A Reliability Benchmarking Method for Linux

Zhangjun Lu, Wei Zhang, Hao Xu, and Jianhui Jiang*
School of Software Engineering, Tongji University, Shanghai, China
2131482@tongji.edu.cn, 1910134@tongji.edu.cn, 2131483@tongji.edu.cn, jhjiang@tongji.edu.cn
*corresponding author

Abstract—The main challenges faced in conducting reliability assessment of highly reliable computer products by using traditional methods are high cost, long cycle, and low automation level. Although the reliability model based on small samples can acquire reliability attributes, the time expenditure remains significant. The accelerated life testing method is still in the exploratory stage due to the complexity of failure mechanisms in highly reliable computer products. This paper presents an implementation scheme of dependability benchmarking for reliability assessment of Linux operating system. The benchmarking results obtained can be used to compare the differences in reliability of different Linux distributions. The occurrence process of Linux kernel faults is modeled as Poisson process, and the timing of fault injection is determined according to the rule of exponential distribution of fault occurrence interval. The faults composing the fault injection sequence are derived from a validated fault mode library, in which the proportion of the number of fault modes of different kernel functional modules is used to characterize the spatial distribution of faults in actual situations. The selected stable operating phase software reliability model is an exponentially distributed homogeneous Poisson process model. It can produce reference values for the main reliability attributes of Linux, such as failure rate, Mean Time to Failure ($MTTF$), and reliability function. We also develop a tool prototype based on the proposed implementation scheme and use it for reliability assessment of CentOS Stream 8, AnolisOS-8.4-GA, openSUSE 15.3, Ubuntu 20.04.3, Fedora 35 and openEuler-20.03-LTS-SP3 Linux distributions released during the same period.

Keywords–Linux; reliability assessment; dependability benchmarking; software fault injection

1. Introduction

Linux is widely favored for its open-source, free, stable, and secure characteristics, dominating not only in servers but also in embedded systems, supercomputers, cloud computing platforms, and other fields [1].

Linux has spawned numerous distributions, each with its unique development trajectory and community support. Well-known distributions include Ubuntu and CentOS, which is built from the source code of Red Hat Enterprise Linux (RHEL). Additionally, there are emerging distributions such as openEuler and AnolisOS in recent years.

However, as the scale of Linux expands, functionalities become more complex, and application scenarios diversify, users increasingly demand higher reliability, security, and other performance aspects from Linux [2]. This paper focuses on how to effectively assess the reliability of Linux.

Traditional system reliability assessment methods rely on failure data collected over a while, such as failure times and failure intervals. However, for complex software, methods based on on-site failure data collection typically take long evaluation cycles. Meanwhile, the method of reliability testing, due to the low automation level in the test case generation process, are also time-consuming and labor-intensive. Therefore, the concept of dependability benchmarking based on fault injection has been proposed [3]. The core idea is to introduce faults into the target system according to pre-selected fault models through certain strategies, collect behavioral data of the system under the injected faults, conduct reliability analysis, and obtain measurable attributes, such as fault coverage rate, error latency, error rate, etc. The general principles of dependability benchmarking defined by DBench [4] and ISO/IEC 25045 [5] have been specifically applied to several different domains. The most notable benchmark tests target operating systems (including UNIX and Windows [6], [7]), DBMS [8], embedded systems [9], cloud services and infrastructure [10], among others. These studies define methods for evaluating reliability-related attributes of specific target systems, such as throughput, response time, and error rates, as well as operating system restart time and fault severity.

To reduce costs while obtaining the main reliability attributes of computer products, research efforts have mainly focused on three areas. Firstly, models for software reliability assessment under small-sample failure data conditions have been developed [11]. Although this approach reduces the demand for the number of samples, obtaining small-sample failure data is still time-consuming and labor-intensive for high-reliability systems. Secondly, more failure data is obtained by conducting accelerated life tests [12]. This method draws on the principle of hardware reliability assessment based on accelerated life test, but is limited by our understanding of the failure mechanism of computer products. Moreover, it is lack of explainability, and is still in the exploratory stage of research. The third one is to simulate the real fault situation of computer products by designing fault injection mechanism and strategy. For example, factors such as fault type distribution and fault spatial distribution of the tested object are considered in [13], but key factors such as fault time distribution still depend on the hypothesis.

This paper presents an implementation scheme of dependability benchmarking for reliability assessment of Linux operating system. The benchmarking results obtained can be used to compare the differences in reliability of different Linux distributions. By simulating the actual fault occurrence, interface errors are injected into the Linux kernel to obtain the failure data with statistical significance. Subsequently, we input the failure data into an appropriate software reliability model to derive reference values of the main reliability attributes for different distributions of Linux.

The main contributions of this paper are as follows:

- We analyze the temporal distribution of real Linux faults and establish a model of the occurrence process of Linux kernel faults using the Homogeneous Poisson Process (HPP) model. Analysis of fault reports from the Kernel community revealed that the occurrence intervals of Linux kernel faults follow an exponential distribution.
- We present an implementation scheme of dependability benchmarking for reliability assessment of different Linux distributions. The temporal and spatial distribution of real faults are used to guide fault injection experiments. Each fault injection experiment generates a statistically significant set of failure data. Inputting these failure data into a software reliability model can produce reference values for the main reliability attributes of Linux, such as failure rate, Mean Time to Failure ($MTTF$), and reliability function.
- We develop a tool prototype based on our scheme, and used it to compare the reliability of six Linux distributions released during the same period, i.e. CentOS Stream 8, AnolisOS-8.4-GA, openSUSE 15.3, Ubuntu 20.04.3, Fedora 35 and openEuler-20.03-LTS-SP3. The experimental results show that AnolisOS, openEuler, openSUSE, Ubuntu and Fedora have similar stability and reliability, among which openSUSE is the most stable and reliable, while CentOS has relatively poor stability and reliability.

2. Related Work

Currently, the main challenges faced in evaluating high-reliability computer products using traditional reliability assessment methods include high evaluation costs, long evaluation cycles, and low automation levels. To address this issue, researchers have proposed the use of dependability benchmarking based on fault injection [3]. An earlier research work was the DBench project [4]. Subsequently, many studies have defined reliability benchmarks for different types of systems (e.g., OLTP systems, embedded systems) [14]. These studies define methods for evaluating specific target systems, such as throughput, response time, error rate, as well as operating system restart time, and fault severity. For example, [8] extends the TPC-C standard performance benchmark [15], specifying the metrics and steps required to evaluate the performance and reliability characteristics of OLTP systems and compares two different versions of the Oracle transaction engine running on two different operating systems. [10] proposes a reliability benchmark to support NFV providers in making informed decisions, and determining which virtualization, management, and application-level solutions can achieve optimal reliability. For operating systems, Duraes et al. [6] utilized software faults within device drivers as fault payloads to conduct reliability benchmark testing on three Commercial Off-The-Shelf (COTS) operating systems (Windows NT4, Windows 2000, and Windows XP). They ranked the target operating systems based on the severity of their fault patterns. Kalakech et al. [7], on the other hand, meticulously considered the role of workloads. They used actual workloads instead of synthetic test drivers and conducted reliability benchmark testing on Windows NT, Windows 2000, and Windows XP in scenarios where faults existed in user applications. They compared the fault patterns and performance of these systems, including response times and reboot times. Cotroneo et al. [16] performed fault injection on Android, categorizing the impacts of system-level faults into *Crash, ANR (Application Not Responding), Fatal, and No Failure*, to assess the impact of Android faults on user experience quality.

However, so far, the dependability benchmarking mainly provide directly measurable attributes such as fault coverage rate, error latency, error rate, etc., making it difficult to provide comprehensive metrics such as reliability, availability, etc. To obtain the main reliability attributes of computer products, in addition to traditional reliability assessment methods, researchers have conducted a series of research works. [11], [17] attempts to reduce the demand for the number of samples by constructing models for software reliability assessment that can accurately work with small-sample failure data, thereby reducing the time and effort required to obtain failure data. However, for high-reliability systems, obtaining small-sample failure data remains time-consuming and labor-intensive. [12] obtains more failure data through accelerated life tests, it treats fault conditions as stress and injects software faults into the runtime support environment of the test object. By constructing different stress levels based on factors such as fault quantity and severity, reliability attributes under normal stress levels are estimated based on failure data under different stress levels. However, due to our limited understanding of the failure mechanisms of computer products, this method has poor interpretability and is still in the exploratory stage of research. [13] considers factors such as the distribution of fault types and fault spatial distribution when designing the fault injection mechanism. However, critical factors such as the temporal distribution of faults still rely on assumptions.

3. the Temporal and Spatial Distribution of Linux Kernel Faults

3.1 Basic Concept

Here are some basic concepts:

Fault/Error refers to the faults/errors in the relevant kernel functions and system calls that constitute the Linux kernel.

Failure refers to operating system level failures caused by faults/errors in the Linux kernel. It includes crashes, unresponsiveness, partial functionality unavailability, and degradation of performance metrics beyond thresholds.

Temporal Distribution of Faults refers to the statistical distribution followed by the occurrence times of faults in the relevant kernel functions and system calls that constitute the Linux kernel.

Spatial Distribution of Faults refers to the proportion of faults in different kernel functional modules of Linux. It is used to simulate the real occurrence probability of faults in the relevant kernel functions and system calls of different kernel functional modules.

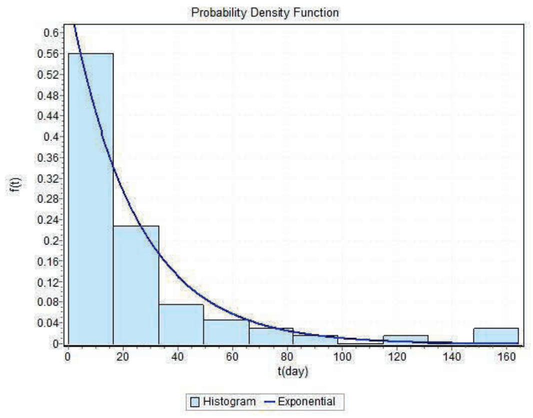
3.2 Modeling the Occurrence Process of Linux Kernel Faults Based on the HPP Model

To demonstrate the feasibility of modeling the occurrence process of internal faults within Linux kernel functional modules by using the Poisson Process, we collected fault reports from various kernel main functional modules (Kernel, FileSystem, MemoryManagement, ProcessManagement, IO/Storage, Networking) in the Linux 4.19, 5.4, 5.10 , and 5.15 version from the official release to March 1, 2024. After removing entries with unclear descriptions, user mishandling, or marked as duplicates or unreproducible, 66 fault records were identified for the 4.19 kernel version , 70 for the 5.4 kernel version, 60 for the 5.10 kernel version, and 58 for the 5.15 kernel version. We fitted the distribution of time intervals of these fault records, the results are shown in Figure 1. While the significance level $\alpha = 0.05$, for 4.19 kernel version, the result of K-S test is 0.16042, which is less than the *critical value* equals 0.16443. For the 5.4 kernel version, the result of the K-S test is 0.15748, which is less than the *critical value* equals 0.15975. For the 5.10 kernel version, the result of the K-S test is 0.1039, which is less than the *critical value* equals 0.17231. For the 5.15 kernel version, the result of the K-S test is 0.16696, which is less than the *critical value* equals 0.17519. To sum up, it can be considered that the time intervals of internal faults within Linux kernel functional modules follow an exponential distribution.

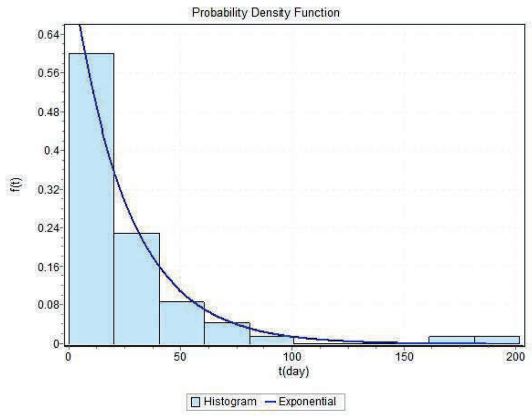
Defects that cause internal failures of kernel functional modules are not fixed after an operating system failure is observed. Therefore, the fault generation process within Linux kernel functional modules exhibits a stationary and independent increment characteristic. Based on this characteristic and the conclusion that the fault intervals follow an exponential distribution, we can employ the HPP model to describe the occurrence process of Linux kernel faults. Within any time interval of length *t*, the occurrence frequency of faults in Linux kernel functional modules can be considered to follow a Poisson distribution with a mean of θt.

$$P\{N(t+q) - N(q) = u\} = e^{-\theta t}\frac{(\theta t)^u}{u!}, u = 0, 1, ... \quad (1)$$

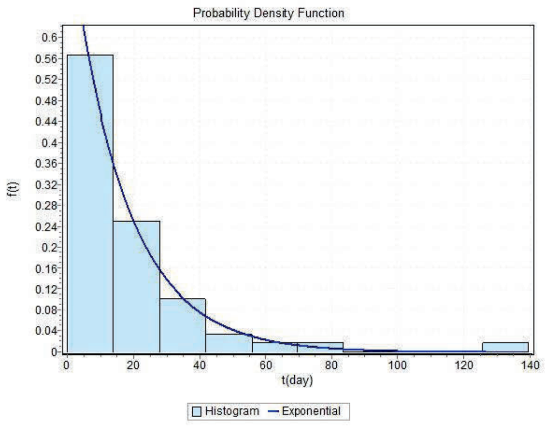
The time interval between two consecutive faults within Linux kernel functional modules follows an exponential distribution



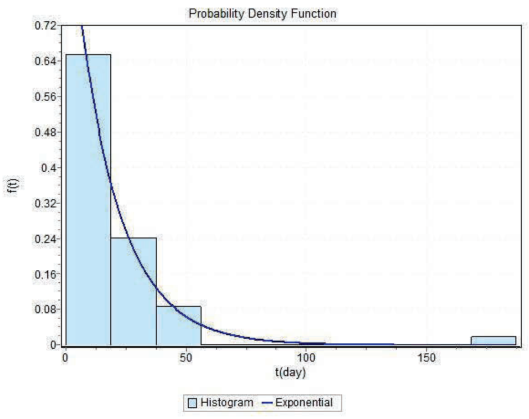
(a) Fault interval of Linux 4.19 follows an exponential distribution



(b) Fault interval of Linux 5.4 follows an exponential distribution



(c) Fault interval of Linux 5.10 follows an exponential distribution



(d) Fault interval of Linux 5.15 follows an exponential distribution

Figure 1. The fitting results of the time interval distribution of Linux faults

with a parameter θ. Here, θ reflects the frequency of faults occurring within the Linux kernel functional modules.

$$P\{X_n > t | X_{n-1} = q\} = e^{-\theta t} \tag{2}$$

3.3 The Proportion of Faults in Different Kernel Functional Modules of Linux

We use the proportion of faults in different kernel functional modules of Linux to describe the spatial distribution of Linux kernel faults. The determination of the proportion of faults in different kernel functional modules of Linux depends on the construction process of the Linux fault mode library [18].

The Linux kernel comprises five major functional modules, i.e. file system (fs), interrupt management (int), process management (pro), memory management (mem), and I/O management (io). The kernel functions are numerous and highly complex, making direct failure analysis of its code time-consuming. Analyzing system calls can help in summarizing these fault modes. We mainly focus on system calls associated with the above modules.

Additionally, we inject Orthogonal Defect Classification (ODC) defects into the Linux kernel code and extract fault modes related to Linux system calls. By comparing them with fault modes obtained through code analysis, it can confirm the correctness of existing fault modes in the Linux kernel fault mode library. Those mismatched fault modes will be added as new fault modes to the fault mode library.

Figure 2 depicts the distribution of fault modes collected in the Linux kernel fault mode library across different kernel functional modules. Assuming equal probabilities for all fault modes, these proportions indicate the likelihood of a single fault occurring within each kernel functional module, thus partially reflecting the spatial distribution characteristics of Linux kernel faults.

4. Linux Kernel Fault Injection Sequence Generation

To obtain failure data with statistical significance, it is imperative to develop a sound fault injection strategy that simulates real-world failure scenarios. This entails creating a fault injection sequence that accurately reflects genuine fault incidents, which can then be utilized to steer a fault injection experiment. When formulating the fault injection sequence, it is essential to account for the temporal and spatial distribution of faults, alongside adhering to constraints regarding the number of faults.

4.1 Select the Appropriate Model Parameter θ

The temporal distribution of internal failures occurring in Linux kernel functionality modules depends on the values of parameters n and θ. Here, n represents the number of faults contained in the injected fault sequence, while θ characterizes the frequency of faults in the Linux fault temporal distribution model. The relationship between θ, n, and the time overhead T required for injecting a fault sequence can be expressed as:

$$\theta = n/T \tag{3}$$

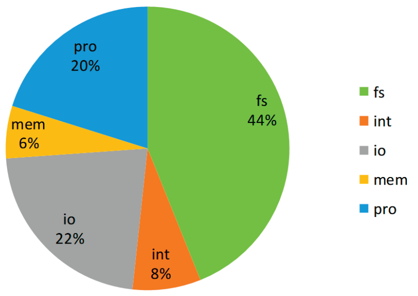


Figure 2. Proportion of faults in Linux kernel modules

According to the analysis of open-source community data, the actual $\theta = 6.8722 \times 10^{-7}$ faults per second. To obtain 100 failure data records, it would require approximately 1684 days of continuous observation experiments on Linux, which leads to a prohibitively high cost. Therefore, it is necessary to select an appropriate θ to expedite the acquisition of failure data.

[19] provides experimental data on fault impact analysis conducted on a PowerEdge R630 equipped with two Intel® Xeon® E5/2650 processors. The results indicate that, for the Linux system, aside from cases where no impact is observed, over 93% of fault impacts manifest shortly after fault injection, with almost all impacts occurring within 50s after injection. Our previous fault impact analysis work also demonstrated that the majority of fault impacts manifest within a short period. To ensure that each fault impact is detected and captured by monitoring tools before the next fault injection, we aim to maximize the time interval between consecutive fault injections in the generated fault injection sequence, with the interval preferably exceeding 50s. Simultaneously, to minimize time overhead, it is necessary to set a maximum time interval. Therefore, we select θ such that the 5^{th} percentile of the interval time is greater than 50 seconds, while the value adjacent to the 95^{th} percentile is chosen as the maximum time interval.

$$P(X \leq 50) = 1 - e^{-50\theta} \leq 0.05 \tag{4}$$

$$P(X > t_{0.95}) = e^{-t_{max}\theta} = 0.05 \tag{5}$$

Based on Equation (4), $\theta \leq 0.001025$, it is suggested to set $\theta = 0.001$. Furthermore, according to Equation (5), $t_{0.95} = 2995$, hence the maximum interval time can be set to 3000 seconds. During the generation process, if an interval time exceeds the maximum interval time threshold, the time points should be regenerated. Considering Equation (3), theoretically, T is around 9×10^4 seconds, but practically, T is determined by the actual time overhead of the generated fault injection sequence.

4.2 Linux Kernel Fault Mode Library and Fault Injection Target

In this paper, we adopt the Linux precise fault injection technique based on kernel functions [18]. To identify potential kernel fault modes stemming from defects in Linux kernel code [20], we utilize a combination of methodologies, including system call anomaly analysis, expert knowledge, user

experience, and historical data from similar systems. These fault modes serve to characterize the potential impacts of Linux kernel interactions with upper-layer applications. By injecting interface errors at the kernel function layer, we simulate these fault modes and construct a fault mode library. Several prior works [21], [22] have discussed representative issues concerning function call interface errors. We have also discussed this issue in [23], By using finite state machine to generate representative code variation errors as a benchmark, and collecting function call interface data to analyze the representativeness of function call interface errors. Experimental results show that 62% of all crashes, hangs, and wrong terminations of SPEC benchmark program occurred with interface errors. We calculated the theoretical lower bound (LB) and upper bound (UB) of the function call interface error frequency. The high LB (46%) and UB (68%) interface corruption rates indicate that function call interface error is a type of major effect of software defects [23].
Figure 3 illustrates the objectives of fault injection in Linux. Interface errors, such as incorrect parameters or erroneous buffer data, are injected at the kernel function layer to precisely simulate potential fault modes in the Linux kernel. Activation of these injected errors can lead to runtime failures in Linux, thereby impacting the performance of the Linux system and potentially causing system crashes.

4.3 The Proportion of Faults in Different Kernel Functional Modules of Linux

We utilize the proportion P of fault modes originating from different kernel functionality modules to simulate the spatial distribution of Linux kernel failures in real-world scenarios. P is defined as: $P = \{p_{fs}, p_{int}, p_{io}, p_{mem}, p_{pro}\}$, where p_{fs}, p_{int}, p_{io}, p_{mem}, p_{pro} represent the proportions of fault modes originating from the file system module, interrupt management module, I/O management module, memory management module, and process management module, respectively. Based on the fault mode library [18], we specify $p_{fs} = 44\%$; $p_{int} = 8\%$; $p_{io} = 22\%$; $p_{mem} = 6\%$; $p_{pro} = 20\%$.

4.4 Constraints on the Number of Faults Contained in a Fault Injection Sequence

The fault injection sequence's fault count constraint refers to the minimum number of faults n_{min} that should be included in the sequence when generating fault injection sequences. As a sample from the fault mode library, the fault injection sequence should have a failure ratio similar to that of the overall population. [24] discusses how to determine the number of faults in a fault injection sequence given the total population size, allowable error, and confidence interval. Assuming equal probabilities of fault occurrences, the actual number of faults n in the injected fault set can be considered to satisfy the following relationship with a given n_{min}, a given error e, total population size NT, the standard deviation z at a given confidence level α, and the characteristic value k:

$$n_{\min} = \frac{NT}{1 + e^2 \times \frac{NT-1}{z^2 \times k \times (1-k)}} \tag{6}$$

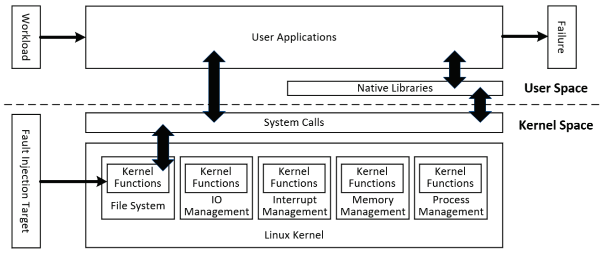


Figure 3. Fault injection target for Linux

$$n = ceiling(n_{\min}, 10) \tag{7}$$

When extracting no fewer than n_{min} fault modes from a fault mode library with a total of NT fault modes to generate a fault injection set, the error of the fault injection set compared to the failure ratio of the fault mode library falls within the given error e at a confidence level α, where n is $nmin$ rounded up to the nearest multiple of 10. Since the value of k is unknown a priori, a conservative approach is to use the value that maximizes the sample size, ensuring that regardless of the actual value of the proportion, the sample size chosen will be sufficient to guarantee the desired confidence level and error range. It has been proven in [24] that in all cases, using $k = 0.5$ is sufficient.
Since the above conclusion is established under the assumption of equal probabilities of fault occurrences, it is necessary to adhere to a uniform distribution during random sampling. Therefore, we conduct stratified random sampling on the fault mode library, determining the number of faults to be extracted from each module based on the proportion of faults originating from different kernel functionality modules and the number of faults in the fault injection set. Subsequently, we perform simple random sampling on the fault subsets of each module using a random number table method.

4.5 Fault Injection Sequence Generation Algorithm

The fault injection case c can be defined as a triple $< ts, m, f >$, representing the injection of a fault mode f into a specific kernel functionality module m of Linux at a certain time ts. Meanwhile, a fault injection sequence is derived from n fault injection cases, sorted by the occurrence time ts, denoted as $C = \{c_1, c_2, c_3, \ldots, c_n | c_i.t < c_{i+1}.t, i \in [1, n-1], i \in N^+\}$, where ts follows the time distribution of fault occurrences, m complies with the proportion of faults originating from different Linux kernel functionality modules, and n satisfies the constraint on the number of faults included in the fault injection sequence.
The following algorithm outlines the generation process of the fault injection sequence with a time complexity of $O(n)$:

- STEP 1: Calculate the minimum number of faults n_{min} required for the fault injection sequence under given e, α, and NT, rounding up to the nearest multiple of 10 as the number of faults n in the fault injection sequence.
- STEP 2: Generate n timestamps ts_i according to the time distribution in ascending order, forming a time sequence $TS = \{ts_1, ts_2, ts_3, \ldots, ts_n | ts_i < ts_{i+1}, i \in [1, n-1], i \in N^+\}$ based on the given fault occurrence frequency θ.

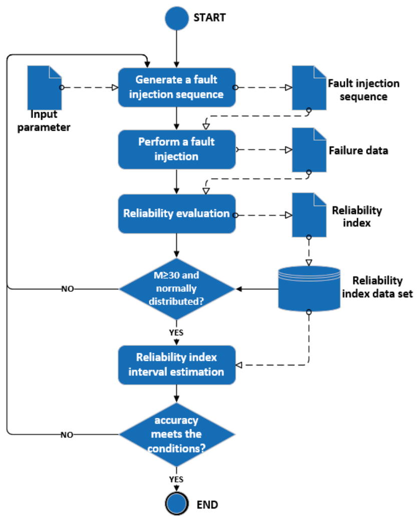

START
Generate a fault injection sequence
Input parameter
Fault injection sequence
Perform a fault injection
Failure data
Reliability evaluation
Reliability index
M≥30 and normally distributed?
NO
YES
Reliability index data set
Reliability index interval estimation
accuracy meets the conditions?
NO
YES
END

Figure 4. Benchmarking process for Linux reliability

- STEP 3: Determine the count of faults $count_{module}$ to be extracted from the fault pattern repository for each module based on the proportion P of faults originating from different modules and the number of faults n in the fault injection sequence. Use the Mersenne Twister (MT) algorithm to generate random numbers for each module, forming a set of fault IDs $FID_{module} = \{fid_1, fid_2, fid_3, \ldots, fid_n\}$. Retrieve fault modes from the fault mode library $F = \{f_1, f_2, f_3, ..., f_{NT}\}$ based on the fault IDs to construct the fault injection set $FS = \{f_1, f_2, f_3, \ldots, f_n | f_i \in F, i \in [1, n], i \in N^+\}$.
- STEP 4: Combine the elements from FS and TS to form a fault injection case c_i, and arrange all fault injection cases in ascending order of time ts to construct the fault injection sequence C.

5. Benchmarking Linux Reliability

5.1 The Evaluation Process for Linux Reliability

The evaluation process for Linux reliability based on fault injection is depicted in Figure 4. The fault injection sequence is generated based on the method introduced in Section 3. The reliability index dataset is used for interval estimation of reliability indicators, while the normality test and accuracy requirement test are primarily aimed at ensuring the accuracy and reliability of the estimates, and these steps will be detailed in Section 5.2.

To simulate real-world workloads, many performance benchmarking tools have been widely used in reliability benchmarking of operating systems. In this study, MySQL-TPC-C [15] and SPEC2006 [25] are selected as the workload sets.

Figure 5 illustrates the implementation process of fault injection experiments.

Many reliability assessment studies on operating systems [2], [16], [26], [27] have discussed system failures and provided detailed classifications. We define Linux system failure to occur under any of the following conditions:

- System crash, all running tasks are terminated, and the system is forcibly shut down.

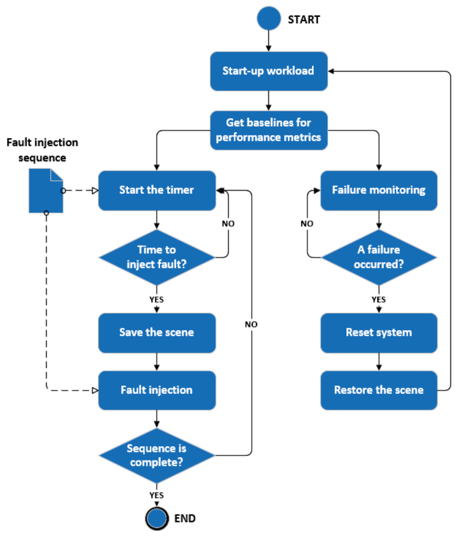


Figure 5. Implementation process of fault injection test

- System not responding, the system is unable to receive user commands or provide specified services.
- Partial unavailability of system functionalities.
- Degradation of specific performance metrics beyond predefined thresholds.

System crash and no responding are among the most discussed failure scenarios in research. Typically, failure detection and data capture are accomplished by setting up crash-handling procedures. We utilize the Kdump to monitor the system's status. When the system crashes or becomes unresponsive, corresponding log files and core dump files are generated [28]. System calls serve as interfaces through which the Linux kernel provides services to users, and their return statuses reflect the availability of system functionalities. We select commonly used Linux system calls and track their return statuses using Kprobe on ftrace [29]. When the return value of a system call is abnormal, it indicates that the functionality provided by that system call is unavailable.

To identify cases where specific performance metrics degrade beyond predefined thresholds, we mainly consider response time and the number of system calls processed per second. We use strace [30] to capture system call details and calculate the number of system calls processed per second. Additionally, the time interval from when a library function is called until it returns is used to measure the operating system's response time.

5.2 Linux Reliability Assessment During Stable Operation

5.2.1 Reliability Evaluation Model

The general principle for selecting a software reliability assessment model [31] is to choose an appropriate model based on the varying trends of reliability exhibited by failure data. We conducted a series of Linux fault injection experiments, the experimental results indicate that the failure data obtained from fault injection experiments exhibit a stable reliability trend, suggesting the applicability of software reliability models with constant failure intensity. Therefore, we select a software reliability model [31] applicable during the stable operation phase. This model follows an exponential distribution known as the HPP model, with the data requirement being complete failure data. Its fundamental assumptions are as follows:

- No modifications are made when defects are found.
- Time between failures follows an exponential distribution.
- Failures are independent of each other.

On the basis of the assumption and the basic theory of reliability engineering, the interval time x_i of the i time failure is a random variable, which follows the exponential distribution with λ (failure rate) as the parameter, and its probability density function is $f(x_i) = \lambda e^{-\lambda x_i}$. Assuming the total number of failures is w, the likelihood function is $L(x_1, \ldots, x_w) = \prod_{i=1}^{w} f(x_i) = \lambda^w e^{-\lambda \sum_{i=1}^{w} x_i}$. The expression of λ obtained by maximum likelihood estimation is shown in equation (8), x_i represents the time interval between failures. In addition, this model can provide reliability metrics including $MTTF = \frac{1}{\lambda}$, and $R(t) = e^{-\lambda t}$.

$$\lambda = \frac{w}{\sum_{i=1}^{w} x_i} \tag{8}$$

5.2.2 Reliability Attribute Estimation

We first focus on estimating the failure rate λ. Let's assume that L fault injection experiments have been conducted, resulting in L sets of failure data. We obtain a sample of failure rates with a size of L and use it to estimate the population mean of the failure rate as the point estimate of λ.

Under the condition of large sample size ($L \geq 30$), according to the Central Limit Theorem, the sample mean of failure rates approximately follows a normal distribution. Utilizing the normal approximation method, we set $\hat{\lambda}$ as the sample mean, with S as the sample standard deviation. The point estimate of the population failure rate is $\hat{\lambda}$, and the interval estimate under a significance level α is given by $[\hat{\lambda} - z_{\frac{\alpha}{2}} \times \frac{S}{\sqrt{L}}, \hat{\lambda} + z_{\frac{\alpha}{2}} \times \frac{S}{\sqrt{L}}]$. Based on the $\hat{\lambda}$, we obtain the point estimate of the $MTTF = \frac{1}{\hat{\lambda}}$, and $R(t) = e^{-\hat{\lambda} t}$.

5.2.3 Normality Test

To meet the large sample condition for estimating reliability attributes, it is stipulated that the L should not be less than 30. Under the condition of $L \geq 30$, a normality test should be conducted on the sample data before estimating reliability attributes in order to satisfy the normal approximation condition.

When the sample size is small, the Shapiro-Wilk (S-W) test method is chosen. The significance level α is set to 0.05. Reliability attributes are estimated only when the $p - value$ of the current sample exceeds α, indicating that the normal approximation condition is met. Otherwise, the number of experimental rounds will increase to ensure an adequate sample size for estimating reliability attributes. This strategy helps ensure that the sample size used in reliability assessment meets statistical requirements, thereby enhancing the accuracy and credibility of the estimates.

5.2.4 Precision Requirement

We have specified the accuracy requirements for estimating the failure rate. The interval estimate of the population failure rate mean at a 95% confidence level is $[\hat{\lambda} - z_{\frac{0.05}{2}} \times \frac{S}{\sqrt{L}}, \hat{\lambda} + z_{\frac{0.05}{2}} \times \frac{S}{\sqrt{L}}]$, with an error margin of $EM = z_{\frac{0.05}{2}} \times \frac{S}{\sqrt{L}}$. It is stipulated that the estimation results are considered to meet the accuracy requirements and can terminate the iterative experiments and output the final reliability indicator estimates only when the error EM obtained from the current sample size is less than or equal to 5% of the current sample mean. Otherwise, the number of experimental rounds will increase to ensure the accuracy and credibility of the estimation.

5.3 Verification

Table 1 shows the configuration information of the verification experiment environment. CentOS Stream 8 with 4.18 kernel version is selected as the experiment system.

5.3.1 Verify the Fault Number Constraint of the Fault Injection Sequence

The constraint on the number of faults affects the selection of the number of faults included in our generated fault injection sequences. Choosing too few failures may result in a lack of completeness and representativeness in the fault injection sequences. Under the conditions of $NT = 2675$, $\alpha = 0.05$, and $e = 0.1$, we obtain $n_{min} = 92$. Here, we set $n = 100$ and conducted 20 experiments, injecting 100 failures each time.

We first conducted Linux failure analysis experiments based on the fault mode library. Table 2 presents the failure analysis results. The "Functional Modules" column identifies different kernel functional modules, while the "Fault Injection Number" indicates the number of fault modes. Failure modes are categorized into five types: *Crash, No Responding, Function Unavailable, Serious Performance Impact, and Mild Performance Impact*. Each data entry represents the proportion of fault modes leading to the corresponding failure mode. From Table 2, it can be observed that the overall proportion of fault modes potentially leading to Linux failures is 38.47%.

Figure 6 illustrates the failure rate of the injection fault set obtained from the 20 fault injection experiments. Considering a 10% error margin, the ranges of the proportions of fault modes causing system failures in the 20 sets of injected fault sets all include 38.47%. This meets the requirement of 95% confidence when calculating constraints. Therefore, it can be considered that the injected fault set inherits the characteristics of the fault mode library, demonstrating a certain degree of completeness and representativeness. This helps ensure that the generated fault injection sequences adequately reflect the possible failure scenarios the system may encounter in real-world scenarios.

5.3.2 Applicability Verification and Assumptions Testing of Reliability Evaluation Model

We conducted 50 fault injection experiments and performed Mann-Kendall (M-K) trend tests on each set of failure data. Under a significance level of $\alpha = 0.05$, we found that 46 sets

Table 1. Configuration of the experimental environment

Hardware Platform		Supporting Tools	
CPU Type	Intel(R) i7	**Workload**	SPEC2006 & MySQL-TPC-C
CPU Clock (GHz)	2.30	**Crash Dump**	Kdump
Cache (MB)	24	**Kernel Tracking**	Ftrace&Kprobe
Memory (GB)	4	**Kernel Debug**	Crash

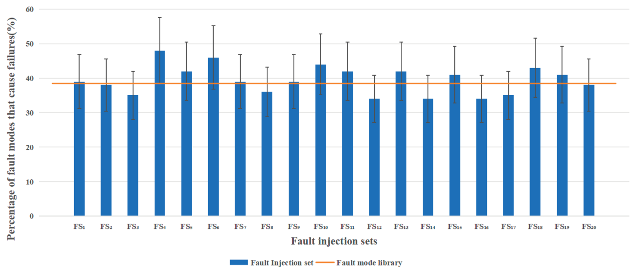


Figure 6. Percentage of fault modes that cause Linux failures

of data exhibited a stable trend in reliability. This suggests that a model with a constant failure intensity should be selected. During the fault injection experiments, it was observed that defects leading to failures were not repaired after the failures occurred, thus satisfying the assumption of not modifying defects when discovered. According to the HPP model, the fault intervals within Linux kernel functional modules follow an exponential distribution. Therefore, it can be inferred that the failure intervals also follow an exponential distribution, satisfying the assumption that each failure corresponds to an exponential lifetime distribution. Since Linux is reset (restarted) after each failure occurrence, it can be considered that failures are independent of each other, fulfilling the assumption of independence between failures. In summary, it is appropriate to use the software reliability assessment model for the stable operational phase to evaluate the reliability of the failure data obtained by this scheme.

6. Linux Reliability Benchmarking Tool Prototype

6.1 The Structure of Tool Prototype

The prototype of the Linux Reliability Benchmarking Tool (*LRBT*) is developed based on the Linux reliability benchmarking method proposed in this paper. The structure is illustrated in Figure 7. It primarily consists of a controller, fault injection sequence generator, fault injector, workload manager, and failure monitor. The fault injection sequence generator implements the fault injection sequence generation algorithm. The workload manager controls the execution of SPEC and TPC workloads. The fault injector utilizes precise fault injection technology based on kernel functions [18]. The failure monitor is implemented according to the failure identification method described in Section 5.1. If the kernel crashes during testing, then it will reboot with the help of the Kdump mechanism, and *LRBT* will continue working with the help of the fault injection context saved before kernel crash.

6.2 Testing of Tool Prototype

6.2.1 Functional, Performance, and Interface Testing

The black-box testing is conducted on the functions of *LRBT* (such as fault injection sequence generation, fault injection,

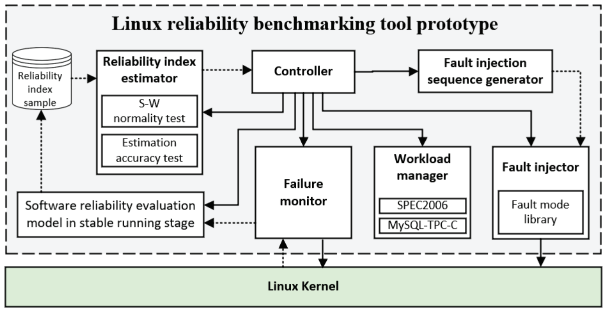


Figure 7. The structure of the *LRBT*

failure monitoring, etc.) by using equivalence partitioning and boundary value analysis methods to design test cases.

The runtime testing is performed on the functionalities in *LRBT* with specified runtime requirements by obtaining the start and end times of executing the targeted functionalities to verify if they can meet the specified runtime requirements of *LRBT*.

A combined approach of black-box and white-box testing is used to test whether each specified interface in *LRBT* correctly transmits the involved data parameters. The testing includes user interfaces (command-line inputs) and internal interfaces (such as interfaces between the fault injector and fault mode library, interfaces between major functional modules and logging modules, etc.). The test results indicate that *LRBT* meets the specified functional, performance, and interface requirements.

6.2.2 Interference Test

We conducted experiments to verify that *LRBT* does not interfere with the results of reliability benchmarking. Firstly, we compared the performance parameters of the Linux system with and without *LRBT* running under the same workload, including CPU utilization and memory utilization. Table 3 presents the average performance parameters obtained from 50 experiments. It showed that *LRBT* consumes only a small amount of system resources and does not affect system performance. Subsequently, fault impact testing was performed under the same workload, with fault injection experiments conducted both with and without *LRBT* running on the target system. Each fault mode in the fault mode library was tested 10 times to eliminate differences in fault impact that may arise from incidental factors. The test results under scenarios with and without *LRBT* running showed identical failure rates for the Linux system, with each fault mode exhibiting the same fault impact.

The above experimental results demonstrate that running *LRBT* during experiments does not interfere with the impact of faults, thereby validating the anti-interference of *LRBT*.

6.3 Case Analysis

As an application of *LRBT*, we conducted reliability assessment on six distributions of Linux during the same period, i.e. CentOS Stream 8, AnolisOS-8.4-GA, openSUSE 15.3, Ubuntu 20.04.3, Fedora 35 and openEuler-20.03-LTS-SP3. We compared the measured reliability for these Linux operating systems. The operating environment configuration information is given in Table 1.

Table 2. Failure analysis results of Linux based on fault mode library

Functional Modules	Fault Injection Number	Performance Degradation			Functional Failure	System Failure	
		No Influence	Mild Performance Impact	Serious Performance Impact	Function Unavailable	No Responding	Crash
fs	1175	732(62.31%)	184(15.66%)	32(2.72%)	103(8.77%)	45(4.19%)	79(7.35%)
io	593	401(67.62%)	60(10.12%)	23(3.88%)	57(9.61%)	16(2.70%)	36(6.07%)
int	208	131(62.98%)	13(6.25%)	8(3.85%)	2(0.96%)	18(8.65%)	36(17.31%)
mem	160	75(46.87%)	18(11.25%)	8(5.00%)	9(5.63%)	32(20.00%)	18(11.25%)
proc	539	307(56.96%)	68(12.62%)	15(2.78%)	64(11.87%)	48(8.91%)	37(6.86%)
Total	2675	1646(61.53%)	343(12.82%)	86(3.21%)	235(8.79%)	159(5.94%)	206(7.70%)

Table 3. Performance parameter test results

Test Environment	Average CPU utilization	Average memory utilization
Run only workloads	87.23%	94.77%
Run workloads and LRBT	87.97%	94.93%

6.3.1 Result Analysis

We applied *LRBT* to six Linux distributions separately (simulating parameters using $\theta = 0.001, n = 100$)). Table 4 presents the reliability benchmarking results of six Linux distributions. Based on the accuracy and normality test requirements, 43 fault injection experiments were conducted for CentOS, 33 for AnolisOS, 36 for openEuler, 37 for openSUSE, 32 for Ubuntu, and 38 for Fedora. The variation in the number of experiments required to achieve accuracy across different Linux distributions is primarily due to the randomness in generating fault injection sequences based on the spatial and temporal distribution of faults. The purpose of setting accuracy requirements in reliability benchmarking is to limit the bias resulting from this randomness and ensure the fairness of reliability benchmarking.

The test results show that openSUSE has the lowest failure rate, AnolisOS, openEuler, openSUSE, Ubuntu and Fedora have little difference in failure rate, while CentOS has a higher failure rate. The $MTTF$ reference value for CentOS decreases by 17% and 21%, respectively, compared to other distributions. Under the load that represents common computationally intensive tasks (SPEC2006) and enterprise database applications (MySQL TPC-C), AnolisOS, openEuler, openSUSE, Ubuntu and Fedora can be considered to have similar stability and reliability. openSUSE is the most stable and reliable, while CentOS has relatively poor stability and reliability.

6.3.2 Discussion

The method of obtaining failure data by generating fault injection sequences essentially simulates the temporal and spatial distributions of failures under actual conditions, by adjusting the θ to accelerate the acquisition of failure data. Let the actual failure occurrence frequency be θ_r, then it can be deduced that the acceleration ratio is given by $AR = \frac{\theta}{\theta_r}$. Taking the experimental results of AnolisOS-8.4-GA as an example, with $\theta = 0.001$ faults per second, we obtained a reference failure rate which equals 1.3657 failures per hour. If we use the failure occurrence frequency provided by the Kernel community, then $\theta_r = 6.8722 \times 10^{-7}$ faults per second, and $AR = 1455$. By multiplying the failure data by AR for restoration, we can obtain intuitively plausible estimated values of reliability attributes, i.e. the actual failure rate equals 9.3862×10^{-4} faults per hour, and the actual $MTTF$ equals 1065.4 hours.

While lacking real dataset support prevents us from verifying the correctness of this restoration approach, it is important to emphasize the feasibility of this idea. Compared to the traditional approach of using acceleration equations based on failure data under stress levels in accelerated life testing, this method offers better interpretability and rationality. Moreover, it effectively shortens the reliability assessment cycle for high-availability targets.

Table 4. Reliability benchmarking results

Linux Distributions	Failure Rate(failures/h)	MTTF(h)	Reliability Function
CentOS Stream 8	1.6183	0.6181	$R(t) = 1 - e^{-1.6183t}$
AnolisOS-8.4-GA	1.3657	0.7322	$R(t) = 1 - e^{-1.3657t}$
openEuler-20.03-LTS-SP3	1.3526	0.7392	$R(t) = 1 - e^{-1.3526t}$
openSUSE 15.3	1.3436	0.7442	$R(t) = 1 - e^{-1.3436t}$
Ubuntu 20.04.3	1.3550	0.7381	$R(t) = 1 - e^{-1.3550t}$
Fedora 35	1.3676	0.7311	$R(t) = 1 - e^{-1.3676t}$

7. Conclusion

This paper presented an implementation scheme of dependability benchmarking for reliability assessment of different Linux distributions. It utilizes software fault injection method to obtain failure data and ultimately estimate meaningful reliability metrics. We developed a tool prototype based on the proposed scheme and applied it to six Linux distributions. The results show that AnolisOS, openEuler, openSUSE, Ubuntu and Fedora have similar stability and reliability, among which openSUSE is the most stable and reliable, while CentOS is relatively poor in stability and reliability. Furthermore, we discuss the feasibility of evaluating Linux reliability indicators by restoring failure data.

Acknowledgment

This work was supported in part by the Alibaba Corporation.

References

[1] A. Adekotujo, A. Odumabo, A. Adedokun, and O. Aiyeniko, "A comparative study of operating systems: Case of windows, unix, linux, mac, android and ios," International Journal of Computer Applications, vol. 176. no. 39, 2020, p. 16–23.

[2] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," In Proceedings of the 2003 International Conference on Dependable Systems and Networks, San Francisco, California, USA, June 22 - 25, 2003, pp. 459-468.

[3] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," ACM Computing Surveys, vol. 48, no. 3, 2016, p. 1–55.

[4] DBench project, "DBench Final Report," 2004. [Online]. Available: www.laas.fr/DBench/, Accessed on: Jan. 3, 2024.

[5] I. O. for Standardization, "Systems and Software Quality Requirements and Evaluation," ISO/IEC Standard 25010:2011 ed, Vernier, Geneva, Switzerland: International Organization for Standardization, 2011.

[6] J. Duraes and H. Madeira, "Multidimensional characterization of the impact of faulty drivers on the operating systems behavior," IEICE Transactions on Information and Systems, vol. 86, no. 12, 2003, pp. 2563-2570.

[7] A. Kalakech, K. Kanoun, Y. Crouzet, and J. Arlat, "Benchmarking the dependability of windows nt4, 2000 and xp," In Proceedings of the 2004 International Conference on Dependable Systems and Networks, Florence, Italy, June 28 - July 1, 2004, pp. 681-686.

[8] M. Vieira and H. Madeira, "Benchmarking the dependability of different oltp systems," In Proceedings of the 2003 International Conference on Dependable Systems and Networks, San Francisco, California, USA, June 22 - 25, 2003, pp. 305-310.

[9] J. Duraes and H. Madeira, "Generic faultloads based on software faults for dependability benchmarking," In Proceedings of the 2004 International Conference on Dependable Systems and Networks, Florence, Italy, June 28 - July 1, 2004, pp. 285-294.

[10] D. Cotroneo, L. De Simone, and R. Natella, "NFV-Bench: A dependability benchmark for network function virtualization systems," IEEE Transactions on Network and Service Management, vol. 14, no. 4, 2017, p. 934–948.

[11] J. G. Lou, J. H. Jiang, Z. L. Shen, and Y. L. Jiang, "Software reliability prediction modeling with relevance vector machine," Journal of Computer Research and Development, vol. 50, no. 7, 2013, pp. 1542–1550. (in Chinese)

[12] A. Jin, J. H. Jiang, and J. G. Lou, "Web server assessment based on accelerated life test," Journal of Computer Research and Development, vol. 47, no. suppl, 2010, pp. 229–236. (in Chinese)

[13] J. W. Hu and J. H. Jiang, "Design and implementation of a fault injection mechanism for software reliability evaluation," Journal of Computer-Aided Design & Computer Graphics, vol. 24, no. 6, 2012, pp. 741–751. (in Chinese)

[14] K. Kanoun and L. Spainhower, "Dependability benchmarking for computer systems," Wiley-IEEE Computer Society Press, 2008.

[15] TPC, "TPC," 2023. [Online]. Available: www.tpc.org/, Accessed on: Jan. 3, 2024.

[16] D. Cotroneo, A. K. Iannillo, R. Natella, and S. Rosiello, "Dependability assessment of the android os through fault injection," IEEE Transactions on Reliability, vol. 70, no. 1, 2021, p. 346–361.

[17] J. Guo, X. W. Kong, N. X. Wu, and L. Y. Xie, "Weibull parameter estimation and reliability analysis with small samples based on successive approximation method," Journal of Mechanical Science and Technology, vol. 37, no. 11, 2023, p. 5797–5811.

[18] H. Xu, Y. X. Hu, B. L. Tan, X. H. Shi, Z. J. Lu, W. Zhang, and J. H. Jiang, "Fault injection based failure analysis of CentOS, Anolis OS and OpenEuler," 2022, arXiv preprint arXiv:2210.08728.

[19] J. R. Campos, E. Costa, and M. Vieira, "A dataset of linux failure data for dependability evaluation and improvement," In Proceedings of the 2022 IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, Baltimore, Maryland, USA, June 27 - 30, 2022, pp. 88-95.

[20] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the accuracy of binary-level software fault injection," IEEE Transactions on Dependable and Secure Computing, vol. 15, no. 1, 2018, p. 40–53.

[21] R. Natella, S. Winter, D. Cotroneo, and N. Suri, "Analyzing the effects of bugs on software interfaces," IEEE Transactions on Software Engineering, vol. 46, no. 3, 2020, p. 280–301.

[22] R. Amarnath, S. N. Bhat, P. Munk, and E. Thaden, "A fault injection approach to evaluate soft-error dependability of system calls," In Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops, Memphis, Tennessee, USA, October 15 - 18, 2018, pp. 71-76.

[23] W. Zhang, H. Xu, Z. J. Lu, and J. H. Jiang, "On error representativeness of function call interfaces for C/C++ program," In Proceedings of the 2023 IEEE International Conference on Software Quality, Reliability, and Security, Chiang Mai, Thailand, October 22 - 26, 2023, pp. 719-728.

[24] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," In Proceedings of the 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, France, April 20 - 24, 2009, pp. 502-506.

[25] SPEC, "The SPEC consortium: Members and associates," 2022. [Online]. Available: www.spec.org/consortium/, Accessed on: Dec. 6, 2023.

[26] W.-I. Kao, R. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," IEEE Transactions on Software Engineering, vol. 19, no. 11, 1993, p. 1105–1118.

[27] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina, and P. Rumeau, "Benchmarking the dependability of windows and linux using postmark/spl trade/ workloads," In Proceedings of the 2005 IEEE International Symposium on Software Reliability Engineering, Chicago, Illinois, USA, November 8 - 11, 2005, pp. 10-20.

[28] The kernel development community, "Kdump," 2023. [Online]. Available: docs.kernel.org/admin-guide/kdump/, Accessed on: Dec. 6, 2023.

[29] The kernel development community, "ftrace," 2023. [Online]. Available: www.kernel.org/doc/html/latest/trace/, Accessed on: Dec. 6, 2023.

[30] strace, "strace," 2023. [Online]. Available: strace.io/, Accessed on: Dec. 6, 2023.

[31] Institute of Electrical and Electronics Engineers, "IEEE Recommended Practice on Software Reliability," in IEEE Std 1633-2016 (Revision of IEEE Std 1633-2008), 2017, pp.1-261.