kR^X

Comprehensive Kernel Protection against Just-In-Time Code Reuse

Marios Pomonis [1]

Theofilos Petsios [1] Angelos D. Keromytis [1]

Michalis Polychronakis [2] Vasileios P. Kemerlis [3]

[1] Columbia University [2] Stony Brook University [3] Brown University

$> whoami

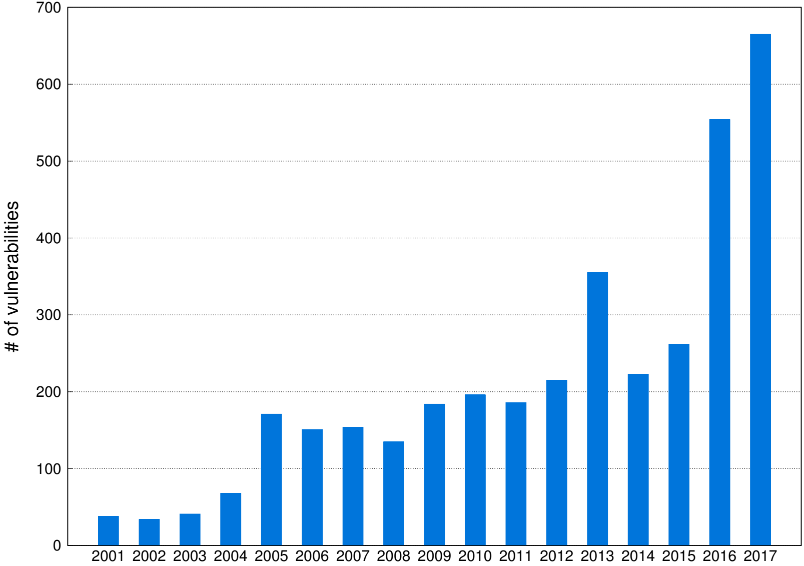
- Ph.D. candidate @Columbia University
- Member of the Network Security Lab
  - `http://nsl.cs.columbia.edu`
- Research interests
  - Kernel security
  - Data-flow tracking
  - `http://www.cs.columbia.edu/~mpomonis`

$> whoami

- Ph.D. candidate @Columbia University
- Member of the Network Security Lab
  - `http://nsl.cs.columbia.edu`
- Research interests
  - **Kernel security**
  - Data-flow tracking
  - `http://www.cs.columbia.edu/~mpomonis`

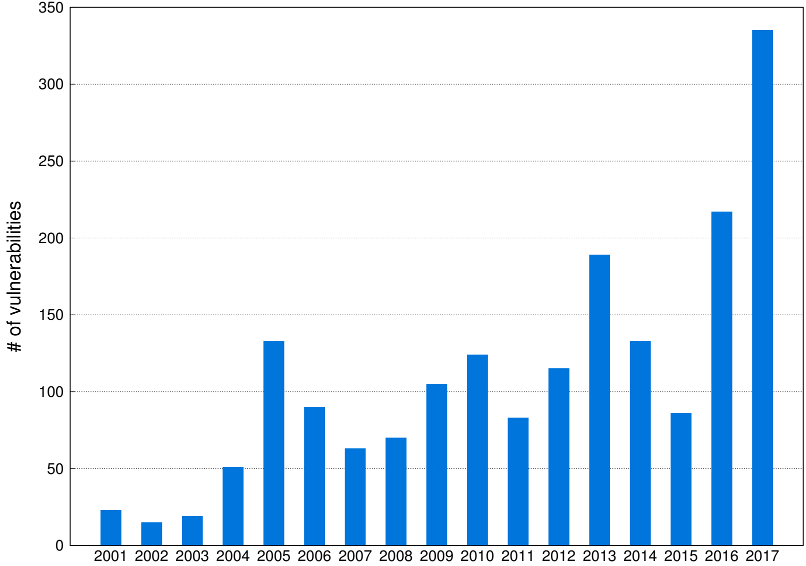
COLUMBIA UNIVERSITY

Kernel Vulnerabilties (all vendors)



Source: National Vulnerability Database (`http://nvd.nist.gov`)

Linux Kernel Vulnerabilties



Source: CVE Details (`http://www.cvedetails.com`)

Kernel Exploitation 101

- Userland Exploitation
  - Code Injection
  - Code Reuse

Kernel Exploitation 101

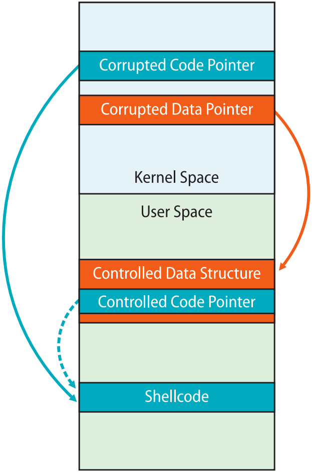
- Userland Exploitation
  - ~~Code Injection~~ [W^X]
  - Code Reuse [ASLR]

Kernel Exploitation 101

- Userland Exploitation
  - ~~Code Injection~~ [W^X]
  - Code Reuse [ASLR]
- Kernel Exploitation

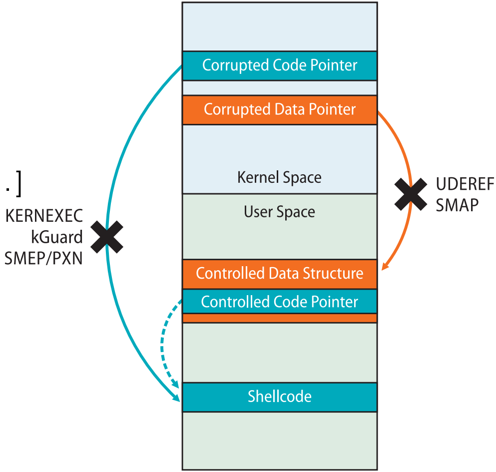
Kernel Exploitation 101

- Userland Exploitation
  - ~~Code Injection~~ [W^X]
  - Code Reuse [ASLR]
- Kernel Exploitation
  - ret2usr



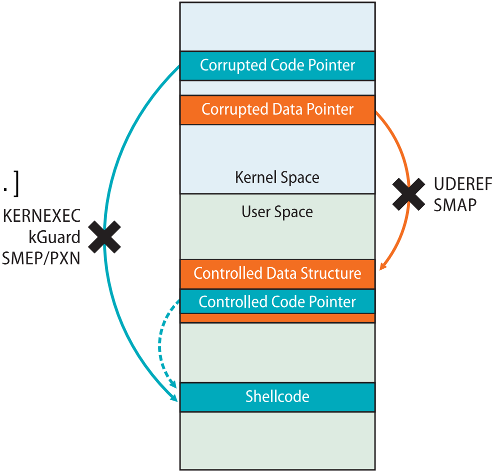
Kernel Exploitation 101

- Userland Exploitation
  - ~~Code Injection~~ [W^X]
  - Code Reuse [ASLR]
- Kernel Exploitation
  - ~~ret2usr~~ [SMEP, SMAP, ...]



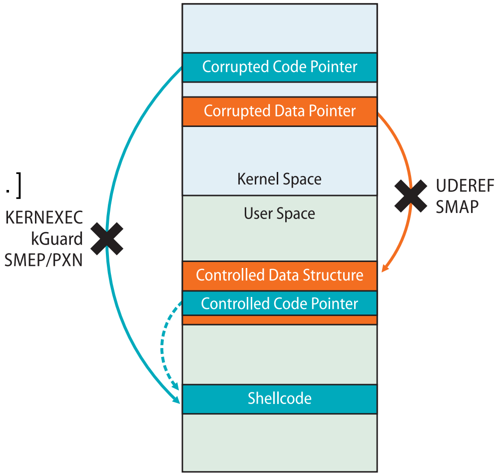
Kernel Exploitation 101

- Userland Exploitation
  - ~~Code Injection~~ [W^X]
  - Code Reuse [ASLR]
- Kernel Exploitation
  - ~~ret2usr~~ [SMEP, SMAP, ...]
  - Code Injection
  - Code Reuse



Kernel Exploitation 101

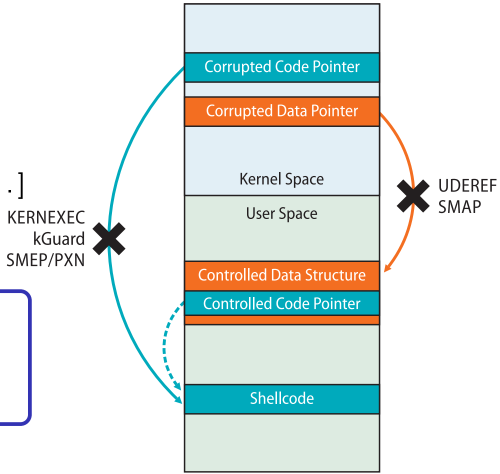
- Userland Exploitation
  - ~~Code Injection~~ [W^X]
  - Code Reuse [ASLR]
- Kernel Exploitation
  - ~~ret2usr~~ [SMEP, SMAP, ...]
  - ~~Code Injection~~ [W^X]
  - Code Reuse [KASLR]



Kernel Exploitation 101

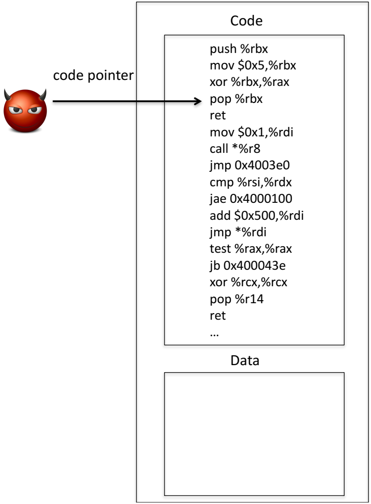
- Userland Exploitation
  - ~~Code Injection~~ [W^X]
  - Code Reuse [ASLR]
- Kernel Exploitation
  - ~~ret2usr~~ [SMEP, SMAP, ...]
  - ~~Code Injection~~ [W^X]
  - Code Reuse [KASLR]

Hund et al.	[Oakland '13]
Jang et al.	[CCS '16]
Gruss et al.	[CCS '16]



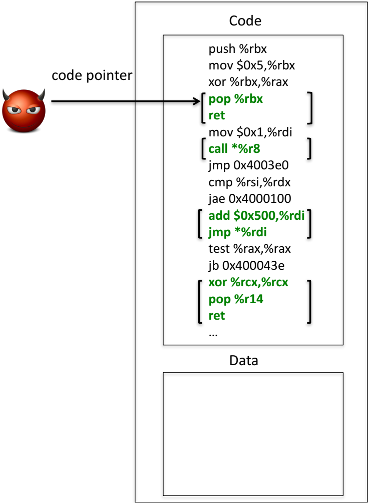
Code Reuse Attacks

- "Offline" Code Reuse



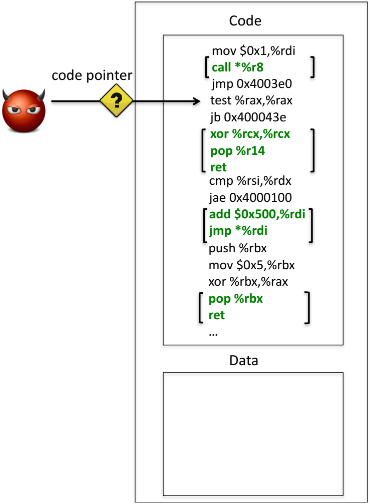
Code Reuse Attacks

- "Offline" Code Reuse
  - Code snippets (**gadgets**)
    - Ending with an indirect branch
  - Stitch gadgets together
    - Perform arbitrary computations



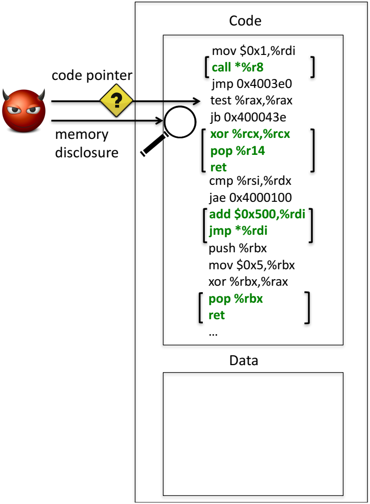
Code Reuse Attacks

- ~~"Offline" Code Reuse~~ [Code Diversification]
  - Code snippets (**gadgets**)
    - Ending with an indirect branch
  - Stitch gadgets together
    - Perform arbitrary computations



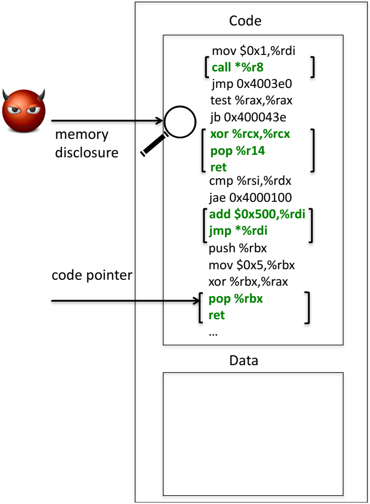
Code Reuse Attacks

- ~~"Offline" Code Reuse~~ [Code Diversification]
  - Code snippets (**gadgets**)
    - Ending with an indirect branch
  - Stitch gadgets together
    - Perform arbitrary computations
- "Just-In-Time" Code Reuse
  - Direct
    - Read the (diversified) code
    - Construct the exploit on-the-fly



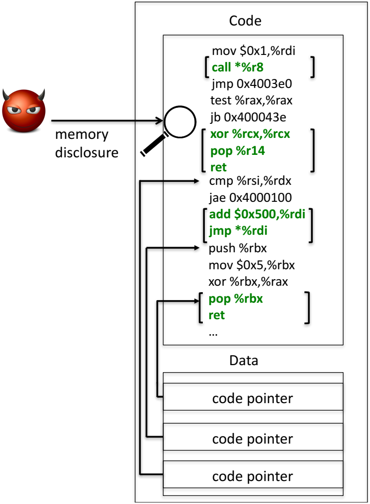
Code Reuse Attacks

- ~~"Offline" Code Reuse~~ [Code Diversification]
  - Code snippets (**gadgets**)
    - Ending with an indirect branch
  - Stitch gadgets together
    - Perform arbitrary computations
- "Just-In-Time" Code Reuse
  - Direct
    - Read the (diversified) code
    - Construct the exploit on-the-fly



Code Reuse Attacks

- ~~"Offline" Code Reuse~~ [Code Diversification]
  - Code snippets (**gadgets**)
    - Ending with an indirect branch
  - Stitch gadgets together
    - Perform arbitrary computations
- "Just-In-Time" Code Reuse
  - Direct
    - Read the (diversified) code
    - Construct the exploit on-the-fly
  - Indirect
    - Read code pointers from the data
    - *Infer* the randomized code layout



kR^X

- **Comprehensive** kernel protection against code reuse attacks
  - ✗ "Offline" Code Reuse ✗JIT Code Reuse (direct/indirect)
  - No privileged entity (e.g., hypervisor)
  - Low overhead

kR^X

- **Comprehensive** kernel protection against code reuse attacks
  - ✗ "Offline" Code Reuse ✗JIT Code Reuse (direct/indirect)
  - No privileged entity (e.g., hypervisor)
  - Low overhead

R^X:

- Execute-only Memory
  - Separate code and data regions
    - New kernel memory layout
  - Mem. read → **range check (RC)**
    - SFI-inspired
    - ✓ Data region ✗ Code region

kR^X

- **Comprehensive** kernel protection against code reuse attacks
  - ✗ "Offline" Code Reuse ✗JIT Code Reuse (direct/indirect)
  - No privileged entity (e.g., hypervisor)
  - Low overhead

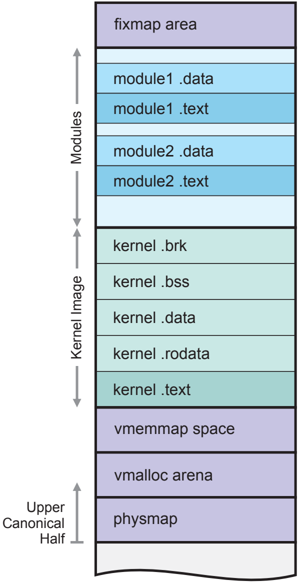
R^X:

- Execute-only Memory
  - Separate code and data regions
    - New kernel memory layout
  - Mem. read → **range check (RC)**
    - SFI-inspired
    - ✓ Data region ✗ Code region

Fine-grained KASLR:

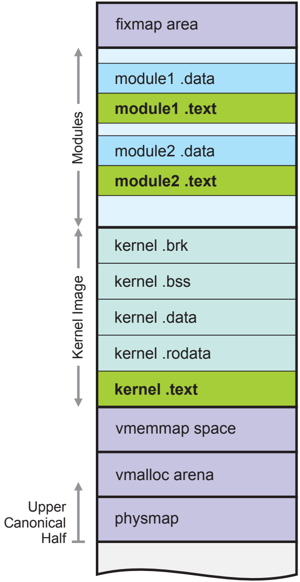
- Randomized Code Layout
  - ✓ No gadgets at known location
  - ✓ High entropy → no guessing
- Return address protection
  - Encryption (`XOR`-based)
  - Deception (Decoys)

R^X: Memory Layout



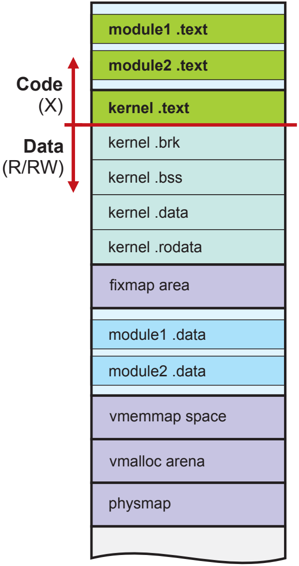
R^X: Memory Layout

✗ Multiple code sections → multiple RCs
  - High overhead
- Interleaved code and data



R^X: Memory Layout

- Disjoint code and data regions
  - Kernel image → linker
  - Modules → module loader
- ✓ Single range check
- No code region synonyms in physmap
- No other region affected
  - ✓ `kmalloc(), vmalloc()...`



R^X: Range Checks

```
      cmpl   $0x7,0x154(%rsi)
      mov    0x140(%rsi),%rcx
      jg     L1
      mov    0x130(%rsi),%rax
      or     $0x400000,%rax
      mov    %rax,%rdx
      shr    $0x20,%rdx
      jmp    L2
L1:
      xor    %edx,%edx
      mov    $0x1,%eax
L2:
      wrmsr
      retq
```

nhm_uncore_msr_enable_event()

x86-64 Linux kernel (v3.19, GCC v4.7.2)

R^X: Range Checks (-O0)

- For every memory read
  - Spill/Fill the `%rflags` register
  - Effective address → reserved register (`%r11`)
  - Compare with the end of the data region (`_krx_edata`)

✓ Data read

✗ Code read

- Violation handler (`krx_handler`)

```
RC1  pushfq
     lea     0x154(%rsi),%r11
     cmp     $_krx_edata,%r11
     ja      L3
     popfq
     cmpl    $0x7,0x154(%rsi)
RC2  pushfq
     lea     0x140(%rsi),%r11
     cmp     $_krx_edata,%r11
     ja      L3
     popfq
     mov     0x140(%rsi),%rcx
     jg      L1
RC3  pushfq
     lea     0x130(%rsi),%r11
     cmp     $_krx_edata,%r11
     ja      L3
     popfq
     mov     0x130(%rsi),%rax
     or      $0x400000,%rax
     mov     %rax,%rdx
     shr     $0x20,%rdx
     jmp     L2
L1:
     xor     %edx,%edx
     mov     $0x1,%eax
L2:
     wrmsr
     retq
L3:
     callq   krx_handler
```

Micro-benchmarks (LMBench)

	Benchmark	SFI(-O0)
Latency	syscall()	126.90%
	open()/close()	306.24%
	read()/write()	215.04%
	select(10 fds)	119.33%
	select(100 TCP fds)	**1037.33%**
	fstat()	489.79%
	mmap()/munmap()	180.88%
	fork()+exit()	208.86%
	fork()+execve()	191.83%
	fork()+/bin/sh	113.77%
	sigaction()	63.49%
	Signal delivery	123.29%
	Protection fault	**13.40%**
	Page fault	202.84%
	Pipe I/O	126.26%
	UNIX socket I/O	148.11%
	TCP socket I/O	171.93%
	UDP socket I/O	208.75%
	Average	**224.89%**
Bandwidth	Pipe I/O	46.70%
	UNIX socket I/O	35.77%
	TCP socket I/O	**53.96%**
	mmap() I/O	**~0%**
	File I/O	23.57%
	Average	**32%**

R^X: Range Checks (-O1)

- For every memory read
  - Spill/Fill the `%rflags` register
  - Effective address → reserved register (`%r11`)
  - Compare with the end of the data region (`_krx_edata`)

✓ Data read

✗ Code read
  - Violation handler (`krx_handler`)

```
RC1  pushfq
     lea    0x154(%rsi),%r11
     cmp    $_krx_edata,%r11
     ja     L3
     popfq
     cmpl   $0x7,0x154(%rsi)
RC2  pushfq
     lea    0x140(%rsi),%r11
     cmp    $_krx_edata,%r11
     ja     L3
     popfq
     mov    0x140(%rsi),%rcx
     jg     L1
RC3  pushfq
     lea    0x130(%rsi),%r11
     cmp    $_krx_edata,%r11
     ja     L3
     popfq
     mov    0x130(%rsi),%rax
     or     $0x400000,%rax
     mov    %rax,%rdx
     shr    $0x20,%rdx
     jmp    L2
L1:
     xor    %edx,%edx
     mov    $0x1,%eax
L2:
     wrmsr
     retq
L3:
     callq  krx_handler
```

R^X: Range Checks (-O1)

- For every memory read
  - ~~Spill/Fill the %rflags register~~
  - pushfq/popfq Elimination [∼94%]
  - Effective address → reserved register (%r11)
  - Compare with the end of the data region (_krx_edata)

✓ Data read

✗ Code read

- Violation handler (krx_handler)

```
RC1  pushfq
     lea    0x154(%rsi),%r11
     cmp    $_krx_edata,%r11
     ja     L3
     popfq
     cmpl   $0x7,0x154(%rsi)
RC2  pushfq
     lea    0x140(%rsi),%r11
     cmp    $_krx_edata,%r11
     ja     L3
     popfq
     mov    0x140(%rsi),%rcx
     jg     L1
RC3  pushfq
     lea    0x130(%rsi),%r11
     cmp    $_krx_edata,%r11
     ja     L3
     popfq
     mov    0x130(%rsi),%rax
     or     $0x400000,%rax
     mov    %rax,%rdx
     shr    $0x20,%rdx
     jmp    L2
L1:
     xor    %edx,%edx
     mov    $0x1,%eax
L2:
     wrmsr
     retq
L3:
     callq  krx_handler
```

Micro-benchmarks (LMBench)

	Benchmark	SFI(-O0)	SFI(-O1)
Latency	`syscall()`	126.90%	13.41%
	`open()/close()`	306.24%	39.01%
	`read()/write()`	215.04%	22.05%
	`select(10 fds)`	119.33%	10.24%
	`select(100 TCP fds)`	**1037.33%**	**59.03%**
	`fstat()`	489.79%	15.31%
	`mmap()/munmap()`	180.88%	7.24%
	`fork()+exit()`	208.86%	14.32%
	`fork()+execve()`	191.83%	10.30%
	`fork()+/bin/sh`	113.77%	11.62%
	`sigaction()`	63.49%	0.19%
	Signal delivery	123.29%	18.05%
	Protection fault	**13.40%**	1.26%
	Page fault	202.84%	**~0%**
	Pipe I/O	126.26%	22.91%
	UNIX socket I/O	148.11%	12.39%
	TCP socket I/O	171.93%	25.15%
	UDP socket I/O	208.75%	25.71%
	Average	**224.89%**	**17.12%**
Bandwidth	Pipe I/O	46.70%	0.96%
	UNIX socket I/O	35.77%	3.54%
	TCP socket I/O	**53.96%**	**10.90%**
	`mmap()` I/O	**~0%**	**~0%**
	File I/O	23.57%	**~0%**
	Average	**32%**	**3.08%**

R^X: Range Checks (-O2)

- For every memory read
  - ~~Spill/Fill the `%rflags` register~~
  - `pushfq`/`popfq` Elimination [∼94%]
  - Effective address → reserved register (`%r11`)
  - Compare with the end of the data region (`_krx_edata`)

✓ Data read

✗ Code read

- Violation handler (`krx_handler`)

```
RC1  pushfq
     lea     0x154(%rsi),%r11
     cmp     $_krx_edata,%r11
     ja      L3
     popfq
     cmpl    $0x7,0x154(%rsi)
RC2  pushfq
     lea     0x140(%rsi),%r11
     cmp     $_krx_edata,%r11
     ja      L3
     popfq
     mov     0x140(%rsi),%rcx
     jg      L1
RC3  pushfq
     lea     0x130(%rsi),%r11
     cmp     $_krx_edata,%r11
     ja      L3
     popfq
     mov     0x130(%rsi),%rax
     or      $0x400000,%rax
     mov     %rax,%rdx
     shr     $0x20,%rdx
     jmp     L2
L1:
     xor     %edx,%edx
     mov     $0x1,%eax
L2:
     wrmsr
     retq
L3:
     callq   krx_handler
```

R^X: Range Checks (-O2)

- For every memory read
  - ~~Spill/Fill the `%rflags` register~~
    - `pushfq/popfq` Elimination [∼94%]
  - ~~Effective address → reserved register (`%r11`)~~
    - `lea` Elimination [∼95%]
  - Compare with the end of the data region (`_krx_edata`)

✓ Data read

✗ Code read

- Violation handler (`krx_handler`)

```
RC1 pushfq
    lea   0x154(%rsi),%r11
    cmp   $(_krx_edata-0x154),%rsi
    ja    L3
    popfq
    cmpl  $0x7,0x154(%rsi)
RC2 pushfq
    lea   0x140(%rsi),%r11
    cmp   $(_krx_edata-0x140),%rsi
    ja    L3
    popfq
    mov   0x140(%rsi),%rcx
    jg    L1
RC3 pushfq
    lea   0x130(%rsi),%r11
    cmp   $(_krx_edata-0x130),%rsi
    ja    L3
    popfq
    mov   0x130(%rsi),%rax
    or    $0x400000,%rax
    mov   %rax,%rdx
    shr   $0x20,%rdx
    jmp   L2
L1:
    xor   %edx,%edx
    mov   $0x1,%eax
L2:
    wrmsr
    retq
L3:
    callq krx_handler
```

Micro-benchmarks (LMBench)

	Benchmark	SFI(-O0)	SFI(-O1)	SFI(-O2)
Latency	syscall()	126.90%	13.41%	13.44%
	open()/close()	306.24%	39.01%	37.45%
	read()/write()	215.04%	22.05%	19.51%
	select(10 fds)	119.33%	10.24%	9.93%
	select(100 TCP fds)	**1037.33%**	**59.03%**	**49.00%**
	fstat()	489.79%	15.31%	13.22%
	mmap()/munmap()	180.88%	7.24%	6.62%
	fork()+exit()	208.86%	14.32%	14.26%
	fork()+execve()	191.83%	10.30%	21.75%
	fork()+/bin/sh	113.77%	11.62%	19.22%
	sigaction()	63.49%	0.19%	**~0%**
	Signal delivery	123.29%	18.05%	16.74%
	Protection fault	**13.40%**	1.26%	0.97%
	Page fault	202.84%	**~0%**	**~0%**
	Pipe I/O	126.26%	22.91%	21.39%
	UNIX socket I/O	148.11%	12.39%	17.31%
	TCP socket I/O	171.93%	25.15%	20.85%
	UDP socket I/O	208.75%	25.71%	30.89%
	Average	**224.89%**	**17.12%**	**17.36%**
Bandwidth	Pipe I/O	46.70%	0.96%	1.62%
	UNIX socket I/O	35.77%	3.54%	4.81%
	TCP socket I/O	**53.96%**	**10.90%**	**10.25%**
	mmap() I/O	**~0%**	**~0%**	**~0%**
	File I/O	23.57%	**~0%**	**~0%**
	Average	**32%**	**3.08%**	**3.34%**

R^X: Range Checks (-O3)

- For every memory read
  - ~~Spill/Fill the %rflags register~~
  - pushfq/popfq Elimination [~94%]
  - ~~Effective address → reserved register (%r11)~~
  - lea Elimination [~95%]
  - Compare with the end of the data region (_krx_edata)

✓ Data read

✗ Code read

- Violation handler (krx_handler)

RC1
~~pushfq~~
~~lea 0x154(%rsi),%r11~~
cmp $(_krx_edata-0x154),%rsi
ja L3
~~popfq~~

cmpl $0x7,0x154(%rsi)

RC2
pushfq
~~lea 0x140(%rsi),%r11~~
cmp $(_krx_edata-0x140),%rsi
ja L3
popfq

mov 0x140(%rsi),%rcx
jg L1

RC3
~~pushfq~~
~~lea 0x130(%rsi),%r11~~
cmp $(_krx_edata-0x130),%rsi
ja L3
~~popfq~~

mov 0x130(%rsi),%rax
or $0x400000,%rax
mov %rax,%rdx
shr $0x20,%rdx
jmp L2

L1:
xor %edx,%edx
mov $0x1,%eax

L2:
wrmsr
retq

L3:
callq krx_handler

R^X: Range Checks (-O3)

- ~~For every memory read~~
  - cmp/ja Coalescing [∼50%]
  - ~~Spill/Fill the %rflags register~~
  - pushfq/popfq Elimination [∼94%]
  - ~~Effective address → reserved register (%r11)~~
  - lea Elimination [∼95%]
  - Compare with the end of the data region (_krx_edata)

✓ Data read

✗ Code read

- Violation handler (krx_handler)

RC1
~~pushfq~~
~~lea 0x154(%rsi),%r11~~
cmp $(_krx_edata-0x154),%rsi
ja L3
~~popfq~~

cmpl $0x7,0x154(%rsi)

RC2
~~pushfq~~
~~lea 0x140(%rsi),%r11~~
~~cmp $(_krx_edata-0x140),%rsi~~
~~ja L3~~
~~popfq~~

mov 0x140(%rsi),%rcx
jg L1

RC3
~~pushfq~~
~~lea 0x130(%rsi),%r11~~
~~cmp $(_krx_edata-0x130),%rsi~~
~~ja L3~~
~~popfq~~

mov 0x130(%rsi),%rax
or $0x400000,%rax
mov %rax,%rdx
shr $0x20,%rdx
jmp L2

L1:
xor %edx,%edx
mov $0x1,%eax

L2:
wrmsr
retq

L3:
callq krx_handler

Micro-benchmarks (LMBench)

	Benchmark	SFI(-O0)	SFI(-O1)	SFI(-O2)	SFI(-O3)
Latency	`syscall()`	126.90%	13.41%	13.44%	12.74%
	`open()/close()`	306.24%	39.01%	37.45%	**24.82%**
	`read()/write()`	215.04%	22.05%	19.51%	18.11%
	`select(10 fds)`	119.33%	10.24%	9.93%	10.25%
	`select(100 TCP fds)`	**1037.33%**	**59.03%**	**49.00%**	**~0%**
	`fstat()`	489.79%	15.31%	13.22%	7.91%
	`mmap()/munmap()`	180.88%	7.24%	6.62%	1.97%
	`fork()+exit()`	208.86%	14.32%	14.26%	7.22%
	`fork()+execve()`	191.83%	10.30%	21.75%	23.15%
	`fork()+/bin/sh`	113.77%	11.62%	19.22%	12.98%
	`sigaction()`	63.49%	0.19%	**~0%**	0.16%
	Signal delivery	123.29%	18.05%	16.74%	7.81%
	Protection fault	**13.40%**	1.26%	0.97%	1.33%
	Page fault	202.84%	**~0%**	**~0%**	7.38%
	Pipe I/O	126.26%	22.91%	21.39%	15.12%
	UNIX socket I/O	148.11%	12.39%	17.31%	11.69%
	TCP socket I/O	171.93%	25.15%	20.85%	16.33%
	UDP socket I/O	208.75%	25.71%	30.89%	16.96%
	Average	**224.89%**	**17.12%**	**17.36%**	**10.88%**
Bandwidth	Pipe I/O	46.70%	0.96%	1.62%	0.68%
	UNIX socket I/O	35.77%	3.54%	4.81%	**6.43%**
	TCP socket I/O	**53.96%**	**10.90%**	**10.25%**	6.05%
	`mmap()` I/O	**~0%**	**~0%**	**~0%**	**~0%**
	File I/O	23.57%	**~0%**	**~0%**	0.67%
	Average	**32%**	**3.08%**	**3.34%**	**2.77%**

R^X: Range Checks (-O3)

- ~~For every memory read~~
  - ➤ cmp/ja Coalescing [∼50%]
  - ~~Spill/Fill the %rflags register~~
  - ➤ pushfq/popfq Elimination [∼94%]
  - ~~Effective address → reserved register (%r11)~~
  - ➤ lea Elimination [∼95%]
  - Compare with the end of the data region (_krx_edata)

```
RC1 cmp    $(_krx_edata-0x154),%rsi
    ja     L3
    cmpl   $0x7,0x154(%rsi)
    mov    0x140(%rsi),%rcx
    jg     L1
    mov    0x130(%rsi),%rax
    or     $0x400000,%rax
    mov    %rax,%rdx
    shr    $0x20,%rdx
    jmp    L2
L1:
    xor    %edx,%edx
    mov    $0x1,%eax
L2:
    wrmsr
    retq
L3:
    callq  krx_handler
```

✓ Data read

✗ Code read

- Violation handler (krx_handler)

R^X: Range Checks (MPX)

- New ISA extension (Intel Skylake CPUs)
  - Registers: `%bnd0` – `%bnd3`
  - Instructions: `bndcu`, `bndcl`, `bndmk`...
- *Hardware-assisted* bounds checking

R^X: Range Checks (MPX)

- New ISA extension (Intel Skylake CPUs)
  - Registers: `%bnd0` – `%bnd3`
  - Instructions: `bndcu`, `bndcl`, `bndmk`...
- *Hardware-assisted* bounds checking
- Upper bound (`%bnd0.ub`) → `_krx_edata`
  - Check before reading memory

```
RC1  bndcu $0x154(%rsi),%bnd0
     cmpl  $0x7,0x154(%rsi)
RC2  bndcu $0x140(%rsi),%bnd0
     mov   0x140(%rsi),%rcx
     jg    L1
RC3  bndcu $0x130(%rsi),%bnd0
     mov   0x130(%rsi),%rax
     or    $0x400000,%rax
     mov   %rax,%rdx
     shr   $0x20,%rdx
     jmp   L2
L1:
     xor   %edx,%edx
     mov   $0x1,%eax
L2:
     wrmsr
     retq
```

R^X: Range Checks (MPX)

- New ISA extension (Intel Skylake CPUs)
  - Registers: `%bnd0` – `%bnd3`
  - Instructions: `bndcu`, `bndcl`, `bndmk`...
- *Hardware-assisted* bounds checking
- Upper bound (`%bnd0.ub`) → `_krx_edata`
  - Check before reading memory
  - `cmp/ja` Coalescing

```
RC1  bndcu  $0x154(%rsi),%bnd0
     cmpl   $0x7,0x154(%rsi)
     mov    0x140(%rsi),%rcx
     jg     L1
     mov    0x130(%rsi),%rax
     or     $0x400000,%rax
     mov    %rax,%rdx
     shr    $0x20,%rdx
     jmp    L2
L1:
     xor    %edx,%edx
     mov    $0x1,%eax
L2:
     wrmsr
     retq
```

Micro-benchmarks (LMBench)

	Benchmark	SFI(-O0)	SFI(-O1)	SFI(-O2)	SFI(-O3)	MPX
Latency	syscall()	126.90%	13.41%	13.44%	12.74%	0.49%
	open()/close()	306.24%	39.01%	37.45%	**24.82%**	3.47%
	read()/write()	215.04%	22.05%	19.51%	18.11%	0.63%
	select(10 fds)	119.33%	10.24%	9.93%	10.25%	1.26%
	select(100 TCP fds)	**1037.33%**	**59.03%**	**49.00%**	**~0%**	**~0%**
	fstat()	489.79%	15.31%	13.22%	7.91%	**~0%**
	mmap()/munmap()	180.88%	7.24%	6.62%	1.97%	1.12%
	fork()+exit()	208.86%	14.32%	14.26%	7.22%	**~0%**
	fork()+execve()	191.83%	10.30%	21.75%	23.15%	**~0%**
	fork()+/bin/sh	113.77%	11.62%	19.22%	12.98%	**6.27%**
	sigaction()	63.49%	0.19%	**~0%**	0.16%	1.01%
	Signal delivery	123.29%	18.05%	16.74%	7.81%	1.12%
	Protection fault	**13.40%**	1.26%	0.97%	1.33%	**~0%**
	Page fault	202.84%	**~0%**	**~0%**	7.38%	1.64%
	Pipe I/O	126.26%	22.91%	21.39%	15.12%	0.42%
	UNIX socket I/O	148.11%	12.39%	17.31%	11.69%	4.74%
	TCP socket I/O	171.93%	25.15%	20.85%	16.33%	1.91%
	UDP socket I/O	208.75%	25.71%	30.89%	16.96%	**~0%**
	Average	**224.89%**	**17.12%**	**17.36%**	**10.88%**	**1.34%**
Bandwidth	Pipe I/O	46.70%	0.96%	1.62%	0.68%	**~0%**
	UNIX socket I/O	35.77%	3.54%	4.81%	**6.43%**	**1.43%**
	TCP socket I/O	**53.96%**	**10.90%**	**10.25%**	6.05%	**~0%**
	mmap() I/O	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**
	File I/O	23.57%	**~0%**	**~0%**	0.67%	0.28%
	Average	**32%**	**3.08%**	**3.34%**	**2.77%**	**0.34%**

Special Cases

- Safe reads [∼4%]
  - `%rip`-relative symbols
  - Absolute memory reads
- String operations (`cmps`, `lods`, `movs`)
  - Check using `%rsi`
  - `rep`-prefixed instructions → place RC **after** read operation
    - Postmortem detection
    - Allows code optimizations

Stack Reads

- `offset(%rsp,%index,scale)` $\rightarrow$ RC
- `offset(%rsp)` $\rightarrow$ no RC
  - Guard section (`.krx_phantom`) between `_krx_edata` and beginning of code
    - `sizeof(.krx_phantom)` $\geq$ **max** offset

Fine-grained KASLR

- Code block permutation

Fine-grained KASLR

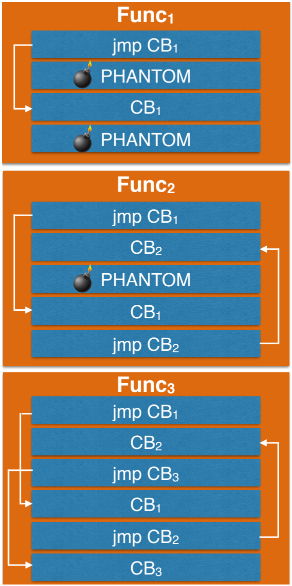
- Code block permutation
  - Too few blocks → **Phantom blocks**
    - `int3` instructions
    - Random size

Fine-grained KASLR

- Code block permutation
  - Too few blocks → **Phantom blocks**
    - `int3` instructions
    - Random size
  - Randomly permute the code blocks

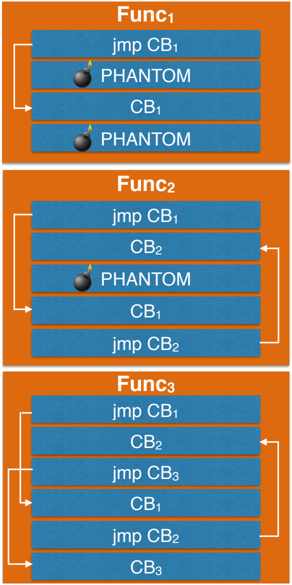
Fine-grained KASLR

- Code block permutation
  - Too few blocks → **Phantom blocks**
    - `int3` instructions
    - Random size
  - Randomly permute the code blocks
    - Preserve control flow → `jmp` instructions



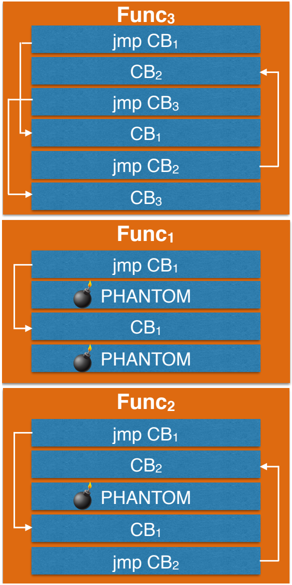
Fine-grained KASLR

- Code block permutation
  - Too few blocks → **Phantom blocks**
    - `int3` instructions
    - Random size
  - Randomly permute the code blocks
    - Preserve control flow → `jmp` instructions
  - Unpredictable internal function layout



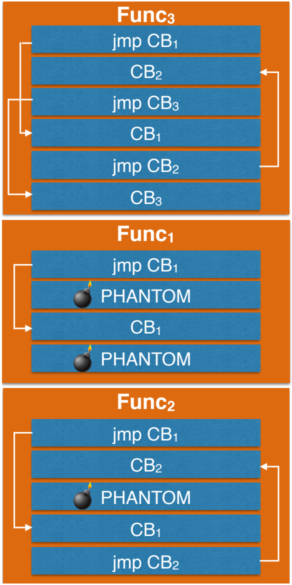
Fine-grained KASLR

- Code block permutation
  - Too few blocks → **Phantom blocks**
    - `int3` instructions
    - Random size
  - Randomly permute the code blocks
    - Preserve control flow → `jmp` instructions
  - Unpredictable internal function layout
- Function permutation



Fine-grained KASLR

- Code block permutation
  - Too few blocks → **Phantom blocks**
    - `int3` instructions
    - Random size
  - Randomly permute the code blocks
    - Preserve control flow → `jmp` instructions
  - Unpredictable internal function layout
- Function permutation
  - Unpredictable surrounding area of function



Return Address Protection

Return Address Encryption (X)

```
mov offset(%rip),%r11
xor %r11,(%rsp)
```

- XOR-based encryption
- Unique key per routine
  - Placed in the non-readable region
  - Replenished at boot/load time

Return Address Protection (cont'd)

Return Address Decoys (D)

Decoy | Real	**Real | Decoy**
push %r11	mov (%rsp),%rax
	mov %r11,(%rsp)
	push %rax

- Decoy return address
  - Point at phantom instructions
    - Call site → address in %r11
  - Placed before/after the real one

Phantom Instructions

```
49 C7 C3 CC 00 00 00        mov $0xcc,%r11
```

- Conceptually NOP instructions
- Contain unaligned “tripwire” opcodes
  - Raise #BR exception
- Inserted in routines’ code stream
- Address of the “tripwire” → callee

Phantom Instructions

49 C7 C3 CC 00 00 00	int3 ~~mov $0xcc,%r11~~

- Conceptually NOP instructions
- Contain unaligned "tripwire" opcodes
  - Raise #BR exception
- Inserted in routines' code stream
- Address of the "tripwire" → callee

Limitations

- Race hazards
  - Read the addresses before being encrypted/hidden among decoys

Limitations

- Race hazards
  - Read the addresses before being encrypted/hidden among decoys
  - Difficult to time reliably in the kernel setting

Limitations

- Race hazards
  - Read the addresses before being encrypted/hidden among decoys
  - Difficult to time reliably in the kernel setting
    - System-call interface
    - Process scheduling
    - Cache/TLB

Limitations

- Race hazards
  - Read the addresses before being encrypted/hidden among decoys
  - Difficult to time reliably in the kernel setting
    - System-call interface
    - Process scheduling
    - Cache/TLB
- Substitution attacks
  - Replace the (protected) return address with one of a different call-site

Limitations

- Race hazards
  - Read the addresses before being encrypted/hidden among decoys
  - Difficult to time reliably in the kernel setting
    - System-call interface
    - Process scheduling
    - Cache/TLB
- Substitution attacks
  - Replace the (protected) return address with one of a different call-site
  - Must use **valid** dynamically leaked return sites
  - Can be prevented with register randomization

Performance Evaluation

Benchmark	Metric	SFI	MPX
Apache	Req/s	0.54%	0.48%
PostgreSQL	Trans/s	3.36%	1.06%
Kbuild	sec	1.48%	0.03%
Kextract	sec	0.52%	˜ 0%
GnuPG	sec	0.15%	˜ 0%
OpenSSL	Sign/s	˜ 0%	˜ 0%
PyBench	msec	˜ 0%	˜ 0%
PHPBench	Score	0.06%	˜ 0%
IOzone	MB/s	4.65%	˜ 0%
DBench	MB/s	0.86%	˜ 0%
PostMark	Trans/s	13.51%	1.81%
Average		**2.15%**	**0.45%**

Macro-benchmarks (Phoronix Test Suite)

Performance Evaluation

Benchmark	Metric	SFI	MPX	SFI+D	SFI+X
Apache	Req/s	0.54%	0.48%	0.97%	1.00%
PostgreSQL	Trans/s	3.36%	1.06%	6.15%	6.02%
Kbuild	sec	1.48%	0.03%	3.21%	3.50%
Kextract	sec	0.52%	˜ 0%	˜ 0%	˜ 0%
GnuPG	sec	0.15%	˜ 0%	0.15%	0.15%
OpenSSL	Sign/s	˜ 0%	˜ 0%	0.03%	˜ 0%
PyBench	msec	˜ 0%	˜ 0%	˜ 0%	0.15%
PHPBench	Score	0.06%	˜ 0%	0.03%	0.50%
IOzone	MB/s	4.65%	˜ 0%	8.96%	8.59%
DBench	MB/s	0.86%	˜ 0%	4.98%	˜ 0%
PostMark	Trans/s	13.51%	1.81%	19.99%	19.98%
Average		**2.15%**	**0.45%**	**4.04%**	**3.63%**

Macro-benchmarks (Phoronix Test Suite)

Performance Evaluation

Benchmark	Metric	SFI	MPX	SFI+D	SFI+X	MPX+D	MPX+X
Apache	Req/s	0.54%	0.48%	0.97%	1.00%	0.81%	0.68%
PostgreSQL	Trans/s	3.36%	1.06%	6.15%	6.02%	3.45%	4.74%
Kbuild	sec	1.48%	0.03%	3.21%	3.50%	2.82%	3.52%
Kextract	sec	0.52%	˜ 0%	˜ 0%	˜ 0%	˜ 0%	˜ 0%
GnuPG	sec	0.15%	˜ 0%	0.15%	0.15%	˜ 0%	˜ 0%
OpenSSL	Sign/s	˜ 0%	˜ 0%	0.03%	˜ 0%	0.01%	˜ 0%
PyBench	msec	˜ 0%	˜ 0%	˜ 0%	0.15%	˜ 0%	˜ 0%
PHPBench	Score	0.06%	˜ 0%	0.03%	0.50%	0.66%	˜ 0%
IOzone	MB/s	4.65%	˜ 0%	8.96%	8.59%	3.25%	4.26%
DBench	MB/s	0.86%	˜ 0%	4.98%	˜ 0%	4.28%	3.54%
PostMark	Trans/s	13.51%	1.81%	19.99%	19.98%	10.09%	12.07%
Average		**2.15%**	**0.45%**	**4.04%**	**3.63%**	**2.32%**	**2.62%**

Macro-benchmarks (Phoronix Test Suite)

Conclusion

- Comprehensive solution against code reuse attacks
  - R^X (Execute-only memory)
  - Fine-grained KASLR
- Utilizes hardware assistance whenever possible
  - Memory Protection Extensions (MPX)
- Low overhead
  - `SFI`-based → 3.63%
  - `MPX`-based → 2.32%

Code available soon:

`http://nsl.cs.columbia.edu/projects/krx`

Backup Slides

Micro-benchmarks (LMBench)

	Benchmark	SFI(-O0)	SFI(-O1)	SFI(-O2)	SFI(-O3)	MPX	D	X	SFI+D	SFI+X	MPX+D	MPX+X
Latency	syscall()	126.90%	13.41%	13.44%	12.74%	0.49%	0.62%	2.70%	13.67%	15.91%	2.24%	2.92%
	open()/close()	306.24%	39.01%	37.45%	**24.82%**	3.47%	**15.03%**	**18.30%**	**40.68%**	**44.56%**	**19.44%**	**22.79%**
	read()/write()	215.04%	22.05%	19.51%	18.11%	0.63%	7.67%	10.74%	29.37%	34.88%	9.61%	12.43%
	select(10 fds)	119.33%	10.24%	9.93%	10.25%	1.26%	3.00%	5.49%	15.05%	16.96%	4.59%	6.37%
	select(100 TCP fds)	**1037.33%**	**59.03%**	**49.00%**	**~0%**	**~0%**	**~0%**	5.08%	1.78%	9.29%	**0.39%**	7.43%
	fstat()	489.79%	15.31%	13.22%	7.91%	**~0%**	4.46%	12.92%	16.30%	26.68%	8.36%	14.64%
	mmap()/munmap()	180.88%	7.24%	6.62%	1.97%	1.12%	4.83%	5.89%	7.57%	8.71%	6.86%	8.27%
	fork()+exit()	208.86%	14.32%	14.26%	7.22%	**~0%**	12.37%	16.57%	24.03%	21.48%	13.77%	11.64%
	fork()+execve()	191.83%	10.30%	21.75%	23.15%	**~0%**	13.93%	16.38%	29.91%	34.18%	17.00%	17.42%
	fork()+/bin/sh	113.77%	11.62%	19.22%	12.98%	**6.27%**	12.37%	15.44%	23.66%	22.94%	18.40%	16.66%
	sigaction()	63.49%	0.19%	**~0%**	0.16%	1.01%	0.59%	**2.20%**	**0.46%**	**2.27%**	0.95%	**2.43%**
	Signal delivery	123.29%	18.05%	16.74%	7.81%	1.12%	3.49%	4.94%	11.39%	13.31%	5.37%	6.52%
	Protection fault	**13.40%**	1.26%	0.97%	1.33%	**~0%**	1.69%	3.27%	3.34%	5.73%	1.60%	3.39%
	Page fault	202.84%	**~0%**	**~0%**	7.38%	1.64%	7.83%	9.40%	15.69%	17.30%	10.80%	12.11%
	Pipe I/O	126.26%	22.91%	21.39%	15.12%	0.42%	4.30%	6.89%	19.39%	22.39%	6.07%	7.62%
	UNIX socket I/O	148.11%	12.39%	17.31%	11.69%	4.74%	7.34%	10.04%	16.09%	16.64%	6.88%	8.80%
	TCP socket I/O	171.93%	25.15%	20.85%	16.33%	1.91%	4.83%	8.30%	21.63%	24.43%	8.20%	9.71%
	UDP socket I/O	208.75%	25.71%	30.89%	16.96%	**~0%**	7.38%	12.76%	24.98%	26.80%	11.22%	13.28%
	Average	**224.89%**	**17.12%**	**17.36%**	**10.88%**	**1.34%**	**6.20%**	**9.3%**	**17.5%**	**20.25%**	**8.43%**	**10.25%**
Bandwidth	Pipe I/O	46.70%	0.96%	1.62%	0.68%	**~0%**	0.59%	1.00%	2.80%	3.53%	0.78%	1.61%
	UNIX socket I/O	35.77%	3.54%	4.81%	**6.43%**	**1.43%**	2.79%	3.39%	5.71%	7.00%	3.17%	3.41%
	TCP socket I/O	**53.96%**	**10.90%**	**10.25%**	6.05%	**~0%**	**3.71%**	**4.40%**	**9.82%**	**9.85%**	**3.64%**	**4.87%**
	mmap() I/O	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**	**~0%**
	File I/O	23.57%	**~0%**	**~0%**	0.67%	0.28%	1.21%	1.46%	1.81%	2.23%	1.74%	1.92%
	Average	**32%**	**3.08%**	**3.34%**	**2.77%**	**0.34%**	**1.66%**	**2.05%**	**4.03%**	**4.52%**	**1.87%**	**2.36%**