NC STATE UNIVERSITY Computer Science

Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification

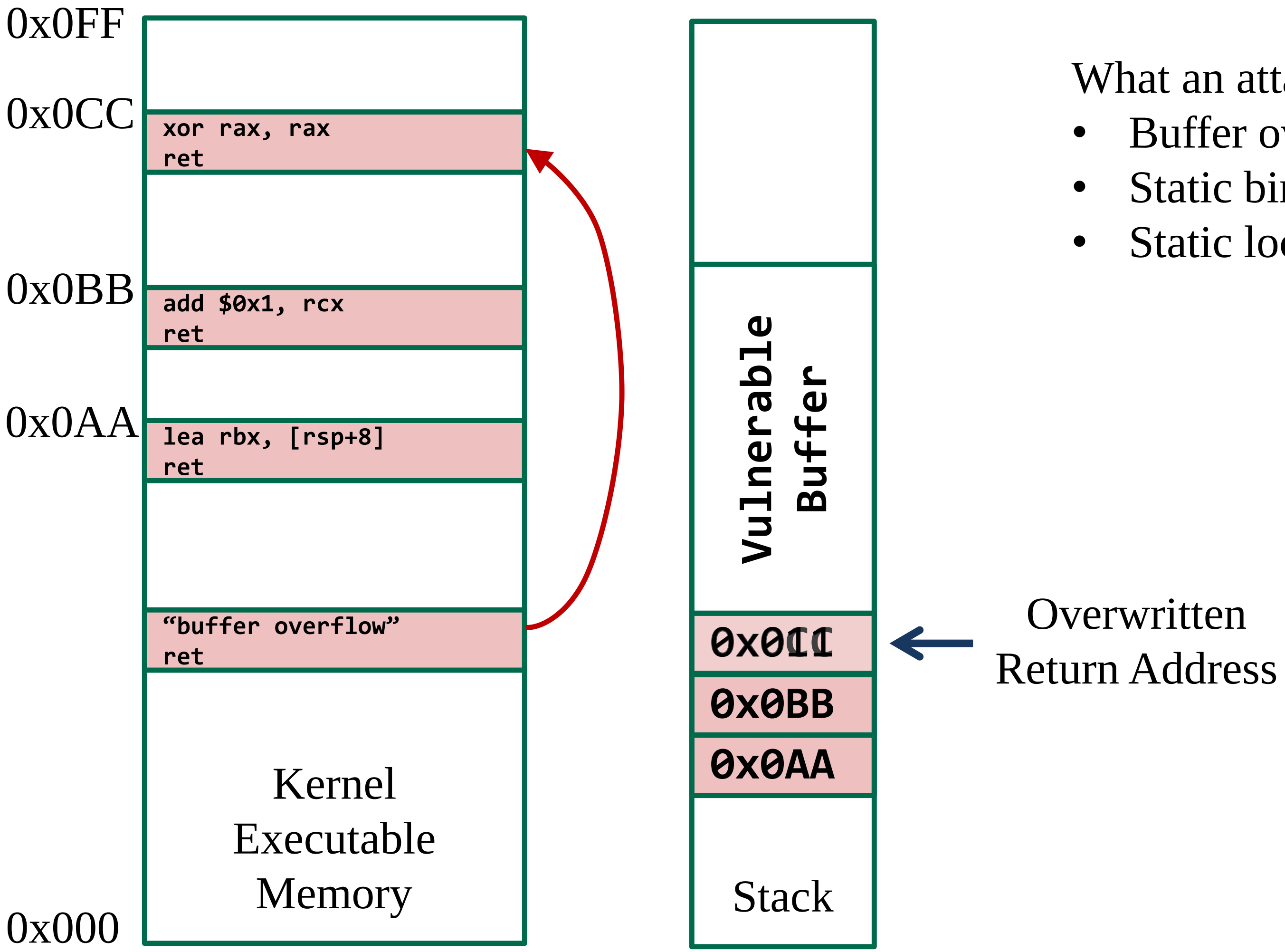
Jason Gionta, William Enck
North Carolina State University

Per Larsen
UC Irvine

Kernel Attack Vectors

- Bypass access controls
  - SELinux
- Modify non-control data
  - Data integrity [Kuzentsov et al. 2014]
- Code-reuse ← our focus
  - Return Oriented Rootkits
  - Enable arbitrary memory writes

Code-Reuse Attacks – Simple Example



What an attacker knows?

- Buffer overflow
- Static binary load location
- Static location of gadgets

Kernel Code Reuse Protections

- Requirements
  - Adoptability
  - Deployability
- Approaches
  - Policy/Access Control
    - Non-comprehensive
  - Control Flow Integrity
    - Large overhead
    - Practical challenges
  - **Software diversity**
    - Low overhead
    - Access to sources

Existing Approaches for Kernel Software Diversity

- Kernel ASLR
  - Already deployed
  - Too coarse-grained
- Function randomization
  - Limited protection
- Fine-grained instruction diversity
  - Not comprehensive
- Re-randomization
  - Micro-kernel modules only
  - Costly

Existing Approaches for Kernel Software Diversity

- Kernel ASLR
  - Already deployed

Fail to address

Memory Disclosure

  - Micro-kernel modules only
  - Costly

Memory Disclosure

- **Executable Data is Readable**
- Execute permissions imply read permissions
  - Required for execution
- Protections rely on memory secrecy
  - Software diversity
    - Just-In-Time Code Reuse [Snow et al. 2013]
  - Relaxed control flow enforcement
    - Out-of-Control: Overcoming Control Flow Integrity [Gotkas et al. 2014]
- Memory disclosure vulnerabilities
  - Originate from memory corruption bugs
  - Leak raw memory
    - Information Leaks Without Memory Disclosure [Seibert et al. 2014]

Challenges

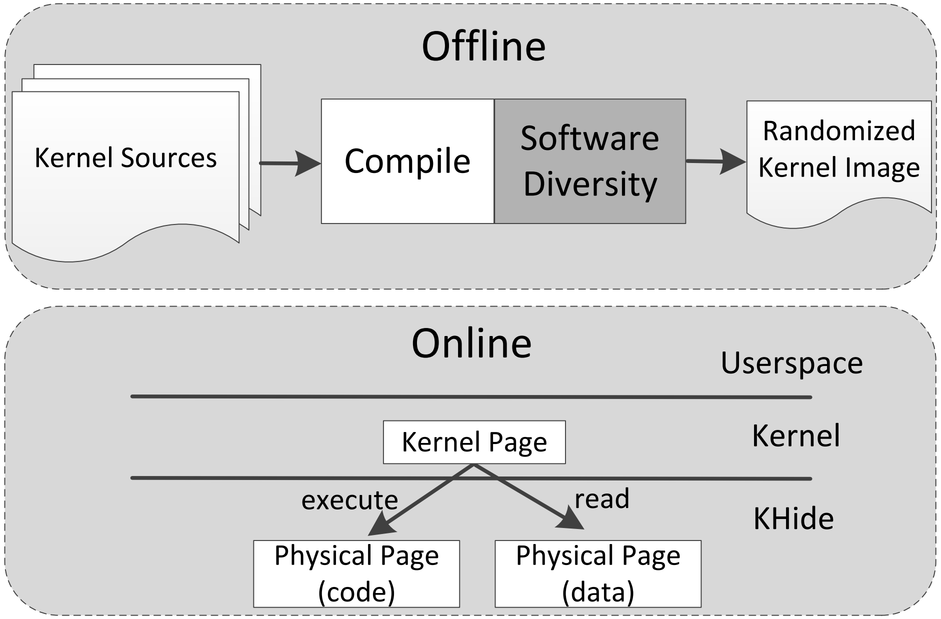
- Comprehensive
  - All code must be randomized
  - Leaked code pointers imply gadget offsets
- Protect kernel code against disclosure
  - Support reading of kernel code
  - Existing approaches rely on kernel

KHide Code Reuse Protections

- Comprehensive fine-grained instruction diversity of kernel code
  - Address limitations of existing approaches
  - Apply to Linux kernel
- Protect against disclosure of kernel code
  - Support existing code reading requirements
- Result:
  - No gadgets exist across diversified kernels
  - Adversaries must guess code reuse gadgets

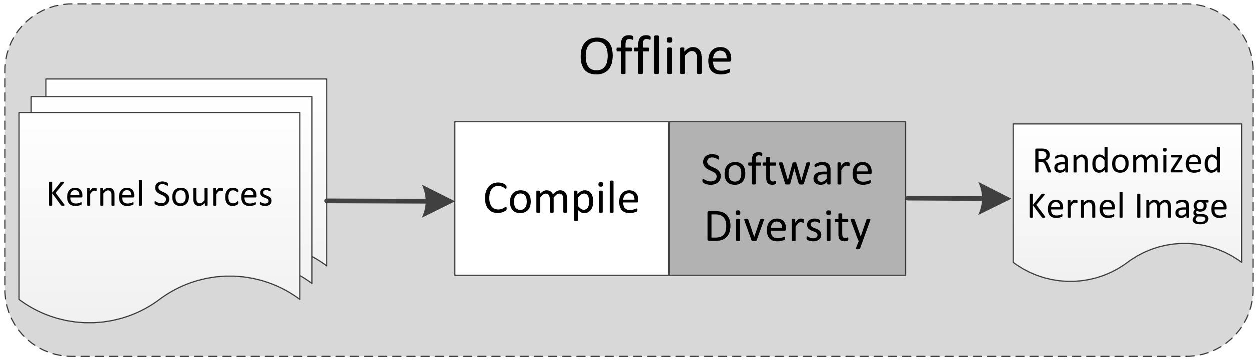
KHide Architecture

- Comprehensive fine-grained instruction diversity of kernel code ← offline
- Protect against kernel code disclosure ← online



Challenges

- Comprehensive
  - All code must be randomized
  - Leaked code pointers imply gadget offsets
- Protect kernel code against disclosure
  - Support reading of kernel code
  - Existing approaches rely on kernel



Existing Fine-Grained Instruction Diversity

- Diversity of kernel sources
- NOP insertions [Homescu et al. 2013]
  - Low overhead
  - Native compiler support
    - No assembly support
  - Not enough

known a priori

```
0xF000:  push rbp
         …
0xFF10:  call 0x5400 ; printk
0xFF13:  pop rax
         pop rbx
         pop rbp
         ret
```

No Diversity

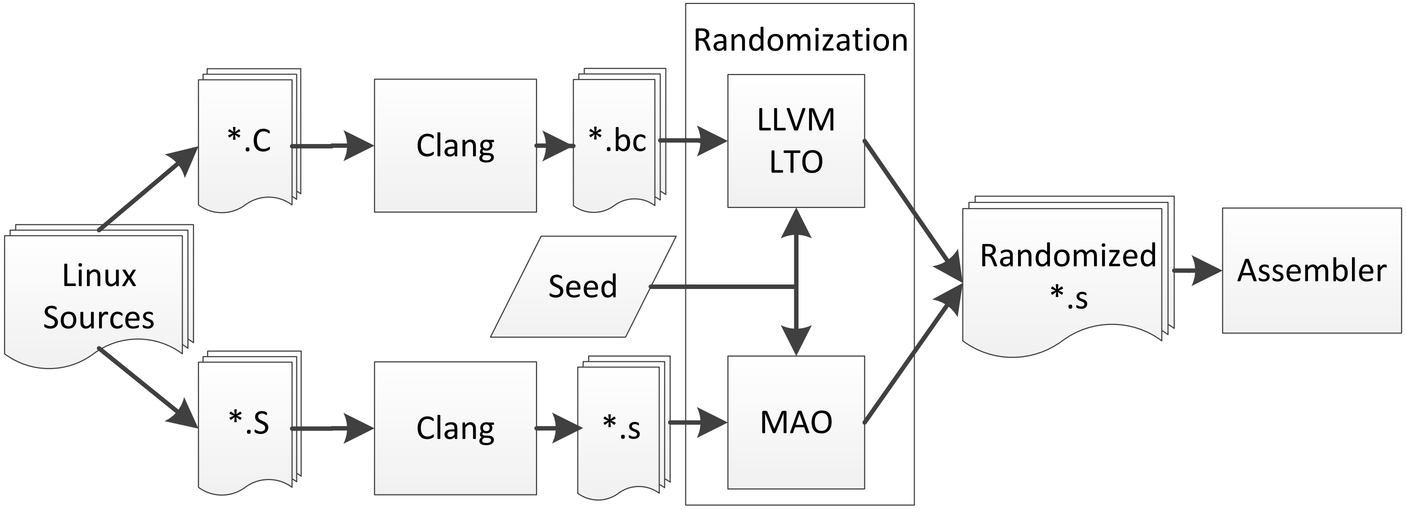
different location

```
0xF000:  push rbp
         nop
         …
0xFF14:  call 0x5400 ; printk
0xFF17:  nop
         pop rax
         nop
         pop rbx
         pop rbp
         ret
```

NOP Insertions

Comprehensive Fine-Grained Instruction Diversity

- Problem 1: diversify Linux kernel
  - LLVMLinux
  - Clang/LLVM - NOP insertions [Homescu et al. 2013]
- Problem 2: assembly source files
  - Micro-Architectural Optimizer (MAO) [Hundt et al. 2011]
    - Decompose post-processed assembly
  - *NOP insertions*



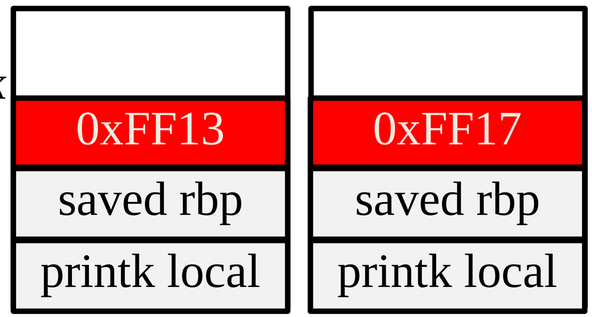
Comprehensive
Fine-Grained Instruction Diversity

- Problem 3: Leaked pointers can imply gadgets

```
0xF000:  push rbp
         …
0xFF10:  call 0x5400 ; printk
0xFF13:  pop rax
         pop rbx
         pop rbp
         ret
```

No Diversity

stacks for printk




```
0xF000:  push rbp
         nop
         …
0xFF14:  call 0x5400 ; printk
0xFF17:  nop
         pop rax
         nop
         pop rbx
         pop rbp
         ret
```

NOP Insertions

```
pop rax
pop rbx
pop rbp
ret
```

Shared Identical Gadget

■ Leaked Return Address

Comprehensive
Fine-Grained Instruction Diversity

- Problem 3: Leaked pointers can imply gadgets
  - Register randomization ← novel approach for LLVM
    - Modify selection order
    - Some register remain static
      - pop rbp; ret;

```
0xF000:  push rbp
         nop
         …
0xFF14:  call 0x5400 ; printk
0xFF17:  nop
         pop rax
         nop
         pop rbx
         pop rbp
         ret
```

NOP Insertions

```
0xF000:  push rbp
         nop
         …
0xFF14:  call 0x5400 ; printk
0xFF17:  nop
         pop rsi
         nop
         pop rdx
         pop rbp
         ret
```

+Register Randomization

Comprehensive Fine-Grained Instruction Diversity

- Problem 3: Leaked pointers can imply gadgets
  - Call Site Lifting ← novel contribution
    - Decouple return address
    - Cannot determine location of gadget

```
0xF000:  push rbp
         nop
         …
0xFF14:  call 0x5400 ; printk
0xFF17:  nop
         pop rax
         nop
         pop rbx
         pop rbp
         ret
```

NOP Insertions

```
0xF000:  push rbp
         nop
         …
0xFF14:  call 0x5400 ; printk
0xFF17:  nop
         pop rsi
         nop
         pop rdx
         pop rbp
         ret
```

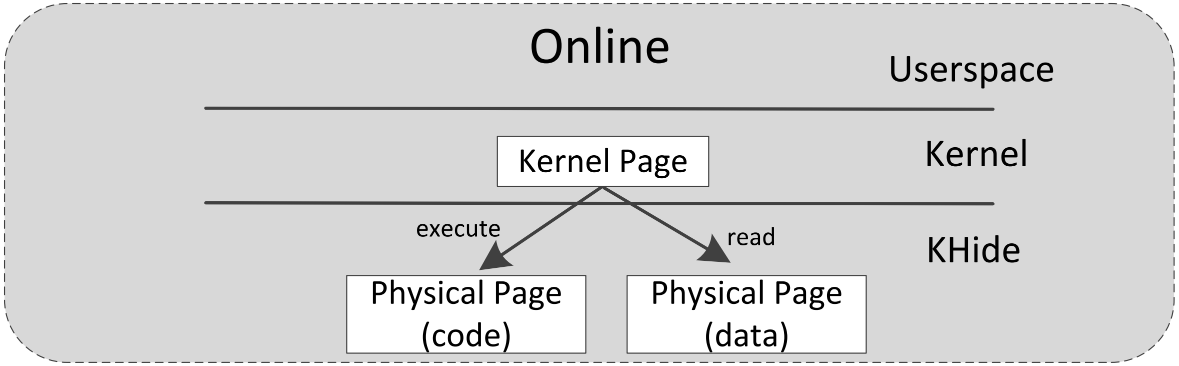
+Register Randomization

```
0xF000:  nop
         jmp 0xF100
0xF004:  call 0x5400 ; printk
0xF009:  jmp 0xFF63
         …
0xF100:  push rbp
         …
         jmp 0xF004
0xFF63:  nop
         pop rsi
         nop
         pop rdx
         pop rbp
         ret
```

+Call Site Lifting

Challenges

- Comprehensive
  - All code must be randomized
  - Leaked code pointers imply gadget offsets
- Protect kernel code against disclosure
  - Support reading of kernel code
  - Existing approaches rely on kernel



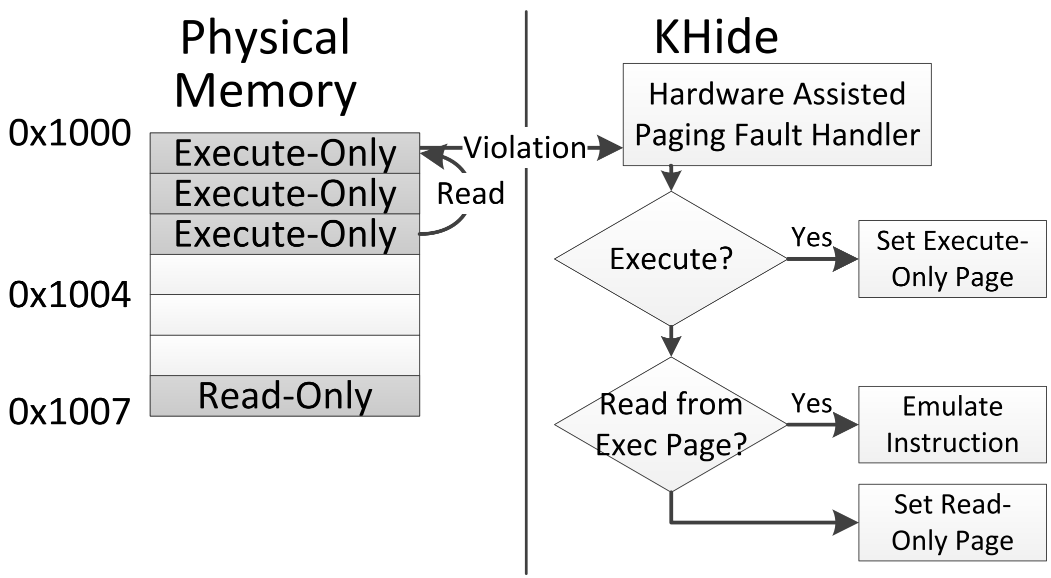
NC STATE UNIVERSITY Computer Science

Memory Disclosure Protection

- Existing approach limitations
  - XnR [Backes et al. 2014]
    - Limited protection model
    - Kernel enforces protection
  - HideM
    - Reliance on split-TLB
      - Limited support in new architectures
      - TLB pressure
    - Kernel enforces protection
- Apply and enforce protections outside kernel

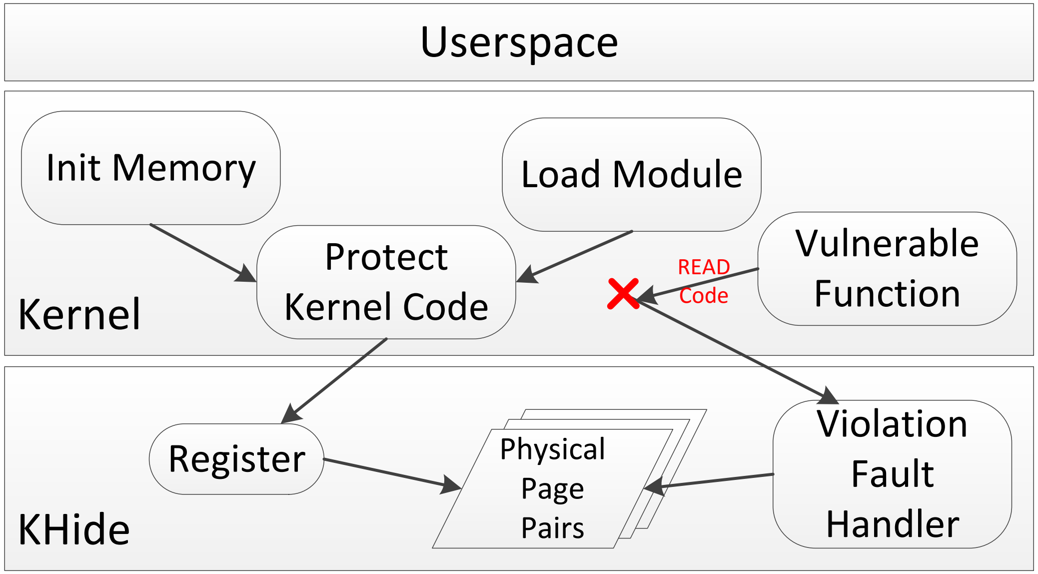
Memory Disclosure Protection

- Virtualization Extensions
  - Hardware Assisted Paging
  - Execute-only support
- Enforce code reading policy



Memory Disclosure Protection

- Protect Executable Pages
  - Register executable-pages
  - On-boot and On-demand
- Pair with readable data pages
  - Data that may be read (code reading policy)



Build Code Reading Policy

- Find executable data to be read
  - Instrument compiler
    - Record locations of non-code is written to code section
    - Write locations to read-only data in kernel image
  - Function tracing support
    - kprobes/ftrace
      - Read first byte; write interrupt
    - Insert NOP instruction to beginning of all functions

Empirical Evaluation: Setup

- LLVMLinux kernel 3.18
  - 5 diversified kernels generated
    - Performance
    - Security
  - NOPs inserted at 50% probability [Homescu et al. 2013]
- KHide built on KVM
- Performance: three configurations
  - Native
  - Diversity
  - KHide

Empirical Evaluation: Performance

- Microbench
  - LMBench [McVoy et al. 1996]

Time in Microseconds

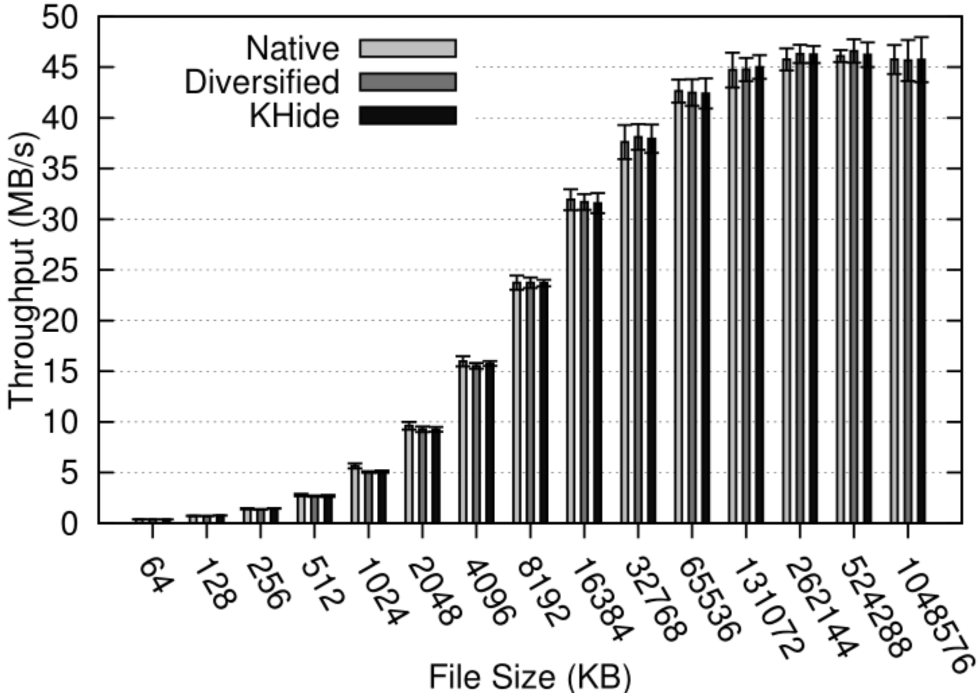
Test	Native	Diverse		KHide		*ASLP*	*KCoFI*	*VGhost*
page fault	0.139	0.153	10%	0.151	8.9%	-	11%	15%
null syscall	0.0352	0.0388	10%	0.0384	9%	-	50%	290%
sig. handler install	0.101	0.117	15.2%	0.116	14.6%	-	113%	224%
sig. handler delivery	0.622	0.749	20.4%	0.743	19.6%	-	-.08%	61%
fork + exit	77.29	93.83	31.4%	92.8	20%	-	250%	340%
fork + exec	81.38	100.4	23.4%	99.38	22.1%	12.52%	210%	320%
select	4.7	6.87	46%	6.84	45.4%	-	60%	240%
open/close	0.777	1.191	53.3%	1.17	50.7%	-	110%	383%

File creations per second

File Size	Native	Diversity		KHide		*KCoFI*	*VGhost*
0 KB	192k	125k	35%	126k	34%	128%	363%
1 KB	125k	79.7k	36%	80.6k	35%	147%	421%
4 KB	123k	79.2k	35%	80k	33%	148%	419%
10 KB	92.5k	60k	35%	60.4k	35%	138%	371%

Empirical Evaluation: Performance

- Macrobench: Network
  - SSH File Transfer – files 1KB-1GB
    - Average: 1%
    - High: 10%
    - KCoFI: 14%



Empirical Evaluation: Memory Overhead

- Kernel image size
  - bzImage
    - 5.1MB (No diversity) → 6.3MB (23%)
  - In memory code
    - 7.9MB (No diversity) → 10MB (27%)
- Code reading policy
  - One shared shadow read page
    - No embedded jump-tables
  - 4KB

Empirical Evaluation

- Security
  - Attacker Model:
    - Attackers guess gadgets
    - Find common gadgets across diversified versions
  - Calculate gadget survivability [Homescu et al. 2013]
    - Find identical gadgets at same offsets
      - Accounting for NOPs
      - Across diversified versions

-	2	3	4	5
# Survived	6258	116	2	0

Conclusion

- KHide provides comprehensive code-reuse protection for kernels
  - All code diversified
  - Addresses limitations of previous diversity techniques
  - Provides runtime protection against code disclosure
- Adoptable design
  - Applied to Linux kernel
  - Low performance impact
  - No gadgets survive diversification

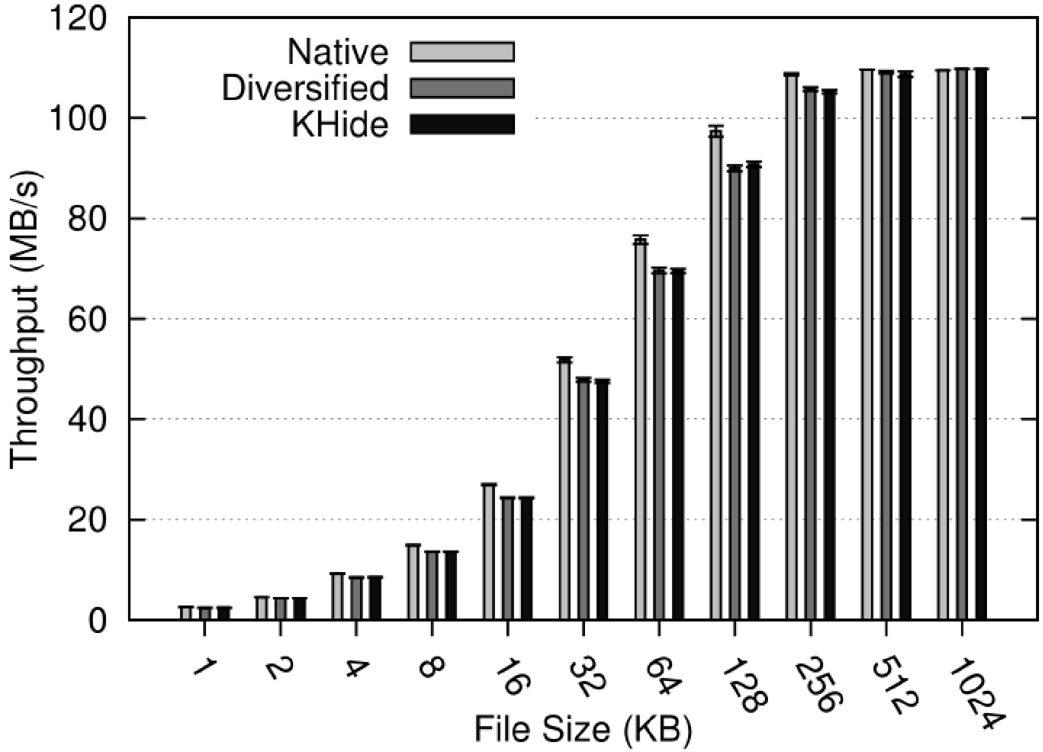
Thanks

- Questions?

✉ jason@gionta.org
💻 gionta.org

Empirical Evaluation: Performance

- Macrobench: Network
  - ApacheBench – 32 clients 10,000 requests
    - Apache webserver
    - Average: 4%
    - High: 9%
    - ASLP:14%
      - 100 Clients
    - KCoFI: 0%
      - thttpd



Empirical Evaluation: Performance

- Macrobench: Disk
  - Postmark – simulate email server
    - 500,000 transactions

Config	Time(s)	Std. Dev	95% Conf.	Overhead
Native	13.57	0.188	±0.052	
Diversify	17.3	0.974	±0.12	28%
KHide	16.63	1.31	±0.16	23%
KCoFI	-	-	-	96%
VGhost	-	-	-	372%