Proactive Defense Against CPU Side Channel Attacks

Kristen Carlson Accardi
No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others

© Intel Corporation.
“a monoculture is a community of computers that all run identical software. All the computer systems in the community thus have the same vulnerabilities, and, like agricultural monocultures, are subject to catastrophic failure in the event of a successful attack.”

https://en.wikipedia.org/wiki/Monoculture_(computer_science)
What does this have to do with side channels?
Software Diversification makes some side channels less useful
Gadge Me If You Can
Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM

Lucas Davi,1 Alexandra Dmitrienko,1 Stefan Nürnberger, Ahmad-Reza Sadeghi

Compiler-assisted Code Randomization

Hyungjoon Koo,* Yaohui Chen,† Long Lu,† Vasileios P. Kemerlis,‡ Michalis Polychronakis*

*Stony Brook University {hykoo, mikepo}@cs.stonybrook.edu †Northeastern University {yaowway, long}@ccs.neu.edu ‡Brown University vpk@cs.brown.edu

Abstract: For research on software diversity, layout randomization has seen limited success, but effective defense under certain circumstances by remotely leaking [18–20] or inferring [21, 22] what code exists at a given memory location. As a response, recent protections against JIT-TROOP exploits rely

Operando System Protection Through Program Evolution
by Dr. Frederick B. Cohen ‡

1. Introduction
The deployment of standard address space layout randomization (ASLR) executables memory [71].
Simplified Elf Executable format

- Kernel is much more complicated
- .text is your executable code
Levels of Randomization

- Instruction level
  - Equivalent instruction substitution
  - Instruction reordering
  - Register allocation reordering
  - Garbage code insertion (nops etc)
- Basic block
  - Reordering
- Function level
  - Stack layout randomization
    - -fstack-shuffle (OpenBSD)
  - Function parameter randomization
  - Inlining, outlining or splitting
  - Jump table insertion
- Program level
  - Function reordering
  - Base Address Randomization
  - Program encoding
  - Data randomization
    - Static data layout
    - Constant blinding
    - Structure layout randomization
    - Heap/stack layout randomization
  - System call mapping randomization
When to Randomize

Vendor System

Implement  Build  Distribute

User System

Install  Load  Execute  Update
Levels of Randomization

- **Instruction level**
  - Equivalent instruction substitution
  - Instruction reordering
  - Register allocation reordering
  - Garbage code insertion (nops etc)
- **Basic block**
  - Reordering
- **Function level**
  - Stack layout randomization
    - -fstack-shuffle (OpenBSD)
  - Function parameter randomization
  - Inlining, outlining or splitting
  - Jump table insertion
- **Program level**
  - Function reordering
  - Base address randomization
    - KASLR
  - Program encoding
  - Data randomization
    - Static data layout
    - Constant blinding
    - Structure layout randomization
      - Gcc plugin currently implemented in kernel
      - Heap/stack layout randomization
  - System call mapping randomization
When to Randomize

- Implement
- Build
- Distribute

Vendor System

---

- Install
- Load
- Execute
- Update

User System

---

ASLR

@kcaccardi
KASLR (Kernel Address Space Layout Randomization)

- Kernel Text section is located within a fixed range
- Exact location is determined at boot
- Order within .text is unchanged

- 0x0
- 0xffffffff80000000
- 0xffffffffc0000000
- 0xfffffffffffffffc0000000
- 0xfffffffffffffff
ASLR is pretty weak

- Kernel ASLR has low entropy
  - Brute Force attacks are possible

- Infoleaks reveal location of entire .text segment
  - Relative distances remain the same
If ASLR is so weak, why don’t we do more?
Monoculture has its benefits

- Ease of Distribution
  - Creating randomized binaries at download time is slow and expensive
  - Would only work for people who create custom kernels
- Code signing
- Load/Run time overhead of diversified binaries
- Debugging & error reporting
- Tracing and live-patching
Can we do better?
When to Randomize

Vendor System

Implement - Build - Distribute

User System

Install - Load - Execute - Update

@kcaccardi
Levels of Randomization

- **Instruction level**
  - Equivalent instruction substitution
  - Instruction reordering
  - Register allocation reordering
  - Garbage code insertion (nops etc)

- **Basic block**
  - Reordering

- **Function level**
  - Stack layout randomization
    - `-fstack-shuffle` (OpenBSD)
  - Function parameter randomization
  - Inlining, outlining or splitting
  - Jump table insertion

- **Program level**
  - Function reordering
  - Base address randomization
    - KASLR
  - Program encoding
  - Data randomization
    - Static data layout
    - Constant blinding
    - Structure layout randomization
      - Gcc plugin currently implemented in kernel
      - Heap/stack layout randomization
  - System call mapping randomization
Implementing kernel function reordering

- Keep it small and simple by starting with modules
- Modules are already linked into the kernel at module load time
- Modules are already relocatable objects
- Modules can have individual Makefiles

```bash
obj-$(CONFIG_TEST_MODULE) +=
test_module.o
ccflags-y := -ffunction-sections
```
<table>
<thead>
<tr>
<th>Relocatable format</th>
<th>-ffunction-sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elf Header</td>
<td>Elf Header</td>
</tr>
<tr>
<td>Segment Table</td>
<td>Segment Table</td>
</tr>
<tr>
<td>.text</td>
<td>.text</td>
</tr>
<tr>
<td>.rela.text</td>
<td>.rela.function1</td>
</tr>
<tr>
<td>.data</td>
<td>.text.function2</td>
</tr>
<tr>
<td>.rela.data</td>
<td>.rela.text.function1</td>
</tr>
<tr>
<td>...</td>
<td>.text.function2</td>
</tr>
<tr>
<td>Symbol table</td>
<td>.rela.text.function2</td>
</tr>
<tr>
<td>String table</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Symbol table</td>
</tr>
<tr>
<td></td>
<td>String table</td>
</tr>
</tbody>
</table>
Simple module example

```c
static void __attribute__((optimize("O0"))) test_module_do_work(void)
{
    ...
}

static void test_module_wq_func(struct work_struct *w)
{
    test_module_do_work();
    queue_work(test_module_wq, w);
    return;
}

static int __init test_module_init(void)
{
    ...
}
```
Implementing kernel function reordering

[kcaccard@kcaccard-mobl3 test_module]$ readelf --sections --wide test_module.ko

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Off</th>
<th>Size</th>
<th>ES Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>PROGBITS</td>
<td>0000000000000000 000064 000000 00 AX 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.text.test_module_do_work</td>
<td>PROGBITS</td>
<td>0000000000000000 000064 000033 00 AX 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.text.test_module_wq_func</td>
<td>PROGBITS</td>
<td>0000000000000000 000097 000023 00 AX 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing kernel function reordering

[kcaccard@kcaccard-mobl3 test_module]$ readelf --sections --wide test_module.ko

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>.text</td>
<td>PROGBITS</td>
<td>0000000000000000 000064 000000 00 AX 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.text.test_module_do_work</td>
<td>PROGBITS</td>
<td>0000000000000000 000064 000033 00 AX 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.text.test_module_wq_func</td>
<td>PROGBITS</td>
<td>0000000000000000 000097 000023 00 AX 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[kcaccard@kcaccard-mobl3 test_module]$ readelf --relocs --wide test_module.ko

Relocation section '.rela.text.test_module_wq_func' at offset 0x1d830 contains 4 entries:
Offset Info Type Symbol's Value Symbol's Name + Addend
0000000000000001 0000032000000002 R_X86_64_PC32 0000000000000000 __fentry__ - 4
000000000000001a 0000003000000002 R_X86_64_PC32 0000000000000000 queue_work_on - 4
Implementing kernel function reordering

[kcaccard@kcaccard-mobl3 test_module]$ readelf --relocs --wide test_module.ko
Relocation section '.rela.text.test_module_wq_func' at offset 0x1d830 contains 4 entries:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Info</th>
<th>Type</th>
<th>Symbol's Value</th>
<th>Symbol's Name + Addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000000a</td>
<td>0000000300000002</td>
<td>R_X86_64_PC32</td>
<td>0000000000000000</td>
<td>.text.test_module_do_work - 4</td>
</tr>
</tbody>
</table>

[kcaccard@kcaccard-mobl3 test_module]$ objdump -d test_module.ko
Disassembly of section .text.test_module_wq_func:

0000000000000000 <test_module_wq_func>:
  0:   e8 00 00 00 00           callq 5 <test_module_wq_func+0x5> 
  5:   53                     push %rbx 
  6:   48 89 fb               mov %rdi,%rbx 
  9:   e8 00 00 00 00          callq e <test_module_wq_func+0xe> 
  e:   48 89 da               mov %rbx,%rdx 
 11:   48 8b 35 00 00 00 00   mov 0x0(%rip),%rsi          # 18
 18:   bf 40 00 00 00           mov $0x40,%edi
 1d:   5b                     pop %rbx 
 1e:   e9 00 00 00 00           jmpq 23 <work+0x3>
Implementing kernel function reordering

[kcaccard@kcaccard-mobl3 test_module]$ readelf --symbols --wide test_module.ko

Symbol table `.symtab' contains 58 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>00000000000000000000000000</td>
<td>51</td>
<td>FUNC</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>3</td>
<td>test_module_do_work</td>
</tr>
<tr>
<td>36</td>
<td>00000000000000000000000000</td>
<td>35</td>
<td>FUNC</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>5</td>
<td>test_module_wq_func</td>
</tr>
<tr>
<td>37</td>
<td>00000000000000000000000000</td>
<td>214</td>
<td>FUNC</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>7</td>
<td>test_module_init</td>
</tr>
<tr>
<td>38</td>
<td>00000000000000000000000000</td>
<td>36</td>
<td>FUNC</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>9</td>
<td>test_module_exit</td>
</tr>
<tr>
<td>49</td>
<td>00000000000000000000000000</td>
<td>36</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>9</td>
<td>cleanup_module</td>
</tr>
<tr>
<td>51</td>
<td>00000000000000000000000000</td>
<td>214</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>7</td>
<td>init_module</td>
</tr>
</tbody>
</table>
Module memory layout

Core Section

- Executable (.text, .exit, etc)
- Read only (text and data)
- Writable data
- Read only after init
- Everything else

Init Section

- Executable (.init)
- etc
Module memory layout

Core Section
- Executable (.text, .exit, etc)
- Read only (text and data)
- Writable data
- Read only after init
- Everything else

Init Section
- Executable (.init)
- etc

Executable section randomized
- .text.function5
- .text.function42
- ...
- .text.function27
Effectiveness

- Depends on the number of functions
  - Inherently not as strong as even finer grained randomization
  - Can expand over time

- Randomizing modules isn’t the end goal
  - Will need some modifications to implementation for kernel .text
  - Implementing for modules lets us take incremental steps
Monoculture benefits revisited

- Ease of Distribution ✔
- Code signing ✔
- Load/Run time overhead of diversified binaries ?
- Debugging & error reporting ?
- Tracing and live-patching ✔ ?
We can do better still
Execute Only Memory

- x86 page tables have bits for Present, Write, and No-eXecute (NX).
- Present must be set for all Write and NX entries. It is not possible to represent a Writeable, but unreadable or executable but unreadable page table entry.
- Extended Page Table (EPT, part of VMX) format contains separate Read, Write and eXecute bits. It can represent Present+eXecutable+non-Readable memory.
  - Function pointers in data would still be readable

- Implementing this requires work both in the guest OS and the VMM.
- Challenges are still unknown
  - Data in text area may still exist (jump tables)
  - May need to be turned on and off for some reason (kprobes)
Just a few resources

- https://www.computer.org/csdl/proceedings/sp/2014/4686/00/4686a276.pdf

POC available at:
https://github.com/kaccardi/linux.git (reorder-module-functions branch)