Utilizing Inherent Diversity in Complex Software Systems

Peter Okech\textsuperscript{1}  Nicholas Mc Guire\textsuperscript{2}  Christof Fetzer\textsuperscript{3}

\textsuperscript{1} Department of Physics
University of Nairobi,
Nairobi, Kenya,
Email: pokech@uonbi.ac.ke

\textsuperscript{2} OpenTech EDV Research GmbH
2193 Bullendorf, Austria,
Email: der.herr@hofr.at

\textsuperscript{3} Systems Engineering Group
Technische Universität Dresden,
Dresden, Germany,
Email: christof.fetzer@tu-dresden.de

Abstract
Diversity in simple software systems is known to be of limited effectiveness with respect to systematic faults, whereas in complex systems this picture drastically changes. Diverse implementation of complex software is generally perceived as expensive. In contrast, security uses a different form of diversity to protect against systematic faults: randomization. Explicit memory-space randomization has been shown to be effective for a large class of vulnerabilities. This diversification is carefully designed into the system to assure its effectiveness, but what about the system’s inherent non-determinism which manifests itself in inherent diversity of complex systems? Could this randomization be exploited in the form of a diversity argument? We propose an alternative safety strategy utilizing inherent diversity for replicated system architectures to mitigate residual systematic faults. We argue that the effective diversity is a result of inherent non-determinism. Our strategy is to review, analyze and test the dominant execution paths and demonstrate statistical independence of rare paths, allowing architectural protection based on runtime diversity. In this paper we present early results of analyzing the inherent non-determinism of the Linux kernel, and its potential to cover residual faults in the untested execution paths of a complex software system.

Keywords: diversity, complex software systems, randomization, residual faults

1 Introduction

Software based systems are increasingly being called upon to perform safety functions that can no longer be left to electromechanical systems. These software based safety-related systems must be certified as adequately safe in accordance with a generic standard such as IEC 61508 (IEC 2010) or with an industry specific standard before installation and operation. The standards prescribe the set of procedures and techniques to be used in the software development process (e.g. part 3 of IEC 61508). In addition, for highly reliable systems, the standards mandate the use of architectural protection to provide coverage for faults during operation of the safety-relevant systems.

A software system typically consists of an application and system software that supports its execution. On one hand, the safety applications that are designed to fulfill the safety function can effectively be designed using the principles of keeping the software simple and safe. In contrast, the system software that forms part of the execution platform is increasingly becoming complex particularly based on the need to manage contemporary hardware resources. We no longer have the luxury of developing all the components in the software stack, but are forced to look at ways of using off-the-shelf complex software components such as general purpose operating systems in safety-related applications.

The use of complex system software in a safety context presents a challenge. The challenge is twofold: first complex systems tend to have a higher density of residual faults and secondly, current prescriptive certification standards are rigid in the approach to demonstrate the safety of the system. A way out of this is to view complexity as a resource that need not be eliminated but exploited. This suggests that we need to look for approaches that enhance safety through the utilization of complexity.

In this paper, we present one such approach to utilizing the non-determinism in complex execution platforms to protect systems against systematic faults in software and suggest a strategy for the demonstration that a system based on this approach is acceptably safe. Specifically, we

- Explain the concept of inherent diversity in a homogeneous multi-channels system, and how it can be used for fault detection in a replicated system architecture (section 4). We argue that using inherent diversity on a 2-out-of-2 (2oo2) system is sufficient to provide safety.
- Use the GNU/Linux kernel as an example of a complex system software that illustrates the variability in path execution. In section 5, we present...
our approach to tracing system calls in the Linux kernel and present the results of our experiments.

- Present our hypothesis on the existence of faulty paths in complex software in section 5.4. We then provide a basic argument for justifying the use of inherent diversity (section 6.1) to argue coverage of a subclass of systematic faults in replicated complex software systems.

To investigate the non-determinism in complex software systems, we have used a function tracer to record the functions in the Linux kernel executed by system calls invoked from a test application. Though the system we used was more or less idle, there was variability both in the path and in the execution times of the system calls. We did not consider timing differences in this study but focused on the differences in the kernel execution paths. For a system with a higher load we expect an increase in non-determinism, which is better for inherent diversity.

The trace data of each system call execution was analyzed. Several paths for a given system were identified, with some paths being more frequently taken than others. It can be assumed that the rare paths are not well tested and thus any residual faults in those paths would stay undetected. Given that these paths are taken in very rare circumstances, the probability of any applications in any two channels running concurrently taking the same path is sufficiently low.

Further, we claim that taking one of the rare paths within the kernel is a random event based on the inherent non-determinism of the system state. If this claim holds, then, we can protect the system against systematic faults in these rare path by architectural means such as a 2oo2 system in the same way the architecture will provide protection against true random events like bit flips. This provides the foundation of the argument that we propose for justifying safety based on inherent diversity.

We believe that it is reasonable to make the above claim of concepts based on the data from the work reported in this paper. However, these are our early results and definitely more work still needs to be done to arrive at a conclusive assessment of inherent diversity.

2 Problem Definition

Consider a software engineer tasked with the problem of developing a software based safety application today. The system will need to be developed according to the prescribed procedures and processes and, be certified as safe by an appropriate safety standard. Suppose the safety standard provides the possibility of using contemporary hardware (with performance enhancing features, multicore, hardware acceleration etc) for which the required selection process is duly followed. The main issues the engineer has to deal with then would be on how to realize the software that performs the required safety functions. Naturally, the software will consist of the safety application and system software for example, an operating system.

Can the engineer make use of a general purpose operating system, such as GNU/Linux that was not specifically developed with safety concerns during its design or implementation? The use of such complex systems would pose two challenges:

- There is an expectation that such a system will contain a high number of residual systematic faults.
- How would the system be justified as adequately safe, given that it was not developed using a process compliant with a safety standard.

We suggest the use of inherent diversity as the architectural means to provide coverage of residual systematic faults in the complex software system and propose an approach to argue the safety of a system based on inherent diversity.

3 Background

Before discussing the approach to inherent diversity and the process of quantification of safety, we provide some background in this section.

3.1 Software Diversity

Dependable systems incorporate fault tolerance features to first detect errors, and then to mask, avoid or recover from the effect of these errors. To achieve these mitigation, redundancy techniques are typically used to implement fault tolerance; hardware redundancy to provide coverage for random faults and diversity for design faults. A central theme in this section is how to generate diversity in software systems. We first provide a brief overview of redundant architecture and how diversity is used in software based systems for the purpose of achieving safety using some adjudication scheme.

3.1.1 Redundant Architecture

The use of redundant channels in safety-related systems predates programmable electronic systems. There are several configurations or architectural protection schemes that can be used to cover random faults, or at least detect such faults. These configurations are generally referred to as M-out-of-N (MooN) systems. The simplest redundant system is the 1oo2 in which one channel is enough to perform its function. The configuration is used to achieve availability, e.g. in hot standby mode, but never used in safety. The 2oo2 system requires both channels to perform the safety function which allows for a single fault detection but provides no fault masking. The 2oo3 or the triple modular redundancy (TMR) provides single fault coverage, masking the fault in one of the three channels enabling it to perform its safety function with two healthy channels as well as providing high reliability. Other configurations require more resources and are rarely used in practice. In this work, we will use the 2oo2 system as an example to illustrate concepts.

3.1.2 Design Diversity

Due to the acceptance of the notion that all software faults are design faults, it has generally been perceived that pure replication of software would result in the channels having the same faults, and thus will result in a common cause failure (CCF). Randell (1975) proposed design diversity, the use of a spare component whose design was independent from the main components, to protect redundant systems from CCFs. A widely used technique for generating diverse versions of software components is N-version programming (Avizienis & Chen 1977). The idea is that N different teams are provided with the same specification to develop different implementations (versions) of the software component. A further modification to the concept is to force the developers to use different
languages and/or methodologies (Littlewood 1996). Given the same input, the different versions run concurrently and their output is compared using some adjudication mechanism.

The basic idea of design diversity is that the versions running will fail independently, thus exhibiting failure diversity. Assuming independence between failures of the channels, it is possible to claim much higher reliability of the system than that of the individual channels. Both empirical analysis (Knight & Leveson 1986) and theoretical work (Eckhardt & Lee 1985, Littlewood & Miller 1989) show that even for independently developed software versions, there are positive correlation between channel failures. This is attributed to the fact that for given programs, failures are more likely to happen on certain demand than others.

Though design diversity is an effective way of improving the dependability of software based systems (Littlewood et al. 2001), its uptake has been low except for very high integrity systems. The main reasons are its cost effectiveness and well as the inability to correctly quantify the gains in dependability of systems based on design diversity.

The automatic generation of diverse versions has been proposed as a means of reducing the high cost involved in manual development, testing and maintenance of different versions. The common approach is to use a compiler to generate the diverse copies from the same base program code. The technique has been used to tolerate hardware faults (Gaiswinkler & Gerstinger 2009) and is also increasingly being used in other dependability contexts such as security.

### 3.1.3 Diversity in the Security Domain

The security community recognizes that design faults can be taken advantage of by malicious users, thus compromising systems. To protect against these incidences, the ability to use the known faults also termed as vulnerabilities, is made as expensive as possible through the use of randomization. By randomizing features such as memory addresses and/or instructions, a large number of vulnerabilities can be effectively dealt with. To ensure its effectiveness, this randomization is explicitly designed into the system.

Unlike the safety domain which focuses on the use of specifically designed and verified software stack, the security domain has for a long time accepted the use of general purpose operating systems for their applications. A key aspect is the recognition that it is not possible to eliminate all systematic faults from complex software systems by methodological life-cycle development only. Rather than eliminating individual defects in such systems, the broad approach used by the community has been to obscure these faults by introducing non-determinism in the execution environment, thus deriving diverse copies of systems.

The main lesson from the security community applicable to our work is that there exists a generalizable strategy to mitigate systematic faults: through randomization of an execution environment, systematic faults can effectively manifest themselves as random faults. Thus, the well established mitigation strategies for random faults can be applied to mitigate randomized systematic faults.

### 4 Inherent Diversity

A complex system always displays behavior that arises from the interaction of its components and such behavior cannot be identified by looking at the components in isolation. Modern hardware/software systems are built from processors that exhibit a certain level of inherent randomness associated with complexity rather than particular code paths (Mc Guire et al. 2009). This inherent randomness is amplified by complex operating systems.

Computing platforms thus have a reasonable amount of inherent diversity. This is a property of a system that for a given input vector, the system will not have a deterministic path of execution. In this work, we describe a path as a sequence of functions or routines that are called during an execution, neglecting execution time differences. In the context of kernel path variability, a path is the set of kernel routines called during the execution of a system call when a user application requests a service by invoking the system call.

Inherent diversity arises due to several factors at play in a complex system. First, these systems have a very large state space, i.e. many variables and other data structures. Due to this it is conceivable that during the lifetime of the system, some of the states are never revisited. Secondly, most complex systems are concurrent systems and the parallelism intrinsic in these systems increases the potential for non-determinism. The third factor contributing to the diversity is the presence of a large number of asynchronous events. Together, these factors lead to unpredictable internal states of the system and thus non-deterministic paths from input to output, without impacting the correctness of a task.

Consider for example a 2oo2 system with replicated non-diverse safety application. We make the assumption that safety application is simple, and thus can easily be verified using existing tools while the operating system is complex and contains residual faults. Due to the inherent randomness of the execution platform, the two channels will diverge in state (see Figure 1). If each of the applications invokes a system call, the execution path in the kernel will likely take different paths. Suppose there is an untested path containing a fault, the probability of two systems taking the same erroneous path at the same time is much less than one of them taking this path. Using adjudication, it is possible to detect the fault. We believe that the 2oo2 architectural configuration based on inherent diversity is sufficient to provide safety even for highly critical systems.

![Figure 1: Diversity in a redundant channel](image-url)

**Figure 1**: Diversity in a redundant channel
5 Evaluation of Non-determinism in the Linux kernel

To support the claim of inherent non-determinism which manifest as inherent diversity in complex software systems, we set to investigate this phenomena in a real world complex system: the GNU/Linux operating system.

We set up an experiment to trace the execution of system calls in kernel space and performed a top level analysis of the data collected. In this section, we describe our experimental set up, method and the results obtained. We then complete the section by a discussion of the results that were obtained.

5.1 Experimental Environment

To run our experiment, we used a generic PC, with the specification typical of machines available to the community for use in control systems. The hardware/software specifications of the experimental platform were as follows:

- Intel Core 2 Duo CPU running at 3.00 GHz with 1GB main memory
- GNU/Linux (Debian release 6.0), Kernel version 3.12.0 #SMP PREEMPT
- GCC 4.4.5

Our interest was the analysis of execution events associated with an application in kernel space. To support kernel tracing, the utility Ftrace (Rostedt 2010) was enabled in the kernel configuration.

5.2 Experiment Description

The first step of the experiment was the development of a user application that would invoke systems calls to request for operating system services. We make use of a simple example that reads from a value from a binary file, increments the value and then writes back the new value into the file. The read-write cycle is performed within a loop. The application’s pseudo-code shown below.

```
BEGIN
  open file
  for count from startValue down to 0
    read data from file
    increment data
    write data to file
    sleep for 10 sec
  close file
END
```

The size of the program’s loop was set to 20,100 iterations. After compiling the test program with gcc defaults, the program was launched immediately after a fresh boot of GNU/Linux in a command line mode. This provided a more or less idle system with minimal services executing in the background.

We used trace-cmd (Rostedt 2010), a front-end tool to Ftrace, to record the traces of the kernel functions called by the test program. To aid the analysis task, the trace file was pre-processed using shell scripts in order to convert the file to text delimited format suitable for exporting/importing to a PostgreSQL database. We chose to transfer the data to a database to make it easy to perform complex queries and analysis on the data.

5.3 Results

In the execution run, a total of 180,975 system calls were recorded in the trace file. Most of these were the repetitive invocations of system calls in the set \{read, write, lseek, fsync, nanosleep\}. These system calls were explicitly invoked by the test program. We then performed a two level analysis on the data collected, one on individual system calls, and the other on the set of system calls in the program iterative loop.

As described in section 4, an execution path of a system call is a sequence of kernel functions that are called during that system call. The length is thus the number of functions in this sequence. The number of functions called during an execution instance depends on both the system call as well as the execution environment. The variability in the duration of execution and the length of the paths for system calls in the set \{read, write, lseek, fsync, nanosleep\} is shown in Table 1. The large difference in both timing and path lengths are due to among other factors, preemption of the system calls during execution.

```
<table>
<thead>
<tr>
<th>System Call</th>
<th>Duration in us</th>
<th>length of path</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>15.279</td>
<td>77</td>
</tr>
<tr>
<td>write</td>
<td>8.836</td>
<td>12</td>
</tr>
<tr>
<td>lseek</td>
<td>1.915</td>
<td>4</td>
</tr>
<tr>
<td>fsync</td>
<td>8418.327</td>
<td>402</td>
</tr>
<tr>
<td>nanosleep</td>
<td>9990266</td>
<td>47</td>
</tr>
</tbody>
</table>
```

Table 1: Timing and Path Variability of the data set

Since our focus is on the paths taken by individual system calls, we present in Table 2 a summary of the path characteristics for the system calls in the test program’s iterative loop.

```
<table>
<thead>
<tr>
<th>System Call</th>
<th>Number of Calls</th>
<th>Most Common Path</th>
<th>Freq.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>20201</td>
<td>636</td>
<td>6057</td>
<td>30.13%</td>
</tr>
<tr>
<td>write</td>
<td>20132</td>
<td>768</td>
<td>11748</td>
<td>58.35%</td>
</tr>
<tr>
<td>lseek</td>
<td>40200</td>
<td>33</td>
<td>40164</td>
<td>99.91%</td>
</tr>
<tr>
<td>fsync</td>
<td>20100</td>
<td>16119</td>
<td>36</td>
<td>0.18%</td>
</tr>
<tr>
<td>nanosleep</td>
<td>20100</td>
<td>177</td>
<td>14189</td>
<td>70.59%</td>
</tr>
<tr>
<td>rt_sigaction</td>
<td>20100</td>
<td>22</td>
<td>20078</td>
<td>99.89%</td>
</tr>
<tr>
<td>rt_hupmask</td>
<td>40200</td>
<td>81</td>
<td>40100</td>
<td>99.75%</td>
</tr>
</tbody>
</table>
```

Table 2: Path Characteristics of the data set

Looking at the iterative loop of our test program, we identified 19,505 unique paths (i.e. the combine paths of the system calls in the body of the program’s loop), with the occurrence frequencies as depicted in Table 3.

```
<table>
<thead>
<tr>
<th>Frequency</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instances</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>17</td>
<td>61</td>
<td>342</td>
<td>19069</td>
<td></td>
</tr>
</tbody>
</table>
```

Table 3: Paths in the Program’s Iterative Loop
5.4 Discussion

In this section, we discuss the results presented above, and provide an interpretation in the context of IEC 61508, with respect to the identification of possible failure mechanisms in software.

Since we are addressing a generic software component for use in a safety-related system we do not define the specific safety function of the component. Rather, we assume that a safety application compliant to IEC 61508 can call upon a particular function in the operating system which, if faulty, has the potential to seed a fault. A conservative approach is thus taken in this work, and each residual fault needs to be considered safety related. As it is difficult to identify each and every fault in a complex software component we seek a generic rather than a specific mitigation.

5.4.1 Variability in the execution path

The experimental results achieved show that there is both temporal and path variability of a system call from repeated execution of a program. Of interest to our work is the variability in the execution path.

In a large set of runs, we can identify several execution path through the kernel space. From the point of view of the invoking application, these are functionally equivalent paths, but internally and transparent to the application, they exhibit diversity. We can attribute the path diversity to the internal state of the operating system, which is different every time a system call is invoked.

For the read system call, there are a total of 636 distinct paths from the 20,101 system call instances. 591 or 92.92% of the path were taken by only one system call instance. This represents only 2.94% of the total number of system calls considered, which represents a small fraction of the total number of calls. These unique instances mainly consist of those system calls that were interrupted by asynchronous events (Okech et al., 2013). We generally expect that with a high system load, the proportion of these paths would increase.

The remaining 19,510 instances took one of the other 45 execution paths. We can view these paths as being in two groups, frequently taken paths and rare paths.

Consider the most frequent path taken in this execution run. It has a frequency of 6,057, meaning that 30.13% of the system calls did take this path. If we consider the two most common paths, we find that these paths were taken by 10,968 system call instances, representing 54.56% of the total. Similarly the top three and four paths are taken by 13,682 (68.03%) and 15,917 (79.19%) respectively. The top ten most frequently taken paths are 92.32% of the total read system calls. We present the cumulative frequencies of the top 14 paths of the read system calls in Table 4. Not shown in the table are paths (1007) with a frequency of less that 95.

The results obtained gives us confidence that we can classify paths into two groups - frequently taken paths and rare paths - for any non-trivial system call. From our experiment, the lseek system call illustrates the properties of what can be referred to as trivial system call. Out of the 40,200 instances, 40,164 of these take the same path (see Table 2). Due to the low variance in its paths, the lseek system call can be well tested.

We contend that with a conservative test coverage of about 80% for non-safety critical systems, a test campaign on complex systems within a reasonable budget will only exhaustively test the most commonly taken paths. We believe that it is not feasible using current techniques to write test cases to exercise all possible paths in a complex system such as the Linux kernel. This situation leads to a very high possibility that the rare paths are not reasonably well tested, and hence will contain residual systematic faults. Inherent diversity as presented in this paper is attractive for addressing this issue as it allows arguing a generic mitigation of residual systematic faults, that, given the overall complexity of the Linux kernel could not be covered analytically or by testing with economically tolerable effort.

5.4.2 Correlation of paths

The main premise of our work to assume independence in the taking of the paths in the rare paths set. Therefore an important issue is to find out whether the kernel paths taken by a system call execution displays an identifiable patterns over time. In this section, our analysis focuses on the read system call only.

We first looked at the relationship between the system call instances and the path taken. A snapshot of the plot of the instances against the index of the path taken is shown in Figure 2.

To find out if the next path taken by a system call instance is dependent on the previous paths i.e. are the path taken conditioned on path history?, we performed tests on the data using the Autocorrelation Function (ACF).

The autocorrelation coefficient of a series of values lies in the range [+1,-1] with values closer to these limits signifying strong correlation, and therefore indicating departures from the assumption of independence. If a series is uncorrelated, then the expectation is that the autocorrelation coefficient will be close to 0. The plot of the autocorrelation coefficients for the read system call is shown in Figure 3.

In our data, the highest coefficient has a magnitude of 0.1536 with the values becoming much smaller as the lags gets larger. Additionally, only 403 (or 8.02%) coefficients fall outside the 95% confidence interval. Due to the low magnitudes of the autocorrelation coefficients, we can safely make the assumption that the paths are independent.

---

<table>
<thead>
<tr>
<th>Top N</th>
<th>Number of Paths</th>
<th>Cumulative Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6057</td>
<td>6057</td>
</tr>
<tr>
<td>2</td>
<td>4911</td>
<td>10968</td>
</tr>
<tr>
<td>3</td>
<td>2714</td>
<td>16682</td>
</tr>
<tr>
<td>4</td>
<td>2242</td>
<td>19924</td>
</tr>
<tr>
<td>5</td>
<td>1018</td>
<td>10942</td>
</tr>
<tr>
<td>6</td>
<td>466</td>
<td>17308</td>
</tr>
<tr>
<td>7</td>
<td>450</td>
<td>17858</td>
</tr>
<tr>
<td>8</td>
<td>306</td>
<td>18164</td>
</tr>
<tr>
<td>9</td>
<td>271</td>
<td>18375</td>
</tr>
<tr>
<td>10</td>
<td>182</td>
<td>18557</td>
</tr>
<tr>
<td>11</td>
<td>152</td>
<td>18709</td>
</tr>
<tr>
<td>12</td>
<td>145</td>
<td>18857</td>
</tr>
<tr>
<td>13</td>
<td>142</td>
<td>18999</td>
</tr>
<tr>
<td>14</td>
<td>95</td>
<td>19094</td>
</tr>
</tbody>
</table>

Table 4: Path Frequency Table
The key issue for us is to show statistical independence in an execution taking one of the rare paths. At the current point we are in our work, we have not yet identified a suitable method to isolate the rare path set, for which tests for independence can be performed.

6 Safety Justification based on Inherent Diversity

Developers of software-based safety-related systems have several available routes to certification of their products. The first and the one with the highest chance of success is following the prescribed process: specify requirements, develop, verify and operate. A second option is provide a proven-in-use argument for the software incorporated in the safety system if the component has not been developed from scratch. There exists a third option if a pre-existing software is used. In this later case, the focus is on the selection process and assessment of available life-cycle data.

The option to select off-the-shelf software has many challenges. There are several issues to be considered if and when a developer decides to use software designed and developed without having functional safety as one of the primary goals. However, there are alternative strategies that could be used to demonstrate safety of such systems.

We suggest an approach justifying safety-related systems based on the inherent diversity. There are two aspects of the strategy:

- Safety argument
- Procedure for assessment of safety

6.1 Safety Argument

The first issue to consider is the validation process of the GNU/Linux kernel. In commercial software development, detailed specification are made and then used to perform validation of the final product. On the other hand, in the Linux development model, the development process and the traceability are informal. Though the kernel is developed in a controlled manner by a high quality development team with a well thought-out development and maintenance process, no (semi-)formal specification of the kernel is available.

One of the most important aims of Linux is the compliance to the POSIX (Institute of Electrical and Electronics Engineers 1988) specification. There exists a semi-formal specification in the form of the POSIX interface specification which can be used to validate the kernel. This is already being done by the Linux Test Project (LTP) (LTP 2013). The project provides a test harness that can be used in a systematic way to run the POSIX test suite against the Linux kernel, thus validating the kernel against the specification to demonstrate that it meets its stated requirements.

However, no amount of testing will result in a complete coverage of a complex system such as the GNU/Linux kernel. It is infeasible to write test cases that will cover the entire possibility of execution paths in the kernel. There is a potential that some residual faults remain in those paths that have not been exhaustively tested. We need to provide protection against these residual systematic faults. We suggest a mitigation though combination of architectural means and inherent diversity.

Safety Risk There is a safety risk associated with the residual faults that remain in the operating system. This could take the following form:

a correct and valid input vector from an application submitted to the operating system through the POSIX interface invokes a system call, but the request made took an execution path that was not covered by the tests that were done during the validation process.

We take the position that for some paths a test case could not be provided, thus these paths belong to the rarely taken path set. Further, we know that an application’s input vector cannot deterministically cause a rare path to be taken. If this was possible, then the it would have been covered by a test case.

Assuming that taking the rare path is a statistically independent event then we are essentially taking residual systematic faults in complex software and transforming the effective behavior of that fault on the system to appear as a random fault based on inherent non-determinism. This transformation can only be possible if we design an appropriate architecture for the overall system, say as a minimum a 2oo2 system.
The probability of both channels entering the same faulty rare path at the same time is sufficiently low. The residual faults can be regarded as random faults and is sufficiently protected by architectural means of a 2oo2 system. The claim is that if the taking of a rare path is a random event based on the inherent diversity of the system, then the systematic faults in these paths can be protected by architectural mechanism in the same way the architecture protects against truly random faults.

The significant threat to the correctness of the system is if the residual faults present in the untested paths are not statistically independent. In this case, these represent common cause faults, which would require further analysis.

The overall approach we use to justify safety is illustrated by the diagram of Figure 4.

![Figure 4: Summary of the safety strategy](image)

6.2 Procedural Perspective

The counterpart to the justification of safety above is the approach to quantification of safety levels achievable through this mechanism. The main goal of the approach is to establish a firm process with sufficient level of assurance so that we can quantify the independence of the paths. This has to be by a reasonable reproducible and sufficiently assured process. This quantification can then be used to do an assessment of a 2oo2 system based on inherent diversity. This is the focus of our ongoing work.

7 Related Work

With respect to the method used to show the diversity in path execution, the SIL4Linux project (Wang et al. 2009) is closest to our work. While one of their main aims was to check the real-time behavior of the Linux kernel in the context of safety, we are more interested in determining the variability in the behavior of specific system calls during its execution.

Similar to the goal of our work is one of the stated research objectives of the INDEXYS project (Eckel et al. 2010) to investigate how inherent diversity of complex operating systems helps in detecting faults in computing platforms. The project targets to instantiate platforms such as the TAS Control Platform (Gerstinger et al. 2008), which employs architectural protection schemes to mask and detect random faults through temporal relaxation. The loose coupling is based on the assumption that the channels are temporally independent so that any fault activation in one of the channel is detectable. This is building on the non-deterministic property of loosely coupled systems, though our work focuses on the path non-determinism in addition the non-determinism in the temporal dimension.

8 Conclusion

We believe that functional safety should be based on confidence and assurance not on prescribed procedures and processes. Technology in the area of programmable electronics has been undergoing fundamental changes in the past decade. At the same time the complexity of system software that support execution of user applications is growing. These changes at the "foundations" of safety-related systems will need to be addressed by the safety community. It is our conviction that diversity in the software stack is a key methodology in tackling the challenges of ensuring the dependability of safety related systems.

Though the core concepts of diversity are established in safety engineering, the traditional approaches to achieving diversity are reaching their limits, both technically and economically. It is on this premise that we propose the concept of inherent diversity. Establishing inherent diversity as a sound methodology for safety-related systems will need considerable more than the proof-of-concept level we have so far engaged in. The inherent diversity approach presented in this paper is by no means a stand-alone argument that can resolve the assurance demands of safety-related systems verification utilizing complex hardware/software but it is, in our opinion, a radically new approach to diversity that shows a promising potential to play a role in enabling complex software, like the GNU/Linux operating system, for safety related systems.

References


