Extending Mondrian Memory Protection

Clemens Kolbitsch
Secure Systems Lab, Vienna University of Technology
Treitlstr. 1/4. Floor/E183-1
A-1040 Vienna
Austria

ck@iseclab.org

Christopher Kruegel
UC Santa Barbara
Department of Computer Science, University of California
Santa Barbara, CA 93106
USA

chris@cs.ucsb.edu

Engin Kirda
Eurecom
Sophia Antipolis science park
2229, Route des Crłtes
06560 Valbonne Sophia Antipolis
France

kirda@eurecom.fr

ABSTRACT

Most modern operating systems implement some sort of memory protection scheme for user processes. These schemes make 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 predefined semantics. This is not sufficient to implement more complex security techniques, for example, a race condition detection system.

In this paper, we propose 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 and suggest further areas of application. 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.

1.0 INTRODUCTION

Most modern operating systems implement some sort of memory protection for user processes [1–3]. 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 [3].

While memory protection is a useful technique to improve the reliability and security of processes, it is fairly coarse-grained. 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 such as on stack memory.

Mondrian memory protection [4, 5] 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. However, Mondrian memory protection is still limited to only *two* permission bits with a predefined 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 vulnerabilities that are difficult to detect [6–9]. This problem is exacerbated by the increasing use of parallel programming and multi-threaded applications.

In this paper, 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. Using this general protection framework, one can implement different techniques such as buffer overflow detection, precise memory protection, or race condition detection. These protection labels are controlled via an extension to the x86 instruction set 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.

In the rest of the paper, we first describe the general mechanisms that our system supports, as well as details of the implementation. Then, we discuss a number of concrete techniques that leverage the general memory protection mechanisms as well as a race condition detector. Finally, we describe our experiments that demonstrate that the system can provide the expected protection and ability to detect races with reasonable overheads.

The contributions of this paper are as follows:

- We present a generalization of the basic Mondrian memory protection scheme that allows user processes and the operating system to operate with flexible 30-bit protection labels at word-granularity. This scheme is supported by extensions to the processor instruction set and the operating system.
- We demonstrate the flexibility of the protection scheme by implementing three techniques that build upon the general protection framework. These techniques provide return address protection, heap management protection, and race condition detection capabilities. Furthermore, we elaborate on further application areas of the general protection scheme.
- We show that our extensions are successful in preventing certain types of attacks and in finding race conditions while incurring a reasonable runtime and memory overhead.

2.0 SYSTEM OVERVIEW

In this section, we briefly explain the inner workings of Intel's x86 memory management as well as the Mondrian memory protection architecture as presented in [5]. Then, we outline the limitations of the current approaches and present our design to overcome the problems.

2.1 Intel x86 Memory Management

Modern operating systems divide the address space visible to a user process (often referred to as the virtual address space) 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 a virtual address.

Besides the mapping information, a page table entry contains 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 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).

2.2 Mondrian Memory Protection

Similar to the x86 architecture, Mondrian memory protection employs two bits to store four different access permissions (*no access*, *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 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). On every access to a memory address, the CPU's protection enhancement looks up the address protection bits stored in the corresponding protection table. To reduce the memory overhead introduced by the protection tables, the implementation in [5] provides different possibilities for storing the table's structure. This allows to adjust the size of the region the protection information applies to.

Despite its flexibility, Mondrian memory protection shares one shortcoming with the standard x86 protection scheme - which is the fact that one cannot associate more than two bits of protection information to a memory region. Moreover, the predefined meaning of the four possible bit combinations significantly limits the flexibility of the protection system. These two drawbacks are the starting point of our extended Mondrian memory protection technique: While trying to combine the simplicity of x86 memory protection with the fine granularity of the original Mondrian memory protection, our implementation allows a *user-specified examination* of *expanded protection information* stored for memory regions. For this purpose, our protection architecture is split into the three components *protection hierarchy*, *access control*, and *access policies*, described in Section 2.3, 2.4, and 2.5 respectively.

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

2.3 Protection Hierarchy

Similar to the page table 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 control register `CR6` serves as entry point into the protection hierarchy. An overview of the protection hierarchy can be seen in Figure 1.

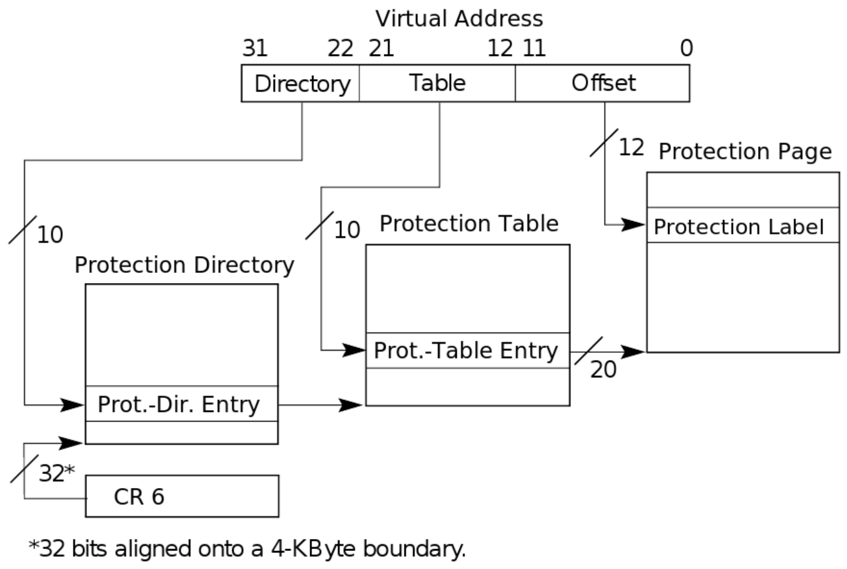

Figure 1: Protection hierarchy.

To save space when the protection labels of all words in a particular page are identical, we use two 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.

Clearly, the operating system has the ability to directly manipulate the content of the protection tables. For example, this can be done in response to a protection fault, or when a process starts (in order to define appropriate protection settings for certain memory regions). However, we also want to provide an interface that allows a user process to modify its memory protection settings in a controlled fashion. For this purpose, we have introduced a set of new machine instructions, allowing a process to read, set, or modify protection information through the instructions `prot_mov`, `prot_and`, and `prot_or`. Further, a thread or process can only modify protection information of memory areas it has read- or write-access to. Thus, one thread cannot undo restrictions imposed previously (e.g., by a controlling thread).

2.4 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 shown in Figure 1). 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 granularity protection*, the corresponding bits of the protection table entry are directly used for access control[2].

In case no protection information is found (because the protection directory or protection table does not contain a corresponding entry), 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, our 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. Figure 2.4 shows two examples for access control decisions that yield different results.

Prot. label	$0b00000000000000100111000111010000$
`CR5`	$0b00000000000000100001000111010000$
`Read mask`	$0b10000000000000011111111111111100$
Result	**Protection violation**
Prot. label	$0b00000000000000100111000111010000$
`CR5`	$0b00000111000000100111000111010000$
`Read mask`	$0b10000000000000011111111111111100$
Result	**Read access granted**

Figure 2: Access control decisions for a read access.

2.5 Memory Access Policies

As previously mentioned, our 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

[2]The two least significant bits of a table entry are used to distinguish between granularity levels and mark entries as valid. Although the high granularity protection would support 32 bits to be used as protection label, the two extra bits remain unused to create a consistent architecture.

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 the following Section 3.0, we will 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 one hand, a program (or a compiler) can use the newly introduced instructions to manipulate the memory protection settings (labels) during process execution. In addition, 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.

2.6 Implementation

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 [10] 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.

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 through the currently unused but specified control register `CR7`. 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 instructions to manipulate the memory protection settings are expressed as instructions that modify the control register `CR7`.

To allow the first component of our architecture to provide the protection hierarchy to the CPU, we had to hook page allocation and destruction code, as well as a few thread scheduling routines in the operating system. For this, we used the Linux kernel, since it allows easy inspection and modification of the source code.

3.0 SYSTEM APPLICATIONS

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 our proposed architecture. More precisely, Section 3.1 shows how stack and heap areas can be protected against memory corruption attacks. In Section 3.2, we discuss how the architecture can be leveraged to implement a race condition detection system, whereas Section 3.3 demonstrates the possibility of protecting sensitive data, even in a multi-threaded environment.

While the former system applications are not novel *per se*, we show how easily 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.

3.1 Buffer Smashing 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, stack, and heap 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 [11]) 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, what protects 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.

Extending this idea, we have added protection code to memory allocation functions in order to prevent heap buffer overruns. Doug Lea's Malloc [12], the memory allocator the GNU C library implementation is based on, uses in-band management information to maintain currently allocated chunks of memory. Protecting these memory areas prevents undeliberate accesses that can also be leveraged to eventually overwrite control data, hijacking the program's control flow.

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 `tinycc` compiler [13]. Further, we have introduced code to the allocator's `alloc`, `realloc`, and `free` routines in the C library to unlock in-band management data prior to each modification. The protection code is quite straightforward. To ensure that a certain memory word (such as the return address, boundary around a buffer, or in-band data) 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 during unprotection, restoring the original label of the memory addresses.

3.2 Race Condition Detection

To show a second application for leveraging our extended Mondrian memory protection architecture, we have made our own implementation of the race condition detection algorithm described in [14]. In this section, we briefly describe the original detection algorithm and how our implementation differs from that. With this system, an application can, prior to its release, be tested for race conditions, a quite easy-to-make and hard-to-find programming error. Section 4.3 then provides an overview of applications tested with our system, as well as of actual race condition bugs that we found.

In [14], 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 where (1) at least one

access is a write, and (2) the threads use no explicit mechanism to prevent the accesses from being simultaneous.

In order to be able to detect possible race conditions in a program, the detection algorithm 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), this memory address is `owned exclusively` by this thread. To indicate this fact, the shadow memory contains the owner's 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. The set is called the lock-set for this location.

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 this intersection ever yields an empty set, the detection algorithm has found a data race and can issue a warning. For a more detailed description of the detection algorithm, refer to [14].

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 PID of the currently executing process into bits 3 through 19 of the control register `CR5`[4], 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 remaining bits can be used to store the lock-set ID (which indicates the locks that were held while accessing this memory region).

The following example shows two threads accessing a common memory location `addr`. We assume that `addr` is initially owned exclusively by the first thread. Thus, the protection label in Line 1 stores the ID of Thread 1 (starting at bit 3). It can be seen how the accesses to this variable change the memory's protection information until a race condition is detected in Line 9. The race is detected when the first thread writes to the shared memory region while holding lock `m1`. The reason is that, previously, in Line 6, the same memory was accessed by Thread 2 using only lock `m2`. Thus, the lock-set is empty, indicating a bug.

[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.

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

	CR5	**Prot. data**	**Locks**	**Threads**	
		of `addr`	**held**	**PID 1**	**PID 2**
1		`000...01000`	$\{\} = ID(0)$		
2	`01000`		$\{m1\} = ID(1)$	`lock(m1);`	
3	..			`write(addr);`	
4	..		$\{\} = ID(0)$	`unlock(m1);`	
5	`10000`		$\{m2\} = ID(2)$		`lock(m2);`
6	..	`010...00010`			`read(addr);`
7	..		$\{\} = ID(0)$		`unlock(m2);`
8	`01000`		$\{m1\} = ID(1)$	`lock(m1);`	
9	..	`010...00000`		`write(addr);`	

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, we extended the system call interface to receive notifications about user-land locking operations (such as calls to the mutex and semaphore code provided by the C library). By patching the dynamically loadable GLibC[5], we are able to test completely unmodified applications for the occurrence of race conditions allowing much higher applicability than the original Eraser implementation [14].

Further, 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.

3.3 Memory Containment

To give a short insight into further areas that could be explored with our generic protection mechanism, we briefly want to mention the idea of memory containment. Today, many applications used in everyday life are expandable with the use of plugins or add-ons. Typically, these extensions are provided by third parties and run within the main application's memory context. Thus, they have complete access to all resources held by the application and may alter these unrestrictedly.

As an example, consider internet browsers, where such add-ons could manifest as search bars, download helper, or language packs. Although these extensions only require very limited access to internal structures (e.g. for installing callback routines), they have full access on sensitive data, such as password management, browser history, connectivity settings, etc.

Due to their vast popularity and apparent abundance of programming errors, many malware variants now use these extensions as target for drive-by download vulnerability spreading [15]. This shows that add-ons often contain poorly written code and could thus easily interfere with the stability of the hosting application. By leveraging extended Mondrian memory protection and placing untrusted code into separate threads, the application can restrain the memory areas add-ons have access to.

As mentioned in Section 2.3, a thread can only alter protection labels of memory areas it can access. Thus, the main application can protect vital memory structures by restricting access based on its own thread ID prior

[5]As part of the GNU C library, the Native Posix Thread Library (NPTL) provides several user-land synchronization capabilities such as *mutexes*, *read/write locks*, *semaphores*, and *spinlocks*.

to loading extension plugins. This prevents any untrusted code from jeopardizing the browser's stability or accessing sensitive data. We can imagine a broad use of this concept when thinking of other multi-threaded applications, such as server applications, mail clients, document readers, and so on.

4.0 EVALUATION

This section 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.

4.1 Performance

Although our prototype implementation did not focus on performance, we have attempted to estimate the performance penalty factor introduced. Table 1 shows averaged results of measuring ten executions of encrypting a binary file with `gpg2`, solving a sudoku puzzle[6], and scanning multiple files using the open source (GPL) anti-virus toolkit daemon of `ClamAV`.

Table 1: Relative performance penalties introduced by the extended Mondrian memory protection compared to the original system (i.e., without any Mondrian memory protection).

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

Analyzing the results, 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 [14] 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.

[6] `http://pubpages.unh.edu/~pas/hacks/sudoku/`

Another critical issue is the additional load on memory caches. The amount of extra memory accesses by the protection enhancement might cause considerable performance loss. To remedy this problem, we propose an additional *protection cache* next to the traditional *instruction-* and *data-caches* or merely increasing the existing cache's size.

4.2 Memory Overhead

As mentioned in Section 2.3, the additional memory required for storing protection labels depends on the granularity level chosen. For 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*.

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.

4.3 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, 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 Apache2 web server.

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 code snippet below (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.

```
 1 pt_mutex_t m1, m2;
 2 int ctr, unprot_ctr;
 3 int incorrect_ctr;
 4
 5 void *concurrent() {
 6   int cpy;
 7   pt_mutex_lock(&m1);
 8   pt_mutex_lock(&m2);
 9   cpy = (ctr++);
10
11   printf("ctr %d", ctr);
12   pt_mutex_unlock(&m2);
13   pt_mutex_unlock(&m1);
14
15   pt_mutex_lock(
16       cpy?(&m1):(&m2));
17   incorrect_ctr++;
18   pt_mutex_unlock(
19       cpy?(&m1):(&m2));
20
21   unprot_ctr++;
22 }
```

The function uses two locks for mutual exclusion, while the variables `ctr`, `unprot_ctr` and `incorrect_ctr` are accessed using both, no, and inconsistently used locks, respectively. Running this program with an active race condition detector, we obtain the results shown in Table 2.

Table 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 below)
2	0x080c9350	_IO_stdfile_1_lock	ioputs.c (Line 2)
3	0x080c6830	unprot_ctr	mutex.c (Line 21)
4	0xb7f9dbd8	n/a (stack location)	pthread_join.c
5	0x080c6834	incorrect_ctr	mutex.c (Line 17)
6	0xb6f99d94	n/a (stack location)	pthread_join.c
7	0x080c6180	_IO_2_1_stdout_+0x14	genops.c
8	0x080c61e8	n/a	genops.c

Whereas race conditions 3 and 5 were anticipated, the other 6 warnings need closer examination: The source code below shows the location of race conditions 1 and 2. While the accesses to `_IO_stdout` and `(_IO_stdout).owner` are race conditions, this does not have any impact in practice. The reason is that 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.

```
1 void *__self = THREAD_SELF;
2 if ((_IO_stdout).owner != __self) {
3   lll_lock((_IO_stdout).lock, LLL_PRIVATE);
4   (_IO_stdout).owner = __self;
```

The next code listing correspond to race conditions 7 and 8. They clearly show that a race condition is present, but this race was deliberately tolerated by the developers.

```
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);
```

Finally, race conditions 4 and 6 are reported because the NPT 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.

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 will discuss as example only one case. Unfortunately, space limitations prevent us from discussing the remaining ones in more detail.

One bug report refers to the unsynchronized access to variable `progexit` (in file `serverth.c`). Looking at the appropriate source code, we were surprised to see that the variable access is actually protected by a mutex `exit_mutex`. However, searching for other references to the `progexit` variable, we found code that accesses this variable without holding the corresponding mutex, thus confirming the warning.

Apache web server.

In addition to its large code base, the Apache web server introduced another burden to the detection tool: As most web servers use the `fork` system call to duplicate the currently running process to be able to be more responsive, the protection system must be aware of this and copy the current protection information into the new process.

In total, our system found 33 potential race conditions for Apache. Analogously to the ClamAV section, we will only deal with one example race condition that was reported during our examination. Moreover, a few other error locations are shown in Table 3.

Table 3: Excerpt of automatically-generated 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

Looking at the source location reported for the fourth race condition in Table 3, we see the function below. The comment clearly provides a strong confirmation for the correctness of this error report.

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

5.0 RELATED WORK

As mentioned previously, the general protection framework that we designed is an extension of the Mondrian memory protection idea [4, 5]. 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 [16]. Our approach offers more flexibility allowing 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 [11]. 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 [17]. 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 [18].

Race condition detection.

Similar to memory corruption bugs, race conditions [6] are an important class of program errors that have received significant attention from the research community.

There are two techniques to approach the problem and to analyze code for the presence of race conditions: Static techniques [7, 19] use compile-time analysis of the program source code, reporting all potential races that could occur in a program execution. Dynamic techniques [8, 14], 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 [14]) 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 [20] 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 are applicable to non-lock-based synchronization operations. However, they are typically less effective in finding race conditions (i.e., produce more false negatives).

Given the implemented system applications, we are aware that they are not novel contributions *per se*. However, they demonstrate the flexibility of our novel memory protection architecture introducing a versatile and general system that can serve as the basis for future security techniques.

6.0 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 paper, we present an extended version of Mondrian memory protection. It allows the system to store generalized protection labels of 30 bits 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 race-condition-related problems.

ACKNOWLEDGMENTS

This work has been supported by the Austrian Research Promotion Agency (FFG) through project SECoverer and by the European Commission through project FP7-ICT-216026-WOMBAT.

REFERENCES

[1] Denning, P., “Virtual Memory,” *ACM Computing Surveys*, Vol. 2, No. 3, 1970.

[2] Silberschatz, A., Galvin, P., and Gagne, G., *Operating System Concepts*, Wiley, 7th ed., 2004.

[3] Tanenbaum, A. and Woodhull, A., *Operating Systems: Design and Implementation*, Pearson Prentice Hall, 3rd ed., 2006.

[4] Witchel, E. and Asanovic, K., “Hardware Works, Software Doesn’t: Enforcing Modularity with Mondrian Memory Protection,” *Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[5] Witchel, E., Cates, J., and Asanovic, K., “Mondrian Memory Protection,” *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

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

[7] Bishop, M. and Dilger, M., “Checking for race conditions in file accesses,” *Computing Systems*, Vol. 9, No. 2, 1996.

[8] Tsyrklevich, E. and Yee, B., “Dynamic detection and prevention of race conditions in file accesses,” *Usenix Security Symposium*, 2003.

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

[10] Bellard, F., “Qemu: A Fast and Portable Dynamic Translator,” *Usenix Tech. Conference, Freenix*, 2005.

[11] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q., “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” *Usenix Security Symposium*, 1998.

[12] Lea, D., “A Memory Allocator,” `http://g.oswego.edu/dl/html/malloc.html`, 2000.

[13] Bellard, F., “Tiny C Compiler,” `http://fabrice.bellard.free.fr/tcc/`, 2008.

[14] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T., “Eraser: A Dynamic Data Race Detector for Multithreaded Programs,” *ACM Transactions on Computer Systems*, Vol. 15, No. 4, 1997.

[15] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M., “A Crawler-based Study of Spyware in the Web,” *NDSS*, 2006.

[16] Witchel, E., Rhee, J., and Asanovic, K., “Mondrix: Memory Isolation for Linux using Mondrian Memory Protection,” *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[17] Chiueh, T. and Hsu, F.-H., “RAD: A Compile-Time Solution to Buffer Overflow Attacks,” *International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[18] Robertson, W., Kruegel, C., Mutz, D., and Valeur, F., “Run-time Detection of Heap-based Overflows,” *Usenix Large Installation Systems Administration Conference (LISA)*, 2003.

[19] Engler, D. and Ashcraft, K., “RacerX: Effective, Static Detection of Race Conditions and Deadlocks,” *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[20] Dinning, A. and Schonberg, E., “An empirical comparision of monitoring algorithms for access anomaly detection,” *ACM Symposium on the Principles and Practice of Parallel Programming*, 1990.