Université catholique de Louvain
École Polytechinique de Louvain
ICTEAM Computer Science Engineering

Université d'Abomey-Calavi
Ecole Doctorale des Sciences de l'Ingénieur (ED-SDI)
Institut de Formation et de Recherche en Informatique (IFRI)

Application hardening by adapting an open source operating system

Ensure the maintenance of services in the presence of failures or transient errors caused by cosmic radiation

---

BY

Emery Kouassi Assogba

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

Thesis committee

1 (Supervisor) **Prof. Marc Lobelle**	UCL, Belgium
2 (Supervisor) **Prof. Eugène C. Ezin**	UAC, Benin
3 (Chairman) **Prof. Charles Pêcheur**	UCL, Belgium
4 (Secretary) **Prof. Olivier Bonaventure**	UCL, Belgium
5 (Member) **Prof. Jean-Didier Legat**	UCL, Belgium
6 (Member) **Prof. Herbert Bos**	VUA, Nederland
7 (Member) **Former CTO, Dr. Marc Durvaux**	Industry, Belgium

The conclusion of the matter, everything having been heard, is: Fear the true God and keep his commandments, for this is the whole obligation of man. (Ecclesiastes 12:13)

To Esther, Ivana, Kate and Jeholia.

Preamble

Since physicist Charles Wilson's work on ionizing currents, there has been a lot of work to better understand the effect of cosmic radiation on electronic equipment. Particles from the activity of the sun are denser in the space environment. However their effects exist also at sea level. Charged particles provided by these radiations are able to ionize directly or indirectly the molecules that constitute the transistors. These charge deposits can have permanent effects that go as far as the destruction of the equipment or transient effects that disappear following a refresh of the state of the circuit.

As long as the voltage levels to represent the information were high enough, electronic circuits were not disturbed by the action of these radiations. In recent years with the reduction of the size of the transistors, the charges to represent the information are lower, which makes the circuits more and more sensitive. In 1999 Sun Microsystems was forced to remove a set of servers from the market. In-depth analysis showed that the SRAM were sensitive to cosmic radiations. In 2008, in Belgium a candidate for election was awarded 4096 votes more than the number of registered voters. There are also soft errors at sea level caused by the material used to manufacture the circuit. An example is Intel 2107 series 16KB DRAM where the package was contamined by radioactive water. Removing the radioactive component removes also the source of the soft errors. That is different from soft errors caused by cosmic radiations (proton, electron or heavy ion) which are probabilistic (random) events.

These radiation effects can be classified as long-term effects or transient effects. The long term radiation effects are caused by the fact that the system was exposed for long time to the radiations. Whereas the transient effect can be caused by individual ionizing particles. The charges induced by these particles can cause several effects. Some of them are destructive for the circuit (Single Event Latchup SEL, Single Event Burnout SEB) others are non destructive (Single Event Upset SEU, Single Event Functional Interrupt SEFI).

It is therefore essential to protect computer systems against this random and unpredictable phenomenon. Protection can be introduced during circuit design. The circuit is then designed in such a way that the effects are either suppressed or tolerated. Redundancy is introduced into the circuit. We are talking about hardware hardening. This approach makes the cost price of the equipment too high to be competitive at ground level. On the other hand, for specific and specialized components

such as elements to be placed in a satellite or a satellite launcher, the circuits are duplicated and triplicated to reduce or completely eliminate the effect of cosmic radiation in a space environment. Another approach is to tolerate the effect at the circuit level and manage it in the software. The program is modified either manually or dynamically using a compiler to add spatial and temporal redundancy. These redundancies make it possible to detect and correct errors due to cosmic radiation with various levels of success. At sea level, servers which operate for long periods of time are also victims of soft errors. To improve their reliability, availability and serviceability manufacturers protect them by adding errors correction codes (ECC) in TLBs, caches and central memories.

In this work, a micro-kernel operating system has been modified so that it can protect its user processes against transient errors caused by cosmic radiations, using a software based technique called blended hardening. This work does not address SEU induced faults and crashes of the operating system itself because it is assumed that the operating system can be modified to be SEU resistant: this is one (large) program to modify. However, there are so many different programs running over an operating systems that modifying them all would be a boundless work. This is the reason why this thesis addresses specifically the case of programs that cannot be modified to become SEU resistant and run on top of the operating system. The hardening job is implemented in the operating system. The focus of this work is on SEU affecting the processor itself and its caches, not the central memory because central memories can be protected in hardware with (relatively) simple circuits such as ECC with scrubbing. It is thus assumed that the central memory will not be modified directly by SEU. However, bit flips in processor registers and caches can cause processes to crash or behave haphazardly and such a faulty process can perform unwanted changes to the central memory. To avoid these indirect consequences of SEU, processes must be constrained, if faulty, to only be able to modify memory areas the contents of which can be afforded to be lost. The rest of the memory is called "protected memory": it must be immune to SEU induced modifications as well direct as indirect. The blended hardening technique is based on dividing each running process in short processing elements. Short enough to be, with a very high probability, only subject to, at most, a single SEU or multiple SEUs not masking each other effects. It consists of exploiting the exceptions initiated by all available hardware fault detection mechanisms and by using a method belonging to the DWC (Double execution With Comparison) class of fault tolerance

techniques to detect silent transient SEU faults, those causing false results but no exception. All the process memory is read-only for the process. Only the parts of the memory of the process needing to be modified during the execution of the processing element are read/write in a partial copy of the process memory (this part of memory is called "working set" or "result" of the processing element). The processing element is run twice. The modified memory areas are then compared. If no fault happened, the result of the two executions will be identical and the next processing element can be run. SEU can cause the process to crash, i.e. produce an exception that will be detected by the operating system, or can cause the results of the two executions to be different. Thus all errors are detected. In case of error, the execution of the same processing element is restarted instead of starting the next one. Any error induced by a transient modification of data contained in the memory can be detected and recovered that way. Any kind of user process can be hardened. This looks simple and, indeed, it is simple to implement if one may modify the program to implement it but applying it to unmodified user processes raises a lot of problems. How to divide the process in processing elements? Which variables of the process must be copied in the read-write memory of the processing element? What if the process uses the values of date and time? And many more. The hardest work of the thesis was identifying and solving these issues for a target operating systems in order to demonstrate the feasibility of the hardening method. For instance, processing elements are limited by system calls, and by a new technique of counting instructions which is used to limit the duration of the processing element to a time where only one SEU can occur when the delay between system calls is too long. The right number of instructions to avoid SEU to be undetected (which can only happen when several SEUs mask each others effects) has been determined by a statistical analysis. The protection of memory against processes behaving haphazardly is obtained by an innovating use of the paging system. The implementation has been based on a micro-kernel operating system, Minix3 v3.4.0. The micro-kernel itself (to handle the hardening itself) and the virtual memory manager (to implement protected memory) have been modified. All other part of Minix remain unchanged. Only a single core of the computers has been used in this work because Minix3 mostly targets these and in order to reduce the solution search space. The implementation has been tested by fault injection in a benchmark program suite and in recompilations of the Minix programs: faults have been injected in user programs at random times at a rate simulating the

worst case rate observed in typical space missions, that had also been used to define the duration of processing elements. The effect of SEU in all processor registers has been simulated. All errors were recovered.

This research target COTS processors. But not all COTS processors are equal regarding resistance to transients faults. Some COTS processors are partially hardened in order not to endure transient faults at ground level. Typically these are designed for high availability servers (for instance Intel Xeon) and mission critical embedded system (for instance Intel Atom). These processors support multibit ECC for protecting memories and critical registers. Of course course we assume that such partially hardened COTS are used. Our software makes them suitable for spatial missions critical.

Acknowledgements

This thesis would not succeed without the unwavering support of some people I want to thank.

I would like to thank Prof. Marc Lobelle for leading this thesis. His advice and direction allowed to realize this thesis. His good humor and his optimism convinced us to persevere when it was difficult. The many discussions we had in his office gradually illuminated the path leading to this thesis.

I would like to thank his wife Hilda for her moral support. She allowed us to spend very pleasant moments of relaxation when she invited us often to lunch or dinner.

I would like to thank Prof. Eugène C. Ezin. Despite his busy schedule he agreed to co-lead this thesis. His advice and guidelines have been invaluable to us.

I would like to thank Professor Olivier Bonaventure for his help. His technical advice on the operating system development environment has been a breath of fresh air for us. His comments on the first 5 chapters of the thesis helped to improve the quality of the document.

I would like to thank Mr. Michel Melotte for his help. His explanations and his experience in radiation allowed help me to compute the single event error rate that contributed to the writing of the chapter 3 of my thesis. Thank you for your explanations and your patience.

I would like to thank Prof Nobert Hounkonnou. He initiated with Belgian cooperation the project that lead to this thesis.

I would like to thank Professor Antoine Vianou and all the members of the doctoral school"Sciences de l'Ingénieur " of the University of Abomey-Calavi for the scientific environment they allowed me to have during my stays in Benin.

I would like to thank ADRI and ARES CCD for providing the necessary funding for the completion of this thesis. Without forgetting Mr Christian Duqué, Mrs Emmanuelle Paul, Mrs Dominique Socquet, Mrs Danisa Zaparata, Mrs Gabriela Bidegain for their warm welcome.

I would like to thank all members of UCL computer engineering department without forgetting Mrs Vanessa Maons, Mrs Chantal Poncin, for their logistical support and smiles.

I would like to thank all the members of the Computer Training and Research Institute, Prof. Eugène C. Ezin, Prof. Gaston Edah, Mrs. Gnonlonfoun Miranda, Dr. Arnaud Ahouandjinou, M. Jerôme Zohoun. They provided me with the right working environment during my stays in Benin.

I would like to thank my research team members Jean-Marie Kabaseley for his good advice and unwavering support, Parfait Tokponnon for the good times we spent discussing our common problems, Laurent Lesage for his early advice.

I would like to thank my friends Fiacre Kinmangbahohoué , Césaire Yadouléton, Hervé Ahouantchédé, Mêton-Mêton Atidehoun, Edoh Maxime, Houndji Ratheil, John Aoga, Lionel Metongnon, Gael Aglin.

I would like to thank all the brothers and sisters of the assembly of Jehovah's Witnesses of "Wavre Ouest" for their support without forgetting Berthe N'sumpi, Tina Mavuela, Christelle KEMBOU NUMBI, Germaine Habiba NGOUA, Véronique Mascard, the DENIS family, the DESTIN family, the STORMS Family, the BERTHE family, the JULIANO family.

I would like to thank my brother Ange Assogba and and my sisters Aurore Assogba, Lucie Assogba, and Eudoxie Bessan for their support.

I would like to thank Patrice Daavo and his family. As a father he took care of me and supported me. Thank you.

I would like to thank my wife Esther for her patience and support at every stage of this thesis. I am infinitely grateful for all the sacrifices she has made. She has always endured my long absences with a smile and hope by continuing to take good care of our daughters. I will not stop saying thank you.

I would like to thank all those whose name I have not mentioned but who in one way or another helped me in the realization of this thesis. Thank you to all of you.

List of Acronyms

AES	Advanced Encryption Standard
BHT	Blended hardening technique
BIOS	Basic Input Output System
CMOS	Complementary Metal Oxide Semiconductor
COTS	commercial off-the-shelf
CPU	Central Processing Unit
CRC	cyclic redundancy check
CR	Control Register
DAFT	Decoupled Acyclic Fault Tolerance
DCE	Detected Corrected Error
DDDC	Double Device Data Correction
DECTED	Double Error Corrected Triple Error Detected
DIMM	Dual Inline Memory Module
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random Access Memory
DUE	Detected Uncorrected Error
DWC	Double Execution With Comparison
ECC	Error Corrector Code
FDSOI	Fully Depleted Silicon On Insulator
FinFET	Fin Field-effect transistor
FIT	Failure In Time
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit

FTR Forward Temporal Redundancy

GDTR Global Descriptor Table Register

GEO Geostationary earth Orbit

IDTR Interrupt Descriptor Table Register

IO Input Output

ISS International Space Station

L1 Level 1 cache

L2 Level 2 cache

L3 Level 3 cache

LDTR Local Descriptor Table Register

LEO Low earth orbit

LET Linear Energy Transfer

LLC Last Level Cache

LMCE Local Machine Check Exception

MBU Multi-bit Upset

MCA Machine Check Architecture

MCE Machine Check Exception

MD5 Message Digest 5

MEO Medium earth orbit

MMX MultiMedia eXtensions

MPX Memory Protection eXtension

MTBF Mean Time Between Failure

MTTF Mean Time To Failure

MTTR Mean Time To Repair

PARSEC Princeton Application Repository for Shared-Memory Computers

PCD Page-level cache

PFN Page Frame Number

RAS Reliability Availability Serviceability

SBU Single Bit Upset

SDDC Single Device Data Correction

SEB Single Event Burnout

SECDED Single Error Corrected Double Error Detected

SEE Single Event Effect

SEFI Single Event Functional Interrupt

SEL Single Event Latchup

SER Soft Error Rate

SET Single event Transient

SEU Single Event Upset

SIHFT Soft Implemented Hardaware Fault Tolerance

SIMD Single Instruction Multiple Data

SPENVIS SPace ENVironment Information System

SPLASH Stanford Parallel Applications for Shared-Memory

SRAM Static Random Access Memory

SSE4.2 Streaming SIMD Extensions version 4

TLB Translation Lookaside Buffer

TMR Triple Modular Redundancy

TSX Transactional Synchronization eXtension

List of Symbols

λ	Errors per bit per second
τ	Average time while a variable remains unchanged in L1 cache

Contents

List of Figures

List of Tables

Part I

Background

CHAPTER 1
Introduction

1.1 Context and Problem statement

The context of this research is coping with a natural phenomenon which, combined with the evolution of technology causes trouble in computers and electronics systems: radiation, generally due to solar activity, consisting of high energy particles that leave charges on microchips when colliding with their surface[Hei11]. This phenomenon creates trouble in electronic devices [VFR07]. The diminishing size and voltages in use in microchips increases their sensitivity to these phenomenons because the charge holding a bit of information becomes smaller and can more easily be changed by the charge transferred by a high energy particle colliding with the chip[Hei11] [VFR07] . Sometimes these effects are permanent, but more often they are transient. This thesis targets the transient effects which cause mainly trouble in memory systems, processors and devices registers. Transient effects of collisions of charged particles with electronic devices are called single event upset (SEU). They are common in space environments, at high altitudes and in irradiated areas. A SEU is a transient fault, in one or several neighboring memory cells, caused by the energy transferred from a high energy particle to an electronic circuit. In a computer, the fault caused by a SEU could affect data, the instruction opcode or the program execution flow [VFR07] [Nic02]. With the diminishing size of memory cells, a single particle may harm several cells. SEU can be classified as [Hei11]:

- Single-bit upset (SBU) when a particle strike toggles a bit-flip in a memory cell or latch
- Multiple-bit upset (MBU), when the event toggles two or more bits in the same word or multi-bit latches
- Multiple-cell upset (MCU) when the event toggles bits of two or more memory cells (words)
- Single-event transient (SET), when the event causes a voltage glitch in a circuit, which becomes a bit error when captured in a storage element

- Single-event functional interrupt (SEFI), when the event causes a loss of functionality due to the perturbation of control registers, clock signals, lockup, etc.

Avoidance or mitigation of SEU effects can be tackled in hardware or in software or both [RRTV99]. They can also cause a malfunctioning of a circuit or even a direct system crash [Hei11]. Crashes can also occur as a consequence of a change of the program execution flow.

1.2 Contribution of the thesis

In this thesis a micro-kernel operating system has been modified in order to protect its users application processes against transient faults. Hardening these processes is done transparently. User application programs are not modified at programming time nor at compilation time. All the software hardening functions are implemented in the operating system.

There are several reasons for preferring a hardening technique that neither modifies the program code nor depends on what this code is actually doing.

A first reason is that the code to harden is not always available. This does, of course, not apply to the code controlling the spacecraft and on which the survival of the spacecraft depends: this code is made to measures, according to the requirements that may include SEU hardening. However, spacecrafts may include computers in their payload and the software in these payloads may be off-the-shelf software that was not designed for space by companies that want neither to release their sources nor to spend resources on hardening their software.

A second reason is that software is expected to be error free, but when the software is used in irradiated areas, this means not only bug free but also not sensible to SEU. These qualities are different and both are expected to be guaranteed, but the required guarantees are not the same for the absence of bugs and for resistance to SEU. The absence of bugs is expected in all software and the way to guarantee it is not dependent on where the software is to be used: on the ground or in space. On the other hand, systems will only be considered as SEU-resistant if they have been tested under irradiation, generally in a cyclotron. If anything is changed in the software (e.g. new release) or its environment, the tests have to be redone.

This is why hardware hardening is often preferred, even if much more expensive. If a system is hardened in hardware, its quality of "be-

ing hardened" does not depend on the software running on it. However the previous sentence said "if", not "if and only if". Indeed the quality of "being hardened" will also be independent on the software to harden if the hardening is not embedded in the software to harden. This is why implementing the hardening in the operating system without any requirement on the programs to harden is interesting: if the operating system is proved to be both bug free and SEU resistant and if it can harden application software running on it, then, like with hardware hardening, when new application software must be added or when a new release must be installed, it must only be guaranteed to be bug free, not SEU resistant, because that quality is provided by something external to this software.

A hardened operating system that is also able to harden its application software, whatever this software is doing would thus be a desirable alternative to hardened hardware. Of course, the hardening of respectively the hardware or the operating system and its application hardening add-on has to be demonstrated.

In this work it is assumed that the operating system itself is immune to SEU effects. This work is thus a first step towards the complete hardening of computer systems including all their unmodified application processes against errors caused by SEU. This hardening of the application processes by the operating system has to be the first step because hardening only the operating system would be of limited use if the same hardening work had to be done for each application program, which would be an unbounded work. It is thus necessary to show first that it is possible to harden unmodified application processes from within an operating system. This is the main contribution of this thesis.

Hardening the operating system itself will be a lot of future work but the operating system is a single software to harden and it may be modified. In order to limit this amount of work, a micro- kernel operating system has been selected. In a micro-kernel operating system, many traditional operating system functions are subcontracted to separate system processes and most of them can be handled in the same way as user application processes, so by hardening the application processes, some system processes that do not include the new hardening functions can be hardened as well. However, some exceptions are expected, in particular system processes handling peripheral devices could have to be adapted to take into account SEU induced errors in peripheral device registers.

The selected micro-kernel operating system, Minix3 only supports single core computer. Only single processing core are considered. Modi-

fying Minix3 to add multi-core support is possible but would have been a completely different development than the purpose of this thesis. It is thus considered as out of the scope of this work. Some of the shelf processor dedicated to servers and mission critical embedded systems include already a few hardware hardening features, but insufficient for use in space environment. The use of such a processor is assumed in this work. TLB, L2 and L3 caches of the chosen processor are considered to be protected against direct SEU effect.

The unmodified user application process hardening by the operating system has been completely implemented and tested with injection of faults similar to those caused by SEU on realistic programs such as classical benchmarks, usual UNIX commands and recompilation of the benchmarks.

The hardening method includes several technical innovations. The first is that the choices are based on hypotheses based on realistic statistical analysis in order to be able to guarantee the efficiency of the protection. Another is a new instruction counting technique allowing to execute exactly twice the same instructions without any knowledge of what the program does. A third innovation is an efficient way of using the pagination system for the protection of the memory of a process against erratic behaviors of this process if it is hit by a SEU.

The measured average overhead is 141% for computing and system call intensive applications (GZIP and MD5 is 2,41 times lower).

The method has been validated by fault injection at run time. Faults have been injected into critical processor registers at random times at a typical rate observed in space conditions (satellites orbiting the Earth). All injected faults have been detected and corrected. Other tests include a higher than natural number of events, but biased by the fact that the hypotheses on which the hardening techniques are based are respected, which would not be the case at these higher event rates.

1.3 Outline of the thesis

The next chapter presents the state of the art on fault, error, failure and fault tolerance techniques. Chapter 3 presents an assessment of the problem and the key features offered by current processor architectures. Chapter 4 presents the choice of an operating system. Chapter 5 presents the principles of hardening user applications using the blended hardening technique (BHT). Chapter 6 presents the implementation of

hardening user applications using the blended hardening technique on the chosen operating system. Chapter 7 presents fault injections tests and performance tests. Chapter 8 presents the conclusion and future works.

CHAPTER 2

Sources and consequences of faults in computer systems

Introduction

This chapter presents the state of the art on fault tolerance principles and techniques with focus on random transient faults. First theoretical aspects of faults, errors, failures are outlined. Then the techniques to mitigate or tolerate faults are presented. After this chapter the reader will have a clear idea on the characteristics of fault tolerance techniques relevant for this work, ie. their strengths and their limitations.

2.1 Faults, errors and failures in programs

2.1.1 Program

According to [KA96] a program is composed of a finite set of variables and a finite set of processes. Each variable belongs to a non-empty domain. Each process is composed of a finite set of actions; each action has a unique name and is in the form: $name :: guard \rightarrow instruction$. Guard is a Boolean expression for program variables. The action instruction updates zero, one or more variables of the program atomically and instantaneously. Let p be a program.

2.1.2 State and state predicate of a program p

According to [AG93, KA96] a state of p is defined by a value of each variable of p chosen in the domains of each of these variables. A state predicate of p is a Boolean expression on the variables of p. An action of p is active in a state if its guard condition (predicate state) is true in this state. A process is active in a state when some process actions are active in this state.

2.1.3 Program's computation

According to [AK98, KA96] a computation of p is a fair and bounded sequence of steps. In each step, the action of p active in the current state is chosen and the instruction associated with this action is executed atomically. Equity comes from the fact that if at each step of the sequence it is the same action of p that is active it will be chosen for execution. So we check if we have to redo the action again before proceeding to the next action. If necessary, we wait for the predicate of the next action to be true to execute it. The bounded number of steps means that when the sequence ends the guard condition of all actions of p are false.

2.1.4 Fault, error, failure

In the field of fault tolerance of computer systems, a fault is considered to be the identified or suspected cause of an error. An error is however a part of the state of the system that can lead to a system failure. The system fails when it no longer respects its specifications [Pul01, KK07]. By definition transients faults are those that cause transient or permanent failures. Transient failures cease as soon as these faults cease. The system converges to a stable state. Permanent failures are irreversible even when the fault disappears. Permanent faults cause irreversible failures even when the fault disappears. Byzantine or intermittent faults are oscillating faults, they do not cease, sometimes they are active and then they are inactive [KK07]

Some faults could be detected or not by the hardware. Detected faults could be corrected or not. They could also be fatal or catastrophic. The undetected fault could be benign or critical.

2.1.5 Fault tolerance

According to [**?**], p is F-tolerant for the invariant S (F is a class of faults and S is the invariant of the program p) if and only if there is a predicate T that satisfies the following three requirements:

- At each state where S is true, T is also true, i.e. $S \Rightarrow T$
- From any state where T is true, if any action of p or F is executed, the resulting state is a state where T is true.
- From any state where T is true, all calculations of p only result in a state where S is true.

The predicate T is a fault-span [AGV96], a limit in the space of the states of p in which the state of p can be disturbed by the occurrence of faults of F. If the faults continue to occur, the states of p remain within this limit. When the faults stop, p converges to states where the invariant S is true.

2.1.6 Types of fault tolerance

Behaviors of a program in presence of faults F can be classified in three types of fault tolerance including masking of faults, non-masking of faults and fail-safe tolerance. This classification is based on how the problem specifications are met in the presence of faults. In other words the program [AGV96]:

- p masks the faults of F if and only if, when a fault of F occurs in a state where S is true, p continues to remain in a state where S is true. (S is the invariant of p). That is, the specifications of the problem are always respected by the program.

- p does not mask the faults of F if and only if, when a fault of F occurs in a state where S is true, p can be disturbed and evolve to a state where S is false. However, the program p is restored in one satisfactory state S a certain time after the end of faults of F.

- p is fail-safe tolerant to F for S if and only if there is a predicate R such that p is F-tolerant for $S \vee R$, the application of any action of p or F in a satisfactory state $S \vee R$ results in a satisfactory state $S \vee R$. This is sometimes called "graceful degradation". A particular case called "fail-stop" is, in case of fault, S remains satisfied or the system stops (R is then stopped).

2.1.7 Measures of fault tolerance

Three metrics are used to measure the fault tolerance of a system. The *Mean Time To Failure (MTTF)* is the average time the system operates until a failure occurs. Whereas the *Mean Time Between Failures* is the average time between two consecutive failures. The Mean Time To Repair (MTTR) is the time needed to repair the system following the first failure [KK07].

$$MTBF = MTTF + MTTR \tag{2.1}$$

2.1.8 Origin of the fault

The fault could come from the program itself. A programming error or an unavailable or invalid input provided by user or others programs could generate a fault inside the program. Such a fault could be avoided by increasing the reliability of the program (checking the input data before using them etc) [Whi03].

The faults could also come from the device on which the program is running (hardware failure). That could be a byzantine fault. The replacement of the hardware could reduce the risk of this kind of fault [AGM+17].

The fault could also come from high energy particles created by radioactivity in the device's package. That is a transient fault. Replacing the device package could remove the risk of this kind of fault [SBD+17]. The high energy particles could also be created intentionally to disturb the normal behavior of the program. This phenomenon could be found in the security field where an attacker would want to bypass the access control of the system or want to bypass the authentication mechanism or want to steal data [GWJL18].

However Single Event Upset (SEU) is a phenomenon that occurs when charged particles (from cosmic radiation) collide with an electronic circuit. The impact causes a transfer of energy of the particles contained in the radiation to the circuit. This can modify the charges of the electronic circuit, causing bit state changes [Nor96]. The consequence of this phenomenon on unprotected devices is the occurrence of errors described as "soft error "because they are random, transient and do not cause the deterioration of the circuit unlike" hard errors "which are permanent and destructive for the material. The memories, the caches, the registers and the processors are the most exposed to the consequences of SEUs. There is data corruption causing any type of error to software level i.e. crash, miscalculation, bad sequencing of processor instructions, execution of a false instruction [TVVB18].

2.2 Techniques to mitigate or prevent SEU

Several techniques have been developed to mitigate or prevent errors caused by SEU in computer systems. There are techniques implemented at hardware design level and techniques implemented at software level also called the software-implemented hardware fault tolerance (SIHFT) paradigm [RRV11] because basically the principles are similar in hard-

ware and software hardening. There are pure software techniques and hybrid techniques, that take into account specific hardware functions.

2.2.1 Hardware approaches

Hardware Protection from SEU effects can either lower the SEU probability of occurrence by using more resistant technologies, such as bigger memory cells holding larger charges, by using sapphire or other insulators as substrate rather that semi-conductors or by shielding the electronic devices. These techniques, already used for a long time are still successful today [BMS07] [BPP+08], but they generally involve more weight, thus more cost to bring the devices to space.

The second hardware protection method is by adding redundant hardware in order to correct SEU caused errors on the fly, for instance, by duplicating (DWC: duplication with comparison) or triplicating all sensitive cells and circuits and adding a comparison or a voting circuit (TMR: triple modular redundancy), by using error correction codes on multi-bit cells etc [Pla80].

With both categories of techniques, specially designed and manufactured electronic devices are needed. When only a few of these special devices are built, they are much more expensive than mass produced devices, usually called commercial off-the-shelf devices (COTS) [Nic11] [CDL+16].

In the case of MBU (SEU affecting several bits in the same memory word), some traditional hardware protection techniques (e.g. single bit error correction techniques based on Hamming codes) are ineffective. Software and software based techniques generally do not try to correct single bits in words and can handle all types of SEU in the same way.

2.2.2 Software approaches

Software hardening techniques are generally also based on DWC or TMR but implemented in software, either by a preprocessor transforming the code before compilation or by a modified compiler [CRK+15, SHD+15, BMD+17]. These techniques are thus limited to a single programming language, sometimes even to a single processor architecture and require the availability of the source code of the program. Programs hardened purely in software have only access to resources accessible to application programs, i.e. their own (virtual) memory and user mode accessible functions of the processor, none of which can really be trusted. Such

approaches can thus only hope to reduce the error rate but not to eliminate completely errors. Based on simulations, Yun Zhang et al [ZLJA12] claim a fault coverage around 99,9%.

2.2.3 Hybrid hardening approaches

Hybrid hardening techniques are software techniques using properties of available hardware or simple additional hardware not requiring changes to processor devices. Hybrid hardening techniques implement redundancy in software, thus avoiding the need for duplicating or triplicating hardware inside the processor [Amo15, WVB+15]. The redundancy in the processor's hardware is generally replaced by a redundancy in time, although it could also be implemented by the simultaneous use of several cores. The difference between pure software and hybrid hardening is that pure software hardening techniques are not dependent on particular hardware features: they can generally be used on any computer while hybrid hardening techniques require access to specific hardware features of the computer and use these to provide a much higher success rate than pure software techniques [Rot99] [SPR00] [MAAB13] [Döb14] . For instance, in this work, error detection systems embedded in processors and activating the exception mechanism, as well as fault avoidance features of caches and paging systems implementations of some common modern processors are used.

A survey of software and hybrid hardening techniques focused on shared Memory Multicore/Multiprocessor systems was published in 2011 by H. Mushaq [MAAB11] He identified four techniques for error detection: using a watchdog timer or a watchdog processor or also by using redundancy DWC or TMR. For error recovery, he identified also two possibilities: checkpoint and repair and error masking.

Software and software based hybrid hardening techniques can target different program granularity levels such as sub-instruction logic, each instruction, a statically defined set of instructions, a dynamically defined sequence of instructions, a procedure, threads of the program or the program itself.

When an occasional fault is acceptable, all the hardening techniques can be used to mitigate the effect of SEU. This may be the case for some workload equipments where a temporary malfunction does not put the whole vehicle at risk. For mission critical systems, all SEU induced faults should be corrected.

2.3 A short survey of pure software hardening techniques

Software hardening techniques use data redundancy and diversity, computation redundancy, time redundancy, procedure-level redundancy, simultaneous multi-threading, executable assertions to detect and if possible try to correct transient errors in software systems.

2.3.1 Single-thread approaches

2.3.1.1 Source-to-source based transformation

One approach in pure software hardening technique consists of transforming high level code by adding redundancy in the code or in the data. Based on a set of rules the source code is transformed by duplicating each variable and instruction. Typically, Rebaudengo et al [RRTV99, GRRV03, CNV+00] proposed a DWC approach to modify a high level code at compilation time. Data and code are duplicated to allow soft error detection by comparing duplicated data. That approach is able to detect soft errors. In [Nic02] B. Nicolescu presents a software approach to detect transient errors in digital architectures. Here a translator transforms a regular C program in a hardened C program, also by duplicating data and instructions. Both proposals are low granularity sequential DWC approaches. However, Nicolescu only detects errors; he does not correct them. The result of that approach is a transformed source code, not a transformed machine code. So it is not architecture dependent and can be applied to any computer and architecture.

The drawbacks are that:

- The source code is needed before the software is hardened. Thus the approach is not applicable to available binaries codes.
- That approach could not detect and correct low level errors such as transient faults triggering unknown exceptions or the reset of the CPU.
- The overhead on code size is around 4 times.

2.3.1.2 Compiler based transformation

SWIFT [RCV+05] (SoftWare Implemented Fault Tolerance) is a compiler time approach which duplicates the instructions in a program and

inserts comparison instructions at strategic points during the code generation. It is also a DWC approach but with a coarser granularity.

2.3.2 Multi-threaded approaches

Several authors proposed comparable approaches but exploiting multi-threading facilities to accelerate the process.

2.3.2.1 Source-to-source based transformation

[OKB+16] use a completely different approach exploiting Intel MPX (Memory Protection eXtension) and TSX (Transactional Synchronization Extensions) facilities to provide tolerance of faulty pointers. MPX is an Intel extension to check pointer bounds. So they use MPX instructions to detect some bit flips in pointers. TSX is another Intel facility. It allows optimistic mutual exclusion with rollback in multi-thread programs. They used it as rollback to recover from a faulty pointer. The interest of this approach is that it mitigates errors that might not be detected in programs protected by any of the preceding proposals. These programs can recover from computational results but may still crash or have an inappropriate behavior in case of pointer or jump errors. This work can improve the situation for pointer errors but does not eliminate all risks with pointers because it is limited to bounds checking.

Interesting work related to pure software hardening can also be found in other fields, such as security. A. Barenghi et al. [BBK+10] introduced instructions duplication and triplication as countermeasures to bit flip injection attacks against the AES algorithm. He exploited the fact that the AES algorithm on ARM processors uses only 9 out of the 12 available registers to duplicate instructions. It is an instruction level approach targeted at a specific piece of code.

2.3.2.2 Compiler based transformation

[ZLJA12] developed Decoupled Acyclic Fault Tolerance (DAFT) another fine grain compiler level fault-tolerance but using simultaneous multi-threading facilities. DAFT replicates all instructions in the original program except memory instructions and library function calls. The duplicated code is spread between a leading and a trailing thread. The memory operations are executed once by the leading thread and are made available as input values to the trailing thread. The same mechanism is used for library function calls. Instructions for fault checking

and communication between the two threads are inserted by the DAFT compiler. DAFT was implemented on the LLVM compiler framework and was evaluated with a mix of SPEC CPU2000 and SPEC CPU2006 on a six-core Intel Xeon X7460 processor with a 16MB shared L3 cache. DAFT used Speculative Decoupled Software Pipelining to gain performance and reduce the cyclic dependencies between the trailing thread and the leading thread. So rather than waiting for the trailing thread the leading one continues its execution. Speculation allowed to announce a reduction of performance overhead of DAFT from 200% average to 38% (or 1.38×) on average. But the speculation wakes up new challenges for detecting faults and ensuring correct execution. Challenges for which Yun Zhang et al [ZLJA12] proposed solutions. The announced fault coverage is 99.993%.

[CC16] proposed a coarser grain TMR framework to develop a fault tolerance program using multi- threading. The original program is replicated in several threads, a watchdog thread is used to recover from non-respondent threads. A majority voting protocol is used to recover from faults. The implementation was based on the POSIX Pthread library. The approach was evaluated with DSPstone benchmark on an Intel 3,4GHz corei7 3770 processor. A majority of injected faults were recovered.

The interesting feature, beside using execution duplication and triplication as protection against bit flips, is taking into account available hardware resources, which characterizes the techniques presented in the next section.

2.4 A short survey of hybrid hardening techniques

Hybrid techniques add leveraging available hardware features of some COTS processors to the techniques used in pure software hardening.

K. Li was the first who used copy-on-write to provide check-pointing capabilities to parallel shared memory systems [LNP94] [LNP90]. This technique allows to keep the state of the last checkpoint and work on a copy of pages to modify when continuing the execution beyond the checkpoint.

[Rot99] developed a micro-architecture method called cooperative redundant threads using the combination of time redundancy and instruction redundancy to provide fault tolerance on a simultaneous multi-

thread architecture. Like in [ZLJA12] , the program is divided in two threads, The A- stream which runs in advance of the R-Stream (redundant). The operating system is not aware of the existence of the R-Stream. The result from [Rot99] the A-Stream is saved in a delay buffer. The result produced by R-Stream is compared to the contents in the delay-Buffer. An error is detected when the comparison does not match and the previous committed state of the R-Stream is used for checkpoint. Otherwise the result of the R-Stream is committed. This is DWC at instruction level implemented in the micro-program of a CISC computer. It is a hybrid technique implemented within the CPU. It is thus not "off the shelf hardware". Besides, not all SEU effects will be detected, even if one assumes that the central memory is adequately protected by other means. One can assume that the two pipelines will not be disrupted by the same SEU, but other parts of the CPU could be hit: only "gentle" errors occurring in a pipeline will be detected and corrected. For instance, an error in an address register when the result is written back to memory will not be detected, so the committed state of the R-Stream could be faulty. A crash of the processor is not detected either. This work is very similar, in its principles and possibilities to [ZLJA12], but it is implemented in the micro-program. This approach was tested on a simulator of the proposed architecture.

H. Mushtaq [MAAB12] developed a user level library to provide a fault tolerance capability to user applications. The code of the user application is modified by using the multi-threading capabilities of recent architectures. 4 threads are used for fault detection: a watchdog thread, a checking point thread, a leader thread and a follower thread. A shared memory is used for communication between these threads. Checkpoint/rollback is used for error recovery. The execution is divided in 1 second processing elements, called "epochs". At the end of each epoch, the follower and the leader compute a checksum that are compared to check the presence of transient errors. This checksum is computed efficiently using the extension SSE4.2 which is available in Intel core i processors since the Nehalem generation (2007).

In case of error, the leader and the follower are killed and the checking point thread is activated. The latter creates a new follower and a new leader from the last checkpoint created at the end of the previous epoch. The dirty pages are identified by giving only read access to each thread at the beginning of the interval. When the thread tries to access the page for writing, the kernel sends a signal to the thread, so the page is noted and a write access is given to the thread. This dirty page identification

mechanism is similar to the one implemented in this thesis.

The announced performance loss is 46%. The test was performed using 5 benchmarks, one from the PARSEC and four from the SPLASH-2. The machine used to run these benchmarks has 8 cores (dual socket with 2 quad cores), 2.67 GHz Intel Xeon processor with 32GB of RAM, running CentOS Linux version 5, with kernel 2.6.18 [MAAB12] . An improvement of that technique was described in [MAAB13] where the CRC32 instruction of SSE4.2 is used for comparison. That is a significant improvement because rather than comparing pairs of pages, they used hardware facilities to compute the data CRC of each page and compared the CRC. This contributed significantly to reducing the overhead from 46% to 18%. Beside, the locality property of memory is exploited. So when the signal comes, rather than giving access to one page, access is given to N pages.

The program is not protected against indirect effects of a SEU, because all writable memory of the process is available during the hardened execution. Indirect effects of SEU are when an SEU occurs and the process misbehaves and writes faulty data in its memory space. Also when the fault causes an exception resulting in the operating system killing the faulty process, the whole process will be killed (the leader thread, the follower thread and the checkpoint/rollback thread).

It must be noted that this work uses very long processing elements. It is a coarse grain multi-threaded DWC approach. The original features are the comparison of dirty pages for error detection and the coarse grain. Since the user program is anyway modified, enforcing the two threads to execute exactly the same instructions is not a hard problem here.

Lesage et al [LML11] use a hardening technique on which the present work is based. Their work was targeted at stand alone programs on small processors, without multi-thread facilities. Like in preceding pure software and software based methods, double execution of processing elements, with comparison (DWC) is implemented in software to detect and correct SEU errors. Processing elements duration's is medium (millisecond range). Their duration is tuned so that the probability of more than one SEU occurring during the handling of one processing element can be neglected for the technology and environment of the system. It was the first DWC hardening technique where the duration of processing elements is not dictated by the hardening solution but by the constraints of the environment (radiation level and the sensibility of the hardware).

It used a simple hardware protected memory (the protection hardware is implemented in a FPGA) The protected memory could not be

modified, except by using a complex writing sequence that programs misbehaving after being disturbed by a SEU will not execute "by chance". The availability of protected memory, that cannot be modified as direct or indirect effect of a SEU, allows to keep safely the program's code and a snapshot of its variables taken before starting the current processing element. The use of protected memory is a second major difference with the preceding methods.

Crashes and exceptions are detected by the standard exception handling mechanism of the processor. In that case, the processing element is also restarted from the last snapshot (after a hot reset keeping the contents of the protected memory in case of system crash). This is a third major difference with the preceding methods. The three innovations were possible because the whole computer was controlled by the stand alone program.

The program has to be written for hardening by calling appropriate functions at appropriate points in its execution, but this was considered acceptable for new small stand-alone programs. The work of Lesage et al. [LML11] was validated by fault injection, both by simulation and in the real device. It gave excellent results but with a huge time cost.

Björn Döbel [Döb14] developed Romain, a framework that provides transparent multi-threading redundancy as operating system service to detect and correct transient hardware errors. The technique was implemented on the Fiasco.OC micro-kernel. Each program written to run on the Fiasco.OC micro-kernel can be replicated in several threads. Each thread runs in its own address space. The number of replica is configurable. A master thread is responsible for managing all replica, comparing their states and recovering if needed. There is only one master thread for each application. Master threads are able to communicate with each other. All interactions of the replica with the outside world (exception, interrupt, system call: called "externalization events)" are done via the master thread. The checking point off all replicas is when they perform a system call or an exception is raised. The master thread compares the "external" state of each replica, i.e. the registers and the parameters of the system call. An error is detected when a total match is not found. A voting mechanism is used to recover from the error. The interesting features of this method are:

- each thread runs in its own address space. Thus if a thread misbehaves it can only harm itself a little more, which is not important because a misbehaving thread's memory will be discarded anyway;

this advantage is similar to that of protected memory in [LML11]

- TMR or more redundant N-out-of-M voting schemes are used, which means that the processing element will not have to be replayed because the good result will be provided by the voting mechanism, but the price to pay is to execute each processing element at least 3 times, which is less efficient than DWC if few processing elements have to be replayed;
- exceptions are processed and recovered from like in [LML11] ;
- the code of the program to harden does not need to be modified, which is a tremendous advantage over all previous methods;
- the processing elements are delimited by internal events of the program: system calls and exceptions, which means that the hardening system cannot enforce processing elements to be short enough to only have to handle a single SEU; on the other hand the number of replicas can be increased for programs risking several SEU in the same processing unit.

However, the major limitation of this approach is that the comparison only involves the part of the state that is externalized, i.e. visible outside the process. In other words, the replica agree only on their output. Thus, undetected errors could accumulate and, later, diffuse in all replicas if, after another SEU, the "silently" corrupted replica is selected as reference.

Frenkel [FLB15] used a different approach, based on a COTS FPGA (Xylinx Zynq Soc) and three new techniques: Forward Temporal Redundancy (FTR) at circuit level, frame and module dynamic partial reconfiguration (DPR) and offloading a circuit state preservation structure based on checkpointing and rollback. This is achieved with only an 85% combinational and 125% sequential overheads. Detection and correction latencies are of 4,5 ms and 320μs respectively.

The approach was validated by fault injection in a simulation and also Proton beam tested. It gave excellent results.

2.5 The blended hardening technique (BHT)

The present work is based on the same principles as Lesage's work. These principles, called "blended hardening technique" we reviewed here independently of Lesage's implementation. Both Lesage's work, published in

2011, and this work are steps of a research project on the applicability of blended hardening to full scale computing systems.

The motivation of [LML11] was to eliminate all SEU effects in a standalone program divided into processing elements.

The aim of this thesis is to contribute to generalize this method to full scale computing systems, with their operating system and many processes.

The BHT uses available hardware facilities such as the exception mechanism present in most processors and the machine check architecture to detect hardware detectable transient faults caused by SEU and DWC to detect silent faults, i.e. faults causing erroneous results that are not detected by hardware.

The blended hardening technique is based on two fundamental assumptions [AL19, LML11] :

1. The rarity of SEU. It states that, below some duration limit, only one single SEU can harm in the system. This time period had been evaluated based on experimental data [Mat10] for the target hardware of [LML11]. In space conditions it was of the order of a msec.

   As a consequence, if processing is split in processing elements and if executing them twice and comparing the result (DWC) is short enough to be harmed at most once by SEU, then, this event can occur during one of the two executions or during the verification but not during more than one of these 3 parts of the handling of the processing element. If the two executions give the same results then an erroneous silent fault in the comparison can only produce the faulty conclusion that the two executions gave different result and the processing element will only be incorrectly restarted.

   This does not mean that a single SEU may hit the system during the two executions and comparison but that if several SEUs occur, they will not mask each other effect, which could only happen if exactly the same fault was produced in each execution. Or if the results of the two runs differ but are considered as identical by a faulty comparison.

   This is called the "single SEU hypothesis" because even if several SEU occur the effect will be the same as if there was only one and the PE will be restarted.

2. The availability of a "protected memory" area that is totally im-

mune to all consequences of SEU, both direct and indirect. This makes it possible to keep trusted snapshots and resume processing safely from them; provided that the processing elements are both atomic (i.e. their effect is always complete or nothing, never partial) and idempotent (i.e. running them twice or more is equivalent to running them once).

In practice "protected memory" must be hardware hardened against SEU (e.g. ECC+hardened scrubbing) [BBN+07] and read only for runaway programs, i.e. programs gone out of control as a result of a SEU-aused error (indirect SEU consequence).

Assuming these two hypotheses, the hardened program is divided into short processing elements. If an error is detected by hardware at any time during the handling of the processing element the processor is restored to a stable state and the execution of the processing element is eventually restarted. Silent errors are detected by DWC:

1. Each processing element is run twice.
2. The results from these two runs are compared.
3. If they do match, the result is saved securely as a snapshot into the protected memory and processing proceeds with the next processing element,
4. else processing is restarted from the last snapshot saved in the PRAM (Protected Random Access memory).

If a SEU occurred, it is detected, either by the exception mechanism of the processor or by comparison of the results of the two executions of the current processing element; if the SEU impacted comparison, it can only be a false negative causing an unnecessary restart of the processing element, but never a false positive because this would violate the single SEU hypothesis. I/O must be hardened independently.

The effectiveness of the technique has been shown experimentally by Lesage on a small stand alone program running on a Linux simulator and also on a hardware prototype. The Linux simulator was a collection of processes exchanging data through shared memory segments, and sending signals to interact with each other. The hardware prototype has been used and applied in an ESA proof-of- concept project with Thalès Aliena Space ETCA on the feasibility of digital control for power management aboard satellites. The software was implemented on 2 naked LEON

computers. The 2 boards were running real-time partially-hardened software, exchanging messages with each other. Error injection was done through the specific FPGA implementation [LML11] .

2.6 Discussion on pure and hybrid hardening techniques

With the notable exception of [FLB15], that targets sub-instruction level and is applicable to any kind of software as long as it runs on a specific FPGA-based system-on-a-chip, pure software or hybrid hardening techniques had only been targeted at standalone programs or single application programs and not at whole computing systems including multiple tasks or threads and an operating system.

There are several reasons for that. The first reason is the high overhead induced by fault-tolerance tasks implemented in software. However in many instances this performance loss is affordable because the performance of the processor is much higher than needed. Programmers take profit of this advantage to provide more reliable software systems in other contexts than SEU hardening. For example to ensure data integrity, file systems use sophisticated algorithms such as CRC, RAID [Pla80]. The overhead induced by these computation has been balanced by the development of the processor industry which provides high speed and high accuracy processors.

The second reason could be the high complexity induced by these methods. It is expected to be complex to maintain the consistency between duplicated data and their states when there are multiple states in different processes to keep consistent with the overall system [RRTV99]. Besides, hardening code must be added and the size of data is at least doubled. Also to ensure the hardened execution of the processing element as proposed in [AL19, LML11], the hardening code should keep the memory state of each runnable process as it was at the start of the current processing element and restore it when the process is ready to run the processing element within the process for the second time.

Third, except in Döbel [Döb14] and Frenkel [FLB15], where the code of the program needs not be modified, the code of the program to harden must be modified, either by hand or by a tool. These modified programs must respect the functional specification of the original program and its timing requirements.

Fourth, so far, only data processing has been discussed. Other prob-

lems can arise from interactions of the software to harden with an environment that is not hardened. For instance, in [LML11] what happens when I/O operations are run twice? Is it always correct with the I/O communication protocol? How to adapt the technique to make the hardening action transparent to the system interaction with its environment?

Fifth, how to guarantee that the hardening code is also hardened? Are his data correct every time? What happens when the hardening code is also victim of SEU and causes an exception. In [LML11] the rarity assumption ensures that only one SEU can occur during the DWC execution of a processing element (run twice, results compared, results saved into the protected memory) [LML11]. Thus a SEU could occur either during one of the two execution phases, or in the comparison phase or in the saving phase. Even if the SEU induced an error within the hardened code and it is uncontrollable, the secure mechanism for writing into the protected memory ensures the integrity of saved data. Thus the runaway program is quickly killed (at the latest when the PE is stopped because it reached its maximum duration) and the program can be resumed from the previous safe state.

Sixth, the technique of running each PE twice ensuring the detection of any data or code corruption error is not always directly applicable: for instance, when it is related to maintaining the system time. How is it possible to distinguish the difference in results of the two runs due to the normal evolution of time or to a fault caused by occurrence of a SEU?

These are a few of the problems expected when a complete computer system with its operating system and application processes is to be hardened. Other problems will of course be identified when trying to solve the ones above.

2.7 Conclusion

The state of the art of fault tolerance, with focus on random transient faults such as SEU, was presented. Hybrid hardening approaches are promising alternatives to hardware hardening approaches. They are able to combine desirable features of both hardware and software techniques, and provide levels of protection impossible for pure software techniques without the cost of hardware hardened processors. The questions are: can they provide the needed level of protection using mass produced commercial off-the-shelf hardware and with an acceptable overhead. The

next chapter will present the assessment of the problem and some key processor architecture features that make possible implementing a hybrid hardening approach in this thesis.

CHAPTER 3

Assessment of the problem

3.1 Introduction

BHT aims to use COTS processors to provide fault tolerance capability to application processes. The assessment of the problem consists of an analytic study of assets and risks. Assets are the fault tolerance features present in some COTS processor architectures that could be used for implementing BHT. The risk assessment is carried out in two parts. The first part consists in analyzing the effect of the SEUs on control registers (control path) that do not directly affect the user process but can disrupt the operation of the system. The faults cause an exception, a reset, or an effect on data processing similar to a SEU on the data path. The second part analyzes the impact of SEUs on the registers of the data path. Here the fault will not necessarily cause an exception. A silent fault is also possible. If the single SEU hypothesis is respected, DWC will detect silent faults. However the risk exists that two faults could mask the effect of each other. The probability of such uncorrected faults will be evaluated.

3.2 COTS processors hardware hardening assets

Software and hybrid hardening techniques are intended to allow the utilization of "off-the-shelf" state-of-the-art processors in environments exposed to radiations. The work presented here is hybrid which means that it uses existing hardware features of state of the art processors. Because state-of-the-art processors are not all alike, a specific target had to be selected: recent Intel Xeon and Atom processors, because they already include useful Reliability, Availability and Serviceability (RAS) features. Some knowledge on the distinctiveness of these processors is needed to understand some aspects of this work. Therefore, a summary of the document "Performance Analysis Guide for Intel Core i7 and Intel Xeon 5500 processors" by Dr. David Levinthal is provided in Annex 1 [Lev09]. This part is common to most recent Intel processors [Lev09].

3.2.1 RAS: Reliability, Availability and Serviceability

The Reliability, Availability and Serviceability (RAS) of COTS processors is an important issue for processor manufacturers. There are roughly three categories of usage of processors: personal computers (smartphones, tablets, laptops, desktops,workstations), servers and embedded computers. Except for workstations for which server grade hardware is sometimes used, personal computers are idle, sleeping or turned off most of the time, either because their user is busy somewhere else or simply because a human being is immensely slower than a computer, which is waiting for input most of the time. This idleness has two consequences:

1. the processor will stay cold which makes it less prone to transient errors [PBR17] and ;
2. most errors will hit unused resources and have no consequences.

Processors designed for personal computers are optimized for speed and energy efficiency (particularly for notebooks). The manufacturer's target for high end personal computers processors is the field of gaming applications. For instance, presentations on new Intel Core I3, I5 and I7 emphasize performance enhancing architectural features. They never mention RAS, nor architectural features contributing to RAS, such as Machine-Check Architecture, Machine-Check exceptions, ECC or parity protection of memory cells in or out of the processors.

On the contrary, server processors are intensively used for massive computation and their RAS characteristics are considered as extremely important: both server unavailability and erroneous results can have serious financial consequences [LS17]. These processors are designed for performance but also for reliability, availability and serviceability. Intel produced already in 2011, an interesting white paper on RAS for servers [Pro11]. They quote a paper of Schroeder et al., on a large scale field study on errors in servers ram [SPW11]. A large fleet of Google servers was monitored during two and a half years, representing several millions of DIMM.Days (usage of a memory module during a day). All the servers in the study were protected by ECC, either SECDED (Single Error Corrected Double Error Detected, the most common ECC scheme) or IBM Chipkill technology [JSDK13] (there are now several similar technologies that interleave bits from several 4-bit-wide chips so that adjacent bits in a chip are never in the same word, which reduces by 42 times [SDB+15] the number of uncorrected faults: Extended ECC,

Chipspare, SDDC,DDDC [CY12, KF82])), that can correct up to 4 adjacent faulty bits in a word. The study found that 8% of the DIMM memory modules and one third of the servers were affected by Detected and Corrected Errors (DCE). However there were also errors that were detected but not corrected (DUE): this affected 1,3% of the machines and 0,22% of the DIMMs per year. And an uncorrectable error leads to a shutdown of the entire machine and the replacement of the faulty DIMM. Correctable errors are more frequent in some DIMMs and these have a higher probability of experiencing an UE. Besides, the order of magnitude of the frequency of DCE was one order of magnitude higher than expected, based on cosmic radiations at ground level, and the distribution of these events was higher for some types of DIMMs and for some DIMMs in particular: in servers, the effect of cosmic radiations is much lower than that of manufacturing defects (this is why faulty DIMMs are replaced). This means that, even for space applications, the choice of DIMMs and tests to reject those that are more error-prone are important.

On servers, the state of the art of RAS aims at avoiding computational errors and crashes using improved ECC techniques able to correct several faulty bits, but also at identifying defective DIMMs before they are victim of an uncorrectable error and, in case of uncorrectable error in DIMM or in the processor itself, to detect it and to limit its scope. The way to try to reach these goals is by including a lot of fault detectors in the system (This is called the Machine Check Architecture) and report these events by exceptions (Machine Check exceptions) in order to let the software handle the event: logging it in case of error corrected by the hardware, or, if the error was not correctable by the hardware, letting the Operating System (or the BIOS, if the operating system cannot cope with the fault) try to confine the error to the process that was affected. If this process had been designed in a fault tolerant way, inform it of the fault to let it recover if it can, else kill it, avoiding to affect other processes. If the processor appears to be unstable, log as much info as possible for later debugging and shutdown the system. In multicore and multiprocessor systems, recent developments (Local Machine Check Exceptions: LMCE) make it sometimes possible to stop a single core rather than the whole system [Ngu17].

For servers, RAS characteristics of processors are advertised alongside their performance enhancing features [Cor18].

For embedded processors, faultless operation is even more important than for servers: faults can result not only in loss of time or erroneous

computations, but equipment can be destroyed and even people killed. Here performance is not the first objective: they are energy efficiency and RAS.

Presentation of RAS features in Intel documents is common for server processors, the Xeon line, and for embedded processors, the Atom line. While Xeons can be also be found in high end workstations, Atoms are also found in low-end notebooks, but Xeons are designed for servers and Atoms for embedded systems. Atoms are much cheaper than Xeons and their RAS features are thus more modest, but sufficient to avoid malfunctions, even though recovery requires more often software participation.

Before looking at what is actually implemented in current processors designed for RAS, it is useful to identify which hardware features are necessary to avoid crashes of systems or programs or faulty results of programs, assuming each RAS actor (hardware, firmware, operating system, programs) does what is expected of it in order to provide reliability, availability and serviceability; and which hardware features can be considered as optimization improving only the availability or the serviceability, i.e. shortening the recovery time in case of incident or providing useful information to the service team, for instance when the incident has been solved by the hardware without consequences for the software.

3.2.2 The central memory

The highest number of incidents are caused by central memory, just because it is the largest. They can be transient and random (such as those caused by cosmic radiations), transient but linked to a deficiency of a device, or permanent. All these faults can flip one or several bits of a word, in the same device, thus, at most n bits if n bit wide chips are used (n is often 4 or 8). Therefore protecting memory with parity or even SECDED or DECTED ECC is insufficient [CY12, KF82]. Techniques using only one bit of each chip in each word, such as chipkill are necessary to avoid errors caused by multi-bit faults. To avoid multiple events to accumulate, scrubbing is necessary and it must be organized in such a way that it does not introduce itself faults.

Hardware memory mirroring is sometimes used but it is only useful if errors are detected and if no faulty write operation can occur (because false data would be written in the two copies). TMR memories are apparently not used in COTS systems.

3.2.3 The caches

The second largest memory area to protect is the caches. There are several big differences between cache memories and main memories. One is that caches only hold copies of information, not the original (except during the short time between writing in write-back caches and when data are rewritten in central memories). This means that faults in caches must be detected, but not necessarily corrected when it is possible to re-fetch the good value from central memory.

Another difference is that central memory cells are one-transistor dynamic rams (DRAM) while cache memory cells are static memory (SRAM), because they are faster than dynamic memory. Static memory cells are 6 transistors flip-flops. Changing their state requires much more energy and, while a SEU hitting DRAMs could change the state of several contiguous bits of a word in the same chip by hitting several neighbouring transistors (this is why sophisticated error correction schemes such as chipkill are necessary), a SEU hitting a SRAM must not only be much more energetic to change its state, but, since the area of a bit is larger, the risk of changing the state of more than two bits in the same word is very low, because a much larger area must be affected to influence the state of transistors belonging to 3 bits than for one or two (two would be affected if the SEU hits just between transistors of the two bits). Therefore simple parity checking in caches will detect most SEU induced faults, SECDED ECC will detect all faults and correct one bit faults and DECTED will make the cache errorless in case of single SEU. Sridharan reaches the same conclusions (not those for DECTED that were not covered by his measurements) by measurements on two HPC systems: Hopper (located at Oakland, California, low altitude) and Cielo (located at Los Alamos, New Mexico , altitude 2400 m) [SDB+15]. SEU in caches are corrected or detected the next time the word is read in the block. If a fault is only detected, a hit to the block is handled as a miss and the block is re-fetched from the next level cache or from the central memory.

3.2.4 Processors registers and logic

Transient faults can occur not only in memory and in caches but also in processors registers and logic, in buses and in I/O devices. In processors, they can happen in the data path registers and logic, resulting generally in erroneous results. Like for caches, these faults must be detected and the current instruction must be aborted and, if possible restarted. If

the fault is detected by hardware, exception will occur. If the fault, is not detected by hardware it will in the case of blended hardening be detected by an exception later or by DWC.

Transient faults can also happen in the control path and, in the worst cases, leave the processor in an inconsistent state. The important here is to confine as much as possible the consequences of the fault and inform the software. Faults in the control path are detected by hardware and cause an exception, or CPU reset or an unrecoverable corruption of the core (or the whole processor) requiring an external reset. In the case of multi-core and multi-processor machines, whenever possible the consequences must be contained to the local core[Pro11].

3.2.5 Buses and I/O systems

Bus transfers must also be secured with error detection and correction systems. Buses are the links used to interconnect the different part of the computer to each other, the memory to the CPU, the CPU to the IO system, the IO system to the memory. Data goes through the bus system when it is read or when it is written. During the transferring, data may be victim of transient faults. Thus the arrival data could be different from the original data. An integrity check is then important on the bus system to ensure data integrity during their transfer. Error reporting is also important to inform software of data corruption when it happens. That is possible with the new generation of Intel bus systems, QuickPath Interconnect (Intel QPI). Intel QPI is an advanced bus system implementing routing mechanism to interconnect multi-cores to each other and to the IO hub. Intel QPI is a high performance communication system which implements an integrity check with CRC on data travelling through its links. When an error is detected the Intel QPI Retry feature is used for error correction by re-transmitting the failed data [Pro11].

Securing IO means, on one side, avoiding information change within the computer but also making sure that the correct information is exchanged with the external world. One particularity of securing IO is that IO is generally processed by the OS before being transmitted to a process.

3.2.6 Short presentation and discussion of Xeon RAS features

Intel processors are compatible with their ancestors, so features of a generation of architecture will remain present in successor architecture unless Intel has a very good reason to change. And the change will then be abundantly discussed and justified in the presentation of the new architecture. More generally, when new features are introduced, most documents on the new generation of architectures will focus on the new features In 2011, the Xeon E7 (Westmere micro architecture) caches and TLBs had the following characteristics that were discussed in the document describing this architecture [Pro11]. It can be assumed that newer architectures have at least the same level of protection even if it not specially documented.

- 48-bit virtual addressing and 44-bit physical addressing;
- 32 KB level 1 instruction cache, virtually indexed, physically tagged, with ECC (SECDED);
- 32 KB level 1 data cache, virtually indexed, physically tagged, with parity protection ;
- or 16 KB level 1 data cache virtually indexed, physically tagged with ECC (SECDED) on data and on tag;
- 256 KB L2 instruction/data cache with ECC (SECDED);
- 30 MB LLC (Last level Cache, in this case L3) instruction/data inclusive cache with ECC (DECTED on data and SECDED on tag.

There is no explicit indication on the protections mechanisms of TLBs, registers etc. neither in the data-sheet nor in the software developers manuals of the Xeon 7, and this is understandable because developers of peripheral hardware, of firmware and of software are not supposed to interact with these protection mechanisms, except through the machine check architecture that will be discussed below. However, the Intel white-paper *"Intel Xeon processor E7 family: Reliability, Availability and Serviceability"* published in 2011 says explicitly (appendix B)Âă: "*ECC is used to protect processor registers, processor caches and system memory from transient faults that can corrupt data without damaging the hardware. Besides, if TLB faults occur, they are reported in the*

Machine Check Architecture Model Specific registers, which implies at least a detection mechanism" [cgIc11]. In the paper of M. Natu "Autonomic Foundation for Fault Diagnosis", one can read *"The MCA feature provides a mechanism for detecting and reporting hardware (machine) errors such as: system bus errors, memory errors, parity errors, cache errors, and Translation Look-aside Buffer (TLB) errors"* [MN12].

Moreover, Local Machine Check Exceptions (LMCE) [Ngu17] have been recently added to the Machine Check Architecture in order to avoid perturbing all the cores when a fault affects a single core. This is in line with providing the level 3 cache, which is shared between the cores with a better protection (DECTED instead of SECDED ECC) than the Level2 cache, whose access is limited to the local core and the L1 cache that only serves the current process. In other words, protection against faults that can have far reaching consequences is higher than against faults that can only influence the current process or the local core. Bearing this in mind, what can be the consequences of a fault affecting a TLB, and, in particular, the Page Frame Number Field? This is the type of fault that could have the widest reach in a computer system because it could cause an erroneous write operation anywhere in physical memory, thus inducing an error not only in other cores of the same chip but in any core of any processor sharing the same central memory. And, because access restrictions are based on virtual addresses, there is no way to stop such an erroneous access once the PFN has been issued by the TLB. Therefore, to be consistent with the rest of the architecture, the TLB must have the highest level of protection, i. e. at least DECTED on the PFN array. In that case, because the TLB is static memory, an unstopped TLB fault should only be possible in really extreme events.

3.2.7 Exception management in recent architectures

Recent architectural changes such as increasing the size of the level 2 cache, not sharing it with the level 3 (That is now only fed by data ejected from the level 2; new data being now directly imported in level 2, and not in the non inclusive level 3) [Mul17] do not change the above conclusions: TLBs, level 2 and 3 caches, as well as central memory can be adequately protected against direct effects of SEU in XEON (and ATOM) COTS processors, but faults caused by SEU are still possible when other parts of the processor are hit. Most uncorrected faults will be reported by Machine Check Exceptions, but not necessarily all and if they are reported, current OSes usually only log the event and, if they

can, warn or kill the malfunctioning tasks and try to contain the error so that other processes and cores are not affected. Very few programs are written in such a way that they can recover when warned of an error. The purpose of using blended hardening in the operating system is to let the operating system itself correct the errors so that it is not any more necessary to warn or kill the affected process. It exploits and complements the hardware RAS features built in some COTS processors.

Some RAS features are very important for blended hardening. The extremely low probability of errors caused by SEU hitting TLB means that TLB can be trusted to implement PRAM. The extensive fault detection mechanism implemented in the machine check architecture and the detailed reports provided by the MCE make possible for the operating system to handle these events. The efficient protection of the level 2 and 3 caches reduce drastically the surface sensitive to SEU in the processor.

3.3 Risks related to systems and control registers

Beside the context of the running program (which is the set of saved values of the registers needed to restart the process when it is preempted, considered as the process's private registers)

1. there are unnamed register hidden in the data path and data path logic (in pipelines etc);

2. system registers and control registers which are managed by the operating system, that can be corrupted by SEU.

All these registers are announced to be ECC parity protected [Pro11]. But the logic interconnecting the registers is much harder to protect and could result in inserting wrong values in protected registers (ECC protects values staying in registers or memory words but is helpless against faulty values being injected in ECC protected registers). One can deduce from this that the more frequently the value of a register is modified the higher the risk of injecting a faulty value. Therefore if all registers are ECC protected, errors in the data path will be much more frequent than in control registers. However the consequence of faults in control registers will be discussed in the following. Although these errors should be much less frequent than errors in the data path resulting in faults being injected in context registers.

Some of the control registers are initialized when the operating system is loaded and remain unchanged as long as the computer is up. The others are mode dependent, they are initialized when the CPU is switching for one mode to another mode. Some are structural for the system, this means that they are used to store key values of the system components such as addresses of the system tables in memory (interrupt descriptor table, local descriptor table, global descriptor table etc).

This section is based on reference [Pro11] and has been verified by injecting faults in these registers to see what happens. The tests were done on Intel(R) Core(TM) i5 CPU and on Qemu running on Intel(R) Core(TM) i7 CPU L640/2.13GHz.

A running task also has characteristics specific to it, for example the privilege mode or the features to which it has access.

Control registers (CR0 to CR4) allow to configure the processor so that the tasks and the system can operate correctly. The values of these registers do not only affect the operation of the current task but the operation of the whole system. The control registers CR0 to CR4 define the characteristics of the running task and the processor state. They are only available at privilege level 0. If a SEU hits them and changes their state (this should not have bad consequence because they are ECC protected) or if because of a SEU, on the writing logic for instance, a faulty value is written in (as explained above, ECC-like protection does not help in this case), the system will be in an unstable state. For example assuming that the paging is enabled (bit 0 and bit 31 of CR0 are both set), if one of these bits is toggled, the paging will be disabled while the operating system is running in a mode where paging is supposed to be enabled. That will generate a general protection exception. Another example is the bit 6 of CR4 which enables machine check exception when it is set, so if a SEU disables it when it is set, a lot of machine check exceptions could be lost. However, because these registers are ECC protected and rarely modified, this kind of events should be extremely rare.

The memory management registers such as GDTR, IDTR, task register, and LDTR specify the location of the data structures used to manage the protected mode. A SEU can modify their values. That can evolve a special exception. For example if the location of GDTR is modified, the operating system will not be able to restore to a stable state. That will result in a reset of the CPU.

Although the risk of modification of control registers because of SEU is low, the next sections will evaluate what could happen.

3.3.1 CR0 register

This register contains bits that control the state of the processor and its mode of execution. It consists of 32 bits in the case of the IA32 architecture including 11 bits that software can handle and 21 reserved bits. Table 3.1 shows the description of each bit and what can happen when one of them is erroneously modified because of a SEU.

Table 3.1: Effect on the system when CR0's bits are modified by SEU

Bits	**Description**	**Effect on the system**
0	1 enable protected mode. 0 enable real mode	General Protection exception
1-3	controls the CPU's handling of an exception while one of the following modules has current processing (x87 FPU / MMX / SSE / SSE2 / SSE3 / SSE4	nothing to report
4	When set indicates that the processor supports the Intel 387 DX math coprocessor	nothing to report
5	Controls the X87 FPU error report mode	nothing to report
6-15	Reserved	nothing to report
16	When set Prevents the task running in ring 0 from modifying a read-only memory	nothing to report
17	Reserved	nothing to report
18	Enable automatic alignment checking	nothing to report
19-28	Reserved	nothing to report
29	Controls the cache write through properties	General protection exception
30	When set disables the cache	nothing to report
31	When set enable paging	Error report in Qemu

Changing most bits in the CR0 register generates either exceptions or has no effect. Bit 31, which controls the addressing system, puts the system in an unstable state. All these errors do not result in a reset of the processor but are reported to the operating system. The latter can therefore handle them in the appropriate manner to allow the system to continue running.

3.3.2 CR2 register

This register contains the linear address at which a page fault has occurred. It is not editable by software. When a SEU changes one or more bits in this register, that would lead in the worst cases to an erroneous handling of the page fault. Three cases are to be considered:

- The SEU modifies on or more bits in the offset part of the address. The page fault could be correctly handled without detection of the fault.
- The SEU modifies either the page directory entry of the page or the page table entry of the page but the wrong value points to a valid address within the process memory space. Since the process does not want to access this page a new page fault will be triggered which would be handled properly in this case.
- The SEU modifies either the page directory entry of the page or the page table entry but the wrong value points to an invalid address outside the process memory space. In this case the process will be killed because that invalid address violates the memory management policy.

3.3.3 CR3 register

This register contains three types of information: the root page table address, the PCD and PWT bits that control the caching mechanism (cache L1, cache L2, and cache L3) of the data structures of this page table. A bit change of this register can

- Either corrupt the address of the root page table. That would most probably cause a simple page fault when the current task is not running in ring 0 or a reset of the processor in when the task is running in ring 0.

- Disturb the caching mechanism of the data structures of this table of pages. That will cause the cache to be disabled or enabled incorrectly.

The table 3.2 shows what happens when these bits are modified.

Table 3.2: Effect on the system when CR3's bits are modified by SEU

Bits	**Description**	**Effect on the system**
0-2	Reserved	nothing to report
3-4	control caching	nothing to report
5-11	Reserved	nothing to report
12-31	Linear address of root page table	CPU reset

3.3.4 CR4 register

The CR4 register enables or disables processor features. Most of these features are not used within a classic OS. These features are used by the operating system. Since the operating system always restores the actual value of the register at context switch, a change in the register when running a user program will have no impact on the system. Errors may be confined within the current task. The following table 3.3 shows the impacts on the system when these bits are modified by SEUs.

Table 3.3: Effect on the system when CR4's bits are modified by SEU

Bits	**Description**	**Effect on the system**
0	Enables interruption and exception handling in virtual-8086 mode when set	nothing to report
1	Enable hardware support for virtual interrupts when set	nothing to report
2	Restricts the execution of the RDTSC statement to ring 0 tasks when set	nothing to report

3	Allow to support the DR4 and DR5 register when set	nothing to report
4	Enable use of 4Mb page when set	CPU reset
5	Enable use of more than 32 bits physical address when set	CPU reset
6	Enable machine check exception when set	nothing to report
7	Enable the global page feature when set	nothing to report
8	Enable the execution of the RDPMC instruction for tasks running in any ring when set	nothing to report
9	Indicates that the OS supports FXSAVE and FXRSTOR statements	nothing to report
10	Indicates that the OS supports the handling of unmasked SIMD floating-point exception	nothing to report
11-12	Reserved	General protection exception
13	Enable VMX operation	nothing to report
14	Enable SMX operation	General protection exception
15	Reserved	General protection exception
16	Enable RDFSBASE, RDGSBASE, WRFSBASE, WRGSBASE instructions	General protection exception
17	Enable process-context identifiers	General protection exception
18	Indicates that the OS supports XSAVE, XRTOR instructions	General protection exception
19	Reserved	General protection exception

20	Enable supervisor mode execution prevention	General protection exception
21-31	Reserved	General protection exception

3.4 General purpose registers, segment registers and eflags register

3.4.1 General purpose registers

The number of general purpose registers is 8 (EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI). These registers are used for basic operations, procedure calls and system calls. When the values of these registers are changed by a SEU that may cause a miscalculation error or exception. In the case, for example, of the SP and FP registers which control the management of the stack, a modification of the values of these registers because of a SEU could point the stack to an inaccessible memory area which could generate a page fault. That will be the same for SI and DI registers which often contain operands addresses. A fault in these registers could also trigger a page fault or miscalculation. The values in these registers changes up to once by instruction. They are thus the most sensitive to SEU effects. The faults in these registers are expected, detected and corrected by BHT (by replaying the PE).

3.4.2 Segment registers

Segment registers contain 16-bits memory segment selectors. A memory segment selector is an index that identifies the descriptor of the memory segment in the global descriptor table (GDT) or in the local descriptor table (LDT). The segment memory descriptor contains all the attributes of the memory segment (base, limit, privilege level). The segment registers CS, DS, and SS contain the selectors for the code, data, and stack, respectively. An erroneous modification of the 16 least significant bits of these registers would result in a modification of the selector. This could result in an erroneous selector or a valid selector that does not match the characteristics of the current task. In both cases an exception will be reported and the operating system will take the appropriate decision.

These registers are ECC protected. A SEU hitting them would have no consequence but even if it had, the effect would be contained to the

current PE and would be detected and corrected by BHT.

3.4.3 Eflags register

This 32-bit register contains status bits, control bits, and system bits. It impacts both the running task and the system globally. This register controls interrupt activation, single stepping and many other features. The tests performed by modifying the bits of this register only led to exceptions and miscalculations managed by the BHT. However, the modification of bit 17 which corresponds to the activation of the Virtual-8086 mode has resulted in a reset of the CPU.

3.5 Memory management register

3.5.1 GDTR and IDTR

The GDTR register contains the linear address of the global descriptor (32-bits) and its limit (16-bits). A modification because of a SEU of one or more bits in the linear address of this table would result in a reset of the processor. Because that table contains the memory description of the whole system, including the operating system memory. However, a change in the limits just causes a restriction of access to a valid segment descriptor. The same analysis is valid for the IDTR. The IDTR contains the linear address of the interrupt descriptor table. That table contains selectors which point to the global descriptor table for their corresponding segment descriptor. A corrupt address in the IDTR will lead to a reset of the CPU. These registers are rarely modified thus not much at risk.

3.5.2 LDTR and TR

The LDTR and TR registers contain a segment selector (16bits), the linear address of the table (LDT or task segment), the limit and the attributes of the segment. The segment selector is loaded into LDTR or TR by the OS with LLDT or LTR instruction. The processor automatically loads in the register the linear address of the table, the limit and the attributes of the descriptors. If a SEU changes the selector value this could lead to either an invalid selector or a selector that does not match the task. In both cases, either it will generate an exception or a page fault. If a SEU modifies the linear address of the table this could

lead to either a page fault or an exception. If a SEU modifies the limit if it leads to a decrease in the table size, valid descriptors will become inaccessible this could lead to a page fault or a general protection fault. If this leads to an erroneous increase in size, this will be undetected until the next context switch.

3.6 Risks related to undetected SEU effects

Tolerating transient faults in a computer system means detecting the faults and correcting them. However a fault could occur during detection or during correction. Another problem is that a fault could hide the effect of another fault: the typical simplistic example is a second faulty bit in a parity protected word.

This thesis is part of a project aiming to apply the blended hardening technique to complete computer systems. The specific problem addressed in this thesis is preventing all effects of SEU occurring during the execution of application processes over an operating system. The operating system itself is assumed to be immune to SEU effects: it is assumed to have been modified to protect itself.

In order to limit the work to a size compatible with a PhD thesis, only a single processing core are considered. Adapting Minix3 and the result to multicore systems is future work.

One of the principles of the blended hardening technique is to organize fault detection and correction in such a way that one has to cope with a single fault at a time during the whole process of detection and correction of the fault. This will be true either if only a single fault occurred or if several faults occurred but do not mask each other, i. e. that the fault will be detected as easily as if it were a single fault. SEU are uncorrelated random events and, compared to the clock rates of current processors, they are relatively rare, even in space, (although at human perception rate, they are frequent in space). Therefore one should be able to find a time interval sufficiently short that it is possible to have to cope at most with a single SEU (or several not hiding each other) and sufficiently long to allow a processor to do a significant amount of work during this time. One can divide the transient faults induced by SEU in 5 categories:

- those that hit unused hardware: nothing to do about these;
- those that are automatically corrected by hardware (such as ECC): these are not harmful except if they accumulate, which can be

avoided, e.g. by hardened scrubbing in the case of ECC: this will not be discussed in detail in this thesis;

- those that are immediately detected by hardware and cause an exception: the correction of these must be initiated by the exception handling function. Designing efficient functions to correct faults detected by hardware is part of this thesis.
- those that lead to an infinite loop or block the execution: there must be a kind of time-out mechanism to get out of such situations. This too is addressed in this thesis.
- those that silently cause computational errors in the blended hardening technique, these are detected by DWC. If the results of the two executions of a processing element differ, the correction is initiated.

When a corrective action must be initiated, the processing element is restarted after the necessary housekeeping, if needed: detecting the fault(s) is thus mandatory but identifying it exactly is often not necessary because restarting the PE will correct it. The risk of failing detection because two faults might hide each other effect only exists for the two last types of faults. The way to avoid this risk is by limiting the duration of the processing elements to keep to an acceptable level the probability of two faults masking each other. Because the duration of the processing element has been selected to be short enough for the processing element to be exposed to at most one SEU or several ones not masking each other to ensure the detection of the SEU or SEUs effect, the comparison phase will detect the difference in the results.

In the following section, the possible faults that could be induced by SEU in application processes will be identified, and the rate of occurrence of SEU induced errors will be evaluated in order to determine how long a processing element may be so that only at most a single transient faulty situation has to be corrected. Systems running in space environment are the main target of this chapter, those running in terrestrial environments are only considered for reference.

3.6.1 Possible SEU induced faults in application processes

The first case, depending on the duration of PEs is the infinite loop: it will be interrupted, at the latest, at the end of the maximum allowed execution time of the current execution of the processing element.

The second case depending on the duration of PEs is the processing element finishing the two runs apparently in a normal way, but producing different results. That will be detected when the results of the two executions will be compared. In this case the duration of the processing element is critical, because if two SEU could occur, they may not mask each others effect, i.e. produce two identical, but faulty results.

It must be noted that the case of the erroneous comparison must not be considered here, because comparisons are performed in the OS that is assumed to be immune to SEU effects. This part will be briefly discussed in the chapter about future work.

With the two hypotheses of the blended hardening technique (rarity of SEU and availability of protected memory), nothing prevents thus all SEU caused errors happening in application processes to be identified and corrected. Proving that it is possible and showing how to do it is the core of this thesis.

3.6.2 Analysis of BHT sensitivity to multiple SEU

For SEU detected by hardware, the duration of the processing element is irrelevant: as soon as the SEU is detected, the current run (first or second) is aborted and the PE is restarted.

For infinite loops, the shortest the PE, the faster the fault will be detected but there is no critical time limit.

However the duration of the PE is critical for the detection of silent faults. A critical parameter for detecting silent faults in the blended hardening technique is the maximum duration T of the two executions and comparison of results of a processing element in which the occurrence of more than one SEU or several ones not masking each other can safely be neglected.

In the case of hardening application processes by the kernel (which is assumed to be immune to SEU), it is not the real time but the user time, i.e. only time spent in user mode during the execution of the two runs that must be considered.

What happens during T can be seen on Fig 3.1 showing the time line of the handling of a processing element, where:

- T_a is the computation time to launch correctly the first execution of the processing element.

- T_b is the time to run the first execution of the processing element.

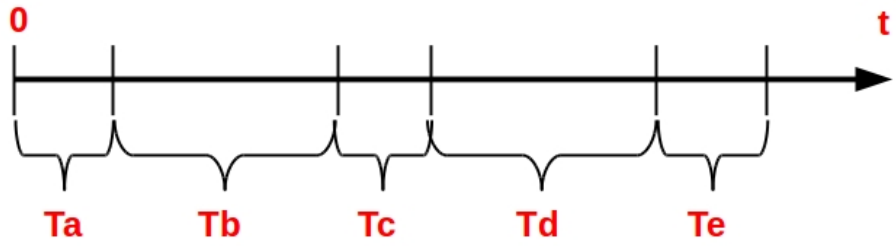

Figure 3.1: Phases of hardening a processing element

- T_c is the computation time to save the current state of the current processing element and restore the current process state as it was just before entering the processing element (time 0 Fig 3.1) and launch the second execution of the processing element.
- T_d is the time to run the second execution of the processing element.
- T_e is the computation time to compare the results of the two executions of the processing element and decide to proceed to the next or to restart the current processing element.

$$T = T_a + T_b + T_c + T_d + T_e \tag{3.1}$$

T_a, T_c and T_e are spent in the system, which is assumed to be immune to SEU. So, only T_b and T_d have to be taken into account for hardening application processes in the operating system,

$$T = T_b + T_d \tag{3.2}$$

The different possible scenarios producing faulty result are:

1. One or more SEU occur during T_b and none occurs during T_d.
2. One or more SEU occur during T_d and none occurs during T_b.
3. One or more SEU occur during T_b and one or more SEU occur during T_d and they produce different results.
4. One or more SEU occur during T_b and one or more SEU occur during T_d and produce the same erroneous results.

Of course, the delicate point is distinguishing scenarios 3 and 4. To obtain 4, the two SEU must hit the same area (in registers or data caches) when it includes the same data and toggle the same bits or trouble the execution of instructions that will eventually cause the same faulty changes.

Scenarios 1, 2 and 3 will be recovered by BHT because either an exception will occur or the two results will be different and it will be detected at comparison stage and the processing element will be restarted. Only scenario 4 can cause a wrong result to be accepted. The sum of the durations of the two executions of the processing element must thus be short enough for the probability of occurrence of that scenario to be negligible. The above probabilities can be computed.

3.6.3 Multiple Single event error rate evaluation method

It must first be remembered that the COTS processors considered here (those destined to high availability servers and mission critical embedded systems at ground level) are partially hardened and some of their parts are adequately protected. This is the case of level 2 and 3 caches and TLBs: faults caused by SEUs hitting these parts will be corrected by DECTED ECC or cause a machine check exception. They will either have no consequence or be handled and will not cause a silent error.

The purpose of this section is to identify a duration short enough to keep P_4 (the probability of occurrence of the fourth scenario above) under an acceptable threshold but not too short in order to avoid introducing too much hardening overhead.

One can start with an educated guess based on the values in [LML11] for an older technology and what could lead to an acceptable overhead, based on experiments on MINIX3 [AL16]. From this, P_4 can be computed and, if needed, the value for T_b and T_d can be adjusted. Let us assume that, for instance, that $T_b = T_d = 250\mu s$ and check if this would produce an acceptable error rate for BHT hardened software [AL19]. SEU are independent discrete random events. The number of such events occurring in a given time interval follows thus a Poisson law [FHR10, vB98], of parameter $\lambda.t$ where λ is the rate of occurrence of SEU (soft error rate, computable with tools such as OMERE [TRA18] and SPENVIS [HQSD00] for the number of bits of the device under test and t is the time interval where the events occur. The probability of getting k events in the time interval t is thus

$$P(N_t = k) = \frac{(\lambda.t)^k}{k!} \times exp(-\lambda.t) \tag{3.3}$$

The probability to have at least one event within time period t is:

$$P(N \geq 1) = 1 - P(N_t = 0) \tag{3.4}$$

$$P(N \geq 1) = 1 - exp(-\lambda.t) \tag{3.5}$$

The probability of occurrence of one or more errors in T_b and one or more errors in T_d is

$$P_0 = P_b \times P_d \tag{3.6}$$

Let $T_b = T_d$ and λ in errors/s in one bit and n be the number of bits in the processor.

$$P_0 = (1 - exp(-\lambda \times n \times t))^2 \tag{3.7}$$

The probabilities of occurrence of the 4 scenarios of section 3.6.2 are

$$P_1 = P_b \times (1 - P_d) \tag{3.8}$$
$$P_2 = P_d \times (1 - P_b) \tag{3.9}$$
$$P_3 = P_b \times (P_d \times (1 - \delta)) \tag{3.10}$$
$$P_4 = P_b \times P_d \times \delta \tag{3.11}$$

Where P_b and P_d are computed using equation 3.5, t is respectively T_b and T_d. And δ is the probability to have one or more SEU producing the same false results from the first and second execution of the processing element.

The most interesting value is P_4 because this is the residual error probability in a processing element after a successful application of BHT. The situations where SEUs hitting the two executions can produce exactly the same erroneous results are the listed below. In all of them one must remember that, because L1 caches are cleared before each execution of a PE and registers are re-initialized, the effect of a SEU in the first run cannot persist in the second run.

Knowing λ in errors per second (also noted SER, but then in errors per billion hours) (Note that it would be more consistent to say faults per second or per billion hours, but maybe because they are evaluated on unhardened devices, they are generally called errors). The following formulas are obtained for P0:

$$P_0 = (1 - exp(-\lambda \times T_b)) \times (1 - exp(-\lambda \times T_d)) \quad (3.12)$$

where λ is in errors/s/bit.
with $T_b = T_d = 2.5 \times 10^{-4} s$
From equation 3.12

$$P_0 = (1 - exp(-2.5 \times 10^{-4} \times \lambda))^2 \quad (3.13)$$

$N = 4MBits$ this is enough to cover all the processor registers (less than $200 \times 64bits$) and the L1 caches (I-cache+D-cache, typically 64Kbytes or 128Kbytes).

To compute P_4 different cases must be considered:

1. During each of the two runs, SEUs hit L1 program cache blocks holding the same instructions in the time interval between they were last used and the time they are re-used in the same place in the program by the PE. The SEU must modify the same bits in that instruction in the same way without making it an illegal instruction, that would produce an exception that would be detected. The time where the SEU could happen is up to the whole duration of the run, but the average length of a loop iteration is more realistic (because reusing the same instruction generally occurs in loops) and loop iterations are often rather short, much less than $100ns$ on the average. The SEU may hit any bit in a word during the first execution, but must hit the same bit of the same word during the second execution.

$$P_{4.1} = \frac{n_1}{w} \times (1 - exp(-\lambda \times w \times t_1)) \times (1 - exp(-\lambda \times t_1)) \quad (3.14)$$

   - $\frac{n_1}{w}$ is the number of words in the L1 program cache.
   - $(1 - exp(-\lambda \times w \times t_1))$ is the probability of hitting any bit in a word.
   - $(1 - exp(-\lambda \times t_1))$ is the probability of hitting a given bit in the word.

   with λ in events/bit.sec, t_1 is 100 ns and n_1 the number of bits in the L1 program cache and w the size of a word.

   Equation 3.14 does not take into account the possibility of producing an illegal instruction. That would be detected by exceptions. It is thus an over-estimation of $P_{4.1}$

2. During each of the two runs, SEUs hit a L1 data cache block holding a variable with the same value in the time interval between it was last used and the time it is re-used in the same place in the program by the PE. Here, three cases must be considered:

   (a) numerical or alphanumerical variables. In this case the SEU must modify the same bits in that variable in the same way. The time where the SEU could happen is up to the whole duration of the run (because this is the longest time a variable value may stay in the L1 data cache. The SEU may hit any bit in a word during the first execution but must hit the same bit of the same word during the second execution.

   $$P_{4.2a} = \frac{n_2}{w} \times (1 - exp(-\lambda \times w \times t)) \times (1 - exp(-\lambda \times t)) \quad (3.15)$$

   with λ in events/bit.sec, t is the maximum duration of a run of the processing element and n_2 the number of bits in the L1 data cache.

   (b) boolean variables. These are more tricky, because, if their correct value is zero (false), changing any bit to one will be interpreted as true, thus all bits must be considered also in the second run instead of just one. However, in most instances, booleans are produced by comparison instructions and used immediately in a conditional jump. Here again, assuming the average time between setting the value of a boolean and its last use may be assumed to be far below $100ns$.

   $$P_{4.2b} = \frac{n_2}{w} \times (1 - exp(-\lambda \times w \times t_2))^2 \quad (3.16)$$

   with λ in events/bit.sec, w the size of a word, $t_2 = 100ns$ and n_2 the number of bits in the L1 data cache.

   (c) boolean-like variables: these are typically numerical variables used in a > ,>=,<, or <= comparison, where the boolean result of the comparison depends on the values of more that one bit. According to an evaluation by Peter Kankowski in 2009 [Kan09], CMP instructions represent 5% of instructions in typical programs. In this case, it seems reasonable to use the same probability as for numerical variables but taking into account all the bits of the variable in both executions and take 5% of the result.

   $$P_{4.2c} = 0,05 \times \frac{n_2}{w} \times (1 - exp(-\lambda \times w \times t))^2 \quad (3.17)$$

with λ in events/bit.sec, w the size of a word, t is the maximum duration of a run of the processing element and n_3 the number of bits in the L1 data cache.

3. During each of the two runs, SEUs modify in the same way a register when it holds the same variable with the same value between the time this value is written in the register and the time it is used in the same place in the program by the PE or the same faulty value is injected at that time by faulty data path logic. The time where the SEU could happen is very short, the worst case being probably the frame pointer that stays unchanged for the duration that a function is executed without calling another one, but again the SEU must hit the same bit. $100ns$ seems to be a safe estimate of the time a value may stay in the register and be reused.

$$P_{4.3} = \frac{n_3}{w} \times (1 - exp(-\lambda \times w \times t_3))^2 \tag{3.18}$$

This is a gross over estimation because the formula for boolean variables is used. with λ in events/bit.sec, w the number of bits in a word, $t_3 = 100ns$ and n_3 the number of bits in the register file.

P_4 is the sum of all the $P_{4.*}$ probabilities. Because $t_1 = t_2 = t_3$, let's call it τ and $n_1 = n_2$, let's call it n_c (n_{cache}) and let's rename n_3, n_r ($n_{register}$).

$$P_4 = P_{4.1} + P_{4.2a} + P_{4.2b} + P_{4.2c} + P_{4.3} \tag{3.19}$$

where $n_c = size$ in bits of L1 program cache : 32 or 64 kbytes = 256 or 512 kbits

$n_r = size$ in bits of register file (including hidden register in the data path): approximately 200 x 64 bits = 12,8 kbits

P_0 and P_4 at ground level can be computed from Software Error Rates (SER) data for unhardened devices in [HAR15] presented in Table 3.4 where G. Hubert et al. showed the soft Error rate as a function of technology for CMOS, FDSOI and FinFET. These result were considered at ground level (Toulouse location, 43 136 0 16" North and 1 126 0 38" East) [HAR15] in FIT/Mbit. One FIT (Failure In Time) equals one failure per billion of hours (10^9 hours).

P_0 is so low at ground level that P_4 is not worth computing. P_0 and P_4 are probabilities to have an undesired situation in a $250\mu s$ processing element. However, the radiation hardening community thinks in FIT,

Table 3.4: SER trends for unhardened bulk CMOS, FDSOI and FinFET technologies in FIT/Mbit at ground level

SER	14nm	22nm	28nm	32nm	45nm	65nm
CMOS	10^4	5×10^3	10^3	10^3	10^3	10^3
FDSOI	5×10^2	10^2	10^3	10^2	10^2	10^2
FinFET	8×10^2	7×10^2	5×10^2	7×10^2	8×10^2	8×10^2

thus they are interested not in P_0 and P_4 but in SER_0 and SER_4 where $SER_i = P_i \times 3600 \times 10^9 = P_i \times 3,6 \times 10^{11}$, expressed in FIT.

These results can be converted in (errors/s)/Mbit.

Table 3.5: λ trends for bulk CMOS, FDSOI and FinFET technologies in (errors/s)/Mbit at ground level

λ	14nm	22nm	28nm	32nm	45nm	65nm
CMOS	2.78×10^{-9}	1.39×10^{-9}	2.78×10^{-10}	2.78×10^{-10}	2.78×10^{-10}	2.78×10^{-10}
FDSOI	1.39×10^{-10}	2.78×10^{-11}	2.78×10^{-10}	2.78×10^{-11}	2.78×10^{-11}	2.78×10^{-11}
FinFET	2.22×10^{-10}	1.94×10^{-10}	1.39×10^{-10}	1.94×10^{-10}	2.22×10^{-10}	2.22×10^{-10}

Table 3.6: P_0 for bulk CMOS, FDSOI and FinFET technologies at ground level for 1Mbit

P_0	14nm	22nm	28nm	32nm	45nm	65nm
CMOS	4.82×10^{-25}	1.21×10^{-25}	4.83×10^{-27}	4.83×10^{-27}	4.83×10^{-27}	4.85×10^{-27}
FDSOI	1.21×10^{-27}	4.89×10^{-29}	4.83×10^{-27}	4.89×10^{-29}	4.89×10^{-29}	4.89×10^{-29}
FinFET	3.08×10^{-27}	2.36×10^{-27}	1.21×10^{-27}	2.36×10^{-27}	3.08×10^{-27}	3.08×10^{-27}

Table 3.7: SER_0 for bulk CMOS, FDSOI and FinFET technologies at ground level for 1Mbit

SER_0	14nm	22nm	28nm	32nm	45nm	65nm
CMOS	1.74×10^{-12}	4.34×10^{-13}	1.73×10^{-14}	1.73×10^{-14}	1.73×10^{-14}	1.73×10^{-14}
FDSOI	4.35×10^{-15}	1.76×10^{-16}	1.73×10^{-14}	1.76×10^{-16}	1.76×10^{-16}	1.76×10^{-16}
FinFET	1.11×10^{-14}	8.51×10^{-15}	4.35×10^{-15}	8.51×10^{-15}	1.11×10^{-14}	1.11×10^{-14}

SER_0 , that represents the worst case residual error rate after successfully applying the BHT, is so low (one billion times or more below the most severe requirement) that one may say that, at ground level, with processing elements of max $250\mu sec$, and SER as published in [HAR15], the BHT can recover all failures caused by SEU even if one assumes that all double SEU (one or more in each execution) are fatal. With the more realistic analysis of SER_4 , the error rate would be immeasurably low.

These values can similarly be analyzed in space environment. To do that, one can use Table 3.8 where Q_{crit}, cross section, T_{si} and h_{fin} were got from [HAR15] which gives the (same) critical charge for CMOS, FDSOI and FinFET transistors for various technologies node from 65nm to 14nm. OMERE-TRAD [TRA18] was used to compute the value of the unprotected Soft Error Rate. The following formula is used to compute the Linear Energy Transfer (LET): if the LET is in $MeV.cm^2/mg$, c in μm and the critical charge Q in pC then the formula is:

$$LET = \frac{22.5 \times Q_{crit} \times 1000}{c \times 232.11} \tag{3.20}$$

The following assumptions were made (they are commonly assumed by our industrial partners and based on their experience [Mat10]):

- For FDSOI nano-scale technology $c = T_{si}$
- For FinFET nano-scale technology $c = 2 \times h_{fin}$
- For Bulk CMOS nano-scale technology $c = 2 \times 0,35\mu m$

The Soft Error Rate (SER) for unprotected devices is computed using OMERE in errors/sec.bit, thus it is noted λ. OMERE is a desktop

application provided by the company TRAD (Test Radiation). Three phases are needed to compute SER in OMERE or SPENVIS:

- The environment definition: the parameters of the spatial environment are defined, including the altitude, the solar activity, etc. Four environments have been considered the LEO (Low earth orbit) where the ISS (International Space Station) is located, the MEO (medium earth orbit), the GEO orbit (geostationary orbit) and open space beyond the geostationary altitude.

- The spectrum definition: This step consists in defining the radiation sources by choosing models representing the energies of particles.

- Computing the soft error rate. At this stage the data in table 3.8 parameters are provided to OMERE or SPENVIS.

These tools provide SER in errors per day per bit. Based on λ, we can compute P_0 and P_4 for space environments. The LET values are normalized (LET=linear Energy Transfer in MeV/cm; normalized LET=LET/density in $Mev.cm^2/mg$).

Based on discussion we have had with our industrial partners the direct effect of protons should be considered for the fine technologies like those we consider. OMERE allows to include the direct effect of protons while SPENVIS does not. We choose OMERE.

Events are grouped into two categories. Events caused by heavy ions and events caused by protons. For the GEO environment shown in Table 3.9, heavy ion events are scarce compared to proton events. The total effect is dominated by the protons. This is the same for the three considered technologies (CMOS, FinFET and FDSOI). The results are shown in Table 3.11 and Table 3.13.

For all three technologies the 14nm node has the lowest SER while the 65nm node has the highest SER. We would expect the opposite because the 14nm node has a gate smaller than all other nodes. The researchers community and the space industry do not yet have a good understanding of the behavior of these technologies exposed to charged particles in space environment.

For the three technologies only the probabilities P_{4a} and P_{42c} are significant. While the probabilities P_{41}, P_{42b}, P_{43} are negligible. The risk therefore lies in considering only the numeric variables and the boolean-like variables.

Table 3.8: Parameters to feed into compute λ using the tools SPENVIS or OMERE

	14nm	22nm	28nm	32nm	45nm	65nm
$Q_{crit}(fC)$	8×10^{-2}	1.2×10^{-1}	2×10^{-1}	3×10^{-1}	4×10^{-1}	6×10^{-1}
Cross section (cm^2)	10^{-9}	1.5×10^{-9}	2×10^{-9}	2.5×10^{-9}	4×10^{-9}	5.5×10^{-9}
(FDSOI) T_{si} (μm)	5×10^{-4}	7×10^{-4}	7×10^{-4}	8×10^{-4}	8.5×10^{-4}	9×10^{-4}
(FinFET) h_{fin} (μm)	1.4×10^{-2}	2.20×10^{-2}	2.8×10^{-2}	3.2×10^{-2}	4.5×10^{-2}	6.5×10^{-2}
(FDSOI) LET (Mev.cm^2/mg)	64.6	45.6	36.4	27.7	16.6	15.5
(FinFET) LET (Mev.cm^2/mg)	4.47×10^{-1}	4.31×10^{-1}	4.54×10^{-1}	3.46×10^{-1}	2.64×10^{-1}	2.77×10^{-11}
(Bulk CMOS) LET (Mev.cm^2/mg)	8.31×10^{-2}	5.54×10^{-2}	4.15×10^{-2}	2.77×10^{-2}	1.66×10^{-2}	1.11×10^{-2}

λ to compute the probability P_4* is obtained by the equation:

$$\lambda = \frac{SER}{86400} \tag{3.21}$$

Given SER, λ and the different probabilities can be computed. In the tables below, it appears clearly that, in LEO, P_0 and SER_0 values are so small compared to the unprotected processor at ground that it is not necessary to consider the values of P_4 and SER_4. It can be seen that with BHT system used in the ISS can provide a higher level of reliability in more hostile environments such as the LEO orbit (the worst case is open space on the earth orbit, which is not very different).

In MEO, GEO and open space orbit, the P_0 and SER_0 values are too high to be acceptable. Here computing P_4 is thus useful (Only GEO is showed here. The complete results can be found in Annex A.2).

SER_4 represents, in FIT, the residual error rate for the whole system after successful application of BHT. These may be compared to the SER for unhardened devices at ground level for the same amount of memory (4Mbit), deduced from Table 3.4 for CMOS.

Table 3.10: P_0 and P_4 in GEO orbit space environment (FD-SOI)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	1.31×10^{-7}	1.90×10^{-7}	1.81×10^{-7}	1.76×10^{-7}	2.19×10^{-7}	1.98×10^{-7}
P_0 $N = 4Mbits$						
λ in error/4Mbits/s	5.49×10^{-1}	7.97×10^{-1}	7.59×10^{-1}	7.37×10^{-1}	9.20×10^{-1}	8.32×10^{-1}
P_0	1.88×10^{-8}	3.97×10^{-8}	3.60×10^{-8}	3.39×10^{-8}	5.29×10^{-8}	4.32×10^{-8}
SER_0 (FIT)	6.78×10^{4}	1.43×10^{5}	1.30×10^{5}	1.22×10^{5}	1.91×10^{5}	1.56×10^{5}
P_4 **For** $n_c = 256kbits$						
P_{41}	4.49×10^{-23}	9.46×10^{-23}	8.58×10^{-23}	8.08×10^{-23}	1.26×10^{-22}	1.03×10^{-22}
P_{42a}	2.81×10^{-16}	5.91×10^{-16}	5.36×10^{-16}	5.06×10^{-16}	7.89×10^{-16}	6.44×10^{-16}
$P_{4_{2b}}$	2.87×10^{-21}	6.06×10^{-21}	5.49×10^{-21}	5.18×10^{-21}	8.08×10^{-21}	6.59×10^{-21}

$P_{4_{2c}}$	8.98×10^{-16}	1.89×10^{-15}	1.72×10^{-15}	1.62×10^{-15}	2.52×10^{-15}	2.06×10^{-15}
P_{43}	1.40×10^{-22}	2.96×10^{-22}	2.68×10^{-22}	2.53×10^{-22}	3.94×10^{-22}	3.22×10^{-22}
P_4	1.18×10^{-15}	2.48×10^{-15}	2.25×10^{-15}	2.12×10^{-15}	3.31×10^{-15}	2.70×10^{-15}
SER_4 (FIT)	4.24×10^{-3}	8.94×10^{-3}	8.11×10^{-3}	7.65×10^{-3}	1.19×10^{-2}	9.74×10^{-3}
P_4 **For** $n_c = 512kbits$						
P_{41}	8.99×10^{-23}	1.89×10^{-22}	1.72×10^{-22}	1.62×10^{-22}	2.53×10^{-22}	2.07×10^{-22}
P_{42a}	5.61×10^{-16}	1.18×10^{-15}	1.07×10^{-15}	1.01×10^{-15}	1.58×10^{-15}	1.29×10^{-15}
$P_{4_{2b}}$	5.75×10^{-21}	1.21×10^{-20}	1.10×10^{-20}	1.04×10^{-20}	1.62×10^{-20}	1.32×10^{-20}
$P_{4_{2c}}$	1.80×10^{-15}	3.79×10^{-15}	3.43×10^{-15}	3.24×10^{-15}	5.05×10^{-15}	4.12×10^{-15}
P_{43}	1.4×10^{-22}	2.96×10^{-22}	2.68×10^{-22}	2.53×10^{-22}	3.94×10^{-22}	3.22×10^{-22}
P_4	2.36×10^{-15}	4.97×10^{-15}	4.51×10^{-15}	4.25×10^{-15}	6.63×10^{-15}	5.41×10^{-15}
SER_4 (FIT)	8.49×10^{-3}	1.79×10^{-2}	1.62×10^{-2}	1.53×10^{-2}	2.39×10^{-2}	1.95×10^{-2}

Table 3.12: P_0 and P_4 in GEO orbit space environment (Bulk CMOS)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	1.78×10^{-7}	2.67×10^{-7}	3.57×10^{-7}	4.44×10^{-7}	7.12×10^{-7}	9.8×10^{-7}
P_0 $N = 4Mbits$						
λ in error/4Mbits/s	7.48×10^{-1}	1.12×10^{0}	1.50×10^{0}	1.86×10^{0}	2.99×10^{0}	4.11×10^{0}
P_0	3.5×10^{-8}	7.86×10^{-8}	1.40×10^{-7}	2.17×10^{-7}	5.57×10^{-7}	1.05×10^{-6}
SER_0 (FIT)	1.26×10^{5}	2.83×10^{5}	5.05×10^{5}	7.81×10^{5}	2.01×10^{6}	3.80×10^{6}
P_4 **For** $n_c = 256kbits$						

P_{41}	8.36×10^{-23}	1.88×10^{-22}	3.35×10^{-22}	5.17×10^{-22}	1.33×10^{-21}	2.52×10^{-21}
P_{42a}	5.22×10^{-16}	1.17×10^{-15}	2.09×10^{-15}	3.24×10^{-15}	8.31×10^{-15}	1.57×10^{-14}
$P_{4_{2b}}$	5.34×10^{-21}	1.20×10^{-20}	2.14×10^{-20}	3.31×10^{-20}	8.51×10^{-20}	1.61×10^{-19}
$P_{4_{2c}}$	1.67×10^{-15}	3.75×10^{-15}	6.70×10^{-15}	1.04×10^{-14}	2.66×10^{-14}	5.04×10^{-14}
P_{43}	2.61×10^{-22}	5.86×10^{-22}	1.05×10^{-21}	1.62×10^{-21}	4.15×10^{-21}	7.87×10^{-21}
P_4	2.19×10^{-15}	4.92×10^{-15}	8.79×10^{-15}	1.36×10^{-14}	3.49×10^{-14}	6.61×10^{-14}
SER_4 (FIT)	7.89×10^{-3}	1.77×10^{-2}	3.16×10^{-2}	4.89×10^{-2}	1.26×10^{-1}	2.38×10^{-1}
P_4 **For** $n_c = 512kbits$						
P_{41}	1.67×10^{-22}	3.75×10^{-22}	6.70×10^{-22}	1.03×10^{-21}	2.66×10^{-21}	5.04×10^{-21}
P_{42a}	1.04×10^{-15}	2.34×10^{-15}	4.18×10^{-15}	6.47×10^{-15}	1.66×10^{-14}	3.15×10^{-14}
$P_{4_{2b}}$	1.07×10^{-20}	2.40×10^{-20}	4.28×10^{-20}	6.63×10^{-20}	1.70×10^{-19}	3.22×10^{-19}
$P_{4_{2c}}$	3.34×10^{-15}	7.50×10^{-15}	1.34×10^{-14}	2.07×10^{-14}	5.32×10^{-14}	1.01×10^{-13}
P_{43}	2.61×10^{-22}	5.86×10^{-22}	1.05×10^{-21}	1.62×10^{-21}	4.15×10^{-21}	7.87×10^{-21}
P_4	4.38×10^{-15}	9.84×10^{-15}	1.76×10^{-14}	2.72×10^{-14}	6.98×10^{-14}	1.32×10^{-13}
SER_4 (FIT)	1.58×10^{-2}	3.54×10^{-2}	6.33×10^{-2}	9.78×10^{-2}	2.51×10^{-1}	4.76×10^{-1}

Table 3.14: P_0 and P_4 in GEO orbit space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	1.78×10^{-7}	2.67×10^{-7}	3.57×10^{-7}	4.44×10^{-7}	7.12×10^{-7}	9.8×10^{-7}
P_0 $N = 4Mbits$						

λ in error/4Mbits/s	7.47×10^{-1}	1.12×10^{0}	1.50×10^{0}	1.86×10^{0}	2.99×10^{0}	4.11×10^{0}
P_0	3.49×10^{-8}	7.85×10^{-8}	1.40×10^{-7}	2.17×10^{-7}	5.57×10^{-7}	1.05×10^{-6}
SER_0 (FIT)	1.26×10^{5}	2.83×10^{5}	5.05×10^{5}	7.81×10^{5}	2.01×10^{6}	3.80×10^{6}
P_4 **For** $n_c = 256kbits$						
P_{41}	8.29×10^{-23}	1.87×10^{-22}	3.35×10^{-22}	5.17×10^{-22}	1.33×10^{-21}	2.52×10^{-21}
P_{42a}	5.20×10^{-16}	1.17×10^{-15}	2.09×10^{-15}	3.24×10^{-15}	8.31×10^{-15}	1.57×10^{-14}
P_{42b}	5.32×10^{-21}	1.20×10^{-20}	2.14×10^{-20}	3.31×10^{-20}	8.51×10^{-20}	1.61×10^{-19}
P_{42c}	1.66×10^{-15}	3.75×10^{-15}	6.69×10^{-15}	1.04×10^{-14}	2.66×10^{-14}	5.04×10^{-14}
P_{43}	2.60×10^{-22}	5.85×10^{-22}	1.04×10^{-21}	1.62×10^{-21}	4.15×10^{-21}	7.87×10^{-21}
P_4	2.18×10^{-15}	4.92×10^{-15}	8.78×10^{-15}	1.36×10^{-14}	3.49×10^{-14}	6.61×10^{-14}
SER_4 (FIT)	7.86×10^{-3}	1.77×10^{-2}	3.16×10^{-2}	4.89×10^{-2}	1.26×10^{-1}	2.38×10^{-1}
P_4 **For** $n_c = 512kbits$						
P_{41}	1.66×10^{-22}	3.75×10^{-22}	6.69×10^{-22}	1.03×10^{-21}	2.66×10^{-21}	5.04×10^{-21}
P_{42a}	1.04×10^{-15}	2.34×10^{-15}	4.18×10^{-15}	6.47×10^{-15}	1.66×10^{-14}	3.15×10^{-14}
P_{42b}	1.06×10^{-20}	2.40×10^{-20}	4.28×10^{-20}	6.63×10^{-20}	1.70×10^{-19}	3.22×10^{-19}
P_{42c}	3.33×10^{-15}	7.49×10^{-15}	1.34×10^{-14}	2.07×10^{-14}	5.32×10^{-14}	1.01×10^{-13}
P_{43}	2.60×10^{-22}	5.85×10^{-22}	1.04×10^{-21}	1.62×10^{-21}	4.15×10^{-21}	7.87×10^{-21}
P_4	4.37×10^{-15}	9.83×10^{-15}	1.76×10^{-14}	2.72×10^{-14}	6.98×10^{-14}	1.32×10^{-13}
SER_4 (FIT)	1.57×10^{-2}	3.54×10^{-2}	6.32×10^{-2}	9.78×10^{-2}	2.51×10^{-1}	4.76×10^{-1}

On average the expected protected values in space are about thousand times better than unprotected values at ground level for meo, geo

Table 3.9: SER rates (/day/bit) from OMERE in GEO orbit space environment (FDSOI)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	2.69×10^{-12}	4.41×10^{-12}	1.37×10^{-13}	4.13×10^{-15}	4.03×10^{-15}	1.19×10^{-15}
Trapped protons (Protons Rate)	0	0	0	0	0	0
Solar protons (Protons Rate)	3.41×10^{-3}	4.94×10^{-3}	4.50×10^{-3}	4.17×10^{-3}	4.90×10^{-3}	3.76×10^{-3}
Cosmic rays (Protons Rate)	3.57×10^{-4}	5.36×10^{-4}	7.14×10^{-4}	8.91×10^{-4}	1.42×10^{-3}	1.95×10^{-3}
Heavy Ions rate	2.69×10^{-12}	4.41×10^{-12}	1.37×10^{-13}	4.13×10^{-15}	4.03×10^{-15}	1.19×10^{-15}
Protons rate	3.77×10^{-3}	5.47×10^{-3}	5.21×10^{-3}	5.06×10^{-3}	6.32×10^{-3}	5.71×10^{-3}
Total rate	3.77×10^{-3}	5.47×10^{-3}	5.21×10^{-3}	5.06×10^{-3}	6.32×10^{-3}	5.71×10^{-3}

Table 3.11: SEE rates (/day/bit) from OMERE in GEO orbit space environment (Bulk CMOS)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	2.21×10^{-7}	3.45×10^{-7}	4.52×10^{-7}	5.36×10^{-7}	8.64×10^{-7}	1.11×10^{-6}
Trapped protons (Protons Rate)	0	0	0	0	0	0
Solar protons (Protons Rate)	4.78×10^{-3}	7.17×10^{-3}	9.56×10^{-3}	1.19×10^{-2}	1.91×10^{-2}	2.63×10^{-2}
Cosmic rays (Protons Rate)	3.58×10^{-4}	5.36×10^{-4}	7.15×10^{-4}	8.94×10^{-4}	1.43×10^{-3}	1.97×10^{-3}
Heavy Ions rate	2.21×10^{-7}	3.45×10^{-7}	4.52×10^{-7}	5.36×10^{-7}	8.64×10^{-7}	1.11×10^{-6}
Protons rate	5.14×10^{-3}	7.70×10^{-3}	1.03×10^{-2}	1.28×10^{-2}	2.05×10^{-2}	2.82×10^{-2}
Total rate	5.14×10^{-3}	7.70×10^{-3}	1.03×10^{-2}	1.28×10^{-2}	2.05×10^{-2}	2.82×10^{-2}

Table 3.13: SER rates (/day/bit) from OMERE in GEO orbit space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	4.32×10^{-8}	8.47×10^{-8}	9.54×10^{-8}	9.45×10^{-8}	2.05×10^{-7}	1.11×10^{-6}
Trapped protons (Protons Rate)	0	0	0	0	0	0
Solar protons (Protons Rate)	4.77×10^{-3}	7.16×10^{-3}	9.54×10^{-3}	1.19×10^{-2}	1.91×10^{-2}	2.63×10^{-2}
Cosmic rays (Protons Rate)	3.58×10^{-4}	5.36×10^{-4}	7.15×10^{-4}	8.94×10^{-4}	1.43×10^{-3}	1.97×10^{-3}
Heavy Ions rate	4.32×10^{-8}	8.47×10^{-8}	9.54×10^{-8}	9.45×10^{-8}	2.05×10^{-7}	1.11×10^{-6}
Protons rate	5.13×10^{-3}	7.70×10^{-3}	1.03×10^{-2}	1.28×10^{-2}	2.05×10^{-2}	2.82×10^{-2}
Total rate	5.13×10^{-3}	7.70×10^{-3}	1.03×10^{-2}	1.28×10^{-2}	2.05×10^{-2}	2.82×10^{-2}

Table 3.15: Unhardened SER at ground level for CMOS for the same amount of bits at risk

	14nm	22nm	28nm	32nm	45nm	65nm
CMOS SER (FIT) at ground level 4Mbits	4×10^4	2×10^4	4×10^3	4×10^3	4×10^3	4×10^3

and open space orbit. It is less than the FIT order.

Several comments should be made about these results. One is that it has not been taken into account here that SEU can not only cause false results to the processing element but also send it into an infinite loop or crash it. Infinite loops are stopped after the maximum allowed execution time of the current execution of the PE and detected like computational errors. Crashes cause exceptions which will be detected even in case of double SEU. The above results, assuming that the SEU only cause computational errors, are thus worst case results.

Another comment is about the sensitivity of the results to the choice of the parameters. It has been assumed in this evaluation that all processing elements had the same duration of $250\mu sec$. However, it will be seen in the next chapter that most processing elements are a lot shorter because of system calls that end processing elements prematurely, reducing that way the risk for the a SEU to occur during an execution of the processing element. The values found here are thus "worst case" values from that point of view too. The orders of magnitude of the result do thus not change abruptly for reasonable variations of the hypotheses. The conclusion is thus that by applying BHT with the proposed parameters the residual error rate in space will be 1000 times lower than on the ground and that it varies linearly with each of the parameters: no bad surprise is thus to be expected.

However because the probabilities are so small, unimagined events could happen like in Fukushima: it should not hence happen in a century as long as you are not in the "wrong" century.

3.7 Conclusion

Processors implementing the RAS features are good support for implementing the BHT. The error handling mechanism is exploitable to restore the system to a stable state regardless of the SEUs effects on any of the processors registers. Bulk CMOS and FinFET nano-scale technologies present the same order of residual error, $1,58 \times 10^{-2}$ for technology 14nm node (the lowest) and 4.76×10^{-1} for technology 65nm. Whereas at ground level the SER in FIT are the order of 10^3 for unhardened devices and the evaluation of the residual error presented here is pessimistic.

CHAPTER 4

Fault tolerance in operating systems

4.1 Introduction

This chapter outlines first the role of an operating system in terms of functionalities and its positioning within computer systems. Different types of operating systems are presented. This will help to understand the fact that an operating system is capable of handling the faults caused by SEUs. The choice of the operating system used will be justified by a short comparison of the classes of operating systems. The internal functions of the related operating system will also be briefly presented.

4.2 Operating system

A computer system is composed of two parts: hardware and software. The hardware consists of electronic circuits. The operating system is a program or software that mainly plays two roles [SGG18].

Firstly it makes the details of using the hardware invisible to application programs by offering an interface that they use. In other words, instead of programmers spending most of their time configuring the hardware before it is used, the operating system does the work for them [TW87]. It provides a generic interface that can be used to program or use a wide range of hardware. The application programmer will therefore have more time to focus on the functional aspects of its application instead of wasting time configuring devices or equipment [TW87, p. 3]. The operating system as software is therefore the intermediary between the application program and the hardware as shown in Figure 4.1.

Secondly, the operating system plays a role of facilitator and resources handler of the machine. A facilitator role because it ensures a good use of resources by several applications by making them available and accessible appropriately [SGG18]. For example, when an application wants to use memory, the operating system will first ensure the availability of this resource and then make it available to that application.

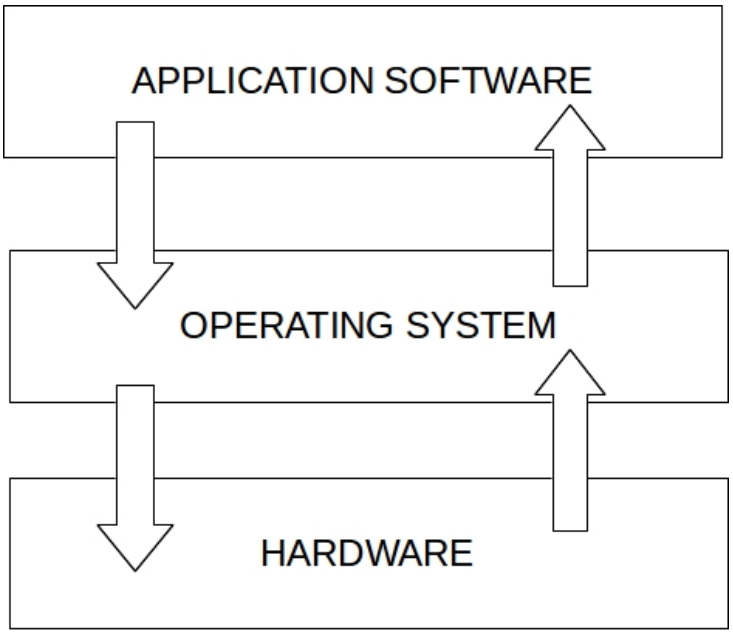

Figure 4.1: OS architecture

It plays the role of resource handler because the amount of available resources within the system is limited (for example, only one network card might be available, the amount of memory might be limited etc.). However there might be a large number of applications or processes within the system. Under these conditions, the operating system ensures that the requests of the processes are satisfied according to the availability of resources and the policy implemented [TW87].

So an operating system can be a simple program but also a complex program. The following section will present fundamental features of an operating system.

4.3 Basic features of an operating system

A computer architecture provides a set of instructions for efficient use of the processor and peripheral modules. Application programs are only allowed to use instructions that cannot interfere with others application programs or the general operation of the computer. The others instructions are called "privileged" instructions. The operating system is the only software capable of executing the privileged instructions [Gui11] of

the processor.

Thus, one of the basic features of an operating system is to allow application processes to have access to the services requiring the use of these privileged instructions by providing them with a suitable interface. These privileged instructions are used for instance for the configuration and use of input and output devices. Others are needed for the configuration and use of the processor or processors or internal modules such as the machine check architecture (MCA) [Gui11] or virtualization[SGG18].

Another feature of operating systems is to allow applications to be identifiable entities within the operating system. That's what leads to the concept of process. A process is an entity that basically represents an application [SGG18]. It has a unique identity, it has resources, it has a life cycle, it interacts with other processes of the system. The life cycle means that at some point the process appears in the system, interacts with other processes, and then ceases to exist in the system. Thus, the operating system makes all this possible by allowing the implementation of the notion of process, the creation of processes, the communication between processes, the management of the resources of a process, the use of the resources of the systems by the processes especially the use of the processor [TW87, p. 56].

Another feature of the operating system is to allow processes to access to the input and output peripherals and different hardware. It provides an interface that allows processes to use and configure hardware. It ensures communication between the I/O peripherals, and processes through the interrupt mechanism [SGG18, p. 560].

4.4 Different classes of operating systems

Operating systems can be classified based on target hardware architectures. They can also be classified based on their software architectures. They can also be classified based on the need they fulfill.

4.4.1 Hardware Architecture

There are operating systems with specialized features for embedded systems. These operating systems are intended for architectures with limited resources such as microcontrollers and connected objects [JKH+16]. They are dedicated to specialized systems such as medical equipments, satellites, printers, electrical appliances, home appliances, medical equipments, vehicles. They have features dedicated to the needs of the sys-

tems that the applications that run on top of them are monitoring or controlling [JKH+16]. We can mention Contiki [DGV04], QNX [Hil92], WinCE, Vxworks.

There are operating systems for personal computers. These are most popular operating systems for the public. They allow the use of laptops and desktops. The most popular are Windows, Linux and Mac-OS [WABM04] [RDH+01]. They are present on purchased computers or can be installed on computers from a storage device (USB, CD / DVD).

There are operating systems for mobile phones and tablets. In recent years there has been an explosion in the more advanced use of mobile terminals. The phone is not used for phone calls only. It can be used to check emails, to do word processing, to surf the Internet [LY09], i.e many features that were those of computers in the past. Mobile devices operating systems have evolved to effectively meet these needs [LLL+11] [HA09]. The most popular ones are Android and iOS. Android is a mobile operating system developed and maintained by the computer giant Google [Dev11]. It is based on the kernel of the Linux operating system. It should be remembered that the kernel is the set of programs that allow the operating system to provide the basic functionality that we have described above. IOS is the operating system developed and maintained by the computer giant Apple [NKH12] [SGG18].

There are server operating systems [KEGW96]. These are operating systems that are adapted to the needs of the critical services provided by the servers. These operating systems allow fault tolerance. They guarantee the availability of services. They ensure a more efficient use of resources by allowing a partitioning of services. We can mention Windows Server, Linux based server Operating systems such as Debian and redHat, and Solaris.

4.4.2 Software architecture

Two software architectures stand out for operating systems, monolithic operating systems and micro-kernel operating systems. Distributed operating systems can also be considered. Before defining what is a monolithic operating system we want to remember that the operating system is also considered as a process within a computer system in use. It is also identified as process, it has an address space, it has resources etc. We also want to remind that the features offered by the operating system to applications can be divided into two major groups, basic features and advanced features. Among these basic features we have

the management of input and output devices, management of interrupt and exceptions, communication between processes. All the advanced features are the high-level services provided by the operating system. Among other things, we have file management, drivers for implementing devices protocols, advanced process management and so on.

4.4.2.1 Monolithic

A monolithic operating system is an operating system that will bring together all the basic features and advanced features in the same process [DKD+15] [Dau16]. The operating system will be a large process that runs within one address space. Basic features are active at all times. Advanced features are generally enabled through libraries loaded dynamically on demand. When they are no longer useful they can be removed from the system's memory.

The biggest benefit to having such an architecture is performance. The operating system is able to respond very quickly to the requests of application processes without the need to switch more than once (in some cases, not even once) address space [JKH+16]. Similarly the amount of messages for processing a request is reduced.

On the other hand, the biggest disadvantage is maintenance and fault isolation. To maintain a monolithic operating system, you have to work and test a very large code, because the basic features and advanced features are strongly coupled. The portability of the modules is more expensive. Moreover, as soon as a module fails in an monolithic operating system it is the entire operating system that is faulty. The most popular monolithic operating systems are Linux and Windows [Rat15].

4.4.2.2 Micro-kernel

On the other hand, a micro-kernel operating system will put in the same process only the most basic functionalities (which we will call micro-kernel or kernel) and sub-contract all advanced features to separate processes (we will call servers). Thus the kernel will have its own address space, different from those in which all other servers run [JKH+16]. It can have the ability to restart a server when it fails. It provides communication and hardware access services to the servers. The micro-kernel forwards requests from application processes to the appropriate server that perform its high level function based on low level services the server asks to the micro-kernel. There are generic servers that provide standard

services. There are also specialized servers that provide specialized services. For example, for file management, there are servers that provide generic file management services. This is the case of the vfs (virtual file system) server of the Minix 3 operating system. And also the servers that provide specialized services for example mfs (minix file system) for the management of file systems of the type minix in the case of the Minix3 operating system.

The biggest advantage of micro-kernel operating systems is the modularity and fault isolation. Modularity provides more flexible and easy maintenance and fault management. A server can be replaced, upgraded or removed, without making changes in the micro-kernel and other servers. Similarly a failure in one server does not propagate systematically in the kernel and other servers. The failure can be controlled either by restarting a new instance of the failed server or by shutting down the server if it is not critical to the entire system. Jorrit Herder's work [HBG+09] has shown how within the Minix3 operating system it is possible to restart as needed faulty services such as hardware drivers.

The biggest disadvantage of micro-kernel operating systems is the loss of performance [Dau16]. It takes a lot of message exchanges for processing a query, because advanced services are scattered in different independent processes. Andrew Tanenbaum in his book has shown that for the processing of a simple request to read a block of data in a file identified by its descriptor requires 8 messages and the participation of 2 servers and the device driver in addition to the micro-kernel [TW87]. Thus performance remains the biggest drawback of micro-kernel operating systems.

However this loss of performance is compensated by their greater reliability compared to monolithic operating systems. The most popular micro-kernel operating systems are Mach UNIX [GJR+92], Minix 3 [THB06], QNX [Hil92] and L4 [EH13].

4.4.2.3 Distributed operating systems

Distributed operating systems are operating systems that run on more than one machine. They combine the resources of two or more machines to provide more efficient services to application processes. They use agents that are deployed on each individual machine to manage local resources and processes. They use network communication protocols for communication between agents. All of this is done transparently because they are designed to make application processes feel like they

are running on a single machine. They are widely used within clusters. The most popular distributed operating systems are Mosix [HAM+18, MJK+18, BL98, BGW93], Amoeba [MJK+18, TVRVS+90, KT91].

4.4.3 The needs in response time

Bounded response times may be required in some applications. We are talking about real-time applications. Events must be taken into account within a specified time. After this time the value of the response to the request may be null or negative. A null value response means that the answer has no effect. While a negative value response means that the delay of the response caused damage in the system or in its environment [Kop11, But11].

There are hard real-time applications, and soft real-time applications. An application is real time hard when time constraints are strict. While in the case of a soft real-time application temporal constraints are more flexible. Thus an operating system on top of which this kind of application runs must have functionalities enabling it to satisfy the requirements of the requests coming from such applications in order to allow them to strictly respect their time constraints [JLT85]. Therefore, there are real-time operating systems that are designed to optimize time management to take events into account to ensure that time constraints are met. The most popular real time operating systems are FreeRTOS [HH16], RealPi [Del18], MicroC OS II [SZF16].

4.5 Operating systems as part of fault tolerance

Fault tolerance is not a new concept in the field of operating systems. Operating systems have often been used to provide continuity of service despite hardware failures in the case of input and output devices.

Remember that a fault that occurs in a running system occurs at a given time and could generate errors. These errors could be silent (no error reported but uncorrected results). These faults could also trigger exceptions (abnormal event) or a reset of the whole machine. Being the central element of the system from the software standpoint, the operating system is the first to be informed of these abnormal events if they are detected. It has the freedom to make decisions that can correct the instability caused by the event to reduce the perimeter of spread of the fault. Thus an operating system can be modified to ensure the redundancy of information, taking into account any abnormal event reported

by a known or unknown exception. Also elements present within the processors can also be used to extend the functionality of the operating system to make it more fault tolerant.

4.6 Choice of an operating system

We choose Minix 3, a micro-kernel operating system. Minix 3 is a micro-kernel operating system developed by Andrew Tanenbaum [TW87]. It has a clear structure. It is well written and adequately documented [Her10]. As a micro-kernel operating system, Minix 3 is also more modular than most traditional operating systems. It is possible to harden one part of the operating system without affecting the other parts. Also, an error inside one module of the operating system is not directly propagated to the others parts of the system thanks to the fault isolation mechanisms implemented inside Minix 3 [HBG+09].

It has thus many features desirable for a proof of concept research on operating systems. In Minix 3, service is provided to application processes via system calls. As a micro-kernel operating system, Minix 3 outsources some traditional kernel services to server processes running in privilege level 3 (on Intel386-like processors) [TW87]. Outsourcing is implemented through messages exchanged between the micro-kernel and the server processes.

4.6.1 Minix3 history and goals

Minix, and its successors, Minix 2 and Minix 3, was originally designed for teaching purposes when AT&T, owner of the UNIX operating system, decided to close its sources that were often used as an example in operating systems design courses. Andrew Tanenbaum and his team of Vrije Universiteit Amsterdam designed then Minix, a simple, small and clean open source operating system, with a Unix-like user interface. The original Minix was targeted at the IBM PC [TW87] . It was based on the micro-kernel paradigm, where each class of services is isolated in a separate server that could easily be understood and modified by students. Now Minix 3 has been ported to several state-of-the-art architectures (IA32, ARM) [Nee16] and targets not only teaching purposes but real-life applications. Because it cannot compete with Linux for desktop or server use, mainly because this requires porting a huge number of applications for a platform to succeed on the desktop, Minix 3 targets embedded systems [Tan16]. The latest version of Minix 3.4.0 supports

live update of user applications and servers. With that capability the system does not need to be rebooted after a security update or when a patch is applied to the system [GKT13b].

Minix 3 suffers from a few shortcomings: the choice of peripheral drivers is limited mainly to the hardware used by the development team and, in particular, USB support is totally missing. This is not important for teaching purposes where Minix3 is generally run on a simulator, but limits its usability in real-life applications.

Besides, having been originally designed as a desktop-oriented operating system (at a time when there were much fewer applications on desktops), many of its characteristics are still tuned to events happening at desktop-like speeds: the system clock is at 60 Hz, the scheduling time quantum is one to ten clock ticks. Fortunately the Minix 3 interrupt handling policy is to keep interrupt handling routines as short as possible, so that they do not interfere too much with scheduling decisions (interrupt routines can be seen as very high priority tasks bypassing the scheduler). As a consequence, Minix 3 can mainly be used for embedded systems needing response times only slightly better than those of a human operator. There are many such applications, and Minix 3 being much simpler than, for instance, real-time Linux, it can be successful in this field.

However, SEU hardening is a job that requires much faster response times, so it was necessary to push Minix 3 in this unexplored territory to see how it would behave.

4.6.2 Minix3 Structure

4.6.2.1 The micro-kernel pseudo processes

Minix3 is a micro-kernel operating system. The microkernel consists of six pseudo processes running in the same address space. They are called kernel tasks.

ASYNC is used for notification of asynchronous events. Minix3 allows synchronous and asynchronous communication between processes. Synchronous communication is when the sender process involved remain blocked until the message has been transmitted and the answer received. While during an asynchronous communication the sender process is not stopped after sending the message and can be notified by the ASYNC pseudo process of the kernel that the message was taken into account. ASYNC is responsible for informing the sender process when the re-

quested answer is ready. Meanwhile the sender process can continue to execute, it is not blocked. As soon as an answer is available, the ASYNC pseudo process informs the sender.

IDLE, as its name indicates, occupies CPU cycles when no task is ready to be executed. That means, the queues of the scheduler are empty. This is just a function that runs in a loop. It halts the processor and counts the number of clock cycles spent doing nothing.

CLOCK, as its name suggests it takes care of all the features related to time management, alarms, clock ticks, watchdogs. We will say that it is the watch of the microkernel.

SYSTEM is the interface between the micro-kernel and the servers. The requests from the servers are received by this pseudo process. It analyzes the requests and judges their authenticity by checking the source, the type of requests and the parameters provided in relation to the existing policy in the system. If the elements are consistent with the policy the corresponding handler is called to respond to the request. It should be noted here that there are two types of queries. The kernel calls that are queries whose destination is the microkernel itself and the system calls whose destination is another server process. The software interrupt mechanisms that the kernel uses for the implementation of this mechanism are SYSENTER and SYSCALL.

KERNEL is the pseudo process that handles communication between processes and scheduling. Does a process need to send a message to another process? It is this pseudo process that takes care of copying the message from one address space to another. It also deals with the change of state of a process (running, blocked, ready).

HARDWARE is the pseudo process that supports all the hardware interrupt mechanism. It handles interrupt vectors and corresponding handlers to respond to hardware interrupts from input and output devices as efficiently as possible.

4.6.2.2 The entry and exit points of the microkernel

Like most operating systems, the Minix3 microkernel has multiple entry points. The first entry point is only used once, when the system is booting, to jump from from the boot loader to the micro-kernel. It is the *kmain* function which is in the file kernel/main.c of the source tree. This function initializes the data structures of the micro-kernel. Tasks performed by this function include configuring the processor to operate in protected mode, initializing the process table, initializing scheduler

queues, initializing interrupt and exception vectors, initializing the privileges tables. Then, one of the most privileged servers previously loaded into memory by the boot loader is selected in the scheduler queue to run. That is the memory manager (VM), which initializes the memory of the boot processes. This transition from kernel to VM goes through the *switch_to_user* function, going from kernel mode to user mode.

The *switch_to_user* function is the exit point of the micro-kernel. From this function the processor moves from the execution of a privileged micro-kernel's pseudo process to the execution of an unprivileged user process or the IDLE process. In this function, the ready process in the head of highest priority non empty queue will be executed if the running process is no longer ready.

After we get out of the micro-kernel via the *switch_to_user* function we will come back when one of the following events occurs: clock interrupt, hardware interrupt, software interrupt including system call or kernel call. Interrupt vectors have been previously configured by the *kmain* function.

All interrupt handlers are defined in the mpx.S file. For clock, input and output devices vectors, the entry points are defined in the assembler macros *hwint_master* and *hwint_slave*. The value of the IRQ allows to determine the source of the interruption, for example when the source is the clock, the value is 0.

For software interrupt vectors, the entry points are different depending on whether they are SYSENTER or SYSCALL. For SYSENTER the entry point is *ipc_entry_sysenter*. While for SYSCALL there are several entry points, *ipc_entry_softint_orig*, *ipc_entry_softint_um, kernel_call_entry_orig, kernel_call_entry_um.*

Ipc_entry_softint_orig and ipc_entry_softint_um are used for interprocess communication This initiates the sending of a message from a sender process (the one that performed the SYSCALL) to a receiver process, (the SYSCALL destination). The message contains the source, the query type, and the parameters to run the query. The response of the request is also sent by the same mechanism, through a SYSCALL initiated by the recipient process, that becomes the sender, to the old sender process that becomes the recipient. The micro-kernel acts as a post officer through the *do_ipc* function.

Kernel_call_entry_orig and kernel_call_entry_um are used to ask for services performed by the micro-kernel itself. In this case the recipient process is the micro-kernel itself. It does the appropriate checking and executes the handler corresponding to the type of request contained

in the message.

In all instances, the answer is copied in the original sender's address space (just above the stack, which grows downwards). The latter becomes ready and continues its execution.

Exception entry points are the *EXCEPTION_ERR_CODE* and *EXCEPTION_NO_ERR_CODE* macros. Values are set for the most common error codes. For example for a page fault the error code is 14. Some handlers are defined for the most well-known exceptions. These handlers are executed when the corresponding event occur. When an unsupported exception occurs, either the process that caused it receives an appropriated signal, or the kernel panics, when the exception occurred during its execution.

4.6.2.3 The main servers

The most important servers are:

- the memory manager (VM) which is in charge of managing the memory space of all the processes;
- the process manager (PM) which is in charge of managing process filiation services such as the fork system call, the exec system call etc;
- the virtual file server (VFS) which is in charge of managing file system servers;
- peripheral driver processes and file system servers; they are submitted to fault isolation constraints so that their failure cannot stop nor harm the system;
- the reincarnation server (RS) which is a kind of watchdog checking periodically if reincarnable processes such as peripheral driver processes are still operational and, if not, replaces them with a fresh instance;[HBG+09]
- the scheduler server (SCHED) that prepares the queues used when the picker in the micro-kernel must decide which process to run next.

One server that does not exist in Minix 3 is a swapper, in charge of keeping enough memory free for new demands, by swapping out pages that are unused for a long time. This is because Minix3 does not support

swapping at all. This is not a limitation: swapping was introduced in the time when memory was a rare and expensive resource, which is generally not the case anymore. In this work it is thus assumed that hardened processes are locked in memory, which is anyway the most obvious and simplest way not to have to add code to harden situations that will, anyway never occur in properly dimensioned systems.

Minix 3 scheduling is based on sixteen privileged queues managed by the SCHED server. The final decision is taken by the process picker in the micro-kernel: a simple routine that goes through the heads of each of these 16 queues, picks the first runnable process and chooses the process to run between this user process and the server processes, then resumes the execution of the process it selected. The picker is called each time an external event (hardware interrupt) or an internal event (exception, trap or system call) is managed by the micro-kernel if the current running process is preempted or not runnable [Her10].

4.6.2.4 The memory manager (VM)

The data structures used by the VM to represent memory are

- Phys_block that represents physical frame in memory.
- Since several pages can map to the same frame in memory (copy-on-write when creating a new process or a memory mapped file), the phys_region data structure has been created. It allows to put in a linked list all the pages that map to the same frame. Thus the memory manager has a clear view of the pages sharing the same frames of virtual memory.
- The pages are organized in virtual regions materialized in the vir_region data structure. A region is a continuous range in the virtual address space of the process. They have a starting address and a length. The data structure include a pointers to a list of *phys_region* . Similarly a *phys_region* belongs to one and only one virtual region. A virtual region is the property of one and only one process. Different regions can start at the same virtual address. A virtual region can be either a directly mapped region, or a region shared between multiple processes or a cached disk block region or region mapped to a portion of a file.

When a page fault occurs, the first check made by the memory manager is to see if the virtual address where the page fault occurred cor-

responds to a region belonging to the faulty process. If this is not the case the process is simply stopped. If this is the case the memory manager checks if the page where the process made the page fault exists in the list of phys_region of the virtual region, if it is not the case a data structure phys_region is created and a frame is allocated. Then access is verified. If the process attempts to write and the page is not writable, the appropriate action is taken. That means a signal is sent to stop the process. If the page is accessible, access is granted and the process can continue to run.

4.6.3 Running benchmarks on Minix3

The implementation of the hardening requires having the process to be very frequently interrupted to respect the principle of the scarcity of SEUs. It will be interrupted about 4000 times per second of execution or once every $250\mu s$. What might be the impact on the overall performance of the Minix 3 operating system? To find out, a benchmark was run with different clock tick values and different quantum values allocated to each process.

Benchmarks [AL16] have been run on standard Minix 3, with different clock periods (called tick) and with different time-sharing quanta for scheduling. The results are presented in the following diagrams. The first one shows all the results; the 3 other ones are zooms on the small variations of the index for each value of the tick. The quantum being a multiple of the tick, short quanta are only possible for short ticks. So the horizontal scale of the three last diagrams is not the same. The benchmark program was Dhrystone. Dhrystone was developed by Reinhold Weicker in 1984[Wei84].

The Dhrystone test contains simple integer arithmetic, string operations, logic decisions and memory accesses intended to reflect the CPU activities in most general purpose computing applications. A moderate performance penalty has been noticed with short ticks.

Small variations have also been noticed for different scheduling quantum values. The tick penalty is around 20% for the shortest tick interval ($39\mu s$) and around 15% of penalty for a tick interval of $78\mu s$. The penalty is caused by the time to handle the clock interrupts. The observation that the changes linked to the quantum are so small could be explained by the fact that the end of a process quantum is only taken into account when no external event occurs before the end of the quantum (system calls,exceptions, hardware interrupts or exception or clock

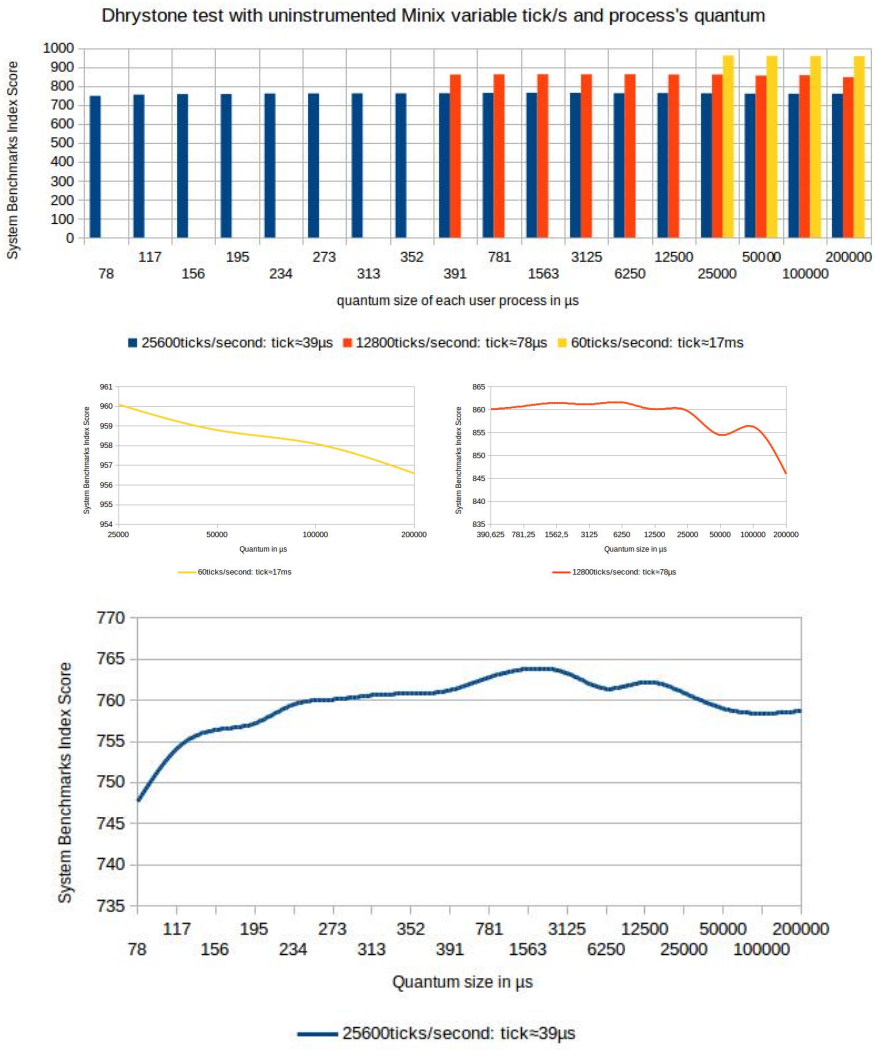

Figure 4.2: Evolution of the Index Score by the tick/second

interrupt). Thus end of quantum effects have a limited impact.

These results show that stopping more often a process during it execution will not impact drastically the overall performance. The penalty of 15% can be afforded. When the process will be divided on processing elements, a penalty of 2 or 3 times is expected. Because the double execution will induce and a performance overhead of 100% and it is expected that all hardening computation will also induced a performance

overhead of 100%.

4.7 Conclusion

This chapter discussed operating system issues relevant for fault tolerance. We were able to make a choice: Minix 3. We have detailed key features of Minix 3. We have seen that an operating system should be able to protect application processes that run above it because all the events that occur and are detected are passed on to the operating system so that it can make the appropriate decisions. In the next chapter we will state the main principles that we will follow in the rest of the work to allow the operating system to protect application processes against transient errors. Not only when the errors are signalled through exceptions but in all cases.

Part II

Methodology

CHAPTER 5

Principles of hardening processes in the operating system, using BHT

5.1 Introduction

Allowing an operating system to harden its application processes requires a rigorous methodology. In this chapter the principles on which the hardening work is based are presented. Problems related to the implementation of the methodology will be clearly stated. Proposed solutions will also be discussed followed by an analysis of the strengths and weaknesses of each approach. By reading this chapter, the reader will have a clear idea of the questions that must be asked in the context of hardening application programs by the operating system. The first section presents the general principles including the hypotheses, the exception mechanism to correct faults detected by hardware and the DWC used to identify and correct fault not detected by the hardware. The second section presents the approach to delimit the frontier of a PE. The third section introduces the mechanism of DWC. The fourth section presents the principles of protected memory. The fifth section presents the principles of external event management. The sixth section discusses the consistency of protected memory with caching and shared memory operations.

5.2 Principles of BHT

In a few words, the application of the BHT principles to hardening user processes from within the operating system consists of exploiting the exceptions initiated by all available hardware fault detection mechanisms and by using a method belonging to the DWC (Double execution With Comparison) class of fault tolerance techniques to detect silent transient SEU faults, those causing false results but no exception.

5.2.1 Definitions

The CPU is modeled by R, H, US. R represents the set of visible registers. These registers include control registers, segment registers, memory management registers, general purpose register. H represents hidden registers. These registers contain all the internal registers of the control path and the data path. US represents the contents of the physical memory of the process.

No state of H is taken into account later. Because the registers in H are not accessible. The BHT method detects and corrects errors that occur within H registers.

A processing element PE, is an instruction flow, idempotent and atomic triggered from an initial state (R_i, US_i) and producing a final state (R_f, US_f). R_i and R_f represent the values in the processor registers. US_i and US_f represent the entire contents of the physical memory of the process. Atomicity means that the intermediate states of the PE are not visible outside of the PE. Only the final state is taken into account. Idempotency means that whatever the number of times PE is executed from the initial state, the final state always remains the same. BHT requires to satisfy 3 requirements:

1. **Duration:** A processing element must be short enough to be executed twice in a time short enough to have to take care of, at most, one SEU. In [LML11] , processing elements were defined when the program was designed, which is not possible here because application programs could have been designed anytime anywhere in the world.

2. **Atomicity :** A processing element must be atomic, which, in the present context, means that there may not be any interference with the outside world during an execution of the PE. In particular, a visible result is only produced at the end of the PE. This result must be complete (and correct) or nothing. This is necessary both to allow an exception to abort the execution of a PE and to obtain the same results after two faultless executions of the PE.

3. **Indempotency :** A processing element must be idempotent, which means that if it is executed several times from the same initial state, its result must be the same as if it had been executed just once. Without this property, DWC would be meaningless.

5.2.2 Exception mechanism

The exception mechanism is used by the hardware to signal abnormal situations to the software. Exceptions may be due to normal running program's behavior, in which case exception handlers can easily handle them. In addition, exceptions may be due to an external independent event (a SEU for example). In this case the exception handler of the OS usually cannot handle the event properly. It should be adapted accordingly. In the context of fault tolerance, this native mechanism, present in all operating systems, can detect faults. However, it could be difficult to distinguish an exception caused by a SEU and an exception due to the operation of the process itself. By coupling the mechanism of exception and the DWC this becomes possible. The DWC can make the difference between a normal exception and an abnormal exception. The technique of the DWC will be detailed below.

5.2.3 Double execution With Comparison (DWC)

Hardening based on DWC fault detection is made possible by keeping the code and state of the process to harden out of harm caused as well by direct hits by SEU as by processes misbehaving after being hit by a SEU. Therefore the code and 3 copies of the data (actually one complete copy and two partial copies including only the frames of the process, that are modified during the execution of the current processing element) are kept in protected memory that cannot be modified, neither by a SEU nor by processes misbehaving after being hit by a SEU. Implementing such protected memory is one of the issues discussed below.

Let's call $Copy_0$ the process data (memory content and CPU registers value) if the BHT is not applied to it. $Copy_1$ represents the data (memory content and CPU registers value) used during the first run. $Copy_2$ represents the data (memory content and CPU registers value) used during the second run. The memory content part of $Copy_1$ and $Copy_2$ are created when the system sets up the page table of the process. They are updated each time the process memory changes (increase or decrease).

$$Copy_i = US_i, R_i \tag{5.1}$$

US_i is the memory content of $Copy_i$ and R_i is the CPU visible register values.

At the beginning of the processing element, the code and data pages are marked as read-only. $Copy_0$ resides in protected memory. $Copy_0$ is

never changed during the execution of the processing element. All the data pages of the process are mapped to US_1 are marked as read-only (at the first PE of this process).

The processing element is then executed a first time. A subset of US_1 is modified during this execution. R_1 is produced during this execution. The frames and registers of $Copy_1$ become protected memory at the end of the first execution. The modified subset of US_1 is called the working set.

The same applies to the second execution. A subset of US_2 is modified during this execution. R_2 is produced during this execution. At the end of the two executions, $Copy_1$ is compared to $Copy_2$. If the results are the same, the contents of $Copy_1$ or $Copy_2$ is copied into Copy 0.

If the results do not match, the double execution is restarted again using Copy 0. This method belongs to the DWC (Double execution With Comparison) class of fault tolerance techniques. The following issues are detailed in the next sections

- How can the limits of a processing element be defined without inserting anything in the code at/or before compile time?
- How can protected memory for application processes be implemented in the operating system?
- How can DWC of application processes be implemented inside the operating system?
- How must external (e.g. interrupts) and internal (e.g. traps and faults) events be handled in the frame of BHT hardening?
- How must memory areas shared by several hardened processes be taken into account by the hardening code inside the operating system?
- How must the results of system calls be incorporated in Copy0, Copy1 and Copy2?
- How can the caches be kept consistent with the memory by the hardening code in the operating system?

5.3 Delimiting processing elements

As in the case of the stand alone program of [LML11], the application process is divided in processing elements (PE). A processing element is

a set of instructions executed within a thread of the running process. In [LML11] , processing elements were defined when the program was designed, which is not possible here because application programs could have been designed anytime anywhere in the world.

Finding suitable statically defined subprogram entities usable as processing elements and manageable by the operating system is not possible because there is no way to make sure that they meet the requirements in duration, atomicity and idempotency. So it is necessary to dynamically find events that can constitute frontiers for the PE in the process execution flow.

5.3.1 System calls as frontier

Indeed, the possible statically defined candidates accessible to the operating system are the whole process itself, its threads and the piece of code between two successive system calls of a single thread (assuming no interrupt has to be handled in between), because these depend only on the code of the program. Some exceptions, such as page faults are also linked to the code. They are discussed later. The whole process and its threads do, in many instances not meet any of the three requirements. The piece of code between two system calls could be a good candidate.

Indeed, because an application process only interacts with the outside world trough system calls (if it does not use shared memory, or instructions reading external state information such as RDTSC in IA32), if a snapshot of its state is taken by the kernel before returning back to user mode at the end of a system call, the processing element from this system call to the next is both atomic and idempotent, because nothing but the memory of the process will have changed and this can be safely discarded and replaced by the snapshot taken before starting the processing element, if needed. (Döbel had come independently to the same conclusion [Döb14]). However the delay between two system calls can be too long to be sure that only a single SEU can occur in between. In order to avoid this, one must add something to interrupt the process before the next system call, if it does not come early enough.

As a general rule, processing elements can not thus be defined statically in all cases.

5.3.2 Timeout as frontier

The most obvious way to stop a program when its duration exceeds some limit is to use a timeout. Time-outs exist in any time sharing operating system because, unless they were interrupted before by an external event, processes are preempted at the end of their scheduling time quantum. Unfortunately this time quantum is usually much longer than the time in which only a single SEU can be expected. Even the system clock period, of which the time quantum is usually a multiple, can be too long. In the particular case of Minix3, it is around 17 msec (60 ticks/sec). Shortening this delay means more frequent clock interrupts and this is acceptable according to the results presented in the preceding section. However, the system clock has another, more critical, default: it is asynchronous with the execution of the program, so, even if its period were short enough, it could not be used to terminate a processing element because each processing element must be executed twice and the clock could not stop it twice exactly after the same instruction. The timer that must end a processing element must be started when the execution of the processing element is started. Using a dedicated hardware timer to trigger an interrupt and stop the running processing element after the required delay T does not work either, even if the hardware timer is synchronized on the same clock as the processor because the process time is suspended during other external interrupts and the hardware counter will go on running during this time. Actually what needs to be counted is not the cycles of the oscillator driving the processor but the instructions of the program being hardened. That way, and only that way, the counter will be synchronous with the program.

The real requirement for DWC in BHT is to execute twice exactly the same sequence of instructions and doing it in a time short enough to avoid having to take care of more than one SEU.

Therefore, a timer could be used for the first execution in order to stop the execution after the maximum acceptable execution time but using it also for the second execution will not necessarily stop the execution after the same number of instructions.

5.3.3 Breakpoint and Timeout as frontier

One possibility, inspired by the design of debuggers, would be to temporally replace the instruction following the last instruction executed in the first execution with a breakpoint instruction for the second execution, in order to trigger a breakpoint exception exactly after the instruction

where the previous execution was interrupted by the time-out. The problem with this approach is that the breakpoint will be triggered the first time the processing element arrives at that address but it might have to pass there many times before the "right time". There are thus two problems.

First, in order to pass over the breakpoint location, this breakpoint will have to be removed and replaced by the original instruction, that will have to be single stepped, then the breakpoint will be reinserted and the processing will go on until the next encounter of the breakpoint. This can significantly slow down the running processing element and involve a high performance loss. Second, how is it possible to decide that the breakpoint has been reached the "right" number of times: because the breakpoints are only used in the second execution, the right number of occurrences of this breakpoint is unknown when starting the second execution. Therefore the only way to know is to check a timer to see if its value is the duration of the first execution. This is almost as imprecise as working with the timer alone, so trying to use breakpoints that way is useless.

The problem is thus to find a counter that is synchronous with the execution of the instructions of the program, even if the time taken to execute these instructions differs in the two executions, because of cache misses, of external interrupts causing delays until the pipeline is emptied, etc.

5.3.4 Instruction retirement counter

Fortunately, recent architectures such as IA-32 and Intel 64 architectures include a set of performance monitoring counters in Model Specific Registers (MSR) which can count several runtime execution parameters of the process: the number of retired instructions, the number of cache misses, the number of cycles and so on can be counted. A NMI can be generated when such a counter overflows. Since it is possible to configure the counter to be only active when the computer runs in USER mode (privilege level 3), instructions executed within interrupt routines (clock interrupts, page faults, external interrupts), are not taken into account because they are executed in another mode (privilege level 0) [TW87]. Counting the number of retired instructions with this infrastructure is exactly what is needed.

Intel performance monitoring counters are available on Intel x86 architectures since the Pentium (1991). Three components are involved,

the performance monitoring selector, the performance monitoring counter and a set of instructions to read and write both the controller and the counter. The performance monitoring selector is used to set the behavior of the performance counter. In this case, the performance counter must be set to count only the instructions retired of the user process. Then the selector must be enabled to allow the processor to trigger a NMI when the counter overflows. The APIC and ACPI facilities allow the processor to trigger such exceptions. They are called Inter Processor Interrupts. The performance monitoring selector and the performance monitoring counter are read with the instruction RDMSR and are written with the instruction WRMSR. Because the performance monitoring infrastructure is used here to count retired instructions, it will henceforth be called the "retirement counter".

5.3.5 Converting time to numbers of instructions

However, since the retirement counter counts instructions and not time, it is not a time-out. There is thus a dilemma: counting time is necessary to respect the maximum processing element execution time requirement, but counting micro-instructions is the only way to execute twice exactly the same instructions. This dilemma can be solved in two ways:

1. For the first execution of the processing element initialize the instruction retirement counter just before starting the processing element and do not activate the NMI, but use a time counter to stop the processing element and then, read the value of the retirement counter in the interrupt routine stopping the first execution of the processing element. Then use that value for the second execution and let the retirement counter overflow stop the second execution of the processing element with a NMI.

2. The time requirement for the processing element is an upper limit, thus everything will be fine as long as the processing element is stopped before that maximum time, so another solution is to convert time into a number of instructions that will be executed in a time shorter than the limit. Since all instructions do not take the same time only a mean number of instructions and a variance could be obtained, but with these data one could find a "safe" number of micro-instructions. This way, it is possible to use the retirement counter to stop both executions.

Table 5.1: Converting time in a number of instructions

Measurements have been performed on an Intel Core2 Quad CPU Q9400 2,66GHzx4. Only one core was used. The test was performed with a time slice of 1/60 sec (about 17 msec), the normal tick value for Minix. We can infer that for a time slice of $250\mu s$ the average instruction will be 1,5% of these, but the standard deviation might be higher.			
Program type	Average instruction count	Standard deviation	Standard deviation %
Pure computation (Drhystone)	84056488	1611308	2%
This is about 2 instructions per clock cycle. This is possible because each core (only one is used here) of a processor with a Core2 architecture can decode 4 X86 instructions per clock cycle and issue 4 micro instructions that, in most instances correspond exactly to x86 instructions. This means that user programs use effectively half the raw computing power of the processor, which is not bad at all.			

Here, we use the second technique, because it does not need an extra timer and is simpler to implement, as will become clearer in the following.

5.3.6 Frontier of a processing element

The frontier of a processing element could be:

1. a kernel call or a system call initiated by the process itself: these are statically defined events, implicitly defined when the program was written;

2. a retirement counter overflow exception.

3. An exception, other than a hardened page fault exception: these are faults caused either by a programming error or a SEU.

All other events do not constitute the normal end point of the processing element. Among these events, there are the clock interrupts, page fault exceptions caused by hardening code itself and all kind of external interrupts

5.4 Discussion on issues raised by retirement counter

Counting instruction exactly using the performance monitoring infrastructure to count instructions exactly involves two types of problems. First, a NMI (or any other type of interrupt) will not stop immediately a modern superscalar core, and, second, the precision of the performance monitoring counters is intended for measuring performance indicators but not for exactly counting instructions: the problem is similar to using floating point for accounting applicationsÂă: these do not require relative precision but absolute precision (up to the cent, whether one counts tens of cents or billions of euros). Using floating point for accounting applications is possible but requires precautions. Using performance monitoring counters for counting exactly instructions is possible but requires a good understanding of what exactly this infrastructure counts. These two aspects are discussed below.

First, using the retirement counter, the two executions will, apparently, stop after exactly the same number of instructions, but, unfortunately, in some cases, they will not. The reason is that the NMI is

triggered when the instruction retirement counter overflows. A NMI is considered as an "external interrupt". As such it will be processed between two architectural instructions. After the execution of the NMI routine or later, the execution of the interrupted program will proceed with the next architectural instruction of this interrupted program. In a super-scalar pipelined processor, such as the recent Intel 32 and 64 bit processors, micro-instructions corresponding to many architectural instructions are being executed simultaneously in the big factory that such a processor is. The Reorder Buffer, that is the last stage in the pipeline, takes care that the effect is the same as if the architectural instructions had been executed sequentially, one at a time, and the NMI is processed when one of the architectural instructions is finished. Which one is not specified in public documents. The simplest way to achieve that is probably to just block the entrance to the pipeline when the NMI is detected; the actual processing of the NMI would then only be fired when all the instructions inside the pipeline would be finished (or aborted in case of failed speculative execution). Since instruction retirement is the last step of the pipeline, the extra number of instructions would be up to the number of instructions lingering in the pipeline at that time. This number could be huge: the "Reservation Station" (RS on the schematic of Annex 1) where decoded instructions wait for a free execution unit, has 18 entries in processors without hyper threading such as the Core2 and the Core i5 used in tests or per thread in Core i7 processors. There are 6 execution units and 128 positions in the Reorder Buffer (ROB): total 152 locations where there could be instructions, all of them already decoded but not yet retired (see Annex 1). It will often be less because there can be bubbles in the pipelines and because some instructions consist of several micro instructions, each using a location.

When the first solution (using a timer to end the first execution and the retirement counter to end the second) is used, the number of instructions executed in the two executions of the processing element is almost always different. Indeed, when the time counter of the first execution triggers an interrupt, the value in the retirement counter readable in the interrupt routine will be that of the last instruction that entered the pipeline before the timer interrupt. So, if that value is used to stop the second execution, in this second execution, the NMI will occur when that instruction leaves the pipeline and all the instructions in the pipeline at that time will have been executed. Thus, the second execution will normally be longer than the first. This fact cannot be ignored because it will happen again if the processing element is restarted: ignoring this

and just restarting the PE would block the program for ever. In the second execution, the value to use to control the issuance of the NMI must be the number read in the retirement counter between the two executions minus the number of instructions that will be in the pipeline when the instruction, the retirement of which causes the NMI, is retired, at the end of the second execution. That number cannot be known in advance.

There is a workaround: after the end of the second execution, the first execution can be continued in single stepping mode (trap after each architectural instruction) until the number of instructions of the second execution is reached.

In the second solution (using the retirement counter to stop both executions), the two executions might stop at the same instruction (which is never the case with the first solution, as just explained) and even if the two executions do not stop after the same architectural instruction (situations where this happens will be explained below), they should be closer to each other. Here too, single stepping can be necessary but for fewer steps.

Tests have shown that, if no interrupt nor exception happens during any of the two executions of the PE, both executions always stop exactly at the same instruction. The effects of exceptions and interrupts are discussed below.

The second problem with the retirement counter is the following: it has been observed in some rare instances that the second execution stops either exactly one micro- instruction before or after the first execution. Tests have shown that this happens when the execution stopping one instruction earlier than the other has been subject to an interrupt. However, in a superscalar architecture, interrupts are handled after emptying the pipeline and the most straightforward way to detect this emptiness is to inject in the pipeline a dummy NOP micro- instruction flagged as depending of the results of all the micro-instructions that are in the pipeline at this time. The pipeline will be empty exactly when that pseudo NOP micro-instruction will be retired. Of course, this explanation is only a speculation but it matches the observed behaviour and, besides, AMD announces that, in their processors, interrupts add 1 to their retirement counter [Adv16].

This effect (1 added to the retirement counter for each interrupt or page fault) is also observed when page faults occur in memory write operations to a page not yet accessed in this execution of the PE.

A closer look at what can happen will clarify the implications and

how to stop exactly after the same architectural instruction.

Let N be the number of instructions programmed for triggering the NMI in the first execution. At the end of this execution, the value read in the retirement counter in the NMI handling routine will be

$$Ne_1 = N + x_1 \tag{5.2}$$

where x_1 is the number of instructions already in the pipeline when happens the NMI caused by the retirement of the Nth micro-instruction of the processing element. Thus x_1 is known because N is a constant and Ne_1 can be read in the NMI handling routine.

There will be exceptions during the first execution: page faults (as will be explained Section 5.7 on the implementation of protected memory). There might be other ones, but these are caused by program faults, and not considered in this section.

The number of page faults could be different in the two executions, in this case, even if the same value N is used in the two executions and even if x_1 is taken into account, the two executions can involve different numbers of instructions, because of the NOPS inserted when handling interrupts as explained above.

At the end of the first execution, the real number of instructions executed is thus

$$N_1 + x_1 - y_1 \tag{5.3}$$

where y_1 is the number of page faults that happened during the first execution

If the retirement counter is again initialized, between the two executions, to trigger a NMI after N instructions, the number of instructions of the second execution will be

$$N + x_2 - y_2 \tag{5.4}$$

x_2 and y_2 can be known like x_1 and y_1.

The only thing to do in order to have exactly the same number of architectural instructions in the two executions is to single step the execution lagging behind the other until

$$N + x_1 - y_1 = N_2 + x_2 - y_2 \tag{5.5}$$

5.5 Precise event based sample

Intel has made available a functionality based from the Intel core architecture, precise event based sample [Spr02a, Spr02b]. This feature

uses a debug store and interrupt performance counter to store a set of architectural state of the processor just after the instruction that generated the overflow exception. This makes it possible to have the precise processor state when the performance event occurred. That is useful for debugging purpose. But this does not stop the execution flow as soon as the overflow interrupt is triggered. It is thus not useful in BHT.

5.6 Double execution With Comparison (DWC) for Minix3 processes

The principle of Double execution With Comparison (DWC) is to execute twice the same code starting with the same initial data and comparing the results of the computation. If they match, they are assumed to be correct and execution proceeds, else execution is restarted from the same initial data. The fact that the two executions occur in the same thread, one after the other, in two threads, strictly in parallel or one thread slightly delayed with respect to the other is not important for DWC. What is important is to have strictly the same initial data for the two executions but also to keep them for later, in case the execution must be restarted. These initial data are the whole process state.

5.6.1 DWC in Minix3

In Minix 3, application processes are tasks running at privilege level 3. Like other processes they are characterized by their memory space (stack, heap, bss, data and text), their context (internal processor registers) and state data maintained by the micro-kernel and servers processes and the initial internal state of the processor.

- The memory space of the process is built at fork system call time (duplication of the parent process memory space to create the child's memory space) or exec system call (creation of a new core image for the process) and augmented each time the stack or the heap grows.
- The process context is the set of internal register values of the processor related to the process when it is running, i.e. any register the contents of which could influence the execution of the PE.
- The state data of each process in the OS are scattered in data structures of the micro-kernel, the memory manager (VM), the

process manager (PM), the virtual file server (VFS), the reincarnation server (RS) and the scheduler (SCHED). Each server has its own process table where it maintains the data structures needed to respond to a request from the process. These state data are private data for each of these server processes and they cannot be modified during any of the executions of a PE except in an interrupt routine and even if an interrupt routine changes this state, this will not interfere with the executions of the PE because the process can only be made aware of this change through a system call. Thus these state data need not be taken into account for the DWC.

Only the memory manager and the micro-kernel have been modified to harden application processes: the VM manages the process memory space and the micro-kernel modifies the process state and takes care of the process context data. The other server processes are not involved during the DWC steps. The micro-kernel is also responsible, when a processing element has succeeded its DWC, to decide to continue with the next processing element of the same process or to choose the next running process, as proposed by SCHED.

5.6.2 Steps of DWC

The steps to perform a sequential Double execution With Comparison of processing elements in a single thread are as follows: When the process is ready to start, at the end of processing the "fork" system call, when Minix 3 has finished preparing the new process for execution, the part of the state of the process used in hardening consists of the initial context called here R_0, and a set of frames called here US_0, including the variables of the process (stack and static variables, the heap is empty at that time).

1. The micro-kernel creates a copy, called here R_0, of the initial context of the processing element in its data structures. At that time, US_1 and US_2 have already been allocated. At that time R_1 and R_2 are empty. All pages of the PE are mapped to US_0 as read-only. R_1 and US_1 will be used during the first execution and R_2 and US_2 during the second execution.

2. Before the first execution begins, the data pages of the processing element point to the frames of US_1. They are set to read-only

thus not accessible in writing by the process. During that first execution, several events can occur:

- copy-on-write page faults happening either after a fork like usually in Minix3. These page faults are processed by the VM server, like always in Minix3. The US_0 frames inherited from the father are replaced by new US_0 frames privates to the child. That ends the PE.
- hardening caused page faults on US_1 frames. These page faults are handled by the micro-kernel that gives access to the process and put the page in the process working set.

3. At the end of the first execution, the values of R_1 have been produced. The contents of the corresponding frames of US_1 has changed. The modified set of frames of US_1 is called "working set".

4. The micro-kernel keeps R_1 and redirects the data pages of the process towards US_0 and sets all pages as read-only.

5. The second execution of the processing element is performed using R_2 and US_2. During that second execution, several events can occur, in particular, the same page faults as during the first execution. At the end, the values of R_2 and US_2 have been changed.

6. The micro-kernel compares R_1 to R_2. If they match, it compares US_1 to US_2. If they match, it copies US_2 to US_0, R_2 to R_0. There are thus now 3 identical copies of each element of information. If one or more of the comparisons fail, the values R_0 and US_0 are respectively copied in R_1, R_2, US_1 and US_2 and the sequence is restarted from these values.

In all future processing elements the DWC is performed as follows:

1. Before first execution of the PE, the data pages of the processing element point to the frames of US_1. They are set to read only except if they were modified during the previous PE.

2. During the first execution, several events can occur,

   - in particular, copy-on-write page faults may still happen, like usually in Minix 3 .These page faults are processed by the VM server, if they did not yet exist in US_1 and US_2. That ends the PE.

- Or because of attempts to write in read-only pages of US_1. These pages are set to read-write by the micro-kernel which adds them to the "working set".

3. At the end of the first execution, the values of R_1 are saved in PRAM.

4. The micro-kernel saves R_1 and restores the saved R_2; the micro-kernel redirects the data pages of the process towards US_2. Before starting the second execution US_2 includes all the frames corresponding to those of US_1 at the end of the first execution

5. The second execution of the processing element is performed using R_2 and US_2. Normally, the page faults that could occur during the second execution are the same as in the first execution At the end, the values of R_2 and US_2 have been changed.

6. The micro-kernel compares R_1 to R_2. If they are identical, it compares the pages of the working set in US_1 and US_2. If they match, it copies the modified pages of US_2 to US_0 and R_2 to R_0 There are thus now 3 identical copies of each element of information. If one or more of the comparisons fail, the original values R_0 and US_0 are respectively copied in R_1, R_2, US_1 and US_2 and the sequence is restarted from these values

5.7 Protected memory

5.7.1 Definition

Protected memory (PRAM) is memory that cannot be modified by a direct hit by a charged particle nor indirectly by any process behaving in an erratic way after having been victim of a SEU. The running process can not write in PRAM. However, it will be shown that protected memory may be read by the running process. Information in PRAM is sufficient to restart the system whatever happens due to a SEU.

5.7.2 How to protect PRAM against direct SEU effects?

Protection of memory against direct hits is classic: that is why ECC memories have been invented and improved with scrubbing and multibit protection for years [EM00] [MB05]. However, protecting central memory with multibit ECC and scrubbing is not enough. The program

memory space consists of different levels: main memory, but also , L1, L2 and L3 data caches, L1, L2 and L3 instructions caches (some caches can be common to program and data). Fortunately caches and TLBs are protected for the target processor (XEON lines and ATOM) [cgIc11]: TLBs as well as L2 and L3 caches are the best protected memory cells in a computer and corruptions to L1 caches could only cause harm to the current discardable copy of the PE variables: such faults are discussed in chapter 3.

5.7.3 How to protect PRAM against indirect SEU effects?

On the other hand, the user processes can misbehave, but, in Minix 3, processes are protected against each other (this is part of the job of the VM server), so the main threat is the hardened process itself. The user accessible registers (called here the context), located in the processor, are not protected and the running process can become erratic after having been subject to a SEU hitting the processor and it can wrongly modify values in its memory space.

Because processes can misbehave, they may not be allowed to modify their memory. Therefore, when they want to modify their memory, they must do it on a discardable copy, one for each of the two executions (these partial copies are called US_1 and US_2). Only when the two modified copies will have been found identical after the two executions, can the original memory area of the process (called US_0) be updated with precaution by the OS.

PRAM is implemented in the same way as protection against modifications of the state of any process by any other process, i.e. using the pagination memory management unit (MMU) of the processor: the standard Minix 3 VM server restricts access of a process to the memory space that has been allocated to it and lets the program modify its data (the code is read-only). In the same way, a process is protected against itself by modifying the VM server to place all its data in read only mode using the pagination memory management unit of the processor.

Of course, for this kind of protection to be effective, no SEU caused error may modify the data used by the MMU in a way causing a change in data located in PRAM. MMU data consists of page tables that are in the memory space of the VM server, thus safe, and processor registers, which might not be safe. The involved processor registers are those in the Translation Lookaside Buffer (TLB) and the registers used for memory

write operations. Whatever the way it is implemented, a TLB is always an associative memory allowing to find a physical frame number given a page number.

If a SEU hits the page number, the faulty page number can be out of the addressing space of the process and will never be used (because only a single SEU needs to be considered in a PE), the fault will not cause any error, just the loss of a few nanoseconds to replace this useless entry. The faulty page number can also be another page number of the process. It could be used but this could only result in faulty data to be read (which will, at worst, cause reexecution of the PE) or written, which could only happen in a writable frame belonging to the process. Since PE can only write to discardable copies, such a fault can, at worst, cause reexecution of the PE. Thus a SEU in the page number cannot harm protected memory, i.e. memory in which the PE is not allowed to write.

On the other hand, if there is a fault in the frame number, anything could be modified anywhere in memory. Therefore these memory cells are the best protected in a computer and really well protected in Xeons and Atoms, because, if they were not, any other protection scheme would be totally useless [cgIc11].

The valid bit is always true because all entries are always used, except just after a TLB flush, but this is rare, does not last long and then all entries are zeroed; thus a fault on the valid bit could only bring it from true to false and invalidate a good entry; as above, only a few nanoseconds would be lost. Therefore using the TLB as our foundation for implementing protected memory is safe.

5.7.4 How to protect process's memory against itself?

5.7.4.1 First approach

A first solution is, that used after a "fork" system call to avoid copying all the memory of the father process in the child's. Indeed, in most instances, the child will call an "exec" immediately after the fork to install a new program in his memory. Thus sharing the code between father and child and allocating lazily data space to the child ("on demand") reduces copying data that might not be used. The adaptation of this method to PRAM is to make the data of the process inaccessible to write access when the first execution of the processing element starts. Of course, as soon as the process tries to access its memory, a page fault

occurs. The VM server can be modified to copy the contents of the page in a second frame and replace the original frame with the copy in the process page tables and set the page in read-write mode.

This way, all the pages that the process needs to access would, one after the other, be replaced by read-write copies during the first execution of the processing element. Between the two executions, the frames corresponding to these read-write pages, accessed during the first execution, would be saved in the PRAM and replaced in the memory space of the process by fresh copies of their original counterparts hidden in PRAM. The same page faults will occur during the second run. They will be handled in the same way so the two execution will be independent to each other.

Only the final memory contents of the process, at the end of the DWC, would be passed to the next PE. All other frames could be freed. This solution minimizes the amount of memory space used for hardening and avoids consistency problems between multiple copies of the same data in different frames, but involves copying several pages for each processing element and as many context changes between the microkernel and the VM server. It is thus very expensive in computing time.

5.7.4.2 Second approach

Another solution exists. It is much less expensive in computing time. In this solution 3 copies of the user space memory are used: US_0, that is always in PRAM, is complete (code and all data) and only changes atomically at the end of each processing element. US_1 and US_2, only include the frames of the process that have been modified since the beginning of the execution of the process. US_1 is used during the first execution and US_2 during the second. The frames of US_0, US_1 and US_2 are always in PRAM, except a few frames of US_1 during the first execution and of US_2 during the second execution, as explained below.

In this solution, like in the first one, when the first execution of the processing element starts, the code pages of the process are in read-only (RO) mode and the data pages of the memory space are set to read-only (RO) too, thus not accessible in writing by the process. The pages point to frames of US_1, if they exist, otherwise, of US_0. When the US_1 frames exist, they are set to read-write when they have been modified by the previous PE or by the system call handler or kernel call handler. Otherwise they are set to read-only.

Of course, as soon as the process tries to access any data page in the

read-only part of its memory, a page fault occurs. If that page is not yet in US_1, a copy on write is performed with the help of the VM, like above (This can happen only once for each page in the whole execution of the hardened process). But if the page is already in US_1, then the VM is not involved: the micro-kernel has been modified to simply set the page to read write mode. This way, all the pages that the process needs to access will, one after the other, become accessible during the first execution of the processing element.

Between the two executions, US_1 frames will be replaced by US_2 frames in the pages of the process for the second execution. The same page faults will occur as in the first execution and the micro-kernel will set these pages to read write mode. At the end of the processing element, the two sets of modified frames (or their CRC, if possible with the processor in use) are compared.

If they match, the frames modified in the second execution (US_2) will be copied in the frames dating from the end of the previous processing element (US_0). The pages will then be pointed to frames of US_0 until the next PE, so the hardening copy-on-write will remain transparent for the whole system. When standard Minix code is executed, the pages of the process point to frames of US_0, the only ones that are known by standard Minix. If the compared frames do not match, the original frames (US_0) are copied in those of US_1 and US_2 and the double execution of the processing element is restarted. In this second solution, the processing time is much lower because avoidable frame copies are avoided, but the existence of several copies of the same information requires care to avoid inconsistencies. Changes to US_0 by Minix3 or others processes sharing memory regions with this one must be reflected in US_1 and US_2 just before starting the next PE.

5.7.4.3 Third approach

The second approach involves the VM server during execution of the processing element. Tests have shown that, this is very expensive in performance lost. The approach has been modified to eliminate the intervention of the VM server during execution of the processing element. US_1 and US_2 frames are allocated when the VM sets the process page table. Before starting the hardened execution of the process US_1 and US_2 are already allocated. They correspond to read-write data pages in the process's table page. To avoid comparing all the memory space of the process hardened page faults are caused. But they are handled by

the micro-kernel that gives access to the process and puts the page into the working set of the process. Only the pages present in the working set are compared during the comparison phase.

5.8 Handling other events during double execution

5.8.1 Asynchronous and synchronous events compared to process execution flow

Minix is a multi task event driven micro-kernel operating system. These events come from outside and inside the running process. Some of these events can interrupt the running process. The outside events which could interrupt the running process are hardware and clock interrupts. Outside events are asynchronous relatively to the process execution and totally transparent to this process. When a processing element is restarted these events will not happen again. Other events of the same type can happen but normally not exactly at the same point during the execution of the process.

The inside events which could interrupt the running process are system calls initiated by the process itself and exceptions caused by the process: division by zero, page fault, etc. Such events are synchronous with the process because, if we restart a processing element in the same conditions (with the same system data, memory space and context) they will re-occur at the same point during the process execution.

When an event (inside event or an outside event) occurs, the execution of the process is suspended, an event handling routine is executed by the operating system, then the execution of the process is resumed.

5.8.2 Interrupt handling mechanism in processor

In the epoch when computers were executing a single instruction at a time, external events were handled at the end of the current instruction. Now, in pipeline processors, there are not one but several instructions being executed in the pipeline at any time. The general rule is that the event will be handled between two architectural instructions, i.e. when an architectural instruction is retired, in order to keep the atomicity of the instructions and to need only the program counter to know where to restart the process after processing the event, but the rule does not say after the retirement of which architectural instruction. Actually, which

architectural instruction is selected is irrelevant for the behavior of the processor, because the only difference would be a delay of at most a few nanosecond for the start time of the event handling function. In many processors, it is the last architectural instruction that entered the pipeline, because, to implement this, the entrance to the pipeline has only to be blocked until that instruction is retired. This is not in the reference manual but an educated guess confirmed by tests. When the interrupt handling routine starts, the pipeline is thus empty and, in order to be able to resume the interrupted process, the processor must only save the instruction pointer and the status register(s) of the process (in most instances other registers accessible to the program will be saved in software in the interrupt routine). Two things must be noted here:

1. The number of instructions leaving the pipeline between the interrupt and the start of the interrupt handling routine is at most the number of places in the pipeline but may be lower because there may be bubbles in the pipeline because some instructions could be needing resources or data not yet made available by another instruction, ahead in the pipeline.

2. The performance monitoring counter of retired instructions, mentioned in section 5.3.4, generates NMI interrupts, not exceptions and these events are thus considered as outside events although they are perfectly synchronized with the process execution. The event occurs after the given number of instructions has been retired, but the event will only be processed when the pipeline is empty. This is why if the retirement counter is checked in the interrupt routine, it will not point to the instruction whose retirement caused the interrupt but to one of the following instructions.

When an inside event occurs, the "simple" solution of waiting for the pipeline to be empty is not acceptable: for instance, if a page fault occurs, it must be handled immediately, i. e. in the middle of the execution of an instruction.

This means much more work (for the processor) than for an outside event, because the entire internal state of the processor, including the state of all the instructions in the pipeline should be saved if the processing were to be resumed exactly from where it stopped (Motorola 68000 processors were working that way). Besides, after the end of the event handling routine, the process may not always be resumed with the next instruction, because the event (in this instance, the page fault) occurs

before the instruction is completed. After solving the cause of the page fault, the instruction must then either be continued from the point were it was suspended (as in Motorola 68000 processors) or the effect must be undone and the instruction must be restarted. This second possibility is the choice of the IA-32 and Intel 64 architectures. Fortunately, programmers of these architectures do not need to take care of this because the effect is the same. They must only know to which of 4 categories the event belongs:

1. Traps: INT I, breakpoints, etc.: the process is resumed after the current instruction.
2. SYSENTER or SYSCALL instructions used by Minix3 for system calls: effect similar to an INT but faster. They need not be transparent because they are part of the process itself. The process is resumed after the current instruction. Actually, they are just a subroutine call that includes a change of the protection level from 3 to 0, which gives access to the code of the function into the kernel space that is included in the memory of all processes, but not accessible when they run in protection level 3.
3. Faults: page faults, divisions by zero, etc.: the process is resumed before the current instruction.
4. Aborts (double faults such as page fault within a page fault handling function, etc.): the process cannot be resumed.

All the possible events must, of course, be taken into account by the hardening software. The good news is that event processing is not done in user mode, thus the instructions performed to handle events are not taken into account by the retirement counter. Besides, the fact that the duration of the processing element could be longer than the time constraint to avoid double SEU is not important because SEU are uncorrelated with each other; thus, if an execution of a processing element is sliced by events, what counts is the sum of the durations of the slices, which remains nearly the same as without events (only the number of bubbles in the pipelines may change).

5.8.3 Scheduling issues

Another common characteristic of typical event handling routines is that they could change the relative priority of processes and cause a rescheduling. The Minix scheduler consists of two parts. The first is the handling

of the scheduling queues in the SCHED server. This one decides in what order user processes should be allowed to run, but does nothing. The second is the picker, in the micro-kernel, that actually replaces the running process by the first user process of the highest priority scheduling queue or by a server process of its own choice [SHT10].

The picker, in the micro-kernel, has been modified by adding state variables to run the processing element twice before switching to another process. The picker will unconditionally select the current hardened process until the end of the DWC processing of the processing element. Interrupt handling routines may let the SCHED server update its priority queues, but the picker will only consult the SCHED server between processing elements, not after interrupts, system calls or NMI caused by the retirement counter overflow, of course, if the process is still runnable. Delaying process priority changes by interrupts until the end of the current processing element is not a necessity but a simplification that should not have a significant influence on the system behavior because processing elements are short.

The different types of events and their handling in the context of BHT hardening are as follows.

5.8.3.1 Clock and others external interrupts

These could only influence the execution of the processing element by changing the priority of its process if its time slot is elapsed. This may not happen because it would stop the PE in a way that cannot be reproduced in the other run and the PE would have to be restarted. Therefore, the only thing to change is, in case the event changed the priority of the running process, not to call the picker immediately but to note to call it before starting the next processing element.

5.8.3.2 Page faults

When BHT is used, there are four types of page faults,

- real ones, occurring typically because the page is swapped out (which never happens because Minix3 does not support swapping, but might happen in other cases such as lazy loading memory mapped files for instance); these are identified because the page has no associated frame. These normally end the PE.

- Copy-on-write page faults identified because the page is valid, is in

read-only mode but is a data page pointing to an associated frame belonging to the parent process. These normally end the PE.

- Copy-on-write page faults identified because the page is in read-only mode but is a data page pointing to an associated frame belonging to US_0.Their processing is then the standard one of Minix 3, except that they are added both to US_1 and US_2. The page points to the US_1 frame and is changed to read-write.

- Other page faults. These occur during both executions. The page is in read-only mode and is pointing to an associated frame of US_1 or US_2. The micro-kernel has been modified to handle these. When they occur, it sets the page to read-write mode. The page is added to the working set.

5.8.3.3 System calls

A system call (SYSENTER or SYSCALL instruction on IA-32) ends the processing element, as explained in section 5.3.1.

5.8.3.4 Breakpoints

A type of system call used for debugging purposes. Can be inserted at compile time or at execution time, but this necessitates modifying the code during execution. Nothing to change from the classical Minix 3 behavior.

5.8.3.5 Others exceptions

Exceptions can have two causes: they can be the consequence of a programming error (say a dangling pointer) or of a SEU. Exceptions are maybe the most frequent and easiest way to detect a SEU and correct its effect because they are detected by the standard hardware of the processor and, if it is the exception event itself that is caused by the SEU, it won't cause any harm, just the unnecessary restart of the processing element.

The only problem is to distinguish exceptions caused by programming errors from SEU induced exceptions. This is easy because SEU induced exceptions are random events asynchronous with the program, while exceptions caused by programming errors are synchronous with the program. In order to distinguish the two kinds of exceptions, one

must note where the exception occurred in the processing element (address and/or value of the retirement counter) and restart the processing element. If the exception had been caused by a programming error it will happen again at the same place. In that case the standard procedure of Minix 3 must be followed (abort the process) else, it was a SEU, and it will not happen a second time, at least not just in the same place and restarting the processing element was the right thing to do.

5.9 Modifications of the process memory by the operating system or other processes

5.9.1 Results of system calls

Results of system calls must be handed over to the process. In Minix 3 there are traditional synchronous system calls (the result is transmitted when the system call returns). In this case the results can be in the registers or in messages and buffers in the process memory. Minix 3 also supports asynchronous system calls. In this case, the result is provided later (at the end of another processing element in a hardened Minix3). Minix 3 sends then a message to the process to warn it that the result of an earlier system call has been provided. A message is actually a 64 bytes data structure written by the OS in the process memory in a reserved area just above the stack. Buffers are OS frames mapped into the process addressing space.

All this must of course be taken into account in the triplicated system used here: the "standard" OS will modify R_0, and US_0.

R_1, R_2, US_1 and US_2 must be modified accordingly by the hardening code. R_0 and the message data structures are modified by the micro-kernel; they are "small" data structures and their modifications are simply performed 3 times on R_0, R_1, and R_2 or on US_0, US_1 and US_2. The buffers are larger. In order to cope with these, all the pages are redirected to US_0 before starting the system call processing. After the execution of the system call the frames modified in US_0 (their "dirty bit is set") are copied to US_1 and US_2.

5.9.2 Shared memories: exec, mmap & Co

5.9.2.1 Exec system call

Before an exec system call is performed, the memory of the process is released. So, if it is a hardened process, all memories frames in US_0, US_1 and US_2 are released too. The process manager, memory manager and the virtual file server builds a new memory core for the hardened process in US_0. The first time the hardened process runs:

- All read-write data pages in process's memory space are set to read-only mode.
- US_1 and US_2 will be build progressively when the process will access these pages.

5.9.2.2 Mmap system call

In case of mmap the process memory space is increased. So there will be new pages in US_0. They are set to read-only mode before the hardened process will run. Besides, the process could share the frames of these pages with another process in case of memory map of a file. This is a shared memory problem that is handled in the next subsection.

5.9.2.3 Shared memory

Besides shared memory mapped files, that can be considered as a persistent type of shared memory, Minix 3 supports classical shared memory. In both cases, the problem is threefold:

- the list of processes which share these frames must be known
- modifications of these frames by other processes must be known to the hardening code
- the modifications must be applied into US_1 and US_2 before starting again the hardened process.

Of course one situation must be strictly forbidden: sharing pages between hardened and unhardened processes. When such a situation is detected, the culprit process is killed and a message is printed in the system logs.

Obtaining the list of processes sharing a frame (whether as shared memory or by sharing the same parts of a memory mapped file) is possible.

Identifying the changes in shared pages is not a big issue: the hardening code must anyway identify all changes of the memory of hardened processes to know which frames to update in US_1 and US_2 at the end of processing elements. When these frames are shared modifying them in US_0 will also modify them in the US_0 of the processes that share it because the frames are shared. The corresponding frames in US_1 and US_2 of the other sharing process become thus inconsistent with their US_0 and must either be updated at this time or flagged as invalid and updated by copy-on write when each other hardened process starts its next execution. We implemented the second solution.

5.10 Consistency issues between caches and central memory

Recent Intel CPUs have 3 levels of cache. The first level has a Harvard architecture, i.e. there are separate caches for instructions and data. These caches are addressed using virtual addresses. However virtual addresses are not used to tag the Level 1 cache blocks. Instead the cache blocks are tagged using the physical addresses, and the virtual addresses issued by the processors are translated into physical addresses by TLB, as explained in the [Int64] [HP11].

This type of architecture using apparently virtual addresses but where cache blocks are tagged with physical addresses is commonly called "virtually addressed/physically tagged". A property of this architecture is that when an entry of one of the two Level 1 TLBs is invalidated, it is not possible any more to access the blocks of the corresponding Level 1 cache that belong to the invalidated page. In other words, these blocks are automatically invalidated. The two Level 1 caches (Instructions and data) and the unified Level 2 cache are dedicated to a single core.

The big Level 3 cache, on the other hand, is common to all the cores and is addressed via the same main TLB as the Level 1 and Level 2 caches of each core, like the central memory. Like central memory, it receives physical addresses. Its blocks are not invalidated by changes in the TLB of one of the cores. However, the block contents must be the same as the contents of the corresponding blocks in central memory.

This is true most of the time except during write operations. The caches of the IA32 architecture are "write back", which means that, from the processor's point of view, the write operation is finished as soon as the data are in the Level 1 cache. When data are written in the Level 1 data cache, the other caches are updated as soon as possible, which is rather fast for the 3 levels of cache. However, the write back scheme of IA32 is not the traditional write back, where data are only written to memory when the cache block has to be reused for other data. This traditional writeback scheme will be called here "alap writeback" (As Late As Possible). In IA 32, writing in central memory happens as soon as the bus is free. Therefore the IA32 like caches will be called here "asap writeback" caches (As Soon As Possible). Write operations to the level 1 cache have the size of the operand of the writing instruction. Write operation to the other caches have the size of a cache block, i.e. 64 bytes.

The cache related issues to consider in this work are:

1. SEU hitting caches

2. impact of the write back nature of the caches on the comparison of the modified frames by the micro-kernel

3. impact of the caches on the replacement of US_1 frames by US_2 frames between the two executions.

The possibility of a SEU affecting a cache has been taken into account in the probabilistic analysis of chapter 3 where Levels 1 caches are included in the risk. Cache Level 2 and 3 are assumed to be adequately ECC protected (DECTED); therefore a single SEU will not have any effect, assuming it will not affect 4 bits or more of the same word. Only one or several SEU on the same word block could cause a fault to be undetected if 4 bits or more are modified. This risk can be considered as negligible because a cache is a temporary storage. To make sure this "temporary" characteristic is short enough, full cache flushes must be performed often enough. Identifying how frequently cache flushes must be forced to be entirely sure that no undetected error can happen in the Level 3 cache between these flushes. Being rather rare, these flushes should have no noticeable impact on performances.

The "asap write back" nature of the caches means that the central memory will be updated shortly but not immediately after a write to memory operation. When comparing US_1 with US_2, US_1 has not been

modified for a long time, but frames of US_2 might have been modified just before the end of the second execution. However, because the micro-kernel uses the same page tables as the hardened process, it can use the logical addresses of the pages pointing to US_2 for this comparison, i.e. read them from the L1 that is up to date, while using other addresses for accessing US_1. Therefore the cache will not induce errors in the comparison.

The third problem is that there are copies of the modified blocks of the frames modified in US_1 in the Level 1 data cache that is using virtual addresses. Which means that, at the beginning of the second run unless the TLB is invalidated between the two executions, and even though the US_1 frames in memory have been replaced by US_2 frames, at the beginning of the second execution, the CPU will still use the copies in the Level 1 cache, that are not copies of US_2 frames but of US_1 frames. This means that the second execution will not use the same initial values as the first and could become erratic as if it had been hit by a SEU. The problem is that if the PE is simply restarted, the same problem will occur again. Fortunately, this situation will not happen because the TLB is invalidated automatically between the two executions because the micro-kernel subcontracts, to the VM server, the replacement of US_1 by US_2 frames and the VM server being a separate process with its own page tables, the CR3 register is modified before starting the VM server in order to replace the hardened process page tables by those of the VM and, in the Intel architecture, replacing the contents of CR3 invalidates the three TLBs and automatically invalidates the corresponding blocks of the L1 cache. Although this invalidation may take a few cycles, the delay between this TLB invalidation, and reuse, by the second execution of the hardened PE, of the same addresses is long enough.

In the current implementation, the hardening work is split between the micro-kernel, that uses the hardened process page tables when performing hardening work, and the VM server that uses its own page tables. Because of the change in the page tables, the CR3 register must be modified both between the two executions of each PE and between successive PEs, and this automatically clears the TLB and invalidates the level 1 and level2 caches. If all the work had been performed in the micro-kernel, the TLB should have been cleared explicitly.

5.11 Machine check architecture

MCA is a feature found in processor architectures that can detect hardware errors such as system bus errors, ECC errors, parity errors, cache errors, and TLB errors. And it can report them to software. It consists of specific registers called MSRs that are used to configure the MCA and a set of MSR registers called register banks that inform about the nature of the errors. The MCA is actually the functionality of the processors that informs what is really happening in the big machine that a processor is. Many errors are detected and corrected by error correction mechanism present within the processor (ECC, DECTED SECDED etc). Errors that are not correctable are reported to the software if the software has provided an ISR (interrupt software routine) for this type of error. Otherwise the processor will just reboot or crash. The MCA is based on the RAS features found in most mission-critical systems such as servers or critical system control or monitoring systems. As part of this thesis the functionality of the MCA is used to recover errors that might escape the classic management of exceptions.

5.12 Conclusion

BHT principles and stated solutions have made it clear that it is possible to allow the operating system to harden its application processes. However, some problems must be taken into account to make it possible. It has also been shown that both interrupt and exceptions will not be able to disturb the atomicity and idempotency of PEs. It has also been shown that the MMU remains a reliable mechanism on which the implementation of the BHT can rely for the implementing the protected memory. The following chapter discusses the actual implementation of the BHT method in Minix 3. The latest version was used, the 3.4.0.

CHAPTER 6

Implementation of hardening processes in the operating system, using BHT

6.1 Introduction

This chapter presents the implementation of the principles outlined in the previous chapter. Remember that the blended hardening technique used in this work is based on dividing programs in processing elements short enough to assume that the hardening must only cope with a single fault. Before starting each processing element, a snapshot of the entire state of the process is saved in protected memory. If a fault is detected, the processing element is aborted and restarted from the snapshot. If no fault is detected, a new snapshot is taken and the next processing element is started. Of course, the aim is to detect all faults. Two complementary detection techniques are used. Many faults cause exceptions or are detected by the machine check architecture that also produces exceptions. These fault are processed in the exception handling routines that immediately abort and restart the processing element. Other faults cause the PE to produce a faulty result but no exception. They are detected by double execution of the processing element and comparison of the results. If the results (i.e the complete state of the process) differ, the processing element is restarted from the original snapshot. All the hardening occurs in the operating system. Minix3 version 3.4.0 has been modified to harden the execution of its application processes The implementation was performed on the x86 architecture on which Minix3 runs. The style used is a narrative style to allow the reader to see step by step the different events that occur during the execution of a processing element (PE). Only the micro-kernel and the memory manager have been modified in this implementation. The figure 6.1 shows the global architecture of hardened Minix3. The micro-kernel part of the the hardening software are subdivided into four modules, HM (Harden-

ing Manager), HEC (Hardening Execution Control), PRAM (Protected RAM), HEH (Hardening Exception Handler). HM sets the hardening environment by initializing hardening data structures. HM is the entry point of the hardening software to enable or disable the hardening within the OS. When HM sets the environment, HEC is responsible to execute each hardened process as long as hardening is enabled. HEC handles the DWC. HEC use functionalities provided by the others modules (PRAM, HEH). PRAM implements the protected memory part. HEH implements the exception handling part of BHT. It informs the HEC module when a PE should be stopped. The VM part of the hardening software includes one hardening module, VPRAM. It is the module for providing protected memory service and handle US1 US2 memory frames within VM from their allocation to their deallocation. In the second section, the functionalities and implementation choices of the HM module will be detailed. In the third section the functionalities and implementation choices of HEC module will be detailed. In the fourth section,the implementation of the protected memory will be presented. The presentation will include both the kernel part and the VM part. In the fifth section the management of exceptions will be addressed including the management of retirement counter, of the MCA exceptions, the single stepping and the faults detected by hardware. In the sixth section the changes made to the Minix3 operating system will also be discussed. And finally the last section presents the user interface of the hardening software.

6.2 Hardening Manager

6.2.1 Requirements

This module is the interface between the hardening software and the rest of the operating system. It contains the following features:

- Initialization of the hardening data structures in the micro-kernel and in VM
- Enabling or disabling hardening in the micro-kernel and in VM
- Starting the HEC module

6.2.2 Implementing the Hardening Manager

The data structures and hardening state variables are initialized with the function *init_hardening*. (See listing A.3.1). They are set to their

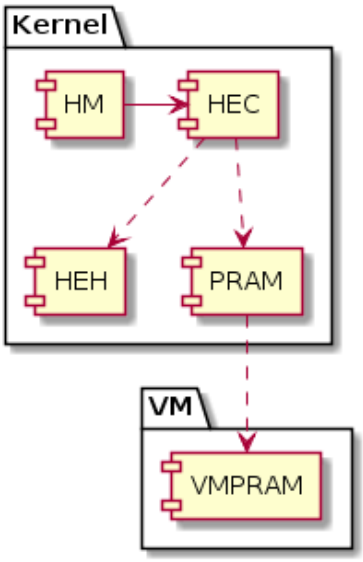

Figure 6.1: Hardening global architecture

respective default value. The instruction retirement counters and the machine check architecture are also initialized. The table 6.1 shows the state variables, their meaning, and their default values.

Another feature of the Hardening Manager is enabling or disabling hardening. This means that when you start the operating system, hardening is not enabled by default. It can be activated through a user library available on the command line. Hardening must therefore be activated in the micro-kernel and in the VM. This is done using the *do_hardening* function which receives from the PM (Process Manager) the parameters to enable or disable hardening. Details on these parameters will be presented in the user library section. The *do_hardening* function will enable or disable hardening in the micro-kernel and send a message to the VM. The VM will do the same with the function *do_hardening* present in the VM address space.

The hardening manager is also used to start the HEC module with the function *start_dwc*. This function is the entry point of the HEC module.

Table 6.1: Hardening state variables

Variables names	Meaning	Default values
h_enable	to tell that a hardening execution of a PE is in progress	0 (means no)
h_proc_nr	the id of the current PE process	0 (no process)
h_step	DWC step (1 or 2)	0
h_restore	Between two runs the PE state should be restored	0 (means no)
h_unstable_state	A fault is detected and the correction is in progress	0
h_can_start_hardening	Hardening is enabled	0 (means no)

6.3 HEC

6.3.1 Requirements

The DWC module is the central module of the hardening software. It provides the following services:

- Saving initial state
- Starting the first run
- Restoring PE context and memory to initial state
- Starting the second run
- Comparing the results
- Restarting the double execution in case of nonconformity of the results.

6.3.2 Starting the first run

6.3.2.1 Requirements

The *restore_user_context* function is the transition point from kernel mode to user mode. This is a point of no return for the kernel. It is in this function that the context of the user process is restored to allow it to run. This function is therefore the exit point from the kernel to the process that will start its execution in a hardened way. So the function *start_dwc* is called from there. Specifically to start the first run the following requirements are taken:

- The initial state should be saved. That means, the process key information should be stored in the protected memory
- The hardening step should be initialized at FIRST step
- Write access to process data should be restricted. That means all the process's pages accessible in writing must be marked read-only. That is done by the PRAM module.
- The instruction retirement counter should be initialized. It must be configured to count and to overflow and send an exception after the instruction number specified in the previous chapter i.e. 1260848 instructions in order to respect the principle of the scarcity of SEUs.

6.3.2.2 Implementation details

In the function *start_dwc* the hardening is enabled for the process configured to be executed in hardened manner. (See listing A.3.2). Hardening is activated through the global variable *h_enable*. If *h_enable* is 0 no hardening is in progress. If it is 1 a hardening is in progress. So at the entry of this function (*start_dwc*), *h_enable* is checked to know if a hardening is in progress. Suppose *h_enable* is 0 so there is no hardening in progress. The criteria necessary for a process to be executed in a hardened way are then checked. Two criteria are that the process to be hardened is not the memory manager (VM) and has the bit $PROC\_TO\_HARD$ in its hardening flag set.

If the process conforms to the criteria. Be sure that there is no active hardening step. The hardening steps are found in the global variable *h_step*. If this variable is 0 there is no active hardening step. When it is $FIRST\_RUN$, the first execution step is active. When

it is $SECOND\_RUN$, the second execution step is active. When it is VM_RUN, the intermediate step that allows the memory manager (VM) to execute during hardening is active. Note that, at this point, only a SEU hitting the kernel could result is h_step to be non-zero.

Suppose there is no active hardening step. The identification number of the process is memorized. That identification is saved in the global variable h_proc_nr. This number will be used later to find the process being hardened. Then the hardening step h_step is changed from 0 to $FIRST\_RUN$. The first step of the PE is started. This is done with the function *update_step*. Then the PRAM module is called to restrict write access to the PE memory data according to the PRAM policy. This is done through the function *set_pe_mem_to_ro*.

Then the initial context of the process is saved. This will be useful to restart execution of the PE in case of comparison failure. That was explained in the previous chapter. This is done with the function *save_copy_0*.

Then the instruction retirement counter is initialized so that it can count the instructions and trigger an exception in case the threshold instruction number is reached. This is done through the *set_remain_ins_counter_value_0* function. The functionalities of this function will be described in more detail later.

And finally the hardening is enabled. The variable h_enable becomes 1. From this moment the user process can begin its execution. That's what happens, its context is restored and it starts to run.

6.3.3 Starting the second run

6.3.3.1 Requirements

Starting for the second run is a little different from starting for the first run. Before starting for the second run the hardening is already activated. The event that causes the end of the first run informs that the second run may begin. Specifically, the following tasks are performed:

- Verification of key information of the PE
- Backup of the $lus1\_us2$ list. The working set list is the set of pages that have been modified during this PE or the preceding one. The $lus1\_us2$ list is the set of (us1,us2) memory frames address allocated to the PE for hardening purpose to be used during first and second run. It contains the addresses of the frames belonging to US0, US1 and US2. The frames of US0 are the frames

of the working set list that belong to the protected memory. They are therefore not modified by the PE during any of the two executions. The frames of US1 and US2 respectively constitute the set of frames modified by the PE during the first and the second execution.

- Restoring the PE register and state variables as they were at the beginning of the first EP run.
- Write access is restricted to the process. That means all these pages accessible in writing are marked read-only.

6.3.3.2 Implementation details

When starting the second execution of the PE, the hardening is already active so *h_enable* is 1. The variable *h_restore* has also been enabled to allow the restoration of the state of the PE as it was at the beginning of the first run. The variable *h_restore* is enabled when the event that ends the first run was handled.

A check is made to ensure that the process to be executed is the current PE. Then the state of the PE is restored to what is was at the beginning of the first run. The pointer to the working set list is copied in a local variable. The size of the working set list is also copied in a local variable. The state at the beginning of the first run will be reused for the second execution.

Then the state at the beginning of the first run is restored using the *restore_copy_0* function The context (ie the values of the registers) and the process's kernel state variables are taken into account during the restoration.

Then the pointer to the working set list and the size of the working set list are restored from the local variables. The function *set_pe_mem_to_ro* is then called to make the PE's pages inaccessible for writing.

Thus the same page faults will be generated and they will be handled by the kernel like during the first run.

6.3.4 Stopping a PE

6.3.4.1 Requirements

The previous chapter defined the events that could be considered as the frontier of PE. It should be ensured that when the process is interrupted

by one of these events, it should not change any state within the micro-kernel before the final commitment of the PE. The various points of entry into the micro-kernel have been identified and the execution flow at these different points has been modified to satisfy this requirement. These entry points are the interfaces between the Minix 3 microkernel and the HEC module. The next section will describe these entry points.

6.3.4.2 Minix 3 entry points

In order to stop a PE in the right way, it was necessary to locate the locations in Minix3 where interrupts, exceptions, traps and faults are taken into account. The *mpx.s* file contains all of these entry points. This file contains all the handlers of traps, interrupts, exceptions, and faults that occur within the Minix3 operating system. As it has been described in Chapter 5 of our thesis a number of events can stop a PE.

In Minix 3 the implementation of the system call is made with the SYSENTER software interrupt and four other software interrupts are defined on the vectors 32 to 35 namely

- IPC_VECTOR_ORIG
- IPC_VECTOR_UM
- $KERN\_CALL\_VECTOR\_ORIG$ and
- $KERN\_CALL\_VECTOR\_UM$

The entry point from the hardware to the software identified in this *mpx.s* file for each of these vectors are

- $ipc\_entry\_sysenter$
- $ipc\_entry\_softint\_orig$
- $ipc\_entry\_softint\_um$
- $kernel\_call\_entry\_orig$ and
- $kernel\_call\_entry\_um$

So when the user process initiates a system call or a kernel call, it is from one of these addresses that the execution continues in the kernel to later execute the appropriate handler for the system call whose query type and the parameters are in the registers initialized in user space

before the software interrupt instruction. The reader will be able to refer to chapter 4 and 3 for more explanation about these events in the x86 architecture and in Minix 3. When the CPU traps at one of these addresses the normal execution of Minix3 leads to either execution of the *do_ipc* function or execution of the *kernel_call* function. Each of these functions handles system calls or kernel calls respectively.

6.3.4.3 DWC and Minix 3

To perform the double execution, this execution flow has been modified. The following objectives have been pursued. As soon as the trap occurs, the DWC mechanism will have to:

- Be able to know if it is time to stop the execution of the PE or not
- Know the current stage of DWC
- Be able to restart the execution of the PE without any changes made within the kernel to handle this event.
- Secondly, this diversion of the flow of execution should not influence the normal handling of the system call or the kernel call. For example, the handling of the system call or the kernel call should not be done twice.

To achieve these objectives, firstly, code for checking the hardened execution steps has been added. If it is step 1 or step 2, the flow of execution is diverted to the *hardening_task_entry* entry point. As soon as the execution flow arrives at this point, the function C (*hardening_task*) is also called. The features of this function are detailed below.

Secondly, the registers necessary to execute the kernel call or the system call are saved before the execution flow deviation. The values of these registers are used during the comparison phase to ensure that both run produces the same states. These values are also restored to the registers before continuing to handle the system call or kernel call at the end of the hardened processing of the PE. Indeed, calling the *hardening_task_entry* function may have changed the values of these registers.

6.3.5 Restarting PE between the two runs : *hardening_task*

6.3.5.1 Requirements

The PE may have to be restarted between the two runs, or when the comparison fails, or when the retirement counter has counted different values during the first and second runs. The processing element must be restarted. Therefore, the PE's memory should be restored as well as its context. The HEC module uses the PRAM module to restore PE memory.

6.3.5.2 Implementation details

Security checks at the beginning of the function *hardening_task*. At the entry of this function (*hardening_task*) the hardening state is verified. Then the running process is read in p. A check is made to ensure that the running process is the current hardened process. Also this function should be called either at the first or the second execution step of the PE. Then the *save_context* function is called.

The function *save_context* as its name suggests, saves the PE context. This backup will be used during the comparison phase. Standard and accessible registers are saved in global variables. Each register has two representatives, one for each run stage. It is the same for the state variables related to PE within the kernel. So depending on whether it is the first or the second execution, the corresponding backup function is called (*save_copy_1* and *save_copy_2*).

Then if it is the first execution, the global variable h_restore is modified to *RESTORE_FOR_SECOND_RUN* in order to remember that the PE will have to be restored to its state before the beginning of hardening at its next execution. This takes us to the second part of the *restore_user_context* function that we explained earlier. The reader should refer for further understandings.

Then the function *vm_reset_pram* of the PRAM module is called. In the case of the first execution, it allows the mapping of each US0 frame to each page in the kernel's working set list. As soon as the mapping is done, the function *vm_reset_pram* returns. And the function *run_proc_2* is responsible for starting the second run. There is no return to the calling point of the *hardening_task* function. In the meantime, there will be a pass through the *restore_user_context* function and the change of state of the global variable *h_restore* will enable

the restoration to the state before the start of the PE as explained above.

But if this is the second PE run, the comparison stage starts.

6.3.6 Comparison stage: *hardening_task*

6.3.6.1 Requirements

The registers produced during the two runs must be compared. Pages modified during the two runs must also be compared. When all compared elements are equal the PRAM module copy the modified US2 frames contents to their corresponding US0 frames. And finally the HEC module hands over to the Minix 3 micro-kernel (the latter will process either a system call, a kernel call, an exception, a scheduling). Otherwise the HEC module restarts the PE.

6.3.6.2 Implementation details

Then a comparison of the registers produced during each run (first run and second run) is performed. (See listing A.3.2). That is the comparison phase. If the comparison is successful, ie each pair of two-by-two register have the same value. The pairs of contents of $us1$ frame and $us2$ frame of each page where a page fault occurred are compared.

The execution goes to the restoration stage, if the comparison of the registers or the comparison of the pairs of contents of $us1$ frame and $us2$ frame of each page where a page fault occurred does not proceed well, that is to say that at least one pair of registers does not have the same value or one of the pairs of contents of $us1$ frame and $us2$ frame of each page where a page fault occurred are not the same. If such comparison failure occurs, the state is restored to the the initial state of the hardened PE.

However, if each comparison succeeds, that is to say that the contents are exactly the same, we proceed to the reset of each hardening variable. Each frame $us0$ is mapped to each corresponding page in the working set list. All hardening state variables are reset. This means that the hardening of the PE is complete. The system can proceed with the nest PE of the hardened process or choose another process to harden. It may be the same process as previously chosen or another process. This decision is made by the native scheduler of Minix 3 that we have not modified.

6.3.7 Restoration stage: *hardening_task*

6.3.7.1 Requirements

One of the features of the HEC module is to restore the PE from the current state (context plus memory) to state saved in protected memory (context plus memory). This restoration is needed between two runs before starting the second run, and when the comparison has failed (restore to the initial state). A restoration of a saved state is also needed when the execution of the first run must be done in single stepping mode.

6.3.7.2 Implementation details

When a fault is detected, the variable $h\_restore$ takes the value $RESTORE\_FOR\_FIRST\_RUN$ to indicate that the initial state of the PE must be restored before restarting the PE.

The variable $h\_unstable\_state$ is set to $H\_UNSTABLE$. That means the system is in an unstable state. The exception handler, the system call handler and the kernel call handler will be informed to not spread the unusual events to the other parts of the system. They will just return. Thus the hardening code will handle that fault by restarting the PE.

Each page in the working set list is mapped to the corresponding frame of US0. So the memory of the PE is restored to US0. The process is prevented from running until the point where it is restarted is reached. The PE is not restarted immediately like during the restoration of the first run. Thus the handlers (exception, system call and kernel call) will have time to be informed to not spread the unstable state to the others parts of the system.

Then the PE's registers and all initial variable of the PE will be restored and the PE will be restarted when the execution switches from the kernel mode to the user mode. The two runs will be executed. The DWC will be proceeded normally.

6.4 Protected memory(PRAM)

Protected memory may not be modified during any of the two executions of the PE. When the process is started all its pages are linked to the frames of US0. At this time, all the pages of the process are set to read-only. Besides, all the memory of the computer is assumed to be efficiently protected by ECC against direct SEU effects. US0 is thus at

that time protected against all modifications by the process itself and by SEU.

The protected memory requires the PE not to be able to modify its memory directly. During execution of the PE, if it tries to modify its memory, new frames must be allocated to it and the contents of the protected frames copied into the newly allocated ones. This can be implemented in two ways:

- to pre-allocate the frames before starting hardened PE execution
- or allocate frames by copy-on-write.

The copy on write is used a lot in the field of file system and memory management usually for sharing pages between several tasks as long as they do not need to modify them for their own and only use. [Pet02, Chu96]. The principle is to copy the data in a new frame when the task wishes to modify it. So a private copy is created for the task.

This is the case for example when creating a new process with the fork system call. The principle requires the memory of the parent process to be duplicated in order to create the memory of the child process. A more efficient way to implement the fork using the copy-on-write principle is to map the pages of the parent process and the child process to the frames belonging to the parent process. However, care must be taken to make all these pages read-only. Thus the father and child could continue to run until one of them tries to modify data, in this case for the page or pages concerned one or more frames are allocated to the process that tries to write (either to the father or the child).

Experience has shown that in Minix3, copy-on-write allocation is very expensive in performance. That needs multiple context switches to allow the VM to run. It was preferable to choose the pre-allocation. For this purpose, before the start of a PE, the $VMPRAM$ module in the VM allocates for each process data page a pair of frames (us1, us2).

6.4.1 Pre-allocation of US1 and US2 : Building PE $lus1\_us2$ list

6.4.1.1 Requirements

US1 and US2 frames are essential for the PE during the first and second run. These frames are used by the PE when it modifies its data. These frames can be allocated during PE execution (via copy-on-write) or before PE execution (when the VM allocates the US0 frame to the PE

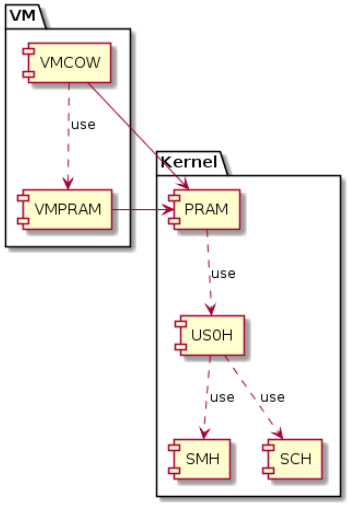

Figure 6.2: Protected RAM architecture

process). Allocation during PE execution is handled by the VMCOW module in the VM and preallocation is handled by the VMPRAM module in the VM. Each PE has a list containing the addresses of US1 and US2 stored in the micro-kernel and in the VM.

The VMPRAM module in the VM is used whenever the VM changes the page table of a hardened process. The VM changes the page table of a process when one of the following events occurs:

- Creation of a new process, so a new page table is created for the process
- The hardened process makes an exec system call
- A page fault occurred on a writable page by the process. In this case a new frame is allocated to it.
- The running process asks for memory allocation

The purpose of this module is to ensure that each page present in the page table and accessible for writing by the PE has the corresponding frames in US1 and US2.

6.4.1.2 Pre-allocation of US1 and US2 in the VM

The us1 and us2 frames allocation is activated in the $pt\_writemap$ function. This is the Minix3 function for modifying the page table in the VM. If the written entry has the write bit set, that means the VM has just given write access to the hardened process. A call is made to the function $tell\_kernel\_for\_us1\_us2$. As the name suggests this function will allocate the frames us1 and us2 and pass their addresses to the micro-kernel.

At the beginning of the $tell\_kernel\_for\_us1\_us2$ function, a check is made whether the page already exist in the $lus1\_us2$ list of the process. If no, new frames us1 and us2 are allocated and associated with the page. Otherwise, that means that us0 was released, just update the us0. The quadruplet (v, us0, us1, us2) is then sent to the micro-kernel by the $sys\_addregions\_to\_ws$ kernel call.

6.4.1.3 Pre-allocation of US1 and US2 in the micro-kernel

In the kernel level, the $add\_regions\_to\_ws$ function (In the PRAM module) retrieves the quadruplet (v, us0, us1, us2). With v as index a quick check is made to see if the page already exists in the $lus1\_us2$ list. If so, us0 is updated. Otherwise a new data structure is created to store the quadruplet. Copying data from us0 to us1 and us2 is not done immediately. Because it is possible that after modifying the page table a server process could copy data in the new pages for the process. A flag is enabled to allow the actual copy to be done later when the HEC module will prepare the execution of the PE.

6.4.2 Copy-on-write allocation of US1 and US2

Before starting the first run or the second run of the PE, all the pages pointing to US0 are set to read-only. All the page pointing to US1 (resp. US2) are also set to RO except those that were modified during the previous PE (in order to identify by page faults which frames are modified by the PE and must be compared for DWC at the end of the PE). Setting in this way pages to read-only will cause page faults. These page faults caused by the hardening are called "caused page faults".

6.4.2.1 Handling caused page faults in kernel

The function *check_vaddr_2* of PRAM module handles the copy-on-write page faults. This function can be called only when one of the hardening steps are enabled:

- $FIRST\_RUN$;
- $SECOND\_RUN$ or VM_RUN.

Hence the verification is done at the beginning of the function.

Then, the entry in the page directory (pde) and the entry in the page table (pte) are computed from the virtual address. The virtual address is also stored in global variables $pagefault\_addr\_1$ and $pagefault\_addr\_2$. These are respectively the page fault address during the first and the second run. That is used during the comparison step.

Using pde as index and the root address of the process page table, the value of the page directory entry is read. This value contains the address of the page table of this directory, and also flags giving the state of this entry. These flags are checked.

If the page directory is not writable ($I386\_VM\_WRITE$) or is not an accessible directory in USER mode ($I386\_VM\_USER$) or is not present ($I386\_VM\_PRESENT$) or is a directory containing global pages ($I386\_VM\_GLOBAL$) or a directory containing big pages (4MB, $I386\_VM\_BIGPAGE$), the page fault is managed by the VM without of intervention of the hardening software. It's a normal page fault. A directory of global pages is used by the kernel for shared code and data (shared libraries, shared data, etc.). Big pages are pages over 4K. Currently, to my knowledge, Minix3, only supports 4K pages.

Then the address of the page table is read. This address is extracted from the value of the page directory entry read previously. With this address as the base and the *pte* as offset, the value of the entry of the page table is read. Two checks are made, the presence of the page. If this check is not verified, the page fault is treated as a normal page fault by the VM. That will end the PE.

The address of the frame (pfa) is extracted from the value read. If this address is null, the page fault is processed by the VM. Then a search is made in the block list of the $lus1\_us2$ list using the function *look_up_pte*. Each block of this structure consists of the data structure, struct *pram_mem_block*.

- The id used to reference the block in the list.

- The variable vaddr : This is the base virtual address of the page.
- The variable $us0$ contains the address of the current frame of the page in US0. This is the address of the frame as known by the rest of the system. In the case of Minix3, it is the VM that knows this address in its data structures. In the case of a serious event, it is from the data of this frame that the data of the page is restored.
- The $us1$ variable contains the address of the frame that stores the results of the first run. $us1$ belongs to US1.
- And finally the variable $us2$ contains the address of the frame that stores the results of the second execution. $us2$ belongs to US2.

When the $look\_up\_pte$ function returns a null element, that means the page fault address does not belong US0. Then the page fault is a normal page fault handled by the VM. Because the $lus1\_us2$ list may not change in any way during hardening, that will end the PE.

Then ensure that the $us0$ attribute of the block returned by the working set matches the address of the frame read using the page table entry (pfa). Make sure that the $us0$ and vaddr attributes of this block are non-null.

Then we mark this block as a block on which a page fault caused by hardening has occurred i.e that the page has been set read-only by the function $vm\_setpt\_root\_to\_ro$, and a page fault occurred on that page during that current PE. This makes it possible to reduce the number of frames to be compared during the comparison step. Only blocks that have this bit enabled in their flags will be processed.

Update the value of h_rw described above.

The kernel can handle the page fault without informing the VM if the $us1$ and $us2$ frames have already been allocated for this page. Otherwise the VM is informed to do this allocation. For this reason, in case the VM must be informed, the bit $PRAM\_LAST\_PF$ is enabled for this block representing the page so that, as soon as the VM makes the allocation, the kernel can update the attributes of the block at its level ($us1$, $us2$ and $us0$).

To avoid having more than one data structure that manages the same page in the working set list, the function $look\_up\_unique\_pte$ allows us to make sure of this. This should not happen except after a SEU touching the micro-kernel.

6.4.2.2 Principles of handling caused page fault in the VM

In the Minix 3 operating system the memory is managed entirely by a server process: the VM. When the system starts, the bootloader informs the kernel of available memory. The latter takes the amount of memory it needs and sends a message to the VM containing the sizes and the base of the available memory blocks. All memory requests are forwarded by the micro-kernel to the VM. There is no dynamic allocation of memory in the micro-kernel. All micro-kernel memory is allocated statically. It is therefore not possible to implement copy-on-write without involving the VM. The VM has to play three roles for the hardening:

- Allocate the frames for US1 and US2 and free them when their corresponding US0 frame is freed.

- If copy-on-write is used ask the kernel to copy the contents of the frame of US0 respectively in US1 and US2 for the second execution, when these frames do not yet exist.

- If copy-on-write is not used the VM just allocates frames for US1 and US2 before starting the PE execution and and frees them when their corresponding US0 frame is freed.

For this the VM maintains a data structure similar to the $lus1\_us2$ list maintained within the micro-kernel. A way has been found to keep these two lists synchronized.

Moreover it is also within the VM that the big changes in the memory space of the process are made. For example, through the mmap, mumap, brk system calls, the memory space of the process may change without the micro-kernel being informed of that change. The $lus1\_us2$ list being maintained from one PE to another, it was necessary to find a way to inform the micro-kernel of each change. The simplest decision was to release the $lus1\_us2$ list each time one of these events occurs and recreate it after needs. It is not very effective. More effective approaches will be discussed in the section 6.4.3 for each of these cases (allocation and free).

There are also other events such as exec system calls that are handled in a similar way except that the kernel is by default informed in this case precisely.

6.4.2.3 Handling messages from micro-kernel

The `main` function of the VM, main.c file was modified. In this function the VM waits to receive a message from the other system processes or from the micro-kernel. As soon as a message is received the VM verifies the source and type in order to call the appropriate routine to handle the request. The VM was modified to take into account the three new message types described above $VM\_HR1PAGEFAULT$ (the faulty page is read-only and the page fault occurred during the first run of the PE) $VM\_HR2PAGEFAULT$ (the faulty page is read-only and the page fault occurred during the second PE run).

If the concerned pages fault is caused by hardening (that means the message type is $VM\_HR1PAGEFAULT$ or $VM\_HR2PAGEFAULT$) we have written a special manager that deals with the treatment of that page fault, $do\_hpagefaults$.

6.4.2.4 Handling caused page faults in VM : $do\_hpagefaults$

The $do\_hpagefaults$ function is a bit like $do\_pagefaults$ but without the unnecessary extra function call to handle the page fault, the Minix3's native page fault handler. The reason to write a new function is to dissociate the handling of page faults caused by the hardening software from normal page faults of Minix 3.

First the process accessing to the faulty page is identified. In $hmap\_pf$ function three frames are allocated for protected memory. Finally the micro-kernel is informed to update the page's attributes within its memory space.

Then one makes sure that the page corresponds to a valid memory area of the process. This makes it possible to recover page faults due to poorly initialized pointers. However in the context of hardening this can only happen if the program is poorly written. It allows to locate the virtual region of the page. This information is sent to the $hmap\_pf$ function.

The purpose of the $hmap\_pf$ function is to extract the physical block corresponding to the page and to make sure that the attempt of the process is legitimate. That is, the process has the right to write to this page. It is after all these checks that the memory allocation function is called.

6.4.2.5 Copy-on-write allocation of US1 and US2 in VM

The function *allocate_mem_4_hardening* in the VMPRAM module handles the VM part of the copy-on-write. This function takes as parameter the process, the virtual region, the physical region and and a parameter to signal whether it is called during the first or second run.

One checks if the block exists in the working set maintained by the VM. It should be noted that in case of copy-on-write allocation unlike the micro-kernel the VM $lus1\_us2$ list is built when the VM processes page faults. So when the search is unsuccessful a new block is allocated and added to the list of the working set of this process within the VM.

Before the block is added one frame was requested from free memory. Then the VM updates the block's attributes within its memory space according to the hardening step (FIRST or SECOND run). The contents of $us0$ are copied into each of the frame to ensure that the US1 (resp US2) frame have the same data at this stage of hardening. The kernel is aware of this through the kernel call *sys_hmem_map* which actually trigger *do_hmem_map* service within the kernel.

In case the block is already present, if $us1$ or $us2$ is null a new frame is requested. The value of $us1$ or $us2$ is changed accordingly.

The content of the current frame is copied to $us1$ (resp in $us2$) depending on whether you are in the first or second execution. Then finally the micro-kernel is informed for synchronization from the *sys_hmmap* system call.

6.4.2.6 Copy-on-write allocation of US1 and US2 in micro-kernel

The *do_hmem_map* function in the PRAM module handles the allocation of US1 and US2 by copy-on-write part in the kernel. This micro-kernel function closes the page fault management loop as part of the hardening process. The page faults hardening work is split between the kernel and the VM. After finishing its job the VM has passed the addresses of the frames $us1$ (resp. $us2$) to the micro-kernel.

This function, called when the *sys_hmem_map* system call is activated, searches for the corresponding block in $lus1\_us2$ list of the micro-kernel. We have already said in the previous pages that the PRAM_LAST_PF bit was enabled at the block level to remember the page concerned by the page fault. So the search is performed based on that bit. The block having this bit enabled is thus returned. The service

therefore updates the values of the block's $us1$, (resp. $us2$) attributes. The services also maps the page to $us1$, (resp. $us2$).

At the end of the execution of this step the kernel and the VM have the same information on the page in which the page fault occurred. This process will be repeated as long as the PE makes page faults.

6.4.3 Changes in process memory space

During the execution of the process its memory space may change. This can be fatal to the process if the $lus1\_us2$ list is not updated in accordance with these changes performed by the VM. Normally the process keeps the working set list until its exit. If its memory space does not change US1, US2 and US0 will remain unchanged. The changes that would be the most fatal if the micro-kernel ignored them are the replacement of one region by another bigger region with change of physical addresses, while the old virtual addresses range is included in the newer address range. This can lead to inconsistencies in protected memory and in the DWC mechanism.

The $lus1\_us2$ list is built when the VM sets the page table of the process.

The process memory may also grow when the process requests a memory allocation. Process memory can be shrunk when the process frees memory. There are also events such as exec, fork, and exit that change the memory image of the process. In the case of fork, the data pages of the parent process are set to read-only for copy-on-write purpose. In the case of exec a new memory image that is allocated to the process.

These events therefore influence the PE $lus1\_us2$ list.

6.4.3.1 Memory allocation to the process

When the process memory is increased VMPRAM module in the VM allocates the corresponding US1 and US2 frames and sends their addresses to the micro-kernel. That was explained in sections 6.4.1.2 and 6.4.1.3.

6.4.3.2 Freeing process memory

The process memory can be shrunk when the PE does a free system call or when the process memory should be changed by new image during an exec system call. When the process memory is shrunk, the VMPRAM

module in the VM frees the corresponding US1 and US2, frees the data structure storing the addresses of US1 and US2 and informs the micro-kernel so that the latter can do the same.

Freeing memory from a process memory space is done in the VM using the function $map\_subfree$ of Minix 3. When the freeing of memory concerns a hardened process, the $free\_region\_pmbs$ function of the VMPRAM module is called. This function frees the data structures of concerned pages in the $lus1\_us2$ list of the VM, the us1 and us2 frames of each concerned page are also freed using the VMPRAM module function $free\_pram\_mem\_block$. Then the function informs the micro-kernel via the kernel call sys_free_pmbs, to do the same.

The micro-kernel handles the request with the do_free_pmbs kernel call service. This service uses the $free\_pram\_mem\_block\_vaddr$ function of the PRAM module which frees the $pram\_mem\_block$ data structure from the $lus1\_us2$ list of the micro-kernel for each page concerned by the memory freeing using the $free\_pram\_mem\_block$ function of the PRAM module.

6.4.3.3 Fork

In the case of a fork system call, the process memory changes. The data pages remain the same, but they are all set to read-only. At the next PE of this process, the memory state would not be consistent with the protected memory policy. Minix 3 set the memory of the process to read-only for copy-on-write purpose.

During the fork system call handling in the kernel, a flag is enabled in the data structures representing the protected memory within the kernel using the set_fork_label function for each process data page. This flag simply allows these pages to be considered as pages that should be processed by the VM in case a page fault occurs on them. The PRAM policy is not applied to them as long as that flag is enabled. So as soon as a page fault occurs on these pages the VM is directly informed. The native Minix 3 handler processes the page fault. Because it is copy-on-write that is made on this page, the VM may allocate a new frame to this page and therefore modify the page table. As soon as the page table is modified the function $tell\_kernel\_for\_us1\_us2$ of the VMPRAM module updates the value of the US0 address within the VM and informs the kernel which does the same as explained in the sections 6.4.1.2 and 6.4.1.3.

6.4.3.4 Freeing US1 and US2

In the micro-kernel, freeing the working set list $lus1\_us2$, consists in removing each struct $pram\_mem\_block$ * from the working set list, starting from the top of the list until there is nothing left in the list. Each block is marked as free, so it can be used by others PEs. The attributes of each block are set to MAP_NONE ($us0$, $us1$, $us2$). In the VM the same operations are performed. In addition the frames $us1$ and $us2$ are released. They are returned to the free memory blocks list managed by the VM. So they can be reused by other processes. Thus the blocks allocated during the copy-on-write are not definitively held by the system. They are freed as soon as the process releases its working set list. This shows an economical use of memory during the hardening mechanism.

6.4.4 Restricting write access to US0 frames

6.4.4.1 Requirements

One of PRAM's module features is to prevent PE write access to US0 frames and to grant write access to US1 and US2 frames depending on the nature of the current PE run. PRAM module uses the USOH module for updating the contents of US1 and US2 when the contents of the corresponding US0s have been modified during a system call or by another process sharing the same US0 with the PE.

1. Before starting the first RUN, the content of US1 and US2 frames are updated if the corresponding frame of US0 has been modified.

2. Before each run, write access is denied to some pages of the process address space. At the beginning of the first PE, all the pages accessible in writing are put in read-only. At the beginning of other PEs when frames correspondents of US0 exist in US1 and US2, the page is set to read-only if it has not been modified during the previous PE, nor during the previous system call or kernel call nor by another process sharing the same pages with the process to harden. Indeed pages recently modified will probably be reused and they are directly added to the list of the pages to compare. Pages not recently used are added to the list only if their DIRTY bit is set during the PE.

6.4.4.2 Implementation: functions *set_pe_mem_to_ro* and *vm_setpt_to_ro*

The function *vm_setpt_to_ro* is used when copy-on-write is used for protected memory implementation. The function *set_pe_mem_to_ro* is used when pre-allocation is used for protected memory allocation. The core of these two function is the same. But the *set_pe_mem_to_ro* is more efficient than *vm_setpt_to_ro*. So the pre-allocation was used in the current implementation.

At the beginning when it is the first run of the PE, the function *handle_hme_events* of US0H module is called to update the US1 and US2 frames, if their corresponding in US0 has been modified. Details on handling the US0 frame changes out of hardening can be found in the section 6.4.5.

In a loop the list *lus*1*_us*2 is browsed. For each *pram_mem_block* from the list:

- The physical address of the frame is extracted from the page table entry
- If the page has been put in the list recently, the contents of us0 are copied to us1 and us2
- If the process is sharing a memory region with other processes, the corresponding page in the working set are marked shared. So when the process modifies contents in that page, the other processes sharing the same page will be notified. So they will update their corresponding frames in US1 and US2.
- If the page was modified during the previous PE or during handling of a system call or a kernel call or by another process sharing the page, the page is set read write. Otherwise it is set read-only.

At the exit of this function (*set_pe_mem_to_ro*) all pages accessible in writing from the PE are read-only and the working set is either built from scratch or is updated.

6.4.5 US0 content change: US0H modules

Frames contained in US1 and US2 are used during the DWC for the first and second run of the PE. They are unknown to the rest of the Minix system. While the frames of US0 are the frames that all of Minix knows to have been allocated to the process by the VM

Between PEs, there are the handling of system calls, and the execution of PEs of others hardened processes. The module US0H handles change in US0 frames (See listing A.3.6)

One of PRAM's module features is to prevent PE write access to US0 frames and to grant write access to US1 and US2 frames depending on the nature of the current PE run. PRAM module uses the USOH module for updating the contents of US1 and US2 when the contents of the corresponding US0s have been modified during a system call or by another process sharing the same US0 with the PE. This can happen:

- during the handling of a system call or a kernel call. That means when the system call or the kernel call was processed, data have been copied in the process's address space. In Minix3, that is done via the function *virtual_copy_f*.
- or when the kernel has copied a message in the process address space.
- or when another process sharing pages with the process to harden has modified the shared memory.

6.4.5.1 Tracking US0 content change

The US0H module provides the *enable_hme_event_in_procs* function that monitors the change in the US0 frames of each hardened process. As soon as the page is modified during the processing of a system call or by another process, this function enables a flag. This flag will be used to update the contents of US1 and US2 frames.

6.4.5.2 Copying US0 content to US1 and US2

Each hardened process has a list of zones changed during the processing of a system call, a kernel call or a modification of a shared memory. The *look_up_hme* function checks if an area of memory starting with the same address does not exist yet in the process's hme list. If it does not exist, a new structure is allocated and initialized based on the information received. However, if a *hardening_mem_event* data structure already exists for a zone starting at this address, a check is made to see whether the size of the new zone is larger than the size of the old zone. When it is the case, the old zone is replaced by the new zone. In the opposite case, the new zone is ignored beacause it is contained in

the old zone. This processing avoids having to copy the same data areas multiple times.

Before starting a PE, the *handle_hme_events* function goes through the PE's *p_hardening_mem_events* list and with each *hardening_mem_event* data structure found, it updates the corresponding area of US1 and US2 using the function *update_range_ws_us1_us2_data*.
The *update_range_ws_us1_us2_data* function has a parameter to point to the PE, the start address of the zone and the number of pages concerned. This allows it to update, for each concerned page, the contents at the addresses of the US1 and US2 frames by calling the *update_ws_us1_us2_data_vaddr* function which does the actual update work by making copies of US0 to US1 and US0 to US2.

6.4.6 US0 contents change during system call handling or kernel call handling: SCH module

6.4.6.1 Requirements

Minix 3 is a multi-tasking operating system; there is a strong interaction between different processes through data exchange. These data exchanges can be made by sending messages (request initiation) or by explicit data copy (copy of data from memory to memory, virtual or physical). These data transfers are done directly on the US0 frames by the micro-kernel. It is therefore necessary to find the means to reflect these data changes on the frames of US1 and US2 if they exist.

6.4.6.2 Implementation

At first, it was necessary to locate where US0 is modified during a system call or kernel call handling in the micro-kernel. Analyzing the code allowed to find the functions *delivermsg*() and *virtual_copy_f*(). (See listing A.3.7).The *delivermsg*() function is used to copy a message into the address space of process. A message is a data structure with a maximum size of 56 bytes used in Minix3 for inter-process communication. The *virtual_copy_f*() function is used to copy variable size data blocks from the address space of one process to the address space of another process.

At the end of these two functions a call to the function *add_hme_events* has been added. This function notes that an area of the process memory has been modified by storing the zone's start

address and zone size in the *hardening_mem_event* data structure. This data structure will be processed later by the US0H module.

```
1 struct hardening_mem_event {
2  int id;
3  int flags;
4  vir_bytes addr_base; // modified area base address
5  vir_bytes nbytes; // size
6  int npages; // number of pages
7  struct hardening_mem_event *next_hme;
8 };
```

6.4.7 US0 contents change in shared memory: SMH module

6.4.7.1 Requirements

In Minix 3, a process cannot directly modify the memory contents of another process. All data changes between processes occur normally through system calls or kernel calls. However with the shared memory mechanism, it is possible for two or more processes to share the same memory area. Thus the modifications made by one are directly perceived by the other sharing processes.(See listing A.3.8)

A request to create shared memory goes through the VM that creates the shared memory region and associates it with the memory space of the process. There are different types of memory managed by the VM:

```
1 mem_type_t mem_type_anon,         /* anonymous memory */
2 mem_type_directphys,     /* direct physical mapping memory */
3 mem_type_anon_contig,    /* physically contig anon memory */
4 mem_type_cache,      /* disk cache */
5 mem_type_mappedfile,     /* memory with file contents */
6 mem_type_shared;     /* memory shared by multiple processes */
```

6.4.7.2 Implementation

For shared memories the type is *mem_type_shared*. When the VM maps a shared memory in the address space of a process by calling the *map_page_region* function, a kernel call is made from the VM to the kernel with the parameters following :

- *vm_endpoint*: identifier of the process
- vaddr: start address of the region

- length: the size of the region
- id: the unique identifier of the region

With this information the kernel calls the function *add_hsr* which will add this information to the attributes of the process.
The *hardening_shared_region* data structure is used to store information related to a shared region within the kernel.

```
1 struct hardening_shared_region {
2   int id; // unique identifier of the region known by the
      kernel
3   int flags; // contains the states
4   int r_id; // unique identifier of the region known by the
      kernel and the VM
5   vir_bytes vaddr; // start address of the region
6   vir_bytes length; // size of the region
7   struct hardening_shared_proc * r_hsp; // list of processes
      sharing this region
8   int n_hsp; // the number of processes sharing this region
9   struct hardening_shared_region * next_hsr;
10 };
```

Each process has a *hardening_shared_region* list linked to its data structure. Likewise, each *hardening_shared_region* shared memory data structure has a list of processes sharing that shared memory region. A light data structure of the proc structure containing only the identifier of the process, an id and a flags has been created to represent the process in this list.

```
1 struct hardening_shared_proc{
2   endpoint_t hsp_endpoint;
3   int flags;
4   int id;
5   struct hardening_shared_proc *next_hsp;
6 };
```

To guarantee the unicity of the shared regions and the processes sharing them, two parallel list have been created. The *all_hsr_s* list and the *all_hsp_s* list. The *all_hsr_s* list contains all uniquely shared regions. While the list *all_hsp_s* contains all processes sharing a region within the system.

So when the kernel is informed that a shared region has been added to the memory space of the process, it calls the *add_hsr* function. The role of this function is globally to add the new shared region to the *p_hardening_shared_regions* list of shared regions of the process.

For this, it goes through the list of shared regions of the process using the *look_up_hsr* function looking for a region with the same characteristics as the new region:

- the same base address
- the same size
- and the same region id

If a region is found, no addition will be made. Otherwise, if a region is not found, the *all_hsr_s* list is searched using the *look_up_unique_hsr* function to check if another process does not share a memory region with the same characteristics. If such a region is found it is added to the *p_hardening_shared_regions* list of shared regions of the process and the process is added to the *r_hsp* list of processes sharing that memory region.

Otherwise, a new *hardening_shared_region* data structure is allocated. This new data structure is added to the *all_hsr_s* list and to the p_h*ardening_shared_regions* list of the process. And finally the process is added to the *r_hsp* list of processes sharing this region.

Later at the end of a PE when the comparison is successful, a call to the *vm_reset_pram* function is used to check the modified pages during the PE execution. When one of these pages is a shared page, the *enable_hme_event_in_procs* function enables the *hme_event* in all processes sharing this page.

6.5 Hardening Exception handler

6.5.1 Requirements

The exception mechanism is a way for the hardware to report unhandled hardware events or unusual events to the software. For example, a counter that overflows (the hardware will reset the counter, but the software will be notified) or a request for access to an area of memory that is normally not accessible (the MMU will prevent access, but the software will be notified). When we say that the software will be notified, this is not systematic, in the sense that if the hardware is intended to signal an event, a handler should be provided by the software to handle this event. If no handler is provided the event will just be ignored. A handler is provided if the event is relevant to the proper operation of the software.

For common exceptions such as division by zero, page faults, handlers are provided by default in almost all operating systems. However for MCA exceptions for example, there is no default handler. Because the events of the MCA remain relevant events only for specialized operating systems.

However, whether there is a handler or not, the expected behavior of the software must be consistent with that of the hardware. This means that the handler must be able to correct the condition that led to the exception situation. Otherwise the hardware will persist indefinitely, signaling the exception. For example, the management of a page fault ends either with the authorization of access or with the termination of the faulty task. In the case of a normal execution the exceptions are deterministic when performing a task. This means that if the task is executed N times under the same conditions without external changes, the same exception will be repeated N times. On the other hand, if there is a change in the execution conditions of the task, for example a handler that corrects the state that caused the exception or if the source of the exception does not depend on the program (if it is the consequence of SEU for instance), the exception will not be repeated. In the case of hardening Minix3, the source of an exception may be due to the normal operation of the PE, to the occurrence of a SEU or to the hardening itself (some exceptions due to page faults, see above and exception due to single stepping). The three categories must be handled differently and must thus be identified .

1. If the exception is due to hardening, it is expected and, after processing, the current execution is resumed

2. If the exception is caused by MCA and reports a fault corrected automatically by the hardware, nothing needs to be done and the current execution is resumed.

3. If the exception occurs for the first time in this PE during the second run, it is caused by a SEU. The PE must be aborted.

4. In all others cases, The DWC will be used to detect the origin of the exception (internal to the program or SEU). If the exception comes from a fault external to the program (SEU for example) the comparison phase will result in a failure because the two runs will be stopped at different places: the execution with the exception caused by SEU will be aborted prematurely. Only if the exception

occurs at the same place the normal exception handling of Minix will be activated.

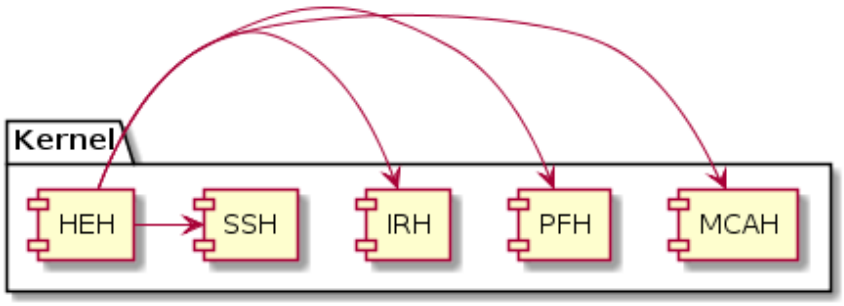

Figure 6.3: Hardening exception handler architecture

HEH (hardening exception handler) is the exception handler. It calls the following modules:

- SSH: Single stepping handler to handle single stepping phase
- IRH : Instruction retirement handler, to handle exception from retirement counter
- PFH: Page fault handler, to handle page fault exception.
- MCAH: MCA Handler, to handle MCA exception.

6.5.2 Implementation

In the implementation of exception processing the processor traps at the $exception\_entry$ or $exception\_entry\_from\_user$ entry point of the Minix3 $mpx.S$ file. All exceptions are trapped to one of these addresses. Exception from user space task traps at $exception\_entry\_from\_user$. As it was done for the system calls, the execution flow is diverted to the $hardening\_exception\_entry$ address which calls the C function $hardening\_exception\_handler$ which will be detailed in the part concerning the exception mechanism as fault detection. However the decision to continue the handling of the exception is taken at return from this function. (See listing A.3.9 and and listing A.3.13)

At the beginning of this function it is verified that hardening is activated. Then it is also verified that the process that is causing the

exception includes the current PE. The h_stop_pe variable is set to H_YES to signal that the decision to interrupt the PE execution stage has already been made. It is the handling of the exception by the hardening module that will decide to not stop the PE.

All the exceptions are not treated in the same way by the hardening module.

- In the case of page fault, the page fault handler (PFH module) is called to handle the exception,
- In the case of MCA exception the MCA / MCE handler (MCAH module) is called to handle the exception,
- In the case of NMI, the retirement counter handler (IRH module) is called to handle the exception,
- In the case of DEBUG exception, the single stepping handler (SSH module) is called to handle the exception.

In the case of all the others exceptions not directly related to hardening, the principle remains the same. which means :

- h_stop_pe remains to H_YES to stop the PE's step
- The event is logged
- Later the comparison phase will reveal whether the exception is normal or due to a fault. In the case of a normal exception the comparison phase will succeed. On the other hand, in the context of an exception due to an external event (a SEU for example) the comparison phase will fail.
- If the exception is a fault, the error will not be propagated throughout the system. The handler will just return without doing anything. Exception management has already been taken into account by the hardening exception manager.

6.5.3 Page fault handler

Both in the first and second run, the hardened process will start its normal execution, at some point of execution it will try to write in its memory. An exception will be triggered. So there will be a trap in the kernel that will result in $exception\_entry$ of the $mpx.S$ assembler file. Assembly code has been added in order to call the C function $hardening\_exception\_handler$.

6.5.3.1 Requirements

In the case of page faults the function *hardening_exception_handler* must be able to:

- Determine if the page fault is a normal page fault or a page fault due to protected memory
- If this is a normal page fault, this function must change the hardening state to stop the execution of the PE. And the page fault will be processed in the standard way by Minix. Otherwise the page fault is handled by the kernel.

The *hardening_exception_handler* function is an exception handling function for hardening. The function *check_vaddr_2* will deal with the details related to the caused pages faults.

6.5.3.2 Exception entry point from hardware to kernel: The *hardening_exception_handler* function: page fault

At the entry point of this function, a check is made to know if hardening is activated. This function can only be called if the hardening is active. Then the pointer to the process structure of the running process is loaded into a variable of type struct *proc, named here p. Through p, it is possible to access all the attributes of the running process as defined in the kernel process table.

A check is then made to find out if the current PE belongs to the running process. Another check is made to make sure that this process is not the VM.

When the exception is a page fault exception, two global variables are reset. These are h_normal_pf and h_rw. The variable h_normal_pf indicates that the page fault being processed is not caused by the hardening itself (because the page has been set to RO and the PE tries to write in it) but a page fault specific to the execution of the process. So if the value of h_normal_pf is $NORMAL\_PF$ then the page fault is normal. But If it is 0 it means that the page fault is caused by the hardening. The variable h_rw informs the VM about the hardening step during the page fault occurs (FIRST or SECOND run) and also it informs Minix3's native page fault handler whether that page fault could be handled by the kernel or not.

There are 3 possible values,

- *RO_PAGE_FIRST_RUN* : means that page fault occurred during the first execution
- *RO_PAGE_SECOND_RUN* : means that page fault occurred during the second execution
- *K_HANDLE_PF* : means that page fault must be handled by the kernel without calling the VM. The page fault is handled by the kernel itself when the frames for US1 and US2 exist. The kernel will just do the mapping between the frame and the page and enable the write bit.

Then the address where the page fault occurred is read. And the function *check_vaddr_2* of the PRAM module is called with a pointer to the current running process (p) data structure and the address of the root page table of this process (*p_seg.p_cr3*). It also takes as parameter the virtual address where the page fault occurred. The function takes also as last parameter the address of the global variable *h_rw*.

On return from this function if the return value is different from OK, *h_normal_pf* becomes *NORMAL_PF*, which means that this page fault is not caused by hardening. In this case the PE is stopped.

When the variable *h_rw* equals to *K_HANDLE_PF* it means that this page fault must be handled by the kernel alone without informing the VM.

6.5.4 Machine check architecture handler

6.5.4.1 Requirements

When an MCA exception is reported the source of the error does not matter. The goal is to know if the PE execution can continue after the error or not. When the execution can continue the PE continues its execution otherwise the PE is aborted and restarted.

6.5.4.2 Implementation details

When an MCA exception occurs, status of the error is read in the *IA32_MCG_STATUS* register. The MCIP bit (2) of this register signals if the exception is MCA exception. When the bit is not set, it assumes that the OS has just received a false exception. In this case the signal to stop the PE is sent to DWC module. (See listing A.3.11)

When it is a MCA exception, the RIPV bit informs if the error was corrected or not by the RAS mechanism of the processor. When the error was already corrected, the execution of the PE can continue normally. Otherwise the signal to stop the PE is sent to the module HEC.

6.5.5 Performance monitoring counters

The $IA32\_PMC1$ architectural performance monitoring counter, found in most Intel processor architectures, was chosen. It allows to count most events including instruction retirement. The selector associated with this counter is $IA\_PERFEVTSEL1$. It is used to configure the counter to count the instructions in user mode or privileged mode. It can be configured to trigger an exception when the counter overflows.

6.5.5.1 Requirements

Many modern processors include Performance Monitoring Counters (PMC) gathering data on performance related events. Supported events differ from model to model [Adv16] [Int16]. The following events are commonly supported:

- Instructions Retired
- Unhalted Core clock Cycles
- Cache misses
- Branch Instruction Retired
- Branch Misses Retired

PMC registers can be configured to monitor various events. They can be initialized to a preset value and incremented each time the monitored event occurs. Each counter is associated with a set of control registers that helps select the counted events, when they are taken into account (in user or kernel mode or both), and other specific parameters related to the event to be counted. Some processors also feature some fixed-function performance counter registers and their associated control registers [Int16].
Furthermore, PMC control registers enable the generation of Performance Monitoring Interrupts (PMI) when the counter overflows. Programming a performance monitoring unit to count instructions retirement can then be implemented as follows by system software, operating with kernel privileges:

1. Select the PMI delivery mode in the APIC (Advanced programmable interrupt controller [Adv16] [Int16]): special interrupt vector, system management interrupt (SMI) or non-maskable interrupt (NMI)
2. Select a PMC and configure its associated control registers to select retirement events.
3. Before launching the counting phase, set the counter with a negative value corresponding to the desired number of instructions to execute.
4. Configure the control register to generate PMI when the counter overflows.
5. Upon return to user mode (privilege level 3), the processor starts executing the user process, incrementing the counter after each instruction until the counter register reaches zero. The PMI will then be delivered through a built-in local APIC, according to the APIC configuration parameters from step (1).

Architectural version of PMC were used. These features are present in all Intel processors since the Pentium, so they are present in Xeons and Atoms processors.

6.5.5.2 Implementation

The strategy adopted consisted (See listing A.3.10) of:

- Configuring the counter to count retirement instructions in user mode and to trigger an exception when the counter overflows
- Let be X the number of instructions the processor will have to execute before the overflow of the counter. The counter is initialized to $Z = (2^{64} - X)$.
- At the beginning of the PE, the counter is initialized Z before the transition from kernel mode to user mode.
- The PE will be interrupted several times, either by page faults exception caused by the hardening, by clock interruptions, or by hardware interruptions. To preserve the integrity of the number of instructions to be executed by the PE before the overflow at each context switch, the contents of the counter is read and stored in an attribute of the process data structure. When the process is restarted, the stored value is written to the counter.

- To avoid having an overlap in the count of the instructions of the PE and the number of instruction of the memory manager, the counter is enabled to count only if the process that will be executed is the current PE. Otherwise, the counter is simply disabled. Because during the hardened execution of a PE only the memory manager or the PE itself are allowed to run

The handler checks the status bit (bit 1) in the MSR register $INTEL\_PERF\_GLOBAL\_STATUS$. When the bit is enabled, that means the counter has overflowed. In this case the handler clears the status bit and resets the counter. This event stops the PE execution. The last value read from the counter is stored in two different variables ($first\_run\_ins$ and $secnd\_run\_ins$) for each run. At the end of both executions, when $first\_run\_ins$ and $secnd\_run\_ins$ are the same, the hardening goes to the comparison step. Otherwise the hardening goes to the single stepping phase.

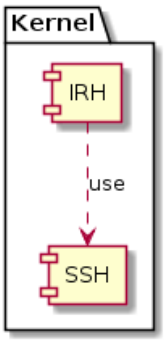

Figure 6.4: Instructions retirement counter architecture

6.5.6 The single stepping handler (SSH)

6.5.6.1 Requirements

The SSH module is called by the IRH module when at the end of the double execution the instruction counter counts different values for the two runs and the context produced by the two runs are differents. The SSH module will have to execute the run that executed fewer instructions (let's call it the late run) in single stepping mode to catch the run that

executed more instruction. The module must configure the processor to execute the PE in single stepping mode and handle the debug exception.

6.5.6.2 Implementation Details

Initializing single steping
The IRH module sends to the SSH module the values of the read int the retirement counter in the exception routine for both runs. With these information the SSH module is able to determine the run to execute in single stepping mode to catch up the other. When it is the second run, there is no need to restore its state. The MF_STEP bit is set to tell Minix3 that this process should be executed in single stepping mode. The variable $h\_unstable\_state$ is set to $H\_STEPPING$ to indicate that the hardening software is in single stepping mode.(See listing A.3.12)

When it is the first execution that must be executed in single stepping, the same configuration is made, the activation of the bit MF_STEP and the update of the variable $h\_unstable\_state$ to $H\_STEPPING$. The PRAM module is called to restore the memory of the PE at the first run (through the call of the function vm_reset_pram). The HEC module is also called to restore the register from R2 to R1 (by calling the $restore\_for\_stepping\_first\_run$ function). The h_step variable is set to $FIRST\_STEPPING$ to indicate that the first run is in single stepping mode. **DEBUG exception handler**

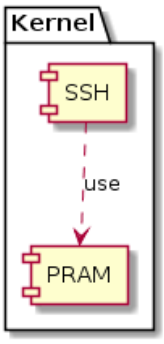

Figure 6.5: Single stepping architecture

The handler changes the $origin\_syscall$ variable to
$PE\_END\_IN\_NMI$ to signal that the event that stopped the PE

is the retirement counter overflow. Because this variable is changed in the HEC module when an exception occurs. The TRACEBIT bit of the PSW register is reset to disable the single stepping mode, the context is saved for comparison by the HEC module. The software instruction counter of the late run is incremented ($first\_run\_ins$ for the first run and $secnd\_run\_ins$). A comparison is made between $first\_run\_in$ and $secnd\_run\_in$. If they are equal the control passes to the HEC module for the comparison phase. If they are not equal the PE is configured to do the single stepping and it is restarted.

6.6 The hardening software and Minix3

6.6.1 The Minix3 native page fault handler

6.6.1.1 Requirements

The Minix 3 page fault manager, like most exception handlers in the Minix3 operating system, has undergone some changes to support hardening. It has been said previously that two methods are used to allow the OS to execute application processes in a hardened way: an innovative use of the exception mechanism present in all architecture and the DWC method. So to allow the exception management mechanism to be a pillar for hardening the following objectives must be achieved regarding the management of page faults:

- Be able to detect page faults caused by hardening
- Be able to not report the same page faults twice. Once from the first run and once from the second run
- Be able to detect page faults caused by hardening but which must be processed by the micro-kernel itself
- Be able to detect page faults caused by hardening but needing to be handled by the memory manager
- Be able to block the spread of errors in the rest of the system from a page fault caused by an external fault, such as a SEU. In other words, as soon as the system is in an unstable state, the page faults must be handled by the exception handler of the hardening which normally will contain the fault within the limits of the PE

victim of the exception. This approach is common in processor architectures with RAS (Reliability, Availability and Serviceability) functionality [BKRF02, Qua00, DeL08].

6.6.1.2 Implementation details

Some changes have been made to the Minix3's native micro-kernel page fault handler so that it can meet the hardening requirements. See listing A.3.13.

After it gets the value of the address where the page fault has occurred, it checks the value of h_rw. If h_rw equals K_HANDLE_PF, it means the kernel has already handled the page fault. The handling has already been done by the kernel in the function $check\_vaddr\_2$ that we have described above. The handling involved enabling the WRITE bit ($I386\_VM\_WRITE$) in the process's page table entry and mapping the page to the corresponding frame in US1 or US2. In this case the page fault handler returns. It has finished his job.

Otherwise if h_rw is equal to one of the values :

- $RO\_PAGE\_FIRST\_RUN$
- or $RO\_PAGE\_SECOND\_RUN$

the type of message to send to the VM is changed.

In this case the VM is allowed to run during hardening. So h_step is VM_RUN and its old value is stored in the global variable h_step_back. Of course the previous value of h_step_back is checked to see if it is zero, h_step, h_step_back are global variables.

The type of message to send to the VM have been changed: one of the following types will be sent to it $VM\_HR1PAGEFAULT$ (the faulty page is read-only and the page fault occurred during the first run of the PE), and $VM\_HR2PAGEFAULT$ (the faulty page is read-only and the page fault occurred during the second run of the PE) based on the values of the variable h_rw as described above.

This message is then sent to the VM.

6.6.2 What happens when, during hardening, the scheduler wants to change process?

6.6.2.1 Requirements

During the execution of a PE of a hardened process, only the kernel, VM or the PE itself can be executed. The kernel can run whenever

needed (activated by system or kernel call, exception or interrupt). The VM can run when ordered by the kernel when there is a page fault it must handle. The PE can run as long as it is executable until it is interrupted by the retirement counter that would send an exception or that it interrupts itself (exception or system call).

6.6.2.2 Implementation

In order to enforce this policy, the function *hpick_proc* is used. See listing A.3.13. At the beginning of this function one verifies that the hardening is enabled. Then we get a pointer to the hardened process from its previously saved id. We choose to retrieve the pointer from the id in case the current process is different from those of the PE (again it should never be the case except if caused by SEU).

Then one checks if the PE is suspended because of a page fault. If the VM is executable, allow it to continue running to allow it to process the page fault. If the VM is not executable, there is a problem, PANIC.

If the PE is not suspended by a page fault. This means that the VM has just finished processing a page fault. Change a transition from the intermediate state to one of the $FIRST\_RUN$ or $SECOND\_RUN$ state is required. First it checks that the PE is executable, if it is not, there is a problem, PANIC. Otherwise we can continue the execution of the PE.

In the case where it is not in the intermediate state and the PE is not executable, it checks if it is because of its quantum which is exhausted, if yes the PE is aborted using the function *abort_pe* and the scheduler can reschedule the process of the PE and choose another process or the PE process to run. Or if it was interrupted by the retirement instruction counter it is renabled if the processing of the exception of the retirement instruction counter is completed and the execution of the PE is continued.

In all other cases, PANIC. There is a problem.

6.6.3 system call, kernel call and interrupt handler

6.6.3.1 Requirements

When a SEU causes a spurious system call, a kernel call, or an exception, double execution can detect and correct the error. However the hardening software does not directly influence the execution of Minix 3. So at the end of the PE processing the system call handling, or kernel call or

exception will handle the event that has been reported by the material. The hardening software must be able to signal to the managers that this event is an erroneous event.

6.6.3.2 Implementation

When such events occur the HEC module sets the variable $h\_unstable\_state$ to $H\_UNSTABLE$. Some lines of code have been added to the system call handler, kernel call handler and interrupt handler, thus if $h\_unstable\_state$ is equal to $H\_UNSTABLE$ they do not handle the event, they just return. listing A.3.13. The HEC module will reset the value of that variable to H_STABLE when the fault will be corrected.

6.6.4 The Minix function *swith_to_user*

6.6.4.1 Requirements

When a fault is detected by exception or when the comparison phase fails, it is possible to restart the double execution immedialty. It has been found that starting it immediatly prevents handling events that have been created by the fault. Another way has been found which consists to:

- Prepare the PE for a restart
- Pass the execution control to Minix handlers. They will abort the events created by the fault.
- And finally restart the PE.

6.6.4.2 Implementation

Switching from kernel mode to user mode after handling a system call, an exception or an interrupt goes through the *switch_to_user* function. A few lines of code were added at the end of this function so that the PE's process is started instead of the process chosen by the Minix 3 scheduler. The HEC module changes the $h\_unstable\_state$ variable to $H\_UNSTABLE$ when the comparison fails. So, in this function, when the value of this variable is $H\_UNSTABLE$ the PE is executed instead of the process chosen by the scheduler. listing A.3.13

6.6.5 Exec, fork clear kernel call

6.6.5.1 Requirements

The memory space of a process issuing exec, fork, clear, kernel calls will be changed. In the case of exec, the process memory space will be replaced by new memory image totally different to the previous. In the case of fork kernel call, the process memory space will be set to read-only for copy-on-write purpose. In the case of clear kernel call, the process memory space was just been released by the VM. To be consistent the $lus1\_us2$ list, should also be modified.

6.6.5.2 Implementation

In the do_exec and do_clear functions of the micro-kernel the function, $free\_pram\_mem\_blocks$ of the PRAM module is called. This function simply frees the list $lus1\_us2$. At the end of this function this list is empty.

In the case of the do_fork function, if hardening is enabled, the hardening parameters of the child process are initialized to allow it to be executed by the hardening software when it is scheduled. listing A.3.13. If the parent process is hardened its $lus1\_us2$ list are labelled to be handled by VM for copy-on-write purpose.

6.6.6 Interrupt handling

An interrupt is non-deterministic, so it cannot end a PE. However an interrupt could have direct effect on the current PE. The possible effects are:

- Change of the state of the process. This means that the process could become blocked. This can be fatal for the hardened execution.
- Copy of data into the PE's stack. This could induce inconsistency in the protected memory.

When an interrupt occurs within Minix3 a notification is sent to all concerned processes. When the source of the interrupt is the clock, (a timer for example has expired) a signal is sent to the process (SIGALRM, SIGPROF). Sending the signal changes the state of the process from running to $RTS\_SIG\_PENDING$. When this happens during hardened execution, that is fatal for the hardened process. In the case of the

others interrupt a notification is sent to the process. Sending a notification could include copying data in the process stack. Doing that during the hardened execution of the process could induce inconsistency in the PRAM

The interrupt is a non-deterministic event, it is impossible to reproduce that event exactly at both executions.

The solution was therefore to delay the signal or the notification until the end of the hardening of the current PE. To do that a new attribute has been added to the *struc proc*. That attributes records non-deterministic events that have happened during the hardened execution. Thus at the end of the hardened execution of the current PE the signals and the pending notifications are sent.

6.7 User library

A user library has been written for using the hardening software. listing A.16. This library provides the following commands:

- hardening <1> to enable hardening for all new process created with fork system call
- hardening <2> to disable hardening for all new process created with fork system call
- hardening <4> <pid> to enable hardening for the process identified by pid
- hardening <8> <pid> to disable hardening for the process identified by pid

6.8 Conclusion

Hardening implementation choices and the exception management mechanism were presented. The exception handling mechanism relies on the HEC for the correction of SEU errors detected by the hardware. Due to a modification of the native exception handling of Minix 3 the faults are contained. The DWC can detect silent faults. Code has been added to Minix 3 to adapt to the hardening of its application programs. In the same way certain functions of the micro-kernel and the VM have been

retouched a little to make them compatible with the hardening software. The other servers were not affected at all. Hardening is therefore transparent to them.

Part III

Results

CHAPTER 7
Results

7.1 Test environment

7.1.1 Qemu

Qemu [Bel05] has been used for the development. Qemu is an emulator. It makes it very easy to emulate many processor architectures. Qemu is very useful for operating system development . It allows to display log information through a virtual serial port. It allows also to debug using gdb. Thus, it is possible with a step-by-step execution to analyze the written code to detect and correct any error [Ass19].

7.1.2 Hardware

The development and the preliminary tests were done in a virtual environment using Qemu. The virtual environment was run on a computer with the characteristics specified in Table 7.1. This computer was acquired for the thesis.

Table 7.1: Hardware for the virtual environment

Characteristics	**Values**
Processor	i7-640LM
CPU Frequency	2.13GHz
Lithography	32nm
Memory	8GB
Cores	2
L3 cache	4 MB 16-way set associative shared cache
L2 cache	2 x 256 KB 8-way set associative caches
L1 cache	2 x 32 KB 4-way set associative instruction caches 2 x 32 KB 8-way set associative data caches

Table 7.2: Hardware for the physical environment

Characteristics	Values
Processor	i5-2400
CPU Frequency	3.1GHz
Lithography	32nm
Memory	4GB
Cores	4
L3 cache	6 MB 12-way set associative shared cache
L2 cache	4 x 256 KB 8-way set associative caches
L1 cache	4 x 32 KB 8-way set associative instruction caches 4 x 32 KB 8-way set associative data caches

When the modified operating system was ready in the virtual environment, it has been installed on a physical computer having the characteristics specified in Table 7.2. For future work the target processor would be acquired (a XEON processor) in order to take advantage of the RAS features offered by these types of processors.

7.1.3 Software

The source code can be downloaded from [Ass19]. Then download version 3.2.1 of Minix3 from the official site www.minix3.org . Install on a computer or a virtual machine (VMWare or Virtualbox). By using an ssh connection overwrite the Minix3 source with the one downloaded from [Ass19]. Then follow the process specified on the official Minix3 website for the compilation of the new kernel. As soon as the new kernel is compiled and the machine is restarted on the new image, the following commands must be executed to compile and install the user library.

```
1  # cd /usr/src/test/
2  # make
3  # cp hardening /bin/
```

Then start the hardening with :

```
1  # hardening 1
```

Initiate the fault injection during hardening with the following command:

```
# hardening 128
```

Disable the fault injection during hardening with the following command:

```
# hardening 256
```

Stop the hardening, with the following command:

```
# hardening 2
```

7.2 Performance tests

A hardening software assessment should be done in three aspects:

- The ability of the hardened software to continue to provide its services like the unhardened software.
- The performance lost must be reasonable so that the CPU time is not fully used to execute hardening code.
- The ability of the hardening software to actually detect and correct all covered classes faults and report not corrected fault.

Hardened Minix3 will be evaluated based on these three criteria. To achieve that, we need to choose appropriated benchmark programs to execute to prove that the hardened Minix3 meets the criteria.

First, an operating system must meet the POSIX standard. The POSIX compliance test programs are chosen to be executed. Second, based on the state-of-the-art a good benchmark has relatively high execution time and combines system calls and computation. So we chose programs like GZIP and MD5 that fall into this category. Third Programs like Dhrystone are computation intensive programs. The compilations of an entire system are complex enough to cover a fairly high range of use cases.

7.2.1 Performance loss due to hardening

Hardening unmodified application programs has a cost, even if no errors have to be corrected. This cost is the computation time of the second execution plus the extra processing involved by each PE, i.e. mainly

the page faults giving access to the PE memory at the beginning of each execution (when the page is not yet in the working set) and the comparison of the results of the two executions, which are included in the set of pages modified by the PE.

The overhead due to the double execution is not compressible and is independent of the length of the PE. The other overheads are linked to the PE independently of its duration: the shortest the PE, the more overhead. Actually most PE end in a system call, so, programs with the highest number of system calls are the most expensive to harden. On the other hand, in computing intensive programs, such as the Dhrystone benchmark, most PE are ended by the retirement counter and may involve single stepping that can also be costly.

7.2.2 Minix 3 POSIX compliance test

The POSIX compliance tests for Minix3 were run on the normal Minix3 and also on the hardened Minix3. The table contains the real time, the user time, and the sys time for each programs. Each one involves several processes.

The real time performance overhead varies between 0.008 and 10 with an average of 3.186 and a standard deviation of 1.025. The user time overhead varies between 1 and 28 with an average of 7.62 and a standard deviation of 6.86. The sys time overhead varies between 0.6 and 27.3 with an average of 7.62 and a standard deviation of 6.86. There is a great fluctuation of the overhead. This is explained by the fact that some programs have a real time (resp. user time and sys time) too small. These programs are sets of small programs that do not run for a long duration. These program can not be used to correctly evaluate the cost of hardening. However they show that the hardened Minix respects the POSIX standard.

7.2.3 MD5 and GZIP

MD5 is a UNIX command that compute the 128-bit hash of a file. It is system call intensive program. GZIP is a program that compresses a file passed as a parameter to produce a file smaller than the original file. It is also a program that makes many system calls. For different file sizes, each of these commands was executed on the hardened Minix and the normal Minix. For each run the CPU usage time allowed to calculate the cost of hardening. The file sizes have been a first time varied between

14MB and 140 MB. Then between 140MB and 1.4GB. The file size did not influence the cost of hardening which remained between 2 and 3 times.

The graphs of Figures 7.1, 7.2 show the performance evolution for each of these commands. We observe a consistency around an average in each case.

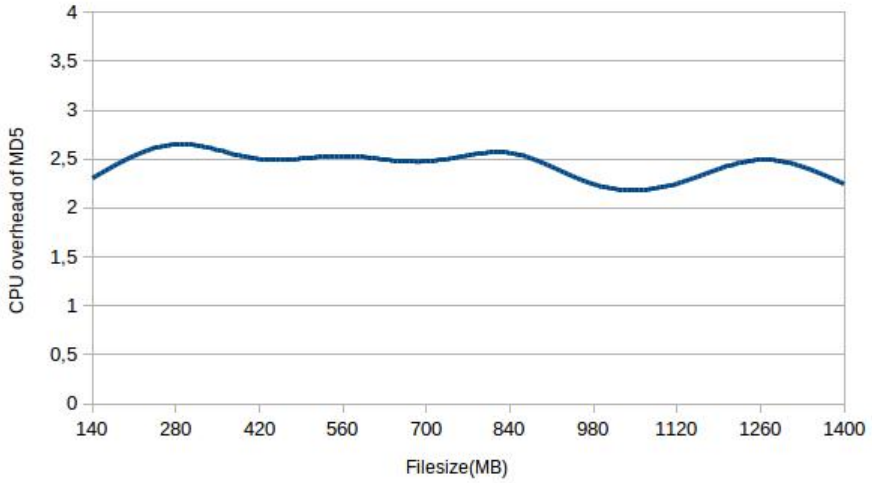

Figure 7.1: Overhead evolution of Md5 on different file size(14MB to 1,40GB)

7.2.4 Dhrystone test

Dhrystone was developed by Reinhold Weicker in 1984 [Sib84].The Dhrystone test contains simple integer arithmetic, string operations, logic decisions and memory accesses intended to reflect the CPU activities in most general purpose computing applications [LKC17, Yor02].

Dhrystone performs a lot of calculation, so it will be more often stopped by the retirement counter. The results obtained in Chapter 3 have shown that it is necessary to maintain the duration of a PE below $t = 250\mu s$. This duration has been converted into a number of instructions. At first the Dhrystone was run for different duration on both the normal Minix and the hardened Minix. The cost of the hardening methods was evaluated and plotted on Figure 7.3. That cost is between 3 and 4 times. The duration of the PEs being long, an

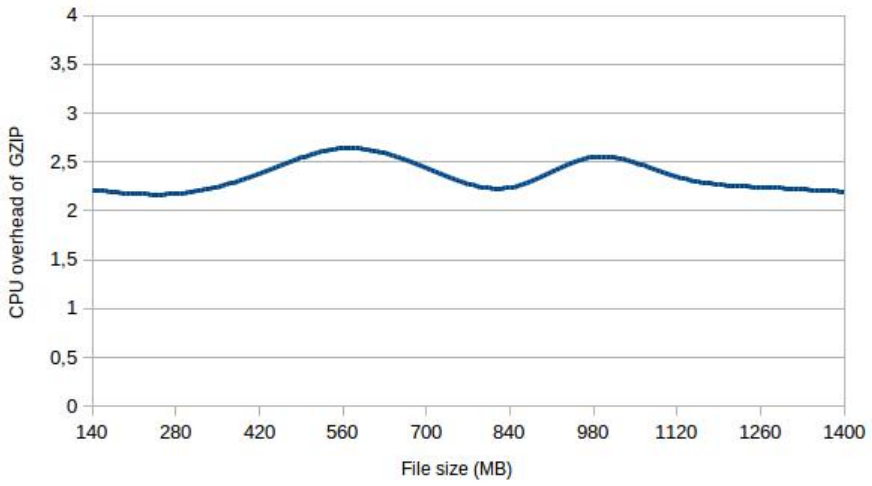

Figure 7.2: Overhead evolution of GZIP on different file size(14MB to 1,40GB)

overhead lower than for system calls intensive programs was expected. Two reasons can explain the obtained results:

- Because of the non-determinism [DWA+19] of the retirement counter, many PE involved single stepping. In average 90% of PE are ended by the retirement counter. Among the PEs ended with the retirement counter 86% of them involved single stepping.
- The second reason is due to the fact that the single stepping and the long duration of the PEs exhaust the quantum of the process more quickly. In these conditions more PE are just aborted. In average 2% of the PEs are aborted. This should thus not have a significant influence.

The results will be better if the retirement counter is more accurate. Nevertheless the constant variation in the cost compared to the execution time of the Dhrystone is reassuring. This shows some stability in the number of PEs executed by single stepping and the number of aborted PEs.

What could be the impact on performance in the case where the duration t is reduced. To find out, the same tests were repeated by reducing the duration t to $25\mu s$. It has been found that the cost of

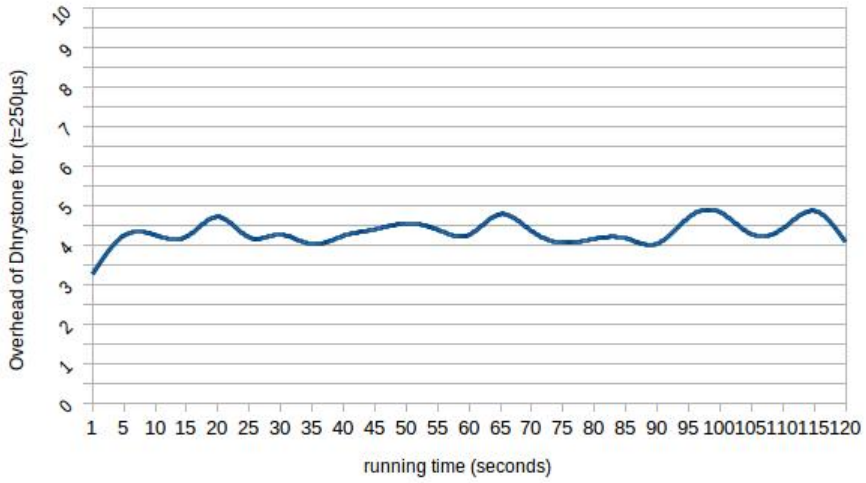

Figure 7.3: CPU intensive program (Dhrystone) for $t = 250\mu s$

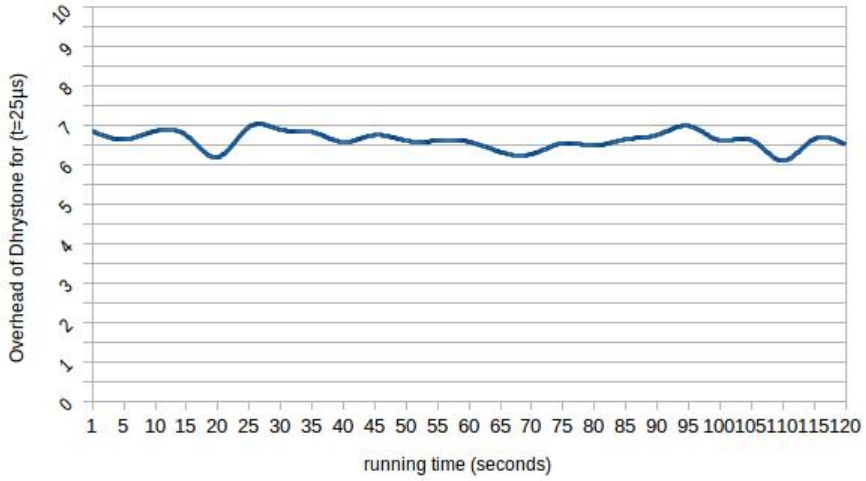

Figure 7.4: CPU intensive program (Dhrystone) for $t = 25\mu s$

hardening has increased from between 3 and 4 to between 5 and 6. There is an increase of two points (See Figure 7.4. This could be explained by the increase in single stepping and the number of PE which is multiplied by 10.

7.2.5 Compiling Minix3

7.2.5.1 Compiling the micro-kernel

The hardened Minix was used to compile the micro-kernel. The execution of the compilation consists of 586 small programs that run independently. Among these programs, the *shell (sh)* represents 48.2%, and the compiler *clang* represents 42.49%. The others programs are *as*, *sed*, *rm*, *mv*, *objcopy*, *ld* and *make*.

The compilation required 278470 PEs. 4297 PEs were aborted because the quantum of the process was exhausted during the hardened execution of the PE (1.54% of the total number of PEs). All PEs are ended by a system call.

The total number of frames needed by the protected memory is 3263226 frames. The maximum number of frames needed by a PE is 14392 frames. The average number of frames needed is 11120 frames.

The total number of ticks in user mode is 136969 (38 min and 6 seconds). The total number of ticks in system mode is 51569 (14min 21s). It required 345 ticks (5.75 s) in user mode and 783 ticks (13.05 s) in system mode for normal Minix. The overhead is 168 times compared to the time of the compilation on the normal Minix3. The overhead seems huge at first sight.

7.2.5.2 Creating new Minix3 boot image

A second compilation of the system was done using the *make install* command in the *src/releasetools* folder. This compilation consists of compiling the micro-kernel, the servers, the drivers. Its produces a new bootable image. It was made on the hardened minix and the resulting image was used to make new tests.

A total of 8597 programs were run during this compilation. The shell *(sh)* represents 42.76% of these programs. The *clang* compiler represents 19.33%. The others programs are *sed, rm, mv, as, objcopy, ld, make, uname, cut, grep, cmp, stat, basename, sync, awk, sysenv, install, cc, abnt*2*.map_genma, dvorak.map_genm, french.map_genm, german.map_genm, japanese.map_ge, latin-america.m,*

olivetti.map_ge, polish.map_genm, russian-cp866.m, russian.map_gen, scandinavian.ma, spanish.map_gen, uk.map_genmap, us − std − esc.map_ , us − std.map_genm, us − swap.map_gen, russian-cp1251., ukraine-koi8-u., portuguese.map_ , abnt2.map_genma, pwd_mkdb, mkfs.mfs, cp, strip, gzip, touch , printroot, tr, cat, ls.

The total number of PEs is 2355456. 0.2% of the PEs were ended by the retirement counter. 0.81% of PEs have been aborted because the process has exhausted its quantum. 98.9% of PEs have been ended by a system call. 49.85% of the PEs belong to the *clang* compiler. This shows that the *clang* compiler runs more than other programs. The shell (*sh*) and the *make* represent respectively 16.03% and 20.51%. These three programs represent 87% of the total PEs. These programs are the most executed.

The protected memory required a total of 22708866 frames of 4KB each. That corresponds to a memory of 86.62 GB. If these programs were in memory from the start to the end of the compilation, it is obvious that it would be impossible to make available such a large amount of memory. Fortunately, each time the program ends, its protected memory are released. 91% of this memory was used by the *clang* compiler, 4.60% by the shell (*sh*), 1.10% by *make* and 1.06% by the *cc* compiler.

The compilation required 621052 ticks (2h 52min 30 s) in user mode and 333145 ticks (1h 32mn 32s) in system. It required 3063 ticks (51.05 s) in user mode and 5993 ticks (1mn 39.88 s) in system mode for normal Minix. The overhead is 106 times compared to the compilation time for the normal Minix3. The overhead seems huge at first sight.

7.2.5.3 Discussion on the overhead

BHT is a fault tolerance technique at process level. Let C be the hardening cost at process level. The previous tests (*GZIP* and *MD5*) have shown that:

$$C = 2 + \epsilon \tag{7.1}$$

with $\epsilon < 1$.

However with the kernel compilation, POSIX conformance tests, the value of ϵ is very high and even higher than 1. An analysis of the elements used during the hardened execution of these programs can help us to understand the reasons for these major differences. During the hardened execution of a PE a list representing the frames US1 and US2

is used respectively for the first and second execution. Elements are often searched in that list and the linked list is often browsed:

- It is browsed at the beginning of the first execution and the second execution in order to implement the protected memory policy to restrict access to some PE's pages to the process.
- It is browsed each time a page fault occurs during the hardened execution.
- It is scanned between the two run in order to keep modified pages during the first run of the PE.
- It is browsed during the comparison phase.
- It is browsed at the end of the comparison to remap each page of the working set on the frame of US0.

The complexity of browsing a linked list is $O(n)$ where n is the size of the linked list. Thus, the higher is n, the less negligible is the time of browsing the list for a process.

That explains the overhead found during the execution of the kernel compilation in the hardened Minix.

In fact for the programs $GZIP$ and $MD5$ the size of the linked list $US1\_US2$ is respectively 193 and 57 elements. While for kernel compilation the maximum size of the linked list for a processes is 7196 elements. More processes run during this compilation have had to use $US1\_US2$ lists whose sizes are beyond 6000. This greatly increases the time spent in the hardening software.

One possible solution would be to use a hash table holding pointers to a set of short linked lists instead of using a single list. This should significantly reduce the browsing time when compared to a single huge list. This way the performance would become more reasonable.

7.2.6 Multi-threading support

Test59 is part of POSIX compliance tests of the Minix3. This program tests the POSIX compliance of the mthread library developed by the Minix3 team. Mthread is a library which implements multi-threading at user level like the GNU pthread library. This program makes 7 tests:

- *test_scheduling*: tests *mthread_create*, *mthread_once*, *mthread_yield*, *mthread_join*

- *test_mutex*: test the implementation of mutex
- *test_rwlock*:
- *test_condition*: tests the implementation of the condition feature for thread synchronization
- *test_attributes*: tests the use of get and set functions to read / modify thread attributes *test_keys*

Test59 was run on the hardened Minix. The goal is to show that the hardened Minix supports multi-threading like the normal Minix. The test59 has passed successfully. A total of 4 programs were executed *rm*, *sh*, *mkdir* and *test59*. The execution required 79 ticks in user mode and 32 ticks in system mode. The total number of PEs is 1721 with 80.9% of PEs belonging to the test59 program. 0.23% of PEs were aborted because the process's quantum was exhausted. All PEs ended with a system call. The number of frames needed for the protected memory is 3298 frames.

7.2.7 Floating point and retirement counter

Sanjeev Das et al. [DWA+19] showed in their recent paper that despite of 10 years of studying performance monitoring counters (PMC), recent processors PMC have the same limitations: non-determinism and over-counting.

In the case of floating-point operations, some xFPU instructions are under-counted or over-counted [MVJ11, ZJH09].

Test47 of the Minix3 POSIX conformance test suite is a floating-point program test. When this test is executed on the hardened Minix with the performance counter enabled, the test failed. An analysis of the log message show that when a PE is interrupted by the retirement counter, the comparison step fails because of the inaccuracy of the retirement counter.

When the retirement counter is disabled the test does not succeed too. In fact it blocks in a loop in which the PE is aborted because the quantum of the process is exhausted. This shows that *test47* is a computing intensive program.

We chose to do another test where the retirement counter is always off. But when the process exhausts its quantum, a new quantum is allocated to it without calling the scheduling process. *Test47* succeeded.

The conclusions we can draw is that due to the non-deterministic nature and the over-count of the retirement counter a program doing a lot of floating-point calculation can not rely on the current version of the retirement counter for their hardened execution. Programs such as test47 that do a lot of calculation, then system calls can be executed in a hardened way by disabling the retirement counter and giving quantum to the process when the latter has exhausted its quantum (the user library has been modified to take into account these features).

However programs that do only calculations hoping to be stopped by an external timer such as whestone which is the floating point version of Dhrystone cannot be taken into account in the current version of the software hardening.

7.3 Tests by fault injection at run-time and evaluation of the results

In the radiation area the traditional way to verify the effectiveness of radiation hardening techniques is by exposing the device to a cyclotron beam, that sends many more particles that what happens in space, allowing this way to verify in a short time the effectiveness of the protection. This is not possible in the present case because pulsing the cyclotron beam in a way to respect the single SEU per PE hypothesis would be very hard. Therefore, the effectiveness was verified by bit flip injection. Fault injection is an activity to test the reliability of a software or a library. There are two categories of fault injection tools: run-time fault injection tools and pre-runtime fault injection tools. The runtime fault injection tools stop the program and inject fault in the current running program context [GDJ+11, NC01] . Whereas pre-runtime fault injection tools need to a compiler to instrument the target code by inserting specific faults at specific points of the program flow [MCV00, GKT13a, YGS08]. Run-time fault injection is easy to implement and the developer does not need to have the source code of the target program. EDFI is a fault injection tool developed by Tanenbaum's team which combines the properties of pre-runtime fault injection tools and runtime fault injection tools to provide more control and high coverage during a fault injection campaign[NCDM13].

Because one of the aims of this research is not to have to modify the application programs to harden, simple runtime injection was selected. The clock interrupts are used to inject bit flips in the registers of the

running hardened process.

The fault injection using the clock interrupt must meet a number of criteria to allow the tests to be relevant:

- In order to avoid having more than one SEU during the hardened execution of a processing element we chose to inject the faults with a periods of 10 PEs. We counts the number of PEs globally in the system. At each clock tick when the difference between the current number of PEs and the the previous injection is not greater than 10, the fault injection is not allowed.

- In the case where fault injection is allowed, at the hardened execution of the next PE (regardless of the process) a fault will be injected. The error is injected either at the beginning of the first execution or at the beginning of the second execution.

- The fault should not be injected into the clock interrupt handler routine. Because it is difficult to know at which stage of its execution the hardened process is at this time. If the fault is injected just after the comparison phase that will induce flaws in the test.

Using this approach, the bits of the registers have been systematically inverted from bit 0 to bit 31. These registers are General-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP), Segment registers (CS, DS, SS, ES, FS, and GS), EFLAGS (program status and control) register, EIP (instruction pointer) register. This choice of registers is classical [YGS08] and easy to implement. They are also validly simulating SEU on combinatorial logic because their effect, if any, ends up in a false bits in registers. The bit flips are introduced in the saved values of the registers just before the beginning of the first execution or at the beginning of the second execution when these values are loaded in the real registers before resuming the interrupted process.

Using the clock (at 60 Hz in Minix 3) means injecting about 20 times more errors than the mean frequency of SEU in space. The 10 hours tests presented here are equivalent to 200 hours in space condition. The tests were first performed on unmodified Minix3. The programs crash quickly. They were then performed on hardening Minix 3. All programs performed their tasks faultless: all errors were recovered.

By making fault injection tests three categories of errors are possible:

1. Computational errors, which are reported in data corruption or erroneous instruction execution.

2. The errors reported by the exception mechanism that lead either to a panic within the micro-kernel or to the termination of the running process.

3. The errors that lead to a hard reset of the CPU. In that case the execution flow goes directly from the hardware to the BIOS, then to the bootloader without a passage through the micro-kernel.

At that moment the first two categories of error are all corrected. The third categories requires taking control of the reset vector, and a modification of the bootloader to make it stateful. This aspect will be solved during the implementation of the hardening of the micro-kernel.

The following programs have been tested:

- md5, on a file of 1.4GB
- gzip, on a file of 1.4GB
- The micro-kernel compilation
- The unixbench compilation
- Running the muti-thread test59 program
- Dhrystone

Table 7.3 contains the different results.

7.3.1 Faults injection during MD5 and GZIP

MD5 and GZIP are single-process programs. The faults were injected a single program. MD5 required 220085 clock ticks in systems mode and user mode while GZIP required 23106 ticks in system mode and user mode. Thus during the tests more faults were injected into MD5 than in GZIP. 118033 faults were injected into MD5 while 4307 faults were injected into GZIP (28 times more faults in MD5 than in GZIP). MD5 is a program that has spent more time in system mode than GZIP.

69.54% of faults injected into GZIP generated no errors. 21.15% generated mis-calculation detected by the DWC. 9.31% were detected by the exception mechanism. None of these injected faults generated a hard reset of the CPU.

73.38% of faults injected into MD5 generated no errors. 18.94% generated mis-calculation detected by the DWC. 7.68% were detected

Table 7.3: Fault injections in various programs

Program	Number of fault injected	Errors detected by DWC	Errors detected by exception	No Error
md5 (1.4GB)	118033	22351	9065	86617
gzip (1.4GB)	4307	911	401	2995
test59	149	71	66	12
make (unixbenchmarks)	3240	1258	807	1175
make (the microkernel)	29732	11838	7753	10141
Dhrystone	2537	987	775	775

by the exception mechanism. None of these injected faults generated a hard reset of the CPU.

Both programs have similar profiles. They perform a lot of disk accesses. They work more on data contained in memory. This explains the fact that the faults in the register did not generate a significant proportion of errors compared to the faults injected.

7.3.2 Fault injection during the compilation of micro-kernel and unixbenckmarks

7.3.2.1 Fault injection during the compilation of unixbenckmarks

The unix benchmarks compilation is a multi process program. A total of 6 processes were executed (*sh, clang, rm, cc, ld and make*).

Figure 7.5 shows for the *clang, ld* and *make* processes more than 66% of the injected faults generated errors. For these processes about 40% of the errors were detected by the DWC while about 26% were detected by the exception mechanism.

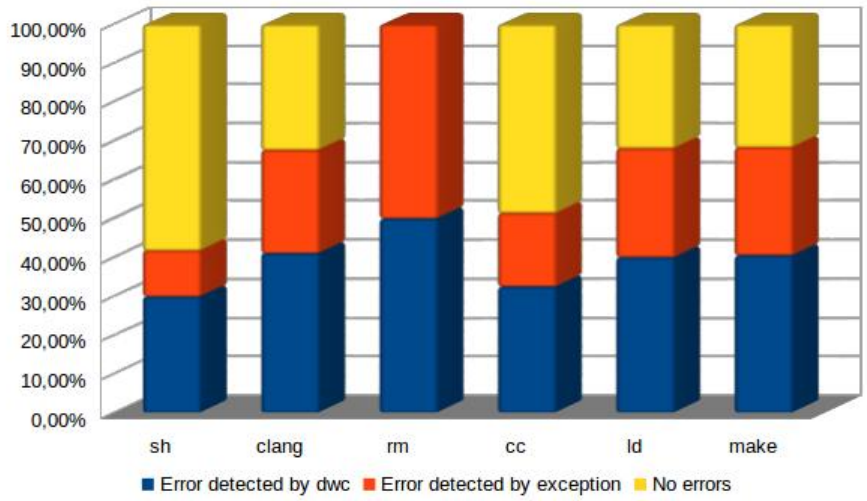

Figure 7.5: Injected errors inside registers during compilation of unixbenchmarks

7.3.2.2 Fault injection during the compilation of the micro-kernel

The micro-kernel compilation is also a multi process program. A total of 9 processes were executed. Figure 7.6 shows the distribution of detected errors. Faults in the process rm did not generate errors because only one error was injected in that process. For processes *clang, as,* and *objcopy* more than 70% of injected faults generated errors. For the process *as* 90% of injected faults generated errors.

7.3.3 Analysis of the results

In these, a first interesting result is that more than 1/3 of injected bit flips do not cause any fault: the program is not disturbed. Faults occur as a consequence of at most 2/3 of injected bit flips and roughly half of these is detected by DWC and half by the hardware of the processor itself (exceptions). Because the number of SEU hitting a processor in space is two to three per second (this is one of the conclusions of the statistical analysis performed to find an appropriate PE duration to avoid a PE being it by multiple SEU) the number of PE to replay is roughly 1 or 2 per second. Since the longest PE executions are stopped by the retirement counter after less than 250μs, one can conclude that

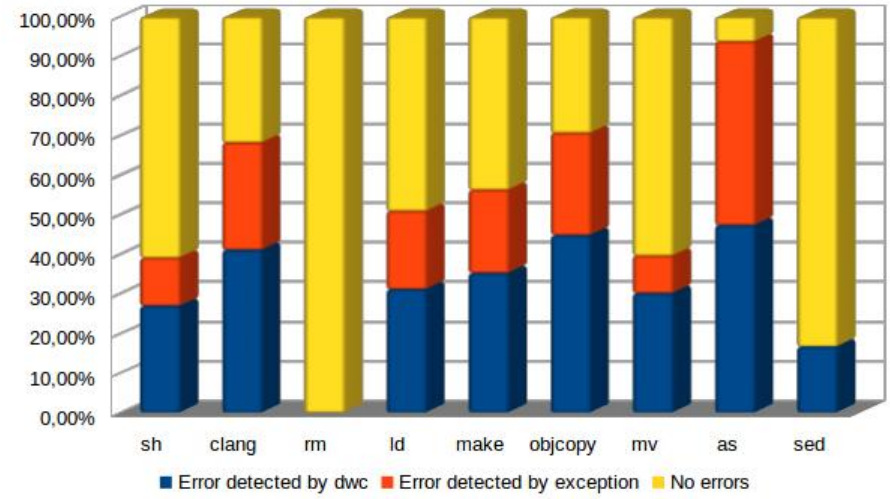

Figure 7.6: Injected errors inside registers during compilation of micro-kernel

less that 1 PE in 1000 has to be replayed: this replay overhead is thus negligible.

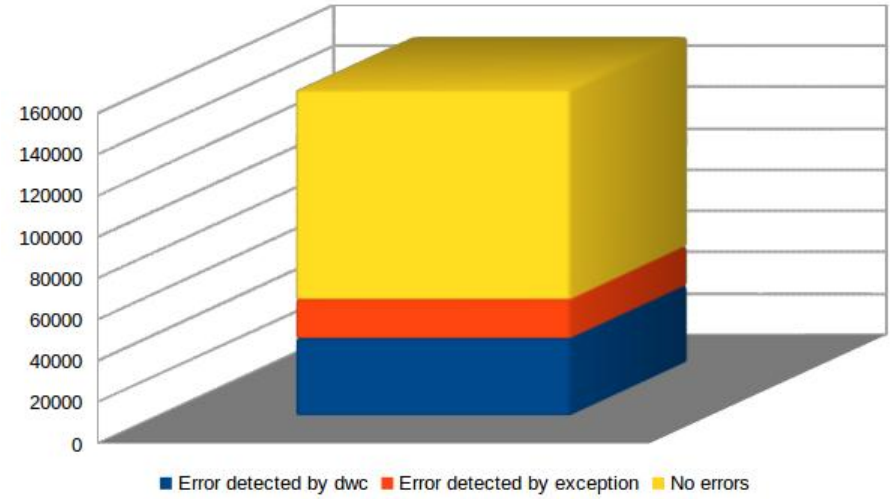

Figure 7.7: Combined injected errors

Figure 7.7 shows the results for 15 days of accumulated tests. All errors were recovered. The injection rate is 20 to 30 times higher than

in real life. It corresponds thus to approximately one year of execution in orbit without a single unrecovered error. Roughly half of the injected errors (simulated SEU) have no consequences 2/3 of those with an effect are detected by DWC and 1/3 by exception.

7.4 Towards cyclotron validation

As stated above, testing in a cyclotron the validity and the effectiveness of the BHT applied to application processes running on a modified Minix3 is not possible using classical cyclotron test methods. Indeed, at the current state of the research, only application processes are hardened, which means that the system will correct any error caused by a SEU hitting the processor during one of the two executions of any processing element, but neither having SEU hitting both executions or causing errors in the operating system itself are acceptable. For cyclotron testing testing the effectiveness of the BHT applied to application processes running on a modified Minix3 would mean turning on or off the cyclotron beam in a few nanoseconds several thousands of time a second. This is not possible: turning on a cyclotron can be fast but stopping the beam would take several milliseconds. A possible test scenario, compatible with what a cyclotron can do, would be using very long processing elements: 10 times longer or more than what was used in computing intensive tests. The cyclotron would be turned on at the beginning of such a long PE, then off and when the cyclotron would have been stopped, the PE would be halted, the OS would then, without irradiation start the second execution of the processing element, compare the results and decide to redo the PE or proceed to the next one. The cyclotron would have to be tuned to a low shooting rate in order not to have more SEU in the long PE than in a real one. This scenario has two major drawbacks:

1. Instead of simulating a space radiation exposure of a duration T in a time t « T, the simulation would take more then 10 times more time than real life in space. A very long cyclotron use would be needed for this, which is definitely not cost effective.

2. No real program would be usable because their PE are too short : it would have to be a program specially written for this experiment.

It is thus clear that testing the results in a cyclotron is not possible. However, because a lot of tests have already been performed by simulation, a verification of some selected results would be possible. In

particular, exposing in a cyclotron, not a real program but a specially made program would be acceptable, if the purpose is only to compare the results of fault injection and of cyclotron testing for selected programs. Therefore, the second drawback could be acceptable. The other problem is the unfordable duration of the test. This problem could be solved by a technique proposed by Michel Melotte of Thales-Alenia-Space: use a bi-processor and run the hardening OS and second execution of each PE run on the second, while only the first execution of each PE is run on the first processor. And of course, only the first processor would be irradiated. That way, the beam would not have to be pulsed. However modifying Minix3 to let it run one of the executions of each PE on a separate processor is a significant modification of the system. The whole process including modification of Minix, preparation of the experiment and cyclotron testing would be several month's work. In cannot fit in the time frame of this thesis but is clearly interesting future work.

Part IV

Conclusions and future works

CHAPTER 8
Conclusions and future works

8.1 Conclusions

The aim of this thesis was to show that an operating system can harden its application processes without modifying their code and even without knowing what they are doing, in order to make these processes insensible to SEU in a space environment. Hardening the operating system itself is out of the scope of this work. This is future work. It was decided to harden application processes first because hardening the OS is much more useful if application programs are hardened by the OS. If not a work similar to harden the OS would have to be performed on each user program. This would be unbounded work. Another good reason to harden the application programs first is that the technique to harden application processes would help simplifying the hardening of the OS particularly in a micro-kernel OS where a lot of OS work is subcontracted to applications programs.

The selected hardening technique is blended hardening, i. e. using software exploiting hardware available in some COTS components, such as exception mechanisms (that are able to detect and report many types of errors), memory protection systems embedded in paging systems, caches, and error correction systems built in memories, such as ECC etc. Software exploits exceptions and complements them to detect errors that the exception mechanisms cannot detect, such as computational faults and unwanted infinite loops. These errors, undetectable by exception mechanisms are often called silent errors. Software error detection and correction is based usually either on majority voting (most frequently TMR: triple Modular Redundancy) or on double execution with comparison (DWC). In this work DWC was selected, because, even though SEU are frequent in space at a human scale (typically several per second on a CPU) they are rare at the scale of the processor clock. Thus needing only two executions instead of at least 3 most of the time was considered an advantage.

A first problem to solve was cutting the code in processing elements

to apply TMR or DWC. Two ideas from the state of the art were reused here: keeping the processing elements short enough to have to handle at most a single SEU in a PE (Lesage et al.) and using system calls as frontiers between PE (Döbel who published it first although we were working on this solution at the same time). These ideas are conflicting because nothing prevents the delay between system calls to be too long to guarantee that at most one SEU could happen in between. This is the main cause why Döbel, who also tried to protect application processes in the OS was not able to correct all errors. Here appear two original contributions of this thesis: a statistical evaluation of the maximum duration of a processing element in order to suffer at most from a single SEU and an innovative technique for counting with precision the number of retired instructions. With these innovations it was possible to define the ending frontier of a PE as either a system call or reaching a predefined number of instructions small enough to guarantee suffering, at most, from a single SEU, whatever happens first.

A second problem to solve was choosing an appropriate operating system. Minix 3 was selected because it is a micro-kernel operating system, with pre-existing fault tolerance features, excellent containment of faults within modules (the micro-kernel, servers), many servers running in user mode, and good available documentation.

The third problem was protecting memory against processes misbehaving after having been hit by a SEU, i.e. indirect SEU effects. This was done using the protections built in the paging system. An analysis of the protection of the paging system showed that its protection is adequate. Without such protection computer would just be too unreliable for servers or mission critical embedded systems.

This a third original contribution of this these.

Memory areas that are neither sensible to direct effects of SEU (howto is out of the scope of this these; it could be by ECC with scrubbing,) nor to indirect effects, using, in an innovative way, the protections built-in the paging system, can thus be implemented. They are called "protected memory". The availability of protected memory allows to restrict the comparison phase of DWC to the data pages accessed by the current processing element. This comparison is performed in the OS, that is assumed to be free of any SEU effect (this protection of the OS is also out of the scope of this these). This comparison of the results of two execution of each PE, along with the possibility to restart the whole processing of the PE in case of error, request keeping 3 copies of the part of the hardened process data memory that had been accessed by

the process. Keeping these three versions consistent was another difficulty, in particular when system calls modify the contents of the process memory (the Minix3 code executing the system calls is unaware of the existence of these three copies), or when that memory is shared between several processes (e.g. after fork and when using memory mapped files or shared memories). The hardening of user processes required a small semantic change to the behavior of the operating system: the scheduler is not allowed anymore to preempt a process in the middle of a PE: rescheduling requests that occur during the processing of a PE are delayed until the end of this PE. With the modifications developed in this these in order to harden application processes, Minix 3 operating system (3.4.0 and 3.2.1) are able to run privilege 3 processes with fault tolerance capability. The micro-kernel and the memory manager have been modified. The results obtained so far show that software hardening in the operating system is possible and that the performance degradation is acceptable. When the hypotheses on the rate of SEU events are satisfied, all errors are corrected and a statistical analysis tells us that the residual errors (those that could happen when the hypotheses are not respected) correspond to a FIT between 8.49×10^{-3} and 1.95×10^{-2} for FDSOI nano-scale technology and 1.58×10^{-2} and 4.76×10^{-1} for Bulk CMOS nano-scale technology in GEO environment. The worst case for FDSOI is the node 45nm with a FIT equals to 2.39×10^{-2}. The best case is the node 14nm with a FIT equals to 8.49×10^{-3}. At the time of writing, the BHT approach allows to let Minix 3, running on Intel COTS processors to remove all SEU effects in its application programs with the required probability of success. This is the best result ever obtained in hardening methods not requiring any specific hardware to be added, but only software. This happens at an affordable cost that can be as low as multiplying by 2,5 the execution time of the programs, but is still often 6 times for pure CPU intensive programs. The implementation has been tested by fault injection in processor registers (program counter, general purposes registers, segment registers and Intel performance monitoring registers). All errors were recovered. A long time (15 days) test has been performed, with both simple and complex programs. The results obtained meet the expectations and the requirements of the space industry with one noticeable exception: the hardening technique has not yet been cyclotron tested. Classical cyclotron testing methods are not possible because respecting the "not more than 1 SEU per processing element" condition for the applicability of the hardening method coupled with the necessity to only irradiate during the execution of application

programs would require to switch on and off the cyclotron several thousand times per second and ideally in less than a nanosecond. This is obviously not possible. The work was thus tested by fault injection, but very thoroughly. Although reobtaining the same information in a cyclotron would not be possible, limited verification of some of the results would be possible after modifying Minix3 to run application programs to harden and operating systems on separate processors and irradiating only the processor executing hardened application programs. Another possibility is to first harden the OS itself then the complete system could be irradiated and the result would cover the complete system, not only application programs. All this is future work: developing and testing this will require several months. Floating point intensive calculation programs are not fully supported by the hardening software. Because the retirement counter is non-deterministic and non accurate for some floating point instructions. Other improvements are planned. The first is to remove the necessity of hardware scrubbing of the memory. This will be tackled as future work by adding a scrubbing server to Minix3. Finally, before actually using this work in space, the micro-kernel will have to be modified to harden itself and some servers. This is feasible because the sources of Minix3 are available.

8.2 Future works

1. **Hardening the original Minix micro-kernel:** The standard minix micro kernel is actually a set of very small programs activated on demand by system or kernel calls, interrupts or exceptions. Each of these programs is short enough to be considered as a PE by itself. A first thing to do will be to evaluate (for instance using the retirement counter) the length of each of these programs, from the interrupt/exception/system or kernel call to the return (often after sending a message). The second thing to do is to evaluate the risk. This involves two aspects: first, what is the probability of a SEU during this time; and second, for each of these programs, what could happen in case of undetected SEU and how can it be detected? Is there for instance a way to verify the correctness of the result and in case of undetected faulty result, can it harm? Finally given the previous information, is it necessary to modify this program to detect faulty results. However in a hardening Minix 3 most of the system time is spent in

the hardening code, not in the original Minix code, so most errors will occur in the hardening code itself.

2. **Hardening the hardening code added to the micro-kernel**
   Executing the hardening code in the micro-kernel is by far the longest activity in the micro-kernel. Indeed, apart from the double execution, most of the hardening happens in the micro-kernel and the hardening increases the duration of the programs by the time of the comparison and copy of pages. The probability of SEU induced faults in these activities must be evaluated. Here again, the SEU can be detected by exception or be silent. We assume here that all comparisons are performed using CRCs. A silent fault in a copy is unacceptable. Copies must thus also be verified by CRC. The time of the copy will thus be increased by the CRC computation, that can be performed by the same loop. A silent fault in a CRC computation can only lead to a faulty CRC that will cause rejecting a good value, thus a small loss of time to copy again the block, or in the worst case to repaly the hardened user PE. Of course all the hardening code will have to be analyzed to find out what could go wrong because of silent errors out of the copies and CRC calculations, if the consequences are acceptable and how to detect these situations.

3. **Hardening the servers**
   Hardening the servers means understanding thoroughly their code to find out

   - if hardening them like application programs is sufficient
   - if hardening like application programs can be simplified (because their processing is short some of them do probably not need the precaution of the retirement counter to avoid being too long),

4. **Retirement counter and floating point**
   Thoroughly investigate the issue of the retirement counter to support programs that perform floating-point computations. Similarly make the retirement counter more deterministic to eliminate the single stepping.

5. **Direct Memory Access issue**
   DMA can write directly to the process memory. That is used

to speed up disk to memory access. How to guarantee that the transferred data are not corrupted by a fault due to a SEU? The integrity of the data should be verified after the transfer. The current version of the hardening software does not take into account this feature. Future work will take it in account by providing a fault tolerant DMA.

6. **PRAM at page table scale**
   A profiling of the hardening software allowed to discover that the handling of the protected memory at the scale of a page is very expensive when the number of page of the working set increases. A normal handling of the protected memory during an execution of a processing element includes four tasks:

   (a) Limiting access to data memory in the process memory space. During this phase each of the data pages of the process memory space is analyzed in order to decide which one to put in read-only or read-write.

   (b) The handling of page faults. When a page fault is signaled by the hardware, you must go through the list of data pages in the process memory space to determine to which page the address belong.

   (c) Comparison of the pages modified during the two runs.

   (d) Restoring the memory space of the process in the initial state as expected by the others part of the OS.

   Tasks (b) and (c) are executed on a subset of the data pages of the process memory space. Their complexity is therefore not a function of the growth of the number of data pages of the address space of the process. On the other hand, operations (a) and (d) are performed on all the data pages of the memory space of the process. So when the number of pages increases, the complexity of these tasks also increases resulting in high overheads for the hardening software. For future work we plan to do operations at the page table scale. In addition to having three frames per data page, we intend to allocate three page tables per process. Thus, operations (a) and (d) will be just a switch of page table instead of an handling of each page.

Publications

- Assogba, Emery K and Lobelle, Marc. Hardening application programs by the operating system on COTS processors: what protection can be expected and at what performance cost. *2017 17th European Conference on Radiation and Its Effects on Components and Systems (RADECS), Geneva*, pp.1-6, Oct. 2017

- Assogba, Emery K and Lobelle, Marc. Can MINIX Be "Tuned" in Order to Satisfy Hard Real-time Constraints without Losing Its Soul? *MinixCon 2016 VUA,Amsterdam* , http://www.minix3.org/conference/2016/program.html#Talk-2, Feb. 2016

- Assogba, Emery K and Lobelle, Marc. A new way to let the operating system harden its application processes against SEU. *23rd annual single event effects (SEE) symposium coupled with the military and aerospace programmable logic devices (MAPLD) workshop, San Diego*, Mai. 2014

APPENDIX A
Annex

A.1 Annex 1

Intel Core processors "Performance Analysis Guide for Intel CoreTM i7 Processor and Intel XeonTM 5500 processors" by Dr. David Levinthal PhD, available at
https://software.intel.com/sites/products/collateral/hpc/vtune/performance-analysis-guide.pdf

A.2 Annex 2

Table A.1: SEE rates (/day/bit) from OMERE in LEOISS orbit space environment (FDSOI)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	3.16×10^{-14}	5.21×10^{-14}	1.73×10^{-15}	9.93×10^{-17}	9.88×10^{-17}	2.99×10^{-17}
Solar ions (Heavy Ions Rate)	0	0	0	0	0	0
Trapped protons (Protons Rate)	1.91×10^{-4}	2.85×10^{-4}	3.57×10^{-4}	4.20×10^{-4}	6.24×10^{-4}	7.12×10^{-4}
Solar protons (Protons Rate)	2.93×10^{-6}	4.39×10^{-6}	5.86×10^{-6}	7.32×10^{-6}	1.17×10^{-5}	1.61×10^{-5}
Cosmic rays (Protons Rate)	3.10×10^{-5}	4.65×10^{-5}	6.20×10^{-5}	7.74×10^{-5}	1.24×10^{-4}	1.70×10^{-4}
Heavy Ions rate (in flare)	3.16×10^{-14}	5.21×10^{-14}	1.73×10^{-15}	9.93×10^{-17}	9.88×10^{-17}	2.99×10^{-17}

Heavy Ions rate (out-of-flare)	3.16×10^{-14}	5.21×10^{-14}	1.73×10^{-15}	9.93×10^{-17}	9.88×10^{-17}	2.99×10^{-17}
Protons rate (in flare)	2.25×10^{-4}	3.36×10^{-4}	4.25×10^{-4}	5.04×10^{-4}	7.59×10^{-4}	8.98×10^{-4}
Protons rate (out-of-flare)	2.25×10^{-4}	3.36×10^{-4}	4.25×10^{-4}	5.04×10^{-4}	7.59×10^{-4}	8.98×10^{-4}
Total rate (in flare)	2.25×10^{-4}	3.36×10^{-4}	4.25×10^{-4}	5.04×10^{-4}	7.59×10^{-4}	8.98×10^{-4}
Total rate (out-of-flare)	2.25×10^{-4}	3.36×10^{-4}	4.25×10^{-4}	5.04×10^{-4}	7.59×10^{-4}	8.98×10^{-4}

Table A.2: P_0 and P_4 in LEOISS orbit space environment (FDSOI)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	1.30×10^{-8}	1.94×10^{-8}	2.46×10^{-8}	2.92×10^{-8}	4.39×10^{-8}	5.20×10^{-8}
P_0 $N = 4Mbits$						
P_0	1.86×10^{-10}	4.16×10^{-10}	6.65×10^{-10}	9.36×10^{-10}	2.12×10^{-9}	2.97×10^{-9}
SER_0 (FIT)	6.71×10^{2}	1.50×10^{3}	2.39×10^{3}	3.37×10^{3}	7.64×10^{3}	1.07×10^{4}
P_4 **For** $n_c = 256kbits$						
P_4	1.17×10^{-17}	2.60×10^{-17}	4.16×10^{-17}	5.86×10^{-17}	1.33×10^{-16}	1.86×10^{-16}
SER_4 (FIT)	4.20×10^{-5}	9.36×10^{-5}	1.50×10^{-4}	2.11×10^{-4}	4.78×10^{-4}	6.69×10^{-4}
P_4 **For** $n_c = 512kbits$						
P_4	2.33×10^{-17}	5.20×10^{-17}	8.32×10^{-17}	1.17×10^{-16}	2.66×10^{-16}	3.72×10^{-16}
SER_4 (FIT)	8.40×10^{-5}	1.87×10^{-4}	3.00×10^{-4}	4.22×10^{-4}	9.56×10^{-4}	1.34×10^{-3}

Table A.3: SEE rates (/day/bit) from OMERE in LEOISS orbit space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	1.69×10^{-9}	4.06×10^{-9}	4.25×10^{-9}	3.75×10^{-9}	1.05×10^{-8}	1.88×10^{-8}
Solar ions (Heavy Ions Rate)	0	0	0	0	0	0
Trapped protons (Protons Rate)	2.01×10^{-4}	3.02×10^{-4}	4.03×10^{-4}	5.04×10^{-4}	8.06×10^{-4}	1.11×10^{-3}
Solar protons (Protons Rate)	2.93×10^{-6}	4.39×10^{-6}	5.86×10^{-6}	7.32×10^{-6}	1.17×10^{-5}	1.61×10^{-5}
Cosmic rays (Protons Rate)	3.10×10^{-5}	4.65×10^{-5}	6.20×10^{-5}	7.74×10^{-5}	1.24×10^{-4}	1.70×10^{-4}
Heavy Ions rate (in flare)	1.69×10^{-9}	4.06×10^{-9}	4.25×10^{-9}	3.75×10^{-9}	1.05×10^{-8}	1.88×10^{-8}
Heavy Ions rate (out-of-flare)	1.69×10^{-9}	4.06×10^{-9}	4.25×10^{-9}	3.75×10^{-9}	1.05×10^{-8}	1.88×10^{-8}
Protons rate (in flare)	2.35×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.88×10^{-4}	9.42×10^{-4}	1.29×10^{-3}
Protons rate (out-of-flare)	2.35×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.88×10^{-4}	9.42×10^{-4}	1.29×10^{-3}
Total rate (in flare)	2.35×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.88×10^{-4}	9.42×10^{-4}	1.29×10^{-3}
Total rate (out-of-flare)	2.35×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.88×10^{-4}	9.42×10^{-4}	1.29×10^{-3}

Table A.4: P_0 and P_4 in LEOISS orbit space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	7.47×10^{-8}	5.45×10^{-8}	3.40×10^{-8}	2.73×10^{-8}	2.04×10^{-8}	1.36×10^{-8}
P_0 $N = 4Mbits$						
P_0	6.14×10^{-9}	3.27×10^{-9}	1.27×10^{-9}	8.17×10^{-10}	4.59×10^{-10}	2.03×10^{-10}
SER_0 (FIT)	2.21×10^{4}	1.18×10^{4}	4.59×10^{3}	2.94×10^{3}	1.65×10^{3}	7.32×10^{2}
P_4 **For** $n_c = 256kbits$						
P_4	3.84×10^{-16}	2.04×10^{-16}	7.97×10^{-17}	5.11×10^{-17}	2.87×10^{-17}	1.27×10^{-17}
SER_4 (FIT)	1.38×10^{-3}	7.36×10^{-4}	2.87×10^{-4}	1.84×10^{-4}	1.03×10^{-4}	4.58×10^{-5}
P_4 **For** $n_c = 512kbits$						
P_4	7.68×10^{-16}	4.09×10^{-16}	1.59×10^{-16}	1.02×10^{-16}	5.74×10^{-17}	2.55×10^{-17}
SER_4 (FIT)	2.77×10^{-3}	1.47×10^{-3}	5.74×10^{-4}	3.68×10^{-4}	2.07×10^{-4}	9.16×10^{-5}

Table A.5: SEE rates (/day/bit) from OMERE in LEOISS orbit space environment (Bulk CMOS)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	2.18×10^{-8}	3.48×10^{-8}	4.57×10^{-8}	5.42×10^{-8}	8.90×10^{-8}	1.14×10^{-7}
Solar ions (Heavy Ions Rate)	1.46×10^{-7}	2.48×10^{-7}	2.92×10^{-7}	2.84×10^{-7}	5.64×10^{-7}	5.16×10^{-7}
Trapped protons (Protons Rate)	2.02×10^{-4}	3.02×10^{-4}	4.03×10^{-4}	5.04×10^{-4}	8.06×10^{-4}	1.11×10^{-3}
Solar protons (Protons Rate)	2.93×10^{-6}	4.39×10^{-6}	5.86×10^{-6}	7.32×10^{-6}	1.17×10^{-5}	1.61×10^{-5}
Cosmic rays (Protons Rate)	3.10×10^{-5}	4.65×10^{-5}	6.20×10^{-5}	7.74×10^{-5}	1.24×10^{-4}	1.70×10^{-4}
Heavy Ions rate (in flare)	1.68×10^{-7}	2.82×10^{-7}	3.38×10^{-7}	3.39×10^{-7}	6.53×10^{-7}	6.30×10^{-7}
Heavy Ions rate (out-of-flare)	1.68×10^{-7}	2.82×10^{-7}	3.38×10^{-7}	3.39×10^{-7}	6.53×10^{-7}	6.30×10^{-7}
Protons rate (in flare)	2.35×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.89×10^{-4}	9.42×10^{-4}	1.29×10^{-3}
Protons rate (out-of-flare)	2.35×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.89×10^{-4}	9.42×10^{-4}	1.29×10^{-3}
Total rate (in flare)	2.36×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.89×10^{-4}	9.42×10^{-4}	1.30×10^{-3}
Total rate (out-of-flare)	2.36×10^{-4}	3.53×10^{-4}	4.71×10^{-4}	5.89×10^{-4}	9.42×10^{-4}	1.30×10^{-3}

Table A.6: P_0 and P_4 in LEOISS orbit space environment (Bulk CMOS)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	1.36×10^{-8}	2.04×10^{-8}	2.73×10^{-8}	3.41×10^{-8}	5.45×10^{-8}	7.50×10^{-8}
P_0 $N = 4Mbits$						
P_0	2.05×10^{-10}	4.59×10^{-10}	8.17×10^{-10}	1.28×10^{-9}	3.27×10^{-9}	6.18×10^{-10}
SER_0 (FIT)	7.36×10^{2}	1.65×10^{3}	2.94×10^{3}	4.60×10^{3}	1.18×10^{4}	2.22×10^{4}
P_4 **For** $n_c = 256kbits$						
P_4	1.28×10^{-17}	2.87×10^{-17}	5.12×10^{-17}	8.00×10^{-17}	2.05×10^{-16}	3.87×10^{-16}
SER_4 (FIT)	4.61×10^{-5}	1.03×10^{-4}	1.84×10^{-4}	2.88×10^{-4}	7.37×10^{-4}	1.39×10^{-3}
P_4 **For** $n_c = 512kbits$						
P_4	2.56×10^{-17}	5.75×10^{-17}	1.02×10^{-16}	1.60×10^{-16}	4.09×10^{-16}	7.74×10^{-16}
SER_4 (FIT)	9.22×10^{-5}	2.07×10^{-4}	3.68×10^{-4}	5.76×10^{-4}	1.47×10^{-3}	2.79×10^{-3}

Table A.7: SEE rates (/day/bit) from OMERE in MEO orbit space environment (FDSOI)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	8.22×10^{-13}	1.35×10^{-12}	4.28×10^{-14}	1.56×10^{-15}	1.54×10^{-15}	4.58×10^{-16}
Solar ions (Heavy Ions Rate)	0	0	0	0	0	0
Trapped protons (Protons Rate)	2.86×10^{-3}	4.23×10^{-3}	4.95×10^{-3}	5.51×10^{-3}	7.85×10^{-3}	8.29×10^{-3}
Solar protons (Protons Rate)	3.03×10^{-3}	4.39×10^{-3}	4.00×10^{-3}	3.71×10^{-3}	4.35×10^{-3}	3.35×10^{-3}
Cosmic rays (Protons Rate)	1.74×10^{-4}	2.62×10^{-4}	3.49×10^{-4}	4.36×10^{-4}	6.97×10^{-4}	9.56×10^{-4}
Heavy Ions rate (in flare)	8.22×10^{-13}	1.35×10^{-12}	4.28×10^{-14}	1.56×10^{-15}	1.54×10^{-15}	4.58×10^{-16}
Heavy Ions rate (out-of-flare)	8.22×10^{-13}	1.35×10^{-12}	4.28×10^{-14}	1.56×10^{-15}	1.54×10^{-15}	4.58×10^{-16}
Protons rate (in flare)	6.07×10^{-3}	8.88×10^{-3}	9.30×10^{-3}	9.66×10^{-3}	1.29×10^{-2}	1.26×10^{-2}
Protons rate (out-of-flare)	6.07×10^{-3}	8.88×10^{-3}	9.30×10^{-3}	9.66×10^{-3}	1.29×10^{-2}	1.26×10^{-2}
Total rate (in flare)	6.07×10^{-3}	8.88×10^{-3}	9.30×10^{-3}	9.66×10^{-3}	1.29×10^{-2}	1.26×10^{-2}
Total rate (out-of-flare)	6.07×10^{-3}	8.88×10^{-3}	9.30×10^{-3}	9.66×10^{-3}	1.29×10^{-2}	1.26×10^{-2}

Table A.8: P_0 and P_4 in MEO orbit space environment (FDSOI)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	3.51×10^{-7}	5.14×10^{-7}	5.38×10^{-7}	5.59×10^{-7}	7.46×10^{-7}	7.29×10^{-7}
P_0 $N = 4Mbits$						
P_0	1.36×10^{-7}	2.90×10^{-7}	3.18×10^{-7}	3.43×10^{-7}	6.12×10^{-7}	5.84×10^{-7}
SER_0 (FIT)	4.88×10^{5}	1.04×10^{6}	1.15×10^{6}	1.24×10^{6}	2.20×10^{6}	2.10×10^{6}
P_4 **For** $n_c = 256kbits$						
P_4	8.49×10^{-15}	1.82×10^{-14}	1.99×10^{-14}	2.15×10^{-14}	3.83×10^{-14}	3.66×10^{-14}
SER_4 (FIT)	3.06×10^{-2}	6.54×10^{-2}	7.18×10^{-2}	7.74×10^{-2}	1.38×10^{-1}	1.32×10^{-1}
P_4 **For** $n_c = 512kbits$						
P_4	1.70×10^{-14}	3.63×10^{-14}	3.99×10^{-14}	4.30×10^{-14}	7.67×10^{-14}	7.32×10^{-14}
SER_4 (FIT)	6.11×10^{-2}	1.31×10^{-1}	1.44×10^{-1}	1.55×10^{-1}	2.76×10^{-1}	2.63×10^{-1}

Table A.9: SEE rates (/day/bit) from OMERE in MEO orbit space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	1.63×10^{-8}	3.39×10^{-8}	3.74×10^{-8}	3.60×10^{-8}	8.35×10^{-8}	1.34×10^{-7}
Solar ions (Heavy Ions Rate)	0	0	0	0	0	0
Trapped protons (Protons Rate)	3.25×10^{-3}	4.88×10^{-3}	6.50×10^{-3}	8.12×10^{-3}	1.30×10^{-2}	1.79×10^{-2}
Solar protons (Protons Rate)	4.24×10^{-3}	6.36×10^{-3}	8.48×10^{-3}	1.06×10^{-2}	1.70×10^{-2}	2.33×10^{-2}
Cosmic rays (Protons Rate)	1.74×10^{-4}	2.62×10^{-4}	3.49×10^{-4}	4.36×10^{-4}	6.98×10^{-4}	9.60×10^{-4}
Heavy Ions rate (in flare)	1.63×10^{-8}	3.39×10^{-8}	3.74×10^{-8}	3.60×10^{-8}	8.35×10^{-8}	1.34×10^{-7}
Heavy Ions rate (out-of-flare)	1.63×10^{-8}	3.39×10^{-8}	3.74×10^{-8}	3.60×10^{-8}	8.35×10^{-8}	1.34×10^{-7}
Protons rate (in flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.06×10^{-2}	4.21×10^{-2}
Protons rate (out-of-flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.06×10^{-2}	4.21×10^{-2}
Total rate (in flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.06×10^{-2}	4.21×10^{-2}
Total rate (out-of-flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.06×10^{-2}	4.21×10^{-2}

Table A.10: P_0 and P_4 in MEO orbit space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	4.44×10^{-7}	6.66×10^{-7}	8.86×10^{-7}	1.11×10^{-6}	1.77×10^{-6}	2.44×10^{-6}
P_0 $N = 4Mbits$						
P_0	2.16×10^{-7}	4.87×10^{-7}	8.62×10^{-7}	1.35×10^{-6}	3.45×10^{-6}	6.51×10^{-6}
SER_0 (FIT)	7.79×10^{5}	1.75×10^{6}	3.10×10^{6}	4.88×10^{6}	1.24×10^{7}	2.34×10^{7}
P_4 **For** $n_c = 256kbits$						
P_4	1.36×10^{-14}	3.05×10^{-14}	5.40×10^{-14}	8.49×10^{-14}	2.16×10^{-13}	4.09×10^{-13}
SER_4 (FIT)	4.88×10^{-2}	1.10×10^{-1}	1.94×10^{-1}	3.06×10^{-1}	7.78×10^{-1}	1.47×10^{0}
P_4 **For** $n_c = 512kbits$						
P_4	2.71×10^{-14}	6.10×10^{-14}	1.08×10^{-13}	1.70×10^{-13}	4.32×10^{-13}	8.17×10^{-13}
SER_4 (FIT)	9.76×10^{-2}	2.19×10^{-1}	3.89×10^{-1}	6.11×10^{-1}	1.56×10^{0}	2.94×10^{0}

Table A.11: SEE rates (/day/bit) from OMERE in MEO orbit space environment (CMOS)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	1.07×10^{-7}	1.69×10^{-7}	2.22×10^{-7}	2.64×10^{-7}	4.27×10^{-7}	5.48×10^{-7}
Solar ions (Heavy Ions Rate)	1.46×10^{-7}	2.48×10^{-7}	2.92×10^{-7}	2.84×10^{-7}	5.64×10^{-7}	5.16×10^{-7}
Trapped protons (Protons Rate)	3.25×10^{-3}	4.88×10^{-3}	6.50×10^{-3}	8.13×10^{-3}	1.30×10^{-2}	1.79×10^{-2}
Solar protons (Protons Rate)	4.25×10^{-3}	6.37×10^{-3}	8.49×10^{-3}	1.06×10^{-2}	1.70×10^{-2}	2.33×10^{-2}
Cosmic rays (Protons Rate)	1.74×10^{-4}	2.62×10^{-4}	3.49×10^{-4}	4.36×10^{-4}	6.98×10^{-4}	9.60×10^{-4}
Heavy Ions rate (in flare)	2.54×10^{-7}	4.17×10^{-7}	5.14×10^{-7}	5.48×10^{-7}	9.91×10^{-7}	1.06×10^{-6}
Heavy Ions rate (out-of-flare)	2.54×10^{-7}	4.17×10^{-7}	5.14×10^{-7}	5.48×10^{-7}	9.91×10^{-7}	1.06×10^{-6}
Protons rate (in flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.07×10^{-2}	4.22×10^{-2}
Protons rate (out-of-flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.07×10^{-2}	4.22×10^{-2}
Total rate (in flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.07×10^{-2}	4.22×10^{-2}
Total rate (out-of-flare)	7.67×10^{-3}	1.15×10^{-2}	1.53×10^{-2}	1.92×10^{-2}	3.07×10^{-2}	4.22×10^{-2}

Table A.12: P_0 and P_4 in MEO orbit space environment (Bulk CMOS)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	4.44×10^{-7}	6.66×10^{-7}	8.86×10^{-7}	1.11×10^{-6}	1.78×10^{-6}	2.44×10^{-6}
P_0 $N = 4Mbits$						
P_0	2.17×10^{-7}	4.87×10^{-7}	8.62×10^{-7}	1.35×10^{-6}	3.46×10^{-6}	6.54×10^{-6}
SER_0 (FIT)	7.80×10^{5}	1.75×10^{6}	3.10×10^{6}	4.88×10^{6}	1.25×10^{7}	2.35×10^{7}
P_4 **For** $n_c = 256kbits$						
P_4	1.36×10^{-14}	3.05×10^{-14}	5.40×10^{-14}	8.49×10^{-14}	2.17×10^{-13}	4.10×10^{-13}
SER_4 (FIT)	4.88×10^{-2}	1.10×10^{-1}	1.94×10^{-1}	3.06×10^{-1}	7.82×10^{-1}	1.48×10^{0}
P_4 **For** $n_c = 512kbits$						
P_4	2.71×10^{-14}	6.10×10^{-14}	1.08×10^{-13}	1.70×10^{-13}	4.34×10^{-13}	8.21×10^{-13}
SER_4 (FIT)	9.76×10^{-2}	2.20×10^{-1}	3.89×10^{-1}	6.11×10^{-1}	1.56×10^{0}	2.95×10^{0}

Table A.13: SEE rates (/day/bit) from OMERE in Open space orbit space environment (FDSOI)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	9.20×10^{-13}	1.51×10^{-12}	4.78×10^{-14}	1.74×10^{-15}	1.71×10^{-15}	5.10×10^{-6}
Solar ions (Heavy Ions Rate)	0	0	0	0	0	0
Trapped protons (Protons Rate)	0	0	0	0	0	0
Solar protons (Protons Rate)	3.43×10^{-3}	4.96×10^{-3}	4.52×10^{-3}	4.19×10^{-3}	4.92×10^{-3}	3.78×10^{-3}
Cosmic rays (Protons Rate)	1.91×10^{-4}	2.87×10^{-4}	3.83×10^{-4}	4.78×10^{-4}	7.64×10^{-4}	1.05×10^{-3}
Heavy Ions rate (in flare)	9.20×10^{-13}	1.51×10^{-12}	4.78×10^{-14}	1.74×10^{-15}	1.71×10^{-15}	5.10×10^{-16}
Heavy Ions rate (out-of-flare)	9.20×10^{-13}	1.51×10^{-12}	4.78×10^{-14}	1.74×10^{-15}	1.71×10^{-15}	5.10×10^{-16}
Protons rate (in flare)	3.62×10^{-3}	5.25×10^{-3}	4.90×10^{-3}	4.67×10^{-3}	5.68×10^{-3}	4.83×10^{-3}
Protons rate (out-of-flare)	3.62×10^{-3}	5.25×10^{-3}	4.90×10^{-3}	4.67×10^{-3}	5.68×10^{-3}	4.83×10^{-3}
Total rate (in flare)	3.62×10^{-3}	5.25×10^{-3}	4.90×10^{-3}	4.67×10^{-3}	5.68×10^{-3}	4.83×10^{-3}
Total rate (out-of-flare)	3.62×10^{-3}	5.25×10^{-3}	4.90×10^{-3}	4.67×10^{-3}	5.68×10^{-3}	4.83×10^{-3}

Table A.14: P_0 and P_4 in Open space environment (FDSOI)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	2.10×10^{-7}	3.04×10^{-7}	2.84×10^{-7}	2.70×10^{-7}	3.29×10^{-7}	2.80×10^{-7}
P_0 $N = 4Mbits$						
P_0	4.82×10^{-8}	1.01×10^{-7}	8.84×10^{-8}	8.03×10^{-8}	1.19×10^{-7}	8.59×10^{-8}
SER_0 (FIT)	1.74×10^{5}	3.65×10^{5}	3.18×10^{5}	2.89×10^{5}	4.28×10^{5}	3.09×10^{5}
P_4 **For** $n_c = 256kbits$						
P_4	3.02×10^{-15}	6.35×10^{-15}	5.53×10^{-15}	5.03×10^{-15}	7.44×10^{-15}	5.38×10^{-15}
SER_4 (FIT)	1.09×10^{-2}	2.29×10^{-2}	1.99×10^{-2}	1.81×10^{-2}	2.68×10^{-2}	1.94×10^{-2}
P_4 **For** $n_c = 512kbits$						
P_4	6.04×10^{-15}	1.27×10^{-14}	1.11×10^{-14}	1.01×10^{-14}	1.49×10^{-14}	1.08×10^{-14}
SER_4 (FIT)	2.17×10^{-2}	4.57×10^{-2}	3.98×10^{-2}	3.62×10^{-2}	5.35×10^{-2}	3.87×10^{-2}

Table A.15: SEE rates (/day/bit) from OMERE in Open space orbit space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	1.81×10^{-8}	4.15×10^{-8}	4.15×10^{-8}	4.00×10^{-8}	9.24×10^{-8}	1.48×10^{-7}
Solar ions (Heavy Ions Rate)	0	0	0	0	0	0
Trapped protons (Protons Rate)	0	0	0	0	0	0
Solar protons (Protons Rate)	4.80×10^{-3}	7.20×10^{-3}	9.59×10^{-3}	1.20×10^{-2}	1.92×10^{-2}	2.64×10^{-2}
Cosmic rays (Protons Rate)	1.92×10^{-4}	2.87×10^{-4}	3.83×10^{-4}	4.79×10^{-4}	7.66×10^{-4}	1.05×10^{-3}
Heavy Ions rate (in flare)	1.81×10^{-8}	4.15×10^{-8}	4.15×10^{-8}	4.00×10^{-8}	9.24×10^{-8}	1.48×10^{-7}
Heavy Ions rate (out-of-flare)	1.81×10^{-8}	4.15×10^{-8}	4.15×10^{-8}	4.00×10^{-8}	9.24×10^{-8}	1.48×10^{-7}
Protons rate (in flare)	4.99×10^{-3}	7.48×10^{-3}	9.97×10^{-3}	1.25×10^{-2}	1.99×10^{-2}	2.74×10^{-2}
Protons rate (out-of-flare)	4.99×10^{-3}	7.48×10^{-3}	9.97×10^{-3}	1.25×10^{-2}	1.99×10^{-2}	2.74×10^{-2}
Total rate (in flare)	4.99×10^{-3}	7.48×10^{-3}	9.97×10^{-3}	1.25×10^{-2}	1.99×10^{-2}	2.74×10^{-2}
Total rate (out-of-flare)	4.99×10^{-3}	7.48×10^{-3}	9.97×10^{-3}	1.25×10^{-2}	1.99×10^{-2}	2.74×10^{-2}

Table A.16: P_0 and P_4 in Open space environment (FinFET)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	2.89×10^{-7}	4.33×10^{-7}	5.77×10^{-7}	7.23×10^{-7}	1.15×10^{-6}	1.59×10^{-6}
P_0 $N = 4Mbits$						
P_0	9.17×10^{-8}	2.06×10^{-7}	3.66×10^{-7}	5.75×10^{-7}	1.46×10^{-6}	2.76×10^{-6}
SER_0 (FIT)	3.30×10^{5}	7.42×10^{5}	1.32×10^{6}	2.07×10^{6}	5.25×10^{6}	9.94×10^{6}
P_4 **For** $n_c = 256kbits$						
P_4	5.74×10^{-15}	1.29×10^{-14}	2.29×10^{-14}	3.60×10^{-14}	9.14×10^{-14}	1.73×10^{-13}
SER_4 (FIT)	2.07×10^{-2}	4.64×10^{-2}	8.25×10^{-2}	1.30×10^{-1}	3.29×10^{-1}	6.23×10^{-1}
P_4 **For** $n_c = 512kbits$						
P_4	1.15×10^{-14}	2.58×10^{-14}	4.58×10^{-14}	7.20×10^{-14}	1.83×10^{-13}	3.46×10^{-13}
SER_4 (FIT)	4.13×10^{-2}	9.29×10^{-2}	1.65×10^{-1}	2.59×10^{-1}	6.58×10^{-1}	1.25×10^{0}

Table A.17: SEE rates (/day/bit) from OMERE in Open space orbit space environment (Bulk CMOS)

	14nm	22nm	28nm	32nm	45nm	65nm
Cosmic rays (Heavy Ions Rate)	1.18×10^{-7}	1.85×10^{-7}	2.43×10^{-7}	2.88×10^{-7}	4.07×10^{-7}	6.00×10^{-7}
Solar ions (Heavy Ions Rate)	1.46×10^{-7}	2.48×10^{-7}	2.92×10^{-7}	2.84×10^{-7}	2.72×10^{-7}	5.16×10^{-7}
Trapped protons (Protons Rate)	0	0	0	0	0	0
Solar protons (Protons Rate)	4.80×10^{-3}	7.20×10^{-3}	9.60×10^{-3}	1.20×10^{-2}	1.92×10^{-2}	2.64×10^{-2}
Cosmic rays (Protons Rate)	1.92×10^{-4}	2.87×10^{-4}	3.83×10^{-4}	4.79×10^{-4}	7.66×10^{-4}	1.05×10^{-3}
Heavy Ions rate (in flare)	2.64×10^{-7}	4.33×10^{-7}	5.35×10^{-7}	5.73×10^{-7}	6.79×10^{-7}	1.12×10^{-6}
Heavy Ions rate (out-of-flare)	2.64×10^{-7}	4.33×10^{-7}	5.35×10^{-7}	5.73×10^{-7}	6.79×10^{-7}	1.12×10^{-6}
Protons rate (in flare)	4.99×10^{-3}	7.49×10^{-3}	9.98×10^{-3}	1.25×10^{-2}	2.00×10^{-2}	2.75×10^{-2}
Protons rate (out-of-flare)	4.99×10^{-3}	7.49×10^{-3}	9.98×10^{-3}	1.25×10^{-2}	2.00×10^{-2}	2.75×10^{-2}
Total rate (in flare)	4.99×10^{-3}	7.49×10^{-3}	9.99×10^{-3}	1.25×10^{-2}	2.00×10^{-2}	2.75×10^{-2}
Total rate (out-of-flare)	4.99×10^{-3}	7.49×10^{-3}	9.99×10^{-3}	1.25×10^{-2}	2.00×10^{-2}	2.75×10^{-2}

Table A.18: P_0 and P_4 in Open space environment (CMOS Bulk)

	14nm	22nm	28nm	32nm	45nm	65nm
λ in error/bit/s	2.89×10^{-7}	4.33×10^{-7}	5.78×10^{-7}	7.23×10^{-7}	1.16×10^{-6}	1.59×10^{-6}
P_0 $N = 4Mbits$						
P_0	9.17×10^{-8}	2.06×10^{-7}	3.67×10^{-7}	5.75×10^{-7}	1.47×10^{-6}	2.78×10^{-6}
SER_0 (FIT)	3.30×10^{5}	7.43×10^{5}	1.32×10^{6}	2.07×10^{6}	5.29×10^{6}	1.00×10^{6}
P_4 **For** $n_c = 256kbits$						
P_4	5.74×10^{-15}	1.29×10^{-14}	2.30×10^{-14}	3.60×10^{-14}	9.21×10^{-14}	1.74×10^{-13}
SER_4 (FIT)	2.07×10^{-2}	4.65×10^{-2}	8.27×10^{-2}	1.30×10^{-1}	3.32×10^{-1}	6.27×10^{-1}
P_4 **For** $n_c = 512kbits$						
P_4	1.15×10^{-14}	2.59×10^{-14}	4.60×10^{-14}	7.20×10^{-14}	1.84×10^{-13}	3.48×10^{-13}
SER_4 (FIT)	4.13×10^{-2}	9.31×10^{-2}	1.65×10^{-1}	2.59×10^{-1}	6.63×10^{-1}	1.25×10^{0}

A.3 Listings

A.3.1 Hardening Manager (HM)

Listing A.1: Hardening Manager (HM)

```
#include "kernel/kernel.h"
#include "kernel/vm.h"

#include <machine/vm.h>

#include <minix/type.h>
#include <minix/syslib.h>
#include <minix/cpufeature.h>
#include <string.h>
#include <assert.h>
#include <signal.h>
#include <stdlib.h>


#include <machine/vm.h>


#include "arch_proto.h"

#include "htype.h"
#include "hproto.h"
#include "rcounter.h"
#include "mca.h"
#include "../../clock.h"

#ifdef USE_APIC
#include "apic.h"
#ifdef USE_WATCHDOG
#include "kernel/watchdog.h"
#endif
#endif
static void init_hardening_features(void);

static void init_hardening_features(void){
#if USE_MCA
    enables_all_mca_features();
    enable_loggin_ofall_errors();
    clears_all_errors();
    enable_machine_check_exception();
#endif

#if USE_INS_COUNTER
#if USE_FIX_CTR
```

```
44    intel_fixed_insn_counter_init();
45  #else
46    intel_arch_insn_counter_init();
47  #endif
48  #endif
49
50  }
51
52  void init_hardening(void){
53    struct pram_mem_block *pmb;
54    struct hardening_mem_event *hme;
55    struct hardening_shared_region *hsr;
56    struct hardening_shared_proc *hsp;
57    int i;
58    int h_enable = 0;
59    int h_proc_nr = 0;
60    int h_do_sys_call = 0;
61    int h_do_nmi = 0;
62    int hprocs_in_use = 0;
63    int h_step = 0;
64    int h_step_back = 0;
65    int vm_should_run = 0;
66    int pe_should_run = 0;
67    int h_restore = 0;
68    int h_vm_end_h_req = 0;
69    int from_exec = 0;
70    int proc_2_delay = 0;
71    int h_unstable_state = 0;
72    int id_last_inject_pe    = 0;
73    int id_current_pe = 0;
74    int h_wait_vm_reply = H_NO;
75    vir_bytes pagefault_addr_1 = 0;
76    vir_bytes pagefault_addr_2 = 0;
77    struct hardening_shared_region *all_hsr_s = NULL;
78    struct hardening_shared_proc    *all_hsp_s = NULL;
79    h_can_start_hardening = 0;
80    int n_hsps = 0;
81    int n_hsrs = 0;
82    u32_t cr0 = 0;
83    u32_t cr2 = 0;
84    u32_t cr3 = 0;
85    u32_t cr4 = 0;
86    u32_t cr0_1 = 0;
87    u32_t cr0_2 = 0;
88    u32_t cr2_1 = 0;
89    u32_t cr2_2 = 0;
90    u32_t cr3_1 = 0;
91    u32_t cr3_2 = 0;
92    u32_t cr4_1 = 0;
```

```
93    u32_t cr4_2 = 0;
94    for(i=0; i<10; i++)
95      hc_proc_nr[i] = 0;
96    for(pmb = BEG_PRAM_MEM_BLOCK_ADDR;
97      pmb < END_PRAM_MEM_BLOCK_ADDR; pmb++){
98                pmb->flags = PRAM_SLOT_FREE;
99                pmb->vaddr = 0;
100               pmb->id = 0;
101               pmb->us0 = 0;
102               pmb->us1 = 0;
103               pmb->us2 = 0;
104               pmb->next_pmb = NULL;
105         }
106
107   for(hme = BEG_HARDENING_MEM_EVENTS_ADDR;
108         hme < END_HARDENING_MEM_EVENTS_ADDR; hme++){
109               hme->flags = HME_SLOT_FREE;
110               hme->addr_base = 0;
111               hme->nbytes = 0;
112               hme->id = 0;
113               hme->npages = 0;
114               hme->next_hme = NULL;
115         }
116
117   for(hsr = BEG_HARDENING_SHARED_REGIONS_ADDR;
118       hsr < END_HARDENING_SHARED_REGIONS_ADDR; hsr++){
119             hsr->id = 0;
120             hsr->flags = HSR_SLOT_FREE;
121             hsr->r_id = 0;
122             hsr->vaddr = 0;
123             hsr->length = 0;
124             hsr->next_hsr = NULL;
125             hsr->r_hsp = NULL;
126             hsr->n_hsp = 0;
127         }
128
129   for(hsp = BEG_HARDENING_SHARED_PROCS_ADDR;
130         hsp < END_HARDENING_SHARED_PROCS_ADDR; hsp++){
131             hsp->hsp_endpoint = 0;
132             hsp->flags = HSP_SLOT_FREE;
133             hsp->id = 0;
134             hsp->next_hsp = NULL;
135       }
136
137 }
```

A.3.2 Double execution with comparison (DWC)

Listing A.2: Double execution with comparison (DWC)

```
#include "kernel/kernel.h"
#include "kernel/vm.h"

#include <machine/vm.h>

#include <minix/type.h>
#include <minix/syslib.h>
#include <minix/cpufeature.h>
#include <string.h>
#include <assert.h>
#include <signal.h>
#include <stdlib.h>

#include <machine/vm.h>
#include "arch_proto.h"

#include "htype.h"
#include "hproto.h"
#include "rcounter.h"
#include "mca.h"
#include "../../clock.h"

#ifdef USE_APIC
#include "apic.h"
#ifdef USE_WATCHDOG
#include "kernel/watchdog.h"
#endif
#endif
static void update_step(int step,
     int p_nr, const char *from);
static void save_copy_0(struct proc* p);
static void restore_copy_0(struct proc* p);
static void save_context(struct proc *p);
static void restore_copy_1(struct proc* p);
static void save_copy_2(struct proc *p);
static int cmp_mem(struct proc *p);
static int cmp_reg(struct proc *p);
static void save_copy_1(struct proc *p);

void start_dwc(struct proc *p){
   /* Added by EKA*/

  /* Here the system is switching from
   * kernel space to user space
   * -- If we are not runnning a hardened PE,
   *          if the runnable process is hardened,
   *          a new PE must be started
   *      $$$ h_enable = 1 and h_proc_nr = p->p_nr
```

```
49      *      $$$ initialize the retirement counter
50      *      $$$ turn on ON the retirement counter
51      * -- else if we are running a hardened PE
52      *           (then h_enable is true)
53      *      $$$ if the runnable process is not
54      *           hardened  or VM, panic
55      *      $$$ if the runnable process is VM, turn  OFF
56      *           the retirement counter
57      *      $$$ if the runnable process is hardened turn
58      *           on the retirement
59      *           counter.
60      * -- else an unhardened process has been scheduled,
61      *           then just do what classical minix does
62      */
63     if((h_enable == H_DISABLE)  &&
64               (p->p_nr != VM_PROC_NR) &&
65                (p->p_hflags & PROC_TO_HARD)){
66       /* start a new PE */
67       /* be sure that no process is
68        * already in the hardening execution */
69        assert(h_wait_vm_reply == H_NO);
70        assert(h_step == NO_HARD_RUN);
71       /* remember the process in the
72        * hardening execution */
73        h_proc_nr = p->p_nr;
74        /**That is the start of the 1st run**/
75        update_step(FIRST_RUN, p->p_nr,
76          "starting pe from arch_system: First run");
77        /*set the next running process frames to RO*/
78        //vm_setpt_root_to_ro(p, (u32_t *)p->p_seg.p_cr3);
79         set_pe_mem_to_ro(p, (u32_t *)p->p_seg.p_cr3);
80        /* Save the initial state of the
81         * process in the kernel*/
82         save_copy_0(p);
83 #if USE_INS_COUNTER
84           /* initialize retirement counter */
85     set_remain_ins_counter_value_0(p);
86 #endif
87               /* enable the hardening */
88               h_enable = H_ENABLE;
89 #if INJECT_FAULT
90         if(could_inject == H_YES){
91               inject_error_in_gpregs(p);
92               could_inject = H_NO;
93          }
94 #endif
95     }
96     if((h_enable == H_ENABLE)  &&
97           ( (h_restore == RESTORE_FOR_SECOND_RUN) ||
```

```
98              (h_restore == RESTORE_FOR_FISRT_RUN) ) ){
99        /* besure it is the hardening PE*/
100       assert(h_proc_nr == p->p_nr);
101       /* save the working set, the woring set
102          size list before restoring*/
103       struct pram_mem_block * pmb = p->p_lus1_us2;
104       int lus1_us2_size =
105              p->p_lus1_us2_size;
106       /* restore the initial state
107        * (context and kernel state data */
108       restore_copy_0(p);
109       /* save the working set, the woring set
110        * size list after restoring */
111       p->p_lus1_us2 = pmb;
112       p->p_lus1_us2_size =
113                lus1_us2_size;
114      /* Two possibilities of restoring
115       * 1- The Processing run correctly the first
116       *     run and we have
117       *     to continue to the second run
118       * 2- An error occurs, so we have to restore
119       *     to the previous state and restart
120       *     the first run **/
121      switch(h_restore){
122        case RESTORE_FOR_SECOND_RUN:
123          /* update the hardening step variables
124          * to 2nd run*/
125          update_step(SECOND_RUN, p->p_nr,
126               "restoring arch_system");
127          break;
128        case RESTORE_FOR_FISRT_RUN:
129          update_step(FIRST_RUN, p->p_nr,
130               "restoring arch_system");
131          break;
132        default:
133          panic("UNKOWN RESTORING STATE");
134      }
135     /* reset the hardening state variables.
136      * The restoring goes well*/
137     /**Restoring to start the second run**/
138     /* set all data pages as not accessible */
139     //vm_setpt_root_to_ro(p, (u32_t *)p->p_seg.p_cr3);
140      set_pe_mem_to_ro(p, (u32_t *)p->p_seg.p_cr3);
141      h_restore = 0;
142      /* be sure the process remain runnable*/
143      assert(proc_is_runnable(p));
144 #if INJECT_FAULT
145     if(could_inject == H_YES){
146         inject_error_in_gpregs(p);
```

```
147             could_inject = H_NO;
148           }
149 #endif
150       }
151       if(h_enable && (h_proc_nr != p->p_nr) &&
152                 (p->p_nr != VM_PROC_NR) )
153        /*Should never happen*/
154         panic("Interference in the hardening"
155                 " task by: %d", p->p_nr);
156        /* Should never happen. VM should run only
157         * when the hardened process
158         * trigger a page fault */
159        if(h_enable && (p->p_nr == VM_PROC_NR) &&
160          !RTS_ISSET(proc_addr(h_proc_nr), RTS_PAGEFAULT)&&
161          (h_step != VM_RUN))
162         panic("Le VM tente de s'exÃ©cuter "
163            "sans une page fault %d\n", h_step);
164 #if USE_INS_COUNTER
165        if((h_enable == H_ENABLE) &&
166             (p->p_nr == h_proc_nr)){
167        /* we resume the current hardened PE
168         * restore value of retirement counter
169         * saved when the system switched
170         * from the hardened PE context to
171         * kernel context */
172            set_remain_ins_counter_value_1(p);
173            enable_counter();
174         }
175         else /* a process which is not
176               * hardened has been selected by
177                     the scheduler */
178           reset_counter();
179 #endif
180         /** Ensure that when we are in step 1 or
181          ** 2 only The PE can run**/
182         if((h_enable == H_ENABLE) &&
183             ((h_step == FIRST_RUN) ||
184                 (h_step == SECOND_RUN)))
185            assert(h_proc_nr == p->p_nr);
186
187          /** when vm_should_run || h_step == VM_RUN
188            **is the turn of VM to run **/
189         if((h_enable == H_ENABLE) && ((h_step == VM_RUN)
190                 || vm_should_run))
191                  assert(VM_PROC_NR == p->p_nr);
192 /**End Added by EKA**/
193
194
195 }
```

```
196
197 /*=============================================*
198  *       hardening_task          *
199  *=============================================*/
200 void hardening_task(void){
201     /*
202      * This function is called to end the PE execution.
203      * Either the 1st execution it should start
204      *  the second execution
205      * Either the 2nd execution
206      *  it should start the comprison phase
207      * running process should be the current
208      *  hardenning process
209      * The running process should not be the VM
210      * The hardening should be enable
211      */
212 #if INJECT_FAULT
213     restore_cr_reg();
214 #endif
215     assert(h_enable);
216     struct proc  *p = get_cpulocal_var(proc_ptr);
217     assert(h_proc_nr == p->p_nr);
218     assert(h_proc_nr != VM_PROC_NR);
219     /* Should only be called during 1st or 2nd run */
220     assert((h_step == FIRST_RUN) ||
221        (h_step == SECOND_RUN));
222     save_context(p);
223     if(h_step == FIRST_RUN){
224        /* End of the 1st execution */
225        assert(h_restore==0);
226        /* remember to restore the state 0 */
227        h_restore = RESTORE_FOR_SECOND_RUN;
228        /* set the page in the working set to first_phys
229         *copy the content of us0 to us1 and the content
230         * of pram2_phys to us0*/
231        vm_reset_pram(p, (u32_t *)p->p_seg.p_cr3,
232            CPY_RAM_FIRST);
233        /* launch the second run */
234        run_proc_2(p);
235        /*not reachable go directly to
236         the hardening process */
237        NOT_REACHABLE;
238     }
239     else if(h_step == SECOND_RUN){
240           /* comparison of context1 and context 2.
241            * Here we compare the saved
242            * registers */
243           if(!cmp_reg(p) || cmp_mem(p)!=OK){
244           /* That means the system is in unstable state
```

```
245             *  because the comparison stage failed.
246             *  The solution is to cancel all
247             *  thing that fault
248             * trigger in the system. The main
249             * objective is to
250             * keep the effect of that event
251             * in the kernel.
252             * it should certainly not be spread
253             * to the rest of the system**/
254               h_unstable_state = H_UNSTABLE;
255               vm_reset_pram(p,(u32_t *)p->p_seg.p_cr3,
256                   CMP_FAILED);
257               h_restore = RESTORE_FOR_FISRT_RUN;
258               /** Prevent the PE from running**/
259               p->p_rts_flags |= RTS_UNSTABLE_STATE;
260               return;
261           }
262
263         /* copy of us2 to us0 */
264         vm_reset_pram(p,(u32_t *)p->p_seg.p_cr3,
265                 CPY_RAM_PRAM);
266
267        /* reset of hardening global variables */
268         h_enable = 0;
269         h_proc_nr = 0;
270         h_step_back = 0;
271         h_step = 0;
272         h_do_sys_call = 0;
273         h_do_nmi = 0;
274         vm_should_run = 0;
275         pe_should_run = 0;
276         h_restore = 0;
277         h_normal_pf = 0;
278         h_pf_handle_by_k = 0;
279          /** The system is in stable state**/
280         h_unstable_state = H_STABLE;
281         /** a normal page fault occur
282          * where the pqge is not in the working set**/
283
284         h_rw = 0;
285          /** Reset the instruction counter process**/
286         p->p_start_count_ins = 0;
287         id_current_pe++; // Comment TODO
288         pagefault_addr_1 = 0;
289         pagefault_addr_2 = 0;
290         return;
291     }
292     else{
293         panic("UNKOWN HARDENING STEP %d\n", h_step);
```

```
294     }
295
296 }
297
298 /*=============================================*
299  *          cmp_mem              *
300  *=============================================*/
301
302 static int cmp_mem(struct proc *p){
303     if(p->p_lus1_us2_size <= 0)
304             return(OK);
305     int r = OK;
306     struct pram_mem_block *pmb = p->p_lus1_us2;
307       while(pmb){
308        if((pmb->us2 == MAP_NONE) ||
309             (pmb->us1 == MAP_NONE)){
310              pmb = pmb->next_pmb;
311              continue;
312        }
313        if(pmb->flags & FWS){
314            if((r = cmp_frames(pmb->us2, pmb->us1))!=OK)
315               return (r);
316            pmb->flags &= ~FWS;
317        }
318        pmb = pmb->next_pmb;
319       }
320     return (r);
321 }
322
323 /*=============================================*
324  *          cmp_reg        *
325  *=============================================*/
326 static int cmp_reg(struct proc *p){
327     /** Compare registers from first run and second run
328      ** If one pair of register is different the return
329      *  value is 0
330      ** Otherwise the return value is non NULL
331      ** The function compares also the origin of the
332      *  trap, fault, or interrupt
333      ** for the two runs. If they are not the same,
334      *  the return value is 0***/
335     return (
336           (gs_1 == gs_2) &&
337    (fs_1 == fs_2) &&
338    (es_1 == es_2) &&
339    (ds_1 == ds_2) &&
340    (di_1 == di_2) &&
341    (si_1 == si_2) &&
342    (fp_1 == fp_2) &&
```

```
343     (bx_1 == bx_2) &&
344     (dx_1 == dx_2) &&
345     (cx_1 == cx_2) &&
346     (retreg_1 == retreg_2) &&
347     (pc_1 == pc_2) &&
348     (cs_1 == cs_2) &&
349     (psw_1 == psw_2) &&
350     (sp_1 == sp_2)    &&
351     (ss_1 == ss_2)    &&
352           (origin_1 == origin_2) &&
353           (eax_s1 == eax_s2) &&
354           (ebx_s1 == ebx_s2) &&
355           (ecx_s1 == ecx_s2) &&
356           (edx_s1 == edx_s2) &&
357           (esi_s1 == esi_s2) &&
358           (edi_s1 == edi_s2) &&
359           (esp_s1 == esp_s2) &&
360           (ebp_s1 == ebp_s2) &&
361           (pagefault_addr_1 == pagefault_addr_2) &&
362           (cr0_1 == cr0_2) &&
363           (cr2_1 == cr2_2) &&
364           (cr3_1 == cr3_2) &&
365           (cr4_1 == cr4_2)
366       );
367 }
368
369
370 static void save_context(struct proc *p){
371     if(h_step == FIRST_RUN)
372         save_copy_1(p);
373     if(h_step == SECOND_RUN)
374         save_copy_2(p);
375 }
376
377 /*==========================================================*
378  *         run_proc_2    *
379  *==========================================================*/
380 void run_proc_2(struct proc *p)
381 {
382     /* This the standard switch_to_user
383      * function of Minix 3 without
384      * kernel verification already performed
385      * before FIRST RUN
386      */
387     /* update the global variable "get_cpulocal_var"
388      * which is a Minix Macro
389      */
390   get_cpulocal_var(proc_ptr) = p;
391
```

```
392   switch_address_space(p);
393         /** Stop counting CPU cycle for the KERNEL**/
394   context_stop(proc_addr(KERNEL));
395
396   /* If the process isn't the owner of FPU,
397           * enable the FPU exception */
398   if(get_cpulocal_var(fpu_owner) != p)
399     enable_fpu_exception();
400   else
401     disable_fpu_exception();
402
403 #if defined(__i386__)
404     assert(p->p_seg.p_cr3 != 0);
405 #elif defined(__arm__)
406   assert(p->p_seg.p_ttbr != 0);
407 #endif
408   restart_local_timer();
409
410   /*
411    * restore_user_context() carries out the
412          * actual mode switch from kernel
413    * to userspace. This function does not return
414    */
415   restore_user_context(p);
416   NOT_REACHABLE;
417 }
418
419 /*=========================================================*
420  *    update_step                                          *
421  *=========================================================*/
422 static void update_step(int step,
423      int p_nr, const char *from){
424    /**Change the hardening step to step**/
425    h_step = step;
426 }
427
428 /*=========================================================*
429  *    save_copy_0                                          *
430  *=========================================================*/
431 static void save_copy_0(struct proc* p){
432    gs = (u16_t)p->p_reg.gs;
433    fs = (u16_t)p->p_reg.fs;
434    es = (u16_t)p->p_reg.es;
435    ds = (u16_t)p->p_reg.ds;
436    di = (reg_t)p->p_reg.di;
437    si = (reg_t)p->p_reg.si;
438    fp = (reg_t)p->p_reg.fp;
439    bx = (reg_t)p->p_reg.bx;
440    dx = (reg_t)p->p_reg.dx;
```

```
441     cx = (reg_t)p->p_reg.cx;
442     retreg = (reg_t)p->p_reg.retreg;
443     pc = (reg_t)p->p_reg.pc;
444     psw = (reg_t)p->p_reg.psw;
445     sp = (reg_t)p->p_reg.sp;
446     cs = (reg_t)p->p_reg.cs;
447     ss = (reg_t)p->p_reg.ss;
448     p_kern_trap_style =
449         (int) p->p_seg.p_kern_trap_style;
450     p_rts_flags = p->p_rts_flags;
451     cr0 = read_cr0();
452     cr2 = read_cr2();
453     cr3 = read_cr3();
454     cr4 = read_cr4();
455 }
456
457 /*=============================================*
458  *       restore_copy_0          *
459  *=============================================*/
460 static void restore_copy_0(struct proc* p){
461     p->p_reg.gs = (u16_t)gs;
462     p->p_reg.fs = (u16_t)fs;
463     p->p_reg.es = (u16_t)es;
464     p->p_reg.ds = (u16_t)ds;
465     p->p_reg.di = (reg_t)di;
466     p->p_reg.si = (reg_t)si;
467     p->p_reg.fp = (reg_t)fp;
468     p->p_reg.bx = (reg_t)bx;
469     p->p_reg.dx = (reg_t)dx;
470     p->p_reg.cx = (reg_t)cx;
471     p->p_reg.psw = (reg_t)psw;
472     p->p_reg.sp = (reg_t)sp;
473     p->p_reg.pc = (reg_t)pc;
474     p->p_reg.cs = (reg_t)cs;
475     p->p_reg.ss = (reg_t)ss;
476     p->p_reg.retreg = (reg_t)retreg;
477     p->p_seg.p_kern_trap_style =
478               (int)p_kern_trap_style;
479     p->p_rts_flags = p_rts_flags;
480     write_cr0(cr0);
481     write_cr3(cr3);
482     write_cr4(cr4);
483 }
484
485 /*=============================================*
486  *         save_copy_1   *
487  *=============================================*/
488 static void save_copy_1(struct proc *p){
489     gs_1 = p->p_reg.gs;
```

```
490     fs_1 = p->p_reg.fs;
491     es_1 = p->p_reg.es;
492     ds_1 = p->p_reg.ds;
493     di_1 = p->p_reg.di;
494     si_1 = p->p_reg.si;
495     fp_1 = p->p_reg.fp;
496     bx_1 = p->p_reg.bx;
497     dx_1 = p->p_reg.dx;
498     cx_1 = p->p_reg.cx;
499     retreg_1 = p->p_reg.retreg;
500     pc_1 = p->p_reg.pc;
501     cs_1 = p->p_reg.cs;
502     psw_1 = p->p_reg.psw;
503     sp_1 = p->p_reg.sp;
504     ss_1 = p->p_reg.ss;
505     p_kern_trap_style_1 =
506           (int) p->p_seg.p_kern_trap_style;
507     origin_1 = origin_syscall;
508     eax_s1 = eax_s;
509     ebx_s1 = ebx_s;
510     ecx_s1 = ecx_s;
511     edx_s1 = edx_s;
512     esi_s1 = esi_s;
513     edi_s1 = edi_s;
514     esp_s1 = esp_s;
515     ebp_s1 = ebp_s;
516     cr0_1  = read_cr0();
517     cr2_1  = read_cr2();
518     cr3_1  = read_cr3();
519     cr4_1 =  read_cr4();
520 }
521
522
523 /*=============================================*
524  *          save_copy_2 *
525  *=============================================*/
526 static void save_copy_2(struct proc *p){
527     gs_2 = p->p_reg.gs;
528     fs_2 = p->p_reg.fs;
529     es_2 = p->p_reg.es;
530     ds_2 = p->p_reg.ds;
531     di_2 = p->p_reg.di;
532     si_2 = p->p_reg.si;
533     fp_2 = p->p_reg.fp;
534     bx_2 = p->p_reg.bx;
535     dx_2 = p->p_reg.dx;
536     cx_2 = p->p_reg.cx;
537     retreg_2 = p->p_reg.retreg;
538     pc_2 = p->p_reg.pc;
```

```
539     cs_2 = p->p_reg.cs;
540     psw_2 = p->p_reg.psw;
541     sp_2 = p->p_reg.sp;
542     ss_2 = p->p_reg.ss;
543     p_kern_trap_style_2 =
544         (int) p->p_seg.p_kern_trap_style;
545     origin_2 = origin_syscall;
546     eax_s2 = eax_s;
547     ebx_s2 = ebx_s;
548     ecx_s2 = ecx_s;
549     edx_s2 = edx_s;
550     esi_s2 = esi_s;
551     edi_s2 = edi_s;
552     esp_s2 = esp_s;
553     ebp_s2 = ebp_s;
554     cr0_2  = read_cr0();
555     cr2_2  = read_cr2();
556     cr3_2  = read_cr3();
557     cr4_2  = read_cr4();
558 }
559
560 /*=============================================*
561  *          restore_copy_1 *
562  *=============================================*/
563 static void restore_copy_1(struct proc* p){
564   p->p_reg.gs = (u16_t)gs_1;
565   p->p_reg.fs = (u16_t)fs_1;
566   p->p_reg.es = (u16_t)es_1;
567   p->p_reg.ds = (u16_t)ds_1;
568   p->p_reg.di = (reg_t)di_1;
569   p->p_reg.si = (reg_t)si_1;
570   p->p_reg.fp = (reg_t)fp_1;
571   p->p_reg.bx = (reg_t)bx_1;
572   p->p_reg.dx = (reg_t)dx_1;
573   p->p_reg.cx = (reg_t)cx_1;
574   p->p_reg.retreg = (reg_t)retreg_1;
575   p->p_reg.pc = (reg_t)pc_1;
576   p->p_reg.cs = (reg_t)cs_1;
577   p->p_reg.psw = (reg_t)psw_1;
578   p->p_reg.sp = (reg_t)sp_1;
579   p->p_reg.ss = (reg_t)ss_1;
580   p->p_seg.p_kern_trap_style =
581         (int)p_kern_trap_style_1;
582   write_cr0(cr0_1);
583   write_cr3(cr3_1);
584   write_cr4(cr4_1);
585 }
```

A.3.3 Protected memory (PRAM)

Listing A.3: Protected memory (PRAM)

```
#include "kernel/kernel.h"
#include "kernel/vm.h"

#include <machine/vm.h>

#include <minix/type.h>
#include <minix/syslib.h>
#include <minix/cpufeature.h>
#include <string.h>
#include <assert.h>
#include <signal.h>
#include <stdlib.h>


#include <machine/vm.h>


#include "arch_proto.h"

#include "htype.h"
#include "hproto.h"
#include "rcounter.h"
#include "mca.h"
#include "../../clock.h"

#ifdef USE_APIC
#include "apic.h"
#ifdef USE_WATCHDOG
#include "kernel/watchdog.h"
#endif
#endif

static int
 look_up_unique_pte(struct proc *current_p,
     int pde, int pte);
static struct pram_mem_block * add_pmb(struct proc *p,
      phys_bytes pfa,
      vir_bytes v, int pte);
static int check_pt_entry(u32_t pte_v);
static struct pram_mem_block *
   add_pmb_vaddr(struct proc *p, phys_bytes pfa,
   vir_bytes vaddr, phys_bytes us1, phys_bytes us2);
static void vm_setpt_to_ro(struct proc *p,
   u32_t *pagetable, const u32_t v);
/*===========================================================*
```

```
46  *     check_vaddr_2                    *
47  *=======================================*/
48
49  int check_vaddr_2(struct proc *p,
50      u32_t *root, vir_bytes vaddr, int *rw){
51  /* A page fault occured. That function
52   * check if the virtual addresse match
53   * with a page entry whose access was
54   * restricted by the  hardened code.
55   * If it is the case, the access is allowed
56   * and  the VM is not called when
57   * the corresponding frame in US1 and US2 exist.
58   * Otherwise the VM is called to
59   * allocate the frame for US1 or US2
60   * if the virtual address does not match any
61   * page in the current working set
62   * that will end the PE. the VM will be called
63   * to handle the  normal page fault
64   * When the corresponding frames in
65   * US1 and US2 exist, the pmb is labelled to
66   * remember that a page fault was occured in that
67   * page. That is used at the
68   * beginning of the fisrt or the second run to
69   * set these page to read-write.
70   * The addresses where the page fault occured is
71   * store for the first and the s
72   * second run. That is used during the comparison step.
73   **/
74     int pde, pte;
75     u32_t pde_v, *pte_a, pte_v;
76     int nblocks;
77     static int cnpf = 0;
78     struct pram_mem_block *pmb, *next_pmb;
79     assert((h_step == FIRST_RUN) ||
80           (h_step == SECOND_RUN));
81     /* read the pde entry and the
82      * pte entry of the PF page */
83     pde = I386_VM_PDE(vaddr);
84     pte = I386_VM_PTE(vaddr);
85     if(h_step == FIRST_RUN)
86        pagefault_addr_1 = vaddr;
87     if(h_step == SECOND_RUN)
88        pagefault_addr_2 = vaddr;
89     /* read the page directory entry value of
90      *the page table related to the virtual
91      * address */
92     pde_v = phys_get32((u32_t) (root + pde));
93     /* check if the pde entry is present ,
94      * writable, not user, global
```

```
95          * bigpage . Let the VM handle */
96         if(check_pt_entry(pde_v)!=OK)
97             return(VM_HANDLED_NO_ACESS);
98         /**read the page table address**/
99         pte_a = (u32_t *) I386_VM_PFA(pde_v);
100        /* read the page table entry value */
101        pte_v = phys_get32((u32_t) (pte_a + pte));
102        /* check the presence. If not present,
103         * let the VM handle it*/
104        if(!(pte_v & I386_VM_PRESENT))
105            return(VM_HANDLED_NO_PRESENT);
106        /* be sure that it is a modified page table entry
107         * otherwise let the VM handle it*/
108        /* read the current frame value*/
109        u32_t pfa = I386_VM_PFA(pte_v);
110        /* check if the page  is already in
111         * the working set*/
112        if(!(pmb = look_up_pte(p, pde, pte))){
113           /* remember we are working with a page
114            * already in the working set*/
115           /* the page is not in the working,
116            * that should never happen */
117           return(VM_HANDLED_PF_ABS_WS);
118        }
119    #if CHECK_DEBUG
120        /* be sure we have a valid data structure */
121        assert(pmb);
122        /* be sure we have a valid virtual address
123         * and a valid ram_phys*/
124        assert(pmb->us0!=MAP_NONE);
125        assert(pmb->us0 == pfa);
126        assert(pmb->vaddr!=MAP_NONE);
127    #endif
128        if((pmb->us1!=MAP_NONE) &&
129           (pmb->us2!=MAP_NONE)){
130           if(h_step == FIRST_RUN){
131              pte_v = (pte_v & I386_VM_ADDR_MASK_INV) |
132                    I386_VM_WRITE | pmb->us1;
133              pmb->flags |= IWS_PF_FIRST;
134           }
135           else  if(h_step == SECOND_RUN){
136              pte_v = (pte_v & I386_VM_ADDR_MASK_INV) |
137                    I386_VM_WRITE | pmb->us2;
138              pmb->flags |= IWS_PF_SECOND;
139           }
140           phys_set32((u32_t) (pte_a + pte), &pte_v);
141           *rw = K_HANDLE_PF;
142           pmb->flags |= HGET_PF;
143           return(OK);
```

```
144     }else if(h_step == FIRST_RUN){
145         /* VM_HR1PAGEFAULT:
146          * Tell the VM it is a Read-Only page*/
147             *rw = RO_PAGE_FIRST_RUN;
148             pmb->flags |= IWS_PF_FIRST;
149         }
150     else if(h_step == SECOND_RUN) {
151 #if CHECK_DEBUG
152         assert(pmb->us0!=MAP_NONE);
153         assert(pmb->us1!=MAP_NONE);
154         assert(pmb->us2!=MAP_NONE);
155 #endif
156         *rw = RO_PAGE_SECOND_RUN;
157         pmb->flags |= IWS_PF_SECOND;
158      }
159 
160     /*It is the first page fault on
161      * that page. Tell the VM to allocate 3 new
162      * frames pram2_phys, us1 and us2 */
163     pmb->flags |= PRAM_LAST_PF;
164     pmb->flags |= HGET_PF;
165     /* be sure that the page is not insert
166      * more than once in the working set*/
167     nblocks = look_up_unique_pte(p,
168       I386_VM_PDE(vaddr),I386_VM_PTE(vaddr) );
169     assert(nblocks <= 1);
170     return(OK);
171 }
172 
173 /*=========================================*
174  *     look_up_pte                         *
175  *=========================================*/
176 struct pram_mem_block * look_up_pte(
177         struct proc *current_p, int pde, int pte){
178     if(current_p->p_lus1_us2_size > 0){
179         struct pram_mem_block *pmb =
180               current_p->p_lus1_us2;
181         int __pte, __pde;
182         while(pmb){
183             __pte = I386_VM_PTE(pmb->vaddr);
184             __pde = I386_VM_PDE(pmb->vaddr);
185             if((__pte==pte)&&(__pde==pde))
186       return(pmb);
187             pmb = pmb->next_pmb;
188         }
189     }
190     return(NULL);
191 }
192 
```

```
193 /*=============================================*
194  *       look_up_unique_pte                    *
195  *=============================================*/
196 static int look_up_unique_pte(
197    struct proc *current_p, int pde, int pte){
198   int npte;
199   if(current_p->p_lus1_us2_size > 0){
200      struct pram_mem_block *pmb = current_p->p_lus1_us2;
201      int __pte, __pde;
202      npte = 0;
203      while(pmb){
204        __pte = I386_VM_PTE(pmb->vaddr);
205        __pde = I386_VM_PDE(pmb->vaddr);
206        if((__pte==pte)&&(__pde==pde))
207    npte++;
208         pmb = pmb->next_pmb;
209      }
210    }
211    return(npte);
212 }
213
214 /*=============================================*
215  *         look_up_lastpf_pte     *
216  *=============================================*/
217 struct pram_mem_block *
218  look_up_lastpf_pte(struct proc *current_p, int normal){
219   if(current_p->p_lus1_us2_size > 0){
220     struct pram_mem_block *pmb =
221                       current_p->p_lus1_us2;
222     while(pmb){
223
224       if(pmb->flags & PRAM_LAST_PF){
225                             pmb->flags &= ~PRAM_LAST_PF;
226                 return(pmb);
227                             }
228       pmb = pmb->next_pmb;
229     }
230   }
231   return(NULL);
232 }
233
234 /*=============================================*
235  *         free_pram_mem_blocks *
236  *=============================================*/
237 void free_pram_mem_blocks(struct proc *current_p,
238           int no_msg_to_vm){
239      /** Delete of the blokcs in the PE working
240       *  set list. If the working
241       ** set list is empty the function return.
```

```
242         *  Otherwise each blocks is delete
243         ** A the end of the function the working set
244         *  of the PE should be empty ***/
245        if(current_p->p_lus1_us2_size <= 0)
246            return;
247        struct pram_mem_block *pmb = current_p->p_lus1_us2;
248        while(pmb){
249            if((pmb->flags & H_TO_UPDATE) &&
250                    (no_msg_to_vm == FROM_EXEC)){
251                pmb = pmb->next_pmb;
252                continue;
253            }
254            free_pram_mem_block(current_p,
255                    pmb->vaddr, (u32_t *)current_p->p_seg.p_cr3);
256    pmb = pmb->next_pmb;
257        }
258        if(no_msg_to_vm != FROM_EXEC){
259            assert(current_p->p_lus1_us2_size == 0);
260            assert(current_p->p_lus1_us2 == NULL);
261        }
262 }
263
264 /*=========================================================*
265  *     free_pram_mem_blocks                *
266  *=========================================================*/
267 void free_pram_mem_block_vaddr(struct proc *rp,
268        vir_bytes raddr, int len){
269     if(rp->p_lus1_us2_size <= 0)
270       return;
271     int p;
272     int pde = I386_VM_PDE(raddr);
273     int pte = I386_VM_PTE(raddr);
274     vir_bytes page_base =
275             pde * I386_VM_PT_ENTRIES * I386_PAGE_SIZE
276                                 + pte * I386_PAGE_SIZE;
277     vir_bytes vaddr;
278     int n_pages_covered = (len +
279          (raddr - page_base) - 1)/(4*1024)  + 1;
280     struct pram_mem_block *pmb ;
281     for(p = 0; p < n_pages_covered; p++){
282       vaddr = page_base + p*4*1024;
283       pde = I386_VM_PDE(vaddr);
284       pte = I386_VM_PTE(vaddr);
285       if(!(pmb =  look_up_pte(rp, pde, pte)))
286           continue;
287       free_pram_mem_block(rp,
288                pmb->vaddr, (u32_t *)rp->p_seg.p_cr3);
289     }
290 }
```

```
291
292 /*=============================================================*
293  *     free_pram_mem_block                                     *
294  *=============================================================*/
295 void free_pram_mem_block(struct proc *current_p,
296     vir_bytes vaddr, u32_t *root){
297    /* Delete the block corresponding to vaddr
298     * from the PE woking set in
299     * VM address space. If the block is found
300     * it is remove from the list
301     * and the data structure in the whole
302     * available blocks is made free
303     * If the end of the list is reached
304     * the function return **/
305    if(current_p->p_lus1_us2_size <= 0)
306       return;
307    struct pram_mem_block *pmb =
308              current_p->p_lus1_us2;
309    struct pram_mem_block *prev_pmb = NULL;
310    struct pram_mem_block *next_pmb = NULL;
311    int __pte, __pde, pte, pde;
312    u32_t pde_v, *pte_a, pte_v;
313    pte = I386_VM_PTE(vaddr);
314    pde = I386_VM_PDE(vaddr);
315    while(pmb){
316       __pte = I386_VM_PTE(pmb->vaddr);
317       __pde = I386_VM_PDE(pmb->vaddr);
318       if((__pte==pte)&&(__pde==pde)){
319          pde_v = phys_get32((u32_t) (root + pde));
320 #if CHECK_DEBUG
321          assert((pde_v & I386_VM_PRESENT));
322          assert((pde_v & I386_VM_WRITE));
323          assert((pde_v & I386_VM_USER));
324          assert(!(pde_v & I386_VM_GLOBAL));
325          assert(!(pde_v & I386_VM_BIGPAGE));
326          pte_a = (u32_t *) I386_VM_PFA(pde_v);
327          pte_v = phys_get32((u32_t) (pte_a + pte));
328          assert((pte_v & I386_VM_PRESENT));
329 #endif
330          pmb->flags = PRAM_SLOT_FREE;
331          pmb->vaddr = MAP_NONE;
332          pmb->id    = 0;
333          pmb->us0 = MAP_NONE;
334          pmb->us1 = MAP_NONE;
335          pmb->us2 = MAP_NONE;
336          current_p->p_lus1_us2_size--;
337          if(prev_pmb)
338            prev_pmb->next_pmb = pmb->next_pmb;
339          else
```

```
340             current_p->p_lus1_us2 = pmb->next_pmb;
341           break;
342       }
343       prev_pmb = pmb;
344       next_pmb = pmb->next_pmb;
345       pmb = pmb->next_pmb;
346   }
347 }
348
349
350 /*=============================================================*
351  *          vm_reset_pram   *
352  *=============================================================*/
353
354 void vm_reset_pram(struct proc *p,
355     u32_t *root, int endcmp){
356    /* This function is called at the end of
357     * the PE, first or second run
358     * It restore the PE memory space to US0
359     * It copy the content of each frame
360     * from the second (US0) run to
361     * the corresponding frame in the US0 when
362     * the comparison succeeded
363     *
364     */
365    assert(p->p_nr != VM_PROC_NR);
366    assert((h_step == FIRST_RUN) ||
367           (h_step == SECOND_RUN));
368    if(p->p_lus1_us2_size <= 0)
369       return;
370    struct pram_mem_block *pmb = p->p_lus1_us2;
371       /** Go through the working set list**/
372    while(pmb){
373 #if CHECK_DEBUG
374       assert(pmb->us0!=MAP_NONE);
375       assert(pmb->us1!=MAP_NONE);
376       assert(pmb->us2!=MAP_NONE);
377       if((pmb->us0 == MAP_NONE) ||
378            (pmb->us1 == MAP_NONE) ||
379          (pmb->us2 == MAP_NONE)){
380          pmb = pmb->next_pmb;
381          continue;
382     }
383 #endif
384    /** Yes a page fault occu get the pte and the pde**/
385    int pde, pte;
386    u32_t pde_v, *pte_a, pte_v;
387    pde = I386_VM_PDE(pmb->vaddr);
388    pte = I386_VM_PTE(pmb->vaddr);
```

```
389     /** Read the page directory value entry**/
390     pde_v = phys_get32((u32_t) (root + pde));
391 #if CHECK_DEBUG
392     /**Check PRESENT, WRITE, USER, GLOBAL and BIGPAGE**/
393     assert((pde_v & I386_VM_PRESENT));
394     assert((pde_v & I386_VM_WRITE));
395     assert((pde_v & I386_VM_USER));
396     assert(!(pde_v & I386_VM_GLOBAL));
397     assert(!(pde_v & I386_VM_BIGPAGE));
398 #endif
399      /** Read the page table addresse**/
400     pte_a = (u32_t *) I386_VM_PFA(pde_v);
401     /** Read the page table entry value**/
402     pte_v = phys_get32((u32_t) (pte_a + pte));
403 #if CHECK_DEBUG
404     /** Verify the PRESENCE**/
405     assert((pte_v & I386_VM_PRESENT));
406 #endif
407     /** Read the frame address**/
408     u32_t pfa = I386_VM_PFA(pte_v);
409
410     if((h_step == FIRST_RUN) &&
411          (endcmp == CPY_RAM_FIRST)){
412        /** That is the end of the first
413         *  run let's map the page to PRAM as
414         ** RW so before the starting of the PE
415         *  the page will be set RO
416         ** During the second RUN the PE will make
417         * the same page fault
418         **/
419        if(!(pte_v & I386_VM_DIRTY) &&
420              (pmb->flags & IWS_PF_FIRST)){
421               pmb->flags &= ~IWS_PF_FIRST;
422        }
423        if((pmb->us1!=MAP_NONE) &&
424             (pmb->us0 !=MAP_NONE) &&
425             ((pmb->flags & HGET_PF)
426             || (pte_v & I386_VM_DIRTY))){
427         if((pmb->flags & IWS_MOD_KERNEL) &&
428                   (pte_v & I386_VM_DIRTY) )
429                      pmb->flags |= IWS_PF_FIRST;
430         pte_v &= ~I386_VM_DIRTY;
431         if(phys_set32((u32_t) (pte_a + pte),
432                   &pte_v)!=OK)
433              panic("Updating page table"
434                   " from second_phy to us0"
435                            "failed\n");
436         pmb->flags &= ~HGET_PF;
437         pmb->flags |= FWS;
```

```
438         }
439       }
440       if(h_step == SECOND_RUN) {
441          switch(endcmp){
442             case CPY_RAM_PRAM:
443                /* The comparison succed let's
444                 * copy the content of
445                 * us2 to us0 and map the page to us0
446                 * Before the starting of the PE all
447                 * writable pages are set to
448                 * RO so we have to put now the page
449                 * as RW thus it will be
450                 * put to RO before starting the PE.
451                 * Otherwise it will be
452                 * ignored **/
453                if(!(pte_v & I386_VM_DIRTY) &&
454                   (pmb->flags & IWS_PF_SECOND)){
455                    pmb->flags &= ~IWS_PF_SECOND;
456                }
457
458                if((pmb->us2!=MAP_NONE) &&
459                    (pmb->us0!=MAP_NONE) &&
460                  ((pmb->flags & HGET_PF)
461                        || (pte_v & I386_VM_DIRTY))){
462
463                   if((pmb->flags & IWS_MOD_KERNEL) &&
464                         (pte_v & I386_VM_DIRTY) )
465                      pmb->flags |= IWS_PF_SECOND;
466
467                   if(cpy_frames(pmb->us2,
468                         pmb->us0)!=OK)
469                      panic("Copy second_phys to us0 "
470                            "failed\n");
471                   if(pmb->flags & WS_SHARED)
472                      enable_hme_event_in_procs(p,
473                      pmb->vaddr, I386_PAGE_SIZE );
474                   pmb->flags &= ~HGET_PF;
475                   pte_v &= ~I386_VM_DIRTY;
476                }
477
478                pte_v = (pte_v & I386_VM_ADDR_MASK_INV) |
479                            I386_VM_WRITE | pmb->us0;
480                if(phys_set32((u32_t) (pte_a + pte),
481                   &pte_v)!=OK)
482                   panic("Updating page table from"
483                      "second_phy to us0 failed\n");
484
485                if(pmb->flags & IWS_MOD_KERNEL){
486                    pmb->flags &= ~IWS_MOD_KERNEL;
```

```
487                 }
488                 break;
489             case CMP_FAILED:
490                 /** The comparison failed let's
491                  ** copy map the page to PRAM
492                  ** and set it to RW. Thus before
493                  ** the starting of the PE
494                  ** the page will be set to RO.
495                  ** The PE state is restored
496                  ** to US0
497                  */
498
499                 pmb->flags &= ~IWS_PF_SECOND;
500                 pmb->flags &= ~IWS_PF_FIRST;
501                 pmb->flags &= ~HGET_PF;
502                 pmb->flags &= ~IWS_MOD_KERNEL;
503
504                 if(((pmb->us1!=MAP_NONE) ||
505                     (pmb->us2!=MAP_NONE)) &&
506                      (pmb->us0!=MAP_NONE)){
507                    if(pmb->us1!=MAP_NONE){
508                      if(cpy_frames(pmb->us0, pmb->us1)!=OK)
509                         panic("Copy second_phys to"
510                                 " us0 failed\n");
511                     }
512                     if(pmb->us2!=MAP_NONE){
513                     if(cpy_frames(pmb->us0, pmb->us2)!=OK)
514                             panic("Copy second_phys"
515                                " to us0 failed\n");
516                     }
517
518                     pte_v =
519                        (pte_v & I386_VM_ADDR_MASK_INV) |
520                             I386_VM_WRITE | pmb->us0;
521                     if(phys_set32((u32_t) (pte_a + pte),
522                          &pte_v)!=OK)
523                      panic("Updating page table from "
524                           "second_phy to us0" "failed\n");
525                 }
526                 break;
527             default :
528                 /**TO COMPLETE**/
529                 panic("Should never happen");
530                 break;
531         }
532       }
533     pmb = pmb->next_pmb;
534   }
535   refresh_tlb();
```

```
536 }
537
538
539 static struct pram_mem_block * add_pmb(struct proc *p,
540       phys_bytes pfa,
541       vir_bytes v, int pte){
542   struct pram_mem_block *pmb, *next_pmb;
543   /* ask for a new pmb block in the free list*/
544   pmb = get_pmb();
545   /* be sure that we got a good block*/
546   assert(pmb);
547   pmb->us0 =  pfa;
548   pmb->vaddr     =  v + pte * I386_PAGE_SIZE;
549   if(!pfa) pmb->flags |= H_HAS_NULL_PRAM;
550   /* Insert the block on the process's linked list*/
551   if(!p->p_lus1_us2_size) p->p_lus1_us2 = pmb;
552   else{
553     next_pmb = p->p_lus1_us2;
554     while(next_pmb->next_pmb) next_pmb = next_pmb->next_pmb;
555     next_pmb->next_pmb = pmb;
556     next_pmb->next_pmb->next_pmb = NULL;
557   }
558   pmb->id = p->p_lus1_us2_size;
559   p->p_lus1_us2_size++;
560   return(pmb);
561 }
562
563 static int check_pt_entry(u32_t pte_v){
564    /* check if the page is present*/
565    if(!(pte_v & I386_VM_PRESENT))
566       return(VM_HANDLED_NO_PRESENT);
567    /* Check if it is a NON USER PAGE*/
568    if(!(pte_v & I386_VM_USER))
569       return(VM_HANDLED_NO_ACESS);
570    /* Check if it is a GLOBAL_PAGE
571     *(buffer shared between os and process)*/
572    if((pte_v & I386_VM_GLOBAL))
573       return( VM_HANDLED_GLOBAL);
574    /* Check if it is a NON BIG_PAGE
575    ( BIGPAGES are not implemented in Minix)*/
576    if((pte_v & I386_VM_BIGPAGE))
577       return(VM_HANDLED_BIG);
578    /* Check if it is a WRITE*/
579    if(!(pte_v & I386_VM_WRITE))
580       return(VM_HANDLED_NO_ACESS);
581    return(OK);
582 }
583
584 /*=========================================================*
```

```
585  *          display_mem *
586  *=============================================================*/
587 void display_mem(struct proc *current_p){
588     if(current_p->p_lus1_us2_size > 0){
589    struct pram_mem_block *pmb =
590                current_p->p_lus1_us2;
591    while(pmb){
592              if(pmb->us0!=MAP_NONE)
593      printf("displaying vaddr 0x%lx   "
594                "pram 0x%lx first 0x%lx second: 0x%lx\n",
595                      pmb->vaddr, pmb->us0,
596                      pmb->us1, pmb->us2);
597        pmb = pmb->next_pmb;
598    }
599      }
600 }
601
602 /*=============================================================*
603  *          get_pmb          *
604  *=============================================================*/
605 /*    Return the first avalaible pmb in lus1_us2 table.
606  **   If the table is full a panic is triggered
607  **   otherwise the found pmb is returnedtable */
608 struct pram_mem_block *get_pmb(void){
609     int pmb_offset = 0;
610     /* start from the first pmb */
611     struct pram_mem_block *pmb = BEG_PRAM_MEM_BLOCK_ADDR;
612     /**If the first block is free return it**/
613    if(pmb->flags & PRAM_SLOT_FREE) {
614                /**Reset the data in the
615                 * block and return the block**/
616        pmb->flags &=~PRAM_SLOT_FREE;
617                pmb->next_pmb = NULL;
618                pmb->us1 = MAP_NONE;
619                pmb->us2 = MAP_NONE;
620                pmb->us0 = MAP_NONE;
621                pmb->vaddr = MAP_NONE;
622        return pmb;
623    }
624    do{
625       /*** Otherwise go through the lus1_us2
626        * and search the first available
627        *** bloc  ***/
628      pmb_offset++;
629      pmb++;
630    }while(!(pmb->flags & PRAM_SLOT_FREE) &&
631              pmb < END_PRAM_MEM_BLOCK_ADDR);
632
633     /**The end of the lus1_us2 is reached panic **/
```

```
634         if(pmb_offset>= WORKING_SET_SIZE)
635       panic("ALERT BLOCK LIST IS "
636             "FULL STOP STOP %d\n", pmb_offset);
637
638         /**The bloc is found, reset the content
639         * of the bloc and return it**/
640   pmb->flags &=~PRAM_SLOT_FREE;
641         pmb->us1 = MAP_NONE;
642         pmb->us2 = MAP_NONE;
643         pmb->us0 = MAP_NONE;
644         pmb->vaddr = MAP_NONE;
645         pmb->next_pmb = NULL;
646   return pmb;
647 }
648
649 int add_region_to_ws(struct proc *p, u32_t *root,
650           vir_bytes r_base_addr,
651           int length, phys_bytes us1,
652           phys_bytes us2){
653    u32_t pde_v, pte_v; // pde entry value
654    u32_t *pte_a; // page table address
655    u32_t pfa;
656    int i;
657    int pde = I386_VM_PDE(r_base_addr);
658    int pte = I386_VM_PTE(r_base_addr);
659    vir_bytes page_base =
660         pde * I386_VM_PT_ENTRIES * I386_PAGE_SIZE
661                                  + pte * I386_PAGE_SIZE;
662    vir_bytes vaddr;
663    int n_pages_covered =
664       (length +
665       (r_base_addr - page_base) - 1)/(4*1024)  + 1;
666    for(i = 0; i < n_pages_covered; i++ ){
667      vaddr = page_base + i*4*1024;
668      pde = I386_VM_PDE(vaddr);
669      pte = I386_VM_PTE(vaddr);
670      pde_v = phys_get32((u32_t) (root + pde));
671      /*read the page table address*/
672      pte_a = (u32_t *) I386_VM_PFA(pde_v);
673      /* read the page table entry value*/
674      pte_v = phys_get32((u32_t) (pte_a + pte));
675      /* read the frame address value*/
676      pfa = I386_VM_PFA(pte_v);
677      struct pram_mem_block *pmb;
678       if(!(pmb = look_up_pte(p, I386_VM_PDE(vaddr),
679            I386_VM_PTE(vaddr) )))
680          pmb = add_pmb_vaddr(p, pfa, vaddr,us1,us2);
681       else
682          pmb->us0 = pfa;
```

```
683         pmb->flags |= H_TO_UPDATE;
684
685       }
686       return(OK);
687   }
688
689   static struct pram_mem_block * add_pmb_vaddr(
690         struct proc *p,
691         phys_bytes pfa, vir_bytes vaddr,
692         phys_bytes us1, phys_bytes us2){
693     struct pram_mem_block *pmb, *next_pmb;
694     /* ask for a new pmb block in the free list*/
695     pmb = get_pmb();
696     /* be sure that we got a good block*/
697     assert(pmb);
698     pmb->us0 =  pfa;
699     pmb->vaddr      =  vaddr;
700     pmb->us1 = us1;
701     pmb->us2 = us2;
702     /* Insert the block on the process's linked list*/
703     if(!p->p_lus1_us2_size)
704         p->p_lus1_us2 = pmb;
705     else{
706       next_pmb = p->p_lus1_us2;
707       while(next_pmb->next_pmb)
708           next_pmb = next_pmb->next_pmb;
709       next_pmb->next_pmb = pmb;
710       next_pmb->next_pmb->next_pmb = NULL;
711     }
712     pmb->id = p->p_lus1_us2_size;
713     p->p_lus1_us2_size++;
714     return(pmb);
715   }
716
717
718   int set_pe_mem_to_ro(struct proc *p, u32_t *root){
719      int pte, pde;
720      u32_t pde_v, pte_v, pfa; // pde entry value
721      u32_t *pte_a; // page table address
722      if(p->p_lus1_us2_size <= 0)
723           return(OK);
724      if(h_step == FIRST_RUN){
725         /* Whether US0 was modified
726          * during system call handling, US1 and
727          * US2 should be updated accordingly.*/
728          if(handle_hme_events(p)!=OK)
729              panic("vm_setpt_root_to_ro: "
730                "handle_hme_events failed\n");
731          free_hardening_mem_events(p);
```

```
732     }
733     struct pram_mem_block *pmb = p->p_lus1_us2;
734     while(pmb){
735         pde = I386_VM_PDE(pmb->vaddr);
736         pte = I386_VM_PTE(pmb->vaddr);
737         pde_v = phys_get32((u32_t) (root + pde));
738         // check if the pde is present
739         if(check_pt_entry(pde_v)!=OK){
740             pmb = pmb->next_pmb;
741             continue;
742         }
743         // read the page table address
744         pte_a = (u32_t *) I386_VM_PFA(pde_v);
745          /* read the page table entry value*/
746         pte_v = phys_get32((u32_t) (pte_a + pte));
747         if(check_pt_entry(pte_v)!=OK){
748             pmb = pmb->next_pmb;
749             continue;
750         }
751         pfa = I386_VM_PFA(pte_v);
752         if((p->p_hflags & PROC_SHARING_MEM) &&
753                         !(pmb->flags & WS_SHARED) &&
754             ((look_up_page_in_hsr(p, pmb->vaddr))==OK)){
755             pmb->flags |= WS_SHARED;
756
757         }
758         if((pmb->flags & H_TO_UPDATE) &&
759             (pmb->us1!= MAP_NONE) &&
760             (pmb->us2!=MAP_NONE) ){
761           if(cpy_frames(pmb->us0, pmb->us1)!=OK)
762                   panic("add_region_to_ws:  "
763                   "first_phys failed\n");
764            if(cpy_frames(pmb->us0, pmb->us2)!=OK)
765                     panic("add_region_to_ws "
766                   " second_phys failed\n");
767            pmb->flags &= ~H_TO_UPDATE;
768
769         }
770         /* disable the WRITE bit*/
771         if(h_step == FIRST_RUN){
772             if(!pfa){
773                 free_pram_mem_block(p, pmb->vaddr,
774                       (u32_t *)root);
775                 pmb = pmb->next_pmb;
776                 continue;
777             }
778             if(pmb->us1!=MAP_NONE){
779                 pte_v =
780                  (pte_v & I386_VM_ADDR_MASK_INV) | pmb->us1;
```

```
781                 if(!(pmb->flags & IWS_PF_FIRST) &&
782                    !(pmb->flags & IWS_MOD_KERNEL) )
783                       pte_v &= ~I386_VM_WRITE;
784              }
785              else
786              pte_v &= ~I386_VM_WRITE;
787             }
788             else{
789               if(pmb->us2!=MAP_NONE){
790                 pte_v =
791                  (pte_v & I386_VM_ADDR_MASK_INV) | pmb->us2;
792                 if(!(pmb->flags & IWS_PF_SECOND) &&
793                    !(pmb->flags & IWS_MOD_KERNEL) )
794                     pte_v &= ~I386_VM_WRITE;
795               }
796               else
797                  pte_v &= ~I386_VM_WRITE;
798              }
799
800              if((pmb->us1!=MAP_NONE) &&
801                   (pmb->us2==MAP_NONE))
802                 pte_v =
803                  (pte_v & I386_VM_ADDR_MASK_INV) | pmb->us0;
804              pte_v &= ~I386_VM_DIRTY;
805              /* update the page table*/
806              phys_set32((u32_t) (pte_a + pte), &pte_v);
807             pmb = pmb->next_pmb;
808        }
809        return(OK);
810    }
811
812
813    /*=========================================================*
814     *       vm_setpt_root_to_ro *
815     *=========================================================*/
816    /** Browse the page table from page
817     *  directory O to page directory
818     ** I386_VM_DIR_ENTRIES
819     ** for each page directory entry the page
820     *  directory is not considered
821     ** if the page directory
822     **    ** Is not present
823     **    ** has not write access
824     **    ** Is not accessible in USER mode
825     **    ** Is a global pde
826     **    ** Is a big page
827     ** The page table of each pde is browse and
828     *  each page table entry
829     ** is considered to restrict access to the
```

```
830  *  processing element by calling the
831  ** static void vm_setpt_to_ro(struct proc *
832  *  , u32_t *, const u32_t ) **/
833 void vm_setpt_root_to_ro(struct proc *p, u32_t *root){
834
835     int pde; // page directory entry
836     assert(!((u32_t) root % I386_PAGE_SIZE));
837     assert(p->p_nr != VM_PROC_NR); // VM is not included
838     if(h_step == FIRST_RUN){
839       /** Whether US0 was modified during
840        * system call handling, US1 and
841        ** US2 should be updated accordingly.*/
842        if(handle_hme_events(p)!=OK)
843            panic("vm_setpt_root_to_ro: "
844                "handle_hme_events failed\n");
845        free_hardening_mem_events(p);
846     }
847     for(pde = 0; pde < I386_VM_DIR_ENTRIES; pde++) {
848        // start from 0 to I386_VM_DIR_ENTRIES = 1024
849        u32_t pde_v; // pde entry value
850        u32_t *pte_a; // page table address
851        // read the pde entry value
852        pde_v = phys_get32((u32_t) (root + pde));
853        // check if the pde is present
854        if(check_pt_entry(pde_v)!=OK) continue;
855        // read the page table address
856        pte_a = (u32_t *) I386_VM_PFA(pde_v);
857        /* call the function to handle the
858         * page table entries */
859        vm_setpt_to_ro(p, pte_a,
860            pde * I386_VM_PT_ENTRIES * I386_PAGE_SIZE);
861     }
862     return;
863 }
864
865 /*=========================================================*
866  *         vm_setpt_to_ro  *
867  *=========================================================*/
868
869
870 static void vm_setpt_to_ro(struct proc *p,
871        u32_t *pagetable, const u32_t v) {
872      /** Browse the page table entry from
873       * page table entry O to page directory
874       ** I386_VM_PT_ENTRIES
875       ** for each page table entry the page table is
876       *  not considered
877       ** if the page table entry
878       **    ** Is not present
```

```
879     **   ** Is not accessible in USER mode
880     **   ** the page does not map to a frame
881     ** for each page the virtual address and the
882     *  physical address are stored in a
883     ** linked list embedded in the process
884     * data structure
885     ** The page access is modified from USER mode
886     * to Kernel mode
887     ** So when a page fault occur we are able to know
888     *  if it is a normal page fault
889     ** or a page fault from that access modification.
890     **/
891     int pte; /* page table entry number*/
892     assert(!((u32_t) pagetable % I386_PAGE_SIZE));
893     /* VM is not included*/
894     assert(p->p_nr != VM_PROC_NR);
895     for(pte = 0; pte < I386_VM_PT_ENTRIES; pte++) {
896     /* start from 0 to I386_VM_PT_ENTRIES = 1024*/
897 u32_t pte_v; /* page table entry value*/
898         u32_t pfa ; /* frame address value*/
899         /* read the page table entry value*/
900 pte_v = phys_get32((u32_t) (pagetable + pte));
901         if(check_pt_entry(pte_v)!=OK) continue;
902         /* read the frame address value*/
903 pfa = I386_VM_PFA(pte_v);
904         /* if the frame address value is zero
905          * continue Not label that page*/
906
907         /* pmb is the data structure to
908          * describe that page*/
909         /* next_pmb is the data structure to put
910          * that data is the linked list*/
911         struct pram_mem_block *pmb, *next_pmb;
912         u32_t vaddr = v + pte * I386_PAGE_SIZE;
913
914         if(!(pmb = look_up_pte(p, I386_VM_PDE(vaddr),
915           I386_VM_PTE(vaddr) )))
916             pmb = add_pmb(p, pfa, v, pte);
917         assert(pmb);
918
919         if((p->p_hflags & PROC_SHARING_MEM) &&
920                     !(pmb->flags & WS_SHARED) &&
921           ((look_up_page_in_hsr(p, pmb->vaddr))==OK)){
922             pmb->flags |= WS_SHARED;
923
924         }
925
926         /* disable the WRITE bit*/
927         if(h_step == FIRST_RUN){
```

```
928                 if(pmb->us1!=MAP_NONE){
929                  pte_v =
930                    (pte_v & I386_VM_ADDR_MASK_INV)|pmb->us1;
931                  if(!(pmb->flags & IWS_PF_FIRST) &&
932                     !(pmb->flags & IWS_MOD_KERNEL) )
933                      pte_v &= ~I386_VM_WRITE;
934                 }
935                 else
936                    pte_v &= ~I386_VM_WRITE;
937             }
938             else{
939                 if(pmb->us2!=MAP_NONE){
940                   pte_v =
941                    (pte_v & I386_VM_ADDR_MASK_INV)|pmb->us2;
942                    if(!(pmb->flags & IWS_PF_SECOND) &&
943                      !(pmb->flags & IWS_MOD_KERNEL) )
944                      pte_v &= ~I386_VM_WRITE;
945                 }
946                 else
947                    pte_v &= ~I386_VM_WRITE;
948             }
949             if((pmb->us1!=MAP_NONE) &&
950                   (pmb->us2==MAP_NONE))
951                 pte_v =
952                    (pte_v & I386_VM_ADDR_MASK_INV)|pmb->us0;
953             pte_v &= ~I386_VM_DIRTY;
954
955             /* update the page table*/
956             phys_set32((u32_t) (pagetable + pte), &pte_v);
957         }
958         return;
959 }
```

A.3.4 VM Protected memory (VMPRAM)

Listing A.4: VM Protected memory (VMPRAM)

```
1  #include <machine/vm.h>
2
3  #include <minix/type.h>
4  #include <minix/syslib.h>
5  #include <minix/cpufeature.h>
6  #include <string.h>
7  #include <assert.h>
8  #include <signal.h>
9  #include <stdlib.h>
10
11
```

```
12 #include <machine/vm.h>
13
14 #include "htype.h"
15 #include "glo.h"
16 #include "proto.h"
17 #include "util.h"
18 #include "region.h"
19 static void free_pram_mem_block(
20       struct vmproc *current_p, vir_bytes vaddr);
21
22 int tell_kernel_for_us1_us2(struct vmproc *vmp,
23       vir_bytes v, phys_bytes physaddr, size_t bytes )
24 {
25   int pages = bytes / VM_PAGE_SIZE, p;
26   phys_bytes us1, us1_cl, us2, us2_cl;
27   struct pram_mem_block *pmb;
28   for(p = 0; p < pages; p++) {
29     /*already in us1_us2 list?*/
30     if(!(pmb = look_up_pte(vmp,
31        I386_VM_PDE(v),I386_VM_PTE(v) ))){
32       /* no allocate us1 and us2 and add it to us1_us2
33        * list */
34       if((us1_cl = alloc_mem(1, PAF_CLEAR)) == NO_MEM)
35          panic("tell_kernel_for_us1_us2 :"
36            " no mem to allocate for copy-on-write\n");
37       us1 = CLICK2ABS(us1_cl);
38       if((us2_cl = alloc_mem(1, PAF_CLEAR)) == NO_MEM)
39          panic("tell_kernel_for_us1_us2 :"
40              " no mem to allocate for copy-on-write\n");
41       us2 = CLICK2ABS(us2_cl);
42       pmb = add_pmb(vmp,  v, physaddr, us1, us2);
43     }
44     else pmb->us0 = physaddr;/*yes update US0*/
45     if(sys_addregionto_ws(vmp->vm_endpoint, v, 1,
46        pmb->us1, pmb->us2)!=OK)
47        return(EFAULT);
48     v += VM_PAGE_SIZE;
49     physaddr+=VM_PAGE_SIZE;
50   }
51   return(OK);
52 }
53
54 struct pram_mem_block * add_pmb(struct vmproc *vmp,
55    vir_bytes v, phys_bytes pfa,
56    phys_bytes us1, phys_bytes us2){
57   struct pram_mem_block *pmb, *next_pmb;
58   /* ask for a new pmb block in the free list*/
59   pmb = get_pmb();
60   /* be sure that we got a good block*/
```

```
61    assert(pmb);
62    pmb->us0 =  pfa;
63    pmb->vaddr      =  v;
64    pmb->us1 = us1;
65    pmb->us2 = us2;
66    /* Insert the block on the process's linked list*/
67    if(!vmp->vm_lus1_us2_size)
68       vmp->vm_lus1_us2 = pmb;
69    else{
70      next_pmb = vmp->vm_lus1_us2;
71      while(next_pmb->next_pmb)
72        next_pmb = next_pmb->next_pmb;
73      next_pmb->next_pmb = pmb;
74      next_pmb->next_pmb->next_pmb = NULL;
75    }
76    pmb->id = vmp->vm_lus1_us2_size;
77    vmp->vm_lus1_us2_size++;
78    return(pmb);
79  }
80
81  int free_region_pmbs(struct vmproc *vmp,
82        vir_bytes raddr, vir_bytes length){
83     if(vmp->vm_lus1_us2_size <= 0)
84             return(OK);
85     int p;
86     int pde = I386_VM_PDE(raddr);
87     int pte = I386_VM_PTE(raddr);
88     vir_bytes page_base =
89           pde * ARCH_VM_PT_ENTRIES * VM_PAGE_SIZE
90                                + pte * VM_PAGE_SIZE;
91     vir_bytes vaddr;
92     int n_pages_covered = (length +
93           (raddr - page_base) - 1)/(4*1024)  + 1;
94     struct pram_mem_block *pmb ;
95     for(p = 0; p < n_pages_covered; p++){
96          vaddr = page_base + p*4*1024;
97          pde = I386_VM_PDE(vaddr);
98          pte = I386_VM_PTE(vaddr);
99          if(!(pmb =  look_up_pte(vmp, pde, pte)))
100          continue;
101         free_pram_mem_block(vmp, pmb->vaddr);
102    }
103    sys_free_pmbs(vmp->vm_endpoint, raddr, length );
104    return(OK);
105 }
106
107 /*=========================================================*
108  *      free_pram_mem_block                    *
109  *=========================================================*/
```

```
110
111 static void free_pram_mem_block(
112       struct vmproc *current_p, vir_bytes vaddr){
113   /* Delete the block corresponding to vaddr
114    * from the PE woking set in
115    * VM address space. If the block is found it is
116    * remove from the list
117    * and the data structure in the whole available
118    * blocks is made free
119    * If the end of the list is reached the function
120    * return*/
121    assert(current_p->vm_lus1_us2_size > 0);
122    struct pram_mem_block *pmb =
123                        current_p->vm_lus1_us2;
124    struct pram_mem_block *prev_pmb = NULL;
125    struct pram_mem_block *next_pmb = NULL;
126    int __pte, __pde, pte, pde;
127    pte = I386_VM_PTE(vaddr);
128    pde = I386_VM_PDE(vaddr);
129    while(pmb){
130      __pte = I386_VM_PTE(pmb->vaddr);
131      __pde = I386_VM_PDE(pmb->vaddr);
132      if((__pte==pte)&&(__pde==pde)){
133       free_mem(ABS2CLICK(pmb->us1), 1);
134       free_mem(ABS2CLICK(pmb->us2), 1);
135       pmb->flags = PRAM_SLOT_FREE;
136       pmb->vaddr = MAP_NONE;
137       pmb->id    = 0;
138       pmb->us1 = MAP_NONE;
139       pmb->us2 = MAP_NONE;
140       pmb->us0 = MAP_NONE;
141             current_p->vm_lus1_us2_size--;
142             if(prev_pmb)
143                prev_pmb->next_pmb = pmb->next_pmb;
144             else
145               current_p->vm_lus1_us2 = pmb->next_pmb;
146              break;
147   }
148   prev_pmb = pmb;
149   next_pmb = pmb->next_pmb;
150   pmb = pmb->next_pmb;
151       }
152 }
153
154
155 /*=========================================================*
156  *          get_pmb          *
157  *=========================================================*/
158 struct pram_mem_block *get_pmb(void){
```

```
159     /*   Return the first avalaible pmb in lus1_us2
160      *   table.
161      **  If the table is full a panic is triggered
162      **  otherwise the found pmb is returnedtable */
163   int pmb_offset = 0;
164     /* start from the first pmb */
165   struct pram_mem_block *pmb = BEG_PRAM_MEM_BLOCK_ADDR;
166     /**If the first block is free return it**/
167   if(pmb->flags & PRAM_SLOT_FREE) {
168       /**Reset the data in the block
169         * and return the block**/
170       pmb->flags &= ~PRAM_SLOT_FREE;
171       pmb->next_pmb = NULL;
172       pmb->us1 = MAP_NONE;
173       pmb->us2 = MAP_NONE;
174       pmb->us0 = MAP_NONE;
175       pmb->vaddr = MAP_NONE;
176       return pmb;
177     }
178     do{
179      /*** Otherwise go through the lus1_us2
180       * and search the first available
181       *** bloc  ***/
182        pmb_offset++;
183        pmb++;
184     }while(!(pmb->flags & PRAM_SLOT_FREE) &&
185       pmb < END_PRAM_MEM_BLOCK_ADDR);
186     /**The end of the lus1_us2 is reached panic **/
187     if(pmb_offset>= WORKING_SET_SIZE)
188        panic("Block list is full stop %d\n",
189          pmb_offset);
190     /**The bloc is found, reset the content of
191      * the bloc and return it**/
192     pmb->flags &= ~PRAM_SLOT_FREE;
193     pmb->us1 = MAP_NONE;
194     pmb->us2 = MAP_NONE;
195     pmb->us0 = MAP_NONE;
196     pmb->vaddr = MAP_NONE;
197     pmb->next_pmb = NULL;
198     return pmb;
199 }
200
201 /*=============================================*
202  *         look_up_pte *
203  *=============================================*/
204 struct pram_mem_block * look_up_pte(
205      struct vmproc *current_p, int pde, int pte){
206     /** Search in the PE working set list the block
207      *   with the given pde and pte
```

```
208       ** if the working set list is empty a NULL
209       *  pointer on block is return.
210       ** If the end of
211       ** the working set list is reached without
212       *  finding the block a NULL
213       ** pointer is return. If the block is found
214       *  a pointer on the block is
215       ** return**/
216    if(current_p->vm_lus1_us2_size <= 0)
217        return(NULL);
218    struct pram_mem_block *pmb = current_p->vm_lus1_us2;
219    int __pte, __pde;
220    while(pmb){
221        __pte = I386_VM_PTE(pmb->vaddr);
222        __pde = I386_VM_PDE(pmb->vaddr);
223        if((__pte==pte)&&(__pde==pde))
224     return(pmb);
225        pmb = pmb->next_pmb;
226     }
227     return(NULL);
228 }
229
230 /*=============================================================*
231  *          free_pram_mem_blocks   *
232  *=============================================================*/
233 void free_pram_mem_blocks(struct vmproc *current_p){
234       /** Delete of the blokcs in the PE working set list.
235        *  If the working
236        ** set list is empty the function return. Otherwise
237        ** each blocks is delete
238        ** A the end of the function the working set of the
239        ** PE should be empty ***/
240       if(current_p->vm_lus1_us2_size <= 0)
241            return;
242       struct pram_mem_block *pmb = current_p->vm_lus1_us2;
243       while(pmb){
244          free_pram_mem_block(current_p, pmb->vaddr);
245    pmb = pmb->next_pmb;
246       }
247       assert(current_p->vm_lus1_us2_size == 0);
248       assert(current_p->vm_lus1_us2 == NULL);
249 }
```

A.3.5 VM Copy-on-write (VMCOW)

Listing A.5: VM Copy-on-write (VMCOW)

```
1 #include <machine/vm.h>
```

```
2
3  #include <minix/type.h>
4  #include <minix/syslib.h>
5  #include <minix/cpufeature.h>
6  #include <string.h>
7  #include <assert.h>
8  #include <signal.h>
9  #include <stdlib.h>
10
11
12 #include <machine/vm.h>
13
14 #include "htype.h"
15 #include "glo.h"
16 #include "proto.h"
17 #include "util.h"
18 #include "region.h"
19 static int hmap_pf(struct vmproc *vmp,
20            struct vir_region *region,
21            vir_bytes offset, int write,
22            u32_t addr, int rw, int what);
23 /*===========================================================*
24  *     allocate_mem_4_hardening                              *
25  *===========================================================*/
26 void allocate_mem_4_hardening (struct vmproc *vmp,
27     struct vir_region *region,
28     struct phys_region *ph, int what ){
29     /** When called allocate_mem_4_hardening
30      ** allocate a frame for the hardening
31      ** If it is called during the first run,
32      ** the frame is allocated for US1
33      ** If it is called during the second run,
34      **  the frame is allocated for US2
35      ** The data in the corresponding frame in
36      ** US0 is copied in the new frame
37      ** The kernel is informed to update the bloc
38      ** data structure within its own
39      ** address space. **/
40     struct pram_mem_block *pmb, *next_pmb;
41     phys_bytes new_page_1, new_page_cl_1, phys_back;
42     vir_bytes vaddr;
43     u32_t allocflags;
44     vaddr = region->vaddr+ph->offset;
45     allocflags = vrallocflags(region->flags);
46
47     vmp->vm_hflags |= VM_PROC_TO_HARD;
48     if(!(pmb = look_up_pte(vmp,  I386_VM_PDE(vaddr),
49       I386_VM_PTE(vaddr) ))){
50     /* rememeber that we are working with page
```

```
51          * already in the working set*/
52             pmb = get_pmb();
53             assert(pmb);
54 #if CHECK_DEBUG
55             assert(pmb->us0 == MAP_NONE);
56             assert(pmb->us1 == MAP_NONE);
57             assert(pmb->us2 == MAP_NONE);
58             assert(pmb->vaddr == MAP_NONE);
59 #endif
60
61              /** Here on frame is allocated ***/
62             if((new_page_cl_1 = alloc_mem(1,
63                 allocflags|PAF_CLEAR)) == NO_MEM)
64                 panic("allocate_mem_4_hardening :"
65                   "no mem to allocate for copy-on-write\n");
66             new_page_1 = CLICK2ABS(new_page_cl_1);
67
68             /* copy the content of the current
69              * frame ram_phys in the new frame
70              * allocated */
71             if(sys_abscopy(ph->ph->phys, new_page_1,
72                VM_PAGE_SIZE) != OK)
73                 panic("VM: abscopy failed: when "
74                       "copying data during copy"
75                       "on write page fault handling\n");
76             /* update the VM data structure *
77              * linked the frames to us0, us1,
78              * us2 in the VM */
79             pmb->us0 =  ph->ph->phys;
80             if(what == VM_FIRST_RUN)
81                  pmb->us1 = new_page_1;
82             if(what == VM_SECOND_RUN)
83                  pmb->us2 = new_page_1;
84
85             pmb->vaddr      =    vaddr;
86
87             /**Insert the new pmb bloc into the
88              * process's VM working set list**/
89             if(!vmp->vm_lus1_us2_size)
90                  vmp->vm_lus1_us2 = pmb;
91             else{
92                   next_pmb = vmp->vm_lus1_us2;
93                   while(next_pmb->next_pmb)
94                     next_pmb = next_pmb->next_pmb;
95                   next_pmb->next_pmb = pmb;
96                   next_pmb->next_pmb->next_pmb = NULL;
97             }
98             pmb->id = vmp->vm_lus1_us2_size;
99             vmp->vm_lus1_us2_size++;
```

```
100
101         /*** Now inform the Kernel so the VM
102          *    will update it part of
103          *** pmb attributes : us0, us1, us2
104          *** after sys_hmem_map the micro-kernel
105          **  and the VM share the same
106          *** information on the pmb attributes.***/
107         if(what == VM_FIRST_RUN){
108             if(sys_hmem_map(vmp->vm_endpoint,
109                new_page_1, -2L, -2L, 0)!=OK)
110                panic("VM: sys_hmem_map failed: when "
111                      "informing the kernel during"
112                 "copy on write page fault handling\n");
113         }
114         if(what == VM_SECOND_RUN){
115             if(sys_hmem_map(vmp->vm_endpoint, -2L,
116                           new_page_1, -2L, 0)!=OK)
117             panic("VM: sys_hmem_map failed: when "
118                    "informing the kernel during"
119                "copy on write page fault handling\n");
120         }
121
122     }
123     else{
124         // TO COMPLETE
125          if((pmb->us1 != MAP_NONE) &&
126                 (pmb->us2!=MAP_NONE)){
127              pmb->us0 = ph->ph->phys;
128
129         }
130
131         if((pmb->us1 == MAP_NONE) ||
132                 (pmb->us2==MAP_NONE)){
133             /** Here on frame is allocated ***/
134          if((new_page_cl_1 = alloc_mem(1,
135            allocflags|PAF_CLEAR)) == NO_MEM)
136              panic("allocate_mem_4_hardening :"
137               "no mem to allocate for copy-on-write\n");
138           new_page_1 = CLICK2ABS(new_page_cl_1);
139         }
140         switch (what){
141            case VM_FIRST_RUN: // TO COMPLETE
142                 if(pmb->us1==MAP_NONE)
143                     pmb->us1 = new_page_1;
144               /* copy the content of the current frame
145                    * ram_phys in the new frame
146                    * allocated */
147                 if(sys_abscopy(ph->ph->phys, pmb->us1,
148                     VM_PAGE_SIZE) != OK)
```

```
149                         panic("VM: abscopy failed: when"
150                           "copying data during copy"
151                           "on write page fault handling\n");
152                     if(sys_hmem_map(vmp->vm_endpoint,
153                         pmb->us1,-2L, -2L, 0)!=OK)
154                       panic("VM: sys_hmem_map failed: "
155                         "when informing the kernel during"
156                         "copy on write page fault "
157                         "handling\n");
158                     break;
159                case VM_SECOND_RUN: // TO COMPLETE
160                     if(pmb->us2==MAP_NONE)
161                         pmb->us2 = new_page_1;
162                     if(sys_abscopy(ph->ph->phys, pmb->us2,
163                        VM_PAGE_SIZE) != OK)
164                         panic("VM: abscopy failed: when"
165                           "copying data during copy"
166                           "on write page fault handling\n");
167                       if(sys_hmem_map(vmp->vm_endpoint, -2L,
168                             pmb->us2, -2L, 0)!=OK)
169                       panic("VM: sys_hmem_map failed: "
170                         "when informing the kernel during"
171                           " copy on write page fault "
172                           "handling\n");
173
174                       break;
175                default:
176                     panic("Unkown value for what in"
177                           " hmap_pf\n");
178
179             }
180         }
181         return;
182 }
183
184 /*===========================================================*
185  *                      do_hpagefaults                       *
186  *===========================================================*/
187 void do_hpagefaults(message *m)
188 {
189   endpoint_t ep = m->m_source;
190   u32_t addr = m->VPF_ADDR;
191   u32_t err = m->VPF_FLAGS;
192   struct vmproc *vmp;
193   int s, mem_flags = 0;
194
195   struct vir_region *region;
196   vir_bytes offset;
197   int p, wr = PFERR_WRITE(err), rw = 0, what = 0;
```

```
198
199   if(vm_isokendpt(ep, &p) != OK)
200       panic("do_pagefaults: endpoint"
201                " wrong: %d", ep);
202
203   vmp = &vmproc[p];
204   assert((vmp->vm_flags & VMF_INUSE));
205
206           /* 1st execution*/
207           if(m->m_type == VM_HR1PAGEFAULT )
208                 what = VM_FIRST_RUN;
209
210           /* 2nd execution */
211           if(m->m_type == VM_HR2PAGEFAULT)
212                 what = VM_SECOND_RUN;
213
214           /* The virtual address is not
215            * belong a valid memory space */
216   if(!(region = map_lookup(vmp, addr, NULL))) {
217                   panic("region null 0x%x\n", addr);
218   }
219   assert(region);
220   assert(addr >= region->vaddr);
221   offset = addr - region->vaddr;
222
223      /* Access is allowed; handle it.
224              * allocate the frame */
225           if((hmap_pf(vmp, region, offset, wr,addr,
226                rw, what)) != OK) {
227     panic("VM: pagefault: SIGSEGV %d "
228                  "pagefault not handled\n", ep);
229     if((s=sys_kill(vmp->vm_endpoint,
230                       SIGSEGV)) != OK)
231           panic("sys_kill failed: %d", s);
232     if((s=sys_vmctl(ep,
233                    VMCTL_CLEAR_PAGEFAULT,
234                    0 /*unused*/)) != OK)
235        panic("do_pagefaults: sys_vmctl"
236                      "failed: %d", ep);
237     return;
238   }
239
240   /* Pagefault is handled, so now
241          * reactivate the process. */
242   if((s=sys_vmctl(ep, VMCTL_CLEAR_PAGEFAULT,
243                0 /*unused*/)) != OK)
244     panic("do_pagefaults: sys_vmctl"
245                  " failed: %d", ep);
246
```

```
247 }
248
249 /*===========================================================*
250  *                        hmap_pf                            *
251  *===========================================================*/
252 static int hmap_pf(vmp, region, offset,
253                    write, addr, rw, what)
254 struct vmproc *vmp;
255 struct vir_region *region;
256 vir_bytes offset;
257 int write;
258 u32_t addr;
259 int rw;
260 int what;
261 {
262   struct phys_region *ph;
263   int r = OK;
264   struct phys_block *pb;
265
266   offset -= offset % VM_PAGE_SIZE;
267
268   assert(offset >= 0);
269   assert(offset < region->length);
270
271   assert(!(region->vaddr % VM_PAGE_SIZE));
272   assert(!(write &&
273            !(region->flags & VR_WRITABLE)));
274
275
276         /* get the phys block in the VM.
277            Phys block is the data structure to
278            handle the page in the VM*/
279
280         ph = physblock_get(region, offset);
281
282         assert(ph);
283
284
285         /*  Do the allocation of US1 or
286          *  US2 frame**/
287         allocate_mem_4_hardening(vmp,
288           region, ph, what);
289         return(OK);
290 }
291
292 int do_memconf(message *m){
293
294         endpoint_t ep = m->m_source;
295   struct vmproc *vmp;
```

```
296     int p;
297
298     if(vm_isokendpt(ep, &p) != OK)
299         panic("do_pagefaults: endpoint
300                 wrong: %d", ep);
301
302     vmp = &vmproc[p];
303             free_pram_mem_blocks(vmp);
304             return(OK);
305
306 }
```

A.3.6 US0 change handler (US0h)

Listing A.6: US0 change handler (US0h)

```
1  #include "kernel/kernel.h"
2  #include "kernel/vm.h"
3
4  #include <machine/vm.h>
5
6  #include <minix/type.h>
7  #include <minix/syslib.h>
8  #include <minix/cpufeature.h>
9  #include <string.h>
10 #include <assert.h>
11 #include <signal.h>
12 #include <stdlib.h>
13
14
15 #include <machine/vm.h>
16
17
18 #include "arch_proto.h"
19
20 #include "htype.h"
21 #include "hproto.h"
22 #include "rcounter.h"
23 #include "mca.h"
24 #include "../../clock.h"
25
26 #ifdef USE_APIC
27 #include "apic.h"
28 #ifdef USE_WATCHDOG
29 #include "kernel/watchdog.h"
30 #endif
31 #endif
32 static int update_ws_us1_us2_data_pmb(struct proc *rp,
```

```
33      struct pram_mem_block *pmb);
34  int update_ws_us1_us2_data_vaddr(struct proc *rp,
35       vir_bytes vaddr){
36      int pde, pte;
37      pde = I386_VM_PDE(vaddr); pte = I386_VM_PTE(vaddr);
38      int r1, r2;
39      struct pram_mem_block *pmb =
40           look_up_pte(rp, pde, pte);
41      if(pmb && (pmb->us0!=MAP_NONE) &&
42          (pmb->us1!=MAP_NONE) &&
43           (pmb->us2!=MAP_NONE)){
44           if(cpy_frames(pmb->us0, pmb->us1)!=OK){
45              panic("update_ws_us1_us2_data:Copy "
46               "first_phys to pram failed\n");
47              return(EFAULT);
48           }
49           if(cpy_frames(pmb->us0, pmb->us2)!=OK){
50              panic("update_ws_us1_us2_data:Copy "
51                "second_phys to pram failed\n");
52              return(EFAULT);
53           }
54           pmb->flags   |= IWS_MOD_KERNEL;
55      }
56      return(OK);
57  }
58
59  static int update_ws_us1_us2_data_pmb(struct proc *rp,
60              struct pram_mem_block *pmb){
61      if(!pmb) return(OK);
62      int r1, r2;
63      if((pmb->us0!=MAP_NONE) && (pmb->us1!=MAP_NONE) &&
64          (pmb->us2!=MAP_NONE)){
65           if(cpy_frames(pmb->us0, pmb->us1)!=OK){
66              panic("update_ws_us1_us2_data_pmb:Copy "
67              "first_phys to pram failed\n");
68              return(EFAULT);
69           }
70           if(cpy_frames(pmb->us0, pmb->us2)!=OK){
71              panic("update_ws_us1_us2_data_pmb:Copy "
72              "second_phys to pram failed\n");
73              return(EFAULT);
74           }
75     }
76      return(OK);
77  }
78
79  int update_all_ws_us1_us2_data(struct proc *rp){
80    if(rp->p_lus1_us2_size <= 0)
81        return(OK);
```

```
82    int r;
83    struct pram_mem_block *pmb = rp->p_lus1_us2;
84    while(pmb){
85      if((r=update_ws_us1_us2_data_pmb(rp, pmb))!=OK)
86          return(r);
87      pmb = pmb->next_pmb;
88    }
89    return(OK);
90  }
91
92  int update_range_ws_us1_us2_data(struct proc *rp,
93                vir_bytes offset,
94                int n_pages_covered){
95      int pde = I386_VM_PDE(offset);
96      int pte = I386_VM_PTE(offset);
97      vir_bytes page_base =
98            pde * I386_VM_PT_ENTRIES * I386_PAGE_SIZE
99                                + pte * I386_PAGE_SIZE;
100     int i = 0;
101     for(i = 0; i < n_pages_covered; i++ ){
102        if(update_ws_us1_us2_data_vaddr(rp,
103            page_base + i*4*1024)!=OK)
104            return(EFAULT);
105     }
106     return(OK);
107 }
```

A.3.7 System call handler (SCH)

Listing A.7: System call handler (SCH)

```
1  #include "kernel/kernel.h"
2  #include "kernel/vm.h"
3
4  #include <machine/vm.h>
5
6  #include <minix/type.h>
7  #include <minix/syslib.h>
8  #include <minix/cpufeature.h>
9  #include <string.h>
10 #include <assert.h>
11 #include <signal.h>
12 #include <stdlib.h>
13
14
15 #include <machine/vm.h>
16
17
```

```
18 #include "arch_proto.h"
19
20 #include "htype.h"
21 #include "hproto.h"
22 #include "rcounter.h"
23 #include "mca.h"
24 #include "../../clock.h"
25
26 #ifdef USE_APIC
27 #include "apic.h"
28 #ifdef USE_WATCHDOG
29 #include "kernel/watchdog.h"
30 #endif
31 #endif
32 static struct hardening_mem_event *get_hme(void);
33 static int free_hardening_mem_event
34                 (struct proc *rp, int id);
35 /*=========================================================*
36  *         get_hme            *
37  *=========================================================*/
38 /*    Return the first avalaible hme in
39  *    hardening_mem_events.
40  **   If the table is full a panic is triggered
41  **   otherwise the found hme is returnedtable */
42 static struct hardening_mem_event *get_hme(void){
43     int hme_offset = 0;
44     /* start from the first hme */
45     struct hardening_mem_event *hme =
46                 BEG_HARDENING_MEM_EVENTS_ADDR;
47     /**If the first block is free return it**/
48   if(hme->flags & HME_SLOT_FREE) {
49               /**Reset the data in the block
50                * and return the block**/
51       hme->flags &= ~HME_SLOT_FREE;
52               hme->next_hme = NULL;
53               hme->addr_base = MAP_NONE;
54               hme->nbytes = 0;
55       return hme;
56   }
57   do{
58      /*** Otherwise go through the lus1_us2 and
59       *  search the first available
60       *** bloc  ***/
61     hme_offset++;
62     hme++;
63   }while(!(hme->flags & HME_SLOT_FREE) &&
64                   hme < END_HARDENING_MEM_EVENTS_ADDR);
65
66     /**The end of the lus1_us2 is reached panic **/
```

```
67          if(hme_offset>= HARDENING_MEM_EVENTS)
68        panic("ALERT BLOCK LIST EVENTS "
69                "IS FULL STOP STOP %d\n", hme_offset);
70
71          /** The bloc is found, reset the content
72           *  of the bloc and return it**/
73    hme->flags &=~HME_SLOT_FREE;
74          hme->next_hme = NULL;
75          hme->addr_base = MAP_NONE;
76          hme->nbytes = 0;
77    return hme;
78  }
79
80  int add_hme_event(struct proc *rp,
81                    vir_bytes offset, vir_bytes size){
82
83     int pde = I386_VM_PDE(offset);
84     int pte = I386_VM_PTE(offset);
85     vir_bytes page_base =
86           pde * I386_VM_PT_ENTRIES * I386_PAGE_SIZE
87                                + pte * I386_PAGE_SIZE;
88     int n_pages_covered =
89      (size + (offset - page_base) - 1)/(4*1024)  + 1;
90     struct hardening_mem_event * hme;
91     if(!(hme = look_up_hme(rp, offset))){
92        /* ask for a new pmb block in the free list*/
93        hme = get_hme();
94        struct hardening_mem_event *next_hme;
95        /* be sure that we got a good block*/
96        assert(hme);
97        hme->addr_base =  offset;
98        hme->nbytes    = size;
99        /* Insert the block on the process's linked list*/
100       hme->npages = n_pages_covered;
101       if(!rp->p_nb_hardening_mem_events)
102          rp->p_hardening_mem_events = hme;
103       else{
104          next_hme = rp->p_hardening_mem_events;
105          while(next_hme->next_hme)
106             next_hme = next_hme->next_hme;
107          next_hme->next_hme = hme;
108          next_hme->next_hme->next_hme = NULL;
109       }
110       hme->id = rp->p_nb_hardening_mem_events;
111       return(++rp->p_nb_hardening_mem_events);
112    }
113    if(hme->npages < n_pages_covered){
114       hme->addr_base =  offset;
115       hme->nbytes    = size;
```

```
116         hme->npages = n_pages_covered;
117       }
118       return(rp->p_nb_hardening_mem_events);
119   }
120
121
122   int   handle_hme_events(struct proc *rp){
123       if(rp->p_nb_hardening_mem_events <= 0)
124           return(OK);
125       struct hardening_mem_event *hme =
126            rp->p_hardening_mem_events;
127       while(hme){
128         if(update_range_ws_us1_us2_data(rp,
129               hme->addr_base, hme->npages)!=OK)
130             return(EFAULT);
131         hme = hme->next_hme;
132     }
133     return(OK);
134   }
135
136   /*=============================================================*
137    *    free_hardening_mem_events                *
138    *============================================================*/
139   void free_hardening_mem_events(struct proc *rp){
140       if(rp->p_nb_hardening_mem_events <= 0)
141           return;
142       struct hardening_mem_event *hme =
143                rp->p_hardening_mem_events;
144       int lsize = rp->p_nb_hardening_mem_events;
145       while(hme){
146             if(free_hardening_mem_event(rp,
147                  hme->id)!=--lsize)
148                  panic("Freeing hme with "
149                        "error %d\n", lsize);
150     hme = hme->next_hme;
151         }
152       assert(rp->p_nb_hardening_mem_events == 0);
153       assert(rp->p_hardening_mem_events == NULL);
154   }
155
156
157   /*=============================================================*
158    *        free_hardening_mem_event          *
159    *============================================================*/
160   static int free_hardening_mem_event(struct proc *rp,
161       int id){
162       if(rp->p_nb_hardening_mem_events <= 0)
163           return(0);
164       struct hardening_mem_event *hme =
```

```
165                     rp->p_hardening_mem_events;
166     struct hardening_mem_event *prev_hme = NULL;
167     struct hardening_mem_event *next_hme = NULL;
168     while(hme){
169       if(hme->id == id){
170         hme->flags = HME_SLOT_FREE;
171         hme->id     = 0;
172         hme->addr_base =  0;
173         hme->nbytes     = 0;
174         hme->npages = 0;
175         rp->p_nb_hardening_mem_events--;
176         if(prev_hme)
177           prev_hme->next_hme = hme->next_hme;
178         else
179           rp->p_hardening_mem_events =
180                 hme->next_hme;
181         break;
182       }
183       prev_hme = hme;
184       next_hme = hme->next_hme;
185       hme = hme->next_hme;
186   }
187   return(rp->p_nb_hardening_mem_events);
188 }
189
190
191 /*=============================================*
192  *     look_up_hme                             *
193  *=============================================*/
194 struct hardening_mem_event *
195     look_up_hme(struct proc *rp, vir_bytes offset){
196     if(rp->p_nb_hardening_mem_events <= 0)
197       return(NULL);
198     int pte = I386_VM_PTE(offset);
199     int pde = I386_VM_PDE(offset);
200     struct hardening_mem_event *hme =
201           rp->p_hardening_mem_events;
202     int __pte, __pde;
203     while(hme){
204         __pte = I386_VM_PTE(hme->addr_base);
205         __pde = I386_VM_PDE(hme->addr_base);
206         if((__pte==pte)&&(__pde==pde))
207      return(hme);
208         hme = hme->next_hme;
209     }
210     return(NULL);
211 }
```

A.3.8 US0 change handler (US0h)

Listing A.8: Shared Memory Handler (SMH)

```
#include "kernel/kernel.h"
#include "kernel/vm.h"

#include <machine/vm.h>

#include <minix/type.h>
#include <minix/syslib.h>
#include <minix/cpufeature.h>
#include <string.h>
#include <assert.h>
#include <signal.h>
#include <stdlib.h>


#include <machine/vm.h>


#include "arch_proto.h"

#include "htype.h"
#include "hproto.h"
#include "rcounter.h"
#include "mca.h"
#include "../../clock.h"

#ifdef USE_APIC
#include "apic.h"
#ifdef USE_WATCHDOG
#include "kernel/watchdog.h"
#endif
#endif
static int free_hsr(struct proc *rp, int id);
static struct hardening_shared_proc *get_hsp(void);
static struct hardening_shared_region *get_hsr(void);
static struct hardening_shared_region * look_up_hsr(
            struct proc *rp, vir_bytes offset,
            vir_bytes length, int r_id);
static struct hardening_shared_proc * look_up_hsp(
                  struct proc *rp,
                  struct hardening_shared_region *hsr);
static struct hardening_shared_proc
                  *look_up_unique_hsp(struct proc *rp);
static struct hardening_shared_region
                  * look_up_unique_hsr(int r_id);
static int add_all_hsr_s(
```

```
46                struct hardening_shared_region * hsr);
47 static int add_all_hsp_s(
48                struct hardening_shared_proc * hsp);
49 static int free_hsp_from_hsr(
50                struct proc *rp,
51                struct hardening_shared_region*hsr);
52 static int free_hsr_from_all_hsr_s(
53                struct hardening_shared_region *phsr);
54 static int free_hsp_from_all_hsp_s(
55               struct proc *rp);
56 static struct hardening_shared_region
57             * look_up_hsr_vaddr(struct proc *rp,
58           vir_bytes vaddr, vir_bytes size);
59
60 void enable_hme_event_in_procs(struct proc *rp,
61             vir_bytes vaddr, vir_bytes size ){
62    if(rp->p_nb_hardening_shared_regions <= 0)
63       return;
64    struct hardening_shared_region *hsr =
65             look_up_hsr_vaddr(rp, vaddr,size);
66    struct hardening_shared_proc *hsp = hsr->r_hsp;
67    while(hsp){
68          add_hme_event(rp, vaddr, size );
69          hsp = hsp->next_hsp;
70    }
71 }
72
73 /*=========================================================*
74  *         get_hsr        *
75  *=========================================================*/
76 static struct hardening_shared_region *get_hsr(void){
77     int hsr_offset = 0;
78     /* start from the first hsr */
79     struct hardening_shared_region *hsr =
80               BEG_HARDENING_SHARED_REGIONS_ADDR;
81     /**If the first block is free return it**/
82   if(hsr->flags & HSR_SLOT_FREE) {
83             /**Reset the data in the block
84              * and return the block**/
85       hsr->flags &=~HSR_SLOT_FREE;
86             hsr->next_hsr = NULL;
87             hsr->vaddr = 0;
88             hsr->length = 0;
89             hsr->r_hsp  = NULL;
90       return hsr;
91   }
92   do{
93      /*** Otherwise go through the lus1_us2
94       *   and search the first available
```

```
95          *** bloc  ***/
96        hsr_offset++;
97        hsr++;
98      }while(!(hsr->flags & HSR_SLOT_FREE) &&
99                hsr < END_HARDENING_SHARED_REGIONS_ADDR);
100
101       /**The end of the lus1_us2 is reached panic **/
102           if(hsr_offset>= HARDENING_SHARED_REGIONS)
103        panic("ALERT HARDENING_SHARED_REGIONS IS"
104                 " FULL STOP STOP %d\n", hsr_offset);
105
106           /**The bloc is found, reset the content of
107            * the bloc and return it**/
108     hsr->flags &=~HSR_SLOT_FREE;
109           hsr->next_hsr = NULL;
110           hsr->vaddr = 0;
111           hsr->length = 0;
112           hsr->r_hsp  = NULL;
113     return hsr;
114   }
115
116
117   /*=============================================*
118    *         get_hsp          *
119    *=============================================*/
120   static struct hardening_shared_proc *get_hsp(void){
121       int hsp_offset = 0;
122       /* start from the first hsp */
123       struct hardening_shared_proc *hsp =
124                   BEG_HARDENING_SHARED_PROCS_ADDR;
125       /**If the first block is free return it**/
126     if(hsp->flags & HSP_SLOT_FREE) {
127                /**Reset the data in the block and
128                 * return the block**/
129         hsp->flags &=~HSP_SLOT_FREE;
130               hsp->next_hsp = NULL;
131               hsp->hsp_endpoint = 0;
132         return hsp;
133     }
134     do{
135        /*** Otherwise go through the lus1_us2
136         *   and search the first available
137         *** bloc  ***/
138       hsp_offset++;
139       hsp++;
140     }while(!(hsp->flags & HSP_SLOT_FREE) &&
141                hsp < END_HARDENING_SHARED_PROCS_ADDR);
142
143       /**The end of the lus1_us2 is reached panic **/
```

```
144         if(hsp_offset>= HARDENING_SHARED_PROCS)
145       panic("ALERT HARDENING_SHARED_REGIONS "
146               "IS FULL STOP STOP %d\n", hsp_offset);
147
148         /**The bloc is found, reset the content
149          of the bloc and return it**/
150   hsp->flags &=~HSP_SLOT_FREE;
151         hsp->next_hsp = NULL;
152         hsp->hsp_endpoint = 0;
153   return hsp;
154 }
155
156 /*=========================================================*
157  *         look_up_hsr *
158  *=========================================================*/
159 static struct hardening_shared_region * look_up_hsr(
160         struct proc *rp, vir_bytes offset,
161               vir_bytes length, int r_id){
162    if(rp->p_nb_hardening_shared_regions <= 0)
163      return(NULL);
164    struct hardening_shared_region *hsr =
165         rp->p_hardening_shared_regions;
166    while(hsr){
167       if((hsr->vaddr == offset)&&
168         (length == hsr->length) && (hsr->r_id == r_id))
169     return(hsr);
170       hsr = hsr->next_hsr;
171    }
172    return(NULL);
173 }
174
175 /*=========================================================*
176  *         look_up_hsp *
177  *=========================================================*/
178 static struct hardening_shared_proc * look_up_hsp(
179         struct proc *rp,
180                  struct hardening_shared_region *hsr){
181    if(hsr->n_hsp <= 0)
182      return(NULL);
183    struct hardening_shared_proc  *hsp = hsr->r_hsp;
184    while(hsp){
185       if(hsp->hsp_endpoint == rp->p_endpoint)
186     return(hsp);
187       hsp = hsp->next_hsp;
188    }
189    return(NULL);
190 }
191
192 /*=========================================================*
```

```
193  *          look_up_unique_hsp    *
194  *================================================================*/
195 static struct hardening_shared_proc * look_up_unique_hsp(
196         struct proc *rp){
197     if(n_hsps <= 0)
198       return(NULL);
199     struct hardening_shared_proc  *hsp = all_hsp_s;
200     while(hsp){
201        if(hsp->hsp_endpoint == rp->p_endpoint)
202      return(hsp);
203        hsp = hsp->next_hsp;
204     }
205     return(NULL);
206 }
207
208
209 /*================================================================*
210  *          look_up_unique_hsr  *
211  *================================================================*/
212 static struct hardening_shared_region *
213       look_up_unique_hsr(int r_id){
214     if(n_hsrs <= 0)
215       return(NULL);
216     struct hardening_shared_region  *hsr = all_hsr_s;
217     while(hsr){
218        if(hsr->r_id == r_id)
219      return(hsr);
220        hsr = hsr->next_hsr;
221     }
222     return(NULL);
223 }
224
225 int add_hsr(struct proc *rp, vir_bytes offset,
226      vir_bytes size, int r_id){
227     struct hardening_shared_region * hsr;
228     if(!(hsr = look_up_hsr(rp, offset,size, r_id))){
229        if(!(hsr = look_up_unique_hsr(r_id))){
230 #if H_DEBUG
231           printf("not found\n");
232 #endif
233           hsr = get_hsr();
234           hsr->vaddr =  offset;
235           hsr->length    = size;
236           hsr->r_id    = r_id;
237           hsr->id = rp->p_nb_hardening_shared_regions;
238           add_all_hsr_s(hsr);
239        }
240        struct hardening_shared_region *next_hsr;
241        assert(hsr); /* be sure that we got a good block*/
```

```
242         if(!rp->p_nb_hardening_shared_regions)
243             rp->p_hardening_shared_regions = hsr;
244         else{
245             next_hsr = rp->p_hardening_shared_regions;
246             while(next_hsr->next_hsr)
247                 next_hsr = next_hsr->next_hsr;
248             next_hsr->next_hsr = hsr;
249             next_hsr->next_hsr->next_hsr = NULL;
250         }
251         add_hsp(rp,hsr);
252         return(++rp->p_nb_hardening_shared_regions);
253     }
254     return(rp->p_nb_hardening_shared_regions);
255 }
256
257
258 int add_hsp(struct proc *rp,
259      struct hardening_shared_region *hsr){
260
261     struct hardening_shared_proc * hsp;
262     if(!(hsp = look_up_hsp(rp, hsr))){
263         if(!(hsp=look_up_unique_hsp(rp))){
264             hsp = get_hsp();
265             hsp->hsp_endpoint =  rp->p_endpoint;
266             hsp->id = hsr->n_hsp;
267             add_all_hsp_s(hsp);
268         }
269         struct hardening_shared_proc *next_hsp;
270         /* be sure that we got a good block*/
271         assert(hsp);
272         if(!hsr->n_hsp) hsr->r_hsp = hsp;
273         else{
274             next_hsp = hsr->r_hsp;
275             while(next_hsp->next_hsp)
276               next_hsp = next_hsp->next_hsp;
277             next_hsp->next_hsp = hsp;
278             next_hsp->next_hsp->next_hsp = NULL;
279         }
280         return(++hsr->n_hsp);
281     }
282     return(hsr->n_hsp);
283 }
284
285 static int add_all_hsr_s(
286     struct hardening_shared_region * hsr){
287     if(all_hsr_s==NULL){
288         all_hsr_s = hsr;
289         return(n_hsrs++);
290     }
```

```
291     struct hardening_shared_region * next_hsr;
292     next_hsr = all_hsr_s;
293     while(next_hsr->next_hsr)
294        next_hsr = next_hsr->next_hsr;
295     next_hsr->next_hsr = hsr;
296     next_hsr->next_hsr->next_hsr = NULL;
297     return(n_hsrs++);
298 }
299 
300 static int add_all_hsp_s(
301     struct hardening_shared_proc * hsp){
302     if(all_hsp_s==NULL){
303        all_hsp_s = hsp;
304        return(n_hsps++);
305     }
306     struct hardening_shared_proc * next_hsp;
307     next_hsp = all_hsp_s;
308     while(next_hsp->next_hsp)
309       next_hsp = next_hsp->next_hsp;
310     next_hsp->next_hsp = hsp;
311     next_hsp->next_hsp->next_hsp = NULL;
312     return(n_hsps++);
313 }
314 
315 void display_all_hsp_s(void){
316   struct hardening_shared_proc * hsp = all_hsp_s;
317   while(hsp){
318     printf("#### Process in all_hsp_s #### %d %d %d\n",
319               hsp->id, hsp->flags, hsp->hsp_endpoint);
320           hsp = hsp->next_hsp;
321   }
322 }
323 
324 void display_all_hsr_s(void){
325   struct hardening_shared_region * hsr = all_hsr_s;
326   while(hsr){
327     printf("#### HARDENING SHARED REGIONS"
328           " #### %d 0x%lx 0x%lx\n",
329              hsr->id, hsr->vaddr, hsr->length);
330           hsr = hsr->next_hsr;
331   }
332 }
333 /*=============================================================*
334  *         free_hsr          *
335  *=============================================================*/
336 static int free_hsr(struct proc *rp, int id){
337     struct hardening_shared_region *hsr =
338         rp->p_hardening_shared_regions;
339     struct hardening_shared_region *prev_hsr = NULL,
```

```
340             *next_hsr = NULL;
341     while(hsr){
342       if(hsr->id == id){
343         free_hsp_from_hsr(rp, hsr);
344         if(hsr->r_hsp == NULL){
345           hsr->flags = HSR_SLOT_FREE;
346           hsr->id     = 0;
347           hsr->vaddr =  0;
348           hsr->length     = 0;
349           free_hsr_from_all_hsr_s(hsr);
350         }
351         rp->p_nb_hardening_shared_regions--;
352         if(prev_hsr)
353           prev_hsr->next_hsr = hsr->next_hsr;
354         else
355           rp->p_hardening_shared_regions =
356                                 hsr->next_hsr;
357         break;
358       }
359       prev_hsr = hsr;
360       next_hsr = hsr->next_hsr;
361       hsr = hsr->next_hsr;
362   }
363   return(rp->p_nb_hardening_shared_regions);
364 }
365
366 static int free_hsp_from_hsr(struct proc *rp,
367                     struct hardening_shared_region*hsr){
368   struct hardening_shared_proc *hsp = hsr->r_hsp;
369   struct hardening_shared_proc *prev_hsp = NULL,
370          *next_hsp = NULL;
371   while(hsp){
372       if(hsp->hsp_endpoint == rp->p_endpoint){
373         hsr->n_hsp--;
374         if(prev_hsp)
375           prev_hsp->next_hsp = hsp->next_hsp;
376         else
377           hsr->r_hsp = hsp->next_hsp;
378         break;
379       }
380       prev_hsp = hsp;
381       next_hsp = hsp->next_hsp;
382       hsp = hsp->next_hsp;
383   }
384   return(hsr->n_hsp);
385 }
386
387 static int free_hsr_from_all_hsr_s(
388    struct hardening_shared_region *phsr){
```

```
389   struct hardening_shared_region *hsr = all_hsr_s;
390   struct hardening_shared_region *prev_hsr = NULL,
391          *next_hsr = NULL;
392   while(hsr){
393      if(hsr->r_id == phsr->r_id){
394        n_hsrs--;
395        if(prev_hsr)
396           prev_hsr->next_hsr = hsr->next_hsr;
397        else
398           all_hsr_s = hsr->next_hsr;
399        break;
400      }
401      prev_hsr = hsr;
402      next_hsr = hsr->next_hsr;
403      hsr = hsr->next_hsr;
404   }
405   return(n_hsrs);
406 }
407
408 static int free_hsp_from_all_hsp_s(struct proc *rp){
409   struct hardening_shared_proc *hsp = all_hsp_s;
410   struct hardening_shared_proc *prev_hsp = NULL,
411   *next_hsp = NULL;
412   while(hsp){
413      if(hsp->hsp_endpoint == rp->p_endpoint){
414        n_hsps--;
415        hsp->hsp_endpoint = 0;
416        hsp->flags = HSP_SLOT_FREE;
417        hsp->id = 0;
418        if(prev_hsp)
419           prev_hsp->next_hsp = hsp->next_hsp;
420        else
421           all_hsp_s = hsp->next_hsp;
422        break;
423      }
424      prev_hsp = hsp;
425      next_hsp = hsp->next_hsp;
426      hsp = hsp->next_hsp;
427   }
428   return(n_hsps);
429 }
430
431 /*=============================================================*
432  *          free_hsrs          *
433  *=============================================================*/
434 void free_hsrs(struct proc *rp){
435    if(rp->p_nb_hardening_shared_regions <= 0)
436       return;
437    struct hardening_shared_region *hsr =
```

```
438         rp->p_hardening_shared_regions;
439     int lsize = rp->p_nb_hardening_shared_regions;
440     while(hsr){
441         if(free_hsr(rp, hsr->id)!=--lsize)
442             printf("Freeing hsr with error %d\n", lsize);
443   hsr = hsr->next_hsr;
444       }
445     assert(rp->p_nb_hardening_shared_regions == 0);
446     assert(rp->p_hardening_shared_regions == NULL);
447     free_hsp_from_all_hsp_s(rp);
448 }
449
450 void  handle_hsr_events(struct proc *rp){
451     if(rp->p_nb_hardening_shared_regions <= 0)
452         return;
453     struct hardening_shared_region *hsr =
454         rp->p_hardening_shared_regions;
455     while(hsr){
456       printf("#### HARDENING SHARED REGIONS ####"
457               " %d 0x%lx 0x%lx\n",
458               hsr->id, hsr->vaddr, hsr->length);
459       printf("#### REGIONS SHARED WITH ####\n");
460       struct hardening_shared_proc *hsp = hsr->r_hsp;
461       while(hsp){
462           printf("#### Process sharing region"
463               " #### %d %d %d\n",
464               hsp->id, hsp->flags, hsp->hsp_endpoint);
465           hsp = hsp->next_hsp;
466       }
467       hsr = hsr->next_hsr;
468   }
469 }
470
471 int look_up_page_in_hsr(struct proc *rp,
472       vir_bytes vaddr){
473     if(rp->p_nb_hardening_shared_regions <= 0)
474       return(VM_VADDR_NOT_FOUND);
475     if(!(rp->p_hflags & PROC_SHARING_MEM))
476       return(PROC_NOT_SHARING);
477     struct hardening_shared_region *hsr =
478         rp->p_hardening_shared_regions;
479     while(hsr){
480           if((hsr->vaddr <= vaddr) &&
481             (vaddr <= hsr->vaddr + hsr->length)){
482             return(OK);
483           }
484           hsr = hsr->next_hsr;
485     }
486     return(VM_VADDR_NOT_FOUND);
```

```
}


static struct hardening_shared_region *
       look_up_hsr_vaddr(struct proc *rp,
         vir_bytes vaddr, vir_bytes size){
   if(rp->p_nb_hardening_shared_regions <= 0)
     return(NULL);
   struct hardening_shared_region *hsr =
         rp->p_hardening_shared_regions;
   while(hsr){
      if((hsr->vaddr <= (vaddr + size)) &&
           ((vaddr + size) <= hsr->vaddr + hsr->length))
    return(hsr);
      hsr = hsr->next_hsr;
   }
   return(NULL);
}
```

A.3.9 Hardening exception Handler

Listing A.9: Hardening exception Handler

```
#include "kernel/kernel.h"
#include "kernel/vm.h"

#include <machine/vm.h>

#include <minix/type.h>
#include <minix/syslib.h>
#include <minix/cpufeature.h>
#include <string.h>
#include <assert.h>
#include <signal.h>
#include <stdlib.h>


#include <machine/vm.h>


#include "arch_proto.h"

#include "htype.h"
#include "hproto.h"
#include "rcounter.h"
#include "mca.h"
#include "../../clock.h"

```

```
26 #ifdef USE_APIC
27 #include "apic.h"
28 #ifdef USE_WATCHDOG
29 #include "kernel/watchdog.h"
30 #endif
31 #endif
32
33 /*=============================================*
34  *     hardening_exception_handler      *
35  *=============================================*/
36 void hardening_exception_handler(
37      struct exception_frame * frame){
38     /*
39      * running process should be the current
40      * hardenning process
41      * the running process should not be the VM
42      * the hardening should be enable
43      * This function check if the nature of the
44      * exception. The exception is
45      * handled by hardening_exception_handler in
46      * three case
47      *      - NMI, so the exception is come from
48      * the retirmen counter
49      *      - MCA/MCE the exception is come from
50      * the MCA/MCE module
51      *      - A pagefault occur
52      * In all others cases the PE is stopped
53      */
54 #if INJECT_FAULT
55     restore_cr_reg();
56 #endif
57     /**This function should be called
58      * when hardening is enable**/
59     assert(h_enable);
60     /*get the running process*/
61     struct proc  *p = get_cpulocal_var(proc_ptr);
62     /**The running process should
63      * be a hardened process**/
64     assert(h_proc_nr == p->p_nr);
65     assert(h_proc_nr != VM_PROC_NR);
66     h_stop_pe = H_YES;
67     switch(frame->vector){
68        case DIVIDE_VECTOR :
69             break;
70        case DEBUG_VECTOR :
71             ssh();
72             break;
73        case NMI_VECTOR :
74             if(handle_ins_counter_over()!=OK);
```

```
75                h_stop_pe = H_NO;
76              break;
77          case BREAKPOINT_VECTOR:
78              break;
79          case OVERFLOW_VECTOR:
80              break;
81          case BOUNDS_VECTOR:
82              break;
83          case INVAL_OP_VECTOR:
84              break;
85          case COPROC_NOT_VECTOR:
86              break;
87          case DOUBLE_FAULT_VECTOR:
88              break;
89          case COPROC_SEG_VECTOR:
90              break;
91          case INVAL_TSS_VECTOR:
92              break;
93          case SEG_NOT_VECTOR:
94              break;
95          case STACK_FAULT_VECTOR:
96              break;
97          case PROTECTION_VECTOR:
98              break;
99          case PAGE_FAULT_VECTOR :
100             /**reset the hardening data**/
101             /*** ADD COMMENT : TODO */
102             h_normal_pf = 0;
103             h_rw = 0;
104             /**read the virtual address
105              *where the page fault occurred**/
106             reg_t pagefaultcr2 = read_cr2();
107             /*** Check the nature of the pagefault:
108              * if it is caused by hardening
109              **   check_r = OK else
110              *    check_r = H_HANDLED_PF**/
111             int check_r =
112               check_vaddr_2(p,
113                     (u32_t *)p->p_seg.p_cr3,
114                     pagefaultcr2, &h_rw);
115             /**set hardening data to
116              * inform the kernel**/
117             /***  TODO CHANGE THE COMPARISON ****/
118             if(check_r==OK)
119                h_stop_pe = H_NO;
120             else
121               h_normal_pf = NORMAL_PF;
122             break;
123
```

```
124         case COPROC_ERR_VECTOR:
125               break;
126         case ALIGNMENT_CHECK_VECTOR:
127               break;
128         case MACHINE_CHECK_VECTOR:
129               if(mca_mce_handler()==OK)
130                  h_stop_pe = H_NO;
131               break;
132         case SIMD_EXCEPTION_VECTOR:
133               break;
134         default :
135               panic("Unkown exception vector in"
136                 "hardening_exception_handler");
137               break;
138
139     }
140
141     return;
142 }
```

A.3.10 Retirement counter

Listing A.10: Retirement counter

```
1  #include "kernel/kernel.h"
2  #include "kernel/watchdog.h"
3  #include "arch_proto.h"
4  #include "glo.h"
5  #include <minix/minlib.h>
6  #include <minix/u64.h>
7
8  #include "apic.h"
9  #include "apic_asm.h"
10 #include "rcounter.h"
11 #include "hproto.h"
12 #include "htype.h"
13
14 #define PMC_IRQ              230
15
16
17 static uint32_t low, high;
18 /*interrupt handler hook intel_arch_insn_counter_int*/
19 static irq_hook_t pic_intel_arch_insn_counter_hook;
20
21 int register_intel_arch_insn_counter_int_handler(
22     const irq_handler_t handler){
23   /* Using PIC, Initialize the PMC  interrupt hook. */
24   pic_intel_arch_insn_counter_hook.proc_nr_e = NONE;
```

```
25    pic_intel_arch_insn_counter_hook.irq = PMC_IRQ;
26    put_irq_handler(&pic_intel_arch_insn_counter_hook,
27                    PMC_IRQ, handler);
28    return(OK);
29  }
30
31  int intel_arch_insn_counter_int_handler(){
32     ia32_msr_read(INTEL_MSR_PERFMON_CRT1, &high, &low);
33     ia32_msr_read(INTEL_PERF_GLOBAL_STATUS, &high, &low);
34     if(low & INTEL_OVF_PMC0){
35       ia32_msr_write(INTEL_PERF_GLOBAL_OVF_CTRL, 0, 2);
36       lapic_write(LAPIC_LVTPCR, PMC_IRQ);
37     }
38     return(OK);
39  }
40
41
42  void intel_arch_insn_counter_init(void){
43    u32_t val = 0;
44    ia32_msr_write(INTEL_MSR_PERFMON_CRT1, 0, 0);
45    /*Int, OS, USR, Instruction retired*/
46    val = 1 << 20 | 1 << 16 | 0xc0;
47    ia32_msr_write(INTEL_MSR_PERFMON_SEL1, 0, val);
48    ia32_msr_write(INTEL_MSR_PERFMON_SEL1, 0,
49    val | INTEL_MSR_PERFMON_SEL1_ENABLE);
50    /* unmask the performance counter interrupt.
51     * Enable NMI*/
52    lapic_write(LAPIC_LVTPCR, APIC_ICR_DM_NMI);
53  }
54
55  void intel_arch_insn_counter_reinit(void){
56    /*Pour activer le NMI*/
57    lapic_write(LAPIC_LVTPCR, APIC_ICR_DM_NMI);
58    ia32_msr_write(INTEL_MSR_PERFMON_CRT1, 0, 0);
59  }
60
61  void intel_fixed_insn_counter_init(void){
62    u32_t val = 0;
63    ia32_msr_write(INTEL_FIXED_CTR0, 0, 0);
64    val = 1 << 3 | 1 << 1;
65    ia32_msr_write(INTEL_MSR_FIXED_CTR_CTRL, 0, val);
66    /* unmask the performance counter interrupt.
67      * Enable NMI */
68    lapic_write(LAPIC_LVTPCR, APIC_ICR_DM_NMI);
69  }
70
71  void intel_fixed_insn_counter_reset(void){
72      ia32_msr_write(INTEL_MSR_FIXED_CTR_CTRL, 0, 0);
73      ia32_msr_write(INTEL_FIXED_CTR0, 0, 0);
```

```
74 }
75
76 void intel_arch_insn_counter_reset(void){
77     ia32_msr_write(INTEL_MSR_PERFMON_SEL1, 0, 0);
78     ia32_msr_write(INTEL_MSR_PERFMON_CRT1,0 , 0);
79 }
80
81
82 void intel_fixed_insn_counter_enable(void){
83     u32_t val = 0;
84     val = 1 << 3 | 1 << 1;
85     ia32_msr_write(INTEL_MSR_FIXED_CTR_CTRL, 0, val);
86 }
87
88 void intel_arch_insn_counter_enable(void){
89     u32_t val = 0;
90     val = 1 << 20 | 1 << 16 | 0xc0;
91     ia32_msr_write(INTEL_MSR_PERFMON_SEL1, 0, val);
92     ia32_msr_write(INTEL_MSR_PERFMON_SEL1, 0,
93    val | INTEL_MSR_PERFMON_SEL1_ENABLE);
94 }
95
96 void reset_counter(void){
97 /**OFF the retirement counter when the
98  * running process is not a PE**/
99 #if USE_FIX_CTR
100     intel_fixed_insn_counter_reset();
101 #else
102   intel_arch_insn_counter_reset();
103 #endif
104 }
105
106 void enable_counter(void){
107 /**ON the retirement counter when the running
108  * process is a PE**/
109 #if USE_FIX_CTR
110   intel_fixed_insn_counter_enable();
111 #else
112   intel_arch_insn_counter_enable();
113 #endif
114
115 }
116
117 void intel_fixed_insn_counter_reinit(void)
118 {
119   /*Pour activer le NMI*/
120   lapic_write(LAPIC_LVTPCR, APIC_ICR_DM_NMI);
121   ia32_msr_write(INTEL_FIXED_CTR0, 0, 0);
122 }
```

```
123
124 void read_ins_64(u64_t* t){
125   u32_t lo, hi;
126 #if USE_FIX_CTR
127   ia32_msr_read(INTEL_FIXED_CTR0, &hi, &lo);
128 #else
129   ia32_msr_read(INTEL_MSR_PERFMON_CRT1, &hi, &lo);
130 #endif
131   *t = make64 (lo, hi);
132 }
133
134 void update_ins_ctr_switch (){
135  /** Read the retirement counter value and update
136   * the global variable
137   ** __ins_ctr_switch **/
138    u64_t ins;
139    u64_t * __ins_ctr_switch =
140        get_cpulocal_var_ptr(ins_ctr_switch);
141    read_ins_64(&ins);
142    *__ins_ctr_switch = ins;
143 }
144
145
146 void set_remain_ins_counter_value(struct proc *p){
147   intel_arch_insn_counter_reinit();
148   if(p->p_start_count_ins){
149 #if USE_FIX_CTR
150     ia32_msr_write(INTEL_FIXED_CTR0,
151             ex64hi(p->p_remaining_ins),
152            ex64lo(p->p_remaining_ins));
153 #else
154     ia32_msr_write(INTEL_MSR_PERFMON_CRT1,
155             ex64hi(p->p_remaining_ins),
156          ex64lo(p->p_remaining_ins));
157 #endif
158     make_zero64(p->p_remaining_ins);
159
160   }
161   else{
162 #if USE_FIX_CTR
163      ia32_msr_write(INTEL_FIXED_CTR0, 0xff,
164   INS_THRESHOLD-ex64lo(INS_2_EXEC)+1);
165 #else
166      ia32_msr_write(INTEL_MSR_PERFMON_CRT1, 0xffff,
167   INS_THRESHOLD-ex64lo(INS_2_EXEC)+1);
168 #endif
169      p->p_start_count_ins = 1;
170
171   }
```

```
172   update_ins_ctr_switch ();
173 }
174
175 void set_remain_ins_counter_value_0(struct proc *p){
176 /** Before starting one of the executions of the PE,
177   * this function
178   ** initializes the retirement counter to
179   ** (Overflow value - Maximum number of
180   * instruction of the PE) **/
181    u64_t ins;
182 #if USE_FIX_CTR
183    ia32_msr_write(INTEL_FIXED_CTR0, INS_THRESHOLD,
184   INS_THRESHOLD-ex64lo(INS_2_EXEC)+1);
185 #else
186    ia32_msr_write(INTEL_MSR_PERFMON_CRT1, INS_THRESHOLD,
187   INS_THRESHOLD-ex64lo(INS_2_EXEC)+1);
188 #endif
189          p->p_start_count_ins = 1;
190          read_ins_64(&ins);
191          p->p_ins_last = ins;
192          update_ins_ctr_switch ();
193          get_remain_ins_counter_value_0(p);
194 }
195
196 void set_remain_ins_counter_value_1(struct proc *p){
197 /** Before resuming the PE after an interrupt or
198   *  exception, this
199   ** function updates the retirement counter to the value
200   ** saved in p_remaining_ins**/
201 #if 1
202 #if USE_FIX_CTR
203    ia32_msr_write(INTEL_FIXED_CTR0,
204         ex64hi(p->p_remaining_ins),
205                        ex64lo(p->p_remaining_ins));
206 #else
207    ia32_msr_write(INTEL_MSR_PERFMON_CRT1,
208        ex64hi(p->p_remaining_ins),
209        ex64lo(p->p_remaining_ins));
210 #endif
211
212    make_zero64(p->p_remaining_ins);
213    update_ins_ctr_switch ();
214 #endif
215
216 }
217
218
219 void get_remain_ins_counter_value(struct proc *p){
220 /** The PE is stopped read the value of the
```

```
221  ** retriement counter
222  ** Store that value in __ins_ctr_switch and
223  **  p_remaining_ins
224  ** 2 storages to keep track of the remaining
225   ** instructions of the PE **/
226    u64_t ins;
227    u64_t * __ins_ctr_switch =
228      get_cpulocal_var_ptr(ins_ctr_switch);
229    read_ins_64(&ins);
230    *__ins_ctr_switch = ins;
231    p->p_remaining_ins =ins;
232    p->p_ins_last = p->p_remaining_ins;
233 }
234
235 void get_remain_ins_counter_value_0(struct proc *p){
236 /** The PE is stopped read the value of
237  *  the retriement counter
238  ** Store that value in __ins_ctr_switch
239  *  and p_remaining_ins
240  ** 2 storages to keep track of the remaining
241  ** instructions of the PE **/
242    u64_t ins;
243    u64_t * __ins_ctr_switch =
244          get_cpulocal_var_ptr(ins_ctr_switch);
245    read_ins_64(&ins);
246    *__ins_ctr_switch = ins;
247    p->p_remaining_ins =ins;
248 }
249
250
251
252 int irh(void){
253 /** Handle the NMI:
254  ** -- Clear the overflow flag
255  ** -- Reinit the NMI
256  ** -- Make tke PE not runnable
257  **/
258   struct proc * p = get_cpulocal_var(proc_ptr);
259   ia32_msr_read(INTEL_PERF_GLOBAL_STATUS, &high, &low);
260 #if USE_FIX_CTR
261   if(high & INTEL_OVF_FIXED_CTR0){
262 #else
263   if(low & INTEL_OVF_PMC0){
264 #endif
265       /**the counter overflowed try to clear
266      register's flags**/
267 #if USE_FIX_CTR
268        ia32_msr_write(INTEL_PERF_GLOBAL_OVF_CTRL,
269         INTEL_OVF_FIXED_CTR0, 0);
```

```
270         intel_fixed_insn_counter_reinit();
271 #else
272         ia32_msr_write(INTEL_PERF_GLOBAL_OVF_CTRL, 0,
273         INTEL_OVF_PMC0);
274         intel_arch_insn_counter_reinit();
275 #endif
276
277         if(h_step==FIRST_RUN)
278             first_run_ins = p->p_ins_last;
279         if(h_step==SECOND_RUN)
280             secnd_run_ins = p->p_ins_last;
281         origin_syscall = PE_END_IN_NMI;
282         return(OK);
283     }
284     return(EFAULT);
285 }
286
287 void reset_ins_params(struct proc *p){
288     make_zero64(p->p_ins_last);
289     make_zero64(p->p_remaining_ins);
290     if(h_step==FIRST_RUN)
291         p->p_ins_first = 0;
292     else
293         p->p_ins_secnd = 0;
294 }
```

A.3.11 Machine check architecture

Listing A.11: Machine check architecture

```
1  /*** Here the MCA/MCE is implemented
2   ***    Author: Emery Kouassi Assogba
3   ***    Email: assogba.emery@gmail.com
4   ***    Phone: 0022995222073
5   ***    22/01/2019 lln***/
6
7  #include "kernel/kernel.h"
8  #include "kernel/vm.h"
9
10 #include <machine/vm.h>
11
12 #include <minix/type.h>
13 #include <minix/syslib.h>
14 #include <minix/cpufeature.h>
15 #include <string.h>
16 #include <assert.h>
17 #include <signal.h>
18 #include <stdlib.h>
```

```
19
20
21 #include <machine/vm.h>
22
23
24 #include "arch_proto.h"
25 #include "mca.h"
26 #include "htype.h"
27
28
29 static int nbanks;
30 static int MAX_BANKS_NUMBER;
31
32 int enables_all_mca_features(void){
33   int status = N_OK;
34   u32_t low, high;
35   ia32_msr_read(IA32_MCG_CAP, &high, &low);
36   if (low & MCG_CTL_P){
37     nbanks = get_n_banks(low);
38     MAX_BANKS_NUMBER = nbanks - 1;
39     ia32_msr_read(IA32_MCG_CTL, &high, &low);
40     ia32_msr_write(IA32_MCG_CTL, ALL_1s, ALL_1s);
41     ia32_msr_read(IA32_MCG_CTL, &high, &low);
42     return(OK);
43   }
44   return(status);
45 }
46
47
48 void enable_loggin_ofall_errors(void){
49    int i, t = 0;
50    if (cpu_info[CONFIG_MAX_CPUS].family == 0x6 &&
51           cpu_info[CONFIG_MAX_CPUS].model < 0x1A)
52      t = 1;
53    for(i=t; i< nbanks ;i++){
54       ia32_msr_write(IA32_MC0_CTL+4*i, ALL_1s, ALL_1s);
55    }
56 }
57
58 void clears_all_errors(void){
59   int i;
60   for(i=0;i<nbanks;i++){
61     ia32_msr_write(IA32_MC0_STATUS+4*i, 0, 0);
62   }
63 }
64
65 void enable_machine_check_exception(void){
66   u32_t cr4 = read_cr4() | CR4_MCE;
67   write_cr4(cr4);
```

```
68 }
69
70 int mca_mce_handler(void){
71   int i;
72   u32_t low, high;
73   ia32_msr_read(IA32_MCG_STATUS,&high,&low);
74   if(low & MCIP){ /* ##### GOT MCA EXCEPTION ###*/
75       /*PE can continue error was corrected*/
76       if(low & RIPV)
77         return(OK);
78   }
79   return(EFAULT);
80 }
```

A.3.12 Single stepping handler

Listing A.12: Single stepping handler

```
1  #include "kernel/kernel.h"
2  #include "kernel/vm.h"
3
4  #include <machine/vm.h>
5
6  #include <minix/type.h>
7  #include <minix/syslib.h>
8  #include <minix/cpufeature.h>
9  #include <string.h>
10 #include <assert.h>
11 #include <signal.h>
12 #include <stdlib.h>
13
14
15 #include <machine/vm.h>
16
17
18 #include "arch_proto.h"
19
20 #include "htype.h"
21 #include "hproto.h"
22 #include "rcounter.h"
23 #include "mca.h"
24 #include "../../clock.h"
25
26 #ifdef USE_APIC
27 #include "apic.h"
28 #ifdef USE_WATCHDOG
29 #include "kernel/watchdog.h"
30 #endif
```

```
#endif

int ssh(struct proc *p){
    if(!h_ss_mode)
        return(OK);
    origin_syscall = PE_END_IN_NMI;
    p->p_reg.psw &= ~TRACEBIT;
    save_context(p);
    if(h_step==FIRST_STEPPING)
       first_run_ins++;
    else
       secnd_run_ins++;
    if( (secnd_run_ins!= first_run_ins) &&!cmp_reg(p)){
        h_stop_pe = H_NO;
        p->p_misc_flags |= MF_STEP;
        h_unstable_state = H_STEPPING;
        p->p_reg.psw &= ~TRACEBIT;
        return(EFAULT);
    }
    else{
        p->p_misc_flags &= ~MF_STEP;
        h_unstable_state = H_STEPPING;
        p->p_reg.psw &= ~TRACEBIT;
        if(h_step==FIRST_STEPPING)
            h_step=SECOND_RUN;
         return(OK);
     }
}

void ssh_init(struct proc *p){
  p->p_nb_ss++;
  h_ss_mode = 1;
  if(secnd_run_ins < first_run_ins){
     p->p_misc_flags |= MF_STEP;
     h_unstable_state = H_STEPPING;
     return;
  }

  if(secnd_run_ins > first_run_ins){
    vm_reset_pram(p, (u32_t *)p->p_seg.p_cr3,
                  CPY_RAM_FIRST_STEPPING);
    restore_for_stepping_first_run(p);
    p->p_misc_flags |= MF_STEP;
    h_unstable_state = H_STEPPING;
    h_step = FIRST_STEPPING;
    return;
  }

}
```

A.3.13 Hardening software and the micro-kernel

Listing A.13: Hardening software and the micro-kernel

```
/*===========================================*
 *    do_fork                                *
 *===========================================*/
int do_fork(struct proc * caller, message * m_ptr)
{
/* Handle sys_fork().
 * m_lsys_krn_sys_fork.endpt has forked.
 * The child is m_lsys_krn_sys_fork.slot.
 */
#if defined(__i386__)
  char *old_fpu_save_area_p;
#endif
  /* child process pointer */
  register struct proc *rpc;
  /* parent process pointer */
  struct proc *rpp;
  int gen;
  int p_proc;
  int namelen;
  static int hcount = 0;

  if(!isokendpt(
      m_ptr->m_lsys_krn_sys_fork.endpt, &p_proc))
  return EINVAL;

  rpp = proc_addr(p_proc);
  rpc = proc_addr(m_ptr->m_lsys_krn_sys_fork.slot);
  if (isemptyp(rpp) || ! isemptyp(rpc)) return(EINVAL);

  assert(!(rpp->p_misc_flags & MF_DELIVERMSG));

  /* needs to be receiving so we
   * know where the message buffer is */
  if(!RTS_ISSET(rpp, RTS_RECEIVING)) {
     printf("kernel: fork not done synchronously?\n");
     return EINVAL;
  }

  /* make sure that the FPU context
   * is saved in parent before copy */
  save_fpu(rpp);
  /* Copy parent 'proc' struct to child.
   * And reinitialize some fields. */
  gen = _ENDPOINT_G(rpc->p_endpoint);
#if defined(__i386__)
```

```
46    old_fpu_save_area_p = rpc->p_seg.fpu_state;
47  #endif
48    /* copy 'proc' struct */
49    *rpc = *rpp;
50  #if defined(__i386__)
51    rpc->p_seg.fpu_state = old_fpu_save_area_p;
52    if(proc_used_fpu(rpp))
53    memcpy(rpc->p_seg.fpu_state,
54            rpp->p_seg.fpu_state, FPU_XFP_SIZE);
55  #endif
56    /* increase generation */
57    if(++gen >= _ENDPOINT_MAX_GENERATION)
58      /* generation number wraparound */
59    gen = 1;
60    /* this was obliterated by copy */
61    rpc->p_nr = m_ptr->m_lsys_krn_sys_fork.slot;
62    /* new endpoint of slot */
63    rpc->p_endpoint = _ENDPOINT(gen, rpc->p_nr);
64    /* child sees pid = 0 to know it is child */
65    rpc->p_reg.retreg = 0;
66    /* set all the accounting times to 0 */
67    rpc->p_user_time = 0;
68    rpc->p_sys_time = 0;
69
70    rpc->p_misc_flags &=
71    ~(MF_VIRT_TIMER | MF_PROF_TIMER
72        | MF_SC_TRACE | MF_SPROF_SEEN | MF_STEP);
73    /* disable, clear the process-virtual timers */
74    rpc->p_virt_left = 0;
75    rpc->p_prof_left = 0;
76
77    /* Mark process name as being a forked copy */
78    namelen = strlen(rpc->p_name);
79  #define FORKSTR "*F"
80    if(namelen+strlen(FORKSTR) < sizeof(rpc->p_name))
81    strcat(rpc->p_name, FORKSTR);
82
83    /* the child process is not
84     * runnable until it's scheduled. */
85    RTS_SET(rpc, RTS_NO_QUANTUM);
86    reset_proc_accounting(rpc);
87
88    rpc->p_cpu_time_left = 0;
89    rpc->p_cycles = 0;
90    rpc->p_kcall_cycles = 0;
91    rpc->p_kipc_cycles = 0;
92
93    rpc->p_tick_cycles = 0;
94    cpuavg_init(&rpc->p_cpuavg);
```

```
95
96    /* If the parent is a privileged process,
97     * take away the privileges from the
98     * child process and inhibit it from running
99     *  by setting the NO_PRIV flag.
100    * The caller should explicitly set the
101    * new privileges before executing.
102    */
103   if (priv(rpp)->s_flags & SYS_PROC) {
104       rpc->p_priv = priv_addr(USER_PRIV_ID);
105       rpc->p_rts_flags |= RTS_NO_PRIV;
106   }
107
108   /* Calculate endpoint identifier,
109    * so caller knows what it is. */
110   m_ptr->m_krn_lsys_sys_fork.endpt = rpc->p_endpoint;
111   m_ptr->m_krn_lsys_sys_fork.msgaddr =
112                rpp->p_delivermsg_vir;
113
114   /* Don't schedule process in VM mode
115    * until it has a new pagetable. */
116   if(m_ptr->m_lsys_krn_sys_fork.flags & PFF_VMINHIBIT){
117     RTS_SET(rpc, RTS_VMINHIBIT);
118   }
119
120   /*
121    * Only one in group should have
122    * RTS_SIGNALED, child doesn't inherit tracing.
123    */
124   RTS_UNSET(rpc, (RTS_SIGNALED |
125              RTS_SIG_PENDING | RTS_P_STOP));
126   (void) sigemptyset(&rpc->p_pending);
127
128#if defined(__i386__)
129   rpc->p_seg.p_cr3 = 0;
130   rpc->p_seg.p_cr3_v = NULL;
131#elif defined(__arm__)
132   rpc->p_seg.p_ttbr = 0;
133   rpc->p_seg.p_ttbr_v = NULL;
134#endif
135/*** Add by EKA**/
136   rpc->p_setcow = 0;
137   rpc->p_hflags = 0;
138   if(hprocs_in_use <= H_NPROCS_TO_START_H)
139   hprocs_in_use++;
140
141   if(hprocs_in_use > 800){
142       rpc->p_setcow = 1;
143   }
```

```
144
145   if(h_can_start_hardening){
146       rpc->p_hflags |= PROC_TO_HARD;
147       hc_proc_nr[(hcount++)%10] = rpc->p_nr;
148   }
149   rpc->p_first_step_workingset_id = 0;
150   rpc->p_working_set = NULL;
151   rpc->p_hardening_mem_events = NULL;
152   rpc->p_nb_hardening_mem_events = 0;
153   rpc->p_hardening_shared_regions = NULL;
154   rpc->p_nb_hardening_shared_regions = 0;
155   if(rpp->p_hflags & PROC_TO_HARD){
156 #if H_DEBUG
157       display_mem(rpp);
158 #endif
159       free_pram_mem_blocks(rpp, 0);
160   }
161 /** End Add by EKA**/
162   return OK;
163 }
164 #endif /* USE_FORK */
165
166 /* The kernel call implemented in this file:
167  *   m_type:  SYS_EXEC
168  *
169  * The parameters for this kernel call are:
170  *   m_lsys_krn_sys_exec.endpt
171  *     (process that did exec call)
172  *   m_lsys_krn_sys_exec.stack
173  *     (new stack pointer)
174  *   m_lsys_krn_sys_exec.name
175  *     (pointer to program name)
176  *   m_lsys_krn_sys_exec.ip
177  *     (new instruction pointer)
178  *   m_lsys_krn_sys_exec.ps_str
179  *      (struct ps_strings *)
180  */
181 #include "kernel/system.h"
182 #include <string.h>
183 #include <minix/endpoint.h>
184 #include "kernel/arch/i386/hproto.h"
185 #include "kernel/arch/i386/htype.h"
186
187 #if USE_EXEC
188
189 /*===========================================================*
190  *    do_exec                                *
191  *===========================================================*/
192 int do_exec(struct proc * caller, message * m_ptr)
```

```
193 {
194 /* Handle sys_exec().  A process has done
195  * a successful EXEC. Patch it up. */
196   register struct proc *rp;
197   int proc_nr;
198   char name[PROC_NAME_LEN];
199 
200   if(!isokendpt(
201      m_ptr->m_lsys_krn_sys_exec.endpt, &proc_nr))
202   return EINVAL;
203 
204   rp = proc_addr(proc_nr);
205 
206   if(rp->p_misc_flags & MF_DELIVERMSG) {
207   rp->p_misc_flags &= ~MF_DELIVERMSG;
208   }
209 
210 
211   /** Add by EKA: free the PE working set list ***/
212     if(rp->p_hflags & PROC_TO_HARD){
213       free_pram_mem_blocks(rp,FROM_EXEC);
214     }
215   /**End Add by EKA**/
216 
217 
218   /* Save command name for debugging,
219           ps(1) output, etc. */
220   if(data_copy(caller->p_endpoint,
221         m_ptr->m_lsys_krn_sys_exec.name,
222   KERNEL, (vir_bytes) name,
223   (phys_bytes) sizeof(name) - 1) != OK)
224     strncpy(name, "<unset>", PROC_NAME_LEN);
225 
226   name[sizeof(name)-1] = '\0';
227 
228   /* Set process state. */
229   arch_proc_init(rp,
230     (u32_t) m_ptr->m_lsys_krn_sys_exec.ip,
231     (u32_t) m_ptr->m_lsys_krn_sys_exec.stack,
232     (u32_t) m_ptr->m_lsys_krn_sys_exec.ps_str, name);
233 
234   /* No reply to EXEC call */
235   RTS_UNSET(rp, RTS_RECEIVING);
236 
237   /* Mark fpu_regs contents as not significant, so fpu
238    * will be initialized, when it's used next time. */
239   rp->p_misc_flags &= ~MF_FPU_INITIALIZED;
240   /* force reloading FPU if the
241    * current process is the owner */
```

```
242   release_fpu(rp);
243   return(OK);
244 }
245 #endif /* USE_EXEC */
246 /* The kernel call implemented in this file:
247  *   m_type:  SYS_CLEAR
248  *
249  * The parameters for this kernel call are:
250  *   m_lsys_krn_sys_clear.endpt
251  * (endpoint of process to clean up)
252  */
253
254 /* The kernel call implemented in this file:
255  *   m_type:  SYS_CLEAR
256  *
257  * The parameters for this kernel call are:
258  *   m_lsys_krn_sys_clear.endpt
259  * (endpoint of process to clean up)
260  */
261
262 #include "kernel/system.h"
263
264 #include <minix/endpoint.h>
265 #include "kernel/arch/i386/hproto.h"
266 #include "kernel/arch/i386/htype.h"
267
268 #if USE_CLEAR
269
270 /*===========================================================*
271  *        do_clear       *
272  *===========================================================*/
273 int do_clear(struct proc * caller, message * m_ptr)
274 {
275 /* Handle sys_clear. Only the PM can request
276  * other process slots to be cleared
277  * when a process has exited.
278  * The routine to clean up a process table
279  * slot cancels outstanding timers,
280  * possibly removes the process from the message
281  * queues, and resets certain
282  * process table fields to the default values.
283  */
284   struct proc *rc;
285   int exit_p;
286   int i;
287
288   if(!isokendpt(
289     m_ptr->m_lsys_krn_sys_clear.endpt, &exit_p)) {
290       /* get exiting process */
```

```
291         return EINVAL;
292     }
293     rc = proc_addr(exit_p); /* clean up */
294 
295     release_address_space(rc);
296 
297     /* Don't clear if already cleared. */
298     if(isemptyp(rc)) return OK;
299 
300     /* Check the table with IRQ hooks
301      * to see if hooks should be released. */
302     for (i=0; i < NR_IRQ_HOOKS; i++) {
303         if (rc->p_endpoint == irq_hooks[i].proc_nr_e) {
304           /* remove interrupt handler */
305           rm_irq_handler(&irq_hooks[i]);
306           /* mark hook as free */
307           irq_hooks[i].proc_nr_e = NONE;
308         }
309     }
310     /* Remove the process' ability
311      * to send and receive messages */
312     clear_endpoint(rc);
313 
314     /* Turn off any alarm timers at the clock. */
315     reset_kernel_timer(&priv(rc)->s_alarm_timer);
316 
317     /* Add by EKA: free the PE working set list */
318      free_pram_mem_blocks(rc, 1);
319      handle_hsr_events(rc);
320      free_hsrs(rc);
321     /**End Add by EKA**/
322 
323     /* Make sure that the exiting process is no
324      * longer scheduled,
325      * and mark slot as FREE. Also mark saved fpu
326      * contents as not significant.
327      */
328     RTS_SETFLAGS(rc, RTS_SLOT_FREE);
329     /* release FPU */
330     release_fpu(rc);
331     rc->p_misc_flags &= ~MF_FPU_INITIALIZED;
332     /* Release the process table slot.
333      * If this is a system process, also
334      * release its privilege structure.  Further
335      * cleanup is not needed at
336      * this point. All important fields are
337      * reinitialized when the
338      * slots are assigned to another, new process.
339      */
```

```
340   if (priv(rc)->s_flags & SYS_PROC)
341     priv(rc)->s_proc_nr = NONE;
342   return OK;
343 }
344 
345 #endif /* USE_CLEAR */
346 
347 
348 int do_ipc(reg_t r1, reg_t r2, reg_t r3)
349 {
350   /* get pointer to caller */
351   struct proc *const caller_ptr =
352       get_cpulocal_var(proc_ptr);
353   int call_nr = (int) r1;
354   /*** Add by EKA ***/
355 
356   if(h_unstable_state == H_UNSTABLE){
357       return(OK);
358   }
359   caller_ptr->p_setcow = 1;
360   /*** End add by EKA ***/
361 
362   assert(!RTS_ISSET(caller_ptr, RTS_SLOT_FREE));
363 
364   /* bill kernel time to this process. */
365   kbill_ipc = caller_ptr;
366 
367   /* If this process is subject to system
368    * call tracing, handle that first. */
369   if (caller_ptr->p_misc_flags &
370             (MF_SC_TRACE | MF_SC_DEFER)) {
371   /* Are we tracing this process, and
372         is it the first sys_call entry? */
373   if ((caller_ptr->p_misc_flags &
374           (MF_SC_TRACE | MF_SC_DEFER)) == MF_SC_TRACE){
375   /* We must notify the tracer before
376         * processing the actual
377         * system call. If we don't, the tracer
378         * could not obtain the
379         * input message. Postpone
380         * the entire system call.
381    */
382       caller_ptr->p_misc_flags &= ~MF_SC_TRACE;
383       assert(!
384              (caller_ptr->p_misc_flags & MF_SC_DEFER));
385       caller_ptr->p_misc_flags |= MF_SC_DEFER;
386       caller_ptr->p_defer.r1 = r1;
387       caller_ptr->p_defer.r2 = r2;
388       caller_ptr->p_defer.r3 = r3;
```

```
389     /* Signal the "enter system call" event.
390              Block the process. */
391      cause_sig(proc_nr(caller_ptr), SIGTRAP);
392     /* Preserve the return register's value. */
393     return caller_ptr->p_reg.retreg;
394   }
395
396   /* If the MF_SC_DEFER flag is set,
397          * the syscall is now being resumed. */
398   caller_ptr->p_misc_flags &= ~MF_SC_DEFER;
399
400   assert (
401           !(caller_ptr->p_misc_flags & MF_SC_ACTIVE));
402
403   /* Set a flag to allow reliable tracing of
404            leaving the system call. */
405   caller_ptr->p_misc_flags |= MF_SC_ACTIVE;
406   }
407
408   if(caller_ptr->p_misc_flags & MF_DELIVERMSG) {
409       panic("sys_call: MF_DELIVERMSG on for %s / %d\n",
410   caller_ptr->p_name, caller_ptr->p_endpoint);
411   }
412
413   /* Now check if the call is known and try to perform
414    * the request. The only
415    * system calls that exist in MINIX are sending and
416    * receiving messages.
417    *   - SENDREC: combines SEND and RECEIVE in a single
418    * system call
419    *   - SEND:    sender blocks until its message has
420    * been delivered
421    *   - RECEIVE: receiver blocks until an acceptable
422    * message has arrived
423    *   - NOTIFY:  asynchronous call; deliver
424    * notification or mark pending
425    *   - SENDA:   list of asynchronous send requests
426    */
427   switch(call_nr) {
428     case SENDREC:
429     case SEND:
430     case RECEIVE:
431     case NOTIFY:
432     case SENDNB:
433     {
434         /* Process accounting for scheduling */
435       caller_ptr->p_accounting.ipc_sync++;
436
437         return do_sync_ipc(caller_ptr, call_nr,
```

```
438                 (endpoint_t) r2,
439             (message *) r3);
440       }
441       case SENDA:
442       {
443         /*
444             * Get and check the size of the argument
445                 * in bytes as it is a
446             * table
447             */
448           size_t msg_size = (size_t) r2;
449
450           /* Process accounting for scheduling */
451         caller_ptr->p_accounting.ipc_async++;
452
453           /* Limit size to something reasonable.
454                 * An arbitrary choice is 16
455             * times the number of process table
456                 * entries.
457             */
458           if (msg_size > 16*(NR_TASKS + NR_PROCS))
459             return EDOM;
460           return mini_senda(caller_ptr,
461                     (asynmsg_t *) r3, msg_size);
462       }
463       case MINIX_KERNINFO:
464     {
465       /* It might not be initialized yet. */
466         if(!minix_kerninfo_user) {
467         return EBADCALL;
468       }
469
470         arch_set_secondary_ipc_return(caller_ptr,
471                         minix_kerninfo_user);
472         return OK;
473     }
474       default:
475     return EBADCALL;/* illegal system call */
476     }
477 }
478
479 /*===========================================================*
480  *      kernel_call                      *
481  *===========================================================*/
482 /*
483  * this function checks the basic syscall parameters
484  * and if accepted it
485  * dispatches its handling to the right handler
486  */
```

```
487 void kernel_call(message *m_user, struct proc * caller)
488 {
489   int result = OK;
490   message msg;
491 
492   caller->p_delivermsg_vir = (vir_bytes) m_user;
493 
494   if(h_unstable_state == H_UNSTABLE){
495         printf("ALERT ALERT FROM KERNEL CALL "
496         "!!!!! \n "
497                "The system is in unstable state"
498          " The guilty is %d\n", h_proc_nr);
499         return(OK);
500   }
501   /*
502    * the ldt and cr3 of the caller process is loaded
503    * because it just've trapped
504    * into the kernel or was already set in
505    * switch_to_user() before we resume
506    * execution of an interrupted kernel call
507    */
508   if (copy_msg_from_user(m_user, &msg) == 0) {
509     msg.m_source = caller->p_endpoint;
510     result = kernel_call_dispatch(caller, &msg);
511   }
512   else {
513     printf("WARNING wrong user pointer "
514                    "0x%08x from process %s / %d\n",
515          m_user, caller->p_name,
516                  caller->p_endpoint);
517     cause_sig(proc_nr(caller), SIGSEGV);
518     return;
519   }
520 
521 
522   /* remember who invoked the kcall so we
523    * can bill it its time */
524   kbill_kcall = caller;
525 
526   kernel_call_finish(caller, &msg, result);
527 }
528 
529 /*===========================================================*
530  *    hpick_proc                                             *
531  *===========================================================*/
532 static struct proc * hpick_proc(void){
533 
534   /* This function is called when a new process
535    * should be chosen to run
```

```
536     * In hardened MINIX3, if this function is called
537     * while a hardened PE was
538     * running (h_enable is true).
539     *  1 - The VM will be returned if the hardened
540     *      process does a page fault and
541     *       the pagefault flags is set.
542     *  2-  The hardened process will be chosen it
543     *      is is runnable
544     *       2-1- If the hardened process doesn't
545     *             have enough quantum,
546     *             The PE is reset
547     *       2-2- If the hardened process was stopped
548     *      by retirement counter,
549     *           The flags is reset
550     *  3- In all others cases a panic is triggered
551     */
552
553    register struct proc *hp = proc_addr(h_proc_nr);
554    if(h_enable){
555         register struct proc *hp = proc_addr(h_proc_nr);
556           /* hardening is enable, only VM or
557            * KERNEL could run
558            * allow for a VM to run for an instant
559            * because of the page fault*/
560         if(h_step == VM_RUN){
561            if((hp->p_rts_flags & RTS_PAGEFAULT)){
562              if(!proc_is_runnable(proc_addr(VM_PROC_NR)))
563                 panic("hardening page fault but "
564                       "VM is not runnable\n");
565                 /**Choose the VM to handle
566                  * the page fault**/
567                 else
568                     return proc_addr(VM_PROC_NR);
569            }
570            else{
571            /**The hardening process is
572              runnable choose it**/
573              if(proc_is_runnable(hp)){
574                 assert((h_step_back == FIRST_RUN) ||
575                        (h_step_back == SECOND_RUN));
576                 h_step = h_step_back;
577                 h_step_back = 0;
578                 if (priv(hp)->s_flags & BILLABLE)
579      get_cpulocal_var(bill_ptr) = hp;
580                 return hp;
581              }
582              else
583                 panic("Hardened process is not"
584                  " runnable after page fault"
```

```
585                         " handled 1\n");
586           }
587         }
588         else {
589            /** The hardening process is not runnable
590             ** because it ends its quantum.
591             ** We choose to give him quantum**/
592             if(RTS_ISSET(hp, RTS_NO_QUANTUM)) {
593                /** abort pe**/
594              abort_pe(hp);
595              return(NULL);
596
597             }
598             if(hp->p_rts_flags & RTS_INS_COUNTER){
599                 RTS_UNSET(hp,RTS_INS_COUNTER);
600                  if(proc_is_runnable(hp)){
601                       if (priv(hp)->s_flags & BILLABLE)
602              get_cpulocal_var(bill_ptr) = hp;
603                       return(hp);
604                  }
605                  /** We UNSET RTS_INS_COUNTER but the
606                   * process is not runnable
607                   ** panic, should never happen **/
608                  else
609                    panic ("Hardened process is not "
610                         "runnable during hardening "
611                            "enable 3");
612             }
613             if(proc_is_runnable(hp)) {
614                 if (priv(hp)->s_flags & BILLABLE)
615      get_cpulocal_var(bill_ptr) = hp;
616                 return(hp);
617           }
618           else
619              panic ("Hardened process is not"
620                     " runnable during hardening "
621                        "enable 4");
622         }
623   }
624   return(NULL);
625
626   /*Fin modification*/
627 }
628
629 static struct proc * pick_proc(void)
630 {
631 /* Decide who to run now.  A new process is
632  * selected and returned.
633  * When a billable process is selected, record it
```

```
634  * in 'bill_ptr', so that the
635  * clock task can tell who to bill for system time.
636  *
637  * This function always uses the run queues
638  * of the local cpu!
639  */
640   register struct proc *rp;/* process to run */
641   struct proc **rdy_head;
642   int q;  /* iterate over queues */
643
644   /* Check each of the scheduling queues for
645    * ready processes. The number of
646    * queues is defined in proc.h, and priorities
647    * are set in the task table.
648    * If there are no processes ready to run, return NULL.
649    */
650
651
652   /**Can we resume the PE?**/
653   if((rp = hpick_proc())) return(rp);
654
655   rdy_head = get_cpulocal_var(run_q_head);
656   for (q=0; q < NR_SCHED_QUEUES; q++) {
657   if(!(rp = rdy_head[q])) {
658     TRACE(VF_PICKPROC,
659           printf("cpu %d queue %d empty\n", cpuid, q););
660     continue;
661   }
662   assert(proc_is_runnable(rp));
663   if (priv(rp)->s_flags & BILLABLE)
664     get_cpulocal_var(bill_ptr) =
665            rp; /* bill for system time */
666#if H_DEBUG
667         if(0 && h_can_start_hardening)
668             print_rdyqueue();
669#endif
670   return rp;
671   }
672   return NULL;
673 }
674
675 /*===========================================================*
676  *    switch_to_user                                         *
677  *===========================================================*/
678 void switch_to_user(void)
679 {
680   /* This function is called an
681         * instant before proc_ptr is
682    * to be scheduled again.
```

```
683     */
684    struct proc * p;
685 #ifdef CONFIG_SMP
686    int tlb_must_refresh = 0;
687 #endif
688 
689    p = get_cpulocal_var(proc_ptr);
690    /*
691     * if the current process is still runnable
692          * check the misc flags and let
693     * it run unless it becomes not runnable in
694          * the meantime
695     */
696    if (proc_is_runnable(p))
697      goto check_misc_flags;
698    /*
699     * if a process becomes not runnable while
700          * handling the misc flags, we
701     * need to pick a new one here and start from
702          * scratch. Also if the
703     * current process wasn't runnable, we pick
704          * a new one here
705     */
706 not_runnable_pick_new:
707    if (proc_is_preempted(p)) {
708      p->p_rts_flags &= ~RTS_PREEMPTED;
709      if (proc_is_runnable(p)) {
710        if (p->p_cpu_time_left)
711          enqueue_head(p);
712        else
713          enqueue(p);
714      }
715    }
716 
717    /*
718     * if we have no process to run, set IDLE as
719          * the current process for
720     * time accounting and put the cpu in an idle
721          * state. After the next
722     * timer interrupt the execution resumes here
723          * and we can pick another
724     * process. If there is still nothing runnable
725          * we "schedule" IDLE again
726     */
727    while (!(p = pick_proc())) {
728      idle();
729    }
730 
731    /* update the global variable */
```

```
732   get_cpulocal_var(proc_ptr) = p;
733
734 #ifdef CONFIG_SMP
735   if (p->p_misc_flags & MF_FLUSH_TLB &&
736                       get_cpulocal_var(ptproc) == p)
737     tlb_must_refresh = 1;
738 #endif
739   switch_address_space(p);
740
741 check_misc_flags:
742
743   assert(p);
744   assert(proc_is_runnable(p));
745   while (p->p_misc_flags &
746     (MF_KCALL_RESUME | MF_DELIVERMSG |
747      MF_SC_DEFER |
748                   MF_SC_TRACE | MF_SC_ACTIVE)) {
749
750     assert(proc_is_runnable(p));
751     if (p->p_misc_flags & MF_KCALL_RESUME) {
752       kernel_call_resume(p);
753     }
754     else
755                     if (p->p_misc_flags & MF_DELIVERMSG) {
756       TRACE(VF_SCHEDULING,
757                     printf("delivering to %s / %d\n",
758             p->p_name, p->p_endpoint););
759       delivermsg(p);
760     }
761     else if (p->p_misc_flags & MF_SC_DEFER) {
762       /* Perform the system call that
763                         * we deferred earlier. */
764
765         assert (!(p->p_misc_flags & MF_SC_ACTIVE));
766
767         arch_do_syscall(p);
768
769         /* If the process is stopped for signal
770                     * delivery, and
771               * not blocked sending a message after
772                     * the system call,
773          * inform PM.
774          */
775         if ((p->p_misc_flags & MF_SIG_DELAY)
776                         && !RTS_ISSET(p, RTS_SENDING))
777          sig_delay_done(p);
778     }
779     else if (p->p_misc_flags & MF_SC_TRACE){
780     /* Trigger a system call leave event
```

```
781                     * if this was a
782          * system call. We must do this after
783                     * processing the
784          * other flags above, both for
785                     * tracing correctness and
786          * to be able to use 'break'.
787          */
788          if (!(p->p_misc_flags & MF_SC_ACTIVE))
789              break;
790          p->p_misc_flags &=
791              ~(MF_SC_TRACE | MF_SC_ACTIVE);
792
793          /* Signal the "leave system call" event.
794           * Block the process.
795           */
796          cause_sig(proc_nr(p), SIGTRAP);
797        }
798        else if (p->p_misc_flags & MF_SC_ACTIVE) {
799          /* If MF_SC_ACTIVE was set, remove it now:
800           * we're leaving the system call.
801           */
802          p->p_misc_flags &= ~MF_SC_ACTIVE;
803          break;
804        }
805        /*
806         * the selected process might not be
807                     * runnable anymore. We have
808         * to checkit and schedule another one
809         */
810        if (!proc_is_runnable(p))
811          goto not_runnable_pick_new;
812      }
813      /*
814       * check the quantum left before it runs again.
815             * We must do it only here
816       * as we are sure that a possible out-of-quantum
817             * message to the
818       * scheduler will not collide
819             * with the regular ipc
820       */
821      if (!p->p_cpu_time_left)
822        proc_no_time(p);
823      /*
824       * After handling the misc flags the selected
825             * process might not be
826       * runnable anymore. We have to checkit and
827             * schedule another one
828       */
829      if (!proc_is_runnable(p))
```

```
830     goto not_runnable_pick_new;
831
832   TRACE(VF_SCHEDULING,
833             printf("cpu %d starting %s / %d "
834         "pc 0x%08x\n",
835          cpuid, p->p_name,p->p_endpoint, p->p_reg.pc););
836 #if DEBUG_TRACE
837   p->p_schedules++;
838 #endif
839
840   p = arch_finish_switch_to_user();
841   assert(p->p_cpu_time_left);
842
843   context_stop(proc_addr(KERNEL));
844
845   /* If the process isn't the owner of FPU,
846           enable the FPU exception */
847   if (get_cpulocal_var(fpu_owner) != p)
848     enable_fpu_exception();
849   else
850     disable_fpu_exception();
851
852   /* If MF_CONTEXT_SET is set, don't
853           * clobber process state within
854    * the kernel. The next kernel entry
855           * is OK again though.
856    */
857   p->p_misc_flags &= ~MF_CONTEXT_SET;
858
859 #if defined(__i386__)
860     assert(p->p_seg.p_cr3 != 0);
861 #elif defined(__arm__)
862   assert(p->p_seg.p_ttbr != 0);
863 #endif
864 #ifdef CONFIG_SMP
865   if (p->p_misc_flags & MF_FLUSH_TLB) {
866     if (tlb_must_refresh)
867       refresh_tlb();
868     p->p_misc_flags &= ~MF_FLUSH_TLB;
869   }
870 #endif
871
872   restart_local_timer();
873
874         /** Add by EKA
875          ** The system is in an unstable stable.
876          ** We already check what goes
877          ** wrong in the micro-kernel and in the CPU.
878          ** It's now time to
```

```
879           ** run the PE again **/
880         if(h_unstable_state == H_UNSTABLE){
881           /** Just to know where we are**/
882           /** set the unstable state to in correction**/
883           h_unstable_state = H_INCORRECTION;
884           /**get the ptr on the PE process**/
885           p = proc_addr(h_proc_nr);
886           /**launch the PE **/
887           run_proc_2(p);
888         }
889
890         /**End add by EKA**/
891
892   /*
893    * restore_user_context() carries out the
894          * actual mode switch from kernel
895    * to userspace. This function does not return
896    */
897   restore_user_context(p);
898   NOT_REACHABLE;
899 }
900
901 /*===========================================================*
902  *    exception                                              *
903  *===========================================================*/
904 void exception_handler(int is_nested,
905 struct exception_frame * frame)
906 {
907 /* An exception or unexpected
908  * interrupt has occurred. */
909   register struct ex_s *ep;
910   struct proc *saved_proc;
911
912   /* Save proc_ptr, because it may be
913    * changed by debug statements. */
914   saved_proc = get_cpulocal_var(proc_ptr);
915
916   ep = &ex_data[frame->vector];
917
918   /* spurious NMI on some machines */
919   if (frame->vector == 2) {
920   return;
921   }
922
923   /*
924    * handle special cases for nested
925    * problems as they might be tricky or filter
926    * them out quickly if the traps are not nested
927    */
```

```
928   if (is_nested) {
929       /*
930        * if a problem occured while copying a
931        * message from userspace because
932        * of a wrong pointer supplied by userland,
933        * handle it the only way we
934        * can handle it ...
935        */
936    if(((void*)frame->eip >= (void*)copy_msg_to_user &&
937            (void*)frame->eip <=
938             (void*)__copy_msg_to_user_end)||
939     ((void*)frame->eip >=
940             (void*)copy_msg_from_user &&
941     (void*)frame->eip <=
942             (void*)__copy_msg_from_user_end)) {
943          switch(frame->vector) {
944          /* these error are expected */
945            case PAGE_FAULT_VECTOR:
946     case PROTECTION_VECTOR:
947       frame->eip =
948                    (reg_t) __user_copy_msg_pointer_failure;
949       return;
950     default:
951          panic("Copy involving a user"
952                       " pointer failed unexpectedly!");
953          }
954   }
955
956   /* Pass any error resulting from
957          * restoring FPU state, as a FPU
958    * exception to the process.
959    */
960       if (((void*)frame->eip >= (void*)fxrstor &&
961    (void *)frame->eip <= (void*)__fxrstor_end) ||
962      ((void*)frame->eip >= (void*)frstor &&
963      (void *)frame->eip <= (void*)__frstor_end)) {
964      frame->eip = (reg_t) __frstor_failure;
965      return;
966   }
967
968       if(frame->vector == DEBUG_VECTOR &&
969         (saved_proc->p_reg.psw & TRACEBIT) &&
970         (saved_proc->p_seg.p_kern_trap_style ==
971         KTS_NONE)) {
972   /* Getting a debug trap in the kernel
973          * is legitimate
974   * if a traced process entered the kernel
975          * using sysenter
976   * or syscall; the trap flag is not
```

```
977             * cleared then.
978     *
979     * It triggers on the first kernel entry
980             * so the trap
981     * style is still KTS_NONE.
982     */
983
984     frame->eflags &= ~TRACEBIT;
985
986     return;
987
988           /* If control passes, this case is not
989            *recognized as legitimate
990     * and we panic later on after all.
991     */
992     }
993     }
994
995     if(frame->vector == PAGE_FAULT_VECTOR) {
996     pagefault(saved_proc, frame, is_nested);
997     return;
998     }
999
1000    /* If an exception occurs while running a
1001     * process, the is_nested variable
1002     * will be zero. Exceptions in interrupt
1003     * handlers or system traps will make
1004     * is_nested non-zero.
1005     */
1006    if (is_nested == 0 && ! iskernelp(saved_proc)) {
1007
1008    cause_sig(proc_nr(saved_proc), ep->signum);
1009    return;
1010    }
1011
1012    /* Exception in system code. This
1013     * is not supposed to happen. */
1014    inkernel_disaster(saved_proc, frame, ep, is_nested);
1015
1016    panic("return from inkernel_disaster");
1017 }
```

A.3.14 Hardening software and the VM

Listing A.14: Hardening software and the VM

```
1 /*=========================================================*
2  *      map_subfree                        *
```

```
 *===========================================================================*/
static int map_subfree(struct vir_region *region,
  vir_bytes start, vir_bytes len)
{
    struct phys_region *pr;
    vir_bytes end = start+len;
    vir_bytes voffset;
#if SANITYCHECKS
    SLABSANE(region);
   for(voffset = 0; voffset < phys_slot(region->length);
   voffset += VM_PAGE_SIZE) {
      struct phys_region *others;
      struct phys_block *pb;
      if(!(pr = physblock_get(region, voffset)))
   continue;
      pb = pr->ph;
      for(others = pb->firstregion; others;
   others = others->next_ph_list) {
     assert(others->ph == pb);
      }
    }
#endif
    for(voffset = start; voffset < end;
          voffset+=VM_PAGE_SIZE) {
      if(!(pr = physblock_get(region, voffset)))
   continue;
      assert(pr->offset >= start);
      assert(pr->offset < end);
      pb_unreferenced(region, pr, 1);
      SLABFREE(pr);
    }
    /* Added by EKA to release the corresponding part
     * in us1_us2 list*/
    struct vmproc *vmp = region->parent;
    if(vmp &&
         (vmp->vm_hflags & VM_PROC_TO_HARD) &&
            (vmp->vm_endpoint != NONE) &&
            (vmp->vm_endpoint != VM_PROC_NR))
         if(free_region_pmbs(vmp, start, len)!=OK)
               panic("free_region_pmbs in vm");
    /* End added by EKA*/
    return OK;
}

/*===========================================================================*
 *     pt_writemap                       *
 *===========================================================================*/
/* path: scr/minix/servers/vm/pagetable.c*/
int pt_writemap(struct vmproc * vmp,
```

```
52        pt_t *pt,
53        vir_bytes v,
54        phys_bytes physaddr,
55        size_t bytes,
56        u32_t flags,
57        u32_t writemapflags){
58 /* Write mapping into page table. Allocate a new
59  * page table if necessary. */
60 /* Page directory and table entries
61  * for this virtual address. */
62   int p, pages;
63   int verify = 0;
64   int ret = OK;
65
66 #ifdef CONFIG_SMP
67   int vminhibit_clear = 0;
68   /* FIXME
69    * don't do it everytime, stop the process
70    * only on the first change and
71    * resume the execution on the last change.
72    * Do in a wrapper of this
73    * function
74    */
75    if (vmp && vmp->vm_endpoint != NONE &&
76        vmp->vm_endpoint != VM_PROC_NR &&
77        !(vmp->vm_flags & VMF_EXITING)) {
78     sys_vmctl(vmp->vm_endpoint,
79                  VMCTL_VMINHIBIT_SET, 0);
80      vminhibit_clear = 1;
81    }
82 #endif
83    if(writemapflags & WMF_VERIFY)
84        verify = 1;
85    assert(!(bytes % VM_PAGE_SIZE));
86    assert(!(flags & ~(PTF_ALLFLAGS)));
87    pages = bytes / VM_PAGE_SIZE;
88    /*Added by EKA: To be sent to kernel*/
89    vir_bytes vi = v;
90    phys_bytes physaddri = physaddr;
91    /* End added by EKA*/
92    /* MAP_NONE means to clear the mapping.
93     * It doesn't matter
94     * what's actually written into the
95     * PTE if PRESENT
96     * isn't on, so we can just write
97     * MAP_NONE into it.
98     */
99     assert(physaddr == MAP_NONE ||
100           (flags & ARCH_VM_PTE_PRESENT));
```

```
101     assert(physaddr != MAP_NONE || !flags);
102    /* First make sure all the necessary
103     * page tables are allocated,
104     * before we start writing in any of
105     * them, because it's a pain
106     * to undo our work properly.
107     */
108     ret = pt_ptalloc_in_range(pt, v,
109           v + VM_PAGE_SIZE*pages, flags, verify);
110     if(ret != OK) {
111        printf("VM: writemap:"
112        " pt_ptalloc_in_range failed\n");
113        goto resume_exit;
114     }
115     /* Now write in them. */
116   for(p = 0; p < pages; p++) {
117     u32_t entry;
118     int pde = ARCH_VM_PDE(v);
119     int pte = ARCH_VM_PTE(v);
120     assert(!(v % VM_PAGE_SIZE));
121     assert(pte >= 0 && pte < ARCH_VM_PT_ENTRIES);
122     assert(pde >= 0 && pde < ARCH_VM_DIR_ENTRIES);
123     /* Page table has to be there. */
124     assert(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT);
125    /* We do not expect it to be a bigpage. */
126     assert(!(pt->pt_dir[pde] & ARCH_VM_BIGPAGE));
127    /* Make sure page directory
128     * entry for this page table
129     * is marked present and page
130     * table entry is available.
131     */
132     assert(pt->pt_pt[pde]);
133
134     if(writemapflags & (WMF_WRITEFLAGSONLY|WMF_FREE)){
135#if defined(__i386__)
136   physaddr =
137                pt->pt_pt[pde][pte] & ARCH_VM_ADDR_MASK;
138#elif defined(__arm__)
139   physaddr =
140                pt->pt_pt[pde][pte] & ARM_VM_PTE_MASK;
141#endif
142       }
143
144       if(writemapflags & WMF_FREE) {
145   free_mem(ABS2CLICK(physaddr), 1);
146       }
147
148       /* Entry we will write. */
149#if defined(__i386__)
```

```
150         entry = (physaddr & ARCH_VM_ADDR_MASK) | flags;
151 #elif defined(__arm__)
152         entry = (physaddr & ARM_VM_PTE_MASK) | flags;
153 #endif
154         if(verify) {
155     u32_t maskedentry;
156     maskedentry = pt->pt_pt[pde][pte];
157 #if defined(__i386__)
158     maskedentry &= ~(I386_VM_ACC|I386_VM_DIRTY);
159 #endif
160   /* Verify pagetable entry. */
161 #if defined(__i386__)
162   if(entry & ARCH_VM_PTE_RW) {
163   /* If we expect a writable page,
164           * allow a readonly page. */
165     maskedentry |= ARCH_VM_PTE_RW;
166   }
167 #elif defined(__arm__)
168   if(!(entry & ARCH_VM_PTE_RO)) {
169    /* If we expect a writable page,
170           * allow a readonly page. */
171     maskedentry &= ~ARCH_VM_PTE_RO;
172   }
173   maskedentry &= ~(ARM_VM_PTE_WB|ARM_VM_PTE_WT);
174 #endif
175   if(maskedentry != entry) {
176      printf("pt_writemap: mismatch: ");
177 #if defined(__i386__)
178   if((entry & ARCH_VM_ADDR_MASK) !=
179     (maskedentry & ARCH_VM_ADDR_MASK)) {
180 #elif defined(__arm__)
181   if((entry & ARM_VM_PTE_MASK) !=
182           (maskedentry & ARM_VM_PTE_MASK)) {
183 #endif
184      printf("pt_writemap: physaddr"
185                     " mismatch (0x%lx, 0x%lx); ",
186     (long)entry, (long)maskedentry);
187   } else printf("phys ok; ");
188           printf(" flags: found %s; ",
189     ptestr(pt->pt_pt[pde][pte]));
190     printf(" masked %s; ",
191     ptestr(maskedentry));
192     printf(" expected %s\n", ptestr(entry));
193     printf("found 0x%x, wanted 0x%x\n",
194           pt->pt_pt[pde][pte], entry);
195     ret = EFAULT;
196     goto resume_exit;
197   }
198         } else {
```

```
199     /* Write pagetable entry. */
200       pt->pt_pt[pde][pte] = entry;
201         }
202         physaddr += VM_PAGE_SIZE;
203         v += VM_PAGE_SIZE;
204     }
205 resume_exit:
206 #ifdef CONFIG_SMP
207     if (vminhibit_clear) {
208       assert(vmp && vmp->vm_endpoint != NONE &&
209              vmp->vm_endpoint != VM_PROC_NR &&
210        !(vmp->vm_flags & VMF_EXITING));
211       sys_vmctl(vmp->vm_endpoint,
212                 VMCTL_VMINHIBIT_CLEAR, 0);
213     }
214 #endif
215 /* Added by EKA*/
216     if((flags & ARCH_VM_PTE_RW) && vmp &&
217          (vmp->vm_hflags & VM_PROC_TO_HARD) &&
218          (vmp->vm_endpoint != NONE) &&
219          (vmp->vm_endpoint != VM_PROC_NR))
220       if(tell_kernel_for_us1_us2(vmp, vi,
221          physaddri, bytes)!=OK)
222          panic("pt_writemap: tell_kernel_for"
223                "_us1_us2 failed\n");
224 /*End added by EKA*/
225     return ret;
226 }
227
228 /*===========================================================*
229  *          do_fork                                          *
230  *===========================================================*/
231 int do_fork(message *msg)
232 {
233   int r, proc, childproc;
234   struct vmproc *vmp, *vmc;
235   pt_t origpt;
236   vir_bytes msgaddr;
237
238   SANITYCHECK(SCL_FUNCTIONS);
239
240   if(vm_isokendpt(msg->VMF_ENDPOINT, &proc) != OK){
241   printf("VM: bogus endpoint VM_FORK %d\n",
242          msg->VMF_ENDPOINT);
243   SANITYCHECK(SCL_FUNCTIONS);
244   return EINVAL;
245   }
246
247   childproc = msg->VMF_SLOTNO;
```

```
248     if(childproc < 0 || childproc >= NR_PROCS) {
249     printf("VM: bogus slotno VM_FORK %d\n",
250             msg->VMF_SLOTNO);
251    SANITYCHECK(SCL_FUNCTIONS);
252     return EINVAL;
253     }
254
255     vmp = &vmproc[proc];     /* parent */
256     vmc = &vmproc[childproc]; /* child */
257     assert(vmc->vm_slot == childproc);
258
259     /* The child is basically a copy of the parent. */
260     origpt = vmc->vm_pt;
261     *vmc = *vmp;
262     vmc->vm_slot = childproc;
263     region_init(&vmc->vm_regions_avl);
264     /* In case someone tries to use it. */
265     vmc->vm_endpoint = NONE;
266     vmc->vm_pt = origpt;
267
268 #if VMSTATS
269     vmc->vm_bytecopies = 0;
270 #endif
271
272     if(pt_new(&vmc->vm_pt) != OK) {
273     return ENOMEM;
274     }
275
276    SANITYCHECK(SCL_DETAIL);
277
278     if(map_proc_copy(vmc, vmp) != OK) {
279     printf("VM: fork: map_proc_copy failed\n");
280     pt_free(&vmc->vm_pt);
281     return(ENOMEM);
282     }
283
284     /* Only inherit these flags. */
285     vmc->vm_flags &= VMF_INUSE;
286
287     /* Deal with ACLs. */
288     acl_fork(vmc);
289
290     /* Tell kernel about the (now successful) FORK. */
291     if((r=sys_fork(vmp->vm_endpoint, childproc,
292    &vmc->vm_endpoint, PFF_VMINHIBIT,
293             &msgaddr)) != OK) {
294             panic("do_fork can't sys_fork: %d", r);
295     }
296
```

```
297     if((r=pt_bind(&vmc->vm_pt, vmc)) != OK)
298     panic("fork can't pt_bind: %d", r);
299
300     /*** Added by EKA ***/
301     if(hardening_enabled)
302        vmc->vm_hflags |= VM_PROC_TO_HARD;
303     if(vmp->vm_hflags & VM_PROC_TO_HARD){
304       free_pram_mem_blocks(vmp);
305     }
306     /*** End Added by EKA ***/
307     /* Inform caller of new child endpoint. */
308
309     {
310     vir_bytes vir;
311     /* making these messages writable
312           * is an optimisation
313      * and its return value needn't be checked.
314      */
315     vir = msgaddr;
316     if (handle_memory_once(vmc, vir,
317                            sizeof(message), 1) != OK)
318         panic("do_fork: handle_memory"
319                               " for child failed\n");
320     vir = msgaddr;
321     if (handle_memory_once(vmp, vir,
322                          sizeof(message), 1) != OK)
323         panic("do_fork: handle_memory"
324                           " for parent failed\n");
325     }
326
327     msg->VMF_CHILD_ENDPOINT = vmc->vm_endpoint;
328
329    SANITYCHECK(SCL_FUNCTIONS);
330     return OK;
331 }
332
333 void init_vm(void)
334 {
335     int s, i;
336     static struct memory mem_chunks[NR_MEMS];
337     struct boot_image *ip;
338     extern void __minix_init(void);
339     multiboot_module_t *mod;
340     vir_bytes kern_dyn, kern_static;
341           struct pram_mem_block *pmb;
342
343 #if SANITYCHECKS
344     incheck = nocheck = 0;
345 #endif
```

```
346
347   /* Retrieve various crucial boot parameters */
348   if(OK != (s=sys_getkinfo(&kernel_boot_info))) {
349     panic("couldn't get bootinfo: %d", s);
350   }
351
352   /* Turn file mmap on? */
353   enable_filemap=1; /* yes by default */
354   env_parse("filemap", "d", 0,
355                   &enable_filemap, 0, 1);
356
357   /* Sanity check */
358   assert(kernel_boot_info.mmap_size > 0);
359   assert(kernel_boot_info.mods_with_kernel > 0);
360
361   /* Get chunks of available memory. */
362   get_mem_chunks(mem_chunks);
363
364   /* Set table to 0. This invalidates
365          * all slots (clear VMF_INUSE). */
366   memset(vmproc, 0, sizeof(vmproc));
367
368         /*Added by EKA*/
369         int hardening_enabled = 0;
370          /*End added by EKA*/
371
372   for(i = 0; i < ELEMENTS(vmproc); i++) {
373     vmproc[i].vm_slot = i;
374                   /*Added by EKA*/
375                   vmproc[i].vm_hflags = 0;
376                   vmproc[i].vm_lus1_us2_size = 0;
377                   vmproc[i].vm_lus1_us2 = NULL;
378                   /*End added by EKA*/
379   }
380         /*Added by EKA*/
381         /**Initialize the hardening copy-on-write
382         * data structure table**/
383         for(pmb = BEG_PRAM_MEM_BLOCK_ADDR;
384              pmb < END_PRAM_MEM_BLOCK_ADDR; pmb++){
385               pmb->flags = PRAM_SLOT_FREE;
386               pmb->vaddr = 0;
387               pmb->us0 = 0;
388               pmb->us1 = 0;
389               pmb->us2 = 0;
390
391         }
392          /*End added by EKA*/
393   /* Initialize ACL data structures. */
394   acl_init();
```

```
395
396   /* region management initialization. */
397   map_region_init();
398
399   /* Initialize tables to all physical memory. */
400   mem_init(mem_chunks);
401
402   /* Architecture-dependent initialization. */
403   init_proc(VM_PROC_NR);
404   pt_init();
405
406   /* Acquire kernel ipc vectors
407          * that weren't available
408    * before VM had determined kernel mappings
409    */
410   __minix_init();
411
412   /* The kernel's freelist does not
413          * include boot-time modules; let
414    * the allocator know that the total memory is bigger.
415    */
416   for (mod = &kernel_boot_info.module_list[0];
417    mod < &kernel_boot_info.module_list
418          [kernel_boot_info.mods_with_kernel-1]; mod++) {
419          phys_bytes len = mod->mod_end-mod->mod_start+1;
420       len = roundup(len, VM_PAGE_SIZE);
421             mem_add_total_pages(len/VM_PAGE_SIZE);
422   }
423        kern_dyn =
424         kernel_boot_info.kernel_allocated_bytes_dynamic;
425        kern_static =
426         kernel_boot_info.kernel_allocated_bytes;
427        kern_static =
428         roundup(kern_static, VM_PAGE_SIZE);
429        mem_add_total_pages(
430         (kern_dyn + kern_static)/VM_PAGE_SIZE);
431        /* Give these processes their
432         * own page table. */
433        for (ip = &kernel_boot_info.boot_procs[0];
434    ip <
435           &kernel_boot_info.boot_procs[NR_BOOT_PROCS];
436           ip++) {
437       struct vmproc *vmp;
438       if(ip->proc_nr < 0) continue;
439       assert(ip->start_addr);
440       /* VM has already been set up by the
441             * kernel and pt_init().
442        * Any other boot process is already
443             * in memory and is set up
```

```
444             * here.
445             */
446            if(ip->proc_nr == VM_PROC_NR) continue;
447            vmp = init_proc(ip->proc_nr);
448            exec_bootproc(vmp, ip);
449            /* Free the file blob */
450            assert(!(ip->start_addr % VM_PAGE_SIZE));
451            ip->len = roundup(ip->len, VM_PAGE_SIZE);
452            free_mem(ABS2CLICK(ip->start_addr),
453                     ABS2CLICK(ip->len));
454    }
455
456    /* Set up table of calls. */
457 #define CALLMAP(code, func) { int _cmi;     \
458    _cmi=CALLNUMBER(code);          \
459    assert(_cmi >= 0);          \
460    assert(_cmi < NR_VM_CALLS);     \
461    vm_calls[_cmi].vmc_func = (func);       \
462    vm_calls[_cmi].vmc_name = #code;    \
463 }
464
465    /* Set call table to 0. This invalidates
466            * all calls (clear
467     * vmc_func).
468     */
469    memset(vm_calls, 0, sizeof(vm_calls));
470
471    /* Basic VM calls. */
472    CALLMAP(VM_MMAP, do_mmap);
473    CALLMAP(VM_MUNMAP, do_munmap);
474    CALLMAP(VM_MAP_PHYS, do_map_phys);
475    CALLMAP(VM_UNMAP_PHYS, do_munmap);
476
477    /* Calls from PM. */
478    CALLMAP(VM_EXIT, do_exit);
479    CALLMAP(VM_FORK, do_fork);
480    CALLMAP(VM_BRK, do_brk);
481    CALLMAP(VM_WILLEXIT, do_willexit);
482
483    CALLMAP(VM_PROCCTL, do_procctl_notrans);
484
485    /* Calls from VFS. */
486    CALLMAP(VM_VFS_REPLY, do_vfs_reply);
487    CALLMAP(VM_VFS_MMAP, do_vfs_mmap);
488
489    /* Calls from RS */
490    CALLMAP(VM_RS_SET_PRIV, do_rs_set_priv);
491    CALLMAP(VM_RS_PREPARE, do_rs_prepare);
492    CALLMAP(VM_RS_UPDATE, do_rs_update);
```

```
493   CALLMAP(VM_RS_MEMCTL, do_rs_memctl);
494
495   /* Generic calls. */
496   CALLMAP(VM_REMAP, do_remap);
497   CALLMAP(VM_REMAP_RO, do_remap);
498   CALLMAP(VM_GETPHYS, do_get_phys);
499   CALLMAP(VM_SHM_UNMAP, do_munmap);
500   CALLMAP(VM_GETREF, do_get_refcount);
501   CALLMAP(VM_INFO, do_info);
502
503   /* Cache blocks. */
504   CALLMAP(VM_MAPCACHEPAGE, do_mapcache);
505   CALLMAP(VM_SETCACHEPAGE, do_setcache);
506   CALLMAP(VM_FORGETCACHEPAGE, do_forgetcache);
507   CALLMAP(VM_CLEARCACHE, do_clearcache);
508
509   /* getrusage */
510   CALLMAP(VM_GETRUSAGE, do_getrusage);
511
512   /* Mark VM instances. */
513   num_vm_instances = 1;
514   vmproc[VM_PROC_NR].vm_flags |= VMF_VM_INSTANCE;
515
516   /* Let SEF know about VM mmapped regions. */
517   s = sef_llvm_add_special_mem_region(
518             (void*)VM_OWN_HEAPBASE,
519       VM_OWN_MMAPTOP-VM_OWN_HEAPBASE, "%MMAP_ALL");
520   if(s < 0) {
521       printf("VM: st_add_special_mmapped_region "
522              "failed %d\n", s);
523   }
524 }
525
526
527 /*===========================================*
528  *       main                *
529  *===========================================*/
530 int main(void)
531 {
532   message msg;
533   int result, who_e, rcv_sts;
534   int caller_slot;
535
536   /* Initialize system so that all
537    * processes are runnable the first time. */
538   if (is_first_time()) {
539   init_vm();
540   __vm_init_fresh=1;
541   }
```

```
542
543   /* SEF local startup. */
544   sef_local_startup();
545   __vm_init_fresh=0;
546
547   SANITYCHECK(SCL_TOP);
548
549   /* This is VM's main loop. */
550   while (TRUE) {
551   int r, c;
552   int type;
553   int transid = 0;  /* VFS transid if any */
554
555   SANITYCHECK(SCL_TOP);
556   if(missing_spares > 0) {
557             /* mem alloc code wants to be called */
558      alloc_cycle();
559   }
560
561     if ((r=sef_receive_status(ANY, &msg,
562                &rcv_sts)) != OK)
563     panic("sef_receive_status()"
564                   " error: %d", r);
565
566   if (is_ipc_notify(rcv_sts)) {
567     /* Unexpected ipc_notify(). */
568     printf("VM: ignoring ipc_notify() "
569                   "from %d\n", msg.m_source);
570     continue;
571   }
572   who_e = msg.m_source;
573   if(vm_isokendpt(who_e, &caller_slot) != OK)
574     panic("invalid caller %d", who_e);
575
576   /* We depend on this being false for
577          * the initialized value. */
578   assert(!IS_VFS_FS_TRANSID(transid));
579
580   type = msg.m_type;
581   c = CALLNUMBER(type);
582         /* Out of range or restricted
583          * calls return this. */
584   result = ENOSYS;
585
586   transid = TRNS_GET_ID(msg.m_type);
587
588          /** Added by EKA**/
589          /**That is a hardening page fault.
590           * The kernel inform the VM
```

```
591             ** VM called the handler do_hpagefault **/
592           if ((msg.m_type == VM_HR1PAGEFAULT) ||
593               (msg.m_type == VM_HR2PAGEFAULT)){
594                 do_hpagefaults(&msg);
595                 continue;
596           }
597
598           if((msg.m_type == VM_TELL_VM_H_ENABLE) ||
599                 (msg.m_type == VM_TELL_VM_H_DISABLE) ||
600                 (msg.m_type == VM_TELL_VM_H_ENABLE_P) ||
601                  (msg.m_type == VM_TELL_VM_H_DISABLE_P)){
602                 do_hardening(&msg);
603                 continue;
604           }
605
606           /** End added by EKA**/
607
608     if((msg.m_source == VFS_PROC_NR) &&
609                IS_VFS_FS_TRANSID(transid)) {
610           /* If it's a request from VFS,
611                  * it might have a transaction id. */
612       msg.m_type = TRNS_DEL_ID(msg.m_type);
613
614       /* Calls that use the transid */
615       result = do_procctl(&msg, transid);
616     } else if(msg.m_type == RS_INIT &&
617                  msg.m_source == RS_PROC_NR) {
618           result = do_sef_init_request(&msg);
619           if(result != OK)
620                  panic("do_sef_init_request failed!\n");
621           result = SUSPEND; /* do not reply to RS */
622     } else if (msg.m_type == VM_PAGEFAULT) {
623        if (!IPC_STATUS_FLAGS_TEST(rcv_sts,
624                  IPC_FLG_MSG_FROM_KERNEL)) {
625       printf("VM: process %d faked "
626                     "VM_PAGEFAULT message!\n",
627                      msg.m_source);
628           }
629           do_pagefaults(&msg);
630         /*
631          * do not reply to this call, the caller
632                * is unblocked by
633          * a sys_vmctl() call in do_pagefaults
634                * if success. VM panics
635          * otherwise
636          */
637       continue;
638     } else if(c < 0 || !vm_calls[c].vmc_func) {
639       /* out of range or missing callnr */
```

```
640     } else {
641       if (acl_check(&vmproc[caller_slot], c)
642                   != OK) {
643       printf("VM: unauthorized %s by %d\n",
644         vm_calls[c].vmc_name, who_e);
645       } else {
646         SANITYCHECK(SCL_FUNCTIONS);
647         result = vm_calls[c].vmc_func(&msg);
648         SANITYCHECK(SCL_FUNCTIONS);
649       }
650     }
651
652     /* Send reply message, unless the
653           * return code is SUSPEND,
654      * which is a pseudo-result suppressing
655           * the reply message.
656      */
657     if(result != SUSPEND) {
658       msg.m_type = result;
659
660       assert(!IS_VFS_FS_TRANSID(transid));
661
662       if((r=ipc_send(who_e, &msg)) != OK) {
663         printf("VM: couldn't send %d"
664                             " to %d (err %d)\n",
665           msg.m_type, who_e, r);
666         panic("ipc_send() error");
667       }
668     }
669     }
670     return(OK);
671 }
672
673
674 void clear_proc(struct vmproc *vmp)
675 {
676           /*** Added by EKA ***/
677           if(vmp->vm_hflags & VM_PROC_TO_HARD)
678             free_pram_mem_blocks(vmp);
679           /*** End Added by EKA ***/
680     region_init(&vmp->vm_regions_avl);
681     acl_clear(vmp);
682           /* Clear INUSE, so slot is free. */
683     vmp->vm_flags = 0;
684 #if VMSTATS
685     vmp->vm_bytecopies = 0;
686 #endif
687     vmp->vm_region_top = 0;
688     reset_vm_rusage(vmp);
```

```
689
690 }
```

A.3.15 Hardening software utility

Listing A.15: Hardening software utility

```
1  /*===========================================*
2   *    lin_lin_cmp                            *
3   *===========================================*/
4  static int lin_lin_cmp(struct proc *srcproc,
5                  vir_bytes srclinaddr,
6      struct proc *dstproc,
7                  vir_bytes dstlinaddr,
8                  vir_bytes bytes)
9  {
10   u32_t addr;
11   proc_nr_t procslot;
12   int i, r  = OK;
13   assert(get_cpulocal_var(ptproc));
14   assert(get_cpulocal_var(proc_ptr));
15   assert(read_cr3() ==
16           get_cpulocal_var(ptproc)->p_seg.p_cr3);
17   procslot = get_cpulocal_var(ptproc)->p_nr;
18   assert(procslot >= 0 &&
19           procslot < I386_VM_DIR_ENTRIES);
20
21   if(srcproc)
22           assert(!RTS_ISSET(srcproc, RTS_SLOT_FREE));
23   if(dstproc)
24           assert(!RTS_ISSET(dstproc, RTS_SLOT_FREE));
25   assert(!RTS_ISSET(get_cpulocal_var(ptproc),
26           RTS_SLOT_FREE));
27   assert(get_cpulocal_var(ptproc)->p_seg.p_cr3_v);
28   if(srcproc)
29            assert(!RTS_ISSET(srcproc, RTS_VMINHIBIT));
30   if(dstproc)
31            assert(!RTS_ISSET(dstproc, RTS_VMINHIBIT));
32
33   while(bytes > 0) {
34     phys_bytes srcptr, dstptr;
35     vir_bytes chunk = bytes;
36     int changed = 0;
37     /* Set up 4MB ranges. */
38     srcptr = createpde(srcproc, srclinaddr,
39                   &chunk, 0, &changed);
40     dstptr = createpde(dstproc, dstlinaddr,
41                   &chunk, 1, &changed);
```

```
42      if(changed)
43      reload_cr3();
44      /* Compare pages. */
45      if(phys_cmp(srcptr, dstptr, chunk)) {
46          r = EFAULT;
47      }
48      /* Update counter and addresses for
49              * next iteration, if any. */
50      bytes -= chunk;
51      srclinaddr += chunk;
52      dstlinaddr += chunk;
53
54    }
55
56          if(srcproc)
57              assert(!RTS_ISSET(srcproc, RTS_SLOT_FREE));
58          if(dstproc)
59              assert(!RTS_ISSET(dstproc, RTS_SLOT_FREE));
60          assert(!RTS_ISSET(get_cpulocal_var(ptproc),
61                  RTS_SLOT_FREE));
62          assert(get_cpulocal_var(ptproc)->p_seg.p_cr3_v);
63
64          return r;
65  }
66
67  /*===========================================================*/
68  /*     phys_cmp                                  */
69  /*===========================================================*/
70  /*
71   * PUBLIC int phys_cmp(phys_bytes source,
72   * phys_bytes destination,
73   *       phys_bytes bytecount);
74   * Cmp a block of data from anywhere to anywhere
75   * in physical memory.
76   */
77  /*     es edi esi eip    src dst len */
78  ENTRY(phys_cmp)
79    push  %ebp
80    mov %esp, %ebp
81
82    cld
83    push  %esi
84    push  %edi
85
86    mov 8(%ebp), %esi
87    mov 12(%ebp), %edi
88    mov 16(%ebp), %ecx
89
90    cld
```

```
91
92    repe cmpsb
93    jne not_equal
94    mov $0, %eax
95    jmp exit
96
97  not_equal:
98    mov $1, %eax /* 0 means: no fault */
99
100 exit:
101   pop %edi
102   pop %esi
103   pop %ebp
104   ret
105
106 int cpy_frames(u32_t src, u32_t dst){
107    if(lin_lin_copy(NULL, src, NULL, dst,
108          I386_PAGE_SIZE)!=OK){
109     return EFAULT;
110   }
111     return(OK);
112 }
113
114 /* This function is used to modified the data at addr
115  * We used it to modified the value
116  * of the   page table entries
117  * add at 30/12/2014
118  */
119 int phys_set32(phys_bytes addr, u32_t *v)
120 {
121    int r;
122    if((r=lin_lin_copy(proc_addr(SYSTEM), (vir_bytes) v,
123   NULL, addr , sizeof(u32_t))) != OK) {
124   return(r);
125    }
126    return(OK);
127 }
```

A.3.16 USER Space library for hardening

Listing A.16: USER Space library for hardening

```
1 /*START USER SPACE LIB*/
2 /*USER SPACE COMMAND*/
3 /* path: src/minix/tests/
4 *  hardening:  src/minix/tests/hardening.c
5 *  This program can be call to enable or disable
6 *  hardening
```

```
7  *  A system cal is made from the user process to the PM
8  *  servers.
9  *  The PM server makes a kernel call to the micro-kernel
10 *  The micro-kernel enable the hardening at micro-kernel
11 *  level
12 *  The micro-kernel sends message to the VM
13 *  The VM enable the hardening at VM level
14 *  USAGE:
15 *       - hardening <1> to enable hardening for all
16 *          new process (fork)
17 *       - hardening <2> to disable hardening for all
18 *          new process (fork)
19 *       - hardening <4> <pid> to enable hardening for
20 *         process reprented by pid
21 *       - hardening <8> <pid> to disable hardening for
22 *          process reprented by pid
23 */
24 /* created by Emery Assogba */
25 /* assogba.emery@gmail.com
26  * 06-Fevr-2019  12:45:04 */
27
28 /* Copyright (C) 2019 by Emery Assogba.
29  * All rights reserved. */
30 /* Used by permission. */
31 #include <unistd.h>
32 #include <stdlib.h>
33 #include "hardening.h"
34 #include <string.h>
35 static void usage(char *prog_name);
36 int main(int argc, char *argv[]){
37    int r = OK;
38    if(argc<2){
39      usage(argv[0]);
40      return(r);
41    }
42    int type = atoi(argv[1]);
43    int pid;
44    switch(type){
45        case HTASK_EN_HARDENING_ALL_F:
46           if((r = hardening(HTASK_EN_HARDENING_ALL_F,
47                   PID_NONE, NULL, 0))!=OK)
48              printf("Enable hardening for all new"
49                        " procs failed %d\n", r);
50           break;
51      case HTASK_DIS_HARDENING_ALL_F:
52           if((r = hardening(HTASK_DIS_HARDENING_ALL_F,
53                     PID_NONE, NULL, 0))!=OK)
54              printf("Disable hardening for all new "
55                           "procs failed %d\n", r);
```

```
56              break;
57          case HTASK_EN_HARDENING_PID:
58              if(argc!=3){
59                usage(argv[0]);
60              return(r);
61              }
62              pid = atoi(argv[2]);
63              if((r=hardening(HTASK_EN_HARDENING_PID,
64                          pid, NULL, 0))!=OK)
65                  printf("Enable hardening for %d"
66                              " procs failed %d\n", pid, r);
67              break;
68          case HTASK_DIS_HARDENING_PID:
69              pid = atoi(argv[2]);
70              if(argc!=3){
71                usage(argv[0]);
72              return(r);
73              }
74              if((r=hardening(HTASK_DIS_HARDENING_PID,
75                          pid, NULL, 0))!=OK)
76                  printf("Disable hardening for %d"
77                          " procs failed %d\n", pid, r);
78              break;
79          default:
80              usage(argv[0]);
81              break;
82      }
83      return(r);
84  }
85  static void usage(char *prog_name){
86      printf("USAGE %s <reqtype> [pid] [name]\n",
87                          prog_name);
88      printf("%s <1> : to enable hardening for all"
89                          " new process \n", prog_name);
90      printf("%s <2> : to disable hardening for all"
91                         " new process \n", prog_name);
92      printf("%s <4> <pid> to enable hardening"
93          " for process reprented by pid \n", prog_name);
94      printf("%s <8> <pid> to disable hardening "
95          " for process reprented by pid \n", prog_name);
96  }
97  /*hardening lib: path:
98   *    src/minix/mib/libc/sys/hardening.c*/
99  #include <lib.h>
100 #include <unistd.h>
101 int hardening(int type, pid_t pid,
102   char *name, int namelen){
103   message m;
104   switch(type){
```

```
105         case HTASK_EN_HARDENING_ALL_F:
106             m.HTASK_TYPE = HTASK_EN_HARDENING_ALL_F;
107             m.HTASK_P_ENDPT       = pid;
108             break;
109
110         case HTASK_DIS_HARDENING_ALL_F:
111             m.HTASK_TYPE = HTASK_DIS_HARDENING_ALL_F;
112             m.HTASK_P_ENDPT       = pid;
113             break;
114         case HTASK_EN_HARDENING_PID:
115             m.HTASK_TYPE = HTASK_EN_HARDENING_PID;
116             m.HTASK_P_ENDPT       = pid;
117             break;
118         case HTASK_DIS_HARDENING_PID:
119             m.HTASK_TYPE = HTASK_DIS_HARDENING_PID;
120             m.HTASK_P_ENDPT       = pid;
121             break;
122         default:
123             return(-1);
124     }
125     return(_syscall(PM_PROC_NR,PM_HARDENING,&m));
126 }
127 /*PM PART OF USER SPACE LIB*/
128 /*===========================================================*
129  *                do_hardening                               *
130  *===========================================================*/
131 /* do_hardening:
132  * path src/minix/servers/pm/misc.c*/
133 int do_hardening(void){
134
135     char *pname;
136     switch(m_in.HTASK_TYPE){
137       case HTASK_EN_HARDENING_ALL_F:
138             sys_hardening(PM_PROC_NR,
139                 HTASK_EN_HARDENING_ALL_F , 0, NULL,0);
140             break;
141
142       case HTASK_DIS_HARDENING_ALL_F:
143             sys_hardening(PM_PROC_NR,
144                 HTASK_DIS_HARDENING_ALL_F , 0, NULL,0);
145             break;
146       case HTASK_EN_HARDENING_PID:
147             sys_hardening(PM_PROC_NR,
148                   HTASK_EN_HARDENING_PID,
149                   m_in.HTASK_P_ENDPT, NULL,0);
150             break;
151       case HTASK_DIS_HARDENING_PID:
152             sys_hardening(PM_PROC_NR,
153                           HTASK_DIS_HARDENING_PID,
```

```
154                             m_in.HTASK_P_ENDPT, NULL,0);
155             break;
156     }
157     return(OK);
158 }
159 /*sys_hardening.c path:
160  * src/minix/lib/libsys/sys_hardening.c*/
161 #include "syslib.h"
162 #include <string.h>
163 int sys_hardening(proc_endpt, htask_type,
164     htask_pendt, htask_pname, namelen)
165 endpoint_t proc_endpt;  /* process endpoint */
166 int  htask_type;  /* type of task */
167 endpoint_t htask_pendt; /* process to hard */
168 char *htask_pname;  /* process to hard */
169 int namelen;
170 {
171     message m;
172     m.HTASK_ENDPT          = proc_endpt;
173     m.HTASK_P_ENDPT        = htask_pendt;
174     if(htask_pname!=NULL)
175         m.HTASK_PNAME = (vir_bytes)htask_pname;
176     else
177         m.HTASK_PNAME = 0L;
178     m.HTASK_TYPE           =  htask_type;
179     return(_kernel_call(SYS_HARDENING, &m));
180 }
181 /*KERNEL PART OF USER SPACE LIB*/
182 /* The kernel call implement in this file
183  *   m_type : SYS_HARDENING
184  * The parameters for this kernel call are:
185  *     - HTASK_ENDPT
186  *     - HTASK_TYPE
187  *         - HTASK_EN_HARDENING_ALL_F :
188  *             Enable hardening for new forked process
189  *         - HTASK_DIS_HARDENING_ALL_F :
190  *             Disable hardening for new forked process
191  *         - HTASK_EN_HARDENING_PID:
192  *             Enable hardening for given pid process
193  *         - HTASK_DIS_HARDENING_PID:
194  *             Disable hardening for given pid process
195  *     - HTASK_PENDPOINT
196  *     - HTASK_PNAME
197  */
198
199 #include "kernel/system.h"
200 #include "kernel/arch/i386/include/arch_proto.h"
201 #include "kernel/arch/i386/hproto.h"
202 #include "kernel/arch/i386/htype.h"
```

```
203 #include <assert.h>
204 /*=========================================================*
205  *                do_hardening                             *
206  *=========================================================*/
207 int do_hardening(struct proc * caller,
208   message * m_ptr){
209   proc_nr_t proc_nr, proc_nr_e, hproc_nr, hproc_nr_e;
210   struct proc *p, *hp;
211   message hm;
212   int err;
213   proc_nr_e= (proc_nr_t) m_ptr->HTASK_ENDPT;
214   if (!isokendpt(proc_nr_e, &proc_nr))
215         return(EINVAL);
216   p = proc_addr(proc_nr);
217   switch(m_ptr->HTASK_TYPE){
218      case HTASK_EN_HARDENING_ALL_F:
219           h_can_start_hardening = ENABLE_HARDENING;
220           hm.m_source = p->p_endpoint;
221           hm.m_type   = VM_TELL_VM_H_ENABLE;
222           if ((err = mini_send(p, VM_PROC_NR,
223       &hm, FROM_KERNEL))) {
224     panic("WARNING: enable_hardening:"
225                   " mini_send returned %d\n", err);
226           }
227           break;
228      case HTASK_DIS_HARDENING_ALL_F:
229           h_can_start_hardening = DISABLE_HARDENING;
230           hm.m_source = p->p_endpoint;
231           hm.m_type   = VM_TELL_VM_H_DISABLE;
232           if ((err = mini_send(p, VM_PROC_NR,
233       &hm, FROM_KERNEL))) {
234     panic("WARNING: disable_hardening:"
235                   " mini_send returned %d\n", err);
236           }
237           break;
238      case HTASK_EN_HARDENING_PID:
239           hproc_nr_e= (proc_nr_t) m_ptr->HTASK_P_ENDPT;
240           if (!isokendpt(hproc_nr_e, &hproc_nr))
241                 return(EINVAL);
242           hp = proc_addr(hproc_nr);
243           hp->p_hflags |= PROC_TO_HARD;
244           hm.m_source = hp->p_endpoint;
245           hm.m_type   = VM_TELL_VM_H_ENABLE_P;
246           if ((err = mini_send(hp, VM_PROC_NR,
247       &hm, FROM_KERNEL))) {
248     panic("WARNING: enable_hardening:"
249                   " mini_send returned %d\n", err);
250           }
251           break;
```

```
252     case HTASK_DIS_HARDENING_PID:
253           hproc_nr_e= (proc_nr_t) m_ptr->HTASK_P_ENDPT;
254           if (!isokendpt(hproc_nr_e, &hproc_nr))
255                 return(EINVAL);
256           hp = proc_addr(hproc_nr);
257           hp->p_hflags &= ~PROC_TO_HARD;
258           hm.m_source = hp->p_endpoint;
259           hm.m_type   = VM_TELL_VM_H_DISABLE_P;
260           if ((err = mini_send(hp, VM_PROC_NR,
261       &hm, FROM_KERNEL))) {
262     panic("WARNING: disable_hardening:"
263                   " mini_send returned %d\n", err);
264           }
265           break;
266   }
267   return(OK);
268 }
269 /*VM PART OF USER SPACE LIB*/
270 /*===========================================================*
271  *                do_hardening                               *
272  *===========================================================*/
273 void do_hardening(message *m)
274 {
275   endpoint_t ep = m->m_source;
276   int reqtype = m->m_type;
277   struct vmproc *vmp;
278   int p;
279   if(vm_isokendpt(ep, &p) != OK)
280       panic("do_hardening: endpoint wrong: %d", ep);
281   vmp = &vmproc[p];
282   assert((vmp->vm_flags & VMF_INUSE));
283   switch(reqtype){
284        case VM_TELL_VM_H_ENABLE:
285           hardening_enabled = ENABLE_HARDENING;
286           break;
287
288      case VM_TELL_VM_H_DISABLE:
289           hardening_enabled = DISABLE_HARDENING;
290           break;
291      case VM_TELL_VM_H_ENABLE_P:
292           vmp->vm_hflags |= VM_PROC_TO_HARD;
293           break;
294      case VM_TELL_VM_H_DISABLE_P:
295           vmp->vm_hflags &= ~VM_PROC_TO_HARD;
296           break;
297   }
298 }
299 /*END USER SPACE LIB*/
```

Bibliography

[Adv16] Advanced Micro Devices, Inc. BIOS and kernel developers guide (BKDG) for AMD family 16h models 30h-3fh processors. (52740), 2016.

[AG93] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, pages 1015–1027, 1993.

[AGM+17] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.

[AGV96] Anish Arora, Mohamed Gouda, and George Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[AK98] A. Arora and S.S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *Software Engineering, IEEE Transactions on*, 24(6):435–450, 1998.

[AL16] Emery Kouassi Assogba and Marc Lobelle. Can MINIX Be "Tuned" in Order to Satisfy Hard Real-time Constraints without Losing Its Soul?, February 2016.

[AL19] Emery K Assogba and Marc Lobelle. Hardening application programs by the operating system on cots processors: what protection to sed can be expected and at what performance cost. In *2017 17th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pages 1–6. IEEE, 2019.

[Amo15] Mohammed Amoon. A framework for providing a hybrid fault tolerance in cloud computing. In *2015 Science and Information Conference (SAI)*, pages 844–849. IEEE, 2015.

[Ass19] Emery K. Assogba. Hardened minix3.2.1. https://github.com/akemery/HardenedMinix3.2.1, 2019.

[BBK+10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 7. ACM, 2010.

[BBN+07] Michael Bajura, Younes Boulghassoul, Riaz Naseer, Sandeepan DasGupta, Arthur F Witulski, Jeff Sondeen, Scott D Stansberry, Jeffrey Draper, Lloyd W Massengill, John N Damoulakis, et al. Models and algorithmic limits for an ECC-based approach to hardening sub-100-nm SRAMs. *Nuclear Science, IEEE Transactions on*, 54(4):935–945, 2007.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[BGW93] Amnon Barak, Shai Guday, and Richard G Wheeler. *The MOSIX distributed operating system: load balancing for UNIX*, volume 13. Springer, 1993.

[BKRF02] Douglas C Bossen, Alongkorn Kitamorn, Kevin F Reick, and Michael S Floyd. Fault-tolerant design of the ibm pseries 690 system using power4 processor technology. *IBM Journal of Research and Development*, 46(1):77–86, 2002.

[BL98] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4-5):361–372, 1998.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[BMS07] Cristiana Bolchini, Antonio Miele, and Marco D Santambrogio. TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs. pages 87–95. IEEE, 2007.

[BPP+08] Melanie Berg, C Poivey, D Petrick, D Espinosa, Austin Lesea, KA LaBel, M Friendlich, H Kim, and Anthony Phan. Effectiveness of internal versus external SEU scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis. *IEEE Transactions on Nuclear Science*, 55(4):2259–2266, 2008.

[But11] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[CC16] Yi-Shen Chen and Peng-Sheng Chen. A Software-Based Redundant Execution Programming Model for Transient Fault Detection and Correction. In *Parallel Processing Workshops (ICPPW), 2016 45th International Conference on*, pages 66–71. IEEE, 2016.

[CDL+16] Ediz Cetin, Oliver Diessel, Tuo Li, Jude A Ambrose, Thomas Fisk, Sri Parameswaran, and Andrew G Dempster. Overview and Investigation of SEU Detection and Recovery Approaches for FPGA-Based Heterogeneous Systems. In *FPGAs and Parallel Architectures for Aerospace Applications*, pages 33–46. Springer, 2016.

[cgIc11] Data center group Intel corporation. Intel xeon processor e7family:reliability,availability,and serviceability. (52740), 2011.

[Chu96] Hsiao-keng Jerry Chu. Zero-copy tcp in solaris. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 21–21. Usenix Association, 1996.

[CNV+00] Phillipe Cheynet, Bogdan Nicolescu, Raoul Velazco, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6):2231–2236, 2000.

[Cor18] Intel Corporation. Mca enhancements in intel xeon processors, 2018.

[CRK+15] Eduardo Chielle, Gennaro S Rodrigues, Fernanda L Kastensmidt, Sergio Cuenca-Asensi, Lucas A Tambara, Paolo Rech, and Heather Quinn. S-seta: Selective software-only error-detection technique using assertions. *IEEE transactions on Nuclear Science*, 62(6):3088–3095, 2015.

[CY12] Sanguhn Cha and Hongil Yoon. Efficient implementation of single error correction and double error detection code with check bit pre-computation for memories. *JSTS: Journal of Semiconductor Technology and Science*, 12(4):418–425, 2012.

[Dau16] Nathan D Dautenhahn. *Protection in commodity monolithic operating systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2016.

[DeL08] Eric DeLano. Tukwila-a quad-core intel® itanium® processor. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–29. IEEE, 2008.

[Del18] Samuel Francis Delaney. *RealPi-A Real Time Operating System on the Raspberry Pi*. PhD thesis, University of Nevada, Reno, 2018.

[Dev11] Android Developers. What is android, 2011.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[DKD+15] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *ACM SIGPLAN Notices*, volume 50, pages 191–206. ACM, 2015.

[Döb14] Björn Döbel. *Operating system support for redundant multithreading*. PhD thesis, Saechsische Landesbibliothek-Staats-und Universitaetsbibliothek Dresden, 2014.

[DWA+19] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[EH13] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[EM00] Eric Espie and Zoltan Menyhart. Method of memory error correction by scrubbing, June 13 2000. US Patent 6,076,183.

[FHR10] Michael Falk, Jürg Hüsler, and Rolf-Dieter Reiss. *Laws of small numbers: extremes and rare events*. Springer Science & Business Media, 2010.

[FLB15] Charlotte Frenkel, Jean-Didier Legat, and David Bol. A Partial Reconfiguration-based scheme to mitigate Multiple-Bit Upsets for FPGAs in low-cost space applications. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on*, pages 1–7. IEEE, 2015.

[GDJ+11] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, page 239, 2011.

[GJR+92] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, 1992.

[GKT13a] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, pages 31–40. IEEE, 2013.

[GKT13b] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Safe and automatic live update for operating systems. *ACM SIGARCH Computer Architecture News*, 41(1):279–292, 2013.

[GRRV03] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 581–588. IEEE, 2003.

[Gui11] Part Guide. Intel® 64 and ia-32 architectures software developerâĂŹs manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[GWJL18] Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. The state of fault injection vulnerability detection. In *International Conference on Verification and Evaluation of Computer and Communication Systems*, pages 3–21. Springer, 2018.

[HA09] Sharon P Hall and Eric Anderson. Operating systems for mobile computing. *Journal of Computing Sciences in Colleges*, 25(2):64–71, 2009.

[HAM+18] Mohd Hakim Abdul Hamid, Nur Azman Abu, Siti Nurul Mahfuzah Mohamad, Ariff Idris, Zahriladha Zakaria, and Zuraidah Sulaiman. Data analytics algorithm benchmark on distributed systems. In *AIP Conference Proceedings*, volume 2016, page 020002. AIP Publishing, 2018.

[HAR15] G Hubert, L Artola, and D Regis. Impact of scaling on the soft error sensitivity of bulk, FDSOI and FinFET technologies due to atmospheric radiation. *Integration, the VLSI journal*, 50:39–47, 2015.

[HBG+09] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 33–42. IEEE, 2009.

[Hei11] Tino Heijmen. Soft errors from space to ground: Historical overview, empirical evidence, and future trends. In *Soft Errors in Modern Electronic Systems*, pages 1–25. Springer, 2011.

[Her10] Jorrit N Herder. Building a dependable operating system: fault tolerance in MINIX 3. 2010.

[HH16] Hsuan Hsu and Chih-Wen Hsueh. Freertos porting on x86 platform. In *2016 International Computer Symposium (ICS)*, pages 120–123. IEEE, 2016.

[Hil92] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[HQSD00] D Heynderickx, B Quaghebeur, E Speelman, and EJ Daly. ESA's SPace ENVironment Information System (SPENVIS): a WWW interface to models of the space environment and its effects. *Proc. AIAA*, 371, 2000.

[Int16] Intel Corporation. Intel 64 and IA-32 architectures software developers manual. 2016.

[Int64] Intel Intel. and IA-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1(64), 64.

[JKH+16] Qamar Jabeen, Fazlullah Khan, Muhammad Nouman Hayat, Haroon Khan, Syed Roohullah Jan, and Farman Ullah. A survey: Embedded systems supporting by different operating systems. *arXiv preprint arXiv:1610.07899*, 2016.

[JLT85] E Douglas Jensen, C Douglas Locke, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. In *Rtss*, volume 85, pages 112–122, 1985.

[JSDK13] Xun Jian, John Sartori, Henry Duwe, and Rakesh Kumar. High performance, energy efficient chipkill correct memory with multidimensional parity. *IEEE Computer Architecture Letters*, 12(2):39–42, 2013.

[KA96] Sandeep S Kulkarni and Anish Arora. Stepwise design of tolerances in barrier computations. Technical report, Citeseer, 1996.

[Kan09] Peter Kankowski. x86 machine code statistics, 2009.

[KEGW96] M Frans Kaashoek, Dawson R Engler, Gregory R Ganger, and Deborah A Wallach. Server operating systems. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, pages 141–148. ACM, 1996.

[KF82] Shigeo Kaneda and Eiji Fujiwara. Single byte error correcting? double byte error detecting codes for memory systems. *IEEE Transactions on Computers*, (7):596–602, 1982.

[KK07] Israel Koren and Mani Krishna. *Fault-tolerant systems*. Elsevier/Morgan Kaufmann, 2007.

[Kop11] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[KT91] M Frans Kaashoek and Andrew S Tanenbaum. Group communication in the amoeba distributed operating system. In *[1991] Proceedings. 11th International Conference on Distributed Computing Systems*, pages 222–230. IEEE, 1991.

[Lev09] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[LKC17] Jongwon Lee, Jaejun Ko, and Young-June Choi. Dhrystone million instructions per second–based task offloading from smartwatch to smartphone. *International Journal of Distributed Sensor Networks*, 13(11):1550147717740073, 2017.

[LLL+11] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 39–50. ACM, 2011.

[LML11] Laurent Lesage, Boris Mejías, and Marc Lobelle. A software based approach to eliminate all SEU effects from mission critical programs. *Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on*, pages 467–472, 2011.

[LNP90] Kai Li, Jeffrey F Naughton, and James S Plank. *Real-time, concurrent checkpoint for parallel programs*, volume 25. ACM, 1990.

[LNP94] Kai Li, Jeffrey F. Naughton, and James S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE transactions on Parallel and Distributed Systems*, 5(8):874–879, 1994.

[LS17] Mao Lucia and Yuan Spike. Server ras and uefi cper, 2017.

[LY09] Feida Lin and Weiguo Ye. Operating system battle in the ecosystem of smartphone industry. In *2009 international symposium on information engineering and electronic commerce*, pages 617–621. IEEE, 2009.

[MAAB11] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 12–17. IEEE, 2011.

[MAAB12] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. A user-level library for fault tolerance on shared memory multicore systems. In *Design and Diagnostics of Electronic*

Circuits & Systems (DDECS), 2012 IEEE 15th International Symposium on, pages 266–269. IEEE, 2012.

[MAAB13] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Efficient software-based fault tolerance approach on multicore platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 921–926. EDA Consortium, 2013.

[Mat10] Thierry Mataigne. Thalès alenia space etca internal document. Technical report, 2010.

[MB05] Riccardo Mariani and Gabriele Boschi. Scrubbing and partitioning for protection of memory systems. In *On-Line Testing Symposium, 2005. IOLTS 2005. 11th IEEE International*, pages 195–196. IEEE, 2005.

[MCV00] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 417–426. IEEE, 2000.

[MJK+18] MA MEMON, AK JUMANI, MY KOONDHAR, M MEMON, and AG MEMON. A technique to differentiate clustered operating systems. *Sindh University Research Journal (Science Series)*, 50(3D):89–94, 2018.

[MN12] Anil Agrawal Mahesh Natu, Narayan Ranganathan. Autonomic foundation for fault diagnosis. *Intel Technology Journal*, 16(2):8–30, 2012.

[Mul17] D Mulnix. Intel® xeon® processor scalable family technical overview, 2017.

[MVJ11] Tipp Moseley, Neil Vachharajani, and William Jalby. Hardware performance monitoring for the rest of us: a position and survey. In *IFIP International Conference on Network and Parallel Computing*, pages 293–312. Springer, 2011.

[NC01] Wee Teck Ng and Peter M Chen. The design and verification of the Rio file cache. *IEEE Transactions on Computers*, 50(4):322–337, 2001.

[NCDM13] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2013.

[Nee16] Ralf Neeb. Porting MINIX3 to x86-based Hardware, February 2016.

[Ngu17] Khang T Nguyen. New reliability, availability, and serviceability (ras) features in the intelÂő xeonÂő processor familyÂă, 2017.

[Nic02] Bogdan Nicolescu. *Détection d'erreurs transitoires survenant dans des architectures digitales par une approche logicielle: principes et résultats expérimentaux.* PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2002.

[Nic11] Michael Nicolaidis. Circuit-Level Soft-Error Mitigation. In *Soft Errors in Modern Electronic Systems*, pages 203–252. Springer, 2011.

[NKH12] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Mobile computing: principles, devices and operating systems. *World Applied Programming*, 2(7):399–408, 2012.

[Nor96] E. Normand. Single event upset at ground level. *Nuclear Science, IEEE Transactions on*, 43(6):2742–2750, 1996.

[OKB+16] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Christof Fetzer, and Pascal Felber. Efficient Fault Tolerance using Intel MPX and TSX. In *Fast Abstract in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[PBR17] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. Fault attacks, injection techniques and tools for simulation. In *Hardware Security and Trust*, pages 27–47. Springer, 2017.

[Pet02] Zachary Nathaniel Joseph Peterson. *Data placement for copy-on-write using virtual contiguity.* PhD thesis, University of California, Santa Cruz 2002., 2002.

[Pla80] DG Platteter. Transparent protection of untestable LSI microprocessors. In *10th Fault-Tolerant Computing Symposium*, pages 345–347, 1980.

[Pro11] Intel Xeon Processor. E7 Family: Reliability. *Availability and Serviceability: Advanced data integrity and resiliency support for mission-critical deployment*, 2011.

[Pul01] L.L. Pullum. *Software fault tolerance techniques and implementation*. Artech House Publishers, 2001.

[Qua00] Nhon Quach. High availability and reliability in the itanium processor. *IEEE Micro*, (5):61–69, 2000.

[Rat15] David Ratajczak. *Is Linux a better desktop operating system than Microsoft Windows?* GRIN Verlag, 2015.

[RCV+05] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[RDH+01] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. Pilot: An operating system for a personal computer. In *Classic operating systems*, pages 433–459. Springer, 2001.

[Rot99] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 84–91. IEEE, 1999.

[RRTV99] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. Soft-error detection through software fault-tolerance techniques. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium on*, pages 210–218. IEEE, 1999.

[RRV11] Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Software-Level Soft-Error mitigation techniques. In *Soft Errors in Modern Electronic Systems*, pages 253–285. Springer, 2011.

[SBD+17] Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. Effectiveness of software-based hardening for radiation-induced soft errors in real-time operating systems. In *International Conference on Architecture of Computing Systems*, pages 3–15. Springer, 2017.

[SDB+15] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *ACM SIGPLAN Notices*, 50(4):297–310, 2015.

[SGG18] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[SHD+15] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 245–255. IEEE, 2015.

[SHT10] Bjorn P Swift, Tomas Hardy, and Andrew Tanenbaum. Individual Programming Assignment User Mode Scheduling in MINIX 3. 2010.

[Sib84] Edgar H Sibley. Dhrystone: A synthetic systems. *Communications of the ACM*, 27(10), 1984.

[SPR00] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000.

[Spr02a] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[Spr02b] Brinkley Sprunt. Pentium 4 performance-monitoring features. *Ieee Micro*, (4):72–82, 2002.

[SPW11] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. *Communications of the ACM*, 54(2):100–107, 2011.

[SZF16] Jinhao Sun, Fang Zou, and Shangchun Fan. A token-ring-like real-time response algorithm of modbus/tcp message based on μc/os-ii. *AEU-International Journal of Electronics and Communications*, 70(2):179–185, 2016.

[Tan16] Andrew S Tanenbaum. Lessons learned from 30 years of MINIX. *Communications of the ACM*, 59(3):70–78, 2016.

[THB06] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[TRA18] TRAD. Omere software. 2018.

[TVRVS+90] Andrew S Tanenbaum, Robbert Van Renesse, Hans Van Staveren, Gregory J Sharp, and Sape J Mullender. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[TVVB18] Rodrigo Travessini, Paulo RC Villa, Fabian L Vargas, and Eduardo Augusto Bezerra. Processor core profiling for seu effect analysis. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6. IEEE, 2018.

[TW87] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[vB98] Ladislaus von Bortkiewicz. *Das Gesetz der kleinen Zahlen The law of small numbers*, volume 1. Leipzig, Germany: B.G. Teubner, 1898.

[VFR07] Raoul Velazco, Pascal Fouillat, and Ricardo Reis. *Radiation effects on embedded systems*. Springer Science & Business Media, 2007.

[WABM04] Alan G White, Jaison R Abel, Ernst R Berndt, and Cory W Monroe. Hedonic price indexes for personal computer operating systems and productivity suites. Technical report, National Bureau of Economic Research, 2004.

[Wei84] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[Whi03] James A Whittaker. *How to break software*. Addison-Wesley, 2003.

[WVB+15] Imran Wali, Arnaud Virazel, Alberto Bosio, Luigi Dilillo, and Patrick Girard. An effective hybrid fault-tolerant architecture for pipelined cores. In *2015 20th IEEE European Test Symposium (ETS)*, pages 1–6. IEEE, 2015.

[YGS08] Jing Yu, María Jesús Garzarán, and Marc Snir. Efficient software checking for fault tolerance. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–5. IEEE, 2008.

[Yor02] Richard York. Benchmarking in context: Dhrystone. *ARM, March*, 2002.

[ZJH09] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32. IEEE, 2009.

[ZLJA12] Yun Zhang, Jae W Lee, Nick P Johnson, and David I August. DAFT: decoupled acyclic fault tolerance. *International Journal of Parallel Programming*, 40(1):118–140, 2012.