4

Analysis of Error Propagation Between Software Processes

Sizarta Sarshar
Institute for Energy Technology
Norway

1. Introduction

All software systems can contain faults. In critical systems, this problem is alleviated by controlling the possible effects of a fault being executed, typically through techniques for achieving fault tolerance. Ensuring that failures are properly isolated, and not allowed to propagate, is essential when developing critical systems.

In much of the research on error propagation analysis the focus has been on probabilistic models. While these models are well suited for quantitative analysis, they are usually not very specific with regard to the actual mechanisms that might allow a failure to propagate between entities. Quantitative analysis is often applied on code level and not seen as influenced by and in conjunction with the operating system. A more detailed insight into the actual mechanisms can be beneficial to decide whether or not error propagation is a concern for a given source code.

A method for studying mechanisms of error propagation between software processes was proposed in (Sarshar, 2007). This chapter describes the method, which (1) facilitates the study of error propagation between software processes; (2) identifies mechanisms for error propagation; and (3) provides means to determine whether these can be automatically detected by a static analyser. In this context a process represents a program in execution, typically managed by an operating system. Processes can communicate with each other via inter-process communication and their shared resources. Examples of shared resources can be the operating system itself and the memory. The analysed problem is how one process can cause another process to fail and concerns interaction methods available in the source code of a program. The work criteria and scope are described in the following:

- *Consider processes running on a single CPU computer with an operating system.*
- *The method should only require the source code and minimal manual input to work.*
- *The source code must compile without any errors prior to the analysis.*
- *The primary interest is to determine whether error propagation is a concern or not.*

This chapter further reports on the applicability of the method in a case where a module of a core surveillance framework named SCORPIO has been analysed. The framework is a support system for nuclear power plants supporting monitoring and prediction of core conditions.

Some of the terminologies used in this chapter are briefly described in the following (Storey, 1996):

- A fault – is a defect within the system.
- An error – is a deviation from the required operation of the system or subsystem.
- A system failure – occurs when the system fails to perform its required function.

This chapter is structured as follows: Section 2 gives a definition of error propagation, describes the mechanisms of error propagation, and previous work on the topic. Section 3 describes the proposed method for studying error propagation between software processes. Section 4 reports on the applicability of the method on one module of the SCORPIO framework. Section 5 addresses the main results. Section 6 discusses the work while section 7 provides conclusions and comments on future work.

2. Background

This section gives a definition of error propagation, describes the mechanisms of error propagation, operating systems and related work on the topic.

2.1 Error propagation

In our work, error propagation is defined as the situation where an error (or failure) propagates from one entity to another (Sarshar et al., 2007). Errors can propagate between different types of entities, including: physical entities, processes running on single or multiple CPUs, data objects in a database, functions in a program, and statements in a program. Our approach concerns propagation of errors between processes running on a single CPU computer.

Systems of interest in our work have not been limited to those that are safety critical only, e.g. systems that are directly involved in controlling a nuclear reactor. A problem of particular interest is the possible negative effect a low criticality application might have on a higher criticality application by means of error propagation because they share common resources.

Programs make use of interaction methods provided by the underlying operating system to communicate with each other, or make use of shared resources. These services are provided through the system call interface of the operating system, and are usually wrapped in functions available using standard libraries. Such interaction methods can cause errors and provide mechanisms for error propagation. A coding fault which may be manifested as an error may in principle be anything, e.g. an incorrect instruction or an erroneous data value. It may be manifested inside a local function or an external function. The propagated error need not be of the same type in different functions, e.g. an instruction error in one function realization causes a data error in another. Even if an error is propagated to one function, this does not necessarily mean that the source function fails functionally. The propagated error may only be a side-effect in this function. Another type of error related to function usage is error caused by passing illegal arguments to functions or misusing their return variables. Error propagation between two programs may occur even if both programs individually operate functionally correct. This can e.g. be caused by erroneous side effect in the implementation or execution of the programs. There are two situations possible for how one process can cause another process to fail:

- One process experiences a failure, which then causes another process to fail.
- One process propagates a fault to another process while not failing itself.

According to (Fredriksen & Winther, 2007), possible ways of characterizing error propagation is as either intended or unintended communication or as resource conflicts.

Error propagation in intended communication channels might consist of erroneous data transfer through parameters or global variables. Writing to the wrong addresses in memory, due e.g. to faulty pointers, exemplifies error propagation through unintended channels. Processes that demand high processor load so that other processes cannot execute are examples of resource conflicts which could cause error propagation. This indicates that error propagation between functions can occur in at least two ways:

- An error in one function is transferred via a communication channel to another function, for example through passing of arguments or return value.
- The execution of one function interacts with another function in an unintended and incorrect way, due to an error, and causes the second function to fail.

Thus error propagation can take place via the intended communication channels, i.e. those that are used by the set of functions to fulfil their tasks. It is also possible that an error in one function generates a communication channel that is not intended and propagates the error through this.

2.2 Operating systems

The references (Nutt, 2004; Bacon & Harris, 2003; Bic & Shaw, 2003; Tanenbaum & Woodhull, 2006; Stallings, 2005) cover the basic principles of a number of important operating systems.

With respect to the Linux operating system and its kernel, one source to its understanding is given in (Bovet & Cesati, 2003). Here, the authors describe the kernel components from how they are built to how they work. (Beck et al., 2002) explains what is in the kernel, and how to write kernel code or a kernel module. The work in (Bic & Shaw, 2003) explains process management and interaction in the UNIX operating system, and in (Pinkert & Wear, 1989), the authors describe all major components of an operating system down to the pseudo code level. The authors employ a generic approach and present the fundamental concepts involved, alternative policies from which a designer can choose, and illustrative mechanisms for implementing selected policies.

In (Kropp et al., 1998), the Ballista methodology is applied on several implementations of the POSIX operating system C language API. The methodology is for automatic creation and execution of invalid input robustness tests designed to detect crashes and hangs caused by invalid inputs to function calls. The Ballista POSIX robustness test suite was ported to ten operating systems where even in the best case, about half of the functions had at least one robustness failure. The results illustrate that error propagation is a concern in operating systems.

A study of operating system errors found by automatic and static compiler analysis applied to the Linux and OpenBSD kernels is reported in (Chou et al., 2001). Static analysis is applied uniformly to the entire kernel source. The scope of errors in the study is limited to those found by their automatic tools. These bugs are mostly straightforward source-level errors. They do not directly track problems with performance, high-level design, user space programs, or other facets of a complete system. (Engler et al., 2000) examines features of operating system errors found automatically by compiler extensions. Some of the results they present include the distribution of errors in the kernel: the vast majority of bugs are in drivers.

Our approach focuses on analysing user space programs. We examine how the operating system manages processes and provides services to user programs through the system call interface, but we do not analyse its code. We assume that the operating system performs its

intended functions correctly and that it is implemented correctly. Instead, we analyse the system call interface and other process interaction mechanisms to identify whether these may cause error propagation.

2.3 Related work

Error propagation analysis has to a large extent been focused on probabilistic approaches (Hiller et al., 2001, Jhumka et al., 2001; Nassar et al., 2004; Abdelmoez et al., 2004) and model based approaches (Voas, 1997; Michael & Jones, 1997; Goradia, 1993).

In (Hiller et al., 2001), the concept of error permeability is introduced as a basic measure upon which a set of related measures is defined. These measures guide the process of analysing the vulnerability of software to find the modules that are most likely to propagate errors. Based on the analysis performed with error permeability and its related measures, how to select suitable locations for error detection mechanisms (EDMs) and error recovery mechanisms (ERMs) are described. Furthermore, a method for experimental estimation of error permeability, based in fault injection, is described and the software of a real embedded control system analysed to show the type of results obtainable by the analysis framework. The results show that the developed framework is very useful for analysing error propagation and software vulnerability, and for deciding where to place EDMs and ERMs.

The paper (Jhumka et al., 2001), assess the impact of inter-modular error propagation between embedded software systems. They develop an analytical framework which enables to systematically design software modules so the inter-modular error propagation is reduced by design. The framework is developed using influence and separation metrics, then the framework is validated using fault injection experiments, which artificially inject faults and errors into the system. Influence metric is in their paper referred to as the probability of a module directly influencing another module, i.e., when no other module is considered while separation metric is referred to as the probability of a module not influencing another one when all other modules are considered. The results showed that the analytical framework can predict the influence value between a pair of modules very accurately.

The study of software architectures is an important discipline in software engineering, due to its emphasis on large scale composition of software products, and its support for emerging software engineering paradigms such as product line engineering, component based software engineering, and software evolution. Architectural attributes differ from code-level software attributes in that they focus on the level of components and connectors, and that they are meaningful for architecture. In (Abdelmoez et al., 2004), focus is on a specific architectural attribute, which is the error propagation probability throughout the architecture, e.g. the probability that an error arising in one component propagates to other components. Formulas for estimating these probabilities using architectural level information are introduced, analysed, and validated.

In (Voas, 1997), error propagation between commercial-off-the-shelf (COTS) components is analysed using an approach termed interface propagation analysis (IPA). IPA is a fault-injection based technique for injecting 'garbage' into the interfaces between components and then observing how that garbage propagates through the system. An example, if component A produces information that is input to component B, then the information is corrupted using fault injection techniques. This simulates the failure of component A. After this corrupt information is passed into B, IPA analyses the behaviour of B (or components

executed after B) to the information. IPA analyses the behaviour of a component by looking for specific outputs that the user wants to be on the lookout for.
(Michael & Jones, 1997) presents an empirical study of an important aspect of software defect behaviour: the propagation of data-state errors. A data-state error occurs when a fault is executed and affects a program's data-state, and it is said to propagate if it affects the outcome of the execution. The results show that data-state errors appear to have a property that is quite useful when simulating faulty code: for a given input, it appears that either all data state errors injected at a given location tends to propagate to the output, or else none of them do. These results are interesting, because of what they indicate about the behaviour of data-state errors in software. They suggest that data state errors behave in an orderly way, and that the behaviour of software may not be as unpredictable as it could theoretically be. Additionally, if all faults behave the same for a given input and a given location, then one can use simulation to get a good picture of how faults behave, regardless of whether the simulated faults are representative of real faults.
Goradia (Goradia, 1993) addresses test effectiveness, i.e. the ability of a test to detect faults. This thesis suggests an analytical approach, introducing a technique of dynamic impact analysis using impact graphs to estimate the error propagation behaviour of various potential sources of errors in the execution. The empirical results in the thesis provide evidence indicating a strong correlation between impact strength and error propagation. The time complexity of dynamic impact analysis is shown to be linear with respect to the original execution time and experimental measurements indicate that the constant proportionality is a small number ranging from 2.5 to 14.5. Together these results indicate that they have been fairly successful in their goal of designing a cost effective technique to estimate error propagation. However, they also indicate that to reach the full potential benefits of the technique the accuracy of the estimate needs to be improved significantly. In particular, better heuristics are needed for handling reference impact and program components tolerant to errors in control paths.
Research on error propagation has identified frameworks and techniques for estimating error propagation, e.g. in (Jhumka et al., 2001; Goradia, 1993). In difference, our goal is to identify sources and mechanisms for error propagation in order to identify potential error propagation scenarios and remove the failures to improve software.

3. Method of analysis

A method for analysing the interfaces between processes and their shared resources in the search for mechanisms for error propagation is provided in (Sarshar, 2007; Sarshar et al., 2007). This section describes this method which starts out by investigating how processes are managed in the relevant operating system, enabling us to identify process characteristics relevant to error propagation. The output of this step includes a list of system calls in the system call interface of the operating system. Secondly, the identified interaction methods are analysed using Failure Mode and Effect Analysis (FMEA) (Stamatis, 1995). This approach helps to identify types of code characteristics that might be a concern in relation to error propagation. The method of analysis can be summarized in three steps:

1. Examination of the operating system for how it interacts with and manages processes to obtain an overview of e.g. a list of system calls and common resources;
2. Analysis of the interaction methods using Failure Mode and Effect Analysis (FMEA) to identify possible faults that can cause error propagation to occur; and

3. Determination of how the mechanisms can be recognized in source code.

The method was developed for C code under the Linux operating system as a case. C was chosen because it is a widely used programming language and Linux because it is an open source operating system. In section 4, the method is applied on one module of the SCORPIO framework.

3.1 How processes run in operating systems

Processes are managed by the operating system. An operating system provides a variety of services that programs can utilise using special instructions called system calls. The typical functions of an operating systems kernel are: process management, memory management, input and output management, and support functions. In Linux, the kernel components managing processes are the following:

- Signals: the kernel uses signals to call into a process.
- System calls (explained below).
- Process manager and scheduler: creates, manages and schedules processes.
- Virtual memory: allocates and manages virtual memory for processes.

A process interface to the operating system is either a result of the use of system calls or through direct memory access. Use of a pointer in the C language is an example of accessing memory without the use of the system call interface. In Linux, system calls are implemented in the kernel. When a program makes a system call, the arguments are handled in the kernel, which takes over the execution of the program until the call completes (Mitchell et al., 2001). System calls are usually wrapped in the standard C library and may require some parameters and return a value. Examples of system calls are low-level input and output functions, such as **open**() and **read**(). The system calls of Linux can be grouped into the following categories (Silberschatz et al., 2005; Bic & Shaw, 2003):

- Process management: create/terminate process, load, execute, end, abort, get/set process attributes, wait for time, wait/signal event, allocate and free memory.
- File management: create/delete file, open, close, read, write, reposition, get/set file attributes.
- Device management: request/release device, read, write, reposition, get/set attributes, logically attach or detach device.
- Inter-process communication: the transfer of data among processes.
- Communications: create, delete connection, end, receive messages, transfer status information, attach or detach remote device.
- Miscellaneous services: get/set time or date, system data.

The essence of our approach is to identify mechanisms for error propagation that have characteristics detectable when analysing source code. We can therefore narrow down our scope to include those parts of the operating system which fulfil this requirement. The kernel components that allow interaction directly in source code of a program include the system call interface and signals. Language specific traps and pitfalls (Hatton, 1995; Koenig, 1989) might also open ways for an error to propagate. Programming errors can give variables incorrect values that can lead to failures. Our analysis does not specifically address general programming errors, but errors related to invoking system calls.

We focus here on programs written to run in user space, and exclude programs written for kernel space, as they have their own kernel API which provide services for kernel programming.

Figure 1 shows a simple illustration of the channels available in source code of a program for interaction with the operating system and its resources. These include the system call interface, signals, and traps and faults, with arrows indicating the interactions.

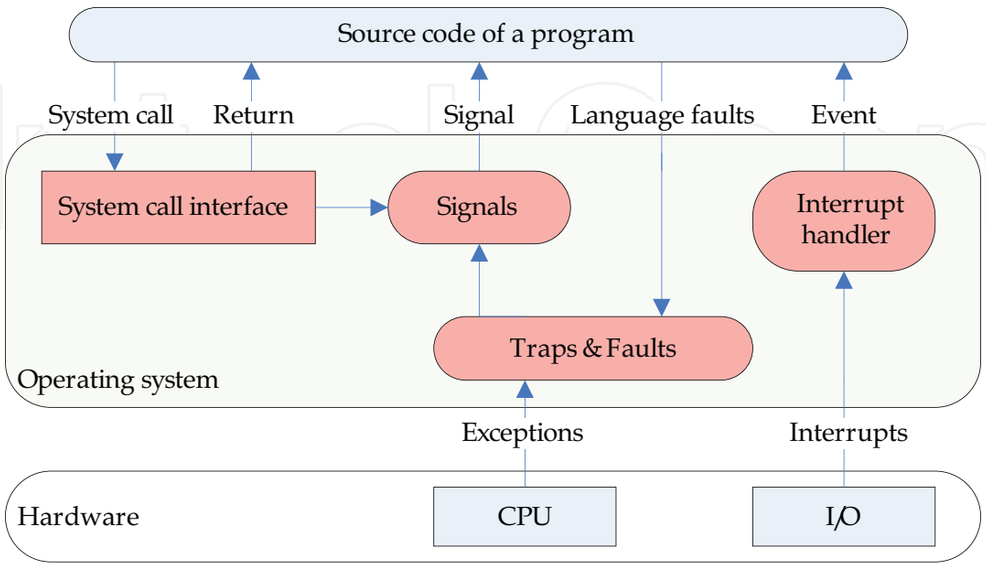


Fig. 1. Illustration of the interaction methods of the operating system on processes

An interrupt is a condition that can cause the normal execution of instructions to be altered. Interrupts and exceptions are known as signals and are used to notify a process of certain faults by the kernel (Pinkert & Wear, 1989):

- Completion of an input or output operation.
- Division by zero.
- Arithmetic overflow or underflow.
- Arrival of a message from another system.
- Passage of an amount of time.
- Power failure.
- Memory parity error.
- Memory protect violation.

A signal might also be altered from another program using the system call interface.
In source code, interaction with the operating system is only available through the system call interface. It is therefore not necessary to examine how processes are handled and managed at deeper levels.

3.2 Identify system call failures causing error propagation

In the proposed method, each system call is analysed using FMEA. The purpose is to identify failure modes that can cause errors to propagate to other processes or the operating system. The focus in this analysis is on failure modes that have characteristics in the source code of a program.
FMEA is a well-known analysis method for risk and reliability analysis. The basis for this analysis is a description of a system in terms of its components and the communication between them. For each of the components in the system, the aim is to identify all potential

modes of failure, by investigating the following questions for each component and communication unit, based on the FMEA framework:

- What can go wrong? (failure mode)
- How can this occur? (failure cause/mechanism)
- Which consequences will this have on the further actions and messages? (failure effects via error propagation)

In our method, the FMEA is targeted on the system call as a component and the focus is on its usage in source code of a program. Once the failure modes have been identified, we determine their potential effects on local and system processes to determine whether any of these can cause error propagation. This can be done in two ways:

- The effect is described in the system call documentation as an error the function can return.
- The effect is determined using fault injection in test programs.

The failure effects will provide information on the severity of failures and help us provide possible mitigation actions.

3.3 Identify the failure mode characteristics in source code

The aim of step three of the method is to determine whether the failure modes identified in the previous step are present in the source code of a program. For each failure mode that can cause error propagation, we determine its characteristics in code so it can be detected when analysing an application's source code. We then examine some existing code analysis tools to check whether any of these will recognise the failure modes, and if they do, determine whether they identify all of them. The next step is to develop an algorithm for identifying the failure modes in source code, including how to traverse and check the code for the identified failures. The result is a prototype tool which demonstrates that failures causing error propagation can be detected by analysing source code.

The steps of the method are performed only once for an operating system and programming language combination. The prototype tool is run for each application source code we wish to analyse for error propagation.

4. Case on SCORPIO

SCORPIO (Surveillance of reactor CORe by Picture On-line display) is a core surveillance framework for nuclear power plants, and is developed at the Institute for Energy Technology (IFE). The framework is a support system for the monitoring and prediction of pressurized water reactors (PWR), boiling water reactors (BWR) and VVER (Russian design series of PWRs) core conditions and is running on several reactors worldwide (Barmsnes et al., 1997). The framework has passed established system tests including factory acceptance testing and site acceptance testing.

The general SCORPIO framework is illustrated in Figure 2. The module administrator is a program that connects the modules to the graphical user interface made using ProcSee (IFE, 2010). ProcSee is a versatile software tool for developing and displaying dynamic graphical user interfaces, particularly aimed at process monitoring and control. All data exchanged between the modules and the operator is transmitted through this program. The Software Bus handles the communication between all modules. In the case study, the input data processing (IDATP) module of the framework has been assessed. The IDATP module consists of 30 files and approximately 5300 lines of code.

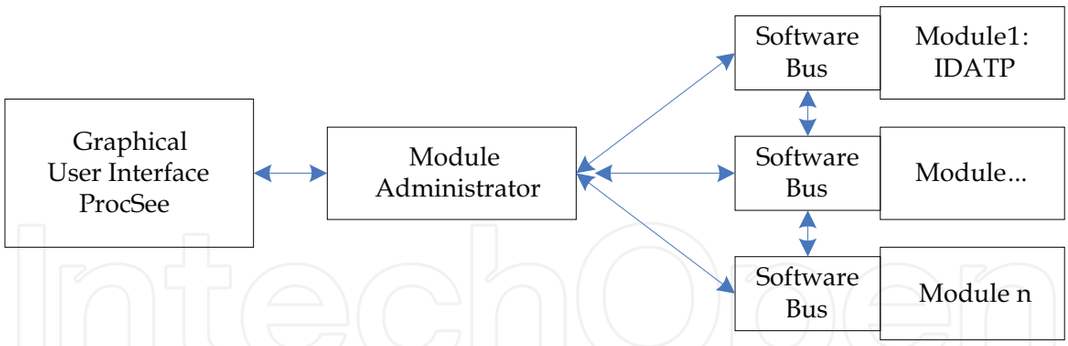


Fig. 2. The general SCORPIO framework

The source code of the IDATP module is first examined to identify which calls it performs to system and library functions. The attributes passed to these external functions and the values retrieved are stored for later analysis.

Function	**System call**	**Library call**	**Description**
close	x		Close a file descriptor
execvp		x	Execute file
fclose		x	Close a stream file
fopen		x	Open a stream file; convert file to stream
fprintf		x	Formatted output conversion to a given stream
fscanf		x	Input format conversion
memcpy		x	Copies *n* bytes from memory area *source* to memory area *destination*
memset		x	Fill memory with a constant byte
pipe	x		Creates a pair of file descriptors
printf		x	Formatted output conversion to standard out stream
shmget	x		Allocate a new shared memory segment
signal	x		Signal handling
sprintf		x	Formatted output conversion to a given character string
sscanf		x	Input format conversion
strcat		x	Concatenate two strings
strcmp		x	Compare two strings
strlcpy		x	Copy string
strlen		x	Calculate length of string
strncmp		x	Compare two strings

Table 1. Analysed system and library calls

4.1 Applying the analysis

Each system and library function of the IDATP module is analysed using FMEA with focus on identifying failure modes that can cause the module or the system itself to encounter failure. A failure mode specifies how an entity may fail. An entity may be e.g. a variable, used as either an argument passed to a function or used as a return variable.

The system manuals for these calls form the basis for this analysis. The IDATP module makes use of several system and library calls. A subset of 19 of these functions, listed in Table 1, were analysed using FMEA.

To exemplify the analysis, we focus on the **shmget**() system call to demonstrate the usage of the method in the following. Thus emphasis is on the steps involved in performing the analysis and understanding the analysis object.

The **shmget**() system call creates or allocates a new shared memory segment for inter-process communication (IPC) between processes. This IPC provides a channel for communication between processes using the memory. The main services related to shared memory are **shmget**(), **shmat**(), **shmctl**(), and **shmdt**(). Other calls related to shared memory include services for managing semaphores. The relation between these calls are as follows: A process starts by issuing a **shmget**() system call to create a new shared memory with the required size. After obtaining the IPC resource identifier, the process invokes the **shmat**() system call, which returns the starting address of the new region within the process address space. When the process wishes to detach the shared memory from its address space, it invokes the **shmdt**() system call.

We begin with an examination of the system call documentation and then perform FMEA on the function. When performing the analysis, the aim is to identify failure modes caused by wrong usage of the service in source code, and determine their effects on local and system processes. The focus is on those failure modes causing error propagation.

The synopsis for the **shmget**() function:

```
# include <sys/types.h>
int shmget(key_t key, size_t size, int shmflg);
```

The **shmget**() function returns the identifier of the shared memory segment associated with the value of the argument *key*. A new shared memory segment, with size equal to the value of size rounded up to a multiple of PAGE_SIZE, is created if:

- key has the value IPC_PRIVATE, or
- key is not IPC_PRIVATE, no shared memory segment corresponding to *key* exists, and IPC_CREAT is specified in *shmflg*

PAGE_SIZE, IPC_PRIVATE and IPC_CREAT are definitions within the operating system.

IPC_PRIVATE is not a flag field but a *key_t* type. If this special value is used for *key* the system call ignores everything but the least significant 9 bits of *shmflg* and creates a new shared memory segment, on success. The value of *shmflg*'s least significant 9 bits specify the permission *mode*, the permissions granted to the owner, group, and world.

The FMEA process starts with identifying failure modes. Table 2 illustrates identified failure modes for the *shmflg* parameter of **shmget**(). This is an excerpt from the complete FMEA sheet for this function.

For each identified failure mode, we now examine its effects on the process itself (indicates "local effect" in the FMEA sheet) and on other processes (indicates "system effect" in the FMEA form). Some of these failure modes are detected by the system call; the function exits

with return value equal to -1, which indicates an error, and the external variable *errno* is set appropriately. Many of these are described in the manual pages and can be identified as the failure effect on the local process. However, not all failure modes are represented as error cases in the manual pages. We make use of test programs to identify these.

Reference	Variable	Failure mode
F.29.3.A	Parameter *shmflg*	Not specified at all
F.29.3.B	Parameter *shmflg*	Is not one of IPC_CREAT, IPC_EXCL, SHM_HUGETLB or SHM_NORESERVE
F.29.3.C	Parameter *shmflg*	Is of wrong type
F.29.3.D	Parameter *shmflg*	No permission *mode* is set
F.29.3.E	Parameter *shmflg*	Access permission is given to all users, instead of user only
F.29.3.F	Parameter *shmflg*	Permission *mode* is write when it should have been read
F.29.3.G	Parameter *shmflg*	Permission *mode* is read when it should have been write
F.29.3.H	Parameter *shmflg*	Permission *mode* is set without user access
F.29.3.I	Parameter *shmflg*	IPC_EXCL specified without IPC_CREAT
F.29.3.J	Parameter *shmflg*	Wrong flag specified i.e. IPC_CREAT | IPC_EXCL when not intended

Table 2. Failure modes for parameter *shmflg* for the **shmget**() system call

A test program is written to execute a failure mode while the failure effect is monitored. Such test programs have the possibility to execute an injected failure mode.
Based on such test programs one can determine the effect of failure modes. E.g. the effect for failure mode F.29.3.D ”no permission mode is set” was determined to be: *no processes can access the shared memory segment unless they are privileged*. Checking the value of the parameter *shmflg* to identify whether the permission mode is set is easily done performing static analysis, thus this failure mode can be detected in source code.
Table 3 shows the complete FMEA for the failure modes related to the *shmflg* parameter of **shmget**() from Table 2.
Similarly, the remaining system and library calls are analysed. The failure modes identified in the analysis of these calls are related to passing of arguments and handling return values, and can be grouped as follows:

- Argument refers to uninitialized variable/pointer.
- Argument is of different type than specified in function definition.
- Argument refers to null-pointer.
- Argument is freed.
- Argument refers outside an arrays size.
- Argument is an array of chars which is not null-terminated when required.
- Return value is not retrieved from a non-void function.
- Return value is not checked to determine successful call.
- Return value is not used in scope.

These failure modes are then compared with the checks that existing tools perform to determine whether any of these are present in their checks.

Ref.	Failure mode	Local effect	System effect	Conclusion
F.29.3.A	Not specified at all	-	-	Does not compile
F.29.3.B	Is not one of IPC_CREAT, IPC_EXCL, SHM_HUGETLB or SHM_NORESERVE	Unknown flag and permission is set	Segment may not be created or accessed	Detectability in source code must be determined
F.29.3.C	Is of wrong type	Uses the *int* value of the type if possible, unknown flag and permissions are set on segment	Segment may not be created or accessed	Detectable in source code
F.29.3.D	No permission *mode* is set	The process cannot access the shared memory segment unless it is run in privileged mode	Other processes cannot access the shared memory segment unless they are run in privileged mode	Detectable in source code
F.29.3.E	Access permission is given to all users, instead of user only	-	Other users can access the shared segment	Detectability in source code must be determined
F.29.3.F	Permission *mode* is write when it should have been read	Can write to segment when not intended	Other processes can write to segment when not intended	Detectability in source code must be determined
F.29.3.G	Permission *mode* is read when it should have been write	Cannot write to segment	Other processes cannot write to segment	Detectability in source code must be determined
F.29.3.H	Permission *mode* is set without user access	The process cannot access the shared segment unless it is run in privileged mode	-	Detectable in source code
F.29.3.I	IPC_EXCL specified without IPC_CREAT	Exits with error if segment already exists	-	Detectable in source code
F.29.3.J	Wrong flag specified i.e. IPC_CREAT | IPC_EXCL when not intended	Tries to create instead of getting identifier for the shared segment	-	Detectability in source code must be determined

Table 3. Example of FMEA for the parameter *shmflg* of the **shmget**() system call

4.2 Analysis tools

There are several existing analysis tools which identify different types of errors. These tools include both static and dynamic analysis methods. In (Sarshar, 2007), over 20 tools were examined and compared to determine what kind of errors they detect. Of these tools, one group performs checks on passing of arguments, another group warns if a return value is not retrieved and a third group warns about sequential issues. The tool Splint (Secure Programming Lint, 2008) was the only tool which gave warnings on all three groups. Therefore, Splint was chosen for assessment of our source code in part three. None of the tools performed checks on argument values and they did not check all argument types to be correct.

Based on the available documentation on existing analysis tools, we assume that some tools can check arguments and some tools can check the return value for the following issues:

- Types – assignment of variables, passing arguments of different type than function expects.
- Null pointers – a common cause of failures is when a null pointer is dereferenced.
- Definitions – all function parameters and global variables used by a function must be defined before a call, and the return value must be defined after the call.
- Allocations – concerns: reallocating storage when there are other live references to the same storage, or failing to reallocate storage before the last reference to it is lost.
- Aliasing – program errors often result when there us unexpected aliasing between parameters, return value, and global variables.

An important difference between the identified failure modes from the FMEA and the checks existing tools perform is to check a variable value in the context of the relevant function it is passed to. E.g. the system call **shmget**() has an argument of type *size_t*; As a data type, the variable must be checked to be of correct type and its value must be checked to be within the variable limits. Most existing analysis tools do these checks. But, in the context of the function the argument is passed to, the variable must be checked to determine e.g. whether its value is smaller than the maximum size of a shared memory segment (set by the operating system).

The next step was to assess the source code for the identified failure modes that existing tools do not check for. To automate this process, we made use of a prototype tool described in (Sarshar & Winther, 2008). The tool was modified for this study and its purpose was to identify different attributes for each argument that was passed to a given function. If statically detectable, the following attributes were determined; the argument type, value, name, whether it was an array and if so, its size. This information was used as input to check the arguments for the potential failure modes. Several of these checks were automated; however, a majority was done manually by examination of the argument attributes against the FMEA sheets for each function.

Splint was also applied on the source code of our case study with the checks described in the list above. However, the tool can also do more powerful checks enabled by source code annotations. Annotations are stylized comments that documents assumptions about functions, variables, parameters and types and follow a predefined syntax. To use the more powerful checks, the source code must be edited to add notations. This requires time and effort and was not applied in this case study. The use of annotations for more powerful checks applies to most static analysis tools.

5. Results

A subset of 19 external calls has been analysed using FMEA to identify potential failure modes that can cause a process to fail or propagate error. The examined functions were called 309 places in the source code.

In 242 of the cases, the return value from an external call was not retrieved or checked. In general, the return value often indicates whether a function succeeded or failed for some reason. If such failure is not handled, unexpected runtime errors can occur in a software system. As an example, consider an application which writes some data to a file regularly. The file is opened for reading successfully and the write function is called without checking its return value. If the file was inaccessible (e.g. lost connection to server) the write function would return a value indicating an error. If the error is not handled explicitly, a runtime error may occur. Such an error often causes the operating system to give an error message to the user and then terminates the application that caused the error. All unsaved data will be lost in such events. However, not all calls are this crucial; it is more vital that the return value from an open or write function is handled than the return value of a print to screen function. 76 of the ignored return cases were for a print function.

Several of the examined functions had potential failure modes regarding the content of arguments they receive. In example, char arrays passed to a group of functions must be null-terminated and for another group they must not contain a given character. Our assessment of the code did not identify any of these failure modes in the module.

The source code was also assessed using the tool Splint which gave near 2000 warnings on the source code of the module. Table 4 (Sarshar, 2009) presents warnings given by Splint and number of instances. In general, the tool reports many false positive warnings (which add noise to the results and make it harder to spot the real problems). Though the number of cases for the warnings on incompatible types and dangerous comparison are equal, there is no relation between them.

Warning on	**Cases**
Incompatible types	444
Dangerous comparison	444
Variable declared but not used	50
Value used before definition	160
Variable initialized to null value	14
Dangerous assignments	237
Test expression issues	163
Storage not released before return	37
Return value ignored	212
Possible buffer overflow with sprintf()	23
Arrow access of non-pointer	54
Other warnings	162

Table 4. Group of warnings given by the tool Splint

Assessment of many existing systems in the industry can only be performed on the available source code, and often, the specification is not available. This is where static analysis is useful, some tools only need the source code to perform their analysis. However, if annotations are necessary to perform an assessment, expertise on the system is required.
The method proposed to use FMEA on system calls to identify potential failure modes and then assess the source code for these potential failures. The intention was not to develop yet another tool, therefore the identified failure modes were checked against the ones that existing tools check. An interesting approach would be, if possible, to write these failure modes as additional checks for existing tools. A disadvantage of the FMEA analysis is that it only identifies a small fraction of the potential failure modes and it requires expert knowledge on the system calls.
System and library calls are complex functions which interact with the kernel of the operating system. The process of analysing such functions takes time and effort, but it only needs to be performed once for each function. The result from this analysis indicates that it is necessary to examine the source code of applications for failures related to system call usage.
The source code of the input data processing module of the SCORPIO framework was assessed using our approach and using the tool Splint. The user of analysis tools must be critical to the results as all vulnerabilities are not guaranteed to be found, and identified vulnerabilities are not all real problems. Splint gave a lot of warnings which were false positives while the checks from the FMEA performed by us gave few false positives. The reason for this is that we used a prototype tool to help us identify variable attributes, but the checks were done manually. Performing manual checks is time consuming, but reduces the chance of false positives since the analyser is required to have insight of the application. Furthermore, it is difficult, if not impossible, to control and check the value of variables that are passed to system services when performing static analysis.
Through the process of analysing the source code of the module, failure modes with the potential to cause harm at runtime as an effect of fault triggering and error propagation have been identified. These failure modes are related to usage of services provided by the underlying operating system. Though the arguments sent to such functions are valid and in accordance with the documentation, the majority of the potential failure modes detected in the code were related to handling of return values from these functions.
We did not expect that this assessment would identify any serious failures in the code, and the result demonstrates that this expectation is valid. Potential failures related to usage of operating system services would have been identified using our method and none of the potential failures identified is likely to cause the module to fail. However, taking these results into account in new releases of the module will reduce its vulnerability.

6. Discussion

The methodology was applied on a subset of system calls, some of them related to shared memory. This target was found to be suitable because it involved an intended channel for communication between processes through a shared resource; the memory. We also performed FMEA on other system calls to evaluate whether the method is applicable to a wider class of functions and not restricted to those related to shared memory. The errors identified in this approach are erroneous values in the variables passed to the system call interface and errors caused when return, or modified, pointer variables are not handled

properly. From the analysis we know not only which functions behave non-robustly, but also the specific input that results in errors and exceptions being thrown by the operating system. This simplifies identification of the characteristics an error has in code, making it easier to locate errors.

The method for analysing error propagation between processes primarily focuses on how the process of interest can interact with and affect the environment (the operating system and other processes). A complementary approach could be to analyse how a process can be affected by its (execution) environment. In (Johansson et al., 2007), the authors inject faults in the interface between drivers and the operating system, and then monitor the effect of these faults in the application layer. This is an example where processes in the application layer are affected by their execution environment. Comparing this method to our approach, it is clear that both methods make use of fault injection to determine different types of failure effects on user programs. However, the examination in (Johansson et al., 2007) only concerns incorrect values passed from the driver interface to the operating system. Passing of incorrect values from one component to another is a mechanism for error propagation and relates to problems for intended communication channels. Fault injection is just one method to evaluate process robustness in regards to incorrect values in arguments. In our work, we examine the failure effects of several mechanisms: passing of arguments and return values, usage of return values, system-wide limitations, and sequential issues. These methods complement each other.

Understanding the failure and error propagation mechanisms in software-based systems will provide the knowledge to develop defences and avoid such mechanisms in software. It is therefore important to be aware of the limitations for the proposed approach. This analysis only identifies failure modes related to the usage of system calls in source code. Other mechanisms for error propagation that do not involve usage of the system call interface will not be covered by this approach. This approach, however, complements existing methods and static analysis tools. An infinite loop structure in code is one example of a failure mode that does not make use of system calls. This failure mode can cause error propagation because it uses a lot of CPU time/resources.

The FMEA method worked well on system calls and identified failure modes that could cause error propagation between processes. However, the identified failure modes from the FMEA do not apply directly to other operating systems. A new analysis must be performed for a new programming language and operating system combination. Even though several operating systems provide the same functionality, e.g. usage of shared memory, the implementation of the service will be different. Thus, some of the failure modes may be similar, yet their effects may not. And, in contrast to general FMEA approaches which analyse functionality of software systems, our aim was to identify failure modes related to the interaction of a program with operating system services.

7. Conclusion

The analysis and results from this case shows that the approach facilitates the detection of potential failure modes related to the use of the system calls in operating systems. However, this is without further analysis about their actual impact in the SCORPIO framework. Future extension of the work can include examining the potential impact of these failure modes. With so many potential failure modes it also seems that there needs to be some way to prioritize or target the “important” failures that should be fixed based on the study. For

example, the missing return values seem to become critical errors only under maintenance, if the return values can change. Even though this is valuable to uncover, it would be more valuable to quantify which potential failures would be critical if they occurred under the current operational mode and which would not. This would help to indicate the usefulness of the technique and provide some evidence that the failures occur with sufficient frequency to justify the definition of a technique that targets them. Further extension of the work can include exploring alternative techniques or quantify effort required to conduct this type of analysis to make it easier to determine the trade-offs of using this technique in practice, providing a quantitative analysis of the types of failure modes the analysis uncover and providing usage guidelines to the practitioner.

8. References

Abdelmoez, W.; Nassar, D.; Shereshevsky, M.; Gradetsky, N.; Gunnalan, R.; Ammar, H. H.; Yu, B. & Mili, A. (2004). Error Propagation in Software Architectures, metrics, *Proceedings of International Symposium on Software Metrics No10*, pp. 384-393, Chicago IL, ETATS-UNIS, USA, September 11, 2004.

Bacon, J. & Harris, T. (2003). *Operating Systems – Concurrent and distributed Software Design*, 1st ed., Great Britain: Pearson Education Limited, 2003.

Barmsnes, K. A.; Johnsen, T. & Sundling, C-V. (1997). Implementation of Graphical User Interfaces in Nuclear Applications, *Proceedings of Topical Meeting on I&C of VVER*, Prague, April 21-24, 1997.

Beck, H.; Bohme, H.; Dziadzka, M.; Kunitz, U.; Magnus, R.; Schroter, C. & Verworner, D. (2002). *Linux Kernel Programming*, 3rd ed., Great Britain: Pearson Education Limited, 2002.

Bic, L. F. & Shaw, A. C. (2003). *Operating Systems Principles*, USA: Pearson Education, Inc., 2003

Bovet, D.P. & Cesati, M. (2003). *Understanding the Linux Kernel*, 2nd ed., USA: O'Reilly & Associates, Inc., 2003.

Chou, A.; Yang, J.; Chelf, B.; Hallem, S. & Engler, D. R. (2001). An Empirical Study of Operating Systems Errors, *Proceedings of the 18th Symposium on Operating System Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, October, 2001.

Engler, D.R.; Chelf, B.; Chou, A. & Hallem, S. (2000). Checking System Rules Using System-Specific, Programmer-Written compiler Extensions, *Proceedings of Operating systems Design and Implementation (OSDI)*, San Diego, California, USA, October, 2000.

Fredriksen, R. & Winther, R. (2007). Challenges Related to Error Propagation in Software Systems, *Proceedings of Risk, Reliability and Societal Safety (ESREL),* pp. 83-90, ISBN 978-0-415-44783-6, Stavanger, Norway, June 25-27, 2007.

Goradia, T. (1993). Dynamic Impact Analysis: A Cost-Effective Technique to Enforce Error Propagation, *Proceedings of the International Symposium on software Testing and Analysis*, pp. 171-181, 1993.

Hatton, L. (1995). *Safer C: Developing for High-Integrati and Safety-Critical Systems*, Great Britain: Mcraw-hill, 1995.

Hiller, M.; Jhumka, A. & Suri, N. (2001). An Approach to Analysing the Propagation of Data Errors in Software. Dependable Systems and Networks (DSN), 2001.

IFE (Institute for Energy Technology) (2010). *ProcSee*, available from: http://www.ife.no/departments/visual_interface_technologies/products/procsee

Jhumka, A.; Hiller, M. & Suri, N. (2001). *Proceedings of 20th IEEE Symposium on Reliable and Distributed Systems*, pp. 152-161, New Orleans, LA, USA, October 23-31, 2001.

Johansson, A.; Suri, N. & Murphy, B. (2007). On the Impact of Injection Triggers for OS Robustness Evaluation, *Proceedings of the 18th International Symposium on software Reliability Engineering (ISSRE)*, pp. 127-136, 2007.

Koenig, A. (1989). *C Traps and Pitfalls*, USA: Addison-Wesley, 1989.

Kropp, N. P.; Koopman, P. J. Jr. & Siewiorek, D. P. (1998). Automated Robustness Testing of Off-the-Shelf Software Components, *Proceedings of the Symposium on Fault-Tolerant Computing*, pp. 230-239, 1998.

Michael, C. & Jones, R. (1997). On the Uniformity of Error Propagation in Software, *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS)*, pp. 68-76, 1997.

Mitchell, M.; Oldman, J. & Samuel, A. (2001). *Advanced Linux Programming*, 1st ed., USA: New Riders Publishing, pp. 45-55, 2001.

Nassar, D.; Rabie, W.; Shereshevsky, M.; Gradetsky, N. & Ammar, H. (2004). Estimating Error Propagation Probabilities in Software Architectures, *Proceedings of International Symposium on Software Metrics No10*, pp. 384-393, Chicago IL, ETATS-UNIS, USA, September 11, 2004.

Nutt, G. (2004). *Operating Systems*, 3rd ed., USA: Pearson Education, Inc., 2004.

Pinkert, J. R. & Wear, L. L. (1989). *Operating Systems – Concepts, Policies, and Mechanisms*, USA: Prentice-Hall, Inc., 1989.

Sarshar, S.; Simensen, J.E.; Winther, R. & Fredriksen, R. (2007). Analysis of Error Propagation Mechanisms between Software Processes, *Proceedings of Risk, Reliability and Societal Safety (ESREL),* pp. 91-98, Taylor & Francis, ISBN 978-0-415-44783-6, Stavanger, Norway, June 25-27, 2007.

Sarshar, S. (2007). Analysis of Error Propagation between Software Processes in Source Code, Master thesis at Østfold University College, Norway, 2007.

Sarshar, S. & Winther, R. (2008). Automatic Source Code Analysis of Failure Modes Causing Error Propagation, *Proceedings of Risk, Reliability and Societal Safety (ESREL),* pp. 183-190, Taylor & Francis, ISBN 978-0-415-48514-2, Valencia, Spain, September 22-24, 2008.

Sarshar, S. (2009). Performing Code Interface Analysis on the SCORPIO Core Surveillance Framework", *Proceedings of the 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT),* American Nuclear Society, LaGrange Park, IL, Knoxville, Tennessee, April 5-9, 2009.

Secure Programming Lint (2008). Annotation-Assisted Lightweight Static Checking, Available from: http://www.splint.org, 2008.

Silberschatz, A.; Galvin, P. B. & Gagne, G. (2005). *Operating System Concepts*, 7th ed., USA: John Wiley & Sons, Inc., pp. 43-55, 2005.

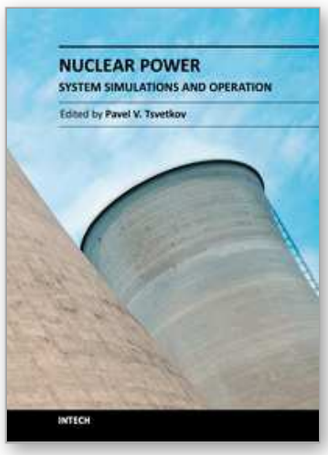
Stallings, W. (2005). *Operating Systems – Internals and Design Principles*, 5th ed., USA: Pearson Education, Inc., 2005.

Stamatis, D. (1995). *Failure Mode and Effect Analysis: FMEA from Theory to Execution*, American Society for Quality, USA, 1995.

Storey, N. (1996). *Safety-Critical Computer Systems*, Britain: Pearson Education Limited, 1996

Tanenbaum, A. S. & Woodhull, A. S. (2006). *Operating Systems – Design and Implementation*, 3rd ed., USA: Pearson Education, Inc., 2006

Voas, J. (1997). Error Propagation analysis in COTS Systems, *IEEE Computing and Control Engineering Journal*, 8(6):269-272, December, 1997.



Nuclear Power - System Simulations and Operation
Edited by Dr. Pavel Tsvetkov

ISBN 978-953-307-506-8
Hard cover, 192 pages
Publisher InTech
Published online 06, September, 2011
Published in print edition September, 2011

At the onset of the 21st century, we are searching for reliable and sustainable energy sources that have a potential to support growing economies developing at accelerated growth rates, technology advances improving quality of life and becoming available to larger and larger populations. The quest for robust sustainable energy supplies meeting the above constraints leads us to the nuclear power technology. Todayâ€™s nuclear reactors are safe and highly efficient energy systems that offer electricity and a multitude of co-generation energy products ranging from potable water to heat for industrial applications. Catastrophic earthquake and tsunami events in Japan resulted in the nuclear accident that forced us to rethink our approach to nuclear safety, requirements and facilitated growing interests in designs, which can withstand natural disasters and avoid catastrophic consequences. This book is one in a series of books on nuclear power published by InTech. It consists of ten chapters on system simulations and operational aspects. Our book does not aim at a complete coverage or a broad range. Instead, the included chapters shine light at existing challenges, solutions and approaches. Authors hope to share ideas and findings so that new ideas and directions can potentially be developed focusing on operational characteristics of nuclear power plants. The consistent thread throughout all chapters is the â€œsystem-thinkingâ€ approach synthesizing provided information and ideas. The book targets everyone with interests in system simulations and nuclear power operational aspects as its potential readership groups - students, researchers and practitioners.

Fax: +385 (51) 686 166
www.intechopen.com

Fax: +86-21-62489821

IntechOpen