Investigating Execution Path Non-determinism in the Linux Kernel

Peter Okech
Department of Physics
University of Nairobi, Nairobi, Kenya
pokech@uonbi.ac.ke

Nicholas Mc Guire
OpenTech EDV Research GmbH
Bullendorf, Austria
der@hofr.at
Christof Fetzer
Systems Engineering Group
Technische Universität Dresden, Dresden, Germany
christof.fetzer@tu-dresden.de
William Okelo-Odongo
School of Computing & Informatics
University of Nairobi, Nairobi, Kenya
wokelo@uonbi.ac.ke

Abstract

In Linux, a program executes either in user or kernel context. There is only a limited set of valid inputs for any program and thus in the absence of faults, the path taken by the program in user space is highly predictable. In contrast, the input space of the kernel is very large so it is expected that for any program executing in kernel context, there could be several possible code paths available to be taken during an execution instance. We set out to investigate the level of non-determinism in the execution path of an application in the Linux kernel. In our work, we define the path as the set of kernel functions that are called by a specific system call invoked from the user space by an application. The experiments involved running a simple program over a large number of iterations and tracing its execution in kernel space using the Ftrace tool. The trace output was then transferred to a PostgreSQL database for analysis. We report the early results of our work, focusing on tracing a single system call, the open() call. The frequency of path occurrences has several peaks representing the common execution paths as well execution paths that are rarely taken. Further analysis shows that interrupts have an effect both on the execution time and control flow through the kernel. The result of this preliminary work points to a strong possibility of both temporal and path non-determinism in tasks executing in kernel space, due to complexity of the execution environment. This inherent diversity of OS kernel state could be useful in the protection of replicated systems against residual faults in the execution platform.

1 Introduction

Programs are generally designed to be deterministic or explicitly employ randomness. Typically, for the same input and assuming correct execution, a program will generate the same output. If the program runs on deterministic hardware, then program

control flow is the same for a given input. The expectation is that the instructions will take the same amount of time during different independent execution of the program. In reality, there is marked variation in the execution times of even a simple application running on contemporary computing platforms. Reducing this variation, by aiming for low latency, is one of the main goals of real-time systems research. A typical approach is to attribute latency or jitter to software constructs associated with specific code-paths or event sequences and perform optimization to eliminate them.

In our earlier work [3], we argued that there are distinct parts of the system jitter that are associated code-paths being executed in system level context. This is due to some level of inherent randomness in complex software systems running on non-deterministic hardware. The assessment of inherent non-determinism in complex computing platforms is a running thread of our work.

In this paper, we report the results of our preliminary work on the assessment of the level of path non-determinism in the Linux Kernel. We set to find out whether an application does or does not follow the same code path while executing in kernel space every time it runs. Our tests are done on an otherwise idle machine so we do not have to factor in the effect of system load on the results. We use the open system call as a case study of a task running in system context. Our approach involved tracing the path taken in the kernel space by the Linux kernel function sys_open(), which implements the open system call invoked from an application, using the tool Ftrace [4]. We analyzed the trace data to determine the code paths in the execution runs. We also examined how interrupts which are events that are asynchronous to the executing task affect the code paths.

The main contributions of our work reported in this paper can be summarized as:

- We provide an insight to the code path of the open system call. Through repeated execution we collected and analyzed data on the code paths. We believe the properties of the code paths hold for a majority of system calls except for the most simple ones.

- We discuss how interrupts affect an application's code path in kernel space and introduce non-determinism in the path.

- We describe an approach to capture and analyze execution trace data. The steps of the method can be automated to enable continuous monitoring and analysis of execution paths in the kernel.

The rest of this paper is structured as follows. Section 2 introduces the problem. In section 3, we describe the method for data capture and analysis and present that result in section 4. We provide a discussion of our results in section 5 which is followed by a review of work related to ours. Finally, section 7 provides a conclusion and gives an indication of the work we intend to do in the future.

2 Problem Description

Since the very early days of computer science the dominant model of computers is to treat them as basically deterministic systems and where this is not the case force them to be so through the use of hardware/software constructs. The question of how non-deterministic these systems actually are does not seem to be asked too often.

2.1 Background

The software architecture of the GNU/Linux system can be viewed as consisting of two layers, the kernel space and user space. Kernel space is the memory area where the kernel code is loaded and executed, and it is usually protected from any unauthorized access. User space, on the other hand, is the memory area where user processes are loaded and executed. User processes usually access kernel code by requesting services by invoking system calls which provides an interface to hardware devices [1] and other system resources. Typically, a user process will invoke several system calls in its interaction with the operating system. The set of system calls that the process makes does not change for a given input vector in a fault free environment, and thus this set can be considered as its fingerprint.

Consider the control flow of a linear non-concurrent program. Since there is only a limited set of valid inputs for any program, in the absence of faults, the control flow of a program is highly predictable. On the other hand, simple user space programs that use non-deterministic inputs for de-

cisions or complex user space applications are non-deterministic from a control flow perspective.

In the safety domain in particular and in other application areas, applications are required to adhere to the KISS principle at design, and thus the aim should be the development of low complexity software with simple control flow in user space. But what is the implication of this design principle on the control flow of a program in kernel space?

The kernels of commercial operating systems are large, and complex with respect to their internal structures and communication. These kernels exhibit vast functional dependencies across subsystems and have practically speaking infinite input space. The expectation is that in such a setup, the control flow is not deterministic.

This provides the motivation for this work. We set out to find whether a program does or does not follow the same code path while executing in kernel space every time it runs.

2.2 Motivating Example

Consider a prototypical safety application - a software watchdog timer that monitors some system property such as liveliness of processes or resource usage or system load. The application will read the attributes of the entity/entities it is monitoring and based on a specified threshold value, perform some prescribed action, for example a shutdown, reboot or sometimes a repair to handle the problem. After the initialization process, the main steps of the such a program are more or less fixed and are performed periodically. Due to complexity of the execution environment, we expect some level of variance in the timing and the path of such a program.

With respect to the path we can ask ourselves the following question: Will this path variation occur in user space or kernel space? We hypothesize that the variability in the execution path will occur in kernel space. To be able to determine if there is the possibility that independent executions of such a program would take different paths in kernel space, we needed to show that this is the case for a simple program while focusing on the code path of a single system call.

2.3 The case study and tools

As an exemplar of the system calls, we chose to look at the open system call. In Linux, the open system call is implemented as sys_open() and is defined in the file fs/open.c. The details of the system call are well documented and we urge those interested to access the relevant sources.

The Linux kernel execution can be traced with several tools. Some popular tools are LTTng [2], Ftrace[4], and System Trap [8]. Since we are interested in what happens in kernel space, we looked for a tool has a simple interface and is built-into the system kernel. We settled on Ftrace since it matched our criteria.

3 Method

3.1 Environment

Our interest is in analyzing the execution events associated with an application in kernel space. The starting point was to configure and rebuild the Linux kernel to enable support of Ftrace. We cloned the Linux Kernel version 3.7.0-rc7 from the git repository, and set the kernel configuration tracers options. Our aim was to identify the functions a task in kernel context was calling, and the flow of execution within the kernel. For this purpose, we configured the CONFIG_FUNCTION_TRACE and CONFIG_FUNCTION_GRAPH kernel options before compiling the kernel.

Our experimental environment consisted of the following:

- AMD Phenom 9560 Quad-Core running at 1150 MHz, with 1.7GB memory
- GNU/Linux Debian release 6.0, Linux Kernel 3.7.0-rc7
- Postgresql version 8.4

3.2 Description of the Experiment

We wrote a test program in C, that directly invokes the open system call as shown in program listing below. The program opens the file in the /dev/random, the system's non-deterministic generator, reads from it and then closes the file. After compiling with the

gcc defaults, we did a trace of the program using the Strace [9] utility. The output of the trace showed 3 calls to open(), two of them to open system libraries and one to open the file /dev/random, in addition to other system calls.

```
int main(int argc, char *argv[]) {
      int fp, ret, n;
      int randomData;
      fp = open("/dev/random", O_RDONLY);

      if (fp != -1){
           ret = read(fp, &randomData,
                                sizeof(randomData));
      }
      n = close(fp);
      return 0;
}
```

In the experiment, the test program was executed via a shell script in a loop of 10,000 runs. This statements of the script are depicted in the pseudocode shown below. Thus for each execution, we captured the activities of each of the CPU cores into a text file.

```
Begin
    set trace options
    for 1 to 10k do
        clear trace buffer
        enable tracing
        execute application
        disable tracing
        copy trace buffer content to file
End
```

The script generated 10,000 data files of execution traces. To enable us to analyze the functions called by the sys_open() function invoked from our application, we then filtered out from each of the trace files instances of the call to sys_open() using a python script. We view these instances as a tree rooted at sys_open with each function it called having a row for its entry and a corresponding row for the exit, except for those function that do not have nested calls. The result of filtering instances of sys_open() function were stored in text files, one for each instance.

Having the information on traces of function calls of the sys_open() tree in text files is however limiting on the type and level of analysis that can be done on the data. To deal with this limitation, we created a database in PostgreSQL to hold this data. Our table design was simple, mirroring the columns of the Ftrace output. We then wrote a script to copy data from the files into the database tables.

The rows of the database tables contained the details of each function called by the sys_open(), and any other kernel functions called in the context of our test process. To identify the path of an execution instance of the open system call, we generated a string by concatenating all the tuples from the table of the trace instance. The resulting string representing the path for each instance was then stored in a table.

The data set was then available for querying and analysis. For example, an important question that this work seeks to answer is if there are possible multiple paths that can taken by a process in kernel context. In other words, when the process switches from user context by invoking the a system call, do all repeated executions take the same path? We used the features of SQL to group the strings representing the paths and find out the distinct paths in the runs of the test program.

4 Experimental Results

As described in section 3.2, the test program makes three open systems calls from user space. Using the python script previously mentioned, we processed the 10,000 trace files and extracted 20,807 well-formed trees rooted at sys_open() and loaded them into the database. Table 1 shows some of the properties of our data set.

No of trace files	10,000
Data size/sys_open() instances	20,807
Interrupted instances	534
Instances that were context-switched	93
Instances in which page faults occurred	5194
Instances with memory caches allocation requests	4307

TABLE 1: *Properties of the data set*

The analysis we performed on the data set can be categorized into two, namely time variability and code path variability.

4.1 Temporal Variability

We looked at the time taken for the open system call to run to completion. This information is available through the time stamp in the trace file for each function exit. We retrieved for each instance of the system call the duration at function exit and this is plotted in the figure 1. Note that the time stamp will also have a small variance, but for this analysis the variance was disregarded and does not impact the conclusion.

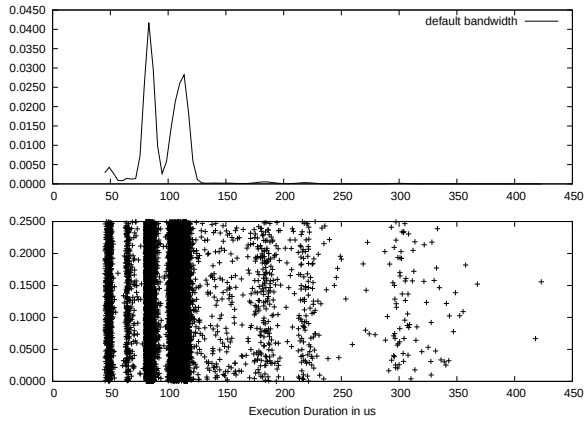

FIGURE 1: *Time distribution of all the open system calls*

We also examined the distribution of the time taken for functions that were interrupted, i.e. those that invoked the function do_IRQ() or smp_epic_timer_interrupt() in the context of our target process. The results are depicted in figure 2.

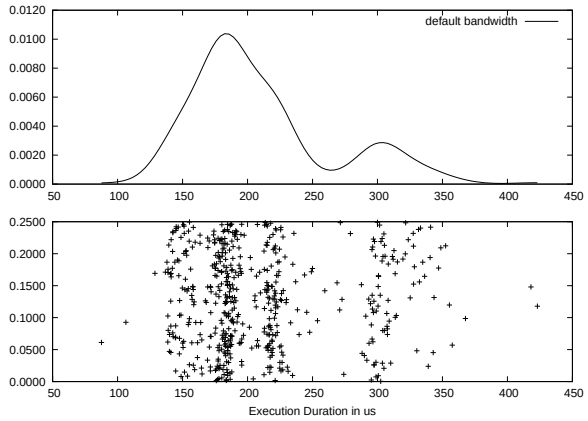

FIGURE 2: *Time distribution of interrupted open system calls*

4.2 Code Path Variability

We examined the number of distinct paths of the open system call in kernel context. The details are summarized in table 2. In this data set, there were a total of 559 distinct paths, with only 34 of these having an occurrence frequency of 2 and above. The remaining have only one instance for the path. The occurrence frequency of the different paths is shown in figure 3.

Total	
Number of distinct paths	559
Distinct paths with occurrence frequency of 1	525
Distinct paths with occurrence frequency >1	34
Interrupted instances	
Number of distinct paths	526
Distinct paths with occurrence frequency of 1	518
Distinct paths with occurrence frequency >1	8

TABLE 2: *Characteristic of the paths*

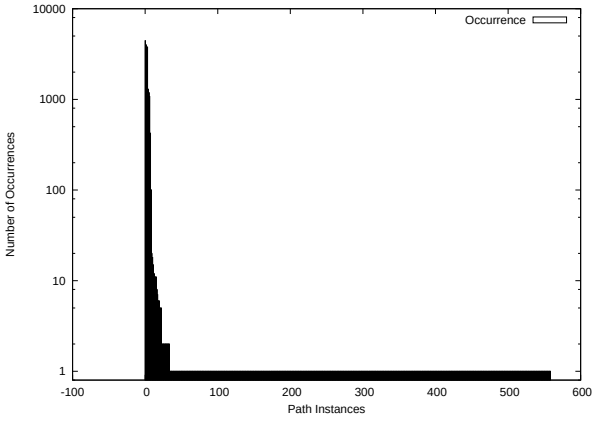

FIGURE 3: *Path occurrence frequencies*

It was our intuition that the single occurrence paths arise due to asynchronous events such as interrupts and context switches. In order to confirm this, we isolated those paths that were interrupted during the execution of the system call. With the interrupted instances filtered out, we ended up with a total of 33 unique paths. The plot of the path occurrences without the interrupted code paths are shown on figure 4.

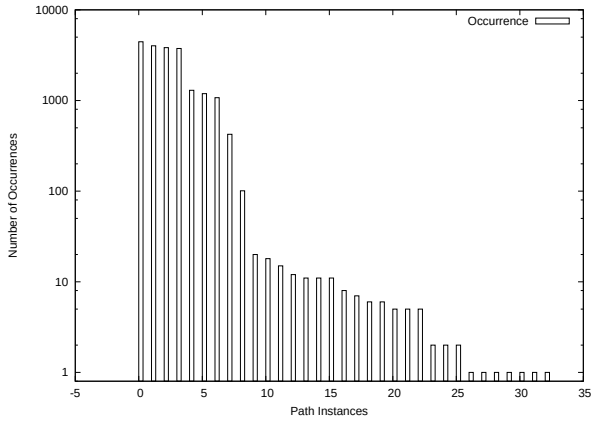

FIGURE 4: *Path occurrence frequencies for non-interrupted open system call instances*

5 Discussion

The results achieved show that there is both temporal and path variability of the open system call from repeated execution of our test program.

The case of path variability is of more interest to us. From the modest data set of 20,807 instances of the open system call, we identified 559 distinct paths. Out of these, 93.92% of the path instance were unique, i.e. had a occurrence frequency of 1. This number is significant given that the occurrence frequency of the other 34 paths ranged from 2 to 4447. This results afford us the following interpretations.

With 559 possible paths, 21.37% of the 20,807 open system call instances in our experiment took the same path as shown by the highest peak of the plot in figure 3. Out of the total, 40.65% took either one of the two (2) most likely paths, 94.18% took one of the seven (7) most likely paths and 96.70% followed one of the ten(10) most likely paths. The results shows that not all the paths are equally likely to be taken. This has an implication on where in the open system call code path residual faults would be present. We contend that during testing, the developers would encounter these most likely paths (the percentage of the paths covered would be based on their budget) and exhaustively test them. This would lead to the possibility that the rarely encountered paths not being well tested and hence would contain bugs.

Our analysis of the unique paths pointed to the role played by interrupt on the control flow of a system call. Linux is a fully preemptible operating system, so tasks in kernel space can also be preempted. The tests were done on an otherwise idle system, and thus it was not expected that preemption would occur frequently. The sys_open() function instances that were interrupted constituted only 2.57% of the total. Though very small in number, these contributed to 98.48% of set of single occurrence paths. We expect that in a system with a higher system load, interrupts would have a higher impact on the path in kernel space.

There were 93 context switches (these are a complete subset of interrupted instances) during the execution of the sys_open() function. All the paths that contained a context switch were found to be unique. On the other hand, none of the sys_open() instances had a case of a switch from one processor core to another. A look at the task representing the test program however had several cases of processor switches, though none of these happened during the open system call.

We also examined if there was dependency between successive paths, for example, if path 15 occurs does the occurrence of path number 16 more likely in the next system call instance? We also tried to find out of there was a correlation between the paths and call instances. We thus enumerated all the paths, and plotted this against the system call instances. For readability, we present in figure 5 part of this plot using two random blocks of 100 system call instance from our data set.

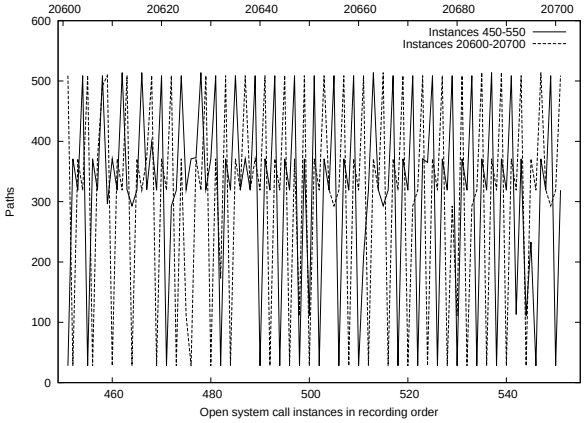

FIGURE 5: *Path taken by a system call*

After filtering out the interrupted instances of the sys_open() function, we ended up with 33 distinct

paths. These are functionally equivalent paths, that is semantically they are equivalent from the point of view of the invoking application, but internally they exhibit functional diversity. This diversity can be attributed to events in the code path which we believe is associated with the state of the execution environment. What comes to mind is the requests for memory pages and/or cache for allocation to kernel objects, which might result in page faults or the need for a cache refill should the cache be empty. At this point in our work, we are yet to perform an analysis of these events, and hope to do so in the future.

This preliminary investigation gives us confidence that the we have been able to set up an experiment to probe execution traces of the open system call and identified the variance in the execution paths of the system call. However, there is a need for deeper analysis of correlations between paths and the execution events that we have identified.

6 Related Work

Dynamic analysis is widely used by software engineers and developers to study system behavior. One of the main techniques of dynamic analysis is the execution of a system and generating execution traces for the system under study, termed as execution tracing. Execution tracing has been applied in different scenarios, including understanding of the behavior of distributed systems [5] and comprehension of operating system behavior [6]. The first work focuses gathering information about distributed applications for maintenance purposes. The second, though discussing understanding for program maintenance, it advocates for the use execution tracing with operating systems kernel as the target software. In our work also targets the operating systems kernel, but with the aim of finding out how different the traces are in different execution instances.

The work closest to ours is that of the SIL4Linux project [7]. Similar to our work, they trace system calls implemented in the Linux kernel. The trace data gathered are similarly stored in database tables to enable efficient querying and analysis. While one of their main aim is to check if the systems calls implemented in Linux conform to the POSIX standard, we are more interested in the variability of the behavior of specific system calls with respect to the functions it calls during its execution.

7 Conclusions

In this paper, we have described our approach to determining whether there is path non-determinism in the execution of a program on a contemporary architecture running Linux. We have shown that for the open system call, there is some common path and non-common paths that to some extent are not correlated. This observation we believe should hold practically for all but the very simple system calls.

It is well known that testing cannot satisfy the needs of complex systems. But if testing covers systematic faults in the common paths and the non-common paths are randomly distributed, then the faults on these non-common paths can be covered by architectural protection. As the occurrence frequency's presented in this paper indicates, there are some non-common paths in the runs. We postulate that these paths that are rare are "untested", but those frequently occurring are "well-tested". If we now have two systems that are essentially identical from the software installation perspective running concurrently (that is in a 2-out-of-2 configuration), then we can assert from our tests results that the probability of both systems being in a rare path is low and thus architectural replication and inherent non-determinism would cover faults in the untested paths.

We intend in the future to extend our investigation in several ways. The first is to repeat the experiment for larger number of runs and a sequence of system calls. Secondly, we would like to design an experiment to run similar tests on a loosely coupled system architecture. By analyzing the data from a larger data set and from isolated runs we would be able to provide a more solid model of inherent non-determinism for protection against residual faults in complex computing platforms.

8 Acknowledgments

The first author acknowledges the financial assistance of the Mundus ACP II Project for the PhD mobility period spent at Technische Universität Dresden where this work was done.

References

[1] *Understanding the Linux Kernel*, Daniel P. Bovet, Marco Cesati, 3rd Edition 2006, O'Reilly

[2] *The LTTng tracer: a low impact performance and behavior monitor for GNU/Linux*, M. Desnoyers and M. R. Dagenais, 2006, in Proceedings of the Ottawa Linux Symposium, Ottawa

[3] *Analysis of Inherent Randomness of the Linux kernel*, Nicholas Mc Guire, Peter Okech, Georg Schiesser, 2009, In Proceedings of the 11th Real-Time Linux Workshop, Dresden

[4] *Finding Origin of Latencies Using Ftrace*, Steven Rostedt, 2009, In Proceedings of the 11th Real-Time Linux Workshop, Dresden

[5] *Understanding distributed systems via execution trace data*, J Moc and David A Carr, 2001, In Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)

[6] *Improving Program Comprehension in Operating System Kernels with Execution Trace Information*, Elder Vicente, Geycy Dyany, Rivalino Matias Jr, Marcelo de Almeida Maia, 2012, In proceedings of the International Conference on Software Engineering and Knowledge Engineering

[7] *SIL4Linux: An attempt to explore Linux satisfying SIL4 in some restrictive conditions*, Lijuan Wang, Chuande Zhang, Zhangjin Wu, Nicholas Mc Guire, Qingguo Zhou, 2009, In Proceedings of the 11th Real-Time Linux Workshop, Dresden

[8] *http://sourceware.org/systemtap/*

[9] *http://sourceforge.net/projects/strace/*