Estimation of Linux Kernel Execution Path Uncertainty for Safety Software Test Coverage

Imanol Allende*, Nicholas Mc Guire†, Jon Perez*, Lisandro G. Monsalve*, Javier Fernández*, Roman Obermaisser‡
*Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), Mondragon, Spain
iallende,jmperez,lgmonsalve,javier.fernandez@ikerlan.es
†OpenTech EDV Research GmbH, Bullendorf, Austria
der.herr@hofr.at
‡University of Siegen, Siegen, Germany
roman.obermaisser@uni-siegen.de

***Abstract*—With the advent of next-generation safety-related systems, different industries face multiple challenges in ensuring the safe operation of these systems according to traditional safety and assurance techniques. The increasing complexity that characterizes these systems hampers the maximum achievable test coverage during system verification and, consequently, it often results in untested behaviors that hinder safety assurance and represent potential risk sources during system operation. In the context of paving the way towards quantifying the risks caused by software malfunction and, hence, towards the safety-compliance of next-generation safety-related systems, this paper studies and provides a method to estimate the probability of Linux kernel execution paths that remain unobserved during the test campaign.**

***Index Terms*—Linux, safety, complex-systems, statistics, uncertainty**

I. Introduction

Next-generation complex safety-critical systems (e.g., autonomous robotic systems) are characterized by the need to integrate heterogeneous complex software applications of different safety criticality and high-computational resource demands supported by multi-core devices and accelerators such as Graphics Processing Units (GPUs). In this context, there is an interest for an open-source safe Operating System (OS) that enables the integration and management of such heterogeneous, complex and high-performance software applications of different safety integrity [1], [3], [15], [16].

Safety-critical systems are defined as systems whose failure could cause human injury (even death) or property / environmental damage. The safety criticality is established with a Safety Integrity Level (SIL) ranging from 1 to 4 (highest) for industrial safety standards (e.g., IEC 61508) and Automotive Safety Integrity Level (ASIL) A to D (highest) for automotive safety standard (ISO 26262). The higher the criticality the lower the permissible probability of dangerous failure, e.g. a SIL4 has associated a probability of dangerous failure in the range of 10^{-8} to 10^{-9} hours of operation (approximately once every 14.155 to 114.155 years).

GNU/Linux is broadly used in non-safety-critical systems, being the leading General Purpose Operating System (GPOS) in different domains [4]: 99% supercomputer market, 90% public cloud workload, 82% smartphone market (Android) and 62% embedded market. Besides, companies nowadays rely on this GPOS for business and mission-critical applications such as banking. Therefore, considering the key role that GNU/Linux is successfully playing in these dependable domains, one can perceive the potential that it could have for safety-critical applications. However, GNU/Linux was not designed for safety-critical systems and whenever such a GPOS is integrated on high-performance multi-core devices that are based on a resource-sharing architecture [2], [15], a highly complex system emerges and, as a result, many of the traditional methods that have been used in the safety domain would need to be revisited (e.g., code test coverage).

The safety certification of a GPOS such as GNU/Linux for its use in these next-generation complex safety-critical systems is a major challenge that in turn needs to address multiple technical challenges beyond the contribution of this paper [1], [2], [16]. With respect to testing, IEC 61508 highly recommends 100% code test coverage for entry points and statements for SIL2 systems and equivalent requirements can be identified in ISO 26262. If this coverage is not achieved, an appropriate explanation should be provided.

If a safety-critical application (e.g., SIL2) is developed, the code test coverage needs to consider all possible kernel execution paths. Nevertheless, the execution traces at the kernel-space are at least dependent on the concurrent execution(s) in the user-space, the device architecture and the system configuration. Thus, during the testing phase it is generally not feasible to guarantee that all possible kernel paths have been covered as some paths might not be possible to be executed for the given applications(s), device architecture and configuration. The complexity of the test coverage analysis increases with systems based on the Linux kernel as they are physically non-deterministic [9], [10]. This means that the execution path at the kernel level does not only depend on the inputs of the application and, hence, the same application with the same inputs may execute a different execution path. This excludes the possibility of forcing the execution of a specific path during testing as it does not only rely on the inputs of the application but on the state of the system.

978-3-9819263-5-4/DATE21/©2021 EDAA

Besides, this state is not generally reproducible due to the great complexity of these systems. For example, an autonomous driving system integrates several applications (e.g., Machine Learning (ML) algorithms, sensor information processing), which run above a middleware (e.g., Robot Operating System (ROS)), on a GPOS (e.g., GNU/Linux) and, finally, on a Commercial Off-The-Shelf (COTS) multi-core device with a shared resources architecture and non-deterministic mechanisms (e.g., branch prediction, Pseudo Least Recently Used (PLRU) cache replacement) [15], being highly-complex (if feasible) to reproduce a specific state of the system.

Therefore, this kernel execution path uncertainty, caused by non-tested paths, jeopardizes the testing phase of the safety-critical application(s). This publication proposes a method for the estimation of execution path uncertainty that provides a probability value for the non-tested (unseen) execution paths that can be used with the As Low As Reasonably Practicable (ALARP) principle to provide the required code test coverage justification or identify the need to increase the testing depth and scope. The method is described and analyzed with a simple case-study, that aims to be a simple and statistically reproducible example for researchers.

The paper is structured with the following sections. Section II summarizes the related work. Section III describes a simple yet reproducible case-study experiment setup. Section IV describes the method and discusses the results obtained with the previously described case-study. Finally, Section V summarizes the conclusions and future work of this research.

II. Related Work

During the last years, projects such as SIL2LinuxMP [16] and Enabling Linux in Safety Applications (ELISA) [1] have examined the viability of certifying a specific configuration of COTS elements including the Linux kernel for a well-specified use-case in a defined mission profile and on a given multi-core device. Thus, addressing the certification only for a fully contextualized Linux kernel.

SIL2LinuxMP project, lead by Open Source Automation Development Lab (OSADL), shows the non-determinism of the Linux kernel execution in different publications [9]–[13]. Mc Guire et al. published the first results related to the inherent non-determinism of the Linux kernel execution while they were searching for jitter sources [9]. This first study concludes that a complex GPOS like GNU/Linux amplifies the intrinsic randomness of modern processors. Taking this first analysis as a basis, Okech et al. show in different articles the execution path non-determinism of the Linux kernel [10]–[13]. In their research they analyze the execution of different system calls by dynamic inspection (*Ftrace* tool), identifying a high number of different execution paths for each call. The authors heuristically categorize the most frequently executed paths as common paths and the rest as non-common paths. These studies reveal the differences in the execution paths of a replicated application execution on two cores. The same application executed at the same time in different cores of the same multi-core device may have different kernel execution paths as any of the rare-paths can be executed. In such a way, Okech et al. conclude that the execution paths at the kernel level are not totally determined by the applications' inputs.

Continuing previous research, Allende et al. study a method that analyzes statistically the Linux kernel test coverage with dynamic data obtained through FTrace [14]. The method estimates statistically the total number of unobserved paths via regression analysis of the observed unique paths over the test-campaign length. While this does result in a reliable estimate of the unobserved path count and, thus, in the coverage-percentage, it is not possible to know the risk that these paths entail since we ignore their execution probability.

The estimation of uncertainty has also been studied in other technological fields. Gal has examined and developed different tools to estimate the uncertainty in deep-learning [5]. Kendall et al. model the uncertainty for autonomous driving computer vision systems [7], [8].

III. Case Study

In order to ease the comprehension and statistical reproducibility of the proposed method, a simple yet representative case-study is proposed and used within the method description (Section IV). The case-study is defined with the following test setup:

- Target system: the research is carried out on a ZynqMP Ultrascale++ consisting of a quad-core ARM Cortex-A53 running the v4.19.75-cip11 Civil Infrastructure Platform (CIP) Super Long-Term Support (SLTS) Linux version.
- Application: The simple case-study application shown in Listing 1 makes six main calls to the Linux kernel (two *open()* calls, a *read()* call, a *write()* call and two *close()* calls).
- System-context: Rather than being executed in a static system-context, this simple application is exposed to a heavy computing load generated with *Hackbench* to simulate a Worst Case Scenario (WCS). This benchmark is a stressing tool widely used by the Linux kernel developers.

Errors are not handled and, for the given setup, all the requested resources are reasonably expected to be available and, thus, the application can be assumed to successfully execute while iterating over possible paths.

IV. Method for Estimation of Execution Path Uncertainty

This section describes the proposed method to estimate the Linux kernel execution path uncertainty during the testing phase of a Linux based safety-critical system, leading to the probability (estimation) of unseen execution traces which, following the ALARP principle, shall be minimized. In addition, if safety standards such as IEC 61508 are taken into account, in the case that coverage is $< 100\%$, a justification is called for. Consequently, this section describes a method that may be of significant value for next-generation safety-related systems that need to provide a justification of test-coverage

Listing 1: Simple case-study application

```
#define DEV1    "/dev/urandom"
#define DEV2    "/dev/null"

int main(int argc, char **argv)
{
  unsigned char result;
  int fd1, fd2, ret;
  char res_str[10] = {0};

  fd1 = open(DEV1, O_RDONLY);
  fd2 = open(DEV2, O_WRONLY);

  ret = read(fd1, &result, 1);
  sprintf(res_str, "%d", result);
  ret = write(fd2, res_str,
              strlen(res_str));

  close(fd1);
  close(fd2);

  return ret;
}
```

completeness. The method is executed in five consecutive steps:

- Experimental setup (see Section IV-A)
- Data-set collection and preprocessing (see Section IV-B)
- Fitting the data-set (see Section IV-C)
- Probability estimation (see Section IV-D)
- Code test coverage justification (see Section IV-E)

A. Experimental setup

In order to perform the estimation of execution path uncertainty, it is required to explicitly define the target-system, the software application(s) and the system context.

Within this publication, a simplified yet reproducible case-study described in Section III is used. For the safety certification of a given safety-critical system, the experiment setup corresponds to a set of test suites and campaigns that already define the target-system, the software application(s) and the system context.

B. Data-set collection and preprocessing

In order to be able to estimate the probability of execution of the unknown paths, it is necessary to first obtain a sample of known paths by recording the execution traces in kernel-space of the given experiment setup. Hence, we record the traces and we calculate their frequency of execution.

1) Recording: The recording of the traces is performed using the kernel tool known as *Ftrace*. *Ftrace* is a highly configurable tracing tool included in the mainline kernel that permits to record the function call sequence and key state information of the Linux kernel. Besides, *Ftrace* can be configured to trace only the kernel execution of a specific application running in user-space.

The recording is performed configuring *Ftrace* to trace the application process ID and then executing the application repeatedly in the defined system-context. As a result, we get a large number of files that contain the path followed by the application in kernel-space.

2) Post-processing: Each of the trace files is composed by a series of system-calls. System-calls are the interface that provide the services of the kernel to the user-space. For instance, the *open(DEV1, O_RDONLY)* calls *sys_open()* at the kernel-space and this one invokes a series of kernel functions. Hence, each trace file is composed by two *sys_open()* (one for each device), one *sys_read()*, one *sys_write()* and two *sys_close()* (one for each device).

The post-processing consists on identifying these system-calls in the whole trace file and separating them in order to perform the analysis per system-call and achieve a more detailed study.

3) Unique traces: Once the trace files have been processed and the traces for each system-call are obtained, the number of *unique traces* that exist in each system-call are enumerated and their frequencies of execution are inspected. We denote each different function call sequence as *unique trace*.

Table I collects the results obtained from 410000 trace files, which means that each system-call has been executed 410000 times. As Table I shows, an inherent diversity exists in the Linux kernel as each system-call provides different execution paths for the same function request. However, Table I also shows that in all of the system-calls there is one common path that constitutes the vast majority of executions and that only a small number of *rare-paths* are executed. Therefore, the question is, how can we estimate the probability of executing a further non-recorded path if we cannot ensure the collection of all existent paths?

TABLE I: Number of unique traces per system-call and frequency of execution of common the trace

system-call	Nº unique traces	Common path execution (%)
sys_open(fd1)	8348	92.21
sys_open(fd2)	7445	81.79
sys_read(fd1)	760	97.91
sys_write(fd2)	253	98.72
sys_close(fd1)	244	98.62
sys_close(fd2)	234	98.68

C. Fitting the data-set

Before estimating the probability of the *unseen traces* it is necessary to model the observed data. Frequency distribution needs to be modeled to represent data collection and summarize data for later interpretation. Frequency distributions follow a power-law distribution when the frequency of an event is correlated with the size of this event [19]. In the case of the Linux kernel, there is a *common path*, which is executed most of the times as shown in Table I, and the rest of the traces are considered variations of this *common path*. *Rare traces* are based on the trunk axis of the *common trace*, but a series of calls may be executed in different points of the trace. These

variations have a significantly lower probability of execution than the *common trace*. However, the frequency of the trace does not follow a linear proportion, since each variation has a different probability of execution. The variation probability is significantly lower than the probability of execution of the *common trace*. For example, if we have a *common trace* that follows the sequence of A-B-C, a trace with a variation that could exist is A-B-X-C, where X is the variation. Therefore, there is a high probability that the system will go from the function B to C and a small probability that it will go to X. At the same time, it may be the case where A-B-X-C-Y exists, with a second variation in C to Y. Therefore, this probability of executing a trace with a larger number of variations is further reduced. This assumption ignores the possibility that a branched-path has a higher probability than the non-branched path. The rational to neglect this scenario it is that Linux has been an evolutionary development model where performance was always crucial, and hence, if a *"dominant"* branched-path would exist then the kernel would have been optimized to reduce its execution probability.

As it is shown in Figure 1, the data-set fits a power-law distribution. This Cumulative Distribution Function (CDF) graph represents the data and model of the system-call *write*, with the horizontal axe collecting the frequencies-of-frequencies values.

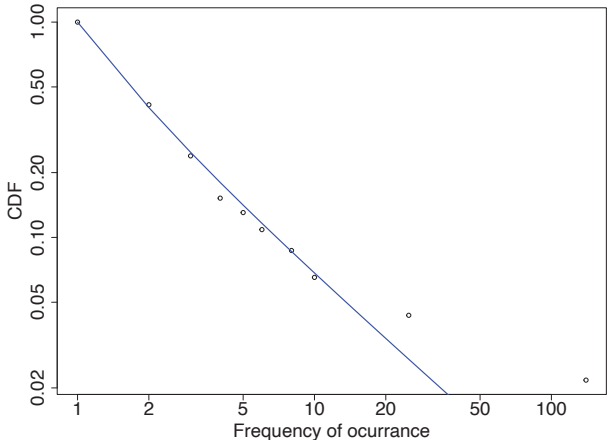

Fig. 1: Data CDF of the *unique traces* of system-call write and the fitted power-law (blue line)

Figure 2 depicts the results obtained in all system-calls. Even though the data-set can be considered relatively small from an empirical stand point, at least in this example application, power-law can be considered as a general distribution for all system-calls. As Figure 2 shows that power-law distribution fits with all data-sets.

Consequently, it is possible to approximate/model adequately the recorded results by fitting N_r values to the power-law function $F(r) = ar^b$. Figure 3 depicts the frequency distribution model of *system-call write* in *blue* color.

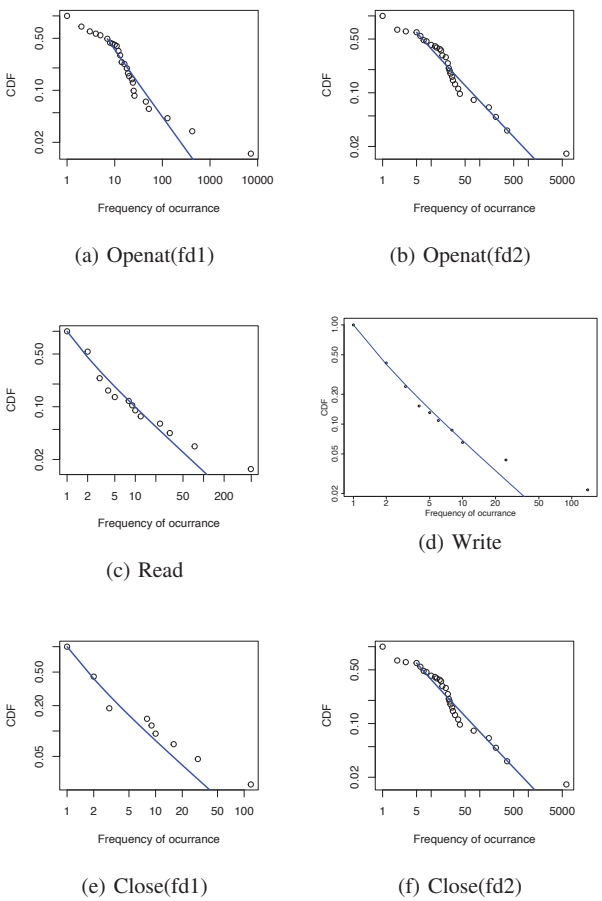

Fig. 2: Data Cumulative Distribution Function (CDF) of the unique traces of system-call write and the fitted power-law (blue line) for all system-calls

D. Probability estimation

Once the model has been obtained, it is possible to make an estimate using appropriate statistical techniques such as Good-Turing and P_{MLE} [18].

- Maximum Likelihood Estimation (MLE) is one of the simplest and most used methods to estimate the probability of events (P_{MLE}). However, this method only works when all the possible events are known, and as a result, the probability of unknown events is considered equal to zero. However, for the current scenario we know that there are several non-recorded traces that may be executed. Smoothing techniques permit to adjust the probability obtained with MLE to achieve a more accurate probability taking into account the unknown events [6]. For instance, Laplace estimator or additive smoothing pretends that all events have occurred at least once and, consequently, the probability is no longer zero [18]. However, this technique gives only a small probability

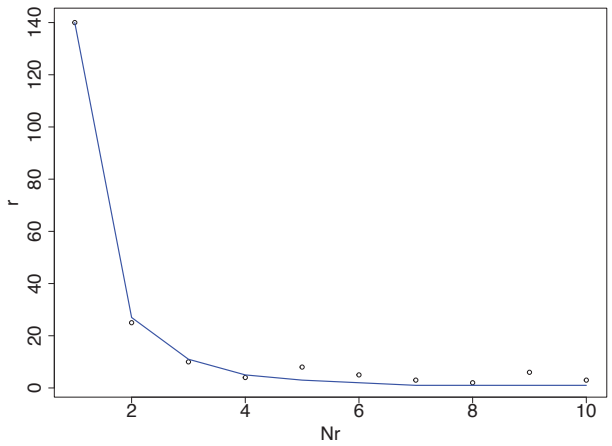

Fig. 3: Frequency distribution of system-call write (black points) and the statistical model following the power-law function $F(r) = ar^b$ (blue line)

to each unseen event and it may be problematic when there are a very large number of potential unseen events.

- Good-Turing statistical technique includes unknown events on the probabilities estimations [17]. The technique is based on pretending that an event observed with rate r in the sample actually occurs r^* times in the population. Equation 1 shows one of the methods for the calculation of r^*, where N_r represents the counts of the events with rate r. Equation 2 calculates the probability for each r^* value.

$$r^* = (r+1)\frac{N_{r+1}}{N_r} \tag{1}$$

$$P(X) = \frac{r^*}{N} \tag{2}$$

The disadvantage of Good-Turing is that it is unreliable for high r values and, in this case, there are traces with significantly large ratios compared to the rarest-paths. As a result, their probability of executions can be estimated with the combination of P_{GT} and P_{MLE} [18]. P_{GT} is used to estimate probabilities of unseen paths and rarely observed paths, while P_{MLE} is used for the calculation of the common (high frequency) paths.

We use Shannon's information theoretical entropy measurement in order to decide with which frequency estimation method we calculate r^* . Entropy provides the average of information offered by a group of variables. Therefore, this allows to divide the group for GT and the group for MLE with a more equitable amount of information between them.

Table II collects the probability results for system-call *write* obtained with Good-Turing technique and the probabilities obtained with MLE for *common-paths* (r values 8317 and 83161), where r^* equal to 0 represents the unseen events.

TABLE II: Probability results for each ratio in system-call write

r*	Probability	r*	Probability
0	3.415009e-04	8	2.195304e-05
1	9.408446-07	9	2.439227e-05
2	2.981277e-06	10	2.683149e-05
3	4.434958e-06	...	...
4	7.317680e-06	...	...
5	9.756907e-06	...	...
6	8.537293e-06	8317	6.583473e-03
7	1.951381e-05	83161	9.914554e-01

E. Code test coverage justification

Table III collects the probability of executing an unseen trace (P_{zero}) in each system-call. The results are obtained using the Equation 2 described in Section IV-D with the data-set of each system-call.

TABLE III: Cumulative frequency of estimated unseen traces per tested system-call

	openat(fd1)	openat(fd2)	read
Probability of unseen traces	1.75e-02	1.48e-02	1.19e-03
	write	**close(fd1)**	**close(fd2)**
Probability of unseen traces	3.41e-04	2.86e-04	2.87e-04

The results can be considered relatively small compared to the probabilities of the paths that form the data-set. In addition, it is possible to observe that the probability varies considerably depending on the system-call. The significant difference can be attributed to the quite different complexity of the studied calls - e.g. *openat(urandom)'s* most common path executed 116 functions while *write()'s* most common path only executed 11 and hence we expect a higher variability of *openat()* than for *write()*.

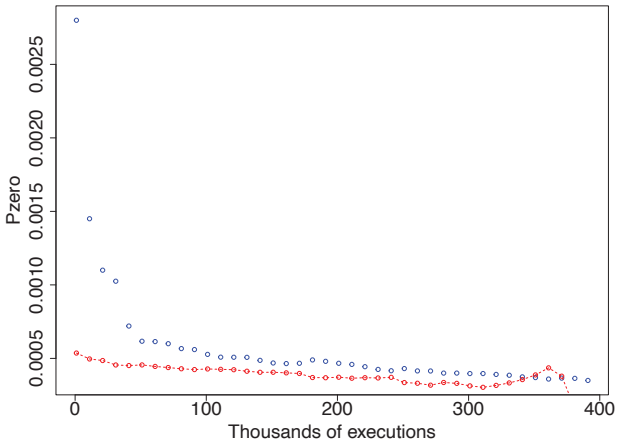

Fig. 4: P_{zero} values along execution traces increase (blue points). P_{zero} real value estimated with the remaining data-set

The results obtained for P_{zero} (probability of ratio 0) can be considered stable by examining Figure 4. The blue points

of the graph illustrate the stabilisation of the P_{zero} value of *system-call write* as the sample database increases. In other words, the probability of execution is reduced by increasing the data-set size, i.e. by gaining more knowledge of the system. Therefore, if the probability obtained is too high, this value can be reduced by the continuation of the system testing.

In order to check if Good-Turing estimations are realistic, we compare (1) the probability of appearance of unknown paths (P_{GT}) estimated using N samples of the data-set with (2) the probability of the new paths that exist in the remaining of the data-set (P_{MLE}). Figure 4 shows, on the on hand, (1) the probability of execution of unknown paths in *blue color* while the N samples increase and, on the other hand, (2) the probability of the paths that still do not appear in the N samples and exist in the data-set in *red color*. Note that, both, the preliminary results of P_{GT} *(blue)* and the tail of P_{MLE} *(red)* are not representative as they have been estimated with reduced data-sets.

V. Conclusions & Future Work

This paper introduces a statistical method for estimating the probability of execution of the non-tested kernel paths with the aim of paving the way towards the verification of safety systems based on the Linux kernel, where 100% test-coverage cannot be achieved. IEC 61508 safety standard states in Part 3-B.2 that appropriate justification needs to be provided in the case that test-coverage is below 100%. Consequently, the proposed method provides an objective acceptance criteria and high (justified) confidence as R2 rigor (adequate for SIL 3) states.

From this research, it is possible to conclude that the proposed method based on Good-Turing is a step forward towards the quantification of residual risk caused by software malfunction as risk is calculated by the multiplication of probability of occurrence and the severity of consequences ($risk = probability * severity$). Traditionally, residual risk caused by software malfunction has not been calculated because it was feasible to achieve full test coverage thanks to the determinism provided by relatively simple software and hardware. Nevertheless, for complex use-cases running on modern platforms, it may be considered a valuable information in order to quantify risks and apply an ALARP principle within the system testing.

Despite the fact that the final target of this research are complex safety-related systems in general, this initial study is more focused on Linux based systems. It is performed with a simple application at the user-space level to facilitate the comprehension of the preliminary results and, also, to get a statistically reproducible analysis. Note that if a simple application can be demonstrated to be well behaved in the statistic domain, we can generally infer that complex applications will behave even better. Since the large path variability obtained through an artificially-loaded-CPU system context makes our simple application into a representative case-study, we consider that the results presented in this work show that statistical based methods may be a powerful tool for next-generation safety-related systems. Nevertheless, we plan to apply the proposed method to an even more representative complex use-case and publish the results in the short term.

Finally, although promising results have been obtained in this first attempt, it is important to remark that before being deployed in a real safety system, this method needs to be further reviewed by safety experts and Certification Authoritys (CAs). Once the general approach is accepted, tolerable probability level shall be evaluated individually for each use case according to their specifics, such as the SIL to be achieved and the execution frequency of each system-call.

References

[1] Enabling Linux in Safety Applications (ELISA). Available: https://elisa.tech/, 2020. [Online].

[2] Imanol Allende, Nicholas Mc Guire, Jon Perez, Lisandro Gabriel Monsalve, Nerea Uriarte, and Roman Obermaisser. Towards linux for the development of mixed-criticality embedded systems based on multi-core devices. In *15th European Dependable Computing Conference (EDCC)*, 2019.

[3] Lukas Bulwahn, Tilmann Ochs, and Daniel Wagner. Research on an open-source software platform for autonomous driving systems. *BMW Car IT GmbH, Munich, Germany*, 2013.

[4] Jonathan Corbet and Greg Kroah-Hartman. Linux Kernel Development Report, 2017.

[5] Yarin Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016.

[6] Amir Hazem and Emmanuel Morin. A Comparison of Smoothing Techniques for Bilingual Lexicon Extraction from Comparable Corpora. In *Proceedings of the Sixth Workshop on Building and Using Comparable Corpora*, pages 24–33, 2013.

[7] Alex Kendall, Vijay Badrinarayanan, and Roberto Cipolla. Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding. *arXiv:1511.02680*, 2015.

[8] Alex Kendall and Roberto Cipolla. Modelling Uncertainty in Deep Learning for Camera Relocalization. In *IEEE international conference on Robotics and Automation (ICRA)*, pages 4762–4769. IEEE, 2016.

[9] Nicholas Mc Guire, Peter Okech, and Georg Schiesser. Analysis of inherent randomness of the Linux kernel. In *Proc. 11th Real-Time Linux Workshop*, 2009.

[10] Peter Okech, Nicholas Mc Guire, and William Okelo-Odongo. Inherent Diversity in Replicated Architectures. *arXiv:1510.02086*, 2015.

[11] Peter Okech and Nicholas Mc Guire. Analysis of Statistical Properties of Inherent Randomness. In *Proc. 12th Real-Time Linux Workshop*, 2010.

[12] Peter Okech, Nicholas Mc Guire, and Christof Fetzer. Utilizing inherent diversity in complex software systems. In *Proc. of The Australian System Safety Conference (ASSC2014). Australian Computer Society, Inc*, 2014.

[13] Peter Okech, Nicholas Mc Guire, Christof Fetzer, and William Okelo-Odongo. Investigating execution path non-determinism in the Linux kernel. In *Proc. 14th Real-Time Linux Workshop, Lugano. OSADL*, 2013.

[14] OSADL. SIL2LinuxMP - The Safety Project for Linux. Available: https://sil2.osadl.org/ [Online].

[15] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4), 2020.

[16] Andreas Platschek, Nicholas Guire, and Lukas Bulwahn. Certifying Linux: Lessons Learned in Three Years of SIL2LinuxMP. 02 2018.

[17] Geoffrey Sampson. *Empirical Linguistics*. A&C Black, 2002.

[18] Prof. Olga Veksler. Lecture notes of: Artificial Intelligence II.

[19] Ethan White, Brian Enquist, and Jessica Green. On estimating the exponent of Power-law frequency distributions. *Ecology*, 89:905–12, 05 2008.