Master's Thesis

Extending Mondrian Memory Protection

carried out at the

Automation Systems Group
Vienna University of Technology

under the guidance of
Priv. Doz. Dipl.-Ing. Dr. techn. Christopher Krügel
and
Priv. Doz. Dipl.-Ing. Dr. techn. Engin Kirda
as the contributing advisors responsible

by

Clemens Kolbitsch
Lambrechtgasse 8/6, 1040 Wien
Matr.Nr. 0126605

Vienna, February 2008

Acknowledgements

I would like to thank a couple of people from the Vienna University of Technology, my family, and friends who have contributed to this thesis or helped me during my studies.
First of all, a big thank you to my academic advisors Engin Kirda and Christopher Krügel for the inspiration for the topic and their many comments and suggestions that helped me keep headed in the right direction. Also, the fellow students at the SecLab offered a great environment to work and have a good time in.
My parents Arno and Maria deserve a very big thank you for supporting me during my studies and also on everything else I have ever done. To my elder brother Philipp, who taught me computer programming in the first place, and his partner Theresa, my younger brother Maximilian, and my sister Eva Maria, I also want to say thank you for supporting me and being a very important part of my life.
Of course, a very special and warm thank you to my partner Manuela for her encouragement and endless hours of listening to my, at times probably quite geeky, thoughts and ideas. At last, I want to thank all my friends for making sure I occasionally spent a night away from my computer and took me out for a beer or two.

Abstract

Most modern operating systems implement some sort of memory protection for user processes. Hence, it is possible to set access permissions that determine whether a region of memory allocated for a process can be read, written, or executed by this process. Mondrian memory protection is a technique that extends the traditional memory protection scheme and allows fine-grain permission settings. Instead of being able to set access permissions on a page-level, Mondrian memory protection supports different access permissions for individual words. However, this protection scheme is still limited to only two permission bits that have a pre-defined semantics. This is not sufficient to implement more complex security techniques, for example, a race condition detection system.
The presented solution proposes an extension to the simple Mondrian protection scheme that provides more flexibility to user programs and the operating system. Based on our extended architecture, we implement mechanisms to protect sensitive data structures on the heap and on the stack. Moreover, we present the implementation of a technique to detect race conditions. Our experiments demonstrate that the system can provide the expected protection and ability to detect races with reasonable overheads. Furthermore, our results show that even large systems such as the GNU C Library and the Apache web server contain problems related to race conditions.

Zusammenfassung

Speicherschutz für Anwenderprogramme ist ein Konzept, das vom Großteil der heutzutage verwendeten Betriebssysteme bereitgestellt wird. Dieses ermöglicht es, für die einzelnen Speicherbereiche eines Prozesses unterschiedliche Zugriffsberechtigungen für Lese- und Schreib-Operationen sowie das Ausführen von Code zu setzen. Eine Erweiterung der traditionellen Schutzmechanismen ist Mondrian Memory Protection. Dieses Schema erlaubt das genaue Spezifizieren unterschiedlicher Berechtigungen auf Wort-Basis anstelle der traditionellen Speicherseiten-Basis. Allerdings ist auch hier die Spezifikation auf zwei Zugriffsbits limitiert. Zusätzlich ist die Bedeutung der einzelnen Bitkombinationen vorgegeben, was es unmöglich macht, damit komplexere Sicherheitstechniken, wie beispielsweise einen Race Condition Detector, zu implementieren.
Der Ansatz, der in dieser Arbeit präsentiert wird, ist eine Erweiterung der einfachen Mondrian Memory Protection. Sie soll eine größere Flexibilität für Anwenderprogramme und das Betriebssystem ermöglichen. Aufbauend auf unserer Architektur zeigen wir die Implementierung von Mechanismen zum Schutz von heiklen Datenstrukturen im Heap und Stack Speicher. Des Weiteren präsentieren wir eine Technik zum Erkennen von Race Conditions, die auf der vorgeschlagenen Architektur basiert.
Unsere Experimente beweisen, dass das System, bei akzeptablem Mehraufwand, den gewünschten Schutz und die Möglichkeit zum Erkennen von Race Conditions bietet. Zusätzlich zeigen die Ergebnisse, dass sogar große Systeme, wie die GNU C Bibliothek und der Apache Webserver, Probleme in Zusammenhang mit Race Conditions aufweisen.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Organization of this thesis 2

2 State of the art **4**
2.1 Intel x86 memory management 4
2.2 Mondrian memory protection 7

3 Description of our approach **10**
3.1 Protection hierarchy 11
3.2 Memory access control 15
3.3 Memory access policies 17

4 System implementation **18**
4.1 Instruction set extension 18
4.2 Kernel adaptions 26

5 Application of extended Mondrian memory protection **35**
5.1 Stack Protection 35
5.2 Heap Protection 44
5.3 Race Condition Detection 48

6 Evaluation **60**
6.1 Performance 60
6.2 Memory Overhead 62
6.3 Effectiveness of System Applications 62

7 Related Work **69**

8 Summary and conclusion 72

List of Figures

2.1 Logical to linear memory translation. [23] 5
2.2 Example segmentation of a memory address space using two segments. [23] 6
2.3 Linear to physical memory translation. [23] 7
2.4 Page-table entry of a 4 kibibyte page. [23] 8
2.5 Mondrian memory protection system design. [38] 9

3.1 Protection hierarchy. 12
3.2 Structure of the newly introduced control register CR6. 12
3.3 Structure of a protection directory entry. 13
3.4 Structure of a protection table entry. 14
3.5 Structure of a protection page entry. 14
3.6 Access control decision for a read access yielding a protection violation. 16
3.7 Access control decision for a read access granting the memory access. 16

4.1 Structure of the newly introduced control register CR7. 22
4.2 Upper half of the common protection fault base handler. . . . 32
4.3 Lower half of the common protection fault base handler. . . . 33

5.1 Stack layouts with (a) no, (b) StackGuard [15], and (c) our extended Mondrian memory protection. 38
5.2 In-band information of a free chunk [3]. 45
5.3 Race condition detection state machine [29]. 50

Chapter 1

Introduction

1.1 Motivation

Most modern operating systems implement some sort of memory protection for user processes [16, 31, 33]. That is, it is possible to set access permissions that determine whether a region of memory allocated for a process can be read, written, or executed by this process. Typically, for operating systems that support paged virtual memory, the granularity of these access permission are on a per-page basis. This means that a process or the operating system can assign different protection settings to each individual page of the process (where the size of a page is typically between 1 and 8 KB [33]).
The page-based protection is useful to mark the text portion of a process as non-writable, preventing unintentional modifications of the code due to programming errors. Another use of access permissions is to flag the data and stack segments as non-executable[1]. This increases the security [27, 32] as an attacker can no longer execute shellcode that he injects into the heap or the stack (e.g., by exploiting a buffer overflow vulnerability).
While memory protection is a useful technique to improve the reliability and security of processes, it is fairly coarse-grain. The reason is that permission settings can only be applied to complete pages. This limits the flexibility, especially when there are small memory fragments located close to each other that would require different permission settings. A standard example for such a memory area is the stack. A stack stores both data (such as function

[1]Provided that the hardware provides the necessary means to set different permissions for read and execute access, such as the No eXecute bit [22].

parameters or local variables) that a process must be able to read and write, as well as function return addresses, which should never be modified by the application. Because data and return addresses are stored in the same page, the least restrictive protection settings must be applied. As a result, a return address can be modified by an application function. This leads to the problem of buffer overflow exploits that trick a memory write operation to change the return address on the stack to a value of the attacker's choice.

Mondrian memory protection [37, 38] is a technique that extends the traditional memory protection scheme and allows fine-grain permission settings. More precisely, instead of being able to set access permissions on a page-level, Mondrian memory protection supports different access permissions for individual words. This allows a process to use different memory protection settings for different words on the stack. However, Mondrian memory protection is still limited to only *two* permission bits with a pre-defined semantics. Similar to the bits at the page-level, these permission bits control read, write, and execute access. This might not be sufficient in all cases. For example, in order to keep track of the memory accesses of multiple threads to detect race conditions, the available mechanism is insufficient. Unfortunately, race conditions are an important problem and lead to bugs and security problems that are difficult to track down [10, 11, 34, 36]. This problem is exacerbated by the increasing use of parallel programming and multi-threaded applications.

In this thesis, we propose an extension to the simple Mondrian protection scheme that provides more flexibility to user programs and the operating system. More precisely, instead of two protection bits, we propose to use 30-bit protection labels that can be assigned to each memory word. These labels can be freely used as a basic mechanism to implement different techniques such as return address protection, heap protection, or race condition detection. The protection labels are controlled via a simple interface that allows user programs controlled access to protection information. In case of a protection fault, the operating system invokes a user-defined module in the kernel that can implement a flexible policy to handle the exception.

1.2 Organization of this thesis

In this thesis, we first describe existing memory protection schemes and the general mechanisms that our extended system supports in Chapters 2

and 3, respectively. Chapter 4 handles the details of implementing the mechanisms using CPU extensions as well as operating system adaptions. Then, we discuss concrete techniques that leverage the general memory protection mechanisms to realize return address protection on the stack, heap memory protection, as well as a race condition detection in Chapter 5.
Finally, we describe our experiments that demonstrate that the system can provide the expected protection and ability to detect races with reasonable overheads in Chapter 6. In a last chapter, we put our work into context with previous publications on memory protection and race condition detection.

Chapter 2

State of the art

In this chapter, we explain memory protection as it is available on current Intel x86 CPUs. Furthermore, we deal with the idea of Mondrian memory protection as described in [38].

2.1 Intel x86 memory management

Intel's x86 processor family provides two basic concepts for protecting memory [23]:

Segmentation allows to split the memory address space visible to a user application into multiple *segments*. In this addressing mode, every *logical memory address* is represented by a segment:offset tuple. Thus, for every access to memory, one of the six *segment registers* (code segment register `CS`, stack segment register `SS` and data segment registers `DS`, `ES`, `FS`, and `GS`) has to be provided. If no such register is provided for the access, the register most appropriate for the operation is chosen (e.g. the stack segment register is automatically used when pushing a value on stack). Figures 2.1 and 2.2 show how these registers are used to access a memory address.
Each segment register either points into the *global descriptor table* or into the *local descriptor table*, holding the segment's

- base address used for translating the logical address into a linear or physical address,
- size (limit),

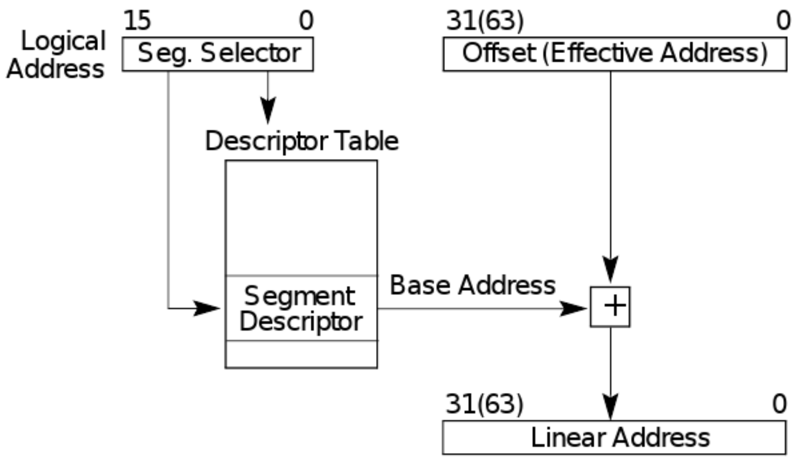

Figure 2.1: Logical to linear memory translation. [23]

- access control information, as well as
- status information.

For every segment, the operating system can provide four bits for controlling memory access, including *read-only*, *read-write*, and *execute-only* access types. To achieve portability with most architectures, Linux only employs very limited use of segmentation [30], however. In fact, Linux only employs a non-standard segment register when accessing thread specific data. By default, logical and linear addresses coincide and access is restricted through the use of *paging*, only.

Paging is Intel's second concept for memory protection: Modern operating systems divide the linear address space visible to a user program into sections of equal size, typically called *pages*. Each memory page allocated for a program is represented by a *page table entry* in the program's *page directory / page table* hierarchy. The page hierarchy is used by the operating system and the CPU's memory management unit to map virtual memory pages to the corresponding physical frames in the RAM[1]. This mapping is necessary to find the location in physical memory that corresponds to an

[1]When referring to physical addresses, a memory region holding data of a virtual page is called a frame.

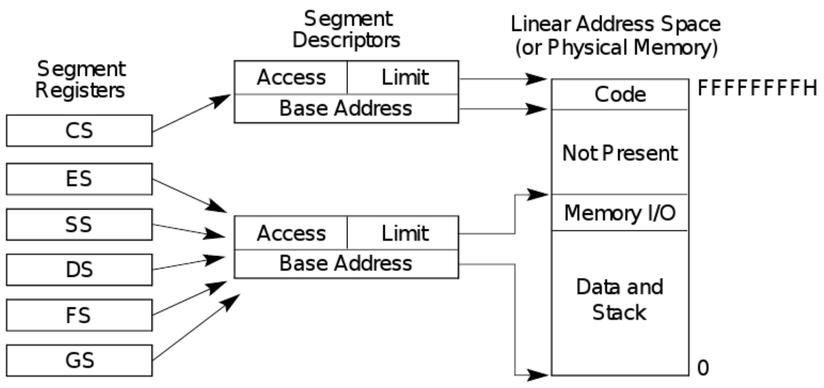

Figure 2.2: Example segmentation of a memory address space using two segments. [23]

address in the virtual address space. Figure 2.3 shows how a virtual memory address is resolved into a physical memory frame using this hierarchy and the per process unique control register `CR3`.
Figure 2.4 shows the structure of a page table entry: Each entry contains the page's physical base address used for the address mapping [2], a set of bits to store access statistics, and two bits informing the CPU whether a page is `read-only` and whether access is restricted to `supervisor` code.
On every access to a virtual memory address, the CPU consults the mapping (or possibly the `TLB`[3]) to find the respective physical memory. It then checks the aforementioned access bits. When an invalid access is detected, the CPU raises a page fault. This signals the operating system's kernel that a problem has occurred and allows for a proper reaction to resolve the problem (e.g., by terminating the offending process).
Summarizing, the use of paging has four implications: For one thing, each process has an unique address spaces (i.e. page hierarchy). This allows the kernel to protect processes from one another. Secondly, processes cannot destroy read-only memory regions (e.g. text mappings) by writing to such addresses accidentally. Furthermore, the operating system's kernel can protect vital memory structures by marking them as supervisor pages and

[2]Virtual pages have to be aligned onto a 4 KByte boundary. This leaves the 10 least significant bits to be used for other purposes.

[3]The *Translation Lookaside Buffer* stores mapping information for all recently accessed pages.

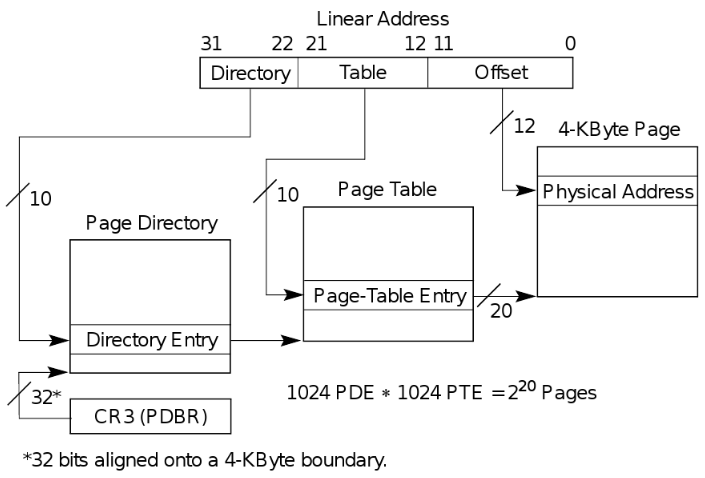

Figure 2.3: Linear to physical memory translation. [23]

lastly, paging does not provide a method for marking memory regions as non-executable.

2.2 Mondrian memory protection

While Intel x86's memory protection techniques are useful in improving the system's stability as well as reliability and security of processes, it is fairly coarse-grain. As mentioned in the introduction already, the reason is that permission settings can only be applied to complete pages. This limits the flexibility, when having to apply the least restrictive protection settings to different adjacent memory areas.
Mondrian[4] memory protection [37, 38] is a technique that extends Intel's traditional memory protection scheme and allows fine-grain permission settings: Similar to the x86 architecture, Mondrian memory protection employs two bits to store four different access permissions (*no*, *read-only*, *read-write*, and *execute-read*) for every memory region available in the system. However, instead of storing the permission information in the per-process unique page

[4]The authors of [38] called their protection system Mondrian memory protection (MMP), because figures of protection ranges occasionally resembled works by the eponymous early twentieth century artist.

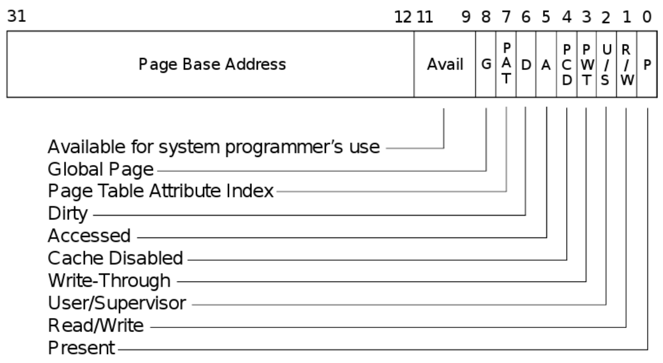

Figure 2.4: Page-table entry of a 4 kibibyte page. [23]

hierarchy, Mondrian memory protection uses an additional *permissions table*. This allows the system to store protection information for each memory word (instead of the page-level granularity of the x86 architecture). Moreover, every thread is a member of one *protection domain*, possibly sharing the permission table with other threads of the same domain.
On every access to a memory address, the CPU's protection enhancement consults the currently executing thread's protection domain register to look up the address protection bits stored in the corresponding protection table. To reduce the memory overhead introduced by the protection tables, the implementation in [38] provides different possibilities for storing the table's structure. This allows to adjust the size of the region the protection information applies to.
Figure 2.5 shows a design overview of Mondrian memory protection. To avoid repeated look ups of access permissions, a *permissions lookaside buffer* (PLB) caches entries from the permissions table. Furthermore, every address register contains an additional *sidecar register*. This sidecar register stores the last table segment accessed through the corresponding address register.

Summarizing, Mondrian memory protection proposes finer-grained memory protection than Intel's x86 standard. Each word in the memory address space can be assigned two permission bits. This overcomes the problem of least restrictive protection on the stack, where read-only and writable memory addresses are stored adjacently. To minimize the memory overhead for storing permission information, multiple words sharing common permissions can be

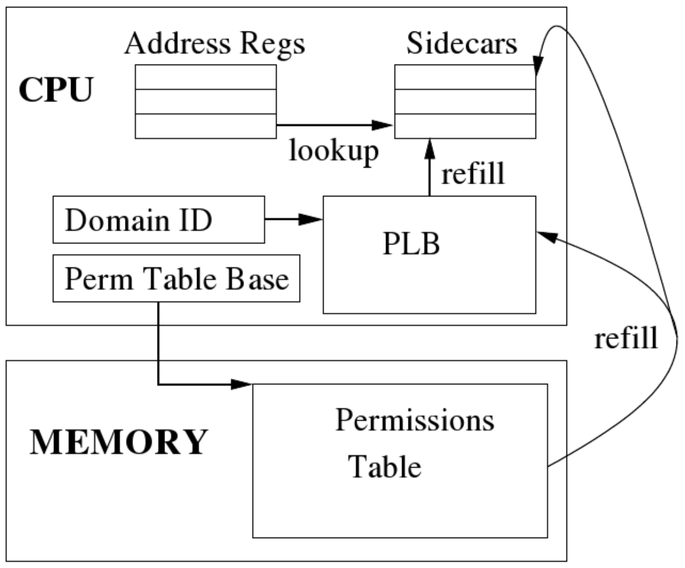

Figure 2.5: Mondrian memory protection system design. [38]

grouped. Furthermore, a permissions lookaside buffer and sidecar registers are used to increase the system's performance.

Chapter 3

Description of our approach

In Chapter 2, we described Intel's standard x86 protection as well as Mondrian memory protection schemes. Despite the simplicity and broad acceptance of the former and flexibility of the latter, they share one shortcoming - which is the fact that one cannot associate more than two bits of protection information with a memory region. Moreover, the pre-defined meaning of the four possible bit combinations significantly limits the flexibility of the protection system.
These two drawbacks are the starting point of the extended Mondrian memory protection technique described in this chapter: While trying to combine the simplicity of x86 memory protection with the fine granularity of the original Mondrian memory protection, this implementation allows a *user-specified examination* of *expanded protection information* stored for memory regions. For this purpose, the protection architecture is split into the following three components:

- The first component implements the *protection hierarchy*. This protection hierarchy stores, for each process, a mapping between *30-bit protection labels* and memory addresses. That is, it is possible to associate a 30-bit protection label with each word in the process address space. This is a generalization of the two bits used by the original Mondrian memory protection. This component also provides an interface that lets user processes and the OS kernel modify the protection settings.

- The second component is responsible for checking memory accesses. To this end, the system uses a special *protection control register*. The content of this control register is compared with the protection label associated with the memory location that is accessed. In case there is a mismatch, a protection fault is invoked. In addition to the control register, there are two bit-masks that allow to refine (or modify) the value of the control register prior to this comparison, depending on whether the process performs a read or a write operation.

- The third component, called the *protection fault handler*, implements the response to a protection fault. This handler code is realized as a loadable kernel module, which allows users to define complex policies that can be exchanged while the system is running. In the protection fault handler, the system can change the protection settings of certain memory areas, as well as the content of the protection control register (and the associated bit-masks).

3.1 Protection hierarchy

Similar to the page hierarchy, which is used in the x86 architecture to perform a mapping from virtual to physical addresses, our extended Mondrian memory protection uses a two-level hierarchy of *protection tables.* That is, there is a *protection directory* that stores entries that point to *protection tables.* Each protection table, in turn, has entries that point to *protection pages.* Each allocated word of virtual memory is represented by an entry in the protection page. The newly introduced register `CR6` serves as entry point into the protection hierarchy. An overview of the protection hierarchy can be seen in Figure 3.1.
To save space when the protection labels of all words in a particular page are identical, we use three different levels of granularity:

- *High granularity protection*: This method adds 30 bits of protection information to every word in the virtual address space. The protection information is stored in a protection page allocated in the process' virtual address space, but is protected from direct access by user code.

- *Low granularity protection*: This method stores protection information directly into the entry of the protection table, allowing to specify 30 bits of protection information for a complete page of virtual memory.

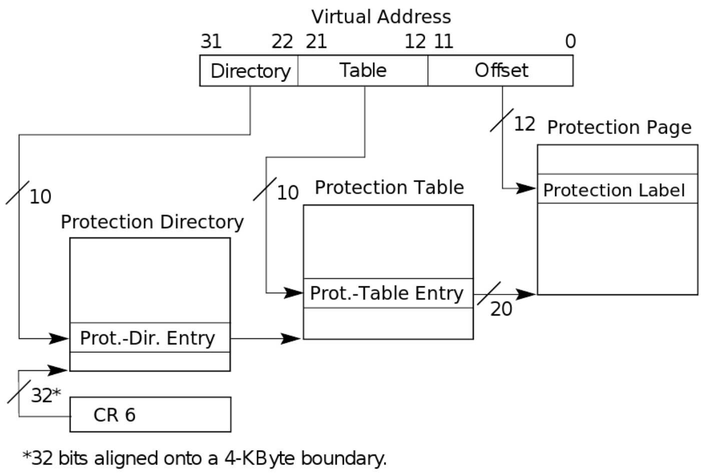

Figure 3.1: Protection hierarchy.

- *Minimal granularity protection*: This method stores protection information directly into the entry of the protection directory, allowing to specify 30 bits of protection information for a set of 1024 adjacent pages of virtual memory.

The structure of the individual entries in Figure 3.1 can be seen in Figures 3.2, 3.3, 3.4, and 3.5. The meanings of the individual fields and bits are as follows:

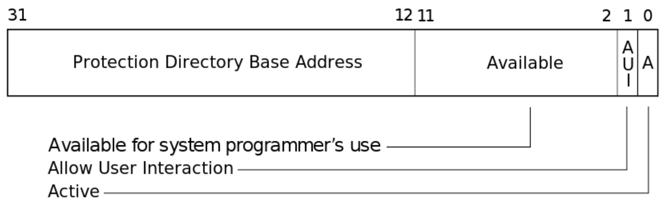
31
12
11
2
1
0
Protection Directory Base Address
Available
AUI
A
Available for system programmer's use
Allow User Interaction
Active

Figure 3.2: Structure of the newly introduced control register CR6.

Control register CR6:

- **Active bit** (bit 0): Indicates whether the extended Mondrian memory protection system is currently active. If this bit is cleared, bits 1 through 31 my be used arbitrarily by the programmer.
- **Allow user interaction bit** (bit 1): Indicates whether non-supervisor code may inspect/modify protection information. For details on protection information interaction, refer to later sections.
- **Protection directory base address** (bits 12 through 31): Specifies the most significant bits of the virtual address of the first byte in the protection directory. This forces the protection directory to be aligned on a 4-KByte boundary.

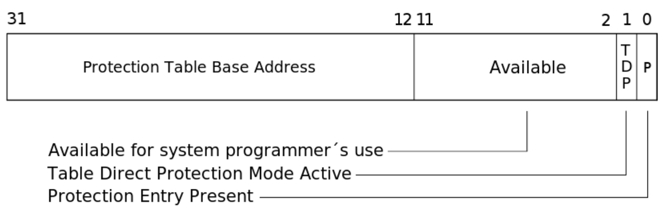
31
12 11
2 1 0
Protection Table Base Address
Available
T D P
P
Available for system programmer´s use
Table Direct Protection Mode Active
Protection Entry Present

Figure 3.3: Structure of a protection directory entry.

Protection directory entry:

- **Present bit** (bit 0): Indicates whether the protection directory entry should be used during a protection look up. If this bit is cleared, bits 1 through 31 may be used arbitrarily by the programmer.
- **Table direct protection mode active** (bit 1): Indicates if the 1024 pages referenced by this protection directory entry use minimal granularity protection. If the bit is set, bits 2 through 31 hold protection information directly.
- **Protection table base address** (bits 12 through 31): Specifies the most significant bits of the virtual address of the first byte in the protection table. This forces protection tables to be page-aligned.

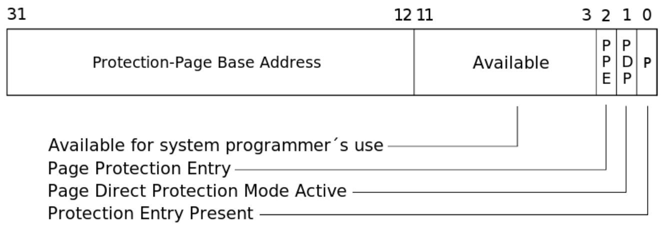
31
12 11
3 2 1 0
Protection-Page Base Address
Available
P P E
P D P
P
Available for system programmer´s use
Page Protection Entry
Page Direct Protection Mode Active
Protection Entry Present

Figure 3.4: Structure of a protection table entry.

Protection table entry:

- **Present bit** (bit 0): Indicates whether the protection table entry should be used during a protection look up. If this bit is cleared, bits 1 through 31 may be used arbitrarily by the programmer.
- **Page direct protection mode active** (bit 1): Indicates if the page referenced by this protection table entry uses low granularity protection. If the bit is set, bits 2 through 31 hold protection information directly.
- **Page protection entry** (bit 2): Indicates if the page referenced by this protection table entry is a protection page. As protection pages lie in the user address space, applications must be prevented from tampering with the protection information directly. This bit is therefore the equivalent to the supervisor bit of x86 memory protection.
- **Protection page base address** (bits 12 through 31): Specifies the most significant bits of the virtual address of the first byte in the protection page. Thus, page-alignment is required for protection pages.

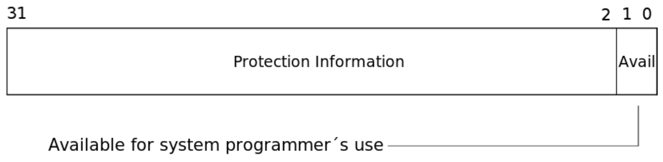

Figure 3.5: Structure of a protection page entry.

3.2 Memory access control

When performing a memory access, the CPU has to do a look up of the protection information for the corresponding address. This is done by navigating through the protection hierarchy, starting from the current value of the control register `CR6` (as described in the previous section). When high granularity protection is used, the protection table entry looked up by the CPU serves as pointer to a protection page, which contains the 30-bit protection label used for memory access control. Otherwise, in case of low or minimal granularity protection, the corresponding bits of the protection directory/table entry are directly used for access control (thus the term *table/page direct protection mode*).
In case no protection information is found (because the protection directory or protection table does not contain a corresponding entry or protection has been disabled through the corresponding bit in the control register `CR6`), the access to the memory address is immediately granted. Also, note that regardless of the granularity level, it is possible that a memory access requires looking up more than one protection label. Typically, this happens when a multi-byte access is unaligned or spans two pages. In these cases, the memory protection checks all protection labels. Access is only granted when all labels permit it.
Once a 30-bit protection label is retrieved, it can be used to perform an access control decision. That is, given this label and additional information, the system must decide whether an access should be granted or whether a protection fault should be thrown. The aforementioned additional information that allows the access decision to be made is the value of a new processor register, the protection control register `CR5`. In addition, there are two access bit-masks, called a `read-mask` and a `write-mask`.
To reach an access control decision, the system takes the 30-bit protection label obtained during look up and performs a logic `AND` operation with the appropriate access bit-mask (depending on whether this is a read or write access). The result of this operation is a *protection token.* Similarly, the values currently stored in the control register `CR5` and the mask are `AND`ed, obtaining a *control token.* Comparing both tokens decides if the current memory access should be granted or not. More precisely, a protection fault is raised in case the two tokens do not match. Figures 3.6 and 3.7 show two examples for access control decisions that yield different results.
Early tests of our protection system showed that it is very desirable to have a

	Hex	**Binary**
Protection label	`0x000271d0`	`0b00000000000000100111000111010000`
`CR5`	`0x000211d0`	`0b00000000000000100001000111010000`
`Read mask`	`0x8003fffc`	`0b10000000000000111111111111111100`
Result		**Protection violation**

Figure 3.6: Access control decision for a read access yielding a protection violation.

	Hex	**Binary**
Protection label	`0x000271d0`	`0b00000000000000100111000111010000`
`CR5`	`0x070271d0`	`0b00000111000000100111000111010000`
`Read mask`	`0x8003fffc`	`0b10000000000000111111111111111100`
Result		**Read access granted**

Figure 3.7: Access control decision for a read access granting the memory access.

mechanism that allows deactivation of the protection examination for a single instruction or memory access. To handle such situations, we introduced two read-and-clear bits in the CPU's flag set, one for read- and one for write-accesses.
The CPU consults these flags before signalling a protection violation. If the according flag bit is set, the fault is skipped and both bits are reset to zero. The use of two flag bits instead of just one comes from the fact that some x86 assembler instructions[1] may cause multiple accesses into memory that might want to be handled individually.

[1]E.g. **cmpxchg mem32 reg32** compares register `EAX` with memory address mem32, modifying either register reg32 or memory mem32, depending on the result of the comparison.

3.3 Memory access policies

As previously mentioned, the extended Mondrian memory protection does not specify any specific meaning for the individual bits protecting a memory address. The system only performs access control checks as outlined above. The way in which the protection labels and the content of the protection register (together with the bit-masks) are used is completely up to the user of the system. In Chapter 5, we demonstrate the flexibility of the approach by showing how different applications can be implemented on top of the general architecture.
To specify rules or policies for using the memory protection system, the user has two mechanisms. On the one hand, a program (or a compiler) can use a set of newly introduced instructions to manipulate the memory protection settings (labels) during process execution. The following set of new machine instructions has been introduced allowing a process to read, set, or modify protection information:

- `prot_mov reg, mem`: Load the protection information of a memory address into a register,
- `prot_mov mem, reg/imm`: Set the protection information of a memory address to an immediate or register's value,
- `prot_and mem, reg/imm`: `AND` the protection information of a memory address with an immediate or register's value, and
- `prot_or mem, reg/imm`: `OR` the protection information of a memory address with an immediate or register's value.

In addition to this first mechanism, the user can load a kernel module into the operating system that defines the protection fault handler. This protection fault handler can be arbitrarily complex and runs in the context of the kernel. Thus, it has full control over both the control registers and the memory protection information. Also, the kernel module is notified whenever a new process or thread is started, or when the operating system schedules a new thread. This allows the system to react to events that might require to load thread- or process-specific protection values.

Chapter 4

System implementation

4.1 Instruction set extension

This section deals with the details of providing extended Mondrian memory protection. To provide the instructions to modify the protection labels, the instruction set of the x86 processor needs to be extended. Also, we had to add additional control registers and a cache similar to a translation look-aside buffer, which is responsible for caching the protection labels for recently accessed memory locations.
The open source system emulator Qemu [8] served as base for our implementation. Besides the necessary processor extensions, we extended the code for translating virtual addresses to also look up protection labels and to do the necessary access control checks. Similar to the occurrence of a page fault, protection faults are passed to the emulated system using interrupts, and thus, need no special extensions. Furthermore, this section explains additional extensions made to Qemu. These extensions are not part of the actual protection system but were necessary to evaluate the experiments described in Chapter 6.

Qemu internals

When simulating a guest system on a host computer, there are two general approaches: *Virtualization* allows to run the guest's machine code on the host directly. Obviously, this requires some sort of support by the underlying hardware and operating system alike. *Emulation*, on the other hand, is a much simpler approach and does not require any specific hardware or

operating system support. The big disadvantage is that emulated systems are much slower, in general.
Qemu uses a combination of both approaches, unifying simplicity of emulation and some speed advantages of virtualization: Every block of machine code executed by the guest system is first translated into code understandable by the host system. Register manipulations are translated into code storing the new values in Qemu's internal memory structures and every memory access inside this *translation block* (or TB) is sanitized by simulating a memory management unit, checking presence of the accessed pages and accordance of their protection bits and access type. This TB is then run directly on the host's CPU and cached to minimize the overhead of translation in case the the same code is executed repeatedly.

Control register extension

Due to Qemu's code translation approach, it is very easy to add new items to the CPU instruction set. To add the control registers `CR5`, `CR6`, and `CR7` (this control register will be discussed shortly), only the functions responsible for code translation and Qemu's internal CPU state structure had to be adjusted. Setting the `CR5` read- and write-masks and the read-and-clear flags mentioned in Section 3.2 was implemented using the two least significant bits of `CR5`. As protection labels can only hold 30 bits of information, these two extra bits in the control register are used as shown in Table 4.1.

Bits 1 and 0	**Meaning**
`0x00`	Remaining bits (bits 31 through 2) are used as new *value* of the control register
`0x10`	Read-and-clear flags for disabling read- and write-protection are set depending on the remaining bits 3 and 4, respectively
`0x01`	Remaining bits are used as `CR5` *read*-mask
`0x11`	Remaining bits are used as `CR5` *write*-mask

Table 4.1: Bits 0 and 1 of control register *CR5*.

Access control enforcement

As described in the previous chapter, every access to a page protected by our extended Mondrian memory protection requires enforcement of the memory access policies. For this reason, TB code not only includes memory access sanitation, as mentioned in Section 4.1, but also resolves the protection label for the accessed memory address.
The code then checks access permissions using `CR5` and the appropriate mask, as described in Section 3.2. In case of an access violation, the page fault method is used to trigger an interrupt. The emulated guest system can distinguish between a casual page fault and a protection violation using the extended error code table, as shown in Table 4.2.

Bit	Bit mask	Meaning
Standard x86:		
0	0x0000000000	Accessed page not found
	0x0000000001	Page access violation
1	0x0000000000	Read access
	0x0000000010	Write access
2	0x0000000000	Kernel mode access
	0x0000000100	User mode access
3	0x0000001000	Access to reserved bit
4	0x0000010000	Access was instruction fetch
Extended Mondrian memory protection:		
5	0x0000100000	Protection violation
6	0x0000000000	4-byte access (possibly unaligned)
	0x0001000000	8-byte access (possibly unaligned)
7	0x0010000000	Reserved for kernel internal use
8	0x0100000000	Reserved for race detection system
9	0x1000000000	Protection violation is user protection information interaction

Table 4.2: Error code signalling a page fault or protection violation.

The x86 memory management unit employs a TLB to store the physical base addresses of recently accessed pages. Qemu's Mondrian memory protection

extension additionally stores the protection table entry found during the look up. Thus, the protection label can be fetched directly from the TLB when using minimal or low granularity protection. For high granularity protection, the virtual address of the protection page can be taken from the TLB allowing to skip parsing the protection hierarchy.
To quicken access control even further, the extended TLB also holds the physical base address of the associated protection page. This allows to reference protection labels directly, even if high granularity protection is employed. The drawback of this last enhancement is the fact that when a protection page is swapped out (i.e. removed from RAM), all references to this now unavailable memory frame have to be removed from the TLB.

Protection manipulation instructions

To keep compliance with existing compilers and code inspection tools (such as debuggers and disassemblers), the machine instructions to access the memory protection settings were realized by adding another control register `CR7`[1]. Furthermore, unlike access to other control registers, user mode code is allowed to access and modify the content of this register.
To implement the protection instructions, each bit assigned to this register was given a special meaning, indicating source and destination registers as well as the requested modification operation. That is, the `prot*` instructions introduced in Section 3.3 are expressed as instructions that modify the control register `CR7`.
Figure 4.1 shows the bit layout used when interacting with the control register. The meaning of the individual bits is as follows[2]:

- **Operation** (bits 1 and 0): Defines the requested operation: `Get` (`0x00`), `Set` (`0x01`), `AND` (`0x10`), and `OR` (`0x11`).
- **Address Register** (bits 4, 3, and 2): The register holding the memory address to be inspected/modified: `EAX` (`0x000`), `ECX` (`0x001`), `EDX` (`0x010`), `EBX` (`0x011`), `ESP` (`0x100`), `EBP` (`0x101`), `ESI` (`0x110`), and `EDI` (`0x111`).

[1]Although only control registers `CR1` through `CR4` and `CR8` may be used currently, Intel has specified the op-codes for `CR1` through `CR15`. Thus, most compilers and code inspection tools can already handle code using these registers.

[2]Bits 31 through 29 will be dealt with in the following sections.

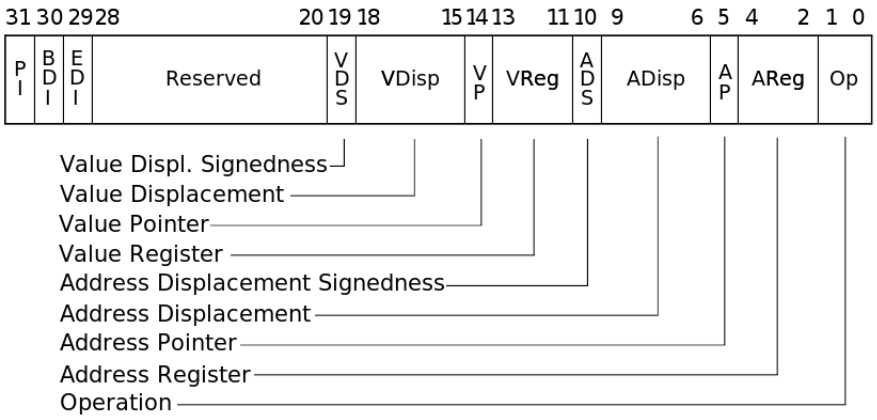

Figure 4.1: Structure of the newly introduced control register CR7.

- **Address Pointer** (bit 5): Specifies whether the *address register* holds the memory address directly (bit 5 cleared) or points to a memory location holding the memory address (bit 5 set).
- **Address Displacement** (bits 9 through 6): An optional displacement added to the memory address. We only allow word-aligned displacement - a displacement of 2 therefore means 8 bytes.
- **Address Displacement Signedness** (bit 10): The sign of the optional address displacement: If this bit is set, the displacement is subtracted from the memory address.
- **Value Register** (bits 13, 12, and 11): The register holding the value to be used for the label manipulation or the register/memory receiving the obtained label value. Register bit representation analogous to *address register*.
- **Value Pointer** (bit 14): Specifies whether the *value register* holds the value directly (bit 14 cleared) or points to a memory location holding the value (bit 14 set).
- **Value Displacement** (bits 18 through 15): An optional displacement added to the value address.

- **Value Displacement Signedness** (bit 19): The sign of the optional value displacement.

For a better understanding of the bit fields described above, the following code snippets show the symbolic code instructions and how they can be represented in c / assembler code:

Clear the protection label of variable `bar`:

```
int bar = 0x0;

/*
 * prot_set %eax, 0x0
 *
 *            =    ------------SDispPValSDispPAddOp
 * 0x00001801 = 0xb00000000000000000001100000000001
 * set protection label of address in %eax to value
 * in %ebx
 */

asm volatile(
    "movl %0, %%eax\n"
    "movl %1, %%ebx\n"
    "movl %2, %%cr7\n"
    :
    : "r" (&bar),
      "i" (0x0),
      "r" (0x00001801)
    : "%eax");
```

Set the most significant bit of the protection label of variable `foo` indirectly using a pointer with displacement:

```
int foo = 0x0;
int *p_foo = &foo;
p_foo += 2;

/*
 * prot_or -0x8(%ebx), 0x80000000
 *
 * i.e.
 *
 * prot_or 0xfffffff8(%ebx), 0x80000000
 *
 *              =    ------------SDispPValSDispPAddMd
 * 0x000034af = 0xb00000000000000000011010010101111
 * OR protection label of address 0xfffffff8(%ebx)
 * with value in %esi
 */

asm volatile(
    "movl %0, %%ebx\n"
    "movl %1, %%esi\n"
    "movl %2, %%cr7\n"
    :
    : "r" (p_foo),
      "i" (0x80000000),
      "r" (0x000034af)
    : "%eax");
```

Clearly, changing an address' protection label requires high granularity protection for the page holding the address' memory. For this reason, the CPU will first check for the required level of protection. If this level is not present, a protection violation will allow the kernel to react appropriately.

Performance measuring extension

Measuring performance of an application or the whole operating system is a non-trivial task, in general. For this purpose, most processor architectures provide a set of instructions to accumulate certain events, like occurrences of page faults, instructions executed, and so forth. Qemu does not provide such a processor extension at the moment, however. But the performance impact introduced by our extended Mondrian memory protection, though not a primary issue in this thesis, is an interesting topic. We thus introduced a set of instructions that allow to monitor and document execution of certain code segments.

Bit 31 in Figure 4.1 enables this new *performance interface.* If a new value, having bit 31 set, is moved into `CR7`, the remaining bits are used as a command to the performance unit. A list of available commands is shown in Table 4.3.

Value	**Bit mask**	**Register included**
`0x8000000`**`0`**	`0x1000 ... 000000`**`00`**	New process created
`0x80000001`	`0x1000 ... 000000`**`01`**	Process closing
`0x8000000`**`2`**	`0x1000 ... 000000`**`10`**	Thread scheduling
`0x8000000`**`3`**	`0x1000 ... 000000`**`11`**	Report output activation

Table 4.3: CR7 values activating the *performance interface.*

Every op-code expects an additional parameter in register `EAX`. This parameter is used as 32-bit key to allow the inspection of multiple processes simultaneously. As soon as the performance unit is activated, it counts

- the number of times the process was scheduled,
- the number of machine code instructions (supervisor and user) executed,
- the number of milliseconds the process was running (measured on the host system),
- TLB misses, i.e. the number of times the memory management unit had to navigate through the page hierarchy to resolve the physical address of a virtual page,
- TLB protection misses, i.e. the number of times the extended memory management unit had to navigate through the protection hierarchy to resolve the protection table entry of a virtual page,
- TLB protection hits, i.e. the number of times the extended memory management unit fetched the protection table entry from the TLB instead of navigating through the protection hierarchy, and
- the number of page faults and protection violations

that occurred during the execution of the process.
As soon as a monitored process stops execution (i.e. the *process closing* notification is triggered), Qemu prints a report to `stdout`. To limit the amount of output, all processes are inspected, but only data whose key has been *activated* through the *report output activation* notification is actually printed.

Debugging extension

Debugging an operating system's kernel is a cumbersome task, because most errors will inevitably lead to a complete system crash. This normally renders debugging output to logfiles or the system's console useless. To accommodate this situation, we have introduced two new *debugging interfaces* using bits 29 and 30 of the control register `CR7` as shown in Figure 4.1.
The *basic debugging interfaces* (BDI) allows to send 29-bit *debugging labels* (bits 28 through 0 of the control register `CR7`) to the CPU. Qemu reacts to such an event and prints the label to `stdout`. This allows the programmer to track the control flow nicely and greatly improves error finding.
The *extended debugging interfaces* (EDI) works similar to the basic interface but allows up to 8 32-bit debugging labels to be passed simultaneously. Instead of passing the label to `CR7` directly, they are taken from the processor's registers, depending on which of the 8 least significant bits of `CR7` is set. Table 4.4 shows the mapping between `CR7` bit-masks and registers used for debugging.

Value	Bit mask	Register
0x2000000**0**	0x00**1**000 ... 000**000**	EAX
0x2000000**1**	0x00**1**000 ... 000**001**	EBX
0x2000000**2**	0x00**1**000 ... 000**010**	ECX
0x2000000**3**	0x00**1**000 ... 000**011**	EDX
0x2000000**4**	0x00**1**000 ... 000**100**	ESP
0x2000000**5**	0x00**1**000 ... 000**101**	EBP
0x2000000**6**	0x00**1**000 ... 000**110**	ESI
0x2000000**7**	0x00**1**000 ... 000**111**	EDI

Table 4.4: CR7 values printing individual register values using the *extended debugging interface.*

4.2 Kernel adaptions

To allow the individual components of our extended architecture to provide the protection hierarchy to the CPU and synchronize executing threads with the protection tables, we had to hook several functions in the operating

system. For this, we used the Linux kernel, since it allows easy inspection and modification of the source code.
Since compiling the whole kernel is a time consuming process, we did not include all of our extensions in the kernel's code directly. We rather decided to introduce a struct holding function pointers and a set of stub-functions that can be used to call these functions. By default, all function pointers dereference into an empty function. This leaves the original kernel intact, introducing only a negligible performance impact.
All code instrumenting our extended Mondrian memory protection was implemented inside a loadable kernel module. On module insertion, all necessary function pointers are redirected into appropriate functions inside the module. Likewise, all pointers are reset to their original value when the module is removed. This allows the code to be tested and exchanged easily without recompiling or restarting the whole system and thus greatly accelerates the development cycle.
Furthermore, the kernel internal structs `task_struct` and `mm_struct` were enlarged by two data pointers to let the protection system store thread- and task-specific data, respectively.

Task creation and destruction

The kernel uses four different routines to handle the creation and destruction of a process or thread: `dup_task_struct` (called by `copy_process` during the `fork` or `vfork` system calls) calls the thread initialization routine for every thread created. A hook inside this function allows the protection module to allocate memory for the thread-specific data pointer stored inside `task_struct`. Likewise, `mm_init` calls the task-specific initialization routine, if the `fork` system call creates a new process (i.e. does not create a sub-thread for the current process).
`free_task` and `mmdrop` call the functions freeing the respective memory structures. Note that despite what the names suggest, `dup_task_struct` and `task_struct` are *thread*-specific, because in Linux every thread is a full standalone task that shares certain memory regions with other tasks. Thus, `mm_struct` and `mm_init` are *process*-specific.

Process execution

The `execve` system call can be used to start new applications. Linux does this by replacing the code section of the currently executing program and resetting all memory regions used by the process. During the call to `execve`, Linux internally calls the thread- and task-creation hooks. This manages the resetting of all memory regions automatically as described in the previous section.
In Section 6.1, we discuss the performance penalty introduced by the extended Mondrian memory protection. To minimize this overhead, the protection hierarchy is maintained but not activated (refer to Section 3.1 for protection activation), by default. During the *process execution* hook in our module, the binary's filename is used to determine if protection should be activated. More precisely, the protection is activated if the filename matches the simple `"^.*/detrace$"` regular expression. This allows us to use the extended Mondrian memory protection for targeted programs only by using a symbolic link named `./detrace` connected with the real application's binary file.
Using the same filename-based activation logic, the kernel uses the process execution hook to *activate* Qemu's performance measuring extension. The location of the thread's `mm_struct` structure is used as key, because it is shared by all threads of a process. Thus, Qemu is able to accumulate the statistics for a complete application.
A last operation done by this hook is to initialize protection of the process' stack by expanding its size from `0` to the stack's default size. Stack expansion will be described in more detail shortly.

Thread scheduling

In the Linux kernel, the function `__switch_to` is used to change the currently executing process. A hook, placed inside this function, allows the module to load the correct protection hierarchy into memory and set control registers `CR5` and `CR6` and the read- and write-masks used for protection examination appropriately. The specific values set depends on what the protection system is currently being used for. In Chapter 5, we describe a few exemplary applications based on our generic framework and deal with how these registers and masks have to be set.
Similar to process execution, the thread scheduling hook notifies the perfor-

mance interface about the thread switch, also.

Page allocation / deallocation

The Linux kernels offers two possibilities for user applications to allocate new memory pages: For one thing, the `mmap` system call can be used to allocate memory regions explicitly. For another, the kernel automatically expands an application's stack if insufficient memory was allocated previously.
Each page that is *mmapped* into the memory context of the running application invokes a function hook inside `do_mmap_pgoff`. The protection module will then insert the new page into the protection directory and protection table, appropriately. The new entry is marked to be using low granularity protection and its protection label is set to the value currently stored in control register `CR5`.
When an application pushes large amounts of data onto stack memory, the stack's base address is likely to exceed the pages that were allocated for this purpose. Since breaching this limit usually accesses an unmapped memory area, the CPU will, in turn, raise a page fault. Linux handles this situation by calling the `expand_stack` function that simply allocates another page for stack memory and continues execution of the application. Similar to `do_mmap_pgoff`, `expand_stack` uses a function hook to call the stack expansion code inside the protection module. The module will add all newly inserted pages between the previous and new stack base address into the protection hierarchy and initialize the protection table's entries.
Alike the explicit allocation of memory pages, the `do_munmap` kernel function contains a hook to notify the protection module, as well. In contradiction to the allocation process, deallocation is a two-step process, however: In a first step, the kernel informs the module that a certain page is about to be removed. This step is necessary because some functions require to have memory pages mapped at a specific address. If this address, however, has previously been mapped by our module to hold protection information of a page using high granularity protection, mapping the address once again either fails or overwrites the stored protection labels.
To accommodate this situation, the protection module *relocates* the protection page by allocating a new page, copying its content to the new location, and adjusting the protection hierarchy accordingly. In the second step, the page is finally taken off the protection hierarchy and its protection page, if present, is freed.

System call interface

Our extended Mondrian memory protection adds a new system call to the Linux kernel. This `sys_prottable`, with signature

```
long sys_prottable(
    int mode,
    int val1,
    int val2,
    unsigned long __user *ret);
```

uses the first parameter `mode` to provide a set of sub-functions using the multiplexer pattern. Amongst other functionality, commonly used during the debugging of our implementation, the system call offers the following functions:

`SYS__NOTIFY_FREE`: Allows to reset protection information of a range of addresses. A reset or *clear* protection label contains a special bitmask that will trigger a protection violation on the next access to the memory. The fault handler, described in the next section, can use this bitmask information and overwrite the protection label with the value currently stored in `CR5`.

`SYS__NOTIFY_TRANSFER_OWNERSHIP`: Changes the protection labels of a range of addresses to a specified value.

`SYS__NOTIFY_COPY_OWNERSHIP`: Copies the protection labels of a range of addresses to another range of addresses.

`SYS__NOTIFY_RANGE_DISABLE_PROTECTION`: Sets a special bitmask to the protection labels of a range of addresses. When the fault handler detects a protection violation on such a *disabled* address, the fault is ignored and access is granted.

`SYS__SET_PROTECTION_ATTRIBUTES`: Allows to activate or deactivate the extended Mondrian memory protection on the currently executing thread or all threads of the currently executing process.

`SYS__NOTIFY_LONGJMP`: This function is used by the stack protection system described in Chapter 5. It works similar to `SYS__NOTIFY_FREE` but only manipulates certain bits of the protection label.

`SYS__GET_PROTECTION_LOG`: This function is used by the race condition detection system described in Chapter 5. It copies certain data about the running process gathered by the kernel into a user provided memory region.

`SYS__NOTIFY_LOCK`: This function is used by the race condition detection system, also. It is used to notify the system about a change of an user-land synchronization object. To allow the notification of kernel synchronization objects, kernel code may use a lock notification hook.

Although some of these functions can be emulated by user code using the protection information manipulation instructions mentioned in Section 3.3, using a system call has two advantages: For one thing, handling protection information using `CR7` manipulation is a non-trivial and error-prone job. The well-defined and tested interface facilitates this to a large extend. For another thing, the user-land code can only manipulate 4 bytes at a time. Besides being a much slower approach, it does not take into account the protection granularity of the page. If, for instance, the protection label of a whole page should be changed and the page uses low granularity protection, the `SYS__NOTIFY_TRANSFER_OWNERSHIP` function can do this by changing the page's protection table entry only.

Protection fault handling

As can be seen in Table 4.2, our extended Mondrian memory protection uses bit 5 of the page fault error code to differential between casual page faults and protection violations. For this reason, the `do_page_fault` function, called by the *interrupt handler*, includes a hook calling the protection fault handler. This protection fault handler can be used to implement various protection schemes. However, every handler includes a base system that extracts information about the pending protection violation. Furthermore, it enforces certain mandatory constraints to ensure stability of the operating system. Figures 4.2 and 4.3 show the flow chart of this handler code. In the remainder of this chapter, we describe this common base system, whereas the individual application-specific fault handlers are described, along with their respective systems, in Chapter 5.

The first constraint, the base fault handler has to ensure, is that the kernel is not running in an atomic context. Code running in such a context may not

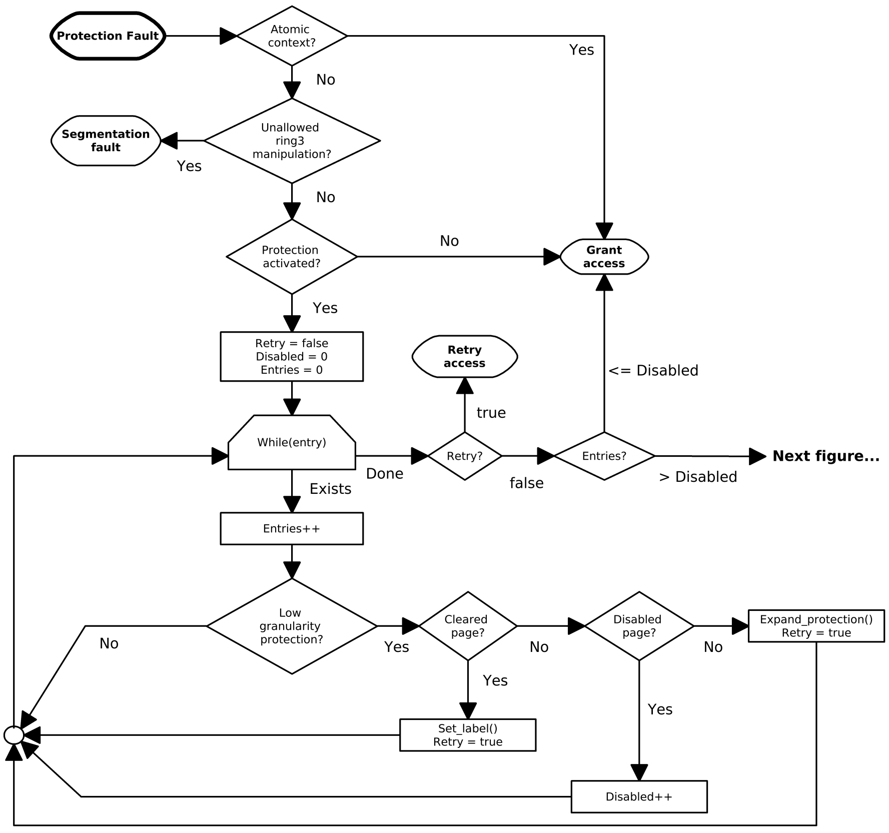

Figure 4.2: Upper half of the common protection fault base handler.

execute any functions that might cause the current thread to be suspended. Regardless of how the application-specific fault handler intends to react to the protection violation, the base handler immediately stops its execution in such a situation. Access to the memory is granted using the read-and-clear CPU flags introduced in Section 3.2. Thus, tough unlikely, it is possible that malicious code accesses protected memory regions while the kernel is running in an atomic context. Although this weakens our implementation of the memory protection architecture to a small degree, we accept this for the sake of maintaining the system's stability easily.
Next, the base fault handler checks if the protection violation is due to an unallowed user code manipulation of a protection label. As described in Section 3.1, the protection module can detain non-supervisor code from using

the protection information manipulation instructions. Furthermore, every *direct* memory access to a protection page by code running in ring3 is handled as well. In both case of illegal access, the base fault handler stops the malicious process by causing a *segmentation fault*, without consulting the application-specific fault handler.

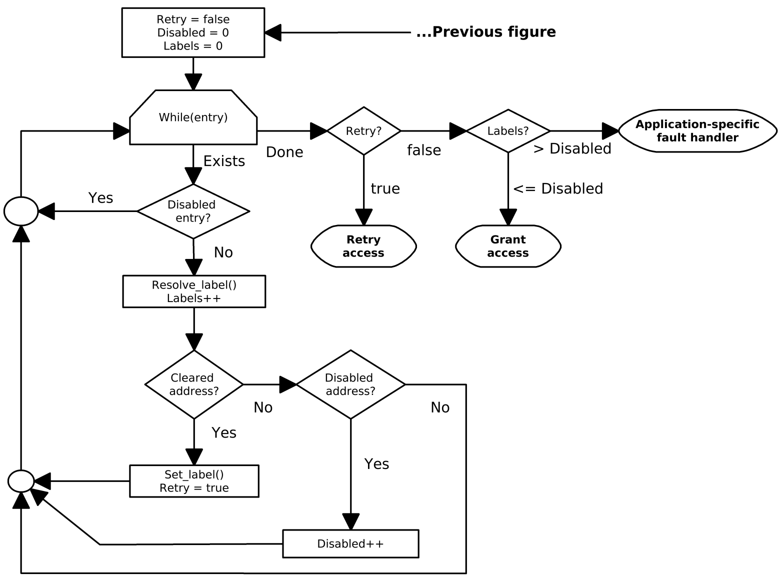

Figure 4.3: Lower half of the common protection fault base handler.

After ensuring that protection for the current thread is activated, the fault handler's base system fetches all protection table entries necessary for the protection examination. In case at least one entry indicates the use of low granularity protection, the system has the following three options:

1. If *any* entry's protection label indicates a *cleared* page, the protection label is set to the value stored in control register `CR5` and the access is *retried.*

2. If *every* entry's label indicates that protection for the whole page should be *disabled*, the access is *granted* in the same manner as described for protection violations in an atomic context.

3. If none of the two previous options match, the low granularity protection of every accessed page is expanded into high granularity. There-

> fore, a protection page is allocated, filled with the entry's protection label, entered into the protection hierarchy, and protected from direct accesses from user code. Eventually, the access is also *retried.*

Otherwise, if every page accessed has already been expanded to high granularity protection, the protection labels are loaded from the appropriate protection pages. The code can identify the number of protection labels necessary by looking at the address and error code provided for the protection violation: By default, only one protection label is resolved. However, both, having bit 6 of the error code set or an unaligned fault address requires an additional label to be examined. The predominant case only requires one single label, the maximum of three labels, on the other hand, is very rare. It can only happen when a protection violation is raised during an unaligned quad-byte access.
For every protection label, options 1 and 2 described above handling *cleared* and *disabled* memory addresses are checked once more. If neither of them can resolve the protection violation, the protection fault base handler has done its job and the application-specific fault handler is called.

Chapter 5

Application of extended Mondrian memory protection

We claim that our extended Mondrian memory protection architecture provides a versatile framework to implement different techniques that allow processes (and threads) to protect sensitive memory regions. These memory regions can be control data (such as return addresses), process management information, or thread-shared data buffers. To demonstrate the versatility of our system, we built three applications on top of the proposed architecture. More precisely, Sections 5.1 and 5.2 show how stack and heap areas can be protected against memory corruption attacks. In Section 5.3, we discuss how the architecture can be leveraged to implement a race condition detection system. While each system application is not novel *per se*, we show how easy each mechanism can be expressed in the context of our protection scheme. This should help the reader understand and appreciate the flexibility and expressiveness of our novel system architecture.

5.1 Stack Protection

The problem of insufficient validation of user-provided input data has been known for a long time. Although many different techniques have been introduced to protect programs against memory corruption, buffer overflow and stack smashing exploits still belong to one of the most popular attack vectors. A possible way to leverage our architecture to protect against a buffer overflow that targets a return address on the stack is to make this address write-

protected. That is, the compiler can use our extended memory protection system to add code to the function prologue that sets the return address as read-only. Thus, when there is a vulnerability inside the body of the function, the attacker cannot overwrite the return address and hijack the control flow of the program. Of course, when the function returns, the memory location on the stack that stores the return address has to be unprotected (i.e., write access has to be enabled again).
In addition to protecting only the function return addresses, we can also add protection boundaries around each local buffer. Such protection boundaries (often called canaries [15]) are realized as write-protected words that are put around each local buffer. As a result, whenever the process attempts to access an out-of-bounds value directly before or after the buffer, the write-protected canary is accessed. This raises a protection fault. Canaries around a local buffer protect against overflows that do not attempt to modify the function return address, but that target another local variable that is adjacent to the exploited buffer.
To add the necessary code that uses our architecture to protect the return address and the local buffers, we have modified the code generation back-end of the `tiny c compiler` [9]. The protection code is quite straightforward. To ensure that a certain memory word (such as the return address or a boundary around a buffer) cannot be modified, we set the most significant bit of its protection label. Moreover, the kernel component sets the most significant bit of the write-mask and clears this bit of the control register `CR5`. Thus, every write access to a canary will lead to a mismatch of the protection and control tokens, causing a protection violation. Likewise, the most significant bits of the canary words are cleared on function exit, restoring the original protection label of the memory addresses.
To see an example for the way in which our protection works, consider the following source code fragment:

```
int function(char arg1, int *arg2)
{
  char *p;
  char buffer[128];
  .....
}
```

The symbolic machine code that is generated for this code can be seen below. Note that the compiler inserts 4-byte canary words before and after the buffer statically allocated on stack. Moreover, the function return address and the saved stack pointer (frame pointer) are write protected.

```
<function>:
  ; function prologue
  push     %ebp
  mov      %esp, %ebp
  sub      $0x8c, %esp

  ; protect function's return address
  ; and saved stack pointer
  prot_or  $0x80000000, %ebp + $0x4
  prot_or  $0x80000000, %ebp

  ; protect canaries around 'buffer'
  prot_or  $0x80000000, %ebp - $0x8
  prot_or  $0x80000000, %ebp - $0x8c

  ; original code of function
  ...

  ; unprotected canaries
  prot_and $0x7fffffff, %ebp - $0x8c
  prot_and $0x7fffffff, %ebp - $0x8

  ; protect function's return address
  ; and saved stack pointer
  prot_and $0x7fffffff, %ebp
  prot_and $0x7fffffff, %ebp + $0x4

  ; function epilogue
  leave
  ret
```

The memory layout on the stack generated by a regular compiler, a compiler that uses StackGuard (with a canary before the function return address), and our proposed technique are shown in Figure 5.1.

tcc stack protection extension

`tcc` is a compiler for the i386 CPU instruction set on *Windows* and *Unix*

Stack memory
...
arg2
arg1
Return addr.
Saved esp
p
buffer[124-127]
...
buffer[4-7]
buffer[0-3]
...

(a)

Stack memory
...
arg2
arg1
Return addr.
Saved esp
Canary word
p
buffer[124-127]
...
buffer[0-3]
...

(b)

Stack memory	Prot. page
...	...
arg2	r/w
arg1	r/w
Return addr.	**read-only**
Saved esp	**read-only**
p	r/w
Canary word	**read-only**
buffer[124-127]	r/w
...	...
buffer[0-3]	r/w
Canary word	**read-only**
...	...

(c)

Figure 5.1: Stack layouts with (a) no, (b) StackGuard [15], and (c) our extended Mondrian memory protection.

systems. Because it is very small[1], it allows easy manipulation and extension of its source code. Furthermore, it is extremely fast[9] when compared with other compilers such as `gcc` [2].
To generate the output binary, the compiler uses a straight-forward one-time pass through the source code: Starting at the top of each input file, the parser generates a tree of *symbols* for a small block of input code and passes it to the code generator. Typically, such a small block consists of a single-line c instruction, a variable allocation, or a function head declaration. The code generator then traverses the generated symbol tree recursively and writes machine code directly to the output buffer.
If the compiler cannot resolve a reference (e.g. to a externally defined variable or function), the memory address for the variable, jump or function call is temporarily filled with a default value and its location in the output buffer is stored in a global *relocation list*. After all input files have been handled, the compiler uses this list to search for previously undefined references and overwrites the temporary values with the correct memory addresses found by the parser.

[1]The whole source, including *lexer*, *parser*, code generator and *linker*, is only about 30000 lines of code.

Besides the necessity for a relocation list, this one-pass approach introduces another difficulty, that is particularly interesting for our stack protection extension: When generating code for a function prologue, the code generator does not have any knowledge about the amount of memory that should be allocated on stack for local variables in the function. This is due to the fact that the function body itself has not been analyzed by the parser. Consider the following, extended code snippet from before:

```
int function(char arg1, int *arg2)
{
  char *p;
  char buffer[128];

  .....
  p = buffer;
  buffer[27] = 'c';
  .....

  .....
  char c = arg1;
  .....
}
```

and the generated function prologue

```
<function>:
  55                    push   %ebp
  89 e5                 mov    %esp,%ebp
  81 ec 88 00 00 00     sub    $0x88,%esp
  .....                 .....
```

When the code for decrementing the stack pointer by `0x88` (128 byte for the array, 4 byte for the pointer, and another 1 + 3 byte for variable `c` and alignment) is generated, the local variable declarations have not been analyzed. Even if the parser continues to analyze the source code until the first non-variable declaration instruction is found before generating the function prologue, the declaration of variable `c` still remains hidden.
For this reason, `tcc` fills the 9 bytes for stack memory allocation with a default value and remembers their location in the output buffer. After the function body has been generated (i.e. during generation of the function epilogue), the instructions for correctly decrementing the stack pointer are

written over the temporary place holder, previously stored at the remembered position.
Dealing with our stack protection, the same problem comes up as well: When the function prologue is generated, the compiler has no knowledge of the number and size of local buffers (i.e. arrays) and cannot know where to put canary words. Thus, we use the approach above and simply increase the size of the temporary place holder to fit the original function prologue *and* our protection code.
Looking at binaries generated by a first implementation of our compiler extension, it was salient that the code areas for protecting and unprotecting canaries were identical to a big extend. To eliminate this redundancy, we removed the second code for canary manipulation and inserted an intelligent jump instruction instead. The new prologue and epilogue for the extended function provided above can be seen in the following binary / assembler section. Since the source is rather tricky and looking at the whole function at once is rather confusing, we divide it into logical subsections and explain each part individually.

```
 1 <function>:
 2   55                      push   %ebp
 3   89 e5                   mov    %esp,%ebp
 4
 5   ; protect location of return address and
 6   ; saved stack pointer
 7   50                      push   %eax
 8   53                      push   %ebx
 9   b8 00 00 00 80          mov    $0x80000000,%eax
10   bb 17 00 00 00          mov    $0x17,%ebx
11   0f 22 fb                mov    %ebx,%cr7
12   bb 57 00 00 00          mov    $0x57,%ebx
13   0f 22 fb                mov    %ebx,%cr7
14   5b                      pop    %ebx
15   58                      pop    %eax
```

As described in Section 4.1, control register `CR7` can be used to manipulate protection labels. Referring to Figure 4.1, the values indirectly set to `CR7` through register `EBX` in lines 11 and 13 can be dismantled to the following symbolic instructions:

		Value Register	Disp. Sign	Addr. Disp.	Addr. Ptr.	Address Register	Operation
0x17	=	0b000	0	0000	0	101	11
		EAX	+	*none*	*no*	EBP	prot_or
0x57	=	0b000	0	0001	0	101	11
		EAX	+	4 byte	*no*	EBP	prot_or

Register EBP contains the stack base address, after the return address and saved stack pointer have been pushed. Therefore, protecting offsets 4 and 0 of this register refers to these to values, respectively. Since we use registers EAX and EBX to hold temporary values, both registers are saved on the stack to be restored after the CR7 manipulation.
After protecting the function return information, ESP is decremented to allocate space for local variables and canary words and the control flow jumps to the canary protection code:

```
  81 ec 90 00 00 00        sub    $0x90,%esp
  e9 41 00 00 00           jmp    function_protect

function_body:
  .....                    .....

  ; function body

  .....                    .....

  eb 0e                    jmp    function_unprotect

function_protect:
  50                       push   %eax
  53                       push   %ebx
  b8 00 00 00 80           mov    $0x80000000,%eax
  bb 17 00 00 00           mov    $0x17,%ebx
  eb 0c                    jmp    function_protection

function_unprotect:
  50                       push   %eax
  53                       push   %ebx
```

```
   b8 ff ff ff 7f          mov    $0x7fffffff,%eax
   bb 16 00 00 00          mov    $0x16,%ebx
```

The two code sections at the end of this snippet set up the operation and value used for the next CR7 manipulation. When called from the function prologue, everything is set up as in the previous snippet. Otherwise, register EAX clears the protection label's most significant bit as the following dismantling shows:

		Value Register	Disp. Sign	Addr. Disp.	Addr. Ptr.	Address Register	Operation
0x16	=	0b000	0	0000	0	101	10
		EAX	+	*none*	*no*	EBP	prot_and

```
function_protection:
   ; set protection label for canary below buffer
   81 ed 8c 00 00 00       sub    $0x8c,%ebp
   0f 22 fb                mov    %ebx,%cr7
   81 c5 8c 00 00 00       add    $0x8c,%ebp

   ; set protection label for canary above buffer
   81 ed 08 00 00 00       sub    $0x8,%ebp
   0f 22 fb                mov    %ebx,%cr7
   81 c5 08 00 00 00       add    $0x8,%ebp
```

For every canary word that was inserted by the compiler, this code uses registers EBP, EAX, and EBX to change the read-only state of the protection information appropriately.

```
   ; decide if we are unprotecting or protecting
   ;   protecting  : jump back to function body
   ;   unprotecting: jump to function epilogue
   0f ba e0 1f             bt     $0x1f,%eax
   5b                      pop    %ebx
   58                      pop    %eax
   73 05                   jae    function_epilogue
   e9 7a ff ff ff          jmp    function_body
```

Eventually, the value of EAX is used to determine if the code was called during the function prologue or epilogue. In the first case, the control flow jumps back to the beginning of the function and executes the function's body. Otherwise, the remaining epilogue is called:

```
function_epilogue:
  ; unprotect location of return address and
  ; saved stack pointer
  50                        push   %eax
  53                        push   %ebx
  b8 ff ff ff 7f            mov    $0x7fffffff,%eax
  bb 16 00 00 00            mov    $0x16,%ebx
  0f 22 fb                  mov    %ebx,%cr7
  bb 56 00 00 00            mov    $0x56,%ebx
  0f 22 fb                  mov    %ebx,%cr7
  5b                        pop    %ebx
  58                        pop    %eax

  c9                        leave
  c3                        ret
```

Before the function exists, saved stack pointer and function return address have to be unprotected. The values assigned to `CR7` should be self-explanatory by now: `0x16` was already used previously and `0x56` also unprotects the memory address in register `EBP`, but with an offset of 4.

Protection fault handler

As mentioned in the beginning of this section, extending the base protection fault handler is straight-forward: If the stack protection-specific handler is called, only the most significant bit of the protection label has to be examined. If this bit is set during a write access, a read-only canary or protected function return information is overwritten. In such a situation, the fault handler terminates the current process using a segmentation fault.
Furthermore, as described above already, bit 31 of control register `CR5` must be cleared and of the write-mask be set for every application that uses the stack protection.

Non-local control flow modification

Some programs rely on non-local control flow modifications. That is, at some point of execution, all registers, including the current instruction pointer and the stack base pointer, are saved to memory. Later, this *snapshot* can be used to resume the program exactly like it was done after taking the snapshot.

The GNU C Library [3], for example, provides such functionality through the `setjmp` / `longjmp` function pair.
When doing non-local control flow modifications, the memory addresses protected on stack since taking the snapshot are not unprotected. This is because the functions called between taking and resuming the snapshot do not return anymore. For this reason, the GNU C Library has been altered to handle this problem: Whenever a snapshot is resumed, all stack addresses between the current and restored stack base pointer are unprotected automatically. This leaves required protection information untouched but clears the freed stack range.
The system call functionality `SYS__NOTIFY_LONGJMP` provides an interface for doing this task. The only change that had to be done to the C Library was thus to include the system call and setting the function parameter (the stack range) appropriately.

5.2 Heap Protection

Similar to the problem of smashing stack buffers, heap buffer overruns have gained attention over the last few years. Although usually more complicated than stack smashing, it is possible to change a program's control flow by modifying the content of certain data structures stored on the heap.
Doug Lea's Malloc [25], the memory allocator the GNU C Library implementation is based on, uses in-band management information to maintain currently allocated chunks of memory. If data is copied into an allocated buffer without checking its length properly, it is possible to overwrite the management information of an adjacent chunk, possibly causing a memory corruption. This memory corruption can be leveraged to eventually overwrite control data, hijacking the program's control flow.

```
struct malloc_chunk {
  size_t      prev_size;
  size_t      size;

  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  .....
};
```

Malloc stores free chunks in circular doubly-linked lists. Figure 5.2 and the

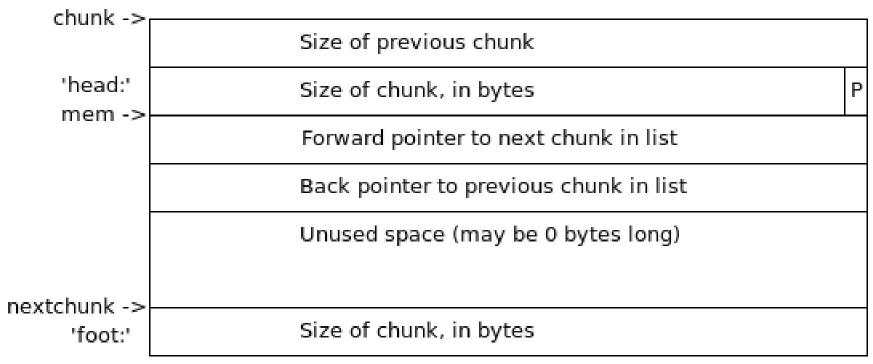
chunk ->
'head:'
mem ->
nextchunk ->
'foot:'
Size of previous chunk
Size of chunk, in bytes
P
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk, in bytes

Figure 5.2: In-band information of a free chunk [3].

code snippet above show the structure of such a free chunk. When the user application wants to allocate a memory region, these lists are searched for an item that fits the requested size best. Using the `unlink` macro (simplified) shown below, malloc takes the chosen chunk off its list and returns its `head` memory address to the application.

```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {
  FD = P->fd;
  BK = P->bk;
  FD->bk = BK;
  BK->fd = FD;
}
```

If a buffer overrun of an adjacent allocated chunk previously changed the free chunk's `fd` and `bk` pointers, dereferencing `fd` allows an attacker to write arbitrary four bytes of data to a location of his/her choosing. Using our memory protection architecture, we have introduced a mechanism for preventing heap based buffer overruns. As the heap management information must only be modified from within the allocation code inside the GNU C Library, we can keep the memory management information write-protected while the user code is executing. Only when heap management data needs to be modified, the read-only memory locations are unprotected. Once the management information is updated, the C library can write-protect this data again. When an attacker later exploits a vulnerability and attempts to overwrite this data, a protection fault is raised.

Malloc heap protection extension

Inside the `malloc_chunk` structure, the member variable `size` is the first memory location that must not be tampered with by code outside the library. Thus, we use this as a read-only border to protect the chunk's memory information. The GNU C Library uses various functions that interact with the lists of free chunks, including

- `malloc`: Allocates a memory area and must thus be able to take chunks off the lists,
- `realloc`: Changes a chunk's size. If this cannot be done directly (because the memory behind the chunk is already in use), the allocation of a new chunk and copying the current content to the new buffer is necessary. Thus, this function must also interact with the lists of free chunks, and
- `free`: Malloc does not allow two adjacent chunks to be free. Thus, whenever a chunk adjacent to a free chunk is freed, the two chunks are merged, changing the first `size` variable accordingly.

To limit the changes done to the original source to a minimum, we have introduced a macro that can be used instead of manipulating the protected variable directly. An exemplary use of this macro is shown in the code snippet below:

```
Void_t*
_int_malloc(mstate av, size_t bytes)
{
  .....
  /* inspected/selected chunk */
  mchunkptr         victim;
  .....

  // victim->size |= NON_MAIN_ARENA;
  alter_protected_size(victim, |= NON_MAIN_ARENA);
  .....
```

The `alter_protected_size` macro is realized very similar to the canary protection code explained in the previous section. Its exact implementation can be seen in the following source listing:

```
#define unprotect_address(__p)
{
  asm volatile(
    "push   %%eax\n"
    "push   %%ebx\n"
    "push   %%ecx\n"
    "mov    %0,%%ecx\n"
    "mov    $0x7fffffff,%%eax\n"
    "mov    $0x6,%%ebx\n"
    "mov    %%ebx,%%cr7\n"
    "pop    %%ecx\n"
    "pop    %%ebx\n"
    "pop    %%eax\n"
  ::"r"(__p));
}

#define protect_address(__p)
{
  asm volatile(
    "push   %%eax\n"
    "push   %%ebx\n"
    "push   %%ecx\n"
    "mov    %0,%%ecx\n"
    "mov    $0x80000000,%%eax\n"
    "mov    $0x7,%%ebx\n"
    "mov    %%ebx,%%cr7\n"
    "pop    %%ecx\n"
    "pop    %%ebx\n"
    "pop    %%eax\n"
  ::"r"(__p));
}

#define alter_protected_size(__p, __code)
{
  unprotect_address(&(__p ->size));
  __p ->size __code;
  protect_address(&(__p ->size));
}
```

Protection fault handler

Since our heap protection works exactly like the previously described stack protection, they share a common protection handler. For further information on the inner workings of this handler, refer to Section 5.1.

5.3 Race Condition Detection

To show a third application for leveraging our extended Mondrian memory protection architecture, we have made our own implementation of the race condition detection algorithm described in [29]. In this section, we describe the original detection algorithm and how our implementation differs from that. Section 6.3 then provides an overview of applications tested with our system, as well as of actual race condition bugs that we found.

In [29], the author describes a data race (condition) as follows:

> A `lock` is a simple synchronization object used for mutual exclusion; it is either available, or owned by a thread. The operations on a lock `mu` are `lock(mu)` and `unlock(mu)`.
>
> A `data race` occurs when two concurrent threads access a shared variable and when
>
> - At least one access is a write, and
> - the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Eraser race detection algorithm

In order to be able to detect possible race conditions in a program, Eraser [29] uses four bytes of `shadow memory` for each memory word in the application's address space. As long as a memory address has been accessed by a single thread only (identified by its PID[2]), this memory address is `owned exclusively` by this thread. To indicate this fact, the shadow memory contains the owner's PID.

[2]In Unix-based operating systems, every thread has a system wide unique ID, called *process identification number*. If multiple streams of execution share a common memory context, they are called threads of a process, indicated by a common *thread group ID*.

Eraser employs a binary rewriting tool that patches every memory access with a call to the race condition checking routine. That is, whenever memory is read or modified, the routine compares the shadow memory's content and the process' PID. As soon as a thread accesses a memory location whose shadow memory contains a different PID, the detection algorithm knows that the data at this memory location is shared. Thus, the first requirement for a data race stated above is met. To test whether there is a real data race, the second requirement needs to be checked as well. To this end, the system employs the lock-set algorithm:

As part of the lock-set algorithm, the system instruments all calls to synchronization procedures to notify the detection system about changes of each thread's currently held locks. This allows the system to determine the set of locks that a thread holds at any point in time. Also, the semantics of the shadow memory is different for shared memory regions. Instead of the owner's PID, the shadow memory of each shared memory location contains two status bits[3] and an ID that tells the detection algorithm which set of locks have been held previously by threads accessing this memory location. This set is called the lock-set for this location. Initially, at the moment when a memory location is marked as shared, the memory's lock-set is set to the locks held by the accessing thread.

On every access to a shared memory location, the lock-set set is recalculated by intersecting the set of locks *currently* held by the running thread with the set identified by the ID in the shadow memory. If the intersection obtained through this *lock refinement* ever yields an empty set, the detection algorithm has found a data race and can issue a warning.

Figure 5.3 shows the state graph a memory location traverses before the warning is issued: Initially, every address is in *virgin* state, indicating that no thread has yet accessed the memory. Once the first thread accesses the location, it changes to state *exclusive*. As soon as a new thread accesses the location, it changes to states *shared* or *shared modified*, depending on whether the access is a read or a write. In these two last states, the lock-set refinement has to be done on every access to the memory address.

Eraser distinguishes between the two last states to support the use of read-/write locks: Programmers often use single-writer, multiple-reader locks to

[3]The status bits indicate that the memory location should be treated as shared and keep track of whether there has been a write access to the memory location since it has been marked as such.

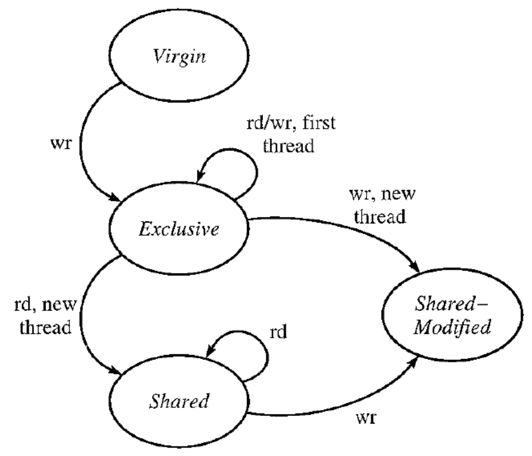

Figure 5.3: Race condition detection state machine [29].

allow one thread to generate data and make it available to others. To support this programming paradigm, the lock-set refinement has to be extended: On every *write* access to a shared memory location, the lock-set set is recalculated by intersecting the set of locks *currently* held in *write mode* by the running thread with the set identified by the ID in the shadow memory. For *read* accesses, the refinement remains unchanged and the set is intersected with the locks held in *any* mode. Furthermore, race condition warnings for memory addresses in the *shared* state are postponed until their state changes to *shared modified.*

Extended Mondrian detection algorithm

To implement the lock-set algorithm on top of our architecture, we require a mechanism to detect the case in which multiple threads access the same memory area. Moreover, we require a way to represent the locks that a thread currently holds. Finally, it is necessary to have a representation for the lock-sets that store, for each shared memory region, the set of locks that were held while accessing this region.
We use an approach that is similar to the original system, but instead of a shadow memory, we use the 30-bit protection labels to hold the shadow memory's content. When using the race condition detection module, the

kernel puts the 16 bits of the current *thread ID*[4] into bits 3 through 19 of the control register `CR5`[5], clearing all other bits. Likewise, bits 3 through 19 of the read- and write-masks are also set. This ensures that a protection violation occurs whenever a process accesses a memory address that it does not own exclusively.
When the kernel protection fault handler identifies a shared memory access, it marks the target of this memory address as shared. This is achieved by setting bit 30 of the protection label to 1. By also setting this bit in both access bit-masks, every further access to that memory location will trigger a protection fault. This allows the protection fault handler to compare and update the set of locks held during the memory access. When a memory region is marked as shared, the 28 least significant bits can be used to store the lock-set ID (which indicates the locks that were held while accessing this memory region). Similar to the original algorithm, bit 29 is used to differentiate between the *shared* and *shared modified* states. Bit 31 is left unused to allow compatibility with the previously explained stack and heap protection mechanisms.
The following example shows two threads accessing a common memory location `addr`. We assume that `addr` is initially owned exclusively by a thread with a *thread ID* (TID) of 1. Thus, the protection label stores this TID (recall that the first three least significant bits are unused). It can be seen how the accesses to this variable change the memory's protection information until a race condition is detected. The race is detected when the first thread writes to the shared memory region while holding lock `m1`. The reason is that, previously, the same memory was accessed by Thread 2 using only lock `m2`. Thus, the lock-set is empty, indicating a bug.

[4]Our detection algorithm uses the difference between *process ID* and *thread group ID* instead of using the PID directly. The first, main thread of a process thus has a thread ID of 0. This brings the advantage that when this thread is forked, the protection labels can be copied directly into the new process and need not be modified to match the new PID.

[5]Bits 0 and 1 of control register CR5 are reserved and may not be used. Bit 2 is not used for historical reasons.

CR5	**Protection label of `addr`**	**Locks held**	**Thread 1 (TID 1)**	**Thread 2 (TID 2)**
	`0b0000...001000`	$\{\} = ID(0)$		
`0b01000`			`lock(m1);`	
		$\{m1\} = ID(1)$		
..			`write(addr);`	
..			`unlock(m1);`	
		$\{\} = ID(0)$		
`0b10000`				`lock(m2);`
		$\{m2\} = ID(2)$		
..				`read(addr);`
	`0b0110...000010`			
..				`unlock(m2);`
		$\{\} = ID(0)$		
`0b01000`			`lock(m1);`	
		$\{m1\} = ID(1)$		
..			`write(addr);`	
	`0b0100...000000`			

Lock-set acquisition

Of course, we require a mechanism to identify which locks are currently held by a process. To this end, we insert hooks into the operating system kernel that get notified when a kernel semaphore is locked or unlocked. In addition, the extended system call interface mentioned in Section 4.2 is used to get notifications about user-land locking operations.
Looking at various Unix applications, we found that a vast majority of programs uses the broad range of synchronization functions provided by the GNU C Library. For this reason, we patched the library's

- *mutex*,
- *semaphore*,
- *read/write lock*, and
- *spinlock*

functions to include a call to the new kernel interface. All other means for mutual exclusion offered by the library are included inherently, because they internally rely on one of the basic functions above.
To identify a lock inside the detection system, every synchronization object (regardless of kernel or user-land) passed to the system interface requires an application-unique key. To achieve this, we used the object's virtual address. While this works well for casual mutexes and semaphores, it does not for read/write locks. Holding such a lock in read-only mode must be differentiable from those locks held in read-write mode to allow correct lock-set refinement. Since the structs storing the locks inside the C Library are guaranteed to be word-aligned, however, we can achieve this by simply passing the lock's ID increased by 1 or 2, depending on the function the call is made from.
Whenever a call to the locking interface is made, the race detection-specific protection module generates a new ID representing the lock-sets held by the current thread. This information is stored inside the thread-specific data buffer referenced by the pointer inside the `task_struct` mentioned in Section 4.2. For read/write locks in read-only mode, only the thread's *read-lock ID* is modified, for all other types of locks, the *read-* and *write-lock IDs* are updated.

Lock-set identification

As described in the previous section, lock-sets associated with memory addresses as well as those held by the individual threads are represented using an identifier unique within all threads of a process. For this purpose, the process-specific data buffer, referenced through `mm_struct`, includes a table of singly-linked lists of lock-sets. The index into the table, where the lock is stored, is used as ID for the lock-set algorithm. Indices 0 and 1 have predefined meanings representing the *empty* lock-set and the lock-set holding *all* possible locks (as the system cannot know which/how many locks will be used by the application, this lock-set cannot be represented by a casual list). Additionally, index 2 represents the *empty* set for addresses that did not trigger a race condition warning previously. This situation is also representable with lock-set ID 0 while having bits 29 and 30 of the protection label set (i.e. the address is in state *shared* but not *shared modified*). Since this extra index was quite nice for debugging during the first implementations of the race detection system, however, it retained its meaning into the final

implementation.
When a lock is added to or removed from a lock-set, the protection module uses the lock-set ID to find the associated singly-linked list of locks in the lock-set table. This list is then duplicated and the lock is inserted or taken out of the newly generated list. To retrieve the ID of this modified set, the module tries to find a clone of this set inside the lock-set table. If one is found, this combination of locks has been used previously. Thus, the newly generated set is deleted and the previous ID found is used. If no such lock combination is found in the table, however, the lock-set's head element is stored in the first free table position and its index is used.
Since each modification of a lock-set requires the comparison of several singly-linked lists, we have made three attempts to increase performance: For one thing, we have changed the singly-linked lists to be sorted by ascending lock IDs. This introduced an additional runtime overhead when inserting a lock, because the lock's position has to be found inside the lock-set prior to the insertion. Deleting a lock from a set and comparing two lock-sets, on the other hand, can be performed much faster in the average case, legitimating the additional overhead.
The second potential improvement we introduced was a lock-set *cache*. We figured that a small set of lock-sets is likely to be used in the majority of cases. That means, searching the table for a lock-set often yields the same index. Thus, we have added a cache holding the results of recent search operations. Whenever the module looks for a particular lock-set inside the table, the cached IDs are used for comparison first. If no result is found using these IDs, the remaining table is searched. In either case, the returned ID is pushed on top of the cache, removing the ID's previous position or the last item in cache. Measuring average runtimes of the system before and after the improvement did not show any significant changes. The performance increase by possibly avoiding to search the whole table for a lock-set was evened out by the extra overhead necessary for maintaining the cache. Since the system's performance did not worsen and we figure it might still have a positive impact on some applications, we decided to keep the lock-set cache as part of the race detection-specific protection module.
A last modification we experimented with was introducing a hash value to fingerprint a lock-set. That is, the head element of each singly-linked list additionally contains a hashed value of all lock IDs contained in the lock-set. Whenever two lock-sets are compared, their individual hash values are inspected first. If they differ, the two lock-sets cannot match. Otherwise,

the lock-sets have to be compared in the traditional manner to eliminate a possible *hash collision*. Sadly, this attempt did not produce the expected results. Quite contrary, the average runtimes changed for the worse. Experiments showed that this was due to a insufficient hash function that produced an overwhelming number of hash collisions. However, the complexity of an acceptable hash function and therefrom resulting overhead made us abandon the idea.

Protection fault handler

Once the race detection-specific fault handler is called by the base system, the accessed protection labels have all been resolved, *disabled* labels have been discarded and *cleared* labels have been assigned to the current thread. Furthermore, all accessed addresses have been expanded to high granularity protection. Thus, the fault handler only has to mark the addresses to be shared and do the lock-sets refinement:
For this, the code first inspects bit 30 of the protection label. If this bit is set, the address has already been marked as shared. Otherwise, the memory address is initialized to state *shared* (i.e. bits 30 and 29 are set) and ID 1 is entered as initial lock-set. Furthermore, if the protection violation occurred during a write access, the state is changed to *shared modified* (i.e. bit 29 is cleared).
In a next step, the code checks if the memory's lock-set has already been refined to the empty set. In this case, a warning has already been issued and the fault handler continues with the next label. Otherwise, the corresponding lock-set is looked up in the lock-set table and intersected with one of the two lock-sets of the current thread. As mentioned previously, the lock-set used for the intersection is the set of locks held in write mode during a write access or that of locks held in any mode otherwise. If the intersection yields the empty set and the address is in state *shared modified*, a race condition warning is issued. In either case, the new ID is saved in the protection label. Eventually, after all labels have been examined, the fault handler grants access to the memory address using the read-and-clear CPU flags mentioned in Section 4.1.

All user-land, as well as kernel synchronization functions internally rely on some mechanism to modify a memory location atomically. Normally, such a mechanism is provided by the CPU's instruction set directly. For instance, the GNU C Library's code for mutual exclusion on i386 architectures is based on Intel's x86 `lock` prefix. Consider the following code, taken from the library's code for read/write locks:

```
     /* Get the lock held in %ebx. */
     movl  $1, %edx
LOCK cmpxchgl %edx, (%ebx)

     jnz 1f
     .....
   1:
```

Using the `LOCK cmpxchgl` instruction combination, it is possible to compare a memory address with a register and, depending on the result of the comparison, modify one of the two in a single instruction. Furthermore, because the `LOCK` prefix is used, this instruction happens atomically, ensuring that no other thread (running on a second CPU) can access the memory location at the same time.
The race detection system described so far is not capable of handling this common approach for guaranteeing atomicy. For this reason, the protection violation error code has been extended (as already indicated in Table 4.2). When raising a protection fault, Qemu sets bit 8 of the error code in case the executing instruction was prefixed by a `lock` instruction. The fault handler then uses this extra information and extends the thread's lock-set with an additional lock before doing the refinement. Finally, before the fault handler returns, the thread's lock-set is restored to its previous, original state.

Memory reuse

We have not dealt with one problem of the race condition detection system yet that can introduce a big amount of false positives: When a common memory buffer is used temporarily by two different threads without interference of the other thread, the detection system still erroneously issues a race condition warning. Consider the following function:

```
void function()
{
  char *buf = (char*)malloc(0x100);
  memset(buf, 0x0, 0x100);
  .....
```

```
    free(buf);
}
```

If this function is called by two independent, *non-concurrent* threads, it is quite likely that the second thread will reference the same memory address as the first thread did. During its first execution, the function implicitly set all protection labels of buffer `buf` to its thread ID. Thus, when the second thread tries to write to `buf`, the corresponding protection labels will mark the addresses as shared. Although there is no race condition present, the protection fault handler will still issue a race condition warning. This is because no explicit locking is used when accessing the buffer previously owned by another thread.

The same problem arises when multiple threads are run successively without providing specific stack memory. The Native Posix Thread Library [19] (NPTL), the thread management library included in the GNU C Library, will most likely use the same memory area as stack for every single thread. Again, because the first thread will get exclusive ownership of the stack memory, every access by a subsequent thread will issue a race condition warning. To circumvent these situations, every allocation function in the GNU C Library has been altered to use the system call interface `SYS__NOTIFY_FREE`. Before a buffer is returned to the user application, the library *clears* the buffer's protection information. This allows the protection fault handler to give each thread exclusive ownership of the buffer during the first access to its memory.

Detection overhead

When comparing our system to Eraser, we note that our memory footprint is much smaller: While Eraser incurs 100% memory overhead (every word is described by 4 bytes of shadow memory), our protection system uses different granularity levels for different memory areas. Although shared memory pages require the same amount of extra memory, we can use low granularity protection on memory areas such as the threads' stacks, read-only data sections, and code mappings, reducing the overhead drastically.

`detrace`: A tool for automatic race detection

To ease the process of finding race conditions in an application, we have created a small tool, automating this process. `detrace` (*det*ect *race*conditions)

is a `perl` program that allows to interact with the protection module in a very simple manner and provides an easily understandable output of all generated data.
When `detrace` is started, it executes the program provided as argument. By changing the `LD_PRELOAD` environment variable, the patched version on the GNU C Library is dynamically loaded into the program instead of the default library installed on the system. This allows the user to test completely unmodified applications without having to install the altered version of the library.
`detrace` waits for the termination of the provided binary (it also allows to directly kill the executed program at any time) and then fetches all data collected by the protection module through the module's system call interface `SYS__GET_PROTECTION_LOG`. For every race condition warning that was issued, the tool tries to extract information on the source code corresponding to the warning and symbols associated with the accessed memory location.

A typical invocation of the tool can be seen below:

```
$ detrace.pl ./program program-argument

+-----------------------------------------------+
| Welcome to detrace 0.1.10 (2008-01-31)
+-----------------------------------------------+

INFO: Writing all output to directory ./output/
   run_DeGPY
INFO: Executing binary './program program-argument'
INFO: PID=3557
INFO: Binary './program program-argument' terminated
INFO: Waiting for binary to finish execution...
INFO: Binary finished!
INFO: Making sure binary terminated...
INFO: Retrieving protection information from binary
INFO: Retrieving symbol information from binary
INFO: Retrieving dynamic relocation information from
    binary
INFO: Extracting dynamic library mapping information
INFO: Extracting race information..................
    ......done
INFO: Extracting lock information
```

```
INFO: Extracting inspection information done
INFO: Extracting lockset information
INFO: detrace run finished successfully
INFO: Have a nice day ;-)
```

Eventually, `detrace` splits the gathered information into multiple files. For instance, all race conditions for addresses in heap memory, issued by the program's code directly (that is, not issued by a dynamically linked library) is gathered in a single file:

```
$ cat ./output/run_DeGPY/races_heap_main

  address: 0x08049140
  thread : 0x2
  process: 0xe62
  eip    : 0x8048a2d file mutex.c, line 213.
  symbol : mutex1+0x4
--
  address: 0x08049170
  thread : 0x2
  process: 0xe62
  eip    : 0x8048a37 file mutex.c, line 215.
  symbol : unprotected_counter
--
  address: 0x08049174
  thread : 0x3
  process: 0xe62
  eip    : 0x80489f4 file mutex.c, line 202.
  symbol : incorrectly_protected_counter
```

Chapter 6

Evaluation

This chapter provides details on the performance and memory overhead introduced by our extended Mondrian memory protection. We also discuss the effectiveness of the previously introduced system applications, in particular, the race detector.

6.1 Performance

Although our implementation did not focus on performance issues primarily, we have attempted to estimate the performance penalty factor introduced. Table 6.1 shows averaged results of measuring ten executions of the following three applications:

1. **gpg2**: Encrypting a binary file with a symmetric key using `gpg2` [5] demonstrates usage of heap memory. For this application, we were unfortunately unable to measure the performance impact introduced by the stack protection mechanism, as the compiler chosen (`tcc`) was not compatible with the `gpg2`'s source code.

2. **sudoku**: A straight forward implementation[1] of a solver for the popular puzzle. This program does not make use of data allocated in heap memory, but uses the stack extensively due to its recursive design.

3. **ClamAV daemon**: An open source (GPL) anti-virus toolkit for UNIX [1]. For better performance on multi-processor systems, the toolkit's dae-

[1]For details, refer to `http://pubpages.unh.edu/~pas/hacks/sudoku/`

mon allows to scan files or directories concurrently using multiple threads. Since it is very handy to measure the daemon's response times, we used this program to measure the impact of activating the race condition detection system.

Analyzing the results in Table 6.1, we can see that the run-time penalties for using the stack and the heap protection are relatively small, making it suitable for deployment in production systems. The overhead for the race detector seems excessive at first glance. However, these numbers are in the same range as for the original Eraser [29] system. Moreover, the race detector is targeted for the testing phase of applications, prior to their deployment. In this phase, even a significant performance penalty can be easily tolerated when the system is able to identify hard-to-detect errors.

Table 6.1: Performance penalties introduced by the extended Mondrian memory protection. The values show the relative increases compared to the original system (and thus, include also the overhead introduced by the Mondrian memory system). Page and protection fault values are given in absolute numbers.

Mode	**Exec. Time**	**Instructions Executed** (ring0 / ring3)	**TLB Miss.**	**Page Faults** (abs.)	**Prot. Faults** (abs.)
gpg2:					
Original system	1.000	1.000 (1.000 / 1.000)	1.000	319	0
MMP present	1.058	1.004 (1.134 / 1.000)	1.090	319	0
Heap protected	1.478	1.011 (1.383 / 1.000)	1.104	344	13
sudoku:					
Original system	1.000	1.000 (1.000 / 1.000)	1.000	109	0
MMP present	1.069	1.011 (2.230 / 1.000)	1.091	115	0
Heap protected	1.248	1.015 (2.604 / 1.000)	1.095	115	1
Stack protected	1.438	1.018 (2.418 / 1.000)	1.103	111	1
Stack & Heap p.	1.487	1.022 (2.701 / 1.000)	1.300	129	2
ClamAV daemon:					
Original system	1.000	1.000 (1.000 / 1.000)	1.000	263	0
MMP present	1.107	1.042 (1.071 / 1.000)	1.441	273	0
Race detection	37.464	23.366 (134.0 / 1.166)	71.596	342	143.627

6.2 Memory Overhead

As mentioned in previous sections, the additional memory required for storing protection labels depends on the granularity level chosen. For minimal and low granularity protection, only the additional protection hierarchy has to be stored, occupying the same amount of memory as the memory necessary to store the page directory and the page tables.
Since stack- and heap-protection as well as the race condition detection system all require high granularity protection, their memory requirements can become significantly large. To keep the overhead as small as possible, all pages are protected using the low granularity level by default. Only when finer-grain protection is required for a certain address, the system switches to high granularity protection *for this page only.*
The lazy expansion of protection pages considerably reduces the memory overhead for read-only data and code areas, heap memory (if no heap protection is active), stack memory (if no stack protection is active), as well as large memory mappings spanning more than one page.
To get a feeling for the memory overhead that can be expected in practice, we measured the additional pages (with a size of 4 KB) that our system required during the experiments to store the necessary protection information. The heap protection for `gpg2` required 13 additional pages. For the stack and the heap protection for the `sudoku` application, the system needed one additional page each. For storing the race detection information, 101 additional pages were necessary. Thus, in all cases, the overhead incurred was less than 500 KB.

6.3 Effectiveness of System Applications

For the stack and heap protection, we developed a number of small applications that contained vulnerabilities that would allow an attacker to launch different attacks to corrupt stack and heap memory. As expected, all exploits that modified write-protected data structures were correctly identified. Thus, for the reminder of this section, we focus in more detail on the effectiveness of the race condition detector.
To test the effectiveness of the race condition detection system, we have examined a number of large, multi-threaded applications:

- hand-crafted applications to test the GNU C Library's locking operations as well as heap-allocation and thread management code,
- several chat server implementations (such as OpenNaken or chat1d),
- a small multi-user game server (Space Tyrant Game Universe),
- ClamAV's scan daemon, and
- the Apache web server [4].

As the detection system is a dynamic analysis tool, only those code regions that are actually executed are examined. Thus, we cannot guarantee the absence of race conditions for a complete application. However, on the positive side, each warning is a strong indication of an actual error because the potential race condition was produced by an *actual* program run.
In the following subsections, we discuss in more detail a subset of the race condition errors that we found during our experiments (and that we believe are most interesting):

GNU C Library, mutex locking

We developed a number of small applications to test the individual locking strategies offered by the Linux kernel and GNU C Library. The following code snippet (taken from `mutex.c`) is run concurrently by multiple threads and was included in all implementations. Of course, to test different locking mechanisms, the calls to the mutex functions were replaced appropriately.

```
pthread_mutex_t mutex1, mutex2;
int counter;
int unprotected_counter;
int incorrectly_protected_counter;

void *concurrently_run_function()
{
    int local_counter_copy;

    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);

    counter++;
    local_counter_copy = (counter == 1) ? 0 : 1;

    printf("Counter value: %d\n", counter);

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);

    pthread_mutex_lock(
        (local_counter_copy)?(&mutex1):(&mutex2));
    incorrectly_protected_counter++;
    pthread_mutex_unlock(
        (local_counter_copy)?(&mutex1):(&mutex2));

    unprotected_counter++;
}
```

The function uses two locks for mutual exclusion, while the variables `counter`, `unprotected_counter` and `incorrectly_protected_counter` are accessed using both, no, and inconsistently used locks, respectively. Running this program with an active race condition detector (e.g. inside the `detrace` tool), we obtain the results shown in Table 6.2.

Table 6.2: Automatically generated report from the race condition detection system applied to the mutex testing application.

	Address	**Symbol**	**Location**
1	`0x080c9358`	`_IO_stdfile_1_lock+0x8`	`ioputs.c` (Line 2 - in listing shown)
2	`0x080c9350`	`_IO_stdfile_1_lock`	`ioputs.c` (Line 2)
3	`0x080c6830`	`unprotected_counter`	`mutex.c` (Line 23)
4	`0xb7f9dbd8`	n/a (stack location)	`pthread_join.c`
5	`0x080c6834`	`incorrectly_protected_counter`	`mutex.c` (Line 20)
6	`0xb6f99d94`	n/a (stack location)	`pthread_join.c`
7	`0x080c6180`	`_IO_2_1_stdout_+0x14`	`genops.c` (Line 8)
8	`0x080c61e8`	n/a	`genops.c` (Line 8)

Whereas race conditions 3 and 5 were anticipated, the other 6 warnings need closer examination: Looking at the source location provided for race conditions 1 and 2, the following code can be found:

```
void *__self = THREAD_SELF;
if ((_IO_stdout).owner != __self)
  {
    lll_lock ((_IO_stdout).lock, LLL_PRIVATE);
    (_IO_stdout).owner = __self;
  }
```

While the accesses to `_IO_stdout` and `(_IO_stdout).owner` are race conditions, this does not have any impact in practice. The reason is the following: Although it is possible that multiple threads enter the body of the if-statement at the same time and invoke `lll_lock` (which is a race condition error), this function then performs correct locking.

Race conditions 7 and 8 reveal the following source lines:

```
int
_IO_flush_all_lockp (int do_lock)
{
  ..
}

int
_IO_cleanup ()
{
  /* We do *not* want locking.  Some threads might
     use streams but that is their problem, we flush
     them underneath them.  */
  int result = _IO_flush_all_lockp (0);

  ..
}
```

This clearly shows that a race condition is present, but this race was deliberately tolerated by the developers.
Finally, race conditions 4 and 6 are reported because the New Posix Thread Library attempts to reset each thread's `THREAD_SELF` variable (stored at the bottom of the stack) to `-1` once this thread has died. As the current implementation of our race detection system is not aware of a thread's termination, it cannot eliminate this false positive automatically.

GNU C Library, read/write locking

Looking at a similar binary as in the previous section (replacing the `mutex` with a `read/write lock`), the report reveals another interesting code location in the library's code for unlocking the synchronization object:

```
1     .text
2
3     .globl __pthread_rwlock_unlock
4     .type  __pthread_rwlock_unlock,@function
5     .align 16
6   __pthread_rwlock_unlock:
7     ...
8     mov    0x19(%edi),%ecx
9     call   __lll_lock_wait
10    ...
```

Line 8 of the code snippet uses the read/write lock in register `EDI` to load the lock's `__shared` variable into register `ECX`, passing it as parameter to function `__lll_lock_wait`.

```
typedef union
{  struct
   {
     int __lock;
     ...
     unsigned char __flags;
     unsigned char __shared;
     unsigned char __pad1;
     unsigned char __pad2;
     int __writer;
   } __data;
  ...
} pthread_rwlock_t;
```

As can be seen from the declaration of the structure `pthread_rwlock_t` above, `__shared` is only one byte long. Moreover, it is unaligned. Since the read operation (on Line 8 of the code) loads four bytes, this access also touches the struct's variables `__pad1`, `__pad2`, and the first byte of `__writer`. As this byte belongs to the next memory word, the Mondrian protection system also checks the protection label that belongs to the `__writer` variable. This leads to a race condition warning, because the `__writer` variable is accessed with locks by other threads. If the 4-byte access is replaced by a single-byte access, the warning disappears as expected.

ClamAV daemon

To demonstrate that the detection system can also handle larger applications, we have checked the anti-virus software ClamAV for possible race conditions. Although the examination reported ten race conditions, we discuss as example only one case.
This bug report refers to the unsynchronized access to variable `progexit` (in file `serverth.c`). Looking at the appropriate source code shown below, we were surprised to see that the variable access is actually protected by a mutex. However, searching for other references to the `progexit` variable, we found code that accesses this variable without holding the `exit_mutex`, thus confirming the race condition warning.

```
static void scanner_thread(void *arg)
{    ...
     switch(ret) {
          case COMMAND_SHUTDOWN:
               pthread_mutex_lock(&exit_mutex);
               progexit = 1;
               ...
               pthread_mutex_unlock(&exit_mutex);
               break;
...
}
```

Apache web server

In addition to its large code base, the Apache web server introduced another burden to the detection tool: Besides using a pool of threads to handle pending tasks, Apache uses, as most web servers, the `fork` system call to duplicate the currently running process. This allows the server be more responsive but forces the protection system to be aware of task duplications to copy the current protection information into the new process.
Furthermore, examining the original version of the server with the race detector showed a plethora of false positives. This resulted from the fact that Apache includes its own heap memory allocator: Instead of relying on the

GNU C Library directly, the code manages *pools* of memory regions. Every pool can include several *subpools* and memory areas that can be deallocated collectively. We have thus employed the same approach to clear the protection information of a memory buffer as done for the Malloc code in the C library. Using the system call interface, we were able to patch the memory allocator by including only two new lines in the allocator's source code.

In total, our system found 33 potential race conditions for Apache. Analogously to the ClamAV section, we only deal with one example race condition that was reported during our examination. Moreover, a few other error locations are shown in Table 6.3.
Looking at the source location reported for the fourth race condition in Table 6.3, we see the following function:

```
static void * APR_THREAD_FUNC worker_thread(...)
{
    ...
    /* FIXME: should be synchronized - aaron */
    requests_this_child--;
```

The comment clearly provides a strong confirmation for the correctness of this error report.

Table 6.3: Automatically generated (incomplete) report from the race condition detection system applied to the Apache web server.

	Address	Symbol	Location
1	0x080c373c	exploded_cache_gmt+0x3c	util_time.c, line 125
2	0x08153058	n/a	fdqueue.c, line 345
3	0x081531d0	n/a	worker.c, line 892
4	0x080c3ca4	requests_this_child	worker.c, line 896

Chapter 7

Related Work

As mentioned previously, the general protection framework that we designed is an extension of the Mondrian memory protection idea [37, 38]. In contrast to the original design, we have extended the two protection bits (that have predefined semantics) with 30-bit protection labels that can freely be used by the operating system and the running processes. This flexible framework allowed us to build different techniques to protect sensitive information from being overwritten, as well as to implement a race detection algorithm.
While Mondrian memory protection was used to define different protection domains, these domains have mostly been used to put different kernel modules in separate compartments so that one faulty module does not lead to a complete OS crash [39]. Our approach offers more flexibility (through larger protection labels and user-defined policies defined in the kernel). This allowed us to directly implement a number of different mechanisms on top of our architecture.

Memory corruption protection. Our stack and heap protection techniques are related to numerous systems that aim to detect or prevent attacks that exploit memory corruption bugs. Here, we can only provide a brief overview of these techniques, discussing a few systems that stand as examples for certain categories. One of the earliest techniques to prevent buffer overflows from overwriting the return address was StackGuard [15]. This system modifies the compiler so that a special canary word is stored next to the return address. This canary is later checked when the function returns. When a modification is detected, this indicates a buffer overrun. StackGuard was later improved by systems such as RAD [14]. RAD is also a compiler modification, but it protects the return address by inserting code that stores a copy of the return address at a safe location when a function is invoked and using this safe copy on function return. A system that works similar to StackGuard, but that protects heap management information, is presented in [28].

In addition to systems that modify the compiler to protect information such as return addresses or heap management information, researchers have proposed static analysis systems to detect possible unsafe code that might lead to memory corruption. One of the first techniques uses integer range analysis for checking buffer accesses for indexes that are out of bounds [35]. This proposal has been followed by numerous others, using for example static program analysis [7, 18] or model checking [13]. Also, there are approaches [24, 26] that aim to retrofit C programs with type safety. To this end, programs have to be (slightly) modified to conform to new programming languages that are very similar to C, but that lack constructs for which the compiler cannot guarantee the absence of memory errors.

Finally, there are techniques that use a combination of static and dynamic program analysis to provide strong security guarantees such as control flow integrity [6] or data integrity [12]. Clearly, these techniques can identify attacks in which return addresses (or other pointers) are overwritten to hijack the execution of a process.

Race condition detection. Similar to memory corruption bugs, race conditions [10] are an important class of program errors that have received significant attention from the research community. Again, there are static and dynamic techniques to approach the problem and to analyze code for the presence of race conditions.

Static techniques [11, 20, 21] use compile-time analysis of the program source code, reporting all potential races that could occur in a program execution. Dynamic techniques [29, 34], on the other hand, execute the program and analyze a history of its memory accesses and synchronization operations. This has the advantage that only feasible program paths are seen. However, dynamic approaches have the limitation that they can typically not inspect all possible execution paths.

Dynamic approaches are usually either based on a lock-set approach or on the happens-before relationship. Systems that use a lock-set approach (such as Eraser [29]) require that all shared variable accesses are protected by a

lock. In case the system identifies different accesses to a shared variable for which there is no lock consistently held, a potential race condition is identified. Systems [17] that leverage the happens-before relationship attempt to establish a partial temporal ordering between all data accesses. If there is a data access for which no such order can be found, the system has detected a race condition. In general, systems that are based on the happens-before relationship are more general, since they can be applied to non-lock-based synchronization operations. However, they are typically less efficient in finding race conditions (i.e., they produce more false negatives).
Given our system applications (stack protection, heap protection, and race condition detection), we are aware of the fact that they are not novel contributions *per se*. However, they demonstrate the flexibility of our novel memory protection architecture. Thus, by introducing a versatile and general protection system, we believe that we have introduced an architecture that can serve as the basis for future security techniques.

Chapter 8

Summary and conclusion

Traditional memory protection, as implemented in Intel's x86 architecture, has the shortcoming of being very coarse-grain. A previous implementation of Mondrian memory protection improved the granularity of protected memory regions, but still lacks flexibility and precision of the protection information that is stored.
In this thesis, we present an extended version of Mondrian memory protection. It allows the system to store generalized protection labels of 30 bit for every word in an application's memory context. A user-defined kernel module allows to specify rules that are examined during memory access by the CPU. Through this, a broad field of applications can be covered by building on top of our general framework.
To demonstrate the usability and effectiveness of our extended Mondrian memory protection, we have implemented a system that provides stack and heap protection as well as dynamic race condition detection. We used our system on a number of large, real-world applications. Our evaluation confirms that the protection mechanisms effectively prevent certain classes of memory corruption errors. Moreover, the race condition detector shows that even well-known code bases such as the GNU C Library and the Apache web server contain problems related to race conditions.

Bibliography

[1] Clam AntiVirus. `http://www.clamav.org/`, 2008.

[2] GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`, 2008.

[3] GNU C Library. `http://www.gnu.org/software/libc/`, 2008.

[4] The Apache HTTP Server Project. `http://httpd.apache.org/`, 2008.

[5] The GNU Privacy Guard. `http://gnupg.org/`, 2008.

[6] Martin Abadi, Ulfar Erlingsson, and Jay Ligatt. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[7] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy*, 2002.

[8] Fabrice Bellard. Qemu: A Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference, Freenix Track*, 2005.

[9] Fabrice Bellard. Tiny C Compiler. `http://fabrice.bellard.free.fr/tcc/`, 2008.

[10] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[11] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2), 1996.

[12] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[13] Hao Chen, Drew Dean, and David Wagner. Model Checking One Million Lines of C Code. In *Network and Distributed System Security (NDSS)*, 2004.

[14] Tzi cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[15] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Usenix Security Symposium*, 1998.

[16] Peter Denning. Virtual Memory. *ACM Computing Surveys*, 2(3), 1970.

[17] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *ACM Symposium on the Principles and Practice of Parallel Programming*, 1990.

[18] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[19] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`, 2005.

[20] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[21] Cormac Flanagan, K. Leino, Mark Lillibridge, Greg Nelson, James Saxe, and Raymie Stata. Extended Static Checking for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[22] Eric Grevstad. CPU-Based Security: The NX Bit. `http://hardware.earthweb.com/chips/article.php/3358421`, 2004.

[23] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part1, 2006.

[24] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Usenix Annual Technical Conference*, 2002.

[25] Doug Lea. A Memory Allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`, 2000.

[26] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Principles of Programming Languages (POPL)*, 2002.

[27] The Pax Team. Pax: Non-executable stack and heap. `http://pax.grsecurity.net/`, 2008.

[28] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time Detection of Heap-based Overflows. In *Usenix Large Installation Systems Administration Conference (LISA)*, 2003.

[29] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[30] Vikram Shukla. Explore the Linux memory model. `http://www.ibm.com/developerworks/linux/library/l-memmod/`, 2006.

[31] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 7th edition edition, 2004.

[32] Solar Designer. Linux kernel patch from the Openwall Project. `http://www.openwall.com/linux/`, 2008.

[33] Andrew Tanenbaum and Albert Woodhull. *Operating Systems: Design and Implementation*. Pearson Prentice Hall, 3rd edition edition, 2006.

[34] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *Usenix Security Symposium*, 2003.

[35] David Wagner, Jeffrey Foster, Eric Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security (NDSS)*, 2000.

[36] David Wheeler. Secure programmer: Prevent race conditions. `http://www.ibm.com/developerworks/linux/library/l-sprace.html`, 2008.

[37] Emmett Witchel and Krste Asanovic. Hardware Works, Software Doesn't: Enforcing Modularity with Mondrian Memory Protection. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[38] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian Memory Protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[39] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: Memory Isolation for Linux using Mondrian Memory Protection. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.