Preventing exploits against memory-corruption vulnerabilities

Chengyu Song

Georgia Tech

Agenda

- Memory corruption vulnerability
- Thesis Statement
- Approaches
  - SDCG
  - Kenali
  - HDFI
- Conclusion

Memory corruption vulnerability

- One of most **prevalent** vulnerabilities
  - Very common for C/C++ programs
- One of most **devastating** vulnerabilities
  - Highly exploitable, e.g., arbitrary code execution
- One of most **widely exploited** vulnerabilities

Root causes

- Spatial errors
  - Missing bound check, incorrect bound check, format string, type confusion, integer overflow, etc.
- Temporal errors
  - Use-after-free, uninitialized data

Exploit techniques

- Code injection (modification) attacks
- Control flow hijacking attacks
- Data-oriented attacks
- Information leak
- Uninitialized data use

Defense mechanisms (1)

- Memory error detector
  - Spatial: adding bound checks for memory accesses
    - Software-based: CCured, Cyclone, SoftBound, etc.
    - Hardware-based: HardBound, CHERI, WatchDog[Lite], MPX, etc.
  - Temporal: tracking initialization/liveness
    - Software-based: memory sanitizer, CETS, DangNull
    - Hardware-based: SafeProc, WatchDogLite

Defense mechanisms (2)

- Exploit prevention techniques
  - Code corruption/injection: W^X, ret2usr protection
  - Control flow hijacking: stack cookies, CFI, vtable pointer protection, etc.
  - Data-oriented attacks: SFI, DFI
  - Code pointers leak: PointerGuard, ASLR-Guard
  - Code leak: execute-only memory
  - Generic information leak: DFI, DIFT

Summary of existing mechanisms

- Memory error detectors
  - Pros: fundamentally solves the problem
  - Cons: high performance overhead, even with hardware
- Exploit prevention techniques
  - Pros: lower performance overhead
  - Cons: bypassable

Problem Statement

- How to build **principled** and **practical** defense techniques against memory-corruption-based exploits
- Two goals
  - Principled: cannot be easily bypassed
  - Practical: low performance overhead, easy to adopt

Approaches

- Preventing **code injection attacks**: SDCG [NDSS'15]
- Preventing **data-oriented attacks**: Kenali [NDSS'16]
- Improving **security** and **performance**: HDFI [SP'16]

Approaches

- Preventing code injection attacks: **SDCG** [NDSS'15]
- Preventing data-oriented attacks: Kenali [NDSS'16]
- Improving security and performance: HDFI [SP'16]

Code Injection Attacks ?!

- Dates back to the Morris worm
- Used to be the most popular exploit technique
- Should have been eliminated by data execution prevention (DEP)

Rising from dead

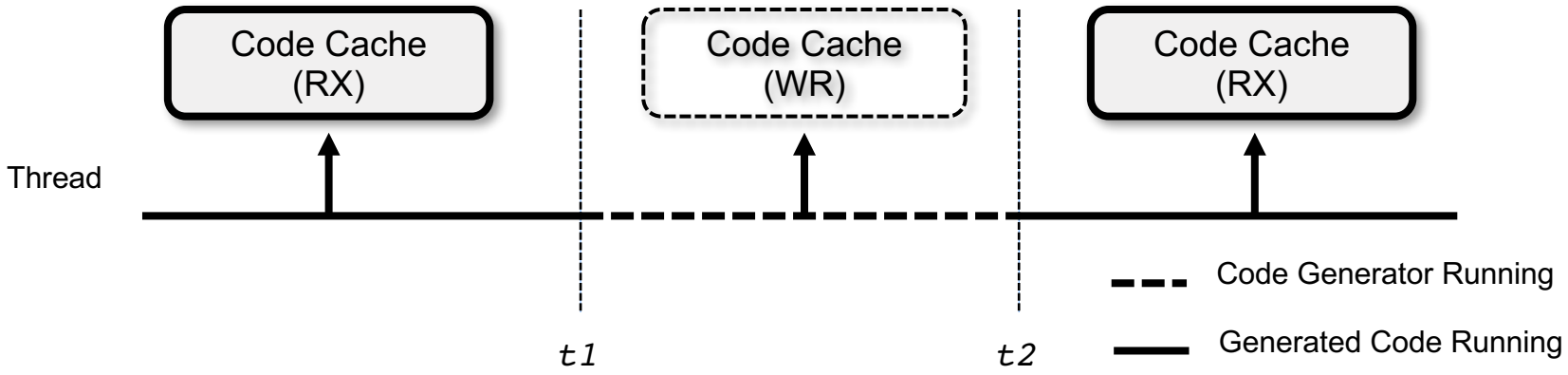
- Dynamic code generation
  - Creates native code at runtime
- Widely used by
  - Just-in-time (JIT) compilers and dynamic binary translators (DBT)
- The confliction
  - Code cache must be both writable and executable

A $50k attack

- Mobile Pwn2Own Autumn 2013 – Chrome browser on Android

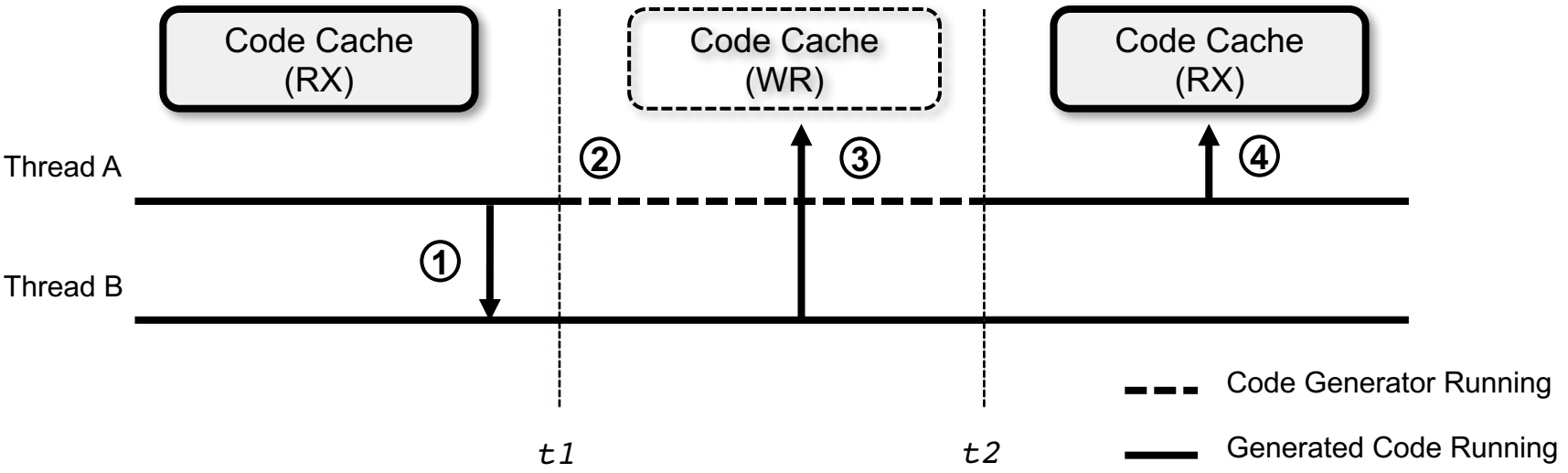
  1) Exploited an integer overflow vulnerability to overwrite the size attribute of a WFT::ArrayBuffer object → **arbitrary memory read/write** capability
  2) Leverage the arbitrary memory read capability to traverse memory and **locate the code cache**;
  3) Leverage the arbitrary memory write capability to **overwrite a JavaScript function** with shellcode that allows attackers to invoke any function with any argument;
  4) Leverage the **arbitrary code execution** capability to take out next attack step.

A simple idea



- Enforce that code pages can never be both writable and executable **at the same time**
  - Has been adopted by some JIT compilers
    - Mobile Safari, Internet Explorer, Firefox

Exploiting race condition



1) Synchronization
2) Thread-A triggers the code generation
3) Thread-B attacks thread-A's code cache
4) Thread-A execute injected shell code

How realistic is the attack

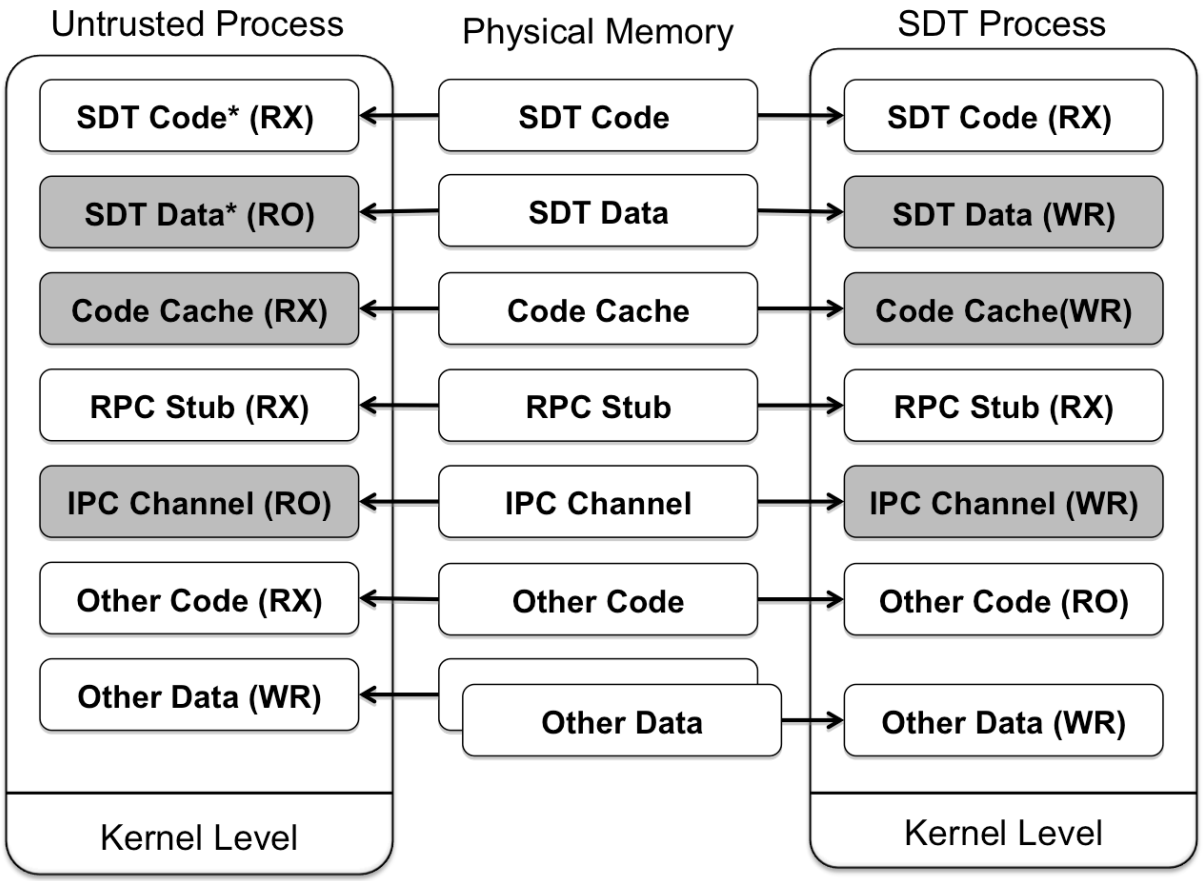
- Multi-thread programming is widely supported
- Thread synchronization latencies are usually smaller than the attack window
- Page access permission change can enlarge the attack window
- Our preliminary experiment had **91%** success rate

Design principles

- Only the code generator can write to the code cache
- W^X policy should always be enforced
  - including temporal: WR → RX

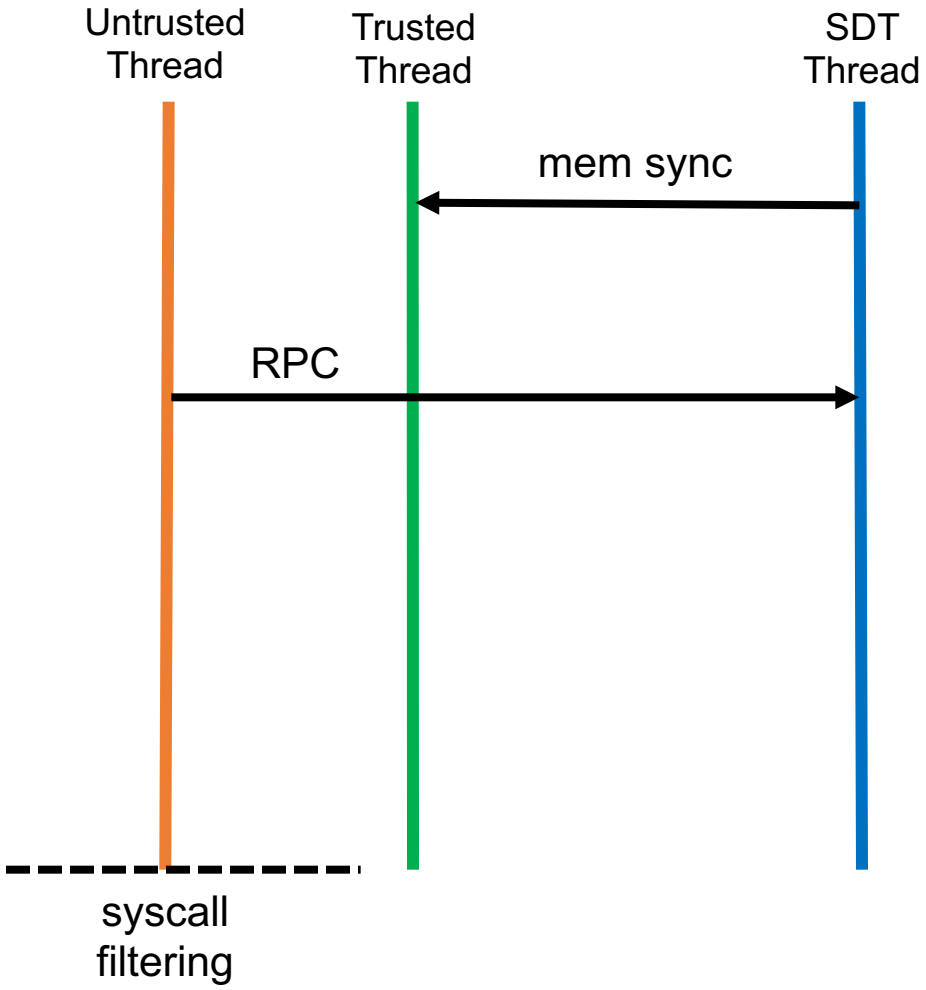
SDCG: overview

- A multi-process-based protection scheme



SDT = software dynamic translator

Implementation challenges



- Memory map synchronization
- Remote procedure call (RPC)
- Access permission enforcement

Two Prototypes

- Sharable infrastructure (~500 LoC)
  - Seccomp-sandbox (from Google Chrome)
  - Shared memory pool
  - System call filtering
- SDT-specific modification
  - Strata (~1000 LoC)
  - V8 (~2500 LoC)

Performance overhead (micro)

- RPC latency
  - Average roundtrip: 8 – 9 μs
  - Requires stack copy: < 24%
- Cache coherency overhead
  - 3x – 4x slower if the execution thread and the translation thread is not on the same core

Performance overhead (macro)

- SPEC CINT 2006 (Strata)
  - **1.46%** for pinned schedule
  - 2.05% for free schedule
- JavaScript benchmarks
  - **6.9%** for 32-bit build, **5.65%** for 64-bit build
  - Comparison: NaCl-JIT 79% for 32-bit build

Summary

- Target exploit technique
  - Code inject attack
- Defense principle
  - W^X policy (including temporal)
- Practical criteria
  - Performance overhead: low
  - Adoption difficulty: low

Approaches

- Preventing code injection attacks: SDCG [NDSS'15]
- Preventing data-oriented attacks: **Kenali** [NDSS'16]
- Preventing illegal data-flow: HDFI [SP'16]
- Remaining tasks

Why kernel

- The de-facto trusted computing base (TCB)
- Foundation of upper level security mechanisms (e.g., app sandbox)
- Kernel vulnerabilities are not rare
  - Written in C
  - Heavy optimizations

Why privilege escalation attacks

- One of the most powerful attacks
- Most popular attack against kernel
  - Sandbox bypassing
  - Jailbreak / rooting
- Hard to prevent

Challenge 1: hard to prevent

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

Code Injection Attack
Disable the check

Control-flow hijacking
Bypass the check

Data-oriented attacks
Manipulate the check

Challenge 2: performance

- Protecting all data is not practical
  - Secure Virtual Architecture [SOSP'07]
    - Enforces kernel-wide memory safety
    - Performance overhead: 5.34x ~ 13.10x (LMBench)

Our approach

- Only protects a subset of data that is enough to enforce **access control invariants**
  - Complete mediation
    - **Control-data** → Code Pointer Integrity [OSDI'14]
  - Tamper proof
    - **Non-control-data** used in **security checks** → this work
  - Correctness

Step 1: discover all related data

- Observation: OS kernels have well defined error code for **security checks** (when they fail)
  - POSIX: EPERM, EACCESS, etc.
  - Windows: ERROR_ACCESS_DENIED, etc.
- Solution: leverage this implicit semantic to automatically infer **security checks**
- Benefits
  - Soundness: capable of detecting all security related data (as long as there is no semantic errors)
  - Automated: no manual annotation required

A simple example

Step 1: collect return values

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

A simple example

Step 2: collect conditional branches

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

A simple example

Step 3: collect dependencies

```
1  static int acl_permission_check
2      (struct inode *inode, int mask)
3  {
4    unsigned int mode = inode->i_mode;
5
6    if (likely(uid_eq(current_fsuid(), inode->i_uid)))
7      mode >>= 6;
8    else if (in_group_p(inode->i_gid))
9      mode >>= 3;
10
11   if ((mask & ~mode &
12        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
13     return 0;
14   return -EACCES;
15 }
```

Be complete

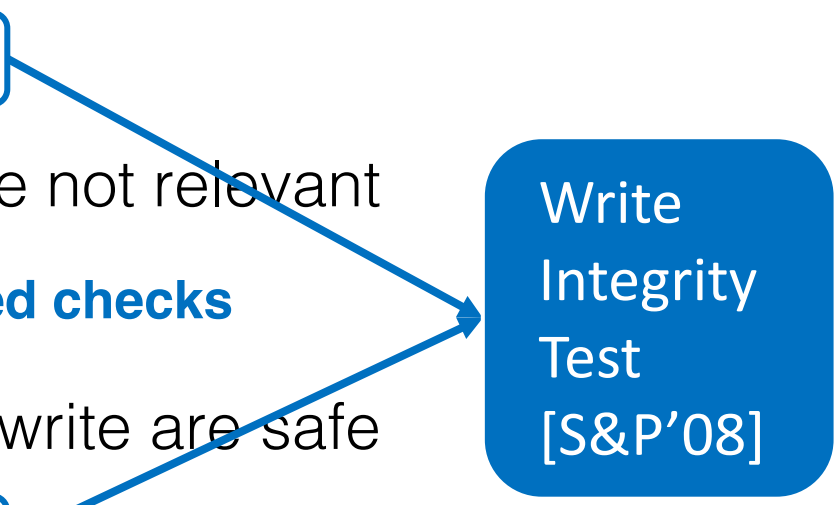
- Collects data- and control-dependencies **transitively**
- Collects sensitive pointers **recursively**

Step 2: protect integrity of the data

- Data-flow integrity [OSDI'06]
  - Runtime data-flow should not deviate from static data-flow graph (similar to control-flow integrity)
    - For example, string should not flow to return address or uid
  - How
    - Check the last writer at every memory read
  - Challenge
    - Performance! (104%)

How to reduce performance overhead

- Observation 1: reads are more frequent than writes
  - **Check write instead of read**
- Observation 2: most writes are not relevant
  - **Use isolation instead of inlined checks**
- Observation 3: most relevant write are safe
  - **Use static analysis to verify**



Two-layered protection

- Layer one: data-flow isolation
  - Prevents unrelated writes from tampering sensitive data
  - Mechanisms: segment (x86-32), access domain (ARM32), WP flag (x86-64), **virtual address space**, hardware virtualization, TrustZone, etc.
- Layer two: WIT
  - Prevents related but unrestricted writes from tampering sensitive data

Additional building blocks

- Shadow objects
  - Lacks fine-grained isolation mechanisms
  - Sensitive data is mixed with non-sensitive data
- Safe stack
  - Certain critical data is no visible at language level, e.g., return address, register spills
  - Access pattern of stack is different
  - Safety is easier to verify

Prototype

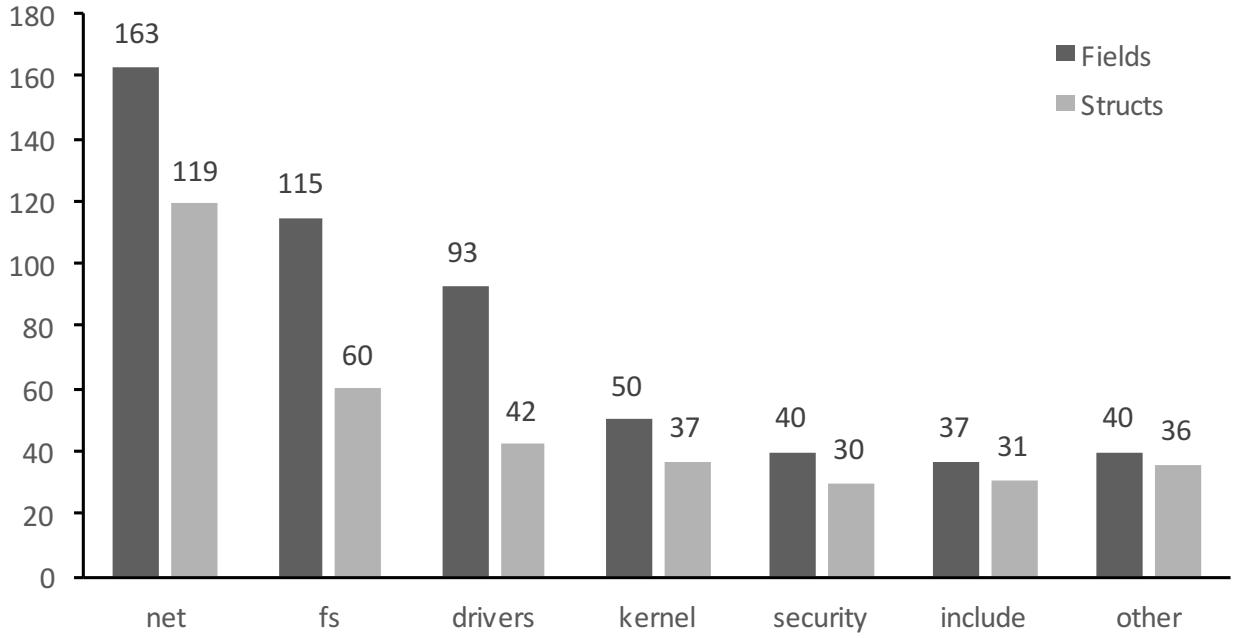
- ARM64 Android
  - For its practical importance and long updating cycle
  - Enough entropy for stack randomization
- Data-flow isolation
  - Heap: virtual address space based, uses ASID to reduce overhead
  - Stack: randomization based
- Shadow objects
  - Modified the SLUB allocator

Implementation

- Kernel
  - Nexus 9 lollipop-release + LLVMLinux patches
  - Our modifications: 1900 LoC
- Static Analysis
  - Framework: KINT [OSDI'12]
  - Point-to analysis: J. Chen's field-sens [GitHub]
    - Context sensitive from KOP [CCS'09]
  - Safe stack: CPI [OSDI'14]
  - Our analysis + modifications: 4400 LoC
  - Instrumentation: 500 LoC

How many sensitive data structures

- Control data: 3699 fields (783 structs), 1490 global objects
- Non-control data: **1731** fields (**855** structs), **279** global objects
  - False positives: 491 fields (221 structs) / 61 fields (25 structs)



How secure is our approach

- Inference
  - Sound → no false negatives
  - Catch: no semantic errors
- Data-flow (point-to) analysis
  - Sound but not complete → over permissive
  - Improve the accuracy with context and field sensitivity
- Against existing attacks
  - All prevented

Performance impact

- Write operations
  - 26645 (4.30%) allowed, 2 checked
- Context switch
  - 1700 cycles
- Benchmarks
  - LMBench (syscalls): 1.42x ~ 3.13x (0% for null syscall)
  - Android benchmarks: 7% ~ 15%

Summary

- Target exploit technique
  - Data-oriented attacks (for kernel privilege escalation)
- Defense principles
  - Access control invariants
  - DFI
- Practical criteria
  - Performance overhead: moderate
  - Adoption difficulty: low

Proposals

- Preventing code injection attacks: SDCG [NDSS'15]
- Preventing data-oriented attacks: Kenali [NDSS'16]
- Improving security and performance: **HDFI** [SP'16]

Limitations from previous projects

- Data isolation
  - Lacks efficient isolation in 64-bit world
    - Performance overhead for switching between virtual address spaces is high
  - Lacks of fine-grained isolation
    - Shadow objects are awkward

Opportunities

- DFI is a great prevention technique
  - Capable of preventing both illegal read and write
  - Do not need to track allocation
- Proposing new hardware features is more feasible
  - Open sourced hardware: RISC-V
  - Chips are cheap and customization is popular
  - Even Intel is more willing to adopt more protection features (SGX, MPX, execute-only page)

Design goals

- Enabling selective memory safety
  - Approach: efficient isolation for critical data
- Practical
  - Commercial-ready platform
  - Minimized hardware changes
  - Low performance overhead
- Flexible
  - Capable of support different security model/mechanisms

HDFI Architecture

- ISA extension
  - Three new instructions to enable DFI-style checks: `sdset1, ldchk0, ldchk1`
- Cache extension
  - Extra bits in the cache line for storing the tag
- Memory Tagger
  - Emulating tagged memory without physically extending the main memory

Optimizations

- Memory Tagger introduces additional performance overhead (memory accesses)
  - Naive implementation: 2x, 1 for data, 1 for tag
- Three optimization techniques
  - Tag cache
  - Tag valid bits (TVB)
  - Meta tag table (MTT)

Security applications (1)

- Shadow stack
  - Approach: return address should always have tag 1
  - Benefits: supports context saving/restoring, deep recursion, modified return address, kernel stack

```
1  main:
2    add      sp,sp,-32
3   ⋆sdset1   ra,24(sp)
4    ld       a1,8(a1)        ; argv[1]
5    mv       a0,sp           ; char buff[16]
6    call     strcpy          ; strcpy(buff, argv[1])
7    li       a0,0
8   ⋆ldchk1   ra,24(sp)
9    add      sp,sp,32
10   jr       ra              ; return
```

Security application (2)

- Standard library enhancement
  - Heap metadata protection
  - setjmp/longjmp
  - GOT protection
  - Exit handler protection

Security application (3)

- VTable pointer (vfptr) protection
  - Approach: vfptr should always have tag 1
    - Only allow constructors to set tag 1
    - Check tagging at every virtual function call

Security applications (4)

- Code pointer separation (CPS)
  - Isolation code pointers from malicious access (from CPI)
  - Our implementation: code pointers should always have tag 1
    - Use `sdset1` to store code pointers
    - Use `ldchk1` to load code pointers

Security application (5)

- Kernel protection - Kenali
  - Approach: critical data should always have tag 1
    - Allow writing instructions that can write to sensitive data to set tag to 1
    - Load sensitive data with **`ldchk1`**

Security applications (6)

- Information leak prevention
  - Approach: mark sensitive data with tag 1
    - Use ldchk0 to construct/scan output buffer
  - A demonstration: Heartbleed attack
    - Store crypto keys with **`sdset1`**
    - Use **`ldchk0`** to sanitize buffer to be written to the network

Implementations

- Hardware
  - RISCV RocketCore: 2198 LoC
- Software
  - Assembler gas: 16 LoC
  - Linux kernel: 60 LoC

Effectiveness of optimizations

- Memory bandwidth and latency

Benchmark	Baseline	Tagger		TVB		MTT		TVB+MTT	
L1 hit	40ns	40ns	(0%)	40ns	(0%)	40ns	(0%)	40ns	(0%)
L1 miss	760ns	870ns	(14.47%)	800ns	(5.26%)	870ns	(14.47%)	800ns	(5.26%)
Copy	1081MB/s	939MB/s	(13.14%)	1033MB/s	(4.44%)	953MB/s	(11.84%)	1035MB/s	(4.26%)
Scale	857MB/s	766MB/s	(10.62%)	816MB/s	(4.79%)	776MB/s	(9.45%)	817MB/s	(4.67%)
Add	1671MB/s	1598MB/s	(4.37%)	1650MB/s	(1.26%)	1602MB/s	(4.13%)	1651MB/s	(1.2%)
Triad	818MB/s	739MB/s	(9.66%)	802MB/s	(1.96%)	764MB/s	(8.8%)	803MB/s	(1.83%)

- SPEC CINT2000

Benchmark	Baseline	Tagger		TVB		MTT		TVB+MTT	
164.gzip	963s	1118s	(16.09%)	984s	(2.18%)	1029s	(6.85%)	981s	(1.87%)
175.vpr	14404s	18649s	(29.51%)	14869s	(3.26%)	15513s	(7.71%)	14610s	(1.43%)
181.mcf	8397s	11495s	(36.89%)	8656s	(3.08%)	9544s	(13.66%)	8388s	(−0.11%)
197.parser	21537s	25005s	(16.11%)	22025s	(2.27%)	23177s	(7.61%)	21866s	(1.53%)
254.gap	4224s	4739s	(12.19%)	4268s	(1.04%)	4500s	(6.53%)	4254s	(0.71%)
256.bzip2	716s	820s	(14.52%)	735s	(2.65%)	742s	(3.63%)	722s	(0.84%)
300.twolf	22240s	28177s	(26.71%)	22896s	(2.97%)	23883s	(7.37%)	22323s	(0.36%)

Security experiments

- With synthesized attacks

Mechanism	Attacks	Result
Shadow stack	RIPE	✓
Heap metadata protection	Heap exploit	✓
VTable protection	VTable hijacking	✓
Code pointer separation (CPS)	RIPE	✓
Code pointer separation (CPS)	Format string exploit	✓
Kernel protection	Privilege escalation	✓
Private key leak prevention	Heartbleed	✓

Impacts on security solutions (1)

- Security guarantee
  - Stronger than randomization-based isolation (shadow stack, CPS)
  - Stronger than ad-hoc encryption-based (ptmalloc, setjmp/longjmp)
  - Stronger than simple integrity-check-based protection (vfptr)

Impacts on security solutions (2)

- Efficiency
  - Faster than masking and shadow address space based
    - Average RPC latency in SDCG is 8-9 μs
    - Context-switch costs 1700 cycles in Kenali
  - Faster than shadow object based
    - Hash table lookup in CPS is 97.1x slower than single memory operation
    - HDFI is 1.6% slower

Impacts on security solutions (3)

- Efficiency
  - Benchmarks

Benchmark	GCC	Shadow Stack	Clang[1]	CPS+SS[1]
164.gzip	981s	992s (1.12%)	1734s	1776s (2.42%)
181.mcf	8388s	8536s (1.76%)	11014s	11403s (3.54%)
254.gap	4254s	4396s (3.34%)	20783s	23526s (13.23%)
256.bzip2	722s	744s (3.05%)	1454s	1521s (4.61%)

Benchmark	Baseline	Kernel Stack Protection
null syscall	8.91μs	8.934μs (0.27%)
open/close	160.6μs	168.7μs (5.04%)
select	285.6μs	287.5μs (0.67%)
signal install	19.3μs	21.5μs (11.4%)
signal catch	99.8μs	105.6μs (5.81%)
pipe	273.6μs	306.6μs (12.06%)
fork+exit	5892μs	6308μs (7.06%)
fork+execv	6510μs	6972μs (7.1%)
page fault	50.0μs	52.6μs (5.2%)
mmap	800μs	880μs (10%)

Impacts on security solutions (4)

- Simplicity of implemented solutions

Shadow Stack	C++ (LLVM 3.3)	4
VTable Protection	C++ (LLVM 3.3)	40
CPS	C++ (LLVM 3.3)	41
Kernel Protection	C (Linux 3.14.41)	70
Library Protection	C (glibc 2.22)	10
Heartbleed Prevention	C (OpenSSL 1.0.1a)	2

- Difficulty of use
  - Source code modification (C library enhancement)
  - Compiler-based (shadow stack, CPS, vptr, Kenali)
  - Binary rewriting

Security analysis

- Attack surface
  - Inaccuracy of data-flow analysis
  - Deputy attacks
- Best practice
  - CFI is necessary (e.g., CPS + shadow stack)
  - Recursive protection of pointers
  - Guarantee the trustworthiness of the written value
  - Use runtime memory safety technique to compensate inaccuracy of static analysis

Summary

- Target exploit technique
  - Malicious write (control-flow hijacking and data-oriented attacks)
  - Malicious read (Heartbleed)
- Defense principles
  - Data-flow integrity
- Practical criteria
  - Performance overhead: low
  - Adoption difficulty: low

Thesis contributions (1)

- New threats highlighting
  - Code cache injection attacks
  - Data-oriented attacks
- New software design
  - SDCG: a new system design that resolves the confliction between W^X policy and dynamic code injection and blocks all code injection attacks

Thesis contributions (2)

- New program analysis technique
  - InferDists: a new technique to automatically infer data that is critical to kernel privilege escalation attacks
- New isolation technique
  - ProtectDists: a two-layer protection scheme to enforce efficient and fine-grained protection over selective memory content

Thesis contributions (3)

- New hardware design
  - HDFI: a new hardware isolation mechanism that is easy to use, imposes low performance overhead, and allows us to create simpler and more secure solutions
- Open source
  - For better adoption in real world
    - Done (SDCG, ProtectDists)
    - WIP (InferDists)
    - TODO (HDFI)

Future work

- Uninitialized data access
  - Information leak, DoS, control-flow hijacking, arbitrary read/write, etc.
- Information leak
  - Practical generic information leak prevention is still an open problem
- Memory safety
  - Performance and compatibility

Conclusion

- Research problem
  - Principled and Practical defense techniques against memory-corruption-based exploits
- Contributions
  - Three exploit prevention techniques that advanced the state-of-art
- Future work
  - Defense against the rest two exploit techniques

Thank you!