Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions

Tzi-cker Chiueh
Ganesh Venkitachalam
Prashant Pradhan

Computer Science Department
State University of New York
Stony Brook, NY 11794-4400

http://www.ecsl.cs.sunysb.edu/palladium.html
Introduction

- Dynamic extensibility emerges as the major research theme and product trend

  *Extensible operating systems: Windows NT*
  *Extensible database systems: Informix, DB2, Oracle*
  *Extensible applications: Adobe’s Premiere, Apache Web Server*
  *Active Networking*

- Component-based software development methodology

  *A single application consists of components produced by multiple vendors ⇒ Whose bugs cause application malfunction?*

- Need an Intra-address space protection mechanism to quarantine erroneous or malicious software components

*Integrating segmentation and paging protection for safe, efficient, and transparent software extensions*
Palladium

- A Linux-based system that supports safe user-level and kernel-level software extensions using Intel X86 architecture’s segmentation and paging hardware

- Provide the same level of protection as using separate address spaces

- Fastest protection domain switching: 142 CPU cycles for a null protected procedure call and return

- Minimal changes required to existing programming tools and conventional linear-address-space programming model
**X86 Virtual Memory Hardware**

- Virtual Address: 16-bit segment selector and 32-bit offset

  \[ \text{Virtual Address} \rightarrow \text{Linear Address} \rightarrow \text{Physical Address} \]

  - Segmentation
  - Paging

- Segment-level Protection Check
  
  4 Segment Protection Levels (SPL)
  
  Segment Limit

- Page-level Protection Check
  
  2 Page Protection Levels (PPL)
  
  Read/Write Permission
Integrating segmentation and paging protection for safe, efficient, and transparent software extensions
X86 Virtual Memory Hardware

- Mapping between SPL and PPL
  - SPL 0, 1, 2 → PPL 0
  - SPL 3 → PPL 1

- Control transfer among protection domains
  - lcall call-gate-ID          lret
    Switch to the stack associated with the destination SPL

- Only supports transfer starting from more privileged level to less privileged level and back

- On a process switch, page-table base address register is reloaded and TLB is flushed
Palladium’s Extension Programming Model

- A main program (kernel or extensible application) is protected from its dynamically-linked extension modules, but not vice versa.

- Extensions are protected function calls. Among extension modules, only safety-strength but not security-strength protection.

- Shared data regions between protection domains are available to reduce data copying.

- User-level extensions make system calls through hosting applications; kernel-level extensions are allowed to access only selective core kernel services.
Integrating segmentation and paging protection for safe, efficient, and transparent software extensions

Linux’s Virtual Address Space

Kernel Code Segment
SPL=0
PPL=0

Kernel Data/Stack Segment
SPL=0
PPL=0

User Code Segment
SPL=3
PPL=1

User Data/Stack Segment
SPL=3
PPL=1

Kernel

4GB

Stack

3GB

Relocated Shared Library

3GB

Heap

0GB

BSS

0GB

Global Offset Table

0GB

Data

0GB

Text

0GB

Procedure Linkage Table

0GB
Integrating segmentation and paging protection for safe, efficient, and transparent software extensions

Kernel Extension Mechanism

Kernel Extension Segment

SPL=1, PPL=0

Extension-1

Extension-2

Kernel Extension Segment

SPL=1, PPL=0

Kernel

User

Kernel Data/Stack Segment

SPL=0

PPL=0

Kernel Code Segment

SPL=0

PPL=0

4GB

3GB

0GB

8
Kernel Extension Mechanism

- Allow multiple extension segments, each of which can hold multiple extension modules that are loaded dynamically via `insmod`
- One stack per extension segment. Modules loaded into the same segment cannot run concurrently
- Kernel extension modules can access selective core kernel services such as `kmalloc`
- Kernel service functions called by kernel extensions execute in the context of the kernel stack of the triggering user process or the “Idle” process
- Shared data region allocated in extension segment
Kernel Extension Mechanism

1. System Call
2. System Call Table
3. Extension Function Table
4. Kernel Service
5. Extension Stack Frame
6. Extension Function
7. Kernel Function Table
8. Kernel Function
9. Per-Process Kernel Stack
10. User Process P

Integrating segmentation and paging protection for safe, efficient, and transparent software extensions
User-Level Extension Mechanism

Why the segmentation approach is not good?

- Passing data/code pointers between protection domains requires swizzling because of different base addresses.

- Gcc and ld need to be modified, because they assume a flat linear address space.

- Difficult to support stateful shared library routines such as fprintf().

Solution: Combining page-level and segment-level protection checks.
Integrating segmentation and paging protection for safe, efficient, and transparent software extensions

User-Level Extension Mechanism

Kernel

User

Extension-1

Extension-2

shared

User Data/Stack Segment

User Code Segment

4GB

3GB

0GB

SPL = 3
PPL = 1

SPL = 3
PPL = 1

SPL = 2
PPL = 1

SPL = 0
PPL = 0

SPL = 2
PPL = 0

SPL = 2
PPL = 0
Programming Interface

- Use seg_dlopen, seg_dlsym and seg_dlclos to load access and close dynamically-loaded modules
- Call init_PL in the beginning to be safely extensible
- Use set_range to expose shared library code pages
- Use set_call_gate to package application service functions that user extensions can invoke
- Use xmalloc rather than malloc
- Invoke gcc with a specific linker script to ensure that Global Offset Table be placed on a separate page
Implementation Issues

- X86 architecture’s lcall goes from less-privileged level to more-privileged level, and lret for the other direction

- Gcc and ld do not know segments

Solution: Add one level of indirection by dynamically generating code sequences to hide inter-domain control transfers and call/return semantic mismatch

- Seg_dlsym returns modified function pointers

- Saving SP and BP at user space to avoid system calls
Integrating segmentation and paging protection for safe, efficient, and transparent software extensions

**Control Transfer**

<table>
<thead>
<tr>
<th>Application Segment (SPL = 2)</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prepare:</strong></td>
<td></td>
</tr>
<tr>
<td>pushl 0x4(%esp)</td>
<td></td>
</tr>
<tr>
<td>popl ExtensionStack</td>
<td></td>
</tr>
<tr>
<td>movl %esp, SP2</td>
<td></td>
</tr>
<tr>
<td>movl %ebp, BP2</td>
<td></td>
</tr>
<tr>
<td>push ExtensionStackSegment</td>
<td></td>
</tr>
<tr>
<td>pushl ExtensionStackPointer</td>
<td></td>
</tr>
<tr>
<td>push ExtensionCodeSegment</td>
<td></td>
</tr>
<tr>
<td>push Transfer</td>
<td></td>
</tr>
<tr>
<td>lret</td>
<td></td>
</tr>
</tbody>
</table>

| AppCallGate:                  |             |
| mov SP2, %esp                 |             |
| mov BP2, %ebp                 |             |
| ret                           |             |

| Transfer:                     |             |
| call ExtensionFunction        |             |
| lcall AppCallGateNum          |             |

<table>
<thead>
<tr>
<th>Extension Segment (SPL = 3)</th>
<th>Extension Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prepare:</strong></td>
<td></td>
</tr>
<tr>
<td>pushl 0x4(%esp)</td>
<td></td>
</tr>
<tr>
<td>popl ExtensionStack</td>
<td></td>
</tr>
<tr>
<td>movl %esp, SP2</td>
<td></td>
</tr>
<tr>
<td>movl %ebp, BP2</td>
<td></td>
</tr>
<tr>
<td>push ExtensionStackSegment</td>
<td></td>
</tr>
<tr>
<td>pushl ExtensionStackPointer</td>
<td></td>
</tr>
<tr>
<td>push ExtensionCodeSegment</td>
<td></td>
</tr>
<tr>
<td>push Transfer</td>
<td></td>
</tr>
<tr>
<td>lret</td>
<td></td>
</tr>
</tbody>
</table>

Local call and return between the application and extension segments.
Kernel Support

- Additional protection check at page fault time based on calling code segment’s SPL and faulted page’s PPL

- System call check based on application’s SPL and the calling code segment’s SPL

- A lifetime timer is set at the beginning of invoking an extension to prevent “infinite-loop” bugs. Timer value is left to be a policy issue

- Deliver signals to user applications when protection faults arise or lifetime timers expire. No support for system state cleanup other than resource reclamation
Performance Results

Micro-Benchmark: Null protected procedure call

<table>
<thead>
<tr>
<th>Component</th>
<th>Intra-Domain</th>
<th>Inter-Domain</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up stack</td>
<td>2</td>
<td>26</td>
<td>5</td>
</tr>
<tr>
<td>Calling function</td>
<td>3</td>
<td>34</td>
<td>22</td>
</tr>
<tr>
<td>Returning to caller</td>
<td>3</td>
<td>75</td>
<td>44</td>
</tr>
<tr>
<td>Restoring state</td>
<td>2</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Total Cost</td>
<td>10</td>
<td>142</td>
<td>89</td>
</tr>
</tbody>
</table>
Performance Results

Micro-Benchmark: Reverse-string function with different input string sizes, in micro-seconds

<table>
<thead>
<tr>
<th>String Size (Bytes)</th>
<th>Unprotected Call</th>
<th>Palladium Call</th>
<th>Linux RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>2.20</td>
<td>2.79</td>
<td>349.19</td>
</tr>
<tr>
<td>64</td>
<td>4.06</td>
<td>4.65</td>
<td>352.55</td>
</tr>
<tr>
<td>128</td>
<td>7.78</td>
<td>8.37</td>
<td>374.20</td>
</tr>
<tr>
<td>256</td>
<td>15.22</td>
<td>15.97</td>
<td>423.33</td>
</tr>
</tbody>
</table>
### Performance Results

**Macro-Benchmark: user-level extension**

Fast CGI Invocation -- CGI script running as a protected function call, in requests/sec

<table>
<thead>
<tr>
<th>HTML file size</th>
<th>CGI</th>
<th>FastCGI</th>
<th>LibCGI (Palladium)</th>
<th>LibCGI (unprotected)</th>
<th>Web Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>28Byte</td>
<td>98</td>
<td>193</td>
<td>437</td>
<td>448</td>
<td>460</td>
</tr>
<tr>
<td>1KByte</td>
<td>92</td>
<td>188</td>
<td>423</td>
<td>431</td>
<td>436</td>
</tr>
<tr>
<td>10KBytes</td>
<td>76</td>
<td>130</td>
<td>311</td>
<td>312</td>
<td>315</td>
</tr>
<tr>
<td>100KBytes</td>
<td>33</td>
<td>52</td>
<td>57</td>
<td>57</td>
<td>57</td>
</tr>
</tbody>
</table>
Performance Results

Macro-Benchmark:

Compiled versus Interpretive Packet Filtering

![Graph showing performance results for BPF and Palladium](image)

- **BPF**
- **Palladium**
Conclusion

- Palladium provides safe, efficient and transparent user-level and kernel-level software extensions

- The key idea is to exploit both paging and segmentation hardware feature in X86 architecture

Future Work

Combine multi-threading with multiple protection domains

Exploit segmentation hardware to implement other kernel services such as narrow-interfaced protected memory

Debugger support for multi-segment programming

More applications development experiences in database and 3D graphics applications