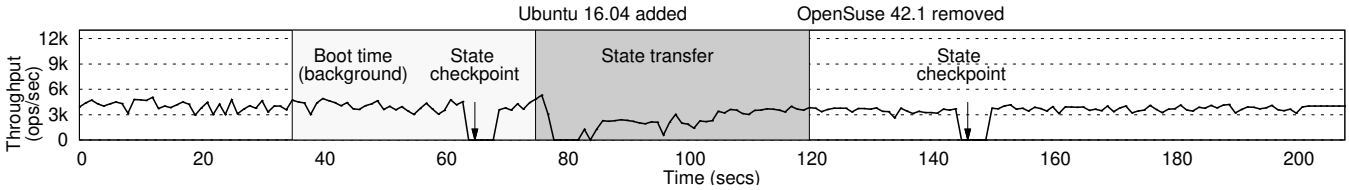
7.3 Performance During Reconfiguration

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

In this experiment, we compare the replicas reconfiguration in the homogeneous BM environment with a Lazarus setup (the initial OS configuration is DE8, OS42, FE26, and



(a) Bare metal (homogeneous).



(b) Lazarus (diverse).

Figure 9. Effect of a reconfiguration on the KVS performance under a 50/50-YCSB workload (1kB values and 500MB state).

SO11) to measure the overhead of Lazarus-induced reconfigurations. We execute this experiment with an in-memory Key-Value Store (KVS) service that comes with the BFT-SMaRT library. The experiment was conducted with a YCSB [24] workload of 50% of reads and 50% of writes, with values of 1024 bytes associated with small numeric keys. Due to the virtualization limitations, shown in the previous experiments, we used a workload that best fits both environments. Then we setup the benchmark to send approximately 4000 operations per second over a state of 500MBs. We selected a time frame of 200 secs to show the performance during a reconfiguration (i.e., the addition of a new replica before the removal of the old one) using the BFT-SMaRt reconfiguration protocol [15].

Results: Figure 9 shows the KVS throughput during reconfigurations when running on bare metal and on Lazarus, as measured with the YCSB benchmark. In both scenarios, there are two types of performance drops. The first one, due to state checkpoints, comprises the period in which BFT-SMaRt replicas trim their operation logs and save a snapshot of their state. The second one, due to state transfers, happens when the newly added replica fetches the state from other replicas. The performance issues caused by these events were already identified, and mitigated in a previous work [14].[5] Both experiments show similar checkpoint duration. Although Lazarus is slower in transferring state, it is faster to execute the log and reach the other replicas state due to its virtualized environment. The BM booting time for Ubuntu 14.04 was more than 2 mins, while, in Lazarus, Ubuntu 16.04 boots in 40 secs.

[5]We employ the standard checkpoint and state transfer protocols of BFT-SMaRt and not the one introduced in [14] as the developers of the system pointed them as more stable.

7.4 Application Benchmarks

Our last set of experiments aims to measure the throughput of three existing BFT services built on top of BFT-SMaRt when running in Lazarus:

- *KVS* is the same application employed in Section 7.3. 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 [12, 48]). In this evaluation, we employ the YCSB 50/50 read/write workload with values of 4k bytes.
- *SieveQ* [36] is a BFT message queue service that can also be used as an application-level firewall. 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 that all the layers were running on the same four physical machines as the diverse BFT-SMaRt replicas (under different OSes). The workload imposed on the system is composed of messages of 1k bytes.
- *BFT ordering for Hyperledger Fabric* [70] is the first BFT ordering service for Fabric [5]. Fabric is an extensible blockchain platform designed for business applications. 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 7.2, we run the applications on the fastest and slowest diverse replica sets and compare them with the results obtained in bare metal.

Results: Figure 10 shows the peak sustained throughput of the applications. The KVS results show a throughput of

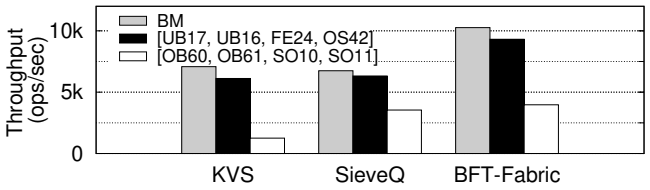


Figure 10. Different BFT applications running in the bare metal, fastest, and slowest OS configurations.

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 BM.

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

The Fabric ordering service results show that running the application on Lazarus virtualization infrastructure lead to 91% (fastest set) and 39% (slowest set) of the throughput achieved on bare metal. Nonetheless, these results do not represent a significant bottleneck considering the performance of the Fabric ordering service [5, 70].

8 Related Work

In the last two decades there were a number of BFT systems/protocols deemed "practical" (e.g., [6, 11, 12, 19, 46, 77]). Most of these, either ignore the issue of fault independence or simply assume it will be solved in some way (e.g., N-version programming [22]). In addition, only a few works address diversity on BFT using automatic techniques [58, 66], which are limited [17, 68]. Moreover, only two works made efforts for evaluating diversity on practical replicated systems.

BASE [21] explores opportunistic OTS diversity in BFT replication. The system provides an abstraction layer for running diverse service codebases on top of the PBFT library [19]. It deals with different representations of the replica 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 NFS implementations: Linux, OpenBSD, Solaris, and FreeBSD. The results, from 16 years ago, show the same performance impacts of diversity. However, BASE does not address the selection of replicas or the reconfiguration of a replica set.

Phoenix [43] is a cooperative backup system build to avoid common weaknesses that can compromise multiple backup nodes. Contrary to Lazarus, which uses vulnerability data, their configurations are build by creating cores with non-overlapping OSes and services. These are discovered via network scans, and then, different services that use the same port are (considered) affected by the same vulnerability. Moreover, there is no mechanism to evolve the configurations over time. Nevertheless, Phoenix mitigates a number of weaknesses that could compromise an entire system.

9 Discussion & Conclusions

Lazarus is a first response to the long-standing open problem of managing the diversity of a BFT system to make it resilient to malicious adversaries. We focused mainly on two issues: how to select the best replicas' set to run given the current threat landscape, and what is the performance overhead of running a diverse BFT system in practice.

One of the main limitations of Lazarus is its performance overhead. We ran the replicas on top of VirtualBox, allowing Lazarus to support 17 OS versions. However, it had a significant impact on the performance as it limits the CPU/memory resources available. Nevertheless, the results also show that less resource-limited OSes have a similar performance. Thus, the main overhead is due to virtualization and not (so much) to diversity.

This paper opens many avenues for future work. First, the current Lazarus prototype can be improved by implementing a bare metal LTU based on Razor [59] and by implementing the BFT controller outlined in Section 5.3. This would enable the integration of Lazarus to decentralized Hyperledger Fabric deployments, for instance. Second, Lazarus replica set reconfiguration can be extended to monitor other OSINT sources (e.g., IP black lists, Twitter) to obtain additional timely information about threats [2, 49, 67]. Similarly, Lazarus can be modified to additionally use the outputs of IDSes and other network sensors to assess the BFT system behavior and trigger replica reconfigurations in case of need. Finally, the performance variance of diverse BFT replication could be alleviated if our heuristics took the performance characteristics of the replicas into consideration when selecting a next configuration. The challenge here would be to balance the lower risk score with the higher expected performance. Alternatively, protocols that consider the heterogeneity of replicas could be employed by the BFT system. For example, the leader could be allocated in the fastest replica, or weighted replication can be employed to give more voting power to faster replicas [13, 69].

Acknowledgements

We thank the anonymous reviewers, and our shepherd, Alberto Montresor, for the comments to improve the paper. This work was supported by FCT through the Abyss project (PTDC/EEI-SCR/1741/2041), the LASIGE Research Unit (UID/CEC/00408/2019), and by an individual doctoral scholarship (SFRH/BD/84375/201), and by the European Commission through the H2020 DiSIEM project (700692).

References

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

[2] F. Alves, A. Bettini, P. M. Ferreira, and A. Bessani. 2019. Processing Tweets for Cybersecurity Threat Awareness. *CoRR* abs/1904.02072 (April 2019). http://arxiv.org/abs/1904.02072

[3] Amazon. 2019. Amazon Managed Blockchain. https://aws.amazon.com/managed-blockchain/.

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

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

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

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

[8] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (Jan. 2004), 11–33.

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

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

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

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

[13] C. Berger, H. P. Reiser, J. Sousa, and A. Bessani. 2019. Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*.

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

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

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

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

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

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

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

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

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

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

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

[25] CVSSv3. 2019. Common Vulnerability Scoring System. https://www.first.org/cvss/specification-document.

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

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

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

[29] Ethernodes.org. 2019. The Ethereum Nodes Explorer. https://www.ethernodes.org.

[30] J. S. Fraga and D. Powell. 1985. A Fault- and Intrusion-tolerant File System. In *Proceedings of the 3rd IFIP Conference on Computer Security*.

[31] FreeBSD. 2019. FreeBSD Advisories. https://www.freebsd.org/security/advisories.html.

[32] B. Freeman. 2015. 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.

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

[34] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. 2011. OS Diversity for Intrusion Tolerance: Myth or Reality?. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

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

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

[37] I. Gashi, P. Popov, and L. Strigini. 2007. Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers. *IEEE Transactions on Dependable and Secure Computing* 4, 4 (Oct 2007), 280–294.

[38] HashiCorp. 2019. Vagrant. https://www.vagrantup.com/.

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

[40] IBM. 2019. IBM Blockchain. https://www.ibm.com/blockchain/platform.

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

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

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

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

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

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

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

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

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

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

[51] F. Massacci and V. Nguyen. 2010. 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*.

[52] Microsoft. 2017. Microsoft Security Advisories and Bulletins. https://technet.microsoft.com/en-us/library/security/.

[53] Mitre. 2019. Common Platform Enumeration. http://cpe.mitre.org/.

[54] Mitre. 2019. CVE Terminology. http://cve.mitre.org/about/terminology.html.

[55] National Institute of Standards and Technology. 2019. National Vulnerability Database. http://nvd.nist.gov/.

[56] Oracle. 2019. Map of CVE to Advisory/Alert. https://www.oracle.com/technetwork/topics/security/public-vuln-to-advisory-mapping-093627.html.

[57] D. Palmer. 2017. Petya ransomware attack: What it is, and why this is happening again. https://www.zdnet.com/article/petya-ransomware-attack-what-it-is-and-why-this-is-happening-again/.

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

[59] Puppet. 2019. How Razor Works. https://puppet.com/docs/pe/2019.1/how_razor_works.html.

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

[61] RedHat. 2019. RedHat CVE Database. https://access.redhat.com/security/cve/.

[62] Machine Learning Group at the University of Waikato. 2019. WEKA. http://www.cs.waikato.ac.nz/ml/weka/.

[63] Oracle. 2019. VirtualBox. https://www.virtualbox.org/.

[64] CVE Details. 2019. The Ultimate Security Vulnerability Datasource. http://www.cvedetails.com/.

[65] Exploit Database. 2019. Offensive SecurityâĂŹs Exploit Database Archive. https://www.exploit-db.com/.

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

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

[68] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. 2013. 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*.

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

[70] J. Sousa, A. Bessani, and M. Vukolić. 2018. 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*.

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

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

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

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

[75] Ubuntu. 2019. Ubuntu Security Notices. https://usn.ubuntu.com/usn/.

[76] P. Veríssimo, N. Neves, and M. Correia. 2003. Intrusion-Tolerant Architectures: Concepts and Design. In *Architecting Dependable Systems*. Springer, Chapter 1, 3–36.

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

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

[79] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras. 2018. From Patching Delays to Infection Symptoms: Using Risk Profiles for an Early Discovery of Vulnerabilities Exploited in the Wild. In *Proceedings of the USENIX Security Symposium*.

[80] H. Xue, N. Dautenhahn, and S. King. 2012. Using Replicated Execution for a More Secure and Reliable Web Browser.. In *Proceedings of the Network and Distributed System Security Symposium*.

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