ERIM: Secure, Efficient in-process Isolation with Memory Protection Keys

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, Deepak Garg
Applications in the **Absence of Isolation**

- All state accessible at **all times** to
  - Bugs
  - Security vulnerabilities
Applications in the **Absence of Isolation**

Heartbleed Bug

~70% of CVE assigned by Microsoft are memory safety issues.

Microsoft Security Response Center: “A proactive approach to more secure code”, 2019
Example In-Process Isolation Use Cases

Cryptographic Secrets

Managed runtimes from native libraries

Untrusted Application

Trusted Crypto Library

Native Library

Managed Runtime
User-space Threat Model

Attacker’s Capabilities include, but not limited to
- Control-flow hijacks
- Memory corruption (i.e., out-of-bounds accesses)

Out of scope:
- Side-channel, row hammer or microarchitectural attacks
## State of In-Application Isolation Techniques

<table>
<thead>
<tr>
<th>OS/VMM Technique</th>
<th>Execution overhead</th>
<th>Switch overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS/VMM-based</strong>&lt;sup&gt;2&lt;/sup&gt;</td>
<td>Low</td>
<td><strong>Medium</strong></td>
</tr>
<tr>
<td>Lang. &amp; RT&lt;sup&gt;3&lt;/sup&gt;</td>
<td>Medium – High</td>
<td>None</td>
</tr>
<tr>
<td>ERIM</td>
<td>Low</td>
<td>None</td>
</tr>
</tbody>
</table>

1. LwC, SMVs, Shreds, Wedge, Nexen, Dune, SeCage, TrustVisor
2. SFI
## State of In-Application Isolation Techniques

<table>
<thead>
<tr>
<th></th>
<th>Execution overhead</th>
<th>Switch overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Untrusted</td>
<td>Trusted</td>
</tr>
<tr>
<td><strong>OS/VMM-based(^2)</strong></td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Lang. &amp; RT(^3)</strong></td>
<td><strong>Medium – High</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>ERIM</strong></td>
<td>Low</td>
<td>None</td>
</tr>
</tbody>
</table>

### Language and Runtime Techniques

- Application
- Sensitive Data
- Operating System

---

1. LwC, SMVs, Shreds, Wedge, Nexen, Dune, SeCage, TrustVisor
2. SFI
# State of In-Application Isolation Techniques

<table>
<thead>
<tr>
<th></th>
<th>Execution overhead</th>
<th>Switch overhead</th>
<th>OS/VMM-based(^2)</th>
<th>Lang. &amp; RT(^3)</th>
<th>ERIM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Untrusted</td>
<td>Trusted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Medium – High</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

1. LwC, SMVs, Shreds, Wedge, Nexen, Dune, SeCage, TrustVisor
2. SFI, Native Client, Memsentry-MPX
Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
Intel Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
Intel Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
- Permission Register (PKRU)

### CPU Core

<table>
<thead>
<tr>
<th>PKRU Register</th>
<th>15</th>
<th>15</th>
<th>...</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
<td>R</td>
<td></td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Page Table Entry (PTE)

<table>
<thead>
<tr>
<th></th>
<th>PKEY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

### Address Space

- Page 1
- Page 2
- Page 3
Intel Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
- Permission Register (PKRU)
- Userspace instruction to update PKRU
  - Fast switch between 11 – 260 cycles/switch
Intel Memory Protection Keys (MPK)

- Available in Skylake server CPUs
- Tag memory pages with PKEY
- Permission Register (PKRU)
- Userspace instruction to update
  - Fast switch at 50 cycles/switch

By itself, MPK does not protect against malicious attacks.
Overview of ERIM

• Prevent MPK exploitation
  • Safe call gates
  • Prevent execution of permission register updates outside of call gates

Code:
48 83 c0 08 44 01 fa
83 fa 07 77 0f 01 ef
83 ff 07 0f 96 c2 80
Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates

```
Code:
48 83 c0 08 44 01 fa
83 fa 07 77 0f 01 ef
83 ff 07 0f 96 c2 80
```
Overview of ERIM

• Prevent MPK exploitation
  • Safe call gates
  • Prevent execution of permission register updates outside of call gates

• Creating usable binaries
  • Inadvertent PKRU update instruction
  • Rewrite strategy
Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates
- Creating usable binaries
  - Inadvertent PKRU update instruction
  - Rewrite strategy
- Evaluation
  - Frequently-switching use cases
  - 10% higher throughput compared to best existing technique
Updating the permission in PKRU register

- WRPKRU
  - Write EAX into PKRU

- XRSTOR
  - If **bit 9** of EAX is set
  - Load PKRU register from specified memory address
Safe switching using call gates

perm = TRUSTED
WRPKRU (perm)
goto trusted_entry(T)

perm = TRUSTED
Untrusted Application

perm = UNTRUSTED
WRPKRU (perm)
Safe switching using call gates

```c
perm = TRUSTED
WRPKRU (perm)
goto trusted_entry(T)

perm = UNTRUSTED
WRPKRU (perm)
if (perm != UNTRUSTED)
exit;
```
Prevent execution of WRPKRU/XRSTOR outside of call gates

Prevent execution of unvetted pages by

1) Monitoring system calls and removing the execute permission

2) ERIM’s fault handler scans memory pages and ensures:
   • WRPKRU is part of a call gate
   • XRSTOR is followed by
     if(eax | 0x100)
     exit();
Overview of ERIM

- Prevent MPK exploitation
  - Safe call gates
  - Prevent execution of permission register updates outside of call gates
- Creating usable binaries
  - Inadvertent PKRU update instruction
  - Rewrite strategy
- Evaluation
  - Frequently-switching use cases
  - 10% higher throughput compared to best existing technique

Untrusted Application

Code:

- 48 83 c0 08 44 01 fa
- 83 fa 07 77 0f 01 ef
- 83 ff 07 0f 96 c2 80

Trusted Compartment

PKEY 0

PKEY 1
Creating usable binaries

- ERIM halts executables with inadvertent WRPKRUs/XRSTORs

→ Eliminate inadvertent WRPKRU/XRSTOR by **binary rewriting** at compile time, runtime prior to enabling execute permission, or via **static** binary rewriting for pre-compiled binaries
Rewriting inadvertent WRPKRUs/XRSTORs

Devise rewrite rules for inadvertent WRPKRUs

**Inter-Instruction:**

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...0F</td>
<td>01EF...</td>
</tr>
</tbody>
</table>

\[
\downarrow
\]

<table>
<thead>
<tr>
<th>...0F</th>
<th>90</th>
<th>01EF...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rewriting inadvertent WRPKRUs/XRSTORs

Devise rewrite rules for inadvertent WRPKRUs

**Intra-instruction WRPKRU**

Simplified x86 instruction format:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod R/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Optional</td>
</tr>
</tbody>
</table>
Rewriting inadvertent WRPKRUs/XRSTORs

Devise rewrite rules for inadvertent WRPKRUs

Example rewrite rule:

\[
\text{add } \text{ecx}, [\text{ebx} + 0x01EF0000] \\
\rightarrow \text{push eax;}
\text{mov eax, ebx;}
\text{add ecx, [eax + 0x01EF0000];}
\text{pop eax;}
\]

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mod R/M</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0F</td>
<td>0x01EF0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mod R/M</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x07</td>
<td>0x01EF0000</td>
</tr>
</tbody>
</table>
Overview of ERIM

• Prevent MPK exploitation
  • Safe call gates
  • Prevent execution of permission register updates outside of call gates

• Creating usable binaries
  • Inadvertent PKRU update instruction
  • Rewrite strategy

• Evaluation
  • Frequently-switching use cases
  • 10% higher throughput compared to best existing technique
Prototype implementation

• ERIM userspace library
  • Call gates
  • Memory allocator for trusted component overloading malloc-like functions
  • Memory inspection (exclude unsafe WRPKRU/XRSTOR)

• Prevent execution on pages with unsafe WRPKRU/XRSTOR
  a) P-Trace and seccomp BPF userspace monitor
  b) Linux Security Module

• Remove inadvertent WRPKRU/XRSTORs
  • Static binary rewrite tool based on DynInst
Evaluation

How frequent are inadvertent WRPKRUs/XRSTORs?
- Inspected about 200,000 executable files of 5 Linux distributions
- Found 1213 inadvertent WRPKRU/XRSTOR in binary code
- DynInst disassembled 1,023
- 100% rewrite success

What is ERIM’s overhead in frequently-switching use cases?
- Isolating session keys in Nginx
- Isolating a managed runtime (node.js) from native libraries
- Isolating in-memory state of reference monitors (CPI/CPS)
Use case: Session Key Isolation

Address Space

NGINX

Connection Management
Content

OpenSSL &
LibCrypto

HTTPS session
Handshake protocol
Cryptographic keys
AES encrypt/decrypt
AES key initialization

AES Compartment
Nginx Throughput with protected session keys

ERIM throughput within 5% of native.

Normalized Throughput

File size in KB

0 1 2 4 8 16 32 64 128

Native

ERIM

0 0.2 0.4 0.6 0.8 1

31
Nginx Throughput with protected session keys

1.3 million switches per second

File size in KB

Native
ERIM
Comparison to Prior Art

Throughput:

- 95.4% ERIM
- 86.4% VMFUNC
- 73.2% MemSentry-MPX

Bar chart showing throughputs at different data sizes (0kb, 1kb, 2kb, 4kb, 8kb, 16kb, 32kb, 64kb, 128kb) for different technologies: Native, ERIM, VMFUNC, MemSentry-MPX, and Light-weight Context.
Summary

• Prevent MPK exploitation
  • Safe call gates
  • Prevent execution of permission register updates outside of call gates
• Creating usable binaries
  • Inadvertent PKRU update instruction
  • Rewrite strategy
• Evaluation
  • Frequently-switching use cases
  • 10% higher throughput compared to best existing technique
Thank you!

ERIM: Secure, Efficient in-process Isolation with Memory Protection Keys

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, Deepak Garg

Code available at https://gitlab.mpi-sws.org/vahldiek/erim
Backup
Intel Memory Protection Keys (MPK)

• Tag memory pages with a memory domains (bits 62:59 in page table)
• Permission register (PKRU) enables R/W to a domain
• Update accessible permissions from userspace
  • Fast switching, without context/PT switch
• By itself, protects against **bugs only**
State of the art: Isolating in-memory state

<table>
<thead>
<tr>
<th>ASLR-based Hiding</th>
<th>OS/VMM-Based</th>
<th>Execution overhead</th>
<th>Switch overhead</th>
<th>Threat model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Sensitive data</td>
<td>Untrusted</td>
<td>Trusted</td>
<td>None</td>
</tr>
<tr>
<td>Operating System</td>
<td>OS + VMM</td>
<td>Low</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Language and Runtime Techniques</td>
<td>ERIM: Memory Isolation using Intel MPK</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Application</td>
<td>Sensitive data</td>
<td>Lang. &amp; RT</td>
<td>Medium – High</td>
<td>None</td>
</tr>
<tr>
<td>Operating System</td>
<td>ERIM</td>
<td>Low</td>
<td>None</td>
<td>Low</td>
</tr>
</tbody>
</table>

1 ASLR-Guard, Near, XnR
2 LwC, SMVs, Shreds, Wedge, Nexen, Dune, SeCage, TrustVisor
3 MemSentry, SFI
Isolating sensitive state with Intel MPK

Address Space

Sensitive State

Untrusted Application State

Domain switch is a user-mode register write: efficient but vulnerable to attack.
Using ERIM to isolate memory

**Inlined switches**

```c
fct_A(...) {
    ....
    switch(Trusted)
    access sensitive data
    switch(Untrusted)
    ...
}
```

**Function overwriting**

```c
BUILD_BRIDGE(fct_A);
fct_B(...) {
    ...
    CALL_BRIDGE(fct_A, args);
    ...
}
```

**Function overloading via LD_PRELOAD**

```c
Shared library defines:
```
```c
fct_A(...) {
    ....
    }
```
```c
f = dlsym(fct_A, ...);
switch(Trusted);
ret = f(args);
switch(Untrusted);
return ret;
```
Comparison to MPX

![Comparison to MPX](image)

Normalized Throughput

File size

ERIM  MPX

0kb  1kb  2kb  4kb  8kb  16kb  32kb  64kb  128kb

[Normalized Throughput Comparison Chart]
Comparison to VMFUNC EPT switch

![Comparison to VMFUNC EPT switch](image)
Comparison to LwC
How frequent are inadvertent WRPKRUs/XRSTORs?

<table>
<thead>
<tr>
<th></th>
<th>Debian 8</th>
<th>Ubuntu 14</th>
<th>Ubuntu 16</th>
<th>Gentoo</th>
<th>Gentoo Gold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elf files</td>
<td>56035</td>
<td>58548</td>
<td>69907</td>
<td>9940</td>
<td>9940</td>
</tr>
<tr>
<td>Elf files with WRPKRU/XRSTOR</td>
<td>665</td>
<td>603</td>
<td>720</td>
<td>73</td>
<td>34</td>
</tr>
<tr>
<td>Executable WRPKRU/XRSTOR</td>
<td>4244</td>
<td>1147</td>
<td>2105</td>
<td>124</td>
<td>46</td>
</tr>
<tr>
<td>WPKRU/XRSTOR in code</td>
<td>481</td>
<td>276</td>
<td>384</td>
<td>41</td>
<td>31</td>
</tr>
<tr>
<td>Disassembled by Dyninst</td>
<td>420</td>
<td>215</td>
<td>332</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>Inter-instruction</td>
<td>30</td>
<td>29</td>
<td>44</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Intra-instruction</td>
<td>390</td>
<td>186</td>
<td>288</td>
<td>27</td>
<td>19</td>
</tr>
</tbody>
</table>
# How frequent are inadvertent WRPKRUs?

<table>
<thead>
<tr>
<th></th>
<th>Debian 8</th>
<th>Ubuntu 14</th>
<th>Ubuntu 16</th>
<th>Gentoo</th>
<th>Gentoo Gold</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elf files 56035</td>
<td>58548</td>
<td>69907</td>
<td>9940</td>
<td>9940</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>WRPKRU</td>
<td>XRSTOR</td>
<td>All</td>
<td>WRPKRU</td>
</tr>
<tr>
<td>Elf files w/ WRPKRU/XRSTOR</td>
<td>665</td>
<td>174</td>
<td>541</td>
<td>603</td>
<td>215</td>
</tr>
<tr>
<td>Executable WRPKRUXRSTOR</td>
<td>4244</td>
<td>288</td>
<td>3956</td>
<td>1147</td>
<td>442</td>
</tr>
<tr>
<td>WPKRU/XRSTOR in code</td>
<td>481</td>
<td>63</td>
<td>418</td>
<td>276</td>
<td>66</td>
</tr>
<tr>
<td>Disassembled by Dyninst</td>
<td>420</td>
<td>52</td>
<td>368</td>
<td>215</td>
<td>55</td>
</tr>
<tr>
<td>Inter-instruction</td>
<td>Number</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Rewritable by NOP</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>Intra-instruction</td>
<td>Number</td>
<td>390</td>
<td>22</td>
<td>368</td>
<td>186</td>
</tr>
<tr>
<td></td>
<td>Rewritable by rule 5</td>
<td>199</td>
<td>22</td>
<td>177</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>Rewritable by rule 4/6</td>
<td>191</td>
<td>0</td>
<td>194</td>
<td>5</td>
</tr>
</tbody>
</table>
ERIM Related Work

Hardware-based Isolation:
• Trusted Execution Engines (TEE) [SGX, TrustZone]
• Reducing TCB of TEE [Flicker]
• Sandbox applications in TEE [Haven, Scone]

Hypervisor/OS-based:
• Reference monitors [Dune, Wedge, LwC]
• Sandboxing Applications [Capsicum]
• Privilege Separation [PrivTrans]
• Hiding secrets in execute-only code [Redactor, Near]
ERIM Related Work

**Software-fault isolation:**
- Compilation-based [NativeClient]
- Emulation [Vx32]
- Just-in-time compiled languages [NativeClient++]

**Inlined Reference Monitoring:**
- Control-Flow Integrity [CPI]
- Sandboxing annotated code [Shreds]
- Intercepting Android framework [Aurasium]
Call Gates

WRPKRU (RW_TRUSTED)

// entry point to trusted

WRPKRU (DIS_TRUSTED)
cmp DIS_TRUSTED, EAX
je continue
exit
continue:

Elevate privileges and transfer to trusted entry point

Remove privileges, check for reduced privileges and return from trusted component
Creating safe binaries

Devise rewrite rules for WRPKRU in code segment

**Inter-instruction WRPKRU (0x0F01EF)**

Example rewrite rule:

```
    ...OF          01EF...
    ▼              ▼
    ...OF          Nop     01EF...
```
Creating safe binaries

Intra-instruction WRPKRU

Simplified x86 instruction format:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod R/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
</table>

Example rewrite rule:

add ecx, [ebx + 0x01EF0000]

\[
\begin{array}{ccc}
\text{Opcode} & \text{Mod R/M} & \text{Displacement} \\
0x01 & 0x0F & 0x01EF0000
\end{array}
\]

→ push eax; mov eax, ebx; add ecx, [eax + 0x01EF0000]; pop eax;

\[
\begin{array}{ccc}
\text{Opcode} & \text{Mod R/M} & \text{Displacement} \\
0x01 & 0x07 & 0x01EF0000
\end{array}
\]
## Creating safe binaries: Rewrite Rules

<table>
<thead>
<tr>
<th>Overlap with</th>
<th>Cases</th>
<th>Rewrite strategy</th>
<th>ID</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Opcode = WRPKRU</td>
<td>Insert privilege check after WRPKRU</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Mod R/M</td>
<td>Mod R/M = 0x0F</td>
<td>Change to unused register + move command</td>
<td>2</td>
<td>add ecx, [ebx + 0x01EF0000] → mov eax, ebx; add ecx, [eax + 0x01EF0000];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Push/Pop used register + move command</td>
<td>3</td>
<td>add ecx, [ebx + 0x01EF0000] → push eax; mov eax, ebx; add ecx, [eax + 0x01EF0000]; pop eax;</td>
</tr>
<tr>
<td>Displacement</td>
<td>Full/Partial sequence</td>
<td>Change mode to use register</td>
<td>4</td>
<td>add eax, 0x0F01EF00 → (push ebx;) mov ebx, 0x0F010000; add ebx, 0x0000EA00; add eax, ebx; (pop ebx;)</td>
</tr>
<tr>
<td></td>
<td>Jump-like instruction</td>
<td>Move code segment to alter constant used in address</td>
<td>5</td>
<td>call [rip + 0xffef010f] → call [rip + 0xffef0100]</td>
</tr>
<tr>
<td>Immediate</td>
<td>Full/Partial sequence</td>
<td>Change mode to use register</td>
<td>6</td>
<td>add eax, 0x0F01EF → (push ebx;) mov ebx, 0x0F01EE00; add ebx, 0x00000100; add eax, ebx; (pop ebx;)</td>
</tr>
<tr>
<td></td>
<td>Associative opcode</td>
<td>Apply instruction twice with different immediates to get equivalent effect</td>
<td>7</td>
<td>add ebx, 0x0F01EF00 → add ebx, 0xE01EF00; add ebx, 0x01000000</td>
</tr>
</tbody>
</table>
WRPKRU Occurrences

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Debian 8</th>
<th>Ubuntu 14</th>
<th>Ubuntu 16</th>
<th>Hardened Gentoo</th>
<th>Hardened Gentoo Gold</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF files</td>
<td>61364</td>
<td>69829</td>
<td>79169</td>
<td>10212</td>
<td>10212</td>
</tr>
<tr>
<td>ELF files with WRPKRU</td>
<td>182 (.30%)</td>
<td>223 (.32%)</td>
<td>219 (.28%)</td>
<td>9 (.09%)</td>
<td>0 (.0%)</td>
</tr>
<tr>
<td>Executable WRPKRUUs</td>
<td>301</td>
<td>454</td>
<td>273</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>WRPKRU in code section</td>
<td>69 (22.9%)</td>
<td>72 (15.9%)</td>
<td>101 (37.0%)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inter-instruction WRPKRU</td>
<td>35 (50.7%)</td>
<td>42 (58.3%)</td>
<td>43 (42.6%)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Intra-instruction WRPKRU</td>
<td>34 (49.3%)</td>
<td>30 (41.6%)</td>
<td>58 (57.4%)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rewritable by Dyninst</td>
<td>58 (84%)</td>
<td>59 (81.9%)</td>
<td>91 (90%)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
# Nginx Throughput with protected session keys

<table>
<thead>
<tr>
<th>File size</th>
<th>Native (req./s)</th>
<th>ERIM rel. (%)</th>
<th>Switches/s</th>
<th>CPU load</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>95,761</td>
<td>95.83</td>
<td>1,342,605</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>87,022</td>
<td>95.18</td>
<td>1,220,266</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>82,137</td>
<td>95.44</td>
<td>1,151,877</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>76,562</td>
<td>95.25</td>
<td>1,073,843</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>67,855</td>
<td>95.98</td>
<td>974,780</td>
<td>100</td>
</tr>
<tr>
<td>16</td>
<td>45,483</td>
<td>97.10</td>
<td>812,173</td>
<td>100</td>
</tr>
<tr>
<td>32</td>
<td>32,381</td>
<td>97.31</td>
<td>779,141</td>
<td>100</td>
</tr>
<tr>
<td>64</td>
<td>17,827</td>
<td>100.00</td>
<td>679,371</td>
<td>96.7</td>
</tr>
<tr>
<td>128</td>
<td>8,937</td>
<td>99.99</td>
<td>556,152</td>
<td>86.4</td>
</tr>
</tbody>
</table>
typedef struct secret {
    int number;
} secret;

secret * initSecret() {
    ERIM_SWITCH_T;
    secret * s = malloc(sizeof(secret));
    ERIM_SWITCH_U;
    s->number = random();
    return s;
}

int compute(secret* s, int m) {
    int ret = 0;
    ERIM_SWITCH_T;
    ret = f(s->number, m);
    ERIM_SWITCH_U;
    return ret;
}
SPEC 2006 with CPS/CPI
## NGINX multiple worker

<table>
<thead>
<tr>
<th>File size (KB)</th>
<th>1 worker</th>
<th>3 workers</th>
<th>5 workers</th>
<th>10 workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Native (req/s)</td>
<td>ERIM rel. (%)</td>
<td>Native (req/s)</td>
<td>ERIM rel. (%)</td>
</tr>
<tr>
<td>0</td>
<td>95.761</td>
<td>95.83</td>
<td>276,736</td>
<td>96.05</td>
</tr>
<tr>
<td>1</td>
<td>87,022</td>
<td>95.18</td>
<td>250,565</td>
<td>94.50</td>
</tr>
<tr>
<td>2</td>
<td>82,137</td>
<td>95.44</td>
<td>235,820</td>
<td>95.12</td>
</tr>
<tr>
<td>4</td>
<td>76,562</td>
<td>95.25</td>
<td>217,602</td>
<td>94.91</td>
</tr>
<tr>
<td>8</td>
<td>67,855</td>
<td>95.98</td>
<td>142,680</td>
<td>100.00</td>
</tr>
</tbody>
</table>

**Table 4.7**: NGinx throughput with multiple workers. The standard deviation is below 1.5% in all cases.