ABSTRACT

LIAKH, SIARHEI. Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach. (Under the direction of Dr. Xuxian Jiang.)

Code injection continues to pose a serious threat to computer systems. Among existing solutions, $W \oplus X$ is a notable approach to prevent the execution of injected code. In this thesis, we focus on the Linux kernel memory protection and systematically check for possible $W \oplus X$ violations in Linux kernel by developing a Murphi-based abstract model. In particular, based on the abstract model, we have discovered several serious shortcomings in the current Linux kernel design and implementation that violate the $W \oplus X$ property. We have confirmed with the Linux community the presence of these problems and accordingly developed five Linux kernel patches. Four of the patches have been accepted for integration into the mainline Linux kernel. Our evaluation with these patches indicate that they involve only minimal changes to the existing code base and incur negligible performance overhead.

Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach

by
Siarhei Liakh

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Sciense

Raleigh, North Carolina

2010

APPROVED BY:

Dr. Vincent v. Freeh	Dr. Frank Mueller

Dr. Xuxian Jiang
Chair of Advisory Committee

BIOGRAPHY

Native of Belarus, Siarhei Liakh attended Belorussian State University of Informatics and Radioengineering from 1997 through 1999. Following the move to the United States of America, he acquired a degree of Bachelor of Siense in Computer Sciense from Western Carolina University in 2004. After being admitted to North Carolina State University in 2007, he graduated with degree of Master of Sciense in Computer Sciense in 2010.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Xuxian Jiang for being an editor in chief for this Thesis, and NCSU Doctorate Student Mike Grace for providing useful feedback.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Despite years of research, code injection attacks continue to be one of the major ways of computer break-ins and malware propagation [22]. Specifically, a code injection attack is a method whereby an attacker inserts malicious code into a running process and transfers execution to the malicious code (e.g., by hijacking its control flow). After that, the attacker can gain control of a running process and carry out other malicious activities, including the installation of bot programs for remote control and the modification of system files to allow for unauthorized access, etc.

There exist a variety of solutions [15, 26, 29, 30, 31, 34] to deal with code injection attacks. Among the most notable, $W \oplus X$ [1] is a scheme that has been proposed to counter code injection attacks. In essence, $W \oplus X$ enforces the following property, "a given memory page will never be both writable and executable at the same time." The basic premise behind it is that if a page cannot be written to and later executed from, code injection becomes impossible. Due to its effectiveness in defending against code injection attacks, since its proposition, $W \oplus X$ has been widely adopted in commodity OSs (e.g., Windows and Linux). Moreover, hardware vendors such as Intel and AMD follow up this scheme by providing necessary hardware support (in the form of NX support [9]) to facilitate the $W \oplus X$ enforcement.

From the OS kernel perspective, establishing and maintaining the $W \oplus X$ property requires a sound design. In this thesis, we look into the Linux kernel and analyze the way it takes to protect its own kernel memory. This is important as the Linux kernel is typically a part of trusted computing base (TCB) in existing solutions to defend against code injection attacks. In our analysis, we took a model checking approach so that we can take advantage of its power to rigorously examine the soundness and completeness of $W \oplus X$ enforcement in Linux kernel. More specifically, we first build a model of Linux kernel memory management subsystem and

[1] Strictly speaking, the property is $\neg(W \wedge X)$, but we chose to use the traditional $W \oplus X$ notation to emphasize mutual exclusivity of write and execute access.

then apply model checking to verify the $W \oplus X$ property. In case of violation, model checking has the unique advantage in accurately pinpointing potential problems in the current Linux kernel design and implementation.

We have successfully developed a Murphi [10]-based abstract model to analyze Linux kernel memory management. In our modeling process, we were surprised to discover several issues related to Linux kernel memory management: (1) First, the current Linux kernel does not strictly separate the kernel code and kernel data, immediately leading to $W \oplus X$ violation. (2) Second, as part of its implementation, Linux kernel promotes creation of multiple virtual aliases, potentially with conflicting permissions, for the same physical memory page, leading to exploitable scenarios for kernel data execution or kernel code modification.

We have confirmed with the Linux kernel community the presence of these issues. We have also accordingly developed kernel patches to fix these problems and these patches [19, 21, 20, 18] are being integrated into mainstream Linux kernel.[2] Our evaluation indicates that these patches are compatible with existing Linux kernel code base and contain minimum modifications to the existing interfaces that manage kernel memory. We also observe that these patches impose virtually no performance overhead despite a moderate increase in memory consumption.

The rest of the thesis is structured as follows. We start with Linux kernel modeling in Chapter 2 and present the model checking results in Chapter 3. Then we present and evaluate our solution in Chapter 4 and Chapter 5, respectively. After that, we examine limitations of our solution and suggest possible improvements in Chapter 6. Finally, we discuss related work in Chapter 7 and conclude our thesis in Chapter 8.

[2]For the convenience of kernel patch debugging and adoption, we have developed five smaller patches in total (Chapter 4). Four of them are being integrated into mainstream Linux kernel and the remaining one is still being evaluated for suitability in mainline Linux kernel.

Chapter 2

Modeling and Design

2.1 Murphi background

Our abstract modeling of Linux kernel memory protection is based on Murphi [10], which is both a language and a tool for model verification with explicit state enumeration. To use Murphi, we need to write a Finite State Machine (FSM) description, which will be taken as an input by Murphi to produce an executable. The executable, once started, will perform the model verification task and produce the output detailing the verification results.

In Murphi, a finite state machine description consists of three parts: a set of *states*, a set of *transition rules* and an *initial state*.

- The set of *states* is implicitly defined through the declaration of global variables. Each combination of values of each variable naturally produces a unique state of the system. Note not all of these potential combinations are reachable in the FSM.
- The transitions between the states of FSM are defined through a set of *transition rules*, each consisting of two parts, a *guard* and an *action*. A guard is a logic expression that determines the conditions under which an action can be taken. An action is a set of instructions to manipulate the global variables, thus transitioning the FSM from one state to another. An action is performed if, and only if the corresponding guard is evaluated to be *TRUE*.
- An *initial state* is defined by a special rule with an action which is executed only once, right before the FSM state exploration begins. This allows to explicitly define an initial state of the FSM.

For model verification purposes, Murphi also adds to the FSM a fourth component: *invariants*. The invariants are a set of logic expressions which define a set of *safe* states of FSM.

A state is only considered *safe* iff all invariants evaluate to *TRUE* in this state. With that, model verification is performed by exploring all reachable states, starting with a given initial state. The state space exploration is performed by applying all possible rules to all states that have already been reached with standard algorithms such as depth- or breadth-first search. For each newly discovered reachable state Murphi evaluates the invariants. Should any invariant evaluate *FALSE*, the system reports an error and prints out a set of states and transitions that led to the unsafe state. The state exploration continues forward only when all invariants evaluate to *TRUE* for the newly discovered state. In this work we apply model checking twice: one in detecting the $W \oplus X$ violation in the current Linux Kernel (Chapter 3) and another in validating our suggested solution (Chapter 4).

2.2 Linux kernel memory model

For proper modeling, it is important to understand how physical memory is being organized and mapped in Linux. On a typical 32-bit Linux system with the paging-based virtual memory enabled, the 4GB virtual memory space is split (Figure 2.1) between user space (bottom 3GB) and kernel space (top 1GB). In other words, the kernel space starts at address *PAGE_OFFSET*= $0xc0000000$ ($3GB$) and stretches up to the end of virtual address space $0xffffffff$ ($4GB-1$). We also notice that on an i386 architecture, physical memory is typically split onto two regions: *low memory* and *high memory*. The low memory region starts at address $0x00000000$ and ends at around $800MB$.[1] In Linux, this region is mapped directly into the kernel space, starting at *PAGE_OFFSET* and called *linear mapping*. Such linear mapping allows for simple translation between the physical and virtual addresses for all pages within this region. Specifically, the virtual address of any location within low memory can be obtained by adding *PAGE_OFFSET* value to the physical address. The reverse translation is performed by simply subtracting *PAGE_OFFSET* from the virtual address, without the need for page table lookup.

We point out that the linear mapping is established at kernel startup and persists all the way through system shutdown [5, 13, 24]. The virtual address space within this region is used to contain the following parts of the Linux kernel: BIOS32 services, kernel text, kernel read-only data, kernel read-write data and dynamic memory allocations that require contiguous physical pages (e.g., through *kmalloc()* call). Kernel initialization routines are also loaded here and later released as free memory.

Assuming the total amount of physical memory installed in the system is greater than the maximum size of low memory for the current kernel configuration, the high memory lays in the physical address space immediately following the low memory. However, unlike low memory,

[1] The exact value depends on the physical memory size and other compile-time and run-time kernel configuration.

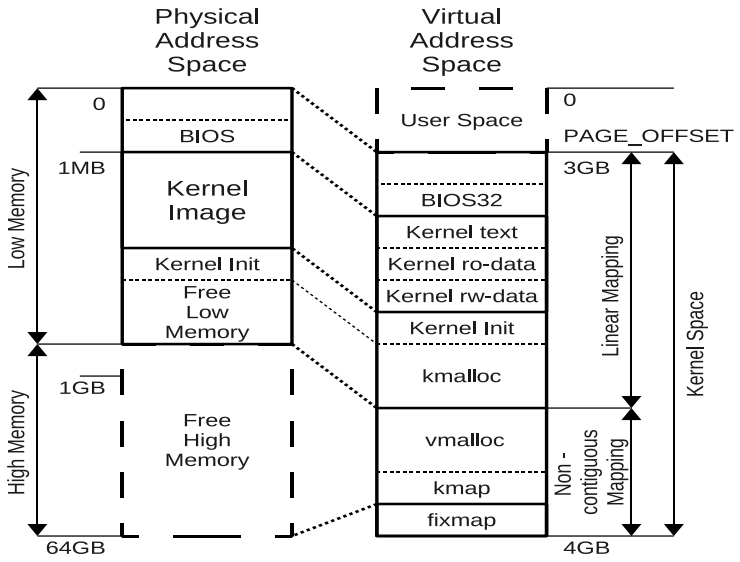


Figure 2.1: Typical Linux memory mapping

high memory is only mapped into kernel space when needed. There are two basic mechanisms through which this memory is mapped: *vmalloc()* and *kmap()*. The major difference between the two is that the former is used for long-term allocation of non-contiguous physical memory into contiguous virtual address space (e.g., for loadable kernel module allocation), while the latter is used strictly for short-term access to physical pages located in the high memory. Another difference is that vmalloc() may allocate pages from either high or low memory, while kmap() is strictly used for high memory only. Also, when vmalloc() allocates a page from low memory, it creates an *alias*: the same physical page will be mapped into the kernel virtual address space twice, one time in linear area and another one in vmalloc() area. The problem with such aliasing is that memory management subsystem has to ensure consistency of page attributes between all of the aliases. In the context of this work, both vmalloc() and kmap() areas play the same role: mapping non-contiguous physical pages into the kernel address space. Therefore, we treat them as the same in our model.

There is another memory area in Linux kernel called *fixmap*, which is located at the very end of the address space and mainly used by kernel to establish reserved, pre-defined fixed mappings such as PCI control registers and other memory-mapped services. Similar to the linear mapping, the fixmap region is established at kernel startup and persists all the way until system shutdown. Since this area does not represent an unique type of mapping that is distinct form the ones described above, we do not explicitly include it in the model.

— page protection attributes

```
pgprot_t: record
    w: boolean;
    x: boolean;
end;

-- page table entry
-- (mem_index is the memory frame number: [1..mem_size])
pte_t: record
    mapped: boolean;
    prot: pgprot_t;
    addr: mem_index;
end;

-- single-level page table
-- (page_index is the page table index: [1..pt_size])
pg_table_t: array[page_index] of pte_t;

-- physical memory frames
frame_t: record
    reference_count: 0..pt_size;
    prot: pgprot_t;
end;
frame_table_t: array[mem_index] of frame_t;
```

Note all modern multi-task operating systems rely on hardware support to provide virtual memory (and the $W \oplus X$ is enforced only when the virtual memory is enabled). In order to accurately model hardware support, we therefore include an abstraction of hardware memory subsystem in our model. Specifically, our model covers the mapping from physical memory to virtual memory as well as the related page tables and access flags. Our model is based on a single-level abstraction of the i386 family paging mechanism [2]. In other words, the mapping of virtual addresses to physical pages is modeled by a flat page table array (*pg_table_t*).[2] Each entry (*pte_t*) of that array corresponds to a virtual page (if *mapped*) and holds related attributes such as physical address (*addr*) it is currently mapped to and access permissions (*prot*). In our model we use *pgprot_t* type for page attribute tracking. This type defines two page access attributes: "write" and "execute". Since kernel pages are not swappable and all mapped pages are always readable, we do not model the "read" and "present" flags. Also, since this model is only concerned with kernel space, we do not model the *user/supervisor* flag. For the physical memory, due to the aliasing, we model them as an array (*frame_table_t*) of physical memory frames, with each physical page (*frame_t*) holding a reference count (*reference_count* or the

[2]Multiple levels of page tables are not necessary as they only allow to map large numbers of pages more efficiently and do not introduce any additional qualities related to this work. The same also stands true for the "large pages" introduced by Page Size Extension (PSE) technology [2].

number of mapped virtual pages) and a most permissive set (*prot*) of the attributes derived from all the virtual aliases pointing to it. As a result, the global definition of the page table array and the physical memory array define the possible *states* in our model.

Based on the above abstraction, our model then captures the specifics of the initial state of Linux kernel. As mentioned earlier and shown in Figure 2.1, the initialization of Linux kernel memory involves three main parts: kernel linear mapping, static kernel image mapping, and mapping of BIOS32 services. Accordingly, our model represents them by establishing three basic properties as part of the model's *initial state*: ***S1*** - *linear mapping*, ***S2*** - *static kernel mapping*, and ***S3*** - *BIOS32*. The detailed initialization sequence can be found in Appendix A.

After that, we further obtain the transition rules in our model. In particular, based on the Linux kernel source code and our domain knowledge, we identify and extract a number of kernel function routines or application program interfaces (APIs) that are used to affect the kernel memory mapping. Some of them are:

- *map_vm_area()* maps a physical page to a virtual address.
- *static_protections()* ensures that pre-defined areas of kernel's virtual address space always have correct attributes (kernel code should stay executable, kernel data readable and so on.)
- *cpa_process_alias()* checks all mapped aliases for a given physical page frame and updates them as necessary.
- *__change_page_attr_set_clr()* receives a block request for memory attribute change and translates it into a series of attribute and alias check calls for each individual page.
- *__change_page_attr()* executes attribute change for the individual pages.

To represent them, we derive a set of basic rules to capture their behavior especially when they perform memory mapping, re-mapping or change memory page attributes. By doing so, we avoid the need of understanding specific memory use-cases such as kernel module loading or unloading (as it is already captured with these APIs). In our model, we have three key transition rules and use them in our Murphi-based FSM description.

- ***T1*** - *Set*: This transition rule sets W and/or X flags for a given page table entry.
- ***T2*** - *Clear*: This transition rule clears W and/or X flags for a given page table entry.
- ***T3*** - *Map*: This transition rule changes the mapping between a physical frame and a virtual page.

```
ruleset i: page_index do
  ruleset x: boolean do
    ruleset w: boolean do

        -- set W and/or X for virtual page i
        rule "T1: Set"
                true ==>
                begin
                        set_mem_perm(i, 1, w, x);
                end;

        -- clear W and/or X for virtual page i
        rule "Clear"
                true ==>
                begin
                        clr_mem_perm(i, 1, w, x);
                end;
    end;
  end;
end;

-- map virtual address va to physical address pa with prot attributes
ruleset pa: frame_index do
  ruleset va: vmalloc_index do
    ruleset x: boolean do
      ruleset w: boolean do

        rule "T3: Map"
                true ==>
                var
                        prot: pgprot_t;
                        ret: boolean;
                begin
                        prot.x := x;
                        prot.w := w;
                        ret := map_vm_area(va, pa, prot);
                end;
      end;
    end;
  end
end;
```

Chapter 3

Analysis

3.1 Defining and applying invariants

After obtaining the abstract model of Linux kernel, we further define the invariants to analyze possible Linux kernel states. Since our focus in this work is on the $W \oplus X$ enforcement, we established the following properties through invariants in the model:

- ***P1***: Kernel code should always be executable and read-only.
- ***P2***: Kernel data should always be non-executable, the read-only kernel data should remain read-only, and read-write kernel data should always be writable.
- ***P3***: No page will be writable and executable at the same time in order to not violate $W \oplus X$.
- ***P4***: All virtual aliases of each physical page should have consistent access permissions.

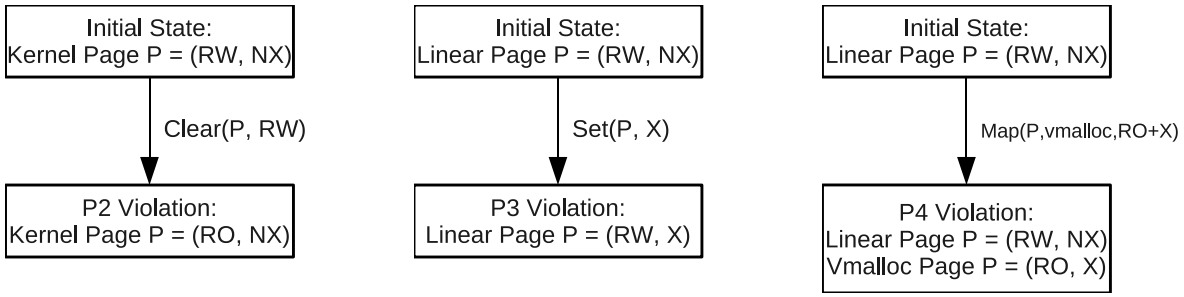
```
invariant "P1: Kernel Code"
  true -> kernel_code_rox() = true;

invariant "P2: Kernel RO data"
  true -> kernel_rodata_ronx() = true;

invariant "P2: Kernel RW data"
  true -> kernel_rwdata_rwnx() = true;

invariant "P3: W xor X"
  true -> w_and_x() = false;

invariant "P4: Alias consistency"
  true -> page_alias_matching() = true;
```



(a) P2 violation: kernel read/write data

(b) P3 violation: W and X

(c) P4 violation: inconsistent aliases

Figure 3.1: Property violations

More specifically, the P1 invariant is necessary to keep kernel code executable as non-executable kernel code will lead to an immediate system crash. The P2 invariant ensures that read-only kernel data that holds constants cannot be modified and that read-write data is always accessible. It also ensures that static kernel data cannot be executed. The P3 invariant states that any given page cannot be writable and executable at the same time. This property is necessary to prevent code injection and is focal point of this work. The P4 invariant is necessary to prevent code injection by accessing an alias which is mapped into a different virtual address with different access permissions.

With invariants in place, the model checker is able to provide us with examples of possible transitions if they lead to an unsafe state that violates $W \oplus X$ policy. Our experience with the model checker indicates that there is no P1 violation in the current Linux design and implementation. However, it reports violations for all other three invariants (Figure 3.1). Since our model relies on simplified abstraction of the kernel code, some false-positives are possible. Therefore, we confirm each of the reported violations by ensuring that the condition detected in the model manifests itself on the live Linux system.

3.2 P2 violation

The violation of P2 arises when kernel read-write data region is set as read-only. The problem stems from the fact that *static_protections()* does not check for the correctness of new access flags set for the kernel *read-write data* region. While this issue does not directly allow for code injection, setting read-write data (Figure 3.1a) as read-only provides a vector for a denial of service attack. The reason is that the kernel assumes that its read-write data is always writable and is not equipped to handle this situation. This issue has been confirmed by a kernel crash, which immediately follows the call of *set_pages_ro()* for the read-write data region. During the investigation of this issue, we have also identified and confirmed a kernel bug that can be used

```
---[ Kernel Mapping ]---
0xc0000000-0xc0200000          2M     RW             GLB x  pte
0xc0200000-0xc0600000          4M     ro         PSE GLB x  pmd
0xc0600000-0xc0843000       2316K     ro             GLB x  pte
0xc0843000-0xc0a00000       1780K     RW             GLB x  pte
0xc0a00000-0xf7800000        878M     RW         PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW             GLB NX pte
0xf79fe000-0xf7a00000          8K                          pte
0xf7a00000-0xf8000000          6M                          pmd
0xf8000000-0xf81fe000       2040K                          pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW     PCD     GLB NX pte
0xf81ff000-0xf8200000          4K                          pte
                     [ . . . ]
0xf8247000-0xf824a000         12K     RW             GLB x  pte
0xf824a000-0xf824c000          8K                          pte
0xf824c000-0xf824d000          4K     RW             GLB x  pte
0xf824d000-0xf824f000          8K                          pte
0xf824f000-0xf8274000        148K     RW             GLB NX pte
0xf8274000-0xf8276000          8K                          pte
0xf8276000-0xf8278000          8K     RW             GLB x  pte
0xf8278000-0xf827a000          8K                          pte
0xf827a000-0xf827b000          4K     RW             GLB x  pte
0xf827b000-0xf827d000          8K                          pte
0xf827d000-0xf8296000        100K     RW             GLB NX pte
0xf8296000-0xf8298000          8K                          pte
                     [ . . . ]
```

(a) Vanilla kernel

```
---[ Kernel Mapping ]---
0xc0000000-0xc00fb000       1004K     RW             GLB NX pte
0xc00fb000-0xc00fd000          8K     ro             GLB x  pte
0xc00fd000-0xc0200000       1036K     RW             GLB NX pte
0xc0200000-0xc0600000          4M     ro         PSE GLB x  pmd
0xc0600000-0xc068e000        568K     ro             GLB x  pte
0xc068e000-0xc0844000       1752K     ro             GLB NX pte
0xc0844000-0xc0a00000       1776K     RW             GLB NX pte
0xc0a00000-0xf7800000        878M     RW         PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW             GLB NX pte
0xf79fe000-0xf7a00000          8K                          pte
0xf7a00000-0xf8000000          6M                          pmd
0xf8000000-0xf81fe000       2040K                          pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW     PCD     GLB NX pte
0xf81ff000-0xf8200000          4K                          pte
                     [ . . . ]
0xf821a000-0xf821d000         12K     ro             GLB x  pte
0xf821d000-0xf821e000          4K     ro             GLB NX pte
0xf821e000-0xf8220000          8K     RW             GLB NX pte
0xf8220000-0xf8222000          8K                          pte
0xf8222000-0xf8223000          4K     RW     PCD     GLB NX pte
0xf8223000-0xf8227000         16K                          pte
0xf8227000-0xf8228000          4K     ro             GLB x  pte
0xf8228000-0xf8229000          4K     ro             GLB NX pte
0xf8229000-0xf822b000          8K     RW             GLB NX pte
                     [ . . . ]
```

(b) Patched kernel

Figure 3.2: Dumping kernel page tables (kernel version 2.6.33)

to apply improper page attributes to a memory region when large (2MB) pages are used in the kernel space (Chapter 4).

3.3 P3 violation

The violation of P3 happens in two scenarios. The first one occurs in the initial state. More specifically, the way the BIOS32 is mapped into the kernel space during the initialization directly contradicts the P3 invariant. Note that the BIOS32 services contain executable code and, therefore, should be set as read-only. However, the Linux kernel simply indiscriminately maps the whole BIOS region into the kernel address space as writable and executable. While the actual BIOS code is typically stored in ROM and cannot be overwritten, such mapping still provides an opportunity for data execution or code modification. In a typical page table dump of the latest Linux kernel (version 2.6.33) shown in Figure 3.2a, this violation manifests itself as a set of $RW + X$ pages within the range *0xc0000000 - 0xc2000000*.

The second scenario is related to the original memory management interface that allows for pages to be mapped as writable and executable at the same time, thus violating P3 (Figure 3.1b). The source of the problem is the absence of any access permission verification system for memory pages outside the static kernel image in the default memory management interface, such as when a kernel module is being loaded. To confirm this violation, we loaded several modules and inspected the vmalloc() area of the page tables for $W \oplus X$ violations. We found that Linux kernel indeed does not enforce $W \oplus X$ for mappings in the non-contiguous memory region (Figure 3.2a).

3.4 P4 violation

This violation of P4 happens when a page from *low memory* region is being mapped into vmalloc() area as executable. This would typically happen during the memory allocation for a loadable kernel module on a system that does not have any *high memory* available for allocation. A module loader would use vmalloc() to load all module sections, including code with execute permissions. However, the physical page is already mapped once in the linear mapping space with $RW + NX$ permissions, creating an opportunity for code injection. We were able to re-create this behavior consistently by loading Linux kernel in a virtual machine with only 128MB of RAM, forcing Linux to allocate all pages from low memory, thus creating aliases for each of vmalloc() allocations.

Chapter 4

Improvements and Prototype

Based on the findings described in the previous section, we propose a few improvements to the Linux Kernel. Our improvement involves the changes to the initial states (S1, S2, and S3) as well as the transition rules (T1, T2, and T3) such that no unsafe states will be reached from the revised FSM description (Appendix A). For debugging and verification purposes, we have developed five self-contained and independent patches (in total 703 source code of lines) to Linux Kernel, four of which have been submitted to LKML [18, 19, 20, 21] and are currently in the process of being integrated into the mainline kernel. The remaining one is still being evaluated for suitability in the mainline kernel.

In the development of these patches, based on the counter-examples reported from Murphi, we revised the memory management subsystem in Linux. Interestingly, we found that there are two distinct levels of abstraction interfaces to manage Linux kernel memory. The low-level interface is primarily tasked with direct manipulation of page table entries, and the high-level interface is used to abstract page table layout and provide functions like *set_memory_nx()*. At first glance, the low-level interface seems a better target to address previous violations. However, the following significant shortcomings in its design complicate such approach. Therefore, we chose the high-level interface as a target for our solution.

- First, the low-level functions (e.g., *pte_mkexec()*) are intended as simple data manipulation primitives which cannot fail. Our experience indicates that any deviation from this assumption would yield unpredictable results, as the current kernel is not yet designed to handle such failures gracefully.

- Second, these functions operate on instances of *pte_t* type, which by itself does not guarantee that the instance is in fact part of active or future page tables. This, in turn, means that we would either need to check the address of each *pte_t* in question, or enforce $W \oplus X$ on all instances of *pte_t*. Neither of these options is desirable: the former creates

a significant performance overhead while the latter one introduces unwanted side effects into all intermediate transformations of *pte_t* instances.

4.1 Fixing P2 violation: preserving the *write* access on kernel read-write data

Our first patch fixes the violation of P2 invariant. More specifically, as discussed earlier, this problem is caused by the inability of *static_protections()* to preserve the *write* access to kernel *read-write data* (including the *BSS* section). The fix is a straightforward one. However, in the process of re-mapping our model back to the original Linux kernel source code, we discovered another implementation bug in the function routine *try_preserve_large_pages()*. Specifically, this function incorrectly processes access permission change requests for areas that start on a boundary of a large page (i.e., $2M$), but are smaller than the page itself. This leads to a possibility of setting improper access flags to the memory area located within the same large page, but immediately after the requested one. Specifically, this problem manifested itself by kernel read-write data becoming read-only simply because it was initially mapped within the same large page as kernel's read-only data. Accordingly, we propose two changes in the patch (that affects the file *arch/x86/mm/pageattr.c*): one is to allow *static_protections()* to preserve the *write* access for kernel's read-write data area and another one is to verify each small page within the large page for access flag compatibility [18]. The source code of the patch can be found in Appendix B.1.

4.2 Fixing P3 violation: removing mixed pages in kernel space

Our next three patches address P3 violation, namely the presence of mixed code and data pages in kernel space. Based on our model checking results, our investigation maps the related transition rules that lead to an unsafe state back to the involved kernel routines. By doing so, we are able to identify three distinct sources: BIOS32, loadable kernel modules (LKMs), and static kernel image management.

BIOS32 As discussed in Chapter 3, the current Linux kernel improperly maps the entire BIOS area into the kernel space as $RW + X$. (Note the BIOS code itself is typically located in read-only memory or ROM.) To resolve this issue, we implemented a patch that dynamically maps BIOS32 services into the kernel space. Based on related BIOS32 documents [4] and [27], it requires at most two pages to be executable per BIOS32 service, and none of them are expected to be writable. As such, a dynamic service mapping of BIOS32 services can be established at the time of service discovery, and with appropriate (and $W \oplus X$-compliant) access permissions. As part of our patch, we added the BIOS32 service mapper to *pci/pcbios.c* and removed unnecessary

protection for the area of physical memory under 2MB from *arch/x86/mm/pageattr.c*. Also, we revised the file *mm/init_32.c* to properly report kernel text addresses [19] because of the BIOS32 changes. Appendix B.2 contains the implementation details of this patch.

LKMs The second source of violating $W \oplus X$ is located in the support of LKMs. In particular, since all vmalloc() allocations default to "data" access mode ($RW + NX$), the only source of mixed pages in this area is LKMs as their code is explicitly marked as "executable". More specifically, dynamic kernel linker allocates each loadable module in two parts: module init and module core. The init part of the module will be discarded after initialization, while core will stay resident in the kernel. In order to minimize a module's footprint, the linker chooses the minimum amount of spacing necessary between each of the module's sections - just enough to accommodate necessary section alignment, which introduces mixed kernel pages. Accordingly, our patch allocates module sections in three groups: text, read-only data, read-write data and further adjusts the linker to align each of the groups on a page boundary, ensuring that each page contains only sections from the same group. Next, our patch assigns a set of appropriate access permissions to all pages of each group as follows: read-only for text and read-only data, non-executable for read-only data and read-write data. As this patch will inevitably introduce additional memory consumption (Chapter 5), we create a compile-time option (i.e., *CONFIG_DEBUG_SET_MODULE_RONX*) to turn on or off the functionality of this patch as needed [21]. Refer to Appendix B.3 for the specific implementation.

Static kernel image Similar to the support of LKMs, our second patch includes the code to split all sections of the static kernel image into three groups: text, read-only data, and read-write data. Specifically, our patch addresses necessary group alignment by modifying the related linker script (*kernel/vmlinux.lds.S*) and assigns proper access permissions to the pages of each group at the end of kernel initialization (*mm/init.c* [20]). The functionality of this patch is always enabled. Source code of this patch can be found in Appendix B.4.

4.3 Fixing P4 violation: disallowing memory aliasing with permission conflicts

The remaining patch addresses the P4 violation. In particular, as we have mentioned in Chapter 2.2, memory alias handling is being implemented in the kernel in *cpa_process_alias()* function. However, the way it handles page aliases specifically excludes the case of an NX update.

```
/* No alias checking for _NX bit modifications */
checkalias = (pgprot_val(mask_set) | pgprot_val(mask_clr)) != _PAGE_NX;
```

To enforce $W \oplus X$, our patch needs to modify the alias handling routine such that it will propagate changes of all access flags to all aliases. This can be achieved by setting *checkalias*

= *1* for all page attribute modifications.

```
static inline pgprot_t
  process_WxorX_violation(pgprot_t prot,
     unsigned long address, unsigned long pfn)
{
    /*
     * We can Oops or Panic here if needed.  But for now we just print out an
         error message.
     */
    printk(KERN_ERR "(W xor X) violation: " "VA=0x%lx, PFN=0x%lx", address, pfn);
    /* Set NX, just in case */
    pgprot_val(prot) |= _PAGE_NX;
    return prot;
}
```

In addition, our patch implemented a helper routine *process_WxorX_violation()* (a part of *mm/pageattr.c*) for the strict enforcement of $W \oplus X$ property. Particularly, this function will be called for each page that is about to violate $W \oplus X$ property, before the new protection attributes can be applied. Because of the presence of aliasing, this routine is called for any inconsistency of memory protection attributes in aliases as well. If there are conflicting attributes, this function will by default set NX flag for all related pages and logs an error message detailing the associated virtual and physical addresses. See Appendix B.5 for details.

Chapter 5

Evaluation

5.1 Soundness

Our improvements aim to make the Linux kernel compliant to the $W \oplus X$ property by guaranteeing exclusivity of write and execute page access. This means that all of the kernel code that follows the interfaces in place will automatically be compliant with $W \oplus X$ policy without additional modifications. To rigorously verify the $W \oplus X$ compliance, we revise the previous FSM description to reflect our patches (Chapter 4). Also, in order to avoid unnecessary state explosion, we chose a minimal configuration where we only modeled a system that contains one virtual page of each type: kernel text, kernel read-only data, kernel read-write data, linear mapping. To allow for variability, we use two pages in non-contiguous mappings, and model the physical memory with one more page frame than the total size of the virtual address space. In addition, the model contained a proposed memory interface, a full set of rules, and invariants to establish and monitor the $W \oplus X$ property (Appendix A). The model checker examined 3402 states and 381024 rules without detecting any violations.

In order to further confirm the validity of our approach in a real system, we used a minimal Ubuntu Server 8.04.4 LTS [33] system that runs the latest vanilla Linux 2.6.33 kernel [16]. Note the vanilla Linux kernel has been compiled with Generic Ubuntu configuration, and then booted and inspected. In Figure 3.2a, we show the virtual memory layout in the vanilla Linux kernel. It shows that while all necessary elements of code and data separation are indeed present, they are not applied in a consistent manner. Specifically, RO and NX flags are used sparsely. As shown from the detailed kernel page table dump (Figure 3.2a), it fails to establish $W \oplus X$ property.

In comparison, we applied our patch set to the same kernel and repeated the inspection. A clear difference can be observed on Figure 3.2b: we have successfully eliminated all pages with mixed access. Specifically, the following changes are noteworthy:

```
fixmap  : 0xfff1e000 - 0xfffff000   ( 900 kB)
pkmap   : 0xffa00000 - 0xffc00000   (2048 kB)
vmalloc : 0xf81fe000 - 0xff9fe000   ( 120 MB)
lowmem  : 0xc0000000 - 0xf79fe000   ( 889 MB)
  .init : 0xc08a3000 - 0xc0916000   ( 460 kB)
  .data : 0xc068d32c - 0xc08a29e8   (2133 kB)
  .text : 0xc0200000 - 0xc068d32c   (4660 kB)
```

(a) Vanilla kernel

```
fixmap  : 0xfff1e000 - 0xfffff000   ( 900 kB)
pkmap   : 0xffa00000 - 0xffc00000   (2048 kB)
vmalloc : 0xf81fe000 - 0xff9fe000   ( 120 MB)
lowmem  : 0xc0000000 - 0xf79fe000   ( 889 MB)
  .init : 0xc08a3000 - 0xc0916000   ( 460 kB)
  .data : 0xc068e000 - 0xc08a3000   (2132 kB)
  .text : 0xc0200000 - 0xc068e000   (4664 kB)
```

(b) Patched kernel

Figure 5.1: Virtual kernel memory layout

- a 2Mb area between *0xc0000000* and *0xc0200000* is now marked as $RW + NX$, except for two $RO + X$ pages *0xc00fb000* - *0xc00fd000* reserved for BIOS32 services.
- Static Kernel image (*0xc02000000* - *0xc08a3000*) is clearly partitioned in three sections: $RO + X$ code (*0xc02000000* - *0xc068e000*), $RO + NX$ read-only data (*0xc08a3000* - *0xc0844000*), and $RW + NX$ read-write data (*0xc0844000* - *0xc08a3000*).
- Loadable Kernel Modules (see addresses *0xf821a000* - *0xf8220000* and *0xf8227000* - *0xf822b000*) are now clearly split into three parts: $RO + X$ code, $RO + NX$ read-only data, and $RW + NX$ read-write data.

The detailed kernel page table dumps can be found in Appendix C.1.1 and Appendix C.2.1.

5.2 Performance and memory overhead

To evaluate the performance overhead introduced, we measured its runtime overhead with three tasks: UnixBench 5.1.2 [32] (index test group, SMP and Uniprocessor configurations), Linux kernel compilation, and compression time of 10GB random data stream. All tests have been performed on Ubuntu Server 8.04.4 LTS with Linux 2.6.33 compiled for i386 architecture in standard Ubuntu Server configuration except for PAE enabled, and XEN disabled. Our test platform is a Gigabyte MA78G-DS3HP system (AMD RS780/SB700 chipset) with dual-core AMD Athlon 4850e processor (family 15, model 107, stepping 2) and 8GB of PC2-6400 RAM in dual-channel configuration (4x2GB, CL5). The summary results are shown in Table 5.1, while full test output can be found in Appendix C.1.3 and Appendix C.2.3.

Our results indicate that our patch set does not affect overall system performance in a measurable way. In particular, the first four patches (related to static kernel image, LKM, and BIOS32) do not introduce any additional performance overhead as all additional work are mainly performed at compile-time. Though there is a slight overhead incurred during the kernel/module initialization, it does not affect the runtime performance after initialization. Among the five patches, the only patch that involves run-time penalty is the compliance checking of $W \oplus X$ for each kernel page table update (i.e., in the helper routine *process_WxorX_violation()*).

Table 5.1: Run-time $W \oplus X$ enforcement overhead

Benchmark	Vanilla	Patched	Overhead%
UnixBench UP (index)	737.88	741.8	0.53%
UnixBench SMP (index)	1317.13	1310.88	-0.47%
Kernel Compilation (seconds)	1712	1725	0.76%
Gzip test (seconds)	2890	2873	-0.59%

Next, we evaluate the memory overhead introduced into the kernel by our patches. We first investigate the difference in the size of the static kernel image and memory allocated to individual modules. As can be seen on Figure 5.1, the size of the static kernel image has increased from $6793KB$ ($4660KB$ text + $2133KB$ data) to $6796KB$ ($4664KB$ text + $2132KB$ data). This increase constitutes a mere 0.04%, and is thus not significant.

We also used the same system to load 44 kernel modules of varying sizes (Appendix C.1.2 and Appendix C.2.2). Due to the fact that our patch set affects the layout of different module sections, the total size of these 44 modules (reported by running the *lsmod* command) is increased from $1,265,502B$ to $1,493,138B$ (an increase of 22.01%). Note that the module sizes reported by *lsmod* could be a little misleading, as they do *not* reflect the fact that kernel allocates memory for modules at the page granularity. This means that the true measure of module memory consumption should be the memory (in terms of pages) allocated to the module, not the module size. With that, if we can take the whole-page allocation into account, the memory overhead is increased from 322 pages (size of $4K$) to 382 pages (an increase of 22.17%). We believe such memory overhead is moderate and within an acceptable range for modern server and desktop systems, especially considering the current price drop of physical memory. In the meantime, we also recognize that such increase in memory consumption might be unwanted in certain memory-constrained applications, such as embedded systems. Because of that, our patch is provided as a compile-time option that may be enabled or disabled as needed.

Chapter 6

Discussion

As with many other real-world protection systems, our approach comes with a few limitations. First of all, we assume that all binaries (static kernel image and LKMs) are trusted. Establishment of such trust may be conducted by means of kernel/driver signing [12, 28, 30, 31]. This falls outside the scope of this work as our main focus here is to analyze Linux kernel memory protection to detect and fix any $W \oplus X$ violation. Second, it is important to note that while providing $W \oplus X$ is helpful to block code injection attacks, $W \oplus X$ itself does not prevent other types of attacks, such as "return-into-libc" [14]. Third, since our modeling is based on the correctness of the internal kernel APIs, any code inside the function that modifies page tables directly will not be prevented from doing so. Finally, although we establish a $W \oplus X$ property in the Linux kernel, there are a few exceptions that we had to consider for the implementation of our patches. For example, a special accommodation had to be made for kernel's built-in function tracing facility - *ftrace* [1]. The function tracing facility essentially requires dynamically modifying kernel code, which is in conflict with our $W \oplus X$ enforcement. As a result, we have to allow a short window of $W \oplus X$ violation while ftrace is in use to place its trace points (that requires modifying kernel's code at run time).

Chapter 7

Related Work

7.1 Model checking for improved security

The first category of related work includes recent efforts that leverage model checking to improve systems security. For example, Mitchell et al. [23] applied model checking to successfully verify the correctness of (and find bugs in) security protocol specifications. Chen et al. [8] utilized model checking to analyze or demystify the confusing setuid system calls. Others [6, 7, 11] have used software model checking and static analysis to find a general class of bugs in source code. In contrast, our focus is on the analysis and verification of $W \oplus X$ enforcement for Linux kernel memory protection.

7.2 $W \oplus X$ enforcement

The second category of related work aims at enforcing $W \oplus X$ as an effective defense against code injection attacks. For example, SecVisor [31] and NICKLE [30] use custom hypervisors to enforce code protection and data non-execution. Such protection is achieved through effectively separating code and data address spaces. Both methods make it impossible to modify code and to execute data without going through the special authentication mechanisms controlled by a hypervisor. Note that even with hardware-based full virtualization support, they inevitably lead to significant performance degradation, as policy enforcement requires additional management activities that consume extra clock cycles and cause cache pollution. Others take advantage of standard hardware protection features in the most straight-forward manner and introduce minimal impact. For examples, both PaX [26] and ExecShield [34] make use of standard memory protection features found in most architectures to achieve $W \oplus X$ property. Shared with our approach, these patches separate memory pages into two categories: code and data. Code pages are set as $RO + X$, while data as $RW + NX$. However, our approach is different from them

in that we use a model checking approach to systematically analyze the $W \oplus X$ protection in the Linux kernel memory space while PaX and ExecShield mainly concentrate their efforts on userspace application protection. Since our solution is based on the model-checking approach, it can be formally verified.

7.3 Other code injection defense mechanisms

The third category of related work contains other approaches to defend against code injection attacks. For example, two other notable ways in this category include Address Space Layout Randomization (ASLR) [25, 26] and Instruction Set Randomization [15, 3]. ASLR is based on the idea of randomization of all major components within the application address space. This typically involves introduction of random offsets in the layout of all major sections of the primary executable and the libraries it requires at link-time. This type of protection is already included in the mainline kernel and used exclusively for userspace. Instruction set randomization takes a somewhat different approach by randomizing the actual machine instruction set. This is achieved by creating a virtual machine with unique instruction encoding for each run. Compatibility with pre-compiled binaries is established by load-time binary translator, which converts the code from well-known "generic" instruction encoding to the encoding used in the specific virtual machine. Such approaches are impractical since dynamic instruction sets are not supported by any modern hardware and software-based emulation likely introduces prohibitive performance overhead.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented a model checking-based approach to analyze the $W \oplus X$ protection in the Linux kernel space. Our modeling has led to the discovery of several real problems in the current Linux kernel design and implementation. Based on the model checking results, we have accordingly developed five kernel patches to fix them. Four of them have been accepted for integration into the mainline Linux kernel. Our evaluation with these patches indicate that they involve minimal changes and incur negligible performance overhead to the Linux kernel.

There are, however, still a number of natural directions for future research. For example, although this work focuses on 32-bit architecture and is compatible with the 64-bit architecture, the $W \oplus X$ property is not maintained when running in 64-bit mode due to the differences in kernel memory layout. An extension of this work would be necessary to provide a set of patches for strict $W \oplus X$ enforcement in 64-bit mode. Another possible extension would be to design and implement an interface for tight integration with existing hypervisors like Xen [35] and KVM [17]. For elevated high-assurance, the enforcement of $W \oplus X$ can be pushed by out interface to the underlying hypervisor. Finally, expansion of formal verification and $W \oplus X$ enforcement to userspace presents itself as another attractive direction for the future research.

REFERENCES

[1] A look at ftrace. `http://lwn.net/Articles/322666/`.

[2] Advanced Micro Devices. *AMD64 Architecture Programmers Manual Volume 2: System Programming*, 3.14 edition, September 2007.

[3] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *CCS '03: Proceedings of the ACM Computer and Communications Security Conference*, 2003.

[4] Thomas C. Block. *Standard BIOS 32-bit Service Directory Proposal*. Phoenix Technologies Ltd., 0.4 edition, June 1993.

[5] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, third edition, 2005.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI'08: Proc. of USENIX OSDI 2008*, December 2008.

[7] Hao Chen and David Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *CCS'02: Proc. of ACM Conference on Computer and Communications Security*, 2002.

[8] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Security '02: Proceedings of the 11th conference on USENIX Security Symposium*, 2002.

[9] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Intel Corp., 2006. Publication number 253668.

[10] David L. Dill. The murphi verification system. `http://eprints.kfupm.edu.sa/70602/1/70602.pdf`, 1996.

[11] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor Research Area. Technical report, June 2008.

[12] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the 19th Symposium on Operating System Principles*, October 2003.

[13] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. Prentice Hall, May 2004.

[14] Ralf Hund, Thorsten Holz, and Felix Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Security '09: Proceedings of the 18th conference on USENIX Security Symposium*. USENIX Association, 2009.

[15] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the ACM Computer and Communications Security Conference*, 2003.

[16] The Linux Kernel Archives. `http://www.kernel.org`.

[17] Kernel based virtual machine. `http://www.linux-kvm.org/page/Main_Page`.

[18] Siarhei Liakh and Xuxian Jiang. [1/4,tip:x86/mm] correcting improper large page preservation. https://patchwork.kernel.org/patch/90045/, 2010.

[19] Siarhei Liakh and Xuxian Jiang. [2/4,tip:x86/mm] set first mb as rw+nx. https://patchwork.kernel.org/patch/90048/, 2010.

[20] Siarhei Liakh and Xuxian Jiang. [3/4,tip:x86/mm] nx protection for kernel data. https://patchwork.kernel.org/patch/90046/, 2010.

[21] Siarhei Liakh and Xuxian Jiang. [4/4,tip:x86/mm] ro/nx protection for loadable kernel modules. https://patchwork.kernel.org/patch/90047/, 2010.

[22] Evangelos P. Markatos Michalis Polychronakis, Kostas G. Anagnostakis. An empirical study of real-world polymorphic code injection attacks. In *LEET '09: Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats*. Usenix Association, April 2009.

[23] John Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-State Analysis of SSL 3.0. In *Security '98: Proceedings of the 7th conference on USENIX Security Symposium*, 1998.

[24] Abhishek Nayani, Mel Gorman, and Rodrigo S. de Castro. Memory management in linux - desktop companion to the linux source code. `http://www.ecsl.cs.sunysb.edu/elibrary/linux/mm/mm.pdf`, May 2002.

[25] Pax aslr. `http://pax.grsecurity.net/docs/aslr.txt`.

[26] Pax noexec. `http://pax.grsecurity.net/docs/noexec.txt`.

[27] PCI Special Interest Group. *PCI BIOS Specification*, 2.1 edition, August 1994.

[28] Siani Pearson. Trusted computing platforms, the next security solution. Technical report, HP Laboratories, November 2002.

[29] Ryan Riley, Xuxian Jiang, and Dongyan Xu. An architectural approach to preventing code injection attacks. In *DSN '07: Proccedings of The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2007.

[30] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Springer-Verlag, 2008.

[31] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM, 2007.

[32] Jon Tombs, Ben Smith, Rick Grehan, Tom Yager, David C. Niemi, and Ian Smith. Unixbench-5.1.2. `http://code.google.com/p/byte-unixbench/`.

[33] Ubuntu. `http://www.ubuntu.com/`.

[34] Arjan van de Ven. Limiting buffer overflows with execshield. *Red Hat Magazine*, July 2005.

[35] Xen hypervisor. `http://www.xen.org`.

APPENDICES

Appendix A

Murphi Model Definition

```
-- Murphi model for (W xor X) patch verification for Linux kernel
-- by Siarhei Liakh <sliakh@ncsu.edu> and Xuxian Jiang <jiang@cs.ncsu.edu>
-- (C) NCSU 2010

const
    -- virtual address of the first page in kernel space
    page_offset: 1;

    -- size and virtual address definitions for
    -- different kernel mappings.
    -- *_start - address of the first page of the region
    -- *_end - address of the last page of the region

    bios_size: 1;
    bios_start: page_offset;
    bios_end: bios_start + bios_size - 1;

    kernel_text_size: 1;
    kernel_text_start: bios_end + 1;
    kernel_text_end: kernel_text_start + kernel_text_size - 1;

    kernel_ro_size: 1;
    kernel_ro_start: kernel_text_end + 1;
    kernel_ro_end: kernel_ro_start + kernel_ro_size - 1;

    kernel_rw_size: 1;
    kernel_rw_start: kernel_ro_end + 1;
    kernel_rw_end: kernel_rw_start + kernel_rw_size - 1;

    kernel_size: kernel_text_size + kernel_ro_size + kernel_rw_size;
    kernel_start: kernel_text_start;
```

```
    kernel_end: kernel_rw_end;

    kmalloc_size: 2;
    kmalloc_start: kernel_end + 1;
    kmalloc_end: kmalloc_start + kmalloc_size - 1;

    linear_size: bios_size + kernel_size + kmalloc_size;
    linear_start: page_offset;
    linear_end: linear_start + linear_size - 1;

    vmalloc_size: 2;
    vmalloc_start: linear_end + 1;
    vmalloc_end: vmalloc_start + vmalloc_size - 1;

    -- physical memory size
    mem_size: linear_size + vmalloc_size + 1;

    -- free physical memory
    free_size: mem_size - bios_size - kernel_size;
    free_start: bios_size + kernel_size;
    free_end: free_start + free_size - 1;

    -- page table size
    pt_size: page_offset + linear_size + vmalloc_size;

type
    -- following two types are used for counting
    page_count: 0 .. pt_size;
    frame_count: 0 .. mem_size;
    vmalloc_count: 0 .. vmalloc_size;
    free_count: 0 .. free_size;
    word: 0 .. pt_size + mem_size + 1;

    -- following types are used for indexing
    page_index: 0 .. (pt_size - 1);
    frame_index: 0 .. (mem_size - 1);
    vmalloc_index: vmalloc_start .. vmalloc_end;
    free_index: free_start .. free_end;

    -- page protection attributes
    pgprot_t: record
            w: boolean;
            x: boolean;
    end;
```

```
    -- page table entry
    -- (frame_index is the memory frame number: [0 .. (mem_size-1)])
    pte_t: record
        mapped: boolean;
        prot: pgprot_t;
        addr: frame_index;
    end;

    -- single-level page table
    -- (page_index is the page table index: [0 .. (pt_size-1)])
    pg_table_t: array[page_index] of pte_t;

    -- physical memory frames
    frame_t: record
        reference_count: page_count;
        prot: pgprot_t;
    end;
    frame_table_t: array[frame_index] of frame_t;

var
    pt: pg_table_t;

-- set page attributes from mask
function pgprot_set(prot:pgprot_t; mask:pgprot_t):pgprot_t;
var
    new: pgprot_t;
begin
    new.w := prot.w | mask.w;
    new.x := prot.x | mask.x;
    return new;
end;

-- clear page attributes from mask
function pgprot_clear(prot:pgprot_t; mask:pgprot_t):pgprot_t;
var
    new: pgprot_t;
begin
    new.w := prot.w & (!mask.w);
    new.x := prot.x & (!mask.x);
    return new;
end;

-- linear conversion of physical address to virtual
function virtual(pfn:frame_index):page_index;
begin
```

```
    return pfn + page_offset;
end;

-- linear conversion of virtual address to physical
function physical(va:page_index):frame_index;
begin
    return va - page_offset;
end;

function within(x:word; a:word; b:word):boolean;
begin
    return (x>=a) & (x<=b);
end;

procedure exception(pa:frame_index; va:page_index);
begin
    -- do not really do anything.
end;

function static_protections(va:page_index; prot:pgprot_t):pgprot_t;
var
    new:pgprot_t;
begin
    new := prot;
    alias
        pa : pt[va].addr;
    do
        -- all aliases for kernel text should stay RO
        if(within(pa, physical(kernel_text_start),
                physical(kernel_text_end)))
        then
            new.w := false;
        end;

        -- primary mapping of kernel text should stay X
        if(within(va, kernel_text_start, kernel_text_end))
        then
            new.x := true;
        end;

        -- all aliases for kernel ro-data shouls stay RO+NX
        if(within(pa, physical(kernel_ro_start),
                physical(kernel_ro_end)))
        then
            new.w := false;
```

```
            new.x := false;
        end;

        -- all aliases for rw-data should stay NX
        if(within(pa, physical(kernel_rw_start),
                physical(kernel_rw_end)))
        then
            new.x := false;
        end;

        -- primary mapping for kernel rw-data should stay RW
        if(within(va, kernel_rw_start, kernel_rw_end))
        then
            new.w := true;
        end;

        if(new.w & new.x)
        then
            exception(pa, va);
            new.x := false;
        end;
    end;

    return new;
end;

function change_page_attr(va:page_index; set:pgprot_t; clr:pgprot_t):boolean;
var
    new_prot: pgprot_t;
begin
    if(pt[va].mapped)
    then
        new_prot := pt[va].prot;
        new_prot := pgprot_clear(new_prot, clr);
        new_prot := pgprot_set(new_prot, set);
        new_prot := static_protections(va, new_prot);
        pt[va].prot := new_prot;
        return true;
    else
        return false;
    end;
end;

function process_alias(va:page_index; set:pgprot_t; clr:pgprot_t):boolean;
begin
```

```
    for i: page_index do
        if( (i != va) & (pt[i].addr = pt[va].addr))
        then
            if(!change_page_attr(i, set, clr))
            then
                return false;
            end;
        end;
    end;
    return true;
end;

function change_page_attr_set_clr(va:page_index; count:page_index;
            set:pgprot_t; clr:pgprot_t):boolean;
begin
    for i:=va to (va + count - 1) do
        if(!change_page_attr(i, set, clr))
        then
            return false;
        end;

        if(!process_alias(i, set, clr))
        then
            return false;
        end;
    end;
    return true;
end;

function map_vm_area(va:page_index; pa:frame_index; prot:pgprot_t):boolean;
var
    clr:pgprot_t;
    old_pte:pte_t;
begin
    if(within(va, kernel_start, kernel_end))
    then
        -- linear kernel mapping
        return false;
    else
        -- user space and vmalloc
        old_pte :=pt[va];
        pt[va].addr := pa;
        pt[va].mapped := true;
        clr.w := !prot.w;
        clr.x := !prot.x;
```

```
        if(!change_page_attr_set_clr(va, 1, prot, clr))
        then
            pt[va] := old_pte;
            return false;
        end;
        return true;
    end;
end;

procedure set_mem_perm(va:page_index; count:page_index; w:boolean; x:boolean);
var
    set: pgprot_t;
    clr: pgprot_t;
    ret: boolean;
begin
    if(w | x)
    then
        set.w := w;
        set.x := x;
        clr.w := false;
        clr.x := false;
        ret := change_page_attr_set_clr(va, count, set, clr);
    end;
end;

procedure clr_mem_perm(va:page_index; count:page_index; w:boolean; x:boolean);
var
    set: pgprot_t;
    clr: pgprot_t;
    ret: boolean;
begin
    if(w | x)
    then
        set.w := false;
        set.x := false;
        clr.w := w;
        clr.x := x;
        ret := change_page_attr_set_clr(va, count, set, clr);
    end;
end;

-- S1: linear mapping
procedure map_linear();
begin
    for va: linear_start .. linear_end do
```

```
        pt[va].addr := physical(va);
        pt[va].mapped := true;
        pt[va].prot.w := true;
        pt[va].prot.x := false;
    end;
end;

-- S2: kernel image mapping
procedure map_static_kernel();
begin
    -- S2: kernel text
    for va: kernel_text_start .. kernel_text_end do
        pt[va].addr := physical(va);
        pt[va].mapped := true;
        pt[va].prot.w := false;
        pt[va].prot.x := true;
    end;

    -- S2: kernel ro-data
    for va: kernel_ro_start .. kernel_ro_end do
        pt[va].addr := physical(va);
        pt[va].mapped := true;
        pt[va].prot.w := false;
        pt[va].prot.x := false;
    end;

    -- S2: kernel rw-data
    for va: kernel_rw_start .. kernel_rw_end do
        pt[va].addr := physical(va);
        pt[va].mapped := true;
        pt[va].prot.w := true;
        pt[va].prot.x := false;
    end;
end;

-- S3: BIOS mapping
procedure map_bios();
begin
    for va: bios_start .. bios_end do
        pt[va].addr := physical(va);
        pt[va].mapped := true;
        pt[va].prot.w := false;
        pt[va].prot.x := true;
    end;
end;
```

```
-- returns true if all aliases have consistent access flags
function page_alias_matching():boolean;
var
    pa: frame_index;
    mem: frame_table_t;
begin
    for i: frame_index do
        mem[i].reference_count := 0;
        mem[i].prot.w := false;
        mem[i].prot.x := false;
    end;

    for i: page_index do
        if(pt[i].mapped)
        then
            pa :=pt[i].addr;
            mem[pa].reference_count := mem[pa].reference_count + 1;
            mem[pa].prot := pgprot_set(mem[pa].prot, pt[i].prot);
            if(mem[pa].prot.w & mem[pa].prot.x)
            then
                return false;
            end;
        end;
    end;
    return true;
end;

-- returns true is there is a virtual page that has W and X set
function w_and_x():boolean;
begin
    for i: page_index do
        if(pt[i].prot.w & pt[i].prot.x)
            then
                return true;
            end;
    end;
    return false;
end;

function kernel_rwdata_rwnx():boolean;
begin
    for i: kernel_rw_start .. kernel_rw_end do
        if((!pt[i].prot.w) | (pt[i].prot.x))
```

```
        then
            return false;
        end;
    end;
    return true;
end;

function kernel_rodata_ronx():boolean;
begin
    for i: kernel_ro_start .. kernel_ro_end do
        if((pt[i].prot.w) | (pt[i].prot.x))
        then
            return false;
        end;
    end;
    return true;
end;

function kernel_code_rox():boolean;
begin
    for i: kernel_text_start .. kernel_text_end do
        if((pt[i].prot.w) | (!pt[i].prot.x))
        then
            return false;
        end;
    end;
    return true;
end;

-- Transition rules
ruleset i: page_index do
  ruleset x: boolean do
    ruleset w: boolean do

    -- set W and/or X for virtual page i
    rule "T1: Set"
        true ==>
        begin
            set_mem_perm(i, 1, w, x);
        end;

    -- clear W and/or X for virtual page i
    rule "T2: Clear"
        true ==>
        begin
```

```
            clr_mem_perm(i, 1, w, x);
        end;
    end;
  end;
end;

-- map virtual address va to physical address pa with prot attributes
ruleset pa: free_index do
  ruleset va: vmalloc_index do
    ruleset x: boolean do
      ruleset w: boolean do

    rule "T3: Map"
        true ==>
        var
            prot: pgprot_t;
            ret: boolean;
        begin
            prot.x := x;
            prot.w := w;
            ret := map_vm_area(va, pa, prot);
        end;
      end;
    end;
  end
end;

-- Invariants
invariant "P1: Kernel Code"
    true -> kernel_code_rox() = true;

invariant "P2: Kernel RO data"
    true -> kernel_rodata_ronx() = true;

invariant "P2: Kernel RW data"
    true -> kernel_rwdata_rwnx() = true;

invariant "P3: W xor X"
    true -> w_and_x() = false;

invariant "P4: Alias consistency"
    true -> page_alias_matching() = true;

startstate
begin
```

```
        clear pt;
        map_linear();
        map_static_kernel();
        map_bios();
end;
```

Appendix B

Patches for Linux Kernel

B.1 Preserving write-access on kernel read-write data

```
diff --git a/arch/x86/mm/pageattr.c b/arch/x86/mm/pageattr.c
index cf07c26..f4ed8eb 100644
--- a/arch/x86/mm/pageattr.c
+++ b/arch/x86/mm/pageattr.c
@@ -255,6 +255,7 @@ static inline pgprot_t static_protections(pgprot_t prot,
    unsigned long address,
                  unsigned long pfn)
 {
    pgprot_t forbidden = __pgprot(0);
+   pgprot_t required = __pgprot(0);

    /*
     * The BIOS area between 640k and 1Mb needs to be executable for
@@ -278,6 +279,13 @@ static inline pgprot_t static_protections(pgprot_t prot,
    unsigned long address,
    if (within(pfn, __pa((unsigned long)__start_rodata) >> PAGE_SHIFT,
            __pa((unsigned long)__end_rodata) >> PAGE_SHIFT))
        pgprot_val(forbidden) |= _PAGE_RW;
+   /*
+    * .data and .bss should always be writable.
+    */
+   if (within(address, (unsigned long)_sdata, (unsigned long)_edata) ||
+       within(address, (unsigned long)__bss_start,
+           (unsigned long)__bss_stop))
+       pgprot_val(required) |= _PAGE_RW;

 #if defined(CONFIG_X86_64) && defined(CONFIG_DEBUG_RODATA)
    /*
```

```
@@ -317,6 +325,7 @@ static inline pgprot_t static_protections(pgprot_t prot,
    unsigned long address,
 #endif

    prot = __pgprot(pgprot_val(prot) & ~pgprot_val(forbidden));
+   prot = __pgprot(pgprot_val(prot) | pgprot_val(required));

    return prot;
 }
@@ -393,7 +402,7 @@ try_preserve_large_page(pte_t *kpte, unsigned long address,
 {
    unsigned long nextpage_addr, numpages, pmask, psize, flags, addr, pfn;
    pte_t new_pte, old_pte, *tmp;
-   pgprot_t old_prot, new_prot;
+   pgprot_t old_prot, new_prot, req_prot;
    int i, do_split = 1;
    unsigned int level;

@@ -438,10 +447,10 @@ try_preserve_large_page(pte_t *kpte, unsigned long address,
     * We are safe now. Check whether the new pgprot is the same:
     */
    old_pte = *kpte;
-   old_prot = new_prot = pte_pgprot(old_pte);
+   old_prot = new_prot = req_prot = pte_pgprot(old_pte);

-   pgprot_val(new_prot) &= ~pgprot_val(cpa->mask_clr);
-   pgprot_val(new_prot) |= pgprot_val(cpa->mask_set);
+   pgprot_val(req_prot) &= ~pgprot_val(cpa->mask_clr);
+   pgprot_val(req_prot) |= pgprot_val(cpa->mask_set);

    /*
     * old_pte points to the large page base address. So we need
@@ -450,17 +459,17 @@ try_preserve_large_page(pte_t *kpte, unsigned long address,
    pfn = pte_pfn(old_pte) + ((address & (psize - 1)) >> PAGE_SHIFT);
    cpa->pfn = pfn;

-   new_prot = static_protections(new_prot, address, pfn);
+   new_prot = static_protections(req_prot, address, pfn);

    /*
     * We need to check the full range, whether
     * static_protection() requires a different pgprot for one of
     * the pages in the range we try to preserve:
     */
-   addr = address + PAGE_SIZE;
```

```
-    pfn++;
-    for (i = 1; i < cpa->numpages; i++, addr += PAGE_SIZE, pfn++) {
-        pgprot_t chk_prot = static_protections(new_prot, addr, pfn);
+    addr = address & pmask;
+    pfn = pte_pfn(old_pte);
+    for (i = 0; i < (psize >> PAGE_SHIFT); i++, addr += PAGE_SIZE, pfn++) {
+        pgprot_t chk_prot = static_protections(req_prot, addr, pfn);

         if (pgprot_val(chk_prot) != pgprot_val(new_prot))
             goto out_unlock;
@@ -483,7 +492,8 @@ try_preserve_large_page(pte_t *kpte, unsigned long address,
      * that we limited the number of possible pages already to
      * the number of pages in the large page.
      */
-    if (address == (nextpage_addr - psize) && cpa->numpages == numpages) {
+    if (address == (address & pmask) &&
+        cpa->numpages == (psize >> PAGE_SHIFT)) {
         /*
          * The address is aligned and the number of pages
          * covers the full page.
```

B.2 BIOS32 mapping

```
diff --git a/arch/x86/Kconfig.debug b/arch/x86/Kconfig.debug
index 4814d35..6dbcb6f 100644
--- a/arch/x86/Kconfig.debug
+++ b/arch/x86/Kconfig.debug
@@ -123,6 +123,17 @@ config DEBUG_NX_TEST
       and the software setup of this feature.
       If in doubt, say "N"

+config DEBUG_SET_1ST_MB_RWNX
+       bool "Set 1st MB of kernel space mapping as RW+NX"
+       default n
+       depends on X86
+       ---help---
+         This option helps to catch unintended modifications/execution of
+         1st megabyte of kernel address space. Such protection may interfere
+         with run-time code patching and legacy BIOS services residing within
+         the 1st MB of physical address space.
+         If in doubt, say "N".
+
 config 4KSTACKS
     bool "Use 4Kb for kernel stacks instead of 8Kb"
     depends on X86_32
```

```
diff --git a/arch/x86/mm/init_32.c b/arch/x86/mm/init_32.c
index 5cb3f0f..22ccf08 100644
--- a/arch/x86/mm/init_32.c
+++ b/arch/x86/mm/init_32.c
@@ -52,6 +52,12 @@
 #include <asm/page_types.h>
 #include <asm/init.h>

+#ifdef CONFIG_DEBUG_SET_1ST_MB_RWNX
+#define TEXT_START _text
+#else
+#define TEXT_START PAGE_OFFSET
+#endif
+
 unsigned long highstart_pfn, highend_pfn;

 static noinline int do_test_wp_bit(void);
@@ -225,7 +231,8 @@ page_table_range_init(unsigned long start, unsigned long end,
     pgd_t *pgd_base)

 static inline int is_kernel_text(unsigned long addr)
 {
-    if (addr >= PAGE_OFFSET && addr <= (unsigned long)__init_end)
+    if (addr >= (unsigned long)TEXT_START
+        && addr <= (unsigned long)__init_end)
         return 1;
     return 0;
 }
diff --git a/arch/x86/mm/pageattr.c b/arch/x86/mm/pageattr.c
index cf07c26..595398e 100644
--- a/arch/x86/mm/pageattr.c
+++ b/arch/x86/mm/pageattr.c
@@ -256,13 +256,14 @@ static inline pgprot_t static_protections(pgprot_t prot,
    unsigned long address,
 {
     pgprot_t forbidden = __pgprot(0);

+#ifndef CONFIG_DEBUG_SET_1ST_MB_RWNX
     /*
      * The BIOS area between 640k and 1Mb needs to be executable for
      * PCI BIOS based config access (CONFIG_PCI_GOBIOS) support.
      */
     if (within(pfn, BIOS_BEGIN >> PAGE_SHIFT, BIOS_END >> PAGE_SHIFT))
         pgprot_val(forbidden) |= _PAGE_NX;
-
```

```
+#endif
     /*
      * The kernel text needs to be executable for obvious reasons
      * Does not cover __inittext since that is gone later on. On
diff --git a/arch/x86/pci/pcbios.c b/arch/x86/pci/pcbios.c
index 1c975cc..ec3b3db 100644
--- a/arch/x86/pci/pcbios.c
+++ b/arch/x86/pci/pcbios.c
@@ -8,6 +8,7 @@
 #include <linux/uaccess.h>
 #include <asm/pci_x86.h>
 #include <asm/pci-functions.h>
+#include <asm/cacheflush.h>

 /* BIOS32 signature: "_32_" */
 #define BIOS32_SIGNATURE    (('_' << 0) + ('3' << 8) + ('2' << 16) + ('_' << 24))
@@ -58,6 +59,16 @@ static struct {
     unsigned short segment;
 } bios32_indirect = { 0, __KERNEL_CS };

+/* Set two consecutive pages as read-only and executable */
+void set_2_pages_rox(unsigned long addr)
+{
+#ifdef CONFIG_DEBUG_SET_1ST_MB_RWNX
+    unsigned long pg_address = PFN_DOWN(addr + PAGE_OFFSET) << PAGE_SHIFT;
+    set_memory_ro(pg_address, 2);
+    set_memory_x(pg_address, 2);
+#endif
+}
+
 /*
  * Returns the entry point for the given service, NULL on error
  */
@@ -82,15 +93,19 @@ static unsigned long bios32_service(unsigned long service)
     local_irq_restore(flags);

     switch (return_code) {
-        case 0:
-            return address + entry;
-        case 0x80:  /* Not present */
-            printk(KERN_WARNING "bios32_service(0x%lx): not present\n", service);
-            return 0;
-        default: /* Shouldn't happen */
-            printk(KERN_WARNING "bios32_service(0x%lx): returned 0x%x -- BIOS bug
    !\n",
```

```
-                service, return_code);
-            return 0;
+    case 0: /* Service present, set proper page access */
+        address += entry;
+        set_2_pages_rox(address);
+        return address;
+    case 0x80: /* Not present */
+        printk(KERN_WARNING "bios32_service(0x%lx): not present\n",
+            service);
+        return 0;
+    default: /* Shouldn't happen */
+        printk(KERN_WARNING "bios32_service(0x%lx): "
+            "returned 0x%x -- BIOS bug!\n",
+            service, return_code);
+        return 0;
     }
 }

@@ -331,6 +346,7 @@ static struct pci_raw_ops * __devinit pci_find_bios(void)
             DBG("PCI: BIOS32 Service Directory entry at 0x%lx\n",
                 bios32_entry);
             bios32_indirect.address = bios32_entry + PAGE_OFFSET;
+            set_2_pages_rox(bios32_entry);
             if (check_pcibios())
                 return &pci_bios_access;
         }
```

B.3 Protection for loadable kernel modules

```
diff --git a/arch/x86/Kconfig.debug b/arch/x86/Kconfig.debug
index 4814d35..2b1d2bd 100644
--- a/arch/x86/Kconfig.debug
+++ b/arch/x86/Kconfig.debug
@@ -115,6 +115,17 @@ config DEBUG_RODATA_TEST
       feature as well as for the change_page_attr() infrastructure.
       If in doubt, say "N"

+config DEBUG_SET_MODULE_RONX
+   bool "Set loadable kernel module data as NX and text as RO"
+   default n
+   depends on X86 && MODULES
+   ---help---
+     This option helps to catch unintended modifications to loadable
+     kernel module's text and read-only data. It also prevents execution
+     of LKM's data. Such protection may interfere with run-time code
```

```
+      patching and dynamic kernel tracing.
+      If in doubt, say "N".
+
 config DEBUG_NX_TEST
     tristate "Testcase for the NX non-executable stack feature"
     depends on DEBUG_KERNEL && m
diff --git a/arch/x86/kernel/ftrace.c b/arch/x86/kernel/ftrace.c
index cd37469..fa9f170 100644
--- a/arch/x86/kernel/ftrace.c
+++ b/arch/x86/kernel/ftrace.c
@@ -19,6 +19,7 @@
 #include <linux/sched.h>
 #include <linux/init.h>
 #include <linux/list.h>
+#include <linux/module.h>

 #include <trace/syscall.h>

@@ -49,6 +50,7 @@ static DEFINE_PER_CPU(int, save_modifying_code);
 int ftrace_arch_code_modify_prepare(void)
 {
     set_kernel_text_rw();
+    set_all_modules_text_rw();
     modifying_code = 1;
     return 0;
 }
@@ -56,6 +58,7 @@ int ftrace_arch_code_modify_prepare(void)
 int ftrace_arch_code_modify_post_process(void)
 {
     modifying_code = 0;
+    set_all_modules_text_ro();
     set_kernel_text_ro();
     return 0;
 }
diff --git a/include/linux/module.h b/include/linux/module.h
index 5e869ff..239a34c 100644
--- a/include/linux/module.h
+++ b/include/linux/module.h
@@ -301,6 +301,9 @@ struct module
     /* The size of the executable code in each section.  */
     unsigned int init_text_size, core_text_size;

+    /* Size of RO sections of the module (text+rodata) */
+    unsigned int init_ro_size, core_ro_size;
+
```

```
     /* Arch-specific module values */
     struct mod_arch_specific arch;

@@ -534,6 +537,9 @@ extern void print_modules(void);
 extern void module_update_tracepoints(void);
 extern int module_get_iter_tracepoints(struct tracepoint_iter *iter);

+void set_all_modules_text_rw(void);
+void set_all_modules_text_ro(void);
+
 #else /* !CONFIG_MODULES... */
 #define EXPORT_SYMBOL(sym)
 #define EXPORT_SYMBOL_GPL(sym)
@@ -654,6 +660,13 @@ static inline int module_get_iter_tracepoints(struct
     tracepoint_iter *iter)
     return 0;
 }

+static inline void set_all_modules_text_rw(void)
+{
+}
+
+static inline void set_all_modules_text_ro(void)
+{
+}
 #endif /* CONFIG_MODULES */

 struct device_driver;
diff --git a/kernel/module.c b/kernel/module.c
index c968d36..2020c6d 100644
--- a/kernel/module.c
+++ b/kernel/module.c
@@ -55,6 +55,7 @@
 #include <linux/async.h>
 #include <linux/percpu.h>
 #include <linux/kmemleak.h>
+#include <linux/pfn.h>

 #define CREATE_TRACE_POINTS
 #include <trace/events/module.h>
@@ -71,6 +72,26 @@ EXPORT_TRACEPOINT_SYMBOL(module_get);
 #define ARCH_SHF_SMALL 0
 #endif

+/*
```

```
+ * Modules' sections will be aligned on page boundaries
+ * to ensure complete separation of code and data, but
+ * only when CONFIG_DEBUG_SET_MODULE_RONX=y
+ */
+#ifdef CONFIG_DEBUG_SET_MODULE_RONX
+#define debug_align(X) ALIGN(X, PAGE_SIZE)
+#else
+#define debug_align(X) (X)
+#endif
+
+/*
+ * Given BASE and SIZE this macro calculates the number of pages the
+ * memory regions occupies
+ */
+#define NUMBER_OF_PAGES(BASE, SIZE) (((SIZE) > 0) ?         \
+        (PFN_DOWN((unsigned long)(BASE) + (SIZE) - 1) - \
+             PFN_DOWN((unsigned long)BASE) + 1) \
+        : (0UL))
+
 /* If this is set, the section belongs in the init part of the module */
 #define INIT_OFFSET_MASK (1UL << (BITS_PER_LONG-1))

@@ -1377,6 +1398,126 @@ static int __unlink_module(void *_mod)
     return 0;
 }

+#ifdef CONFIG_DEBUG_SET_MODULE_RONX
+/*
+ * LKM RO/NX protection: protect module's text/ro-data
+ * from modification and any data from execution.
+ */
+void set_page_attributes(void *start, void *end,
+        int (*set)(unsigned long start, int num_pages))
+{
+    unsigned long begin_pfn = PFN_DOWN((unsigned long)start);
+    unsigned long end_pfn = PFN_DOWN((unsigned long)end);
+    if (end_pfn > begin_pfn)
+        set(begin_pfn << PAGE_SHIFT, end_pfn - begin_pfn);
+}
+
+static void set_section_ro_nx(void *base,
+            unsigned long text_size,
+            unsigned long ro_size,
+            unsigned long total_size)
+{
```

```
+   /* begin and end PFNs of the current subsection */
+   unsigned long begin_pfn;
+   unsigned long end_pfn;
+
+   /*
+    * Set RO for module text and RO-data:
+    * - Always protect first page.
+    * - Do not protect last partial page.
+    */
+   if (ro_size > 0)
+       set_page_attributes(base, base + ro_size, set_memory_ro);
+
+   /*
+    * Set NX permissions for module data:
+    * - Do not protect first partial page.
+    * - Always protect last page.
+    */
+   if (total_size > text_size) {
+       begin_pfn = PFN_UP((unsigned long)base + text_size);
+       end_pfn = PFN_UP((unsigned long)base + total_size);
+       if (end_pfn > begin_pfn)
+           set_memory_nx(begin_pfn << PAGE_SHIFT,
+                         end_pfn - begin_pfn);
+   }
+}
+
+/* Setting memory back to RW+NX before releasing it */
+void unset_section_ro_nx(struct module *mod, void *module_region)
+{
+   unsigned long total_pages;
+
+   if (mod->module_core == module_region) {
+       /* Set core as NX+RW */
+       total_pages = NUMBER_OF_PAGES(mod->module_core, mod->core_size);
+       set_memory_nx((unsigned long)mod->module_core, total_pages);
+       set_memory_rw((unsigned long)mod->module_core, total_pages);
+
+   } else if (mod->module_init == module_region) {
+       /* Set init as NX+RW */
+       total_pages = NUMBER_OF_PAGES(mod->module_init, mod->init_size);
+       set_memory_nx((unsigned long)mod->module_init, total_pages);
+       set_memory_rw((unsigned long)mod->module_init, total_pages);
+   }
+}
+
```

```
+/* Iterate through all modules and set each module's text as RW */
+void set_all_modules_text_rw()
+{
+   struct module *mod;
+
+   mutex_lock(&module_mutex);
+   list_for_each_entry_rcu(mod, &modules, list) {
+       if ((mod->module_core) && (mod->core_text_size)) {
+           set_page_attributes(mod->module_core,
+                       mod->module_core
+                           + mod->core_text_size,
+                       set_memory_rw);
+       }
+       if ((mod->module_init) && (mod->init_text_size)) {
+           set_page_attributes(mod->module_init,
+                       mod->module_init
+                           + mod->init_text_size,
+                       set_memory_rw);
+       }
+   }
+   mutex_unlock(&module_mutex);
+}
+
+/* Iterate through all modules and set each module's text as RO */
+void set_all_modules_text_ro()
+{
+   struct module *mod;
+
+   mutex_lock(&module_mutex);
+   list_for_each_entry_rcu(mod, &modules, list) {
+       if ((mod->module_core) && (mod->core_text_size)) {
+           set_page_attributes(mod->module_core,
+                       mod->module_core
+                           + mod->core_text_size,
+                       set_memory_ro);
+       }
+       if ((mod->module_init) && (mod->init_text_size)) {
+           set_page_attributes(mod->module_init,
+                       mod->module_init
+                           + mod->init_text_size,
+                       set_memory_ro);
+       }
+   }
+   mutex_unlock(&module_mutex);
+}
```

```
+#else
+static void set_section_ro_nx(void *base,
+          unsigned long text_size,
+          unsigned long ro_size,
+          unsigned long total_size) { }
+
+void unset_section_ro_nx(struct module *mod, void *module_region) { }
+void set_all_modules_text_rw() { }
+void set_all_modules_text_ro() { }
+#endif
+
 /* Free a module, remove from lists, etc (must hold module_mutex). */
 static void free_module(struct module *mod)
 {
@@ -1398,6 +1539,7 @@ static void free_module(struct module *mod)
     destroy_params(mod->kp, mod->num_kp);

     /* This may be NULL, but that's OK */
+    unset_section_ro_nx(mod, mod->module_init);
     module_free(mod, mod->module_init);
     kfree(mod->args);
     if (mod->percpu)
@@ -1410,6 +1552,7 @@ static void free_module(struct module *mod)
     lockdep_free_key_range(mod->module_core, mod->core_size);

     /* Finally, free the core (containing the module structure) */
+    unset_section_ro_nx(mod, mod->module_core);
     module_free(mod, mod->module_core);

 #ifdef CONFIG_MPU
@@ -1587,8 +1730,19 @@ static void layout_sections(struct module *mod,
             s->sh_entsize = get_offset(mod, &mod->core_size, s, i);
             DEBUGP("\t%s\n", secstrings + s->sh_name);
         }
-        if (m == 0)
+        switch (m) {
+        case 0: /* executable */
+            mod->core_size = debug_align(mod->core_size);
             mod->core_text_size = mod->core_size;
+            break;
+        case 1: /* RO: text and ro-data */
+            mod->core_size = debug_align(mod->core_size);
+            mod->core_ro_size = mod->core_size;
+            break;
+        case 3: /* whole core */
```

```
+           mod->core_size = debug_align(mod->core_size);
+           break;
+       }
    }

    DEBUGP("Init section allocation order:\n");
@@ -1605,8 +1759,19 @@ static void layout_sections(struct module *mod,
                        | INIT_OFFSET_MASK);
            DEBUGP("\t%s\n", secstrings + s->sh_name);
        }
-       if (m == 0)
+       switch (m) {
+       case 0: /* executable */
+           mod->init_size = debug_align(mod->init_size);
            mod->init_text_size = mod->init_size;
+           break;
+       case 1: /* RO: text and ro-data */
+           mod->init_size = debug_align(mod->init_size);
+           mod->init_ro_size = mod->init_size;
+           break;
+       case 3: /* whole init */
+           mod->init_size = debug_align(mod->init_size);
+           break;
+       }
    }
 }

@@ -2386,6 +2551,18 @@ static noinline struct module *load_module(void __user *
    umod,

    trace_module_load(mod);

+   /* Set RO and NX regions for core */
+   set_section_ro_nx(mod->module_core,
+               mod->core_text_size,
+               mod->core_ro_size,
+               mod->core_size);
+
+   /* Set RO and NX regions for init */
+   set_section_ro_nx(mod->module_init,
+               mod->init_text_size,
+               mod->init_ro_size,
+               mod->init_size);
+
    /* Done! */
```

```
     return mod;

@@ -2508,6 +2685,7 @@ SYSCALL_DEFINE3(init_module, void __user *, umod,
     mod->symtab = mod->core_symtab;
     mod->strtab = mod->core_strtab;
 #endif
+    unset_section_ro_nx(mod, mod->module_init);
     module_free(mod, mod->module_init);
     mod->module_init = NULL;
     mod->init_size = 0;
```

B.4 Linux static kernel image protection

```
diff --git a/arch/x86/kernel/vmlinux.lds.S b/arch/x86/kernel/vmlinux.lds.S
index 2cc2497..7f82438 100644
--- a/arch/x86/kernel/vmlinux.lds.S
+++ b/arch/x86/kernel/vmlinux.lds.S
@@ -69,7 +69,7 @@ jiffies_64 = jiffies;

 PHDRS {
     text PT_LOAD FLAGS(5);          /* R_E */
-    data PT_LOAD FLAGS(7);          /* RWE */
+    data PT_LOAD FLAGS(6);          /* RW_ */
 #ifdef CONFIG_X86_64
     user PT_LOAD FLAGS(5);          /* R_E */
 #ifdef CONFIG_SMP
@@ -108,6 +108,8 @@ SECTIONS
         IRQENTRY_TEXT
         *(.fixup)
         *(.gnu.warning)
+        /* .text should occupy whole number of pages */
+        . = ALIGN(PAGE_SIZE);
         /* End of text section */
         _etext = .;
     } :text = 0x9090
@@ -121,6 +123,7 @@ SECTIONS
     X64_ALIGN_DEBUG_RODATA_END

     /* Data */
+    . = ALIGN(PAGE_SIZE);
     .data : AT(ADDR(.data) - LOAD_OFFSET) {
         /* Start of data section */
         _sdata = .;
@@ -143,6 +146,8 @@ SECTIONS
         /* rarely changed data like cpu maps */
```

```
        READ_MOSTLY_DATA(INTERNODE_CACHE_BYTES)

+       /* .data should occupy whole number of pages */
+       . = ALIGN(PAGE_SIZE);
        /* End of data section */
        _edata = .;
    } :data
@@ -307,7 +312,7 @@ SECTIONS
        __bss_start = .;
        *(.bss.page_aligned)
        *(.bss)
-       . = ALIGN(4);
+       . = ALIGN(PAGE_SIZE);
        __bss_stop = .;
    }

diff --git a/arch/x86/mm/init.c b/arch/x86/mm/init.c
index 452ee5b..240dcb0 100644
--- a/arch/x86/mm/init.c
+++ b/arch/x86/mm/init.c
@@ -361,8 +361,9 @@ void free_init_pages(char *what, unsigned long begin,
    unsigned long end)
    /*
     * We just marked the kernel text read only above, now that
     * we are going to free part of that, we need to make that
-    * writeable first.
+    * writeable and non-executable first.
     */
+   set_memory_nx(begin, (end - begin) >> PAGE_SHIFT);
    set_memory_rw(begin, (end - begin) >> PAGE_SHIFT);

    printk(KERN_INFO "Freeing %s: %luk freed\n", what, (end - begin) >> 10);
@@ -377,11 +378,29 @@ void free_init_pages(char *what, unsigned long begin,
    unsigned long end)
 #endif
 }

+void mark_nxdata_nx(void)
+{
+#ifdef CONFIG_DEBUG_RODATA
+   /*
+    * When this called, init has already been executed and released,
+    * so everything past _etext sould be NX.
+    */
+   unsigned long start = PAGE_ALIGN((unsigned long)(&_etext));
```

```
+   unsigned long size = PAGE_ALIGN((unsigned long)(&_end)) - start;
+
+   printk(KERN_INFO "NX-protecting the kernel data: %lx, %lu pages\n",
+       start, size >> PAGE_SHIFT);
+   set_memory_nx(start, size >> PAGE_SHIFT);
+#endif
+}
+
 void free_initmem(void)
 {
    free_init_pages("unused kernel memory",
            (unsigned long)(&__init_begin),
            (unsigned long)(&__init_end));
+   /* Set kernel's data as NX */
+   mark_nxdata_nx();
 }

 #ifdef CONFIG_BLK_DEV_INITRD
```

B.5 Enforcing $W \oplus X$

```
diff --git a/arch/x86/mm/pageattr.c b/arch/x86/mm/pageattr.c
index 5d231d1..7f64c79 100644
--- a/arch/x86/mm/pageattr.c
+++ b/arch/x86/mm/pageattr.c
@@ -245,6 +245,19 @@ static void cpa_flush_array(unsigned long *start, int
    numpages, int cache,
    }
 }

+static inline pgprot_t process_WxorX_violation(pgprot_t prot, unsigned long
    address,
+                   unsigned long pfn)
+{
+   /*
+    * We can Oops here of even panic if needed.
+    * however, for now we just print out an error message.
+    */
+   printk(KERN_ERR "(W xor X) violation: VA=0x%lx, PFN=0x%lx", address, pfn);
+   /* Set NX, just in case */
+   pgprot_val(prot) |= _PAGE_NX;
+   return prot;
+}
+
 /*
```

```
  * Certain areas of memory on x86 require very specific protection flags,
  * for example the BIOS area or kernel text. Callers don't always get this
@@ -298,6 +311,9 @@ static inline pgprot_t static_protections(pgprot_t prot,
    unsigned long address,

    prot = __pgprot(pgprot_val(prot) & ~pgprot_val(forbidden));

+   if((pgprot_val(prot) & _PAGE_RW) &&
+           (!(pgprot_val(prot) & _PAGE_NX)))
+       prot = process_WxorX_violation(prot, address, pfn);
    return prot;
 }

@@ -880,7 +896,8 @@ static int change_page_attr_set_clr(unsigned long *addr, int
    numpages,
        cpa.flags |= in_flag;

    /* No alias checking for _NX bit modifications */
-   checkalias = (pgprot_val(mask_set) | pgprot_val(mask_clr)) != _PAGE_NX;
+   /* checkalias = (pgprot_val(mask_set) | pgprot_val(mask_clr)) != _PAGE_NX; */
+   checkalias = 1;

    ret = __change_page_attr_set_clr(&cpa, checkalias);
```

Appendix C

Testing Data

C.1 Vanilla kernel

C.1.1 Page table dump

```
---[ User Space ]---
0x00000000-0xc0000000          3G                                  pmd
---[ Kernel Mapping ]---
0xc0000000-0xc0100000          1M     RW                GLB x    pte
0xc0100000-0xc04fc000       4080K     ro                GLB x    pte
0xc04fc000-0xc0600000       1040K     RW                GLB x    pte
0xc0600000-0xf7800000        882M     RW            PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW                GLB NX pte
0xf79fe000-0xf7a00000          8K                                  pte
0xf7a00000-0xf8000000          6M                                  pmd
0xf8000000-0xf81fe000       2040K                                  pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW         PCD     GLB NX pte
0xf81ff000-0xf8202000         12K                                  pte
0xf8202000-0xf8203000          4K     RW         PCD     GLB NX pte
0xf8203000-0xf8206000         12K                                  pte
0xf8206000-0xf8207000          4K     RW         PCD     GLB NX pte
0xf8207000-0xf8208000          4K                                  pte
0xf8208000-0xf820f000         28K     RW         PCD     GLB NX pte
0xf820f000-0xf8212000         12K                                  pte
0xf8212000-0xf8213000          4K     RW         PCD     GLB NX pte
0xf8213000-0xf8215000          8K                                  pte
0xf8215000-0xf8220000         44K     RW                 GLB NX pte
0xf8220000-0xf8222000          8K                                  pte
0xf8222000-0xf8230000         56K     RW                 GLB x    pte
0xf8230000-0xf8232000          8K                                  pte
0xf8232000-0xf8237000         20K     RW                 GLB x    pte
```

0xf8237000–0xf8239000	8K				pte
0xf8239000–0xf823c000	12K	RW		GLB x	pte
0xf823c000–0xf823e000	8K				pte
0xf823e000–0xf8240000	8K	RW		GLB x	pte
0xf8240000–0xf8242000	8K				pte
0xf8242000–0xf8243000	4K	RW	PCD	GLB NX	pte
0xf8243000–0xf8246000	12K				pte
0xf8246000–0xf8247000	4K	RW	PCD	GLB NX	pte
0xf8247000–0xf824a000	12K				pte
0xf824a000–0xf824b000	4K	RW	PCD	GLB NX	pte
0xf824b000–0xf824d000	8K				pte
0xf824d000–0xf8309000	752K	RW		GLB NX	pte
0xf8309000–0xf830b000	8K				pte
0xf830b000–0xf8328000	116K	RW		GLB NX	pte
0xf8328000–0xf832a000	8K				pte
0xf832a000–0xf834a000	128K	RW		GLB NX	pte
0xf834a000–0xf834c000	8K				pte
0xf834c000–0xf834d000	4K	RW		GLB x	pte
0xf834d000–0xf834f000	8K				pte
0xf834f000–0xf8350000	4K	RW		GLB x	pte
0xf8350000–0xf8352000	8K				pte
0xf8352000–0xf837e000	176K	RW		GLB NX	pte
0xf837e000–0xf8380000	8K				pte
0xf8380000–0xf8383000	12K	RW		GLB x	pte
0xf8383000–0xf8385000	8K				pte
0xf8385000–0xf8387000	8K	RW		GLB x	pte
0xf8387000–0xf8389000	8K				pte
0xf8389000–0xf85c2000	2276K	RW		GLB NX	pte
0xf85c2000–0xf85c4000	8K				pte
0xf85c4000–0xf85e7000	140K	RW		GLB x	pte
0xf85e7000–0xf85e9000	8K				pte
0xf85e9000–0xf85f5000	48K	RW		GLB x	pte
0xf85f5000–0xf85f7000	8K				pte
0xf85f7000–0xf87e0000	1956K	RW		GLB NX	pte
0xf87e0000–0xf87e2000	8K				pte
0xf87e2000–0xf8805000	140K	RW		GLB x	pte
0xf8805000–0xf8807000	8K				pte
0xf8807000–0xf8814000	52K	RW		GLB x	pte
0xf8814000–0xf8816000	8K				pte
0xf8816000–0xf8972000	1392K	RW		GLB NX	pte
0xf8972000–0xf8974000	8K				pte
0xf8974000–0xf899c000	160K	RW		GLB x	pte
0xf899c000–0xf899e000	8K				pte
0xf899e000–0xf89ad000	60K	RW		GLB x	pte
0xf89ad000–0xf89af000	8K				pte

0xf89af000–0xf89f2000	268K	RW		GLB NX	pte
0xf89f2000–0xf89f4000	8K				pte
0xf89f4000–0xf89f9000	20K	RW		GLB x	pte
0xf89f9000–0xf89fb000	8K				pte
0xf89fb000–0xf89fd000	8K	RW		GLB x	pte
0xf89fd000–0xf8a00000	12K				pte
0xf8a00000–0xf8a01000	4K	RW	PCD	GLB NX	pte
0xf8a01000–0xf8a03000	8K				pte
0xf8a03000–0xf8a59000	344K	RW		GLB NX	pte
0xf8a59000–0xf8a5b000	8K				pte
0xf8a5b000–0xf8a64000	36K	RW		GLB x	pte
0xf8a64000–0xf8a66000	8K				pte
0xf8a66000–0xf8a68000	8K	RW		GLB x	pte
0xf8a68000–0xf8a6a000	8K				pte
0xf8a6a000–0xf8b59000	956K	RW		GLB NX	pte
0xf8b59000–0xf8b5b000	8K				pte
0xf8b5b000–0xf8b6e000	76K	RW		GLB x	pte
0xf8b6e000–0xf8b70000	8K				pte
0xf8b70000–0xf8b78000	32K	RW		GLB x	pte
0xf8b78000–0xf8b7a000	8K				pte
0xf8b7a000–0xf8bb8000	248K	RW		GLB NX	pte
0xf8bb8000–0xf8bba000	8K				pte
0xf8bba000–0xf8bc1000	28K	RW		GLB x	pte
0xf8bc1000–0xf8bc3000	8K				pte
0xf8bc3000–0xf8bc5000	8K	RW		GLB x	pte
0xf8bc5000–0xf8bc8000	12K				pte
0xf8bc8000–0xf8bc9000	4K	RW	PCD	GLB NX	pte
0xf8bc9000–0xf8bcc000	12K				pte
0xf8bcc000–0xf8bcd000	4K	RW	PCD	GLB NX	pte
0xf8bcd000–0xf8bd0000	12K				pte
0xf8bd0000–0xf8bd1000	4K	RW	PCD	GLB NX	pte
0xf8bd1000–0xf8bd3000	8K				pte
0xf8bd3000–0xf8bfe000	172K	RW		GLB NX	pte
0xf8bfe000–0xf8c00000	8K				pte
0xf8c00000–0xf8c01000	4K	RW		GLB x	pte
0xf8c01000–0xf8c03000	8K				pte
0xf8c03000–0xf8c04000	4K	RW		GLB x	pte
0xf8c04000–0xf8c06000	8K				pte
0xf8c06000–0xf8c54000	312K	RW		GLB NX	pte
0xf8c54000–0xf8c56000	8K				pte
0xf8c56000–0xf8c5e000	32K	RW		GLB x	pte
0xf8c5e000–0xf8c60000	8K				pte
0xf8c60000–0xf8c63000	12K	RW		GLB x	pte
0xf8c63000–0xf8c66000	12K				pte
0xf8c66000–0xf8c67000	4K	RW	PCD	GLB NX	pte

0xf8c67000–0xf8c69000	8K				pte
0xf8c69000–0xf8c95000	176K	RW		GLB NX	pte
0xf8c95000–0xf8c97000	8K				pte
0xf8c97000–0xf8c98000	4K	RW		GLB x	pte
0xf8c98000–0xf8c9a000	8K				pte
0xf8c9a000–0xf8c9b000	4K	RW		GLB x	pte
0xf8c9b000–0xf8c9e000	12K				pte
0xf8c9e000–0xf8c9f000	4K	RW	PCD	GLB NX	pte
0xf8c9f000–0xf8ca2000	12K				pte
0xf8ca2000–0xf8ca3000	4K	RW	PCD	GLB NX	pte
0xf8ca3000–0xf8ca6000	12K				pte
0xf8ca6000–0xf8ca7000	4K	RW	PCD	GLB NX	pte
0xf8ca7000–0xf8caa000	12K				pte
0xf8caa000–0xf8cab000	4K	RW	PCD	GLB NX	pte
0xf8cab000–0xf8cad000	8K				pte
0xf8cad000–0xf8cd8000	172K	RW		GLB NX	pte
0xf8cd8000–0xf8cda000	8K				pte
0xf8cda000–0xf8cdb000	4K	RW		GLB x	pte
0xf8cdb000–0xf8cdd000	8K				pte
0xf8cdd000–0xf8cde000	4K	RW		GLB x	pte
0xf8cde000–0xf8ce0000	8K				pte
0xf8ce0000–0xf8d08000	160K	RW		GLB NX	pte
0xf8d08000–0xf8d0a000	8K				pte
0xf8d0a000–0xf8d0b000	4K	RW		GLB x	pte
0xf8d0b000–0xf8d0d000	8K				pte
0xf8d0d000–0xf8d0e000	4K	RW		GLB x	pte
0xf8d0e000–0xf8d10000	8K				pte
0xf8d10000–0xf8d54000	272K	RW		GLB NX	pte
0xf8d54000–0xf8d56000	8K				pte
0xf8d56000–0xf8d5d000	28K	RW		GLB x	pte
0xf8d5d000–0xf8d5f000	8K				pte
0xf8d5f000–0xf8d61000	8K	RW		GLB x	pte
0xf8d61000–0xf8d63000	8K				pte
0xf8d63000–0xf8da4000	260K	RW		GLB NX	pte
0xf8da4000–0xf8da6000	8K				pte
0xf8da6000–0xf8dad000	28K	RW		GLB x	pte
0xf8dad000–0xf8daf000	8K				pte
0xf8daf000–0xf8db1000	8K	RW		GLB x	pte
0xf8db1000–0xf8db3000	8K				pte
0xf8db3000–0xf8dd6000	140K	RW		GLB NX	pte
0xf8dd6000–0xf8dd8000	8K				pte
0xf8dd8000–0xf8dda000	8K	RW		GLB x	pte
0xf8dda000–0xf8ddc000	8K				pte
0xf8ddc000–0xf8ddd000	4K	RW		GLB x	pte
0xf8ddd000–0xf8ddf000	8K				pte

```
0xf8ddf000-0xf8e9b000      752K     RW          GLB NX pte
0xf8e9b000-0xf8e9d000        8K                      pte
0xf8e9d000-0xf8ea7000       40K     RW          GLB x  pte
0xf8ea7000-0xf8ea9000        8K                      pte
0xf8ea9000-0xf8ead000       16K     RW          GLB x  pte
0xf8ead000-0xf8eaf000        8K                      pte
0xf8eaf000-0xf90c7000     2144K     RW          GLB NX pte
0xf90c7000-0xf90c9000        8K                      pte
0xf90c9000-0xf90e6000      116K     RW          GLB x  pte
0xf90e6000-0xf90e8000        8K                      pte
0xf90e8000-0xf90ee000       24K     RW          GLB x  pte
0xf90ee000-0xf90f0000        8K                      pte
0xf90f0000-0xf9107000       92K     RW          GLB NX pte
0xf9107000-0xf9109000        8K                      pte
0xf9109000-0xf910a000        4K     RW          GLB x  pte
0xf910a000-0xf910c000        8K                      pte
0xf910c000-0xf910d000        4K     RW          GLB x  pte
0xf910d000-0xf910f000        8K                      pte
0xf910f000-0xf9137000      160K     RW          GLB NX pte
0xf9137000-0xf9139000        8K                      pte
0xf9139000-0xf913b000        8K     RW          GLB x  pte
0xf913b000-0xf913d000        8K                      pte
0xf913d000-0xf913e000        4K     RW          GLB x  pte
0xf913e000-0xf9140000        8K                      pte
0xf9140000-0xf916c000      176K     RW          GLB NX pte
0xf916c000-0xf916e000        8K                      pte
0xf916e000-0xf9173000       20K     RW          GLB x  pte
0xf9173000-0xf9175000        8K                      pte
0xf9175000-0xf9178000       12K     RW          GLB x  pte
0xf9178000-0xf917a000        8K                      pte
0xf917a000-0xf9199000      124K     RW          GLB NX pte
0xf9199000-0xf919b000        8K                      pte
0xf919b000-0xf919d000        8K     RW          GLB x  pte
0xf919d000-0xf919f000        8K                      pte
0xf919f000-0xf91a0000        4K     RW          GLB x  pte
0xf91a0000-0xf91a2000        8K                      pte
0xf91a2000-0xf91cb000      164K     RW          GLB NX pte
0xf91cb000-0xf91cd000        8K                      pte
0xf91cd000-0xf91d1000       16K     RW          GLB x  pte
0xf91d1000-0xf91d3000        8K                      pte
0xf91d3000-0xf91d5000        8K     RW          GLB x  pte
0xf91d5000-0xf91d7000        8K                      pte
0xf91d7000-0xf91fd000      152K     RW          GLB NX pte
0xf91fd000-0xf91ff000        8K                      pte
0xf91ff000-0xf9201000        8K     RW          GLB x  pte
```

0xf9201000–0xf9203000	8K			pte
0xf9203000–0xf9205000	8K	RW	GLB x	pte
0xf9205000–0xf9207000	8K			pte
0xf9207000–0xf929d000	600K	RW	GLB NX	pte
0xf929d000–0xf929f000	8K			pte
0xf929f000–0xf92a6000	28K	RW	GLB x	pte
0xf92a6000–0xf92a8000	8K			pte
0xf92a8000–0xf92ab000	12K	RW	GLB x	pte
0xf92ab000–0xf92ad000	8K			pte
0xf92ad000–0xf9347000	616K	RW	GLB NX	pte
0xf9347000–0xf9349000	8K			pte
0xf9349000–0xf9350000	28K	RW	GLB x	pte
0xf9350000–0xf9352000	8K			pte
0xf9352000–0xf9354000	8K	RW	GLB x	pte
0xf9354000–0xf9356000	8K			pte
0xf9356000–0xf936b000	84K	RW	GLB NX	pte
0xf936b000–0xf936d000	8K			pte
0xf936d000–0xf936e000	4K	RW	GLB x	pte
0xf936e000–0xf9370000	8K			pte
0xf9370000–0xf9371000	4K	RW	GLB x	pte
0xf9371000–0xf9373000	8K			pte
0xf9373000–0xf9399000	152K	RW	GLB NX	pte
0xf9399000–0xf939b000	8K			pte
0xf939b000–0xf939e000	12K	RW	GLB x	pte
0xf939e000–0xf93a0000	8K			pte
0xf93a0000–0xf93a1000	4K	RW	GLB x	pte
0xf93a1000–0xf93a3000	8K			pte
0xf93a3000–0xf9446000	652K	RW	GLB NX	pte
0xf9446000–0xf9448000	8K			pte
0xf9448000–0xf944f000	28K	RW	GLB x	pte
0xf944f000–0xf9451000	8K			pte
0xf9451000–0xf9454000	12K	RW	GLB x	pte
0xf9454000–0xf9456000	8K			pte
0xf9456000–0xf9478000	136K	RW	GLB NX	pte
0xf9478000–0xf947a000	8K			pte
0xf947a000–0xf947b000	4K	RW	GLB x	pte
0xf947b000–0xf947d000	8K			pte
0xf947d000–0xf947e000	4K	RW	GLB x	pte
0xf947e000–0xf9480000	8K			pte
0xf9480000–0xf94a4000	144K	RW	GLB NX	pte
0xf94a4000–0xf94a6000	8K			pte
0xf94a6000–0xf94a8000	8K	RW	GLB x	pte
0xf94a8000–0xf94aa000	8K			pte
0xf94aa000–0xf94ab000	4K	RW	GLB x	pte
0xf94ab000–0xf94ad000	8K			pte

0xf94ad000–0xf9530000	524K	RW		GLB NX	pte
0xf9530000–0xf9532000	8K				pte
0xf9532000–0xf953a000	32K	RW		GLB x	pte
0xf953a000–0xf953c000	8K				pte
0xf953c000–0xf953f000	12K	RW		GLB x	pte
0xf953f000–0xf9541000	8K				pte
0xf9541000–0xf958a000	292K	RW		GLB NX	pte
0xf958a000–0xf958c000	8K				pte
0xf958c000–0xf9593000	28K	RW		GLB x	pte
0xf9593000–0xf9595000	8K				pte
0xf9595000–0xf9598000	12K	RW		GLB x	pte
0xf9598000–0xf959a000	8K				pte
0xf959a000–0xf9724000	1576K	RW		GLB NX	pte
0xf9724000–0xf9726000	8K				pte
0xf9726000–0xf9743000	116K	RW		GLB x	pte
0xf9743000–0xf9745000	8K				pte
0xf9745000–0xf974c000	28K	RW		GLB x	pte
0xf974c000–0xf974e000	8K				pte
0xf974e000–0xf974f000	4K	RW	PCD	GLB NX	pte
0xf974f000–0xf9751000	8K				pte
0xf9751000–0xf9755000	16K	RW		GLB x	pte
0xf9755000–0xf9757000	8K				pte
0xf9757000–0xf9759000	8K	RW		GLB x	pte
0xf9759000–0xf975b000	8K				pte
0xf975b000–0xf977f000	144K	RW		GLB NX	pte
0xf977f000–0xf9780000	4K				pte
0xf9780000–0xf97a0000	128K	RW	PCD	GLB NX	pte
0xf97a0000–0xf97a2000	8K				pte
0xf97a2000–0xf97d4000	200K	RW		GLB NX	pte
0xf97d4000–0xf97d6000	8K				pte
0xf97d6000–0xf97d8000	8K	RW		GLB x	pte
0xf97d8000–0xf97da000	8K				pte
0xf97da000–0xf97db000	4K	RW		GLB x	pte
0xf97db000–0xf97dd000	8K				pte
0xf97dd000–0xf981b000	248K	RW		GLB NX	pte
0xf981b000–0xf981d000	8K				pte
0xf981d000–0xf9822000	20K	RW		GLB x	pte
0xf9822000–0xf9824000	8K				pte
0xf9824000–0xf9826000	8K	RW		GLB x	pte
0xf9826000–0xf9828000	8K				pte
0xf9828000–0xf985d000	212K	RW		GLB NX	pte
0xf985d000–0xf985f000	8K				pte
0xf985f000–0xf988f000	192K	RW		GLB NX	pte
0xf988f000–0xf9891000	8K				pte
0xf9891000–0xf9895000	16K	RW		GLB x	pte

0xf9895000–0xf9897000	8K			pte
0xf9897000–0xf989a000	12K	RW	GLB x	pte
0xf989a000–0xf989c000	8K			pte
0xf989c000–0xf98d4000	224K	RW	GLB NX	pte
0xf98d4000–0xf98d6000	8K			pte
0xf98d6000–0xf98d9000	12K	RW	GLB x	pte
0xf98d9000–0xf98db000	8K			pte
0xf98db000–0xf98dc000	4K	RW	GLB x	pte
0xf98dc000–0xf98de000	8K			pte
0xf98de000–0xf9908000	168K	RW	GLB NX	pte
0xf9908000–0xf990a000	8K			pte
0xf990a000–0xf990b000	4K	RW	GLB x	pte
0xf990b000–0xf990d000	8K			pte
0xf990d000–0xf990e000	4K	RW	GLB x	pte
0xf990e000–0xf9910000	8K			pte
0xf9910000–0xf9911000	4K	RW	GLB NX	pte
0xf9911000–0xf9913000	8K			pte
0xf9913000–0xf9955000	264K	RW	GLB NX	pte
0xf9955000–0xf9957000	8K			pte
0xf9957000–0xf995b000	16K	RW	GLB x	pte
0xf995b000–0xf995d000	8K			pte
0xf995d000–0xf995f000	8K	RW	GLB x	pte
0xf995f000–0xf9961000	8K			pte
0xf9961000–0xf9eff000	5752K	RW	GLB NX	pte
0xf9eff000–0xf9f01000	8K			pte
0xf9f01000–0xf9f03000	8K	RW	GLB NX	pte
0xf9f03000–0xf9f05000	8K			pte
0xf9f05000–0xf9f3f000	232K	RW	GLB x	pte
0xf9f3f000–0xf9f41000	8K			pte
0xf9f41000–0xf9f4f000	56K	RW	GLB x	pte
0xf9f4f000–0xf9f51000	8K			pte
0xf9f51000–0xf9f53000	8K	RW	GLB NX	pte
0xf9f53000–0xf9f55000	8K			pte
0xf9f55000–0xf9f57000	8K	RW	GLB NX	pte
0xf9f57000–0xfa000000	676K			pte
0xfa000000–0xff600000	86M			pmd
0xff600000–0xff602000	8K	RW	GLB NX	pte
0xff602000–0xff680000	504K			pte
0xff680000–0xff682000	8K	RW	GLB NX	pte
0xff682000–0xff700000	504K			pte
0xff700000–0xff702000	8K	RW	GLB NX	pte
0xff702000–0xff780000	504K			pte
0xff780000–0xff782000	8K	RW	GLB NX	pte
0xff782000–0xff800000	504K			pte
0xff800000–0xffa00000	2M			pmd

```
---[ vmalloc() End ]---
0xffa00000-0xffa01000              4K     RW              GLB NX pte
---[ Persisent kmap() Area ]---
0xffa01000-0xffa69000            416K     RW              GLB NX pte
0xffa69000-0xfff1f000           4824K                             pte
---[ Fixmap Area ]---
0xfff1f000-0xfff22000             12K                             pte
0xfff22000-0xfff23000              4K     RW PWT PCD      GLB NX pte
0xfff23000-0xffffa000            860K                             pte
0xffffa000-0xffffc000              8K     RW PWT PCD      GLB NX pte
0xffffc000-0x00000000             16K                             pte
```

C.1.2 Loaded module listing

```
Module                  Size  Used by
ipv6                  237023  18
af_packet              15491  2
iptable_filter          1297  0
ip_tables               9449  1 iptable_filter
x_tables               13731  1 ip_tables
sbp2                   18736  0
lp                      7058  0
loop                   12891  0
e1000e                117413  0
parport_pc             26557  1
parport                31062  2 lp,parport_pc
button                  4482  0
serio_raw               3907  0
processor              28501  0
i2c_piix4               8263  0
k8temp                  3171  0
shpchp                 27865  0
pci_hotplug            26335  1 shpchp
tpm_tis                 7361  0
tpm                    12878  1 tpm_tis
tpm_bios                5124  1 tpm
i2c_core               19453  1 i2c_piix4
evdev                   7057  0
ext3                  115323  1
jbd                    40395  1 ext3
mbcache                 5650  1 ext3
sd_mod                 27689  3
sg                     24695  0
atiixp                  2297  0
pata_acpi               2475  0
pata_atiixp             3076  2
```

```
ahci                   31509  0
ata_generic             2607  0
ohci1394               27115  0
ieee1394               77229  2 sbp2,ohci1394
ehci_hcd               33297  0
ohci_hcd               20211  0
libata                162601  4 pata_acpi,pata_atiixp,ahci,ata_generic
scsi_mod              140457  4 sbp2,sd_mod,sg,libata
usbcore               141176  3 ehci_hcd,ohci_hcd
thermal                10416  0
fan                     2731  0
thermal_sys            12268  3 processor,thermal,fan
fuse                   53463  1
```

C.1.3 UnixBench test results

```
   BYTE UNIX Benchmarks (Version 5.1.2)

   System: test: GNU/Linux
   OS: GNU/Linux -- 2.6.33-vanilla -- #1 SMP Sun Mar 14 20:20:39 EDT 2010
   Machine: i686 (unknown)
   Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
   CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5011.6 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   22:54:06 up 22 min,  1 user,  load average: 0.08, 0.02, 0.01; runlevel 2

------------------------------------------------------------------------

Benchmark Run: Thu Mar 18 2010 22:54:06 - 23:22:05
2 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables       8076304.2 lps   (10.0 s, 7 samples)
Double-Precision Whetstone                    2137.6 MWIPS (10.0 s, 7 samples)
Execl Throughput                              3491.9 lps   (29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks       347173.2 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks         106129.4 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks       711015.1 KBps  (30.0 s, 2 samples)
Pipe Throughput                             823718.5 lps   (10.0 s, 7 samples)
Pipe-based Context Switching                 45819.5 lps   (10.0 s, 7 samples)
Process Creation                              8140.6 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)                  6165.1 lpm   (60.0 s, 2 samples)
Shell Scripts (8 concurrent)                  1118.8 lpm   (60.0 s, 2 samples)
System Call Overhead                       1839012.2 lps   (10.0 s, 7 samples)
```

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	8076304.2	692.1
Double-Precision Whetstone	55.0	2137.6	388.7
Execl Throughput	43.0	3491.9	812.1
File Copy 1024 bufsize 2000 maxblocks	3960.0	347173.2	876.7
File Copy 256 bufsize 500 maxblocks	1655.0	106129.4	641.3
File Copy 4096 bufsize 8000 maxblocks	5800.0	711015.1	1225.9
Pipe Throughput	12440.0	823718.5	662.2
Pipe-based Context Switching	4000.0	45819.5	114.5
Process Creation	126.0	8140.6	646.1
Shell Scripts (1 concurrent)	42.4	6165.1	1454.0
Shell Scripts (8 concurrent)	6.0	1118.8	1864.6
System Call Overhead	15000.0	1839012.2	1226.0
System Benchmarks Index Score			734.2

Benchmark Run: Thu Mar 18 2010 23:22:05 - 23:50:05
2 CPUs in system; running 2 parallel copies of tests

Dhrystone 2 using register variables	16642419.7	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	4274.0	MWIPS	(10.0 s, 7 samples)
Execl Throughput	6668.7	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	379758.9	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	109045.6	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	965825.9	KBps	(30.0 s, 2 samples)
Pipe Throughput	1499057.4	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	444181.4	lps	(10.0 s, 7 samples)
Process Creation	18278.2	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	8743.4	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1132.0	lpm	(60.1 s, 2 samples)
System Call Overhead	2889360.1	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	16642419.7	1426.1
Double-Precision Whetstone	55.0	4274.0	777.1
Execl Throughput	43.0	6668.7	1550.9
File Copy 1024 bufsize 2000 maxblocks	3960.0	379758.9	959.0
File Copy 256 bufsize 500 maxblocks	1655.0	109045.6	658.9
File Copy 4096 bufsize 8000 maxblocks	5800.0	965825.9	1665.2
Pipe Throughput	12440.0	1499057.4	1205.0
Pipe-based Context Switching	4000.0	444181.4	1110.5
Process Creation	126.0	18278.2	1450.7
Shell Scripts (1 concurrent)	42.4	8743.4	2062.1

```
Shell Scripts (8 concurrent)                  6.0       1132.0    1886.6
System Call Overhead                      15000.0    2889360.1    1926.2
                                                                  ========
System Benchmarks Index Score                                      1313.2

   BYTE UNIX Benchmarks (Version 5.1.2)

   System: test: GNU/Linux
   OS: GNU/Linux -- 2.6.33-vanilla -- #1 SMP Sun Mar 14 20:20:39 EDT 2010
   Machine: i686 (unknown)
   Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
   CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5011.6 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   23:57:32 up  1:25,  1 user,  load average: 0.00, 1.41, 2.01; runlevel 2

------------------------------------------------------------------------

Benchmark Run: Thu Mar 18 2010 23:57:32 - 00:25:30
2 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables   8367215.8 lps   (10.0 s, 7 samples)
Double-Precision Whetstone                2137.6 MWIPS (10.0 s, 7 samples)
Execl Throughput                          3440.6 lps   (30.0 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks   336046.3 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks     104946.4 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks   726916.1 KBps  (30.0 s, 2 samples)
Pipe Throughput                         768411.2 lps   (10.0 s, 7 samples)
Pipe-based Context Switching             46155.6 lps   (10.0 s, 7 samples)
Process Creation                          8142.1 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)              6069.6 lpm   (60.0 s, 2 samples)
Shell Scripts (8 concurrent)              1114.2 lpm   (60.0 s, 2 samples)
System Call Overhead                   2167935.6 lps   (10.0 s, 7 samples)

System Benchmarks Index Values            BASELINE       RESULT    INDEX
Dhrystone 2 using register variables      116700.0    8367215.8    717.0
Double-Precision Whetstone                    55.0       2137.6    388.7
Execl Throughput                              43.0       3440.6    800.1
File Copy 1024 bufsize 2000 maxblocks       3960.0     336046.3    848.6
File Copy 256 bufsize 500 maxblocks         1655.0     104946.4    634.1
File Copy 4096 bufsize 8000 maxblocks       5800.0     726916.1   1253.3
Pipe Throughput                            12440.0     768411.2    617.7
Pipe-based Context Switching                4000.0      46155.6    115.4
Process Creation                             126.0       8142.1    646.2
```

Shell Scripts (1 concurrent)	42.4	6069.6	1431.5
Shell Scripts (8 concurrent)	6.0	1114.2	1857.1
System Call Overhead	15000.0	2167935.6	1445.3
System Benchmarks Index Score			739.2

Benchmark Run: Fri Mar 19 2010 00:25:30 - 00:53:32
2 CPUs in system; running 2 parallel copies of tests

Dhrystone 2 using register variables	16533367.1	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	4275.5	MWIPS	(10.0 s, 7 samples)
Execl Throughput	6612.4	lps	(29.8 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	367973.3	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	106643.2	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	957888.1	KBps	(30.0 s, 2 samples)
Pipe Throughput	1619968.7	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	442304.5	lps	(10.0 s, 7 samples)
Process Creation	18256.4	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	8731.3	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1128.7	lpm	(60.1 s, 2 samples)
System Call Overhead	2849212.1	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	16533367.1	1416.7
Double-Precision Whetstone	55.0	4275.5	777.4
Execl Throughput	43.0	6612.4	1537.8
File Copy 1024 bufsize 2000 maxblocks	3960.0	367973.3	929.2
File Copy 256 bufsize 500 maxblocks	1655.0	106643.2	644.4
File Copy 4096 bufsize 8000 maxblocks	5800.0	957888.1	1651.5
Pipe Throughput	12440.0	1619968.7	1302.2
Pipe-based Context Switching	4000.0	442304.5	1105.8
Process Creation	126.0	18256.4	1448.9
Shell Scripts (1 concurrent)	42.4	8731.3	2059.3
Shell Scripts (8 concurrent)	6.0	1128.7	1881.2
System Call Overhead	15000.0	2849212.1	1899.5
System Benchmarks Index Score			1310.7

BYTE UNIX Benchmarks (Version 5.1.2)

System: test: GNU/Linux
OS: GNU/Linux -- 2.6.33-vanilla -- #1 SMP Sun Mar 14 20:20:39 EDT 2010
Machine: i686 (unknown)
Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5010.5 bogomips)

```
        Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
            SYSEXIT, AMD virtualization, SYSCALL/SYSRET
    CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
        Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
            SYSEXIT, AMD virtualization, SYSCALL/SYSRET
    11:53:51 up 12 min,  2 users,  load average: 0.57, 0.60, 0.32; runlevel 2
```

Benchmark Run: Thu Mar 18 2010 11:53:51 - 12:21:48
2 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables	8168907.8	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	2137.6	MWIPS	(10.0 s, 7 samples)
Execl Throughput	3467.2	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	353845.1	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	121034.2	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	719062.9	KBps	(30.0 s, 2 samples)
Pipe Throughput	789139.0	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	45758.8	lps	(10.0 s, 7 samples)
Process Creation	8117.4	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	6126.8	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1119.8	lpm	(60.0 s, 2 samples)
System Call Overhead	1802747.8	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	8168907.8	700.0
Double-Precision Whetstone	55.0	2137.6	388.7
Execl Throughput	43.0	3467.2	806.3
File Copy 1024 bufsize 2000 maxblocks	3960.0	353845.1	893.5
File Copy 256 bufsize 500 maxblocks	1655.0	121034.2	731.3
File Copy 4096 bufsize 8000 maxblocks	5800.0	719062.9	1239.8
Pipe Throughput	12440.0	789139.0	634.4
Pipe-based Context Switching	4000.0	45758.8	114.4
Process Creation	126.0	8117.4	644.2
Shell Scripts (1 concurrent)	42.4	6126.8	1445.0
Shell Scripts (8 concurrent)	6.0	1119.8	1866.4
System Call Overhead	15000.0	1802747.8	1201.8
System Benchmarks Index Score			739.9

Benchmark Run: Thu Mar 18 2010 12:21:48 - 12:49:48
2 CPUs in system; running 2 parallel copies of tests

Dhrystone 2 using register variables	16666729.9	lps	(10.0 s, 7 samples)

```
Double-Precision Whetstone                      4273.5 MWIPS (10.0 s, 7 samples)
Execl Throughput                                 6633.6 lps   (29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks          381867.4 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks            110355.2 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks          968359.2 KBps  (30.0 s, 2 samples)
Pipe Throughput                               1760959.3 lps   (10.0 s, 7 samples)
Pipe-based Context Switching                   460490.4 lps   (10.0 s, 7 samples)
Process Creation                                17971.4 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)                     8671.7 lpm   (60.0 s, 2 samples)
Shell Scripts (8 concurrent)                     1133.2 lpm   (60.0 s, 2 samples)
System Call Overhead                          2859563.4 lps   (10.0 s, 7 samples)

System Benchmarks Index Values               BASELINE       RESULT    INDEX
Dhrystone 2 using register variables         116700.0     16666729.9   1428.2
Double-Precision Whetstone                       55.0         4273.5    777.0
Execl Throughput                                 43.0         6633.6   1542.7
File Copy 1024 bufsize 2000 maxblocks          3960.0       381867.4    964.3
File Copy 256 bufsize 500 maxblocks            1655.0       110355.2    666.8
File Copy 4096 bufsize 8000 maxblocks          5800.0       968359.2   1669.6
Pipe Throughput                               12440.0      1760959.3   1415.6
Pipe-based Context Switching                   4000.0       460490.4   1151.2
Process Creation                                126.0        17971.4   1426.3
Shell Scripts (1 concurrent)                     42.4         8671.7   2045.2
Shell Scripts (8 concurrent)                      6.0         1133.2   1888.6
System Call Overhead                          15000.0      2859563.4   1906.4
                                                                       ========
System Benchmarks Index Score                                          1332.9

   BYTE UNIX Benchmarks (Version 5.1.2)

   System: test: GNU/Linux
   OS: GNU/Linux -- 2.6.33-vanilla -- #1 SMP Sun Mar 14 20:20:39 EDT 2010
   Machine: i686 (unknown)
   Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
   CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5011.6 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   09:21:11 up 10:49,  1 user,  load average: 0.00, 0.00, 0.00; runlevel 2

------------------------------------------------------------------------

Benchmark Run: Fri Mar 19 2010 09:21:11 - 09:49:09
2 CPUs in system; running 1 parallel copy of tests
```

Dhrystone 2 using register variables	7983531.6	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	2137.6	MWIPS	(10.0 s, 7 samples)
Execl Throughput	3362.7	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	341778.1	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	112178.3	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	714380.9	KBps	(30.0 s, 2 samples)
Pipe Throughput	745231.9	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	45616.5	lps	(10.0 s, 7 samples)
Process Creation	8169.9	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	6064.3	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1111.1	lpm	(60.0 s, 2 samples)
System Call Overhead	2234557.4	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	7983531.6	684.1
Double-Precision Whetstone	55.0	2137.6	388.7
Execl Throughput	43.0	3362.7	782.0
File Copy 1024 bufsize 2000 maxblocks	3960.0	341778.1	863.1
File Copy 256 bufsize 500 maxblocks	1655.0	112178.3	677.8
File Copy 4096 bufsize 8000 maxblocks	5800.0	714380.9	1231.7
Pipe Throughput	12440.0	745231.9	599.1
Pipe-based Context Switching	4000.0	45616.5	114.0
Process Creation	126.0	8169.9	648.4
Shell Scripts (1 concurrent)	42.4	6064.3	1430.3
Shell Scripts (8 concurrent)	6.0	1111.1	1851.8
System Call Overhead	15000.0	2234557.4	1489.7
System Benchmarks Index Score			738.2

Benchmark Run: Fri Mar 19 2010 09:49:09 - 10:17:11
2 CPUs in system; running 2 parallel copies of tests

Dhrystone 2 using register variables	16299084.2	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	4275.2	MWIPS	(10.0 s, 7 samples)
Execl Throughput	6588.2	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	383280.1	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	106660.5	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	979477.5	KBps	(30.0 s, 2 samples)
Pipe Throughput	1556430.5	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	449086.5	lps	(10.0 s, 7 samples)
Process Creation	18112.8	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	8677.9	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1125.1	lpm	(60.0 s, 2 samples)
System Call Overhead	2865605.4	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	16299084.2	1396.7
Double-Precision Whetstone	55.0	4275.2	777.3
Execl Throughput	43.0	6588.2	1532.1
File Copy 1024 bufsize 2000 maxblocks	3960.0	383280.1	967.9
File Copy 256 bufsize 500 maxblocks	1655.0	106660.5	644.5
File Copy 4096 bufsize 8000 maxblocks	5800.0	979477.5	1688.8
Pipe Throughput	12440.0	1556430.5	1251.1
Pipe-based Context Switching	4000.0	449086.5	1122.7
Process Creation	126.0	18112.8	1437.5
Shell Scripts (1 concurrent)	42.4	8677.9	2046.7
Shell Scripts (8 concurrent)	6.0	1125.1	1875.2
System Call Overhead	15000.0	2865605.4	1910.4
System Benchmarks Index Score			1311.7

C.2 Patched kernel

C.2.1 Page table dump

```
---[ User Space ]---
0x00000000-0xc0000000          3G                          pmd
---[ Kernel Mapping ]---
0xc0000000-0xc0100000          1M     RW                GLB NX pte
0xc0100000-0xc03c1000       2820K     ro                GLB x  pte
0xc03c1000-0xc04fd000       1264K     ro                GLB NX pte
0xc04fd000-0xc0600000       1036K     RW                GLB NX pte
0xc0600000-0xf7800000        882M     RW            PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW                GLB NX pte
0xf79fe000-0xf7a00000          8K                          pte
0xf7a00000-0xf8000000          6M                          pmd
0xf8000000-0xf81fe000       2040K                          pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW         PCD     GLB NX pte
0xf81ff000-0xf8202000         12K                          pte
0xf8202000-0xf8203000          4K     RW         PCD     GLB NX pte
0xf8203000-0xf8206000         12K                          pte
0xf8206000-0xf8207000          4K     RW         PCD     GLB NX pte
0xf8207000-0xf8208000          4K                          pte
0xf8208000-0xf820f000         28K     RW         PCD     GLB NX pte
0xf820f000-0xf8212000         12K                          pte
0xf8212000-0xf8213000          4K     RW         PCD     GLB NX pte
0xf8213000-0xf8215000          8K                          pte
0xf8215000-0xf8220000         44K     RW                 GLB NX pte
0xf8220000-0xf8222000          8K                          pte
```

```
0xf8222000-0xf822c000      40K     ro              GLB x   pte
0xf822c000-0xf822d000       4K     ro              GLB NX  pte
0xf822d000-0xf8231000      16K     RW              GLB NX  pte
0xf8231000-0xf8233000       8K                             pte
0xf8233000-0xf8234000       4K     ro              GLB x   pte
0xf8234000-0xf8235000       4K     ro              GLB NX  pte
0xf8235000-0xf8237000       8K     RW              GLB NX  pte
0xf8237000-0xf823b000      16K                             pte
0xf823b000-0xf823d000       8K     ro              GLB x   pte
0xf823d000-0xf823e000       4K     ro              GLB NX  pte
0xf823e000-0xf8240000       8K     RW              GLB NX  pte
0xf8240000-0xf8242000       8K                             pte
0xf8242000-0xf8243000       4K     RW      PCD     GLB NX  pte
0xf8243000-0xf8246000      12K                             pte
0xf8246000-0xf8247000       4K     RW      PCD     GLB NX  pte
0xf8247000-0xf824a000      12K                             pte
0xf824a000-0xf824b000       4K     RW      PCD     GLB NX  pte
0xf824b000-0xf824d000       8K                             pte
0xf824d000-0xf825b000      56K     ro              GLB x   pte
0xf825b000-0xf8268000      52K     ro              GLB NX  pte
0xf8268000-0xf8272000      40K     RW              GLB NX  pte
0xf8272000-0xf8274000       8K                             pte
0xf8274000-0xf8278000      16K     ro              GLB x   pte
0xf8278000-0xf8279000       4K     ro              GLB NX  pte
0xf8279000-0xf827b000       8K     RW              GLB NX  pte
0xf827b000-0xf827f000      16K                             pte
0xf827f000-0xf8281000       8K     ro              GLB x   pte
0xf8281000-0xf8282000       4K     ro              GLB NX  pte
0xf8282000-0xf8284000       8K     RW              GLB NX  pte
0xf8284000-0xf8286000       8K                             pte
0xf8286000-0xf829b000      84K     ro              GLB x   pte
0xf829b000-0xf82a0000      20K     ro              GLB NX  pte
0xf82a0000-0xf82ab000      44K     RW              GLB NX  pte
0xf82ab000-0xf82ae000      12K                             pte
0xf82ae000-0xf82af000       4K     RW      PCD     GLB NX  pte
0xf82af000-0xf82b1000       8K                             pte
0xf82b1000-0xf82b8000      28K     ro              GLB x   pte
0xf82b8000-0xf82b9000       4K     ro              GLB NX  pte
0xf82b9000-0xf82bb000       8K     RW              GLB NX  pte
0xf82bb000-0xf82bd000       8K                             pte
0xf82bd000-0xf82be000       4K     ro              GLB x   pte
0xf82be000-0xf82bf000       4K     ro              GLB NX  pte
0xf82bf000-0xf82c1000       8K     RW              GLB NX  pte
0xf82c1000-0xf82c2000       4K                             pte
0xf82c2000-0xf82c3000       4K     RW      PCD     GLB NX  pte
```

```
0xf82c3000-0xf82c6000    12K                          pte
0xf82c6000-0xf82c7000     4K     RW     PCD     GLB NX pte
0xf82c7000-0xf82ca000    12K                          pte
0xf82ca000-0xf82cb000     4K     RW     PCD     GLB NX pte
0xf82cb000-0xf82cd000     8K                          pte
0xf82cd000-0xf82d8000    44K     ro             GLB x  pte
0xf82d8000-0xf82db000    12K     ro             GLB NX pte
0xf82db000-0xf82e2000    28K     RW             GLB NX pte
0xf82e2000-0xf82e4000     8K                          pte
0xf82e4000-0xf82e5000     4K     RW     PCD     GLB NX pte
0xf82e5000-0xf82e7000     8K                          pte
0xf82e7000-0xf82e8000     4K     ro             GLB x  pte
0xf82e8000-0xf82e9000     4K     ro             GLB NX pte
0xf82e9000-0xf82eb000     8K     RW             GLB NX pte
0xf82eb000-0xf82ee000    12K                          pte
0xf82ee000-0xf82ef000     4K     RW     PCD     GLB NX pte
0xf82ef000-0xf82f0000     4K                          pte
0xf82f0000-0xf82f1000     4K     RW     PCD     GLB NX pte
0xf82f1000-0xf82f4000    12K                          pte
0xf82f4000-0xf82f5000     4K     RW     PCD     GLB NX pte
0xf82f5000-0xf82f7000     8K                          pte
0xf82f7000-0xf82f8000     4K     ro             GLB x  pte
0xf82f8000-0xf82f9000     4K     ro             GLB NX pte
0xf82f9000-0xf82fb000     8K     RW             GLB NX pte
0xf82fb000-0xf82fd000     8K                          pte
0xf82fd000-0xf82fe000     4K     ro             GLB x  pte
0xf82fe000-0xf82ff000     4K     ro             GLB NX pte
0xf82ff000-0xf8301000     8K     RW             GLB NX pte
0xf8301000-0xf8303000     8K                          pte
0xf8303000-0xf8305000     8K     RW             GLB NX pte
0xf8305000-0xf8306000     4K                          pte
0xf8306000-0xf8307000     4K     RW     PCD     GLB NX pte
0xf8307000-0xf830a000    12K                          pte
0xf830a000-0xf830b000     4K     RW     PCD     GLB NX pte
0xf830b000-0xf830d000     8K                          pte
0xf830d000-0xf8312000    20K     ro             GLB x  pte
0xf8312000-0xf8313000     4K     ro             GLB NX pte
0xf8313000-0xf8315000     8K     RW             GLB NX pte
0xf8315000-0xf8317000     8K                          pte
0xf8317000-0xf8318000     4K     ro             GLB x  pte
0xf8318000-0xf8319000     4K     ro             GLB NX pte
0xf8319000-0xf831b000     8K     RW             GLB NX pte
0xf831b000-0xf831c000     4K                          pte
0xf831c000-0xf8321000    20K     ro             GLB x  pte
0xf8321000-0xf8323000     8K     ro             GLB NX pte
```

0xf8323000–0xf8325000	8K	RW	GLB NX	pte
0xf8325000–0xf8327000	8K			pte
0xf8327000–0xf832d000	24K	ro	GLB x	pte
0xf832d000–0xf832f000	8K	ro	GLB NX	pte
0xf832f000–0xf8333000	16K	RW	GLB NX	pte
0xf8333000–0xf8337000	16K			pte
0xf8337000–0xf8338000	4K	ro	GLB x	pte
0xf8338000–0xf8339000	4K	ro	GLB NX	pte
0xf8339000–0xf833b000	8K	RW	GLB NX	pte
0xf833b000–0xf833d000	8K			pte
0xf833d000–0xf833e000	4K	ro	GLB x	pte
0xf833e000–0xf833f000	4K	ro	GLB NX	pte
0xf833f000–0xf8341000	8K	RW	GLB NX	pte
0xf8341000–0xf8343000	8K			pte
0xf8343000–0xf8345000	8K	RW	GLB NX	pte
0xf8345000–0xf8346000	4K			pte
0xf8346000–0xf834b000	20K	ro	GLB x	pte
0xf834b000–0xf834d000	8K	ro	GLB NX	pte
0xf834d000–0xf834f000	8K	RW	GLB NX	pte
0xf834f000–0xf8351000	8K			pte
0xf8351000–0xf8352000	4K	ro	GLB x	pte
0xf8352000–0xf8353000	4K	ro	GLB NX	pte
0xf8353000–0xf8355000	8K	RW	GLB NX	pte
0xf8355000–0xf8356000	4K			pte
0xf8356000–0xf835a000	16K	ro	GLB x	pte
0xf835a000–0xf835e000	16K	ro	GLB NX	pte
0xf835e000–0xf8360000	8K	RW	GLB NX	pte
0xf8360000–0xf8362000	8K			pte
0xf8362000–0xf8378000	88K	ro	GLB x	pte
0xf8378000–0xf837c000	16K	ro	GLB NX	pte
0xf837c000–0xf8380000	16K	RW	GLB NX	pte
0xf8380000–0xf8382000	8K			pte
0xf8382000–0xf8386000	16K	ro	GLB x	pte
0xf8386000–0xf8388000	8K	ro	GLB NX	pte
0xf8388000–0xf838b000	12K	RW	GLB NX	pte
0xf838b000–0xf838d000	8K			pte
0xf838d000–0xf838e000	4K	ro	GLB x	pte
0xf838e000–0xf838f000	4K	ro	GLB NX	pte
0xf838f000–0xf8391000	8K	RW	GLB NX	pte
0xf8391000–0xf8393000	8K			pte
0xf8393000–0xf8394000	4K	ro	GLB x	pte
0xf8394000–0xf8395000	4K	ro	GLB NX	pte
0xf8395000–0xf8397000	8K	RW	GLB NX	pte
0xf8397000–0xf8399000	8K			pte
0xf8399000–0xf839c000	12K	ro	GLB x	pte

0xf839c000	0xf839d000	4K	ro	GLB NX	pte
0xf839d000	0xf83a0000	12K	RW	GLB NX	pte
0xf83a0000	0xf83a2000	8K			pte
0xf83a2000	0xf83a3000	4K	ro	GLB x	pte
0xf83a3000	0xf83a4000	4K	ro	GLB NX	pte
0xf83a4000	0xf83a6000	8K	RW	GLB NX	pte
0xf83a6000	0xf83a8000	8K			pte
0xf83a8000	0xf83a9000	4K	ro	GLB x	pte
0xf83a9000	0xf83aa000	4K	ro	GLB NX	pte
0xf83aa000	0xf83ac000	8K	RW	GLB NX	pte
0xf83ac000	0xf83ad000	4K			pte
0xf83ad000	0xf83af000	8K	ro	GLB x	pte
0xf83af000	0xf83b0000	4K	ro	GLB NX	pte
0xf83b0000	0xf83b3000	12K	RW	GLB NX	pte
0xf83b3000	0xf83b5000	8K			pte
0xf83b5000	0xf83b6000	4K	ro	GLB x	pte
0xf83b6000	0xf83b7000	4K	ro	GLB NX	pte
0xf83b7000	0xf83b9000	8K	RW	GLB NX	pte
0xf83b9000	0xf83ba000	4K			pte
0xf83ba000	0xf83bc000	8K	ro	GLB x	pte
0xf83bc000	0xf83bd000	4K	ro	GLB NX	pte
0xf83bd000	0xf83bf000	8K	RW	GLB NX	pte
0xf83bf000	0xf83c1000	8K			pte
0xf83c1000	0xf83c5000	16K	ro	GLB x	pte
0xf83c5000	0xf83c7000	8K	ro	GLB NX	pte
0xf83c7000	0xf83ca000	12K	RW	GLB NX	pte
0xf83ca000	0xf83cc000	8K			pte
0xf83cc000	0xf83d1000	20K	ro	GLB x	pte
0xf83d1000	0xf83d3000	8K	ro	GLB NX	pte
0xf83d3000	0xf83d6000	12K	RW	GLB NX	pte
0xf83d6000	0xf83d8000	8K			pte
0xf83d8000	0xf83dc000	16K	ro	GLB x	pte
0xf83dc000	0xf83de000	8K	ro	GLB NX	pte
0xf83de000	0xf83e0000	8K	RW	GLB NX	pte
0xf83e0000	0xf83e2000	8K			pte
0xf83e2000	0xf83e7000	20K	ro	GLB x	pte
0xf83e7000	0xf83e9000	8K	ro	GLB NX	pte
0xf83e9000	0xf83eb000	8K	RW	GLB NX	pte
0xf83eb000	0xf83ed000	8K			pte
0xf83ed000	0xf8402000	84K	ro	GLB x	pte
0xf8402000	0xf8405000	12K	ro	GLB NX	pte
0xf8405000	0xf840c000	28K	RW	GLB NX	pte
0xf840c000	0xf840e000	8K			pte
0xf840e000	0xf8411000	12K	ro	GLB x	pte
0xf8411000	0xf8412000	4K	ro	GLB NX	pte

0xf8412000–0xf8414000	8K	RW		GLB NX	pte
0xf8414000–0xf8415000	4K				pte
0xf8415000–0xf8416000	4K	ro		GLB x	pte
0xf8416000–0xf8417000	4K	ro		GLB NX	pte
0xf8417000–0xf8419000	8K	RW		GLB NX	pte
0xf8419000–0xf841b000	8K				pte
0xf841b000–0xf8432000	92K	ro		GLB x	pte
0xf8432000–0xf8439000	28K	ro		GLB NX	pte
0xf8439000–0xf8445000	48K	RW		GLB NX	pte
0xf8445000–0xf8447000	8K				pte
0xf8447000–0xf844a000	12K	ro		GLB x	pte
0xf844a000–0xf844b000	4K	ro		GLB NX	pte
0xf844b000–0xf844d000	8K	RW		GLB NX	pte
0xf844d000–0xf844f000	8K				pte
0xf844f000–0xf8452000	12K	ro		GLB x	pte
0xf8452000–0xf8453000	4K	ro		GLB NX	pte
0xf8453000–0xf8455000	8K	RW		GLB NX	pte
0xf8455000–0xf845c000	28K				pte
0xf845c000–0xf845e000	8K	ro		GLB x	pte
0xf845e000–0xf845f000	4K	ro		GLB NX	pte
0xf845f000–0xf8462000	12K	RW		GLB NX	pte
0xf8462000–0xf8464000	8K				pte
0xf8464000–0xf8466000	8K	ro		GLB x	pte
0xf8466000–0xf8467000	4K	ro		GLB NX	pte
0xf8467000–0xf8469000	8K	RW		GLB NX	pte
0xf8469000–0xf8470000	28K				pte
0xf8470000–0xf847f000	60K	RW		GLB NX	pte
0xf847f000–0xf8480000	4K				pte
0xf8480000–0xf84a0000	128K	RW	PCD	GLB NX	pte
0xf84a0000–0xf84a2000	8K				pte
0xf84a2000–0xf84d7000	212K	RW		GLB NX	pte
0xf84d7000–0xf8a79000	5768K				pte
0xf8a79000–0xf8aa5000	176K	ro		GLB x	pte
0xf8aa5000–0xf8aaa000	20K	ro		GLB NX	pte
0xf8aaa000–0xf8ab5000	44K	RW		GLB NX	pte
0xf8ab5000–0xf8c00000	1324K				pte
0xf8c00000–0xff600000	106M				pmd
0xff600000–0xff602000	8K	RW		GLB NX	pte
0xff602000–0xff680000	504K				pte
0xff680000–0xff682000	8K	RW		GLB NX	pte
0xff682000–0xff700000	504K				pte
0xff700000–0xff702000	8K	RW		GLB NX	pte
0xff702000–0xff780000	504K				pte
0xff780000–0xff782000	8K	RW		GLB NX	pte
0xff782000–0xff800000	504K				pte

```
0xff800000-0xffa00000     2M                              pmd
---[ vmalloc() End ]---
0xffa00000-0xffa01000     4K     RW             GLB NX pte
---[ Persisent kmap() Area ]---
0xffa01000-0xffa08000    28K     RW             GLB NX pte
0xffa08000-0xfff1f000  5212K                            pte
---[ Fixmap Area ]---
0xfff1f000-0xfff22000    12K                            pte
0xfff22000-0xfff23000     4K     RW PWT PCD     GLB NX pte
0xfff23000-0xffffa000   860K                            pte
0xffffa000-0xffffc000     8K     RW PWT PCD     GLB NX pte
0xffffc000-0x00000000    16K                            pte
```

C.2.2 Loaded module listing

```
Module                  Size  Used by
ipv6                  244751  18
iptable_filter         12721  0
ip_tables              17573  1 iptable_filter
x_tables               21407  1 ip_tables
af_packet              22051  2
sbp2                   22884  0
lp                     13358  0
loop                   22191  0
e1000e                124181  0
i2c_piix4              12983  0
shpchp                 36453  0
parport_pc             32057  1
parport                40874  2 lp,parport_pc
processor              34501  0
button                 13710  0
i2c_core               27717  1 i2c_piix4
k8temp                 12835  0
serio_raw              13055  0
pci_hotplug            34495  1 shpchp
evdev                  17297  0
tpm_tis                13969  0
tpm                    21006  1 tpm_tis
tpm_bios               13444  1 tpm
pcspkr                 12614  0
ext3                  122771  1
jbd                    46583  1 ext3
mbcache                14478  1 ext3
sd_mod                 34693  3
sg                     31555  0
atiixp                 12789  0
```

```
pata_acpi              12875  0
pata_atiixp            12932  2
ohci1394               35323  0
ahci                   39973  0
ata_generic            12779  0
ieee1394               84169  2 sbp2,ohci1394
ehci_hcd               39713  0
ohci_hcd               26659  0
libata                170141  4 pata_acpi,pata_atiixp,ahci,ata_generic
usbcore               148072  3 ehci_hcd,ohci_hcd
scsi_mod              148245  4 sbp2,sd_mod,sg,libata
thermal                18772  0
fan                    12935  0
thermal_sys            19300  3 processor,thermal,fan
fuse                   59731  1
```

C.2.3 UnixBench test results

```
BYTE UNIX Benchmarks (Version 5.1.2)

System: test: GNU/Linux
OS: GNU/Linux -- 2.6.33-ronx -- #1 SMP Mon Mar 15 08:23:29 EDT 2010
Machine: i686 (unknown)
Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5011.8 bogomips)
       Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
          SYSEXIT, AMD virtualization, SYSCALL/SYSRET
CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
       Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
          SYSEXIT, AMD virtualization, SYSCALL/SYSRET
20:54:27 up  4:04,  1 user,  load average: 0.00, 0.00, 0.00; runlevel 2

------------------------------------------------------------------------

Benchmark Run: Thu Mar 18 2010 20:54:27 - 21:22:26
2 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables       8362834.4 lps   (10.0 s, 7 samples)
Double-Precision Whetstone                    2137.6 MWIPS (10.0 s, 7 samples)
Execl Throughput                              3408.8 lps   (29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks       344523.5 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks         113206.1 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks       723748.7 KBps  (30.0 s, 2 samples)
Pipe Throughput                             688761.7 lps   (10.0 s, 7 samples)
Pipe-based Context Switching                 45414.6 lps   (10.0 s, 7 samples)
Process Creation                              8095.7 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)                  6049.5 lpm   (60.0 s, 2 samples)
```

Shell Scripts (8 concurrent)	1103.6	lpm	(60.0 s, 2 samples)
System Call Overhead	2195860.4	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	8362834.4	716.6
Double-Precision Whetstone	55.0	2137.6	388.7
Execl Throughput	43.0	3408.8	792.7
File Copy 1024 bufsize 2000 maxblocks	3960.0	344523.5	870.0
File Copy 256 bufsize 500 maxblocks	1655.0	113206.1	684.0
File Copy 4096 bufsize 8000 maxblocks	5800.0	723748.7	1247.8
Pipe Throughput	12440.0	688761.7	553.7
Pipe-based Context Switching	4000.0	45414.6	113.5
Process Creation	126.0	8095.7	642.5
Shell Scripts (1 concurrent)	42.4	6049.5	1426.8
Shell Scripts (8 concurrent)	6.0	1103.6	1839.3
System Call Overhead	15000.0	2195860.4	1463.9
System Benchmarks Index Score			736.4

Benchmark Run: Thu Mar 18 2010 21:22:26 - 21:50:29
2 CPUs in system; running 2 parallel copies of tests

Dhrystone 2 using register variables	16751876.9	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	4273.4	MWIPS	(10.0 s, 7 samples)
Execl Throughput	6589.5	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	369116.4	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	109721.1	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	965942.5	KBps	(30.0 s, 2 samples)
Pipe Throughput	1555463.8	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	428903.5	lps	(10.0 s, 7 samples)
Process Creation	17806.5	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	8635.8	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1121.0	lpm	(60.1 s, 2 samples)
System Call Overhead	2891030.5	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	16751876.9	1435.5
Double-Precision Whetstone	55.0	4273.4	777.0
Execl Throughput	43.0	6589.5	1532.5
File Copy 1024 bufsize 2000 maxblocks	3960.0	369116.4	932.1
File Copy 256 bufsize 500 maxblocks	1655.0	109721.1	663.0
File Copy 4096 bufsize 8000 maxblocks	5800.0	965942.5	1665.4
Pipe Throughput	12440.0	1555463.8	1250.4
Pipe-based Context Switching	4000.0	428903.5	1072.3

```
Process Creation                          126.0      17806.5   1413.2
Shell Scripts (1 concurrent)               42.4       8635.8   2036.8
Shell Scripts (8 concurrent)                6.0       1121.0   1868.3
System Call Overhead                    15000.0    2891030.5   1927.4
                                                               ========
System Benchmarks Index Score                                  1305.2

   BYTE UNIX Benchmarks (Version 5.1.2)

   System: test: GNU/Linux
   OS: GNU/Linux -- 2.6.33-ronx -- #1 SMP Mon Mar 15 08:23:29 EDT 2010
   Machine: i686 (unknown)
   Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
   CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5010.4 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   11:48:19 up 2 min,  1 user,  load average: 0.02, 0.03, 0.00; runlevel 2

------------------------------------------------------------------------

Benchmark Run: Fri Mar 19 2010 11:48:19 - 12:16:17
2 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables   8365125.1 lps   (10.0 s, 7 samples)
Double-Precision Whetstone                2137.6 MWIPS (10.0 s, 7 samples)
Execl Throughput                          3542.5 lps   (29.8 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks   332961.7 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks     112963.9 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks   753050.0 KBps  (30.0 s, 2 samples)
Pipe Throughput                         871440.8 lps   (10.0 s, 7 samples)
Pipe-based Context Switching             46029.3 lps   (10.0 s, 7 samples)
Process Creation                          8207.4 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)              6133.7 lpm   (60.0 s, 2 samples)
Shell Scripts (8 concurrent)              1119.8 lpm   (60.0 s, 2 samples)
System Call Overhead                   2352684.9 lps   (10.0 s, 7 samples)

System Benchmarks Index Values            BASELINE       RESULT    INDEX
Dhrystone 2 using register variables      116700.0    8365125.1    716.8
Double-Precision Whetstone                    55.0       2137.6    388.7
Execl Throughput                              43.0       3542.5    823.8
File Copy 1024 bufsize 2000 maxblocks       3960.0     332961.7    840.8
File Copy 256 bufsize 500 maxblocks         1655.0     112963.9    682.6
File Copy 4096 bufsize 8000 maxblocks       5800.0     753050.0   1298.4
Pipe Throughput                            12440.0     871440.8    700.5
```

Pipe-based Context Switching	4000.0	46029.3	115.1
Process Creation	126.0	8207.4	651.4
Shell Scripts (1 concurrent)	42.4	6133.7	1446.6
Shell Scripts (8 concurrent)	6.0	1119.8	1866.4
System Call Overhead	15000.0	2352684.9	1568.5
System Benchmarks Index Score			761.5

Benchmark Run: Fri Mar 19 2010 12:16:17 - 12:44:19
2 CPUs in system; running 2 parallel copies of tests

Dhrystone 2 using register variables	16719531.6	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	4275.2	MWIPS	(10.0 s, 7 samples)
Execl Throughput	6642.5	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	373513.6	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	109739.8	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	945366.3	KBps	(30.0 s, 2 samples)
Pipe Throughput	1602392.2	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	435733.9	lps	(10.0 s, 7 samples)
Process Creation	17547.9	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	8662.5	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1124.4	lpm	(60.1 s, 2 samples)
System Call Overhead	2903261.0	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	16719531.6	1432.7
Double-Precision Whetstone	55.0	4275.2	777.3
Execl Throughput	43.0	6642.5	1544.8
File Copy 1024 bufsize 2000 maxblocks	3960.0	373513.6	943.2
File Copy 256 bufsize 500 maxblocks	1655.0	109739.8	663.1
File Copy 4096 bufsize 8000 maxblocks	5800.0	945366.3	1629.9
Pipe Throughput	12440.0	1602392.2	1288.1
Pipe-based Context Switching	4000.0	435733.9	1089.3
Process Creation	126.0	17547.9	1392.7
Shell Scripts (1 concurrent)	42.4	8662.5	2043.1
Shell Scripts (8 concurrent)	6.0	1124.4	1874.0
System Call Overhead	15000.0	2903261.0	1935.5
System Benchmarks Index Score			1309.4

BYTE UNIX Benchmarks (Version 5.1.2)

System: test: GNU/Linux
OS: GNU/Linux -- 2.6.33-ronx -- #1 SMP Mon Mar 15 08:23:29 EDT 2010
Machine: i686 (unknown)

```
Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5010.4 bogomips)
       Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
           SYSEXIT, AMD virtualization, SYSCALL/SYSRET
CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
       Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
           SYSEXIT, AMD virtualization, SYSCALL/SYSRET
 12:44:19 up 58 min,  1 user,  load average: 11.80, 6.56, 3.35; runlevel 2
```

```
Benchmark Run: Fri Mar 19 2010 12:44:19 - 13:12:18
2 CPUs in system; running 1 parallel copy of tests
```

Dhrystone 2 using register variables	8395190.7	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	2135.8	MWIPS	(10.1 s, 7 samples)
Execl Throughput	3419.8	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	317434.9	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	104708.0	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	701286.8	KBps	(30.0 s, 2 samples)
Pipe Throughput	795437.7	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	45482.0	lps	(10.0 s, 7 samples)
Process Creation	8159.1	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	6105.0	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1114.0	lpm	(60.0 s, 2 samples)
System Call Overhead	2348292.5	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	8395190.7	719.4
Double-Precision Whetstone	55.0	2135.8	388.3
Execl Throughput	43.0	3419.8	795.3
File Copy 1024 bufsize 2000 maxblocks	3960.0	317434.9	801.6
File Copy 256 bufsize 500 maxblocks	1655.0	104708.0	632.7
File Copy 4096 bufsize 8000 maxblocks	5800.0	701286.8	1209.1
Pipe Throughput	12440.0	795437.7	639.4
Pipe-based Context Switching	4000.0	45482.0	113.7
Process Creation	126.0	8159.1	647.5
Shell Scripts (1 concurrent)	42.4	6105.0	1439.9
Shell Scripts (8 concurrent)	6.0	1114.0	1856.7
System Call Overhead	15000.0	2348292.5	1565.5
System Benchmarks Index Score			739.7

```
Benchmark Run: Fri Mar 19 2010 13:12:18 - 13:40:19
2 CPUs in system; running 2 parallel copies of tests
```

```
Dhrystone 2 using register variables      16708061.9 lps   (10.0 s, 7 samples)
Double-Precision Whetstone                    4274.0 MWIPS (10.0 s, 7 samples)
Execl Throughput                              6549.6 lps   (29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks       375506.2 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks         106349.7 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks       961702.4 KBps  (30.0 s, 2 samples)
Pipe Throughput                            1711486.5 lps   (10.0 s, 7 samples)
Pipe-based Context Switching                442213.5 lps   (10.0 s, 7 samples)
Process Creation                             18047.3 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)                  8677.8 lpm   (60.0 s, 2 samples)
Shell Scripts (8 concurrent)                  1126.7 lpm   (60.0 s, 2 samples)
System Call Overhead                       2958635.1 lps   (10.0 s, 7 samples)

System Benchmarks Index Values               BASELINE       RESULT    INDEX
Dhrystone 2 using register variables         116700.0   16708061.9   1431.7
Double-Precision Whetstone                       55.0       4274.0    777.1
Execl Throughput                                 43.0       6549.6   1523.2
File Copy 1024 bufsize 2000 maxblocks          3960.0     375506.2    948.2
File Copy 256 bufsize 500 maxblocks            1655.0     106349.7    642.6
File Copy 4096 bufsize 8000 maxblocks          5800.0     961702.4   1658.1
Pipe Throughput                               12440.0    1711486.5   1375.8
Pipe-based Context Switching                   4000.0     442213.5   1105.5
Process Creation                                126.0      18047.3   1432.3
Shell Scripts (1 concurrent)                     42.4       8677.8   2046.7
Shell Scripts (8 concurrent)                      6.0       1126.7   1877.8
System Call Overhead                          15000.0    2958635.1   1972.4
                                                                     ========
System Benchmarks Index Score                                         1321.1

   BYTE UNIX Benchmarks (Version 5.1.2)

   System: test: GNU/Linux
   OS: GNU/Linux -- 2.6.33-ronx -- #1 SMP Mon Mar 15 08:23:29 EDT 2010
   Machine: i686 (unknown)
   Language: en_US.utf8 (charmap="UTF-8", collate="UTF-8")
   CPU 0: AMD Athlon(tm) Dual Core Processor 4850e (5010.4 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   CPU 1: AMD Athlon(tm) Dual Core Processor 4850e (5010.7 bogomips)
          Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/
              SYSEXIT, AMD virtualization, SYSCALL/SYSRET
   13:40:19 up  1:54,  1 user,  load average: 11.09, 6.37, 3.37; runlevel 2

------------------------------------------------------------------------

Benchmark Run: Fri Mar 19 2010 13:40:19 - 14:08:18
```

2 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables	8375083.2	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	2136.7	MWIPS	(10.0 s, 7 samples)
Execl Throughput	3455.1	lps	(29.8 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	309652.2	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	97676.8	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	689817.6	KBps	(30.0 s, 2 samples)
Pipe Throughput	803691.7	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	45701.5	lps	(10.0 s, 7 samples)
Process Creation	7986.8	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	6062.2	lpm	(60.0 s, 2 samples)
Shell Scripts (8 concurrent)	1113.0	lpm	(60.0 s, 2 samples)
System Call Overhead	2235644.3	lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	8375083.2	717.7
Double-Precision Whetstone	55.0	2136.7	388.5
Execl Throughput	43.0	3455.1	803.5
File Copy 1024 bufsize 2000 maxblocks	3960.0	309652.2	782.0
File Copy 256 bufsize 500 maxblocks	1655.0	97676.8	590.2
File Copy 4096 bufsize 8000 maxblocks	5800.0	689817.6	1189.3
Pipe Throughput	12440.0	803691.7	646.1
Pipe-based Context Switching	4000.0	45701.5	114.3
Process Creation	126.0	7986.8	633.9
Shell Scripts (1 concurrent)	42.4	6062.2	1429.8
Shell Scripts (8 concurrent)	6.0	1113.0	1855.0
System Call Overhead	15000.0	2235644.3	1490.4
System Benchmarks Index Score			729.6

Benchmark Run: Fri Mar 19 2010 14:08:18 - 14:36:20
2 CPUs in system; running 2 parallel copies of tests

Dhrystone 2 using register variables	16731918.0	lps	(10.0 s, 7 samples)
Double-Precision Whetstone	4275.2	MWIPS	(10.0 s, 7 samples)
Execl Throughput	6428.4	lps	(29.9 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks	364152.5	KBps	(30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks	104320.3	KBps	(30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks	935224.7	KBps	(30.0 s, 2 samples)
Pipe Throughput	1678894.4	lps	(10.0 s, 7 samples)
Pipe-based Context Switching	449983.8	lps	(10.0 s, 7 samples)
Process Creation	17692.7	lps	(30.0 s, 2 samples)
Shell Scripts (1 concurrent)	8644.6	lpm	(60.0 s, 2 samples)

Shell Scripts (8 concurrent)	1121.1 lpm	(60.0 s, 2 samples)
System Call Overhead	2967691.7 lps	(10.0 s, 7 samples)

System Benchmarks Index Values	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	16731918.0	1433.8
Double-Precision Whetstone	55.0	4275.2	777.3
Execl Throughput	43.0	6428.4	1495.0
File Copy 1024 bufsize 2000 maxblocks	3960.0	364152.5	919.6
File Copy 256 bufsize 500 maxblocks	1655.0	104320.3	630.3
File Copy 4096 bufsize 8000 maxblocks	5800.0	935224.7	1612.5
Pipe Throughput	12440.0	1678894.4	1349.6
Pipe-based Context Switching	4000.0	449983.8	1125.0
Process Creation	126.0	17692.7	1404.2
Shell Scripts (1 concurrent)	42.4	8644.6	2038.8
Shell Scripts (8 concurrent)	6.0	1121.1	1868.5
System Call Overhead	15000.0	2967691.7	1978.5
System Benchmarks Index Score			1307.8