上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Security: Attack and Defense

From system's perspective

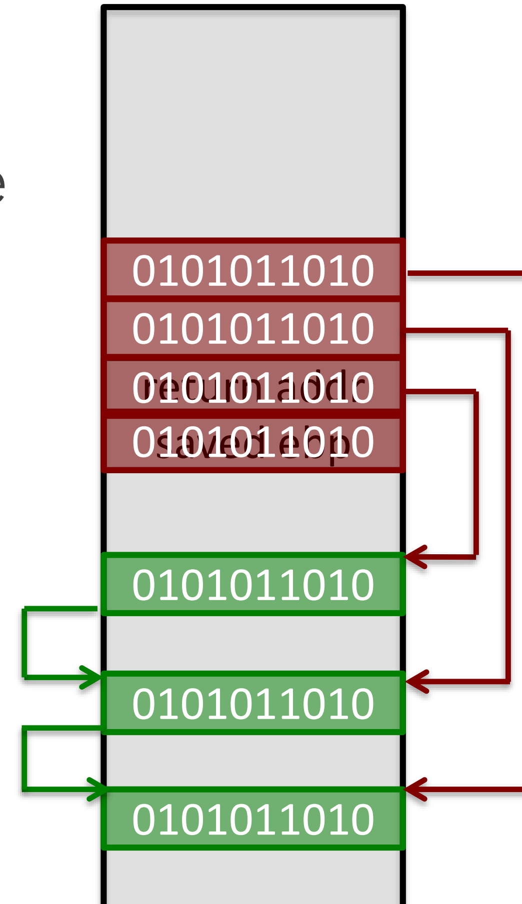
Yubin Xia

return-oriented programming

CONTROL-FLOW ATTACK

Review: ROP Attacks

- ROP: Return-oriented Programming
  - Find code gadgets in existed code base
    - Usually 1-3 instructions, ends with 'ret'
    - In libc and application, intended and unintended
  - Push address of gadgets on the stack
  - Leverage 'ret' at the end of gadget to connect each code gadgets
  - **No code injection**

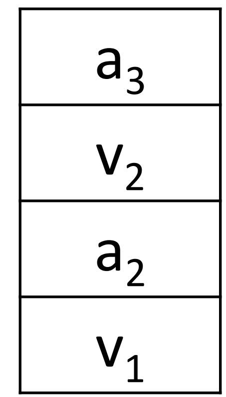


Code Execution – The ROP Way

`Mem[v2] = v1`

Desired Logic

```
mov %eax v1;
mov %ebx v2;
mov [%ebx], %eax
```



Stack

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`

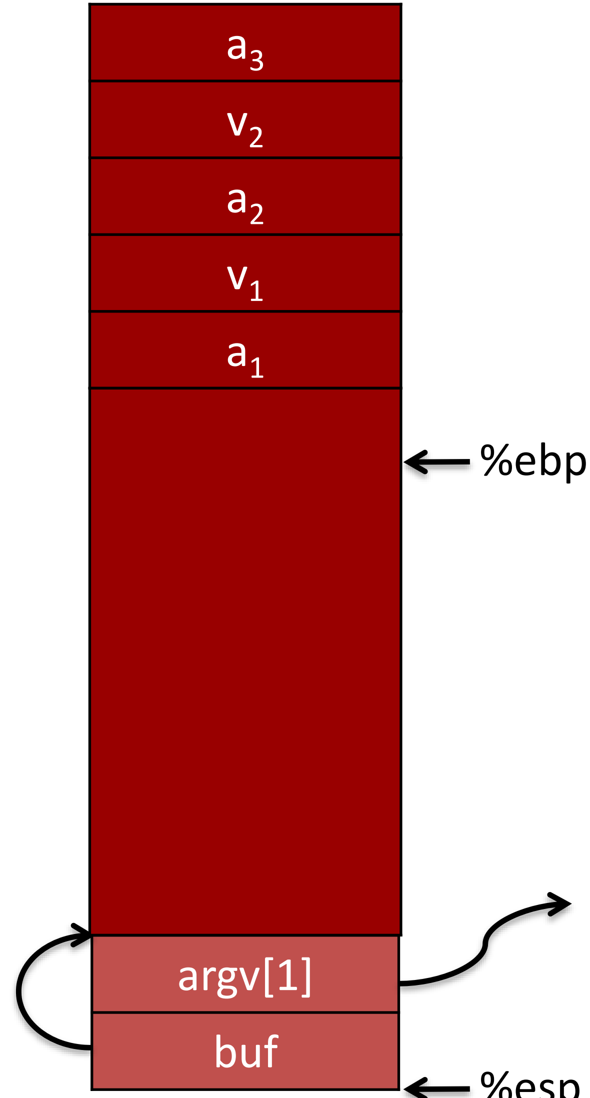
Code Execution – ROP Way

Mem[v2] = v1

Desired *Shellcode*

```
a₁: pop eax; ret
a₂: pop ebx; ret
a₃: mov [ebx], eax
```

Desired store executed!



Defense

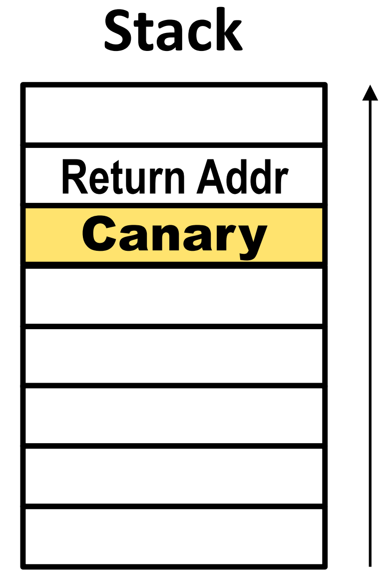
- **Hide** the binary file
  - No way to get any gadget
- **ASLR** to randomize the code position
  - Short for "Address Space Layout Randomization"
  - Harder to find gadgets
- **Canary** to protect the stack
  - Try to detect stack overflow (e.g., overflow return address)

ASLR

- Change the layout of memory space
- Every time when a process is created
  - Question: how about create by fork?
- User-level as well as kernel-level

Canary

- Embed "canaries" in stack frames and verify their integrity prior to function return
  - Canary is just a random number
  - Check canary before return, alert if not equal



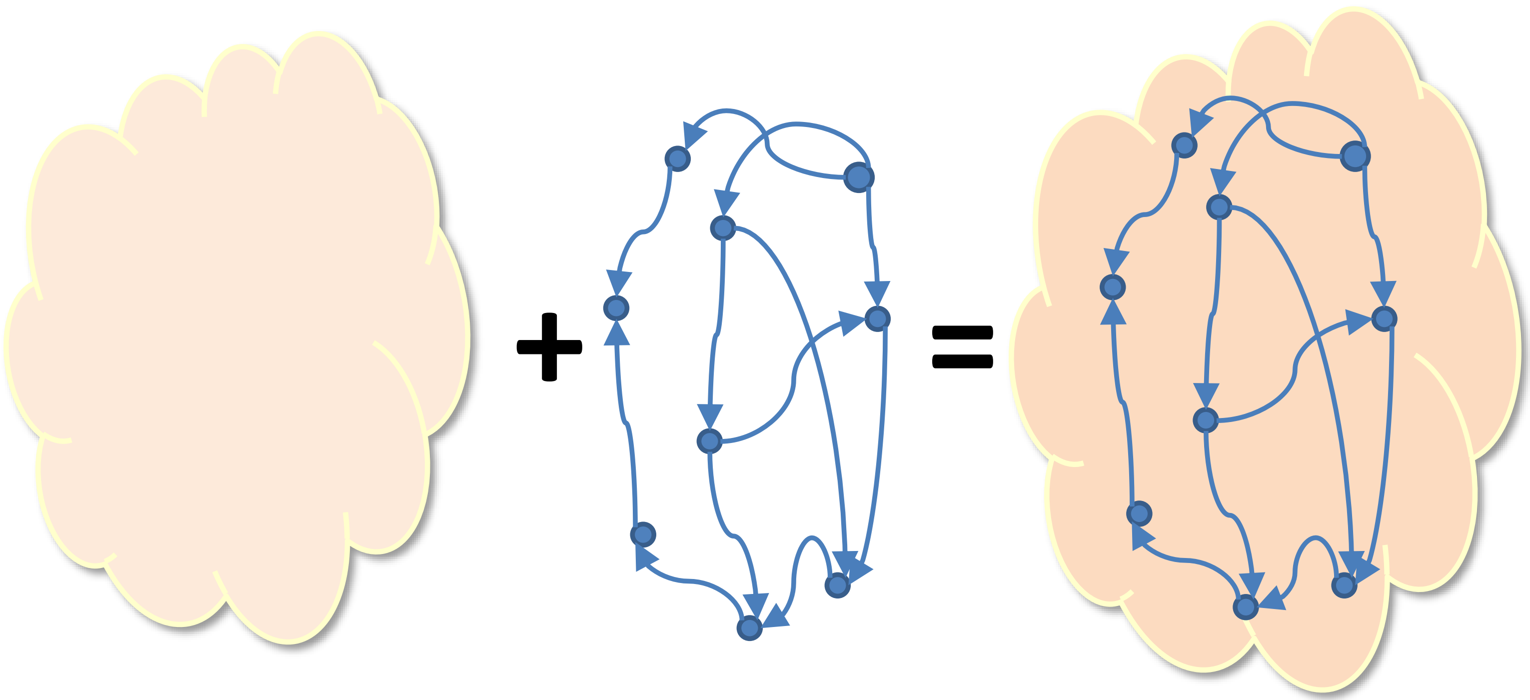
- StackGuard implemented as a GCC patch
  - Program must be recompiled
  - Performance overhead: 8% for Apache

CFI: CONTROL FLOW INTEGRITY

CFI: Control-Flow Integrity

- Main idea: pre-determine **control flow graph** (CFG) of an application
  - Static analysis of source code
  - Static binary analysis ← **CFI**
  - Execution profiling
  - Explicit specification of security policy
- Execution must follow the pre-determined control flow graph

CFI Idea



Executable Control-Flow Graph Self-checking program

CFI: Binary Instrumentation

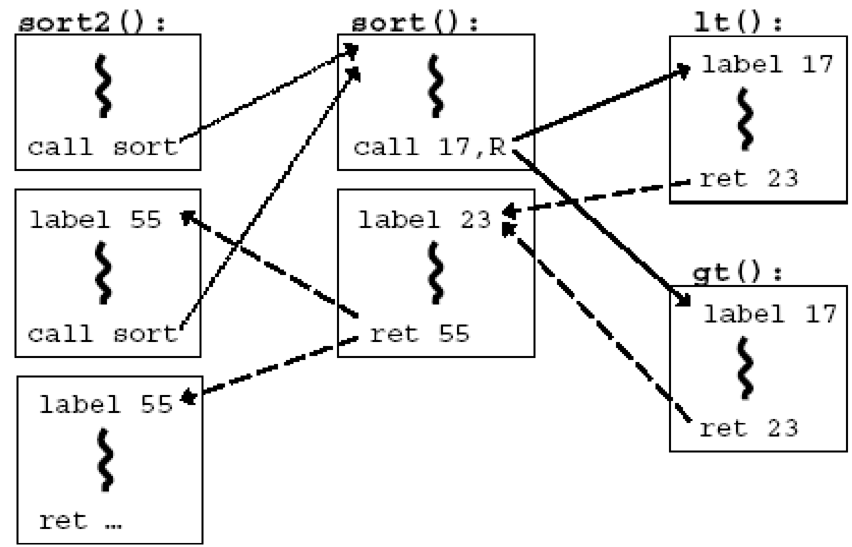
- Use *binary rewriting* to instrument code with runtime checks
- Inserted checks ensure that the execution always stays within the statically determined CFG
- Whenever an instruction transfers control, destination must be valid according to the CFG
- **Goal**: Secure even if the attacker has complete control over the thread's address space

CFG Example

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



CFI: Control Flow Enforcement

- For each control transfer, determine statically its possible destination(s)
- Insert a unique bit pattern at every destination
  - Two destinations are equivalent if CFG contains edges to each from the same source
    - This is imprecise (why?)
  - Use same bit pattern for equivalent destinations

CFI: Example of Instrumentation

Original

```
jmp   ecx
```

```
mov   eax, [esp+4]    ; dst
```

Patched

```
cmp   [ecx], 12345678h
jne   error_label
lea   ecx, [ecx+4]
jmp   ecx
```

```
; data 12345678h      ; ID
mov   eax, [esp+4]    ; dst
```

CFI: Example of Instrumentation

```
mov   eax, 12345677h     ; load ID-1          3E 0F 18 05    prefetchnta        ; label
inc   eax                ; add 1 for ID       78 56 34 12        [12345678h]    ;     ID
cmp   [ecx+4], eax       ; compare w/dst      8B 44 24 04    mov   eax, [esp+4] ; dst
jne   error_label        ; if != fail         ...
jmp   ecx                ; jump to label
```

Prefetchnta: prefetch memory to cache. Become a *nop* if not available.

Improving CFI Precision

- Suppose a call from A goes to C, and a call from B goes to either C, or D (when can this happen?)
  - CFI will use the same tag for C and D, but this allows an "invalid" call from A to D
  - Possible solution: duplicate code or inline
  - Possible solution: multiple tags

Improving CFI Precision

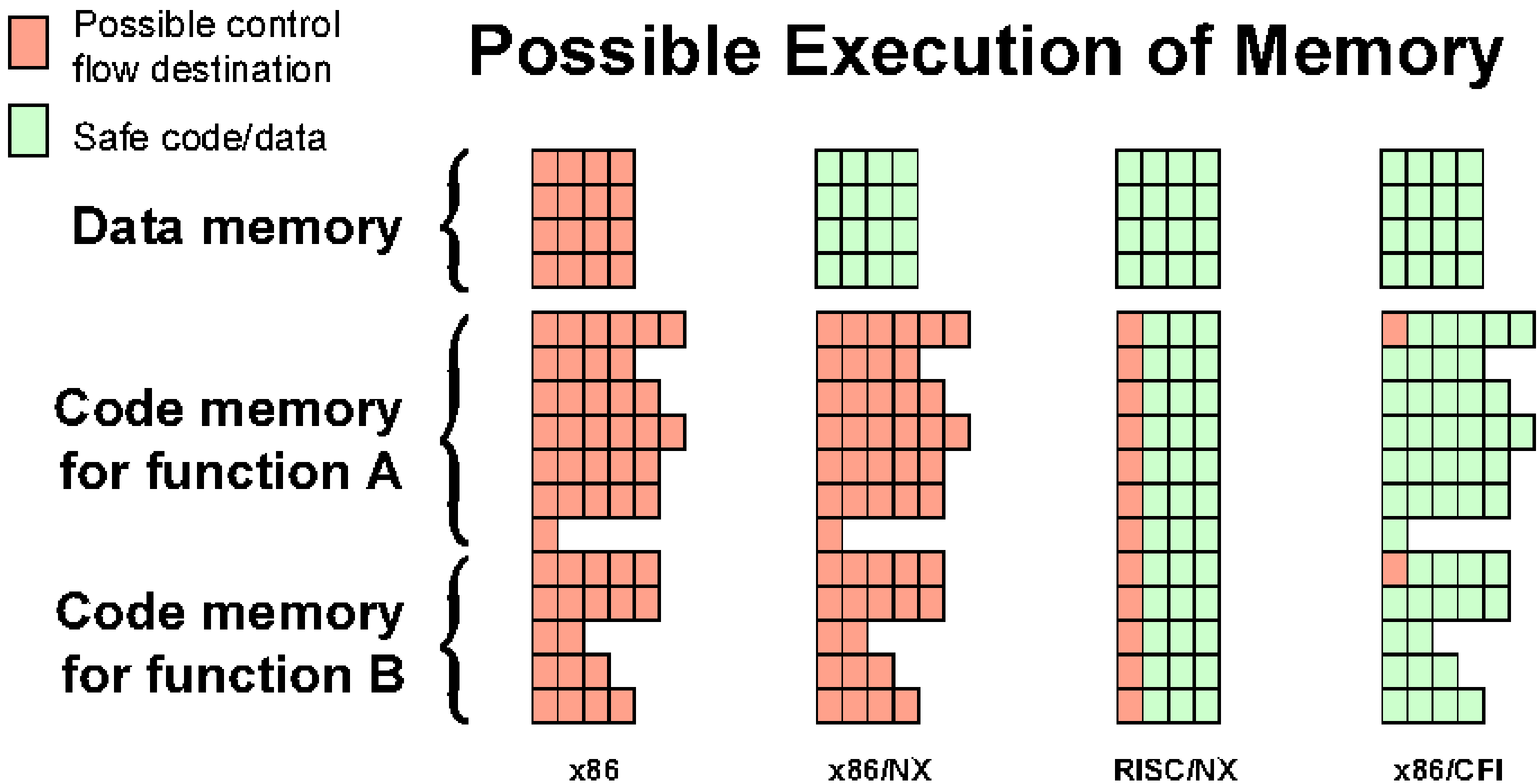
- Function F is called first from A, then from B; what's a valid destination for its return?
  - CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
  - Solution: shadow call stack
    - Maintain another stack, just for return address
    - Intel CET to the rescue (not available yet)

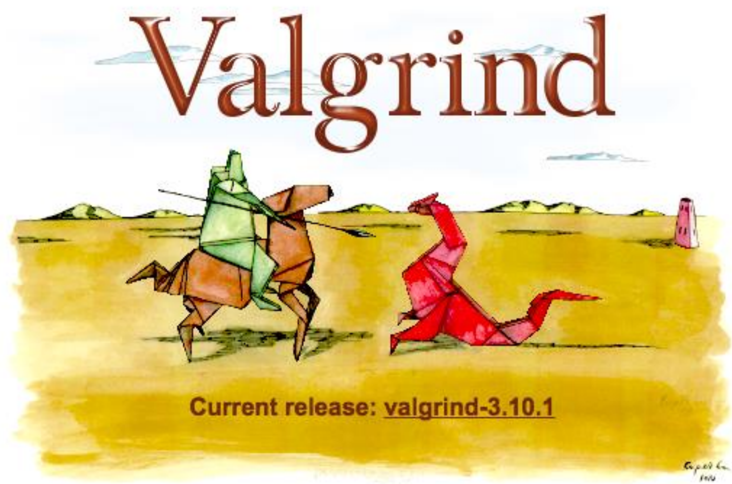
CFI: Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge
- Does <u>not</u> protect against attacks that do not violate the program's original CFG
  - Incorrect arguments to system calls
  - Substitution of file names
  - Other data-only attacks

Possible Execution of Memory

[Erlingsson]





Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, NDSS'05

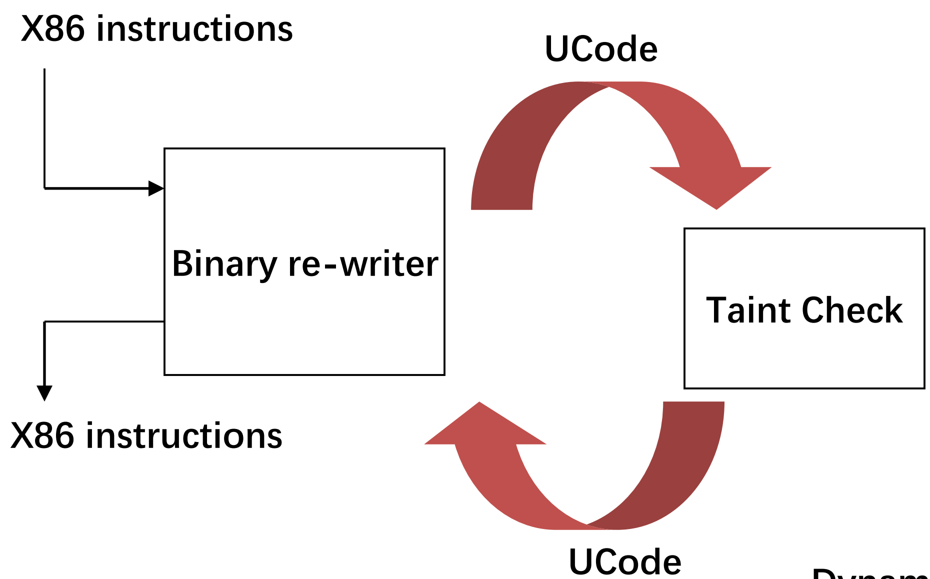
TAINTCHECK

TaintCheck: Basic Ideas

1. Program execution normally derived from trusted sources, not attacker input
2. Mark all input data to the computer as "tainted" (e.g., network, stdin, etc.)
3. Monitor program execution and track how tainted data propagates (follow bytes, arithmetic operations, jump addresses, etc.)
4. Detect when tainted data is used in dangerous ways

Add Taint Checking code

- TaintCheck first runs the code through an emulation environment (Valgrind) and adds instructions to monitor tainted memory



Dynamic taint analysis

TaintCheck Detection Modules

- TaintSeed: Mark untrusted data as tainted
- TaintTracker: Track each instruction, determine if result is tainted
- TaintAssert: Check is tainted data is used dangerously

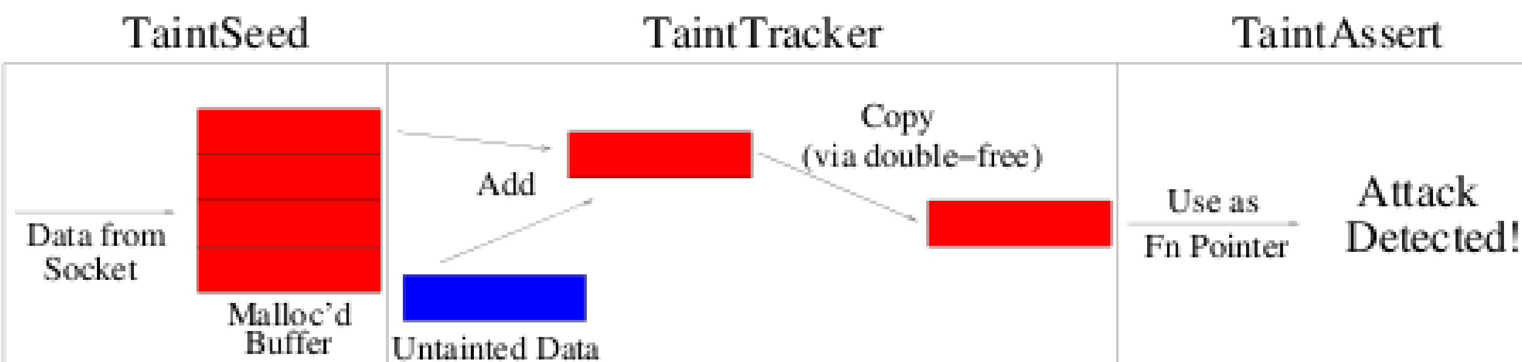


Figure 1. TaintCheck detection of an attack. (Exploit Analyzer not shown).

TaintSeed

- Marks any data from untrusted sources as "tainted"
  - Each byte of memory has a four-byte shadow memory (!) that stores a pointer to a Taint data structure if that location is tainted
  - Else store a NULL pointer

TaintTracker

- Tracks each instruction that manipulates data in order to determine whether the result is tainted
  - When the result of an instruction is tainted by one of the operands, TaintTracker sets the shadow memory of the result to point to the same Taint data structure as the tainted operand

TaintCheck Detection Modules

- TaintAssert
  - Jump addresses: function pointers or offsets
  - Format strings: is tainted data used as a format string arg?
  - System call arguments
  - Application or library customized checks

When does TaintCheck Give a False Positive?

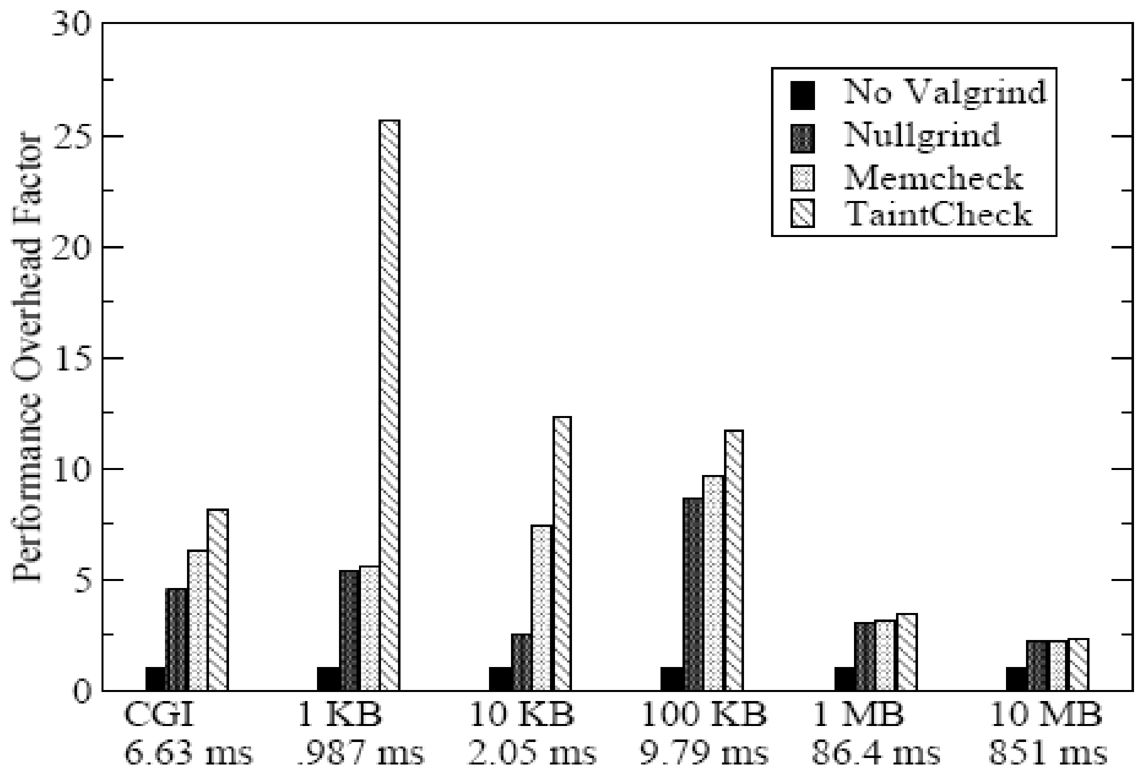
- TaintCheck detects that tainted data is being used in an illegitimate way even when there is no attack taking place. Possibilities:
  - There are vulnerabilities in the program and need to be fixed, or
  - The program performs sanity checks before using the data

Performance Evaluation – CPU Bound Process

- Hardware: 2.00 GHz Pentium 4, 512 MB RAM, RedHat 8.0
- Application: bzip2(15mb)
  - Normal runtime 8.2s
  - Valgrind nullgrind skin runtime: 25.6s (3.1x)
  - Memcheck runtime: 109s (13.3x)
  - TaintCheck runtime: 305s (37.2x)

Performance Overhead of Apache

- A more representative case, network and I/O



Only incur 14% overhead!

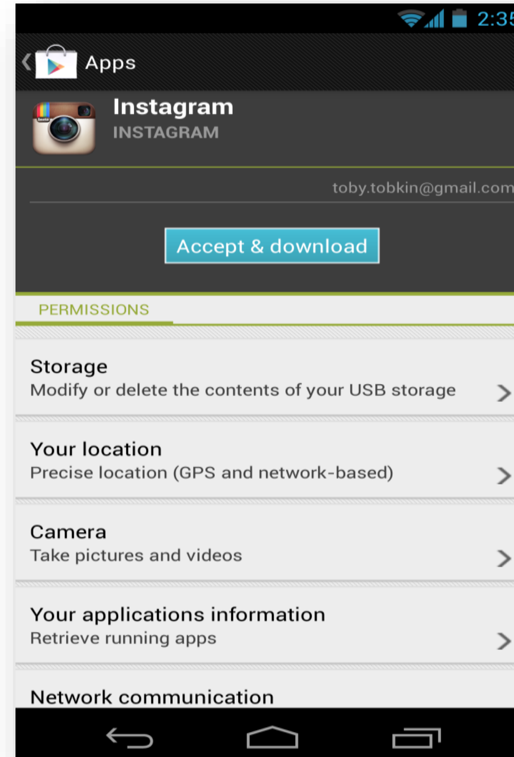
TAINTDROID

Android Background Information

- Applications written in Java
  - Java Native Interface (JNI)
  - Compiled into Dalvik Executable (DEX) byte-code format
  - Executes within Dalvik VM interpreter instance
    - Register-based as opposed to stack-based
    - Runs isolated on the platform
      - Has unique UNIX user identities
  - Components communicate via binder IPC mechanism

Motivation

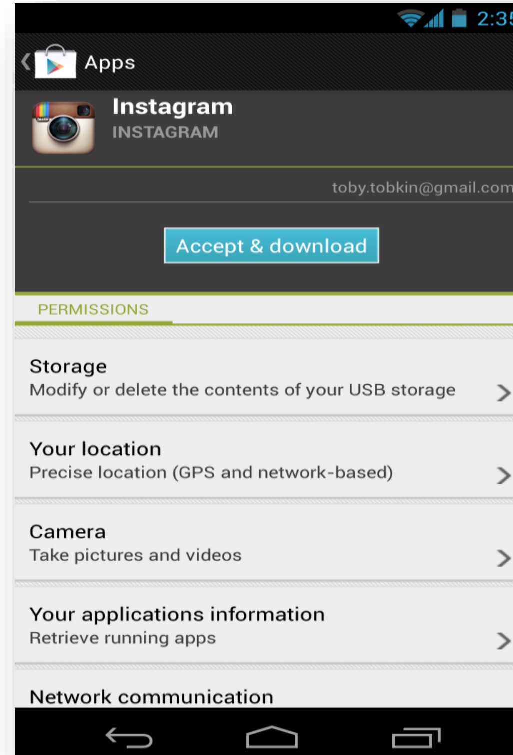
- Historical problem with computer software: privacy violations
  - Unwitting users
- Problem exacerbated by smartphones
  - Almost ubiquitously store private information
  - Large array of sensors
  - Monetization pressures to detriment of user privacy
  - Cited by paper: [12, 19, 35]



Android's coarse-grained privacy control

Motivation

- Current privacy control methods arguably inadequate
- Idea:
  - Can't change the current system without repercussions
  - Instead, create a method to audit untrusted applications
- Execution:
  - Must be able to detect potential misuses of private information,
  - And be fast enough to be usable



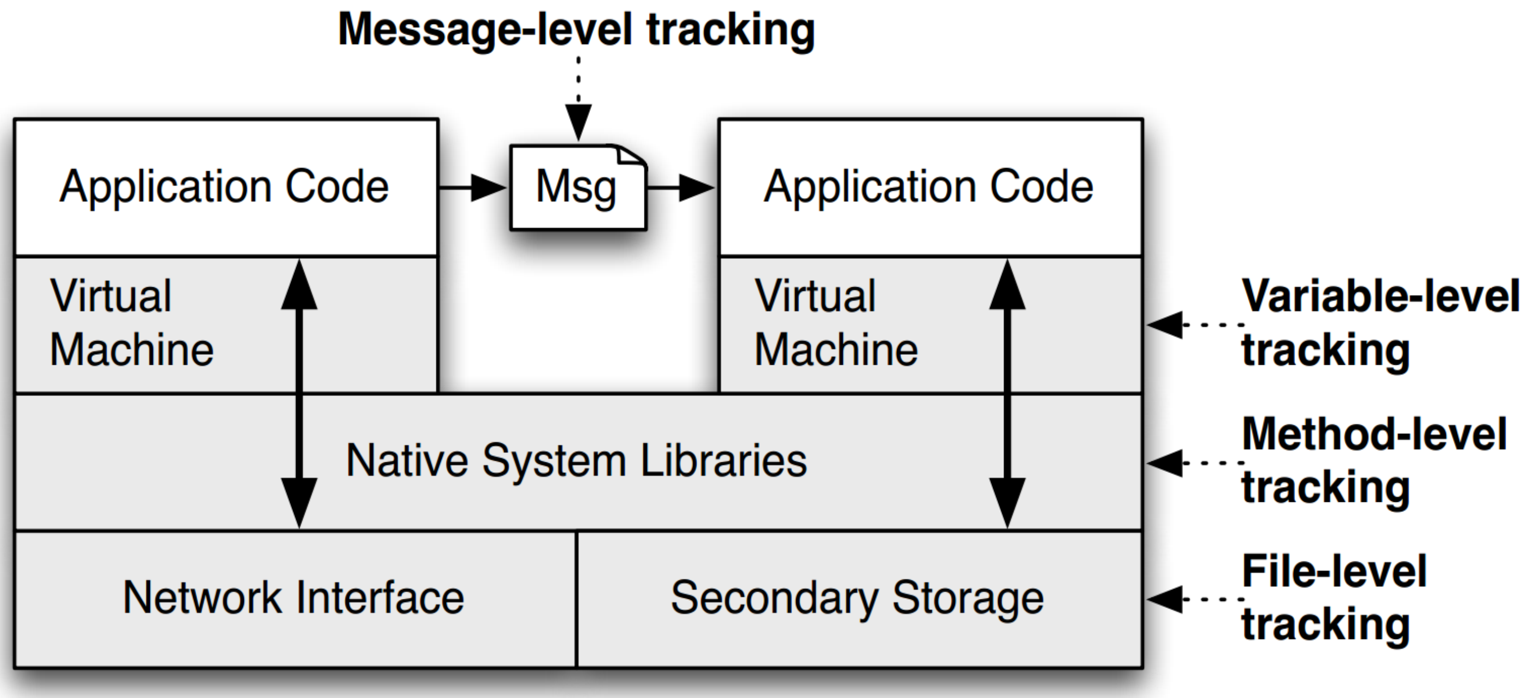
Android's coarse-grained privacy control

Taint Sources & Sink

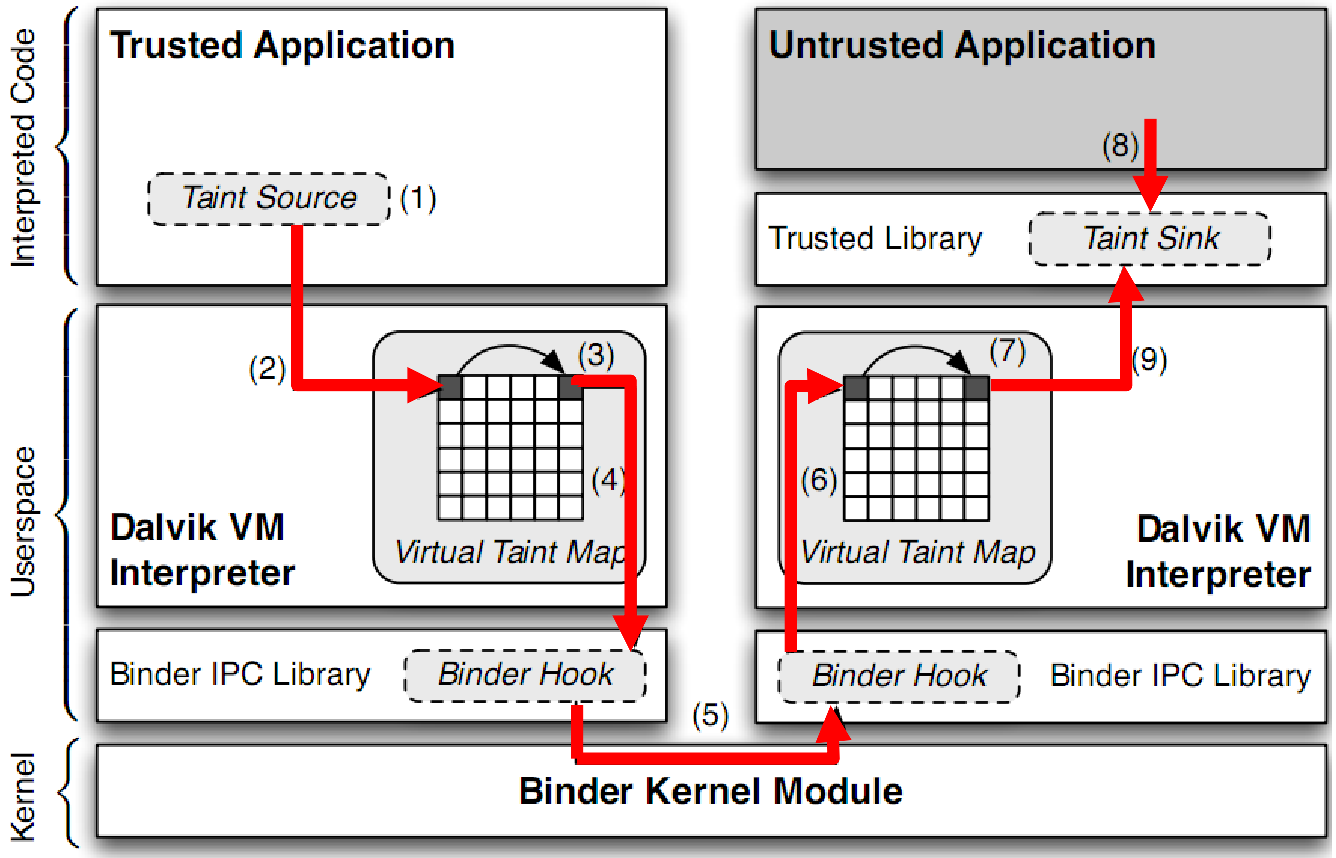
- Low-bandwidth sensors
  - Location, accelerometer
- High-bandwidth sensors
  - Microphone, camera
- Information database
  - Address book, SMS, phone call log, etc.

- Device identifiers
  - IMEI
- Network taint sink
  - WiFi
  - 3G
  - SMS

TaintDroid Architecture

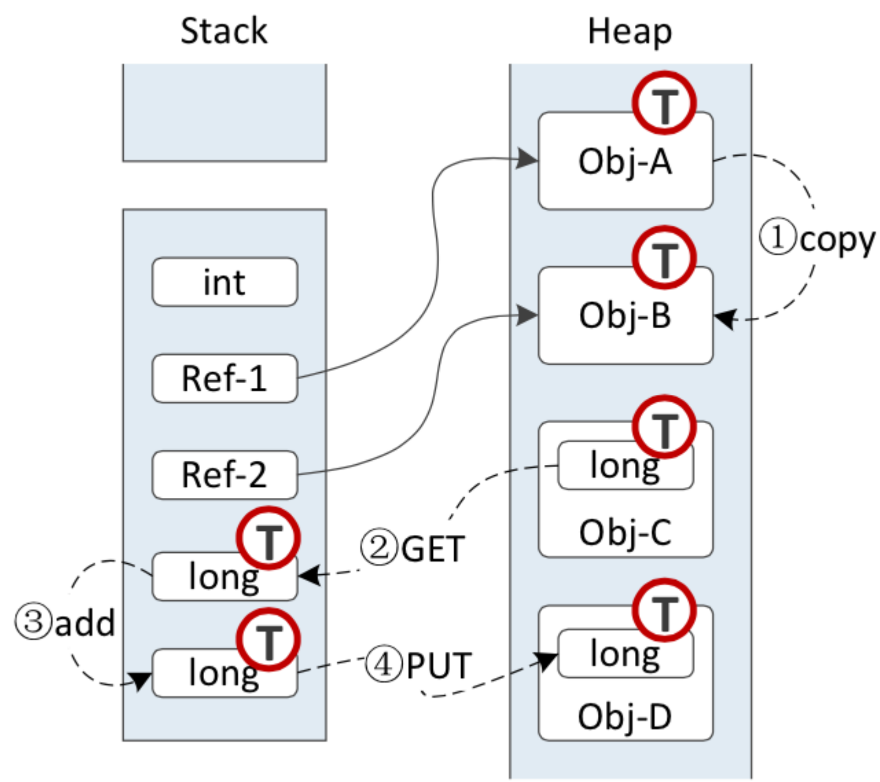


TaintDroid Architecture



Four Types of Tainting

- From heap to heap
- From heap to stack
- From stack to stack
- From stack to heap



Heap to Stack

- A property of Java: all operations must be done through the stack
  - E.g., "A.a = B.b" will be compiled to:
    - GET "A.a" to the stack
    - PUT the value to "B.b"

Heap to Stack

- String orig is tainted
  - On heap: orig and dst
  - On stack: tmp
- Heap to stack
  - The tainted object generates a primitive type, whose taint tag will propagated to the stack variable
- Stack to heap
  - The taint tag will also propagate to the heap

```
// 'orig' is tainted
String orig = new String("I'm secret");
String dst = "";
for(int i=0;i<orig.length();i++){
  Char tmp = orig.charAt(i);
  dst = dst + tmp;
}
```

TaintDroid Design

- Variable Level Tracking
  - Primary Tracking Method
    - Propagation Rules are key
  - Only save one tag for an array, e.g., String
    - Better performance, but may cause false positive

Taint Propagation

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

TaintDroid Design

- Internal VM methods
  - Relatively small number of such methods (5 in 185)
  - Manually inspected and patched to propagate
- JNI methods
  - Patch the call bridge to provide
  - When a JNI returns, consults a method profile table for tag propagation updates

TaintDroid Design

- Message Level Tracking
  - Binder IPC mechanism (centralized location)
  - Entire message is labeled instead of variables
    - Trade off between performance and accuracy
- File Level Tracking
  - Taint label stored in the extended file attribute
  - Entire file is labeled
    - Again, trade off between performance and accuracy
  - Taint label propagated to variables on access

TaintDroid Design

- Identifying Privacy Data
- Each source must be studied carefully
  - False Positives if wrong source is tainted
  - False Negative if a source is missed

TaintDroid Design: Sources

- Low-bandwidth Sensors
  - Variety of privacy data obtain from these sensors
    - Ex: Location, and Accelerometer
  - Changes frequently, and used by many applications
  - Most smartphone OS have some type of manager to multiplex this information
  - Android privacy hook (labels) in *LocationManager* and *SensorManager* Applications

TaintDroid Design: Sources

- High-bandwidth Sensors
  - Include privacy information such Microphones and Cameras
  - Returns a large amount of data and is only used by one application at time
    - Sensor information is propagated through large data buffers and/or files
    - Hooks placed in these data buffers and files

TaintDroid Design: Sources

- Information Databases
  - Contact and SMS (Text) Messages are often stored in file based databases
  - Taint Tags place on these files
- Device Identifiers
  - Information that Identifies the phone or user
    - Ex SIM card identifiers
  - Some are access through well defined APIs
  - APIs are hooked

TaintDroid Design: Sink

- Network Taint Sink
  - When data is transmitted out of the network interface
  - Taint is place in VM Interpreter
  - Label read when the socket library is invoked
- Summary
  - Privacy data is labeled (tainted)
  - Propagates up and down the architecture.
  - These labels are read and thus…

Contributions

- TaintDroid produced useful results for every application tested
- A useful privacy analysis tool was implemented
  - produced no false positives in experiments completed
  - high performance in design
  - also, released to public

Experimental Results

Observed Behavior (# of apps)	Details
Phone Information to Content Servers (2)	2 apps sent out the phone number IMSI, and ICC-ID along with geo-coordinates to the app's content server
Device ID to Content Servers (7)*	2 social, 1 shopping, 1 reference and 3 other apps transmitted the IMEI number to the app's content server
Location to Advertisement Servers (15)	5 apps sent geo-coordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com) and 4 apps sent location to data.flurry.com

Experimental Results

- TaintDroid produced no false positives on the application set tested
- 1/2 of applications shared location data with advertising servers
- ~1/3 expose device ID
- Authors claim no perceived latency in using interactive applications
- TaintDroid shown to be qualitatively useful

Weaknesses

- Mentioned by Enck et al.:
  - TaintDroid can be circumvented by implicit information flow
  - TaintDroid cannot tell if tainted information re-enters the phone after leaving
- Interactive application latency was reported anecdotally, but could have been measured more formally

Evading Tainting

```
if (x == 0)  y = 0;
else if (x == 1) y = 1;
...
 else if (x == 255) y = 255;
```

- Windows 2000 kernel illustrates this problem when translating keyboard scancodes into unicode

Evading Tainting

- Using side-channel attacks
  - Any global resource that can be used to transfer 0/1
  - Timing attack
  - Usually transfer one bit at a time
    - But the sensitive data is also small at most time…

Weaknesses

- Needs a bigger sample size for statistics
- Taint tags are added manually
- Taint tags memory location is predictable

Improvement

- Use the tool to test a larger sample size
- Develop component to automate the tagging process
- Randomized the taint tag memory location, or prevent it from being modified

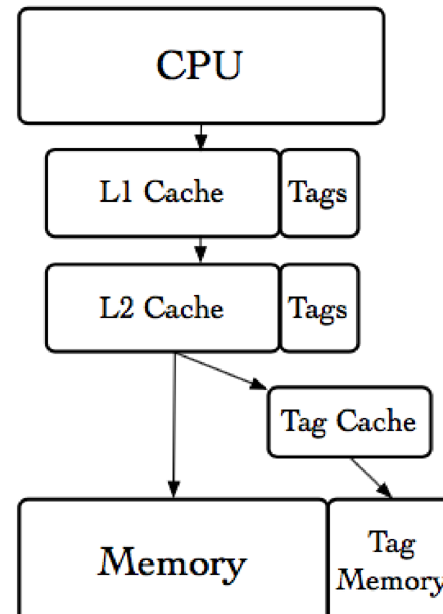
TAXI

Goals

- Defend code reuse attacks
  - With tagged memory
  - A small set of hardware modifications

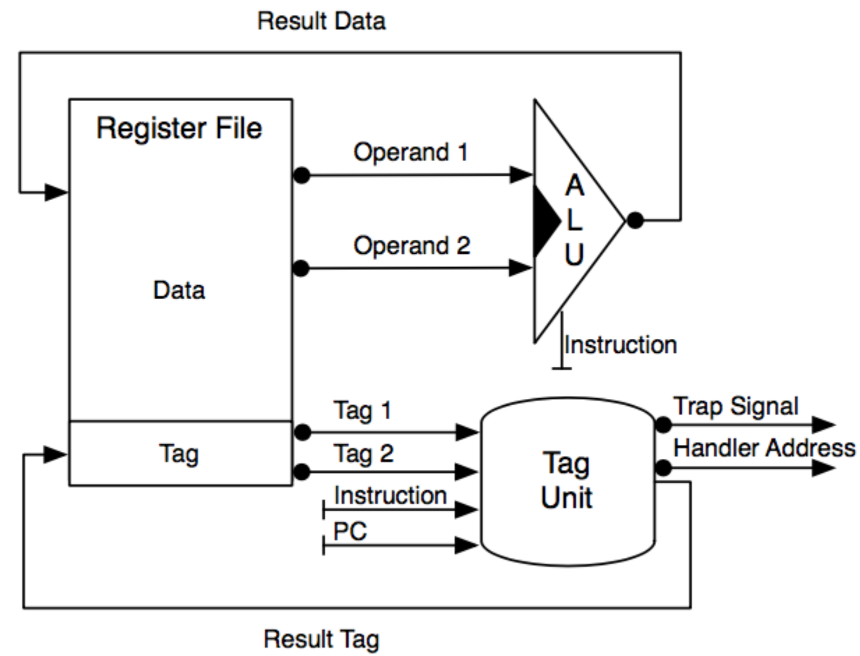
Tagged Memory of Taxi

- Each **words** has a tag
  - 64-bit word + 8-bit tag
- Tags are located separately from data



Tag Propagation

- Tag processing unit (TPU)
  - Parallel computation of ALU
  - Calculate result tag
- TPU generates a **trap** for **unexpected input**

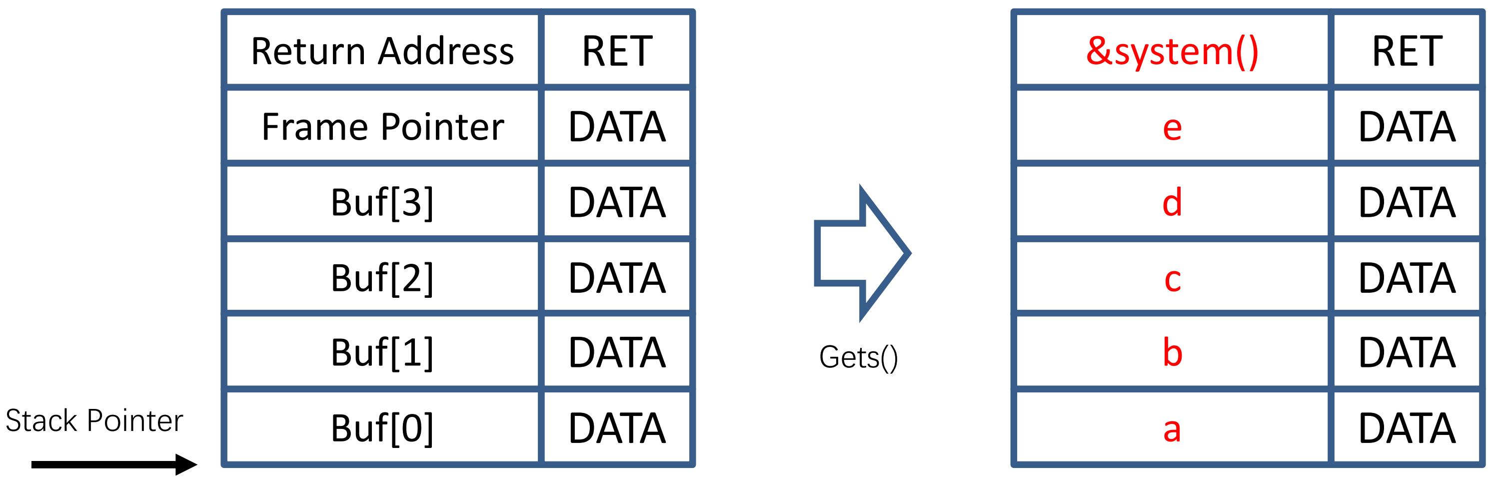


Tag Propagation

Operation	**Example**	**Tag Propagation**
Addition/Subtraction	`ADD RD, RS1, RS2`	`RD.tag = RS1.tag` $\oplus$ `RS2.tag`
Other ALU operations	`SRA RD, RS1, RS2`	`RD.tag = RS1.tag | RS2.tag`
	`XORI RD, RS1, Imm`	`RD.tag = RS1.tag`
Loads	`LW RD, RS1, Imm`	`RD.tag = Mem[RS1 + Imm].tag`
Stores	`SW RS1, RS2, Imm`	`Mem[RS1 + Imm].tag = RS2.tag`
Jump and Link (Call)	`JAL RD, Imm`	`RD.tag = TAG_PC`
Explicit Tag Set	`SETTAG RD, Imm`	`RD.tag = Imm`
Register Clear	`ADDI RD, R0, 0`	`RD.tag = 0`

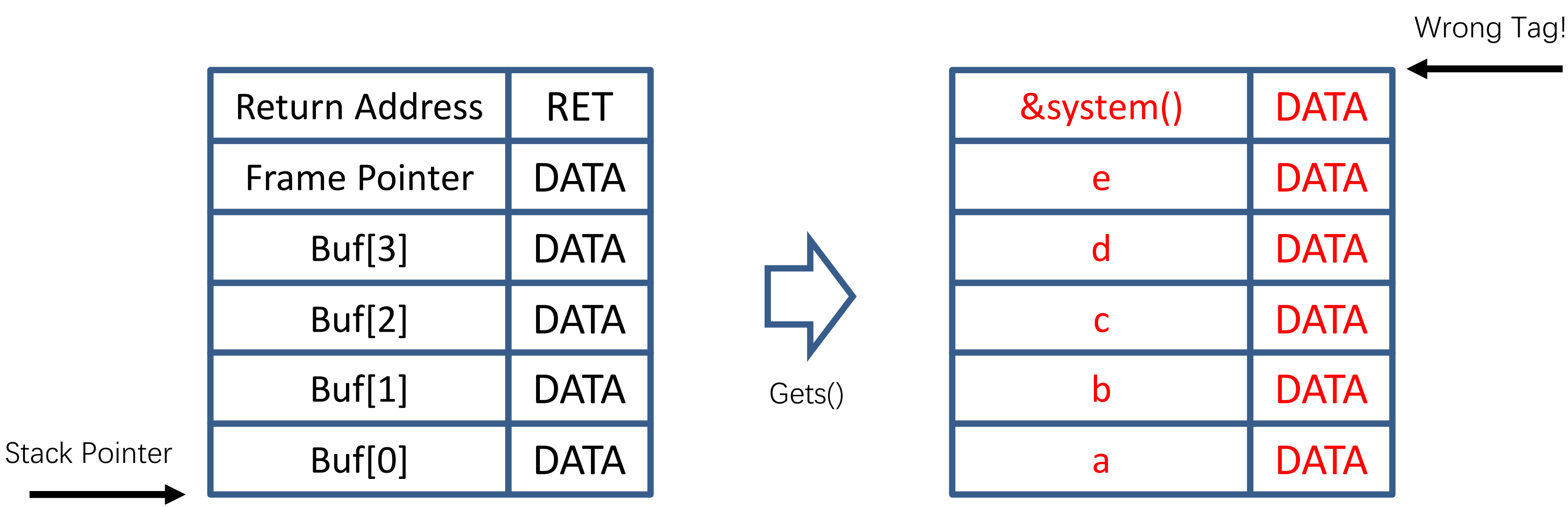
TAXI

- Call/Return Discipline Protection
  - *Return* can only return to the address pushed by *call*



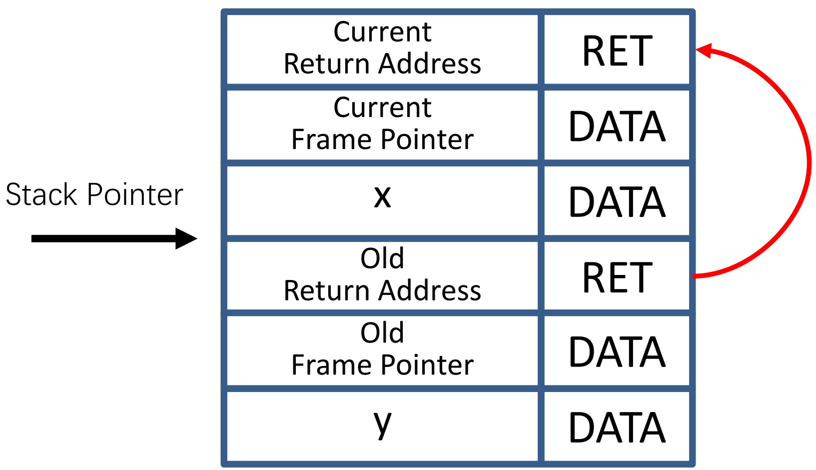
TAXI

- Call/Return Discipline Protection
  - *Return* can only return to the address pushed by *call*



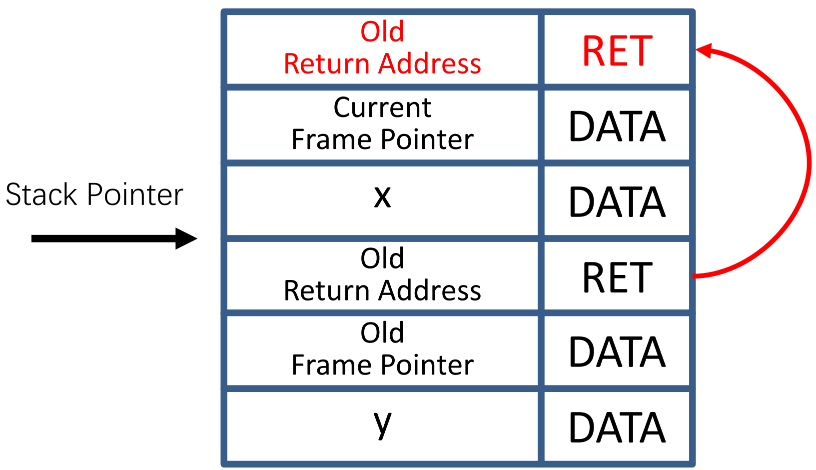
Replay Attack

- Old return addresses are not **cleaned**
- Attacker can "**replay**" existing return addresses



Replay Attack

- Old return addresses are not **cleaned**
- Attacker can "**replay**" existing return addresses



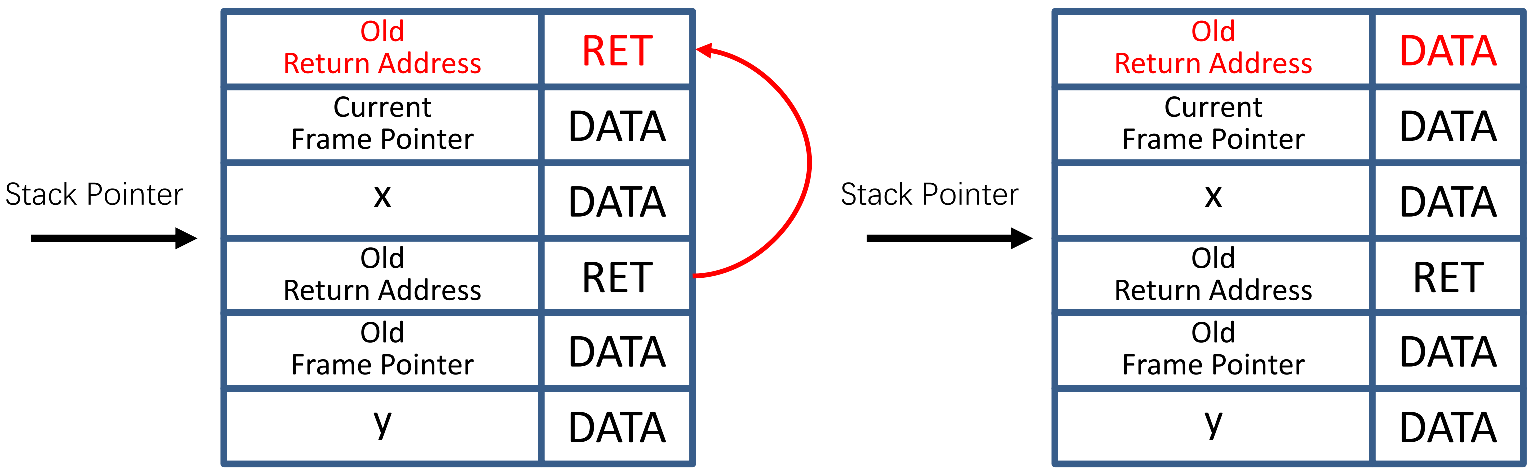
Linearity

Operation	Example	Tag Propagation
Addition/Subtraction	`ADD RD, RS1, RS2`	`RD.tag = MRET(RS1.tag ⊕ RS2.tag)`
Other ALU operations	`SRA RD, RS1, RS2`	`RD.tag = MRET(RS1.tag | RS2.tag)`
	`XORI RD, RS1, Imm`	`RD.tag = MRET(RS1.tag)`

- **Remove** the *return tag* after a return address is copied to a register/memory.

Linearity

- Old return addresses are not **cleaned**
- Attacker can "**replay**" existing return addresses



Architectural Support for Software-Defined Metadata Processing, ASPLOS'15

PUMP

Goals

- User-defined metadata processing
  - **Arbitrary metadata size**
  - **Arbitrary policy**
    - CFI + IFC (Information Flow Control) + DFI + …
  - With **low overhead**
    - Performance overhead and hardware resources overhead

Overview of PUMP

- PUMP: Programmable Unit for Metadata Processing
  - An extension to a conventional RISC processor
  - **Pointer-size** metadata tag for each word
  - Propagation tag on each instruction

Metadata Tag

- Each word has a **pointer-sized** tag
  - Small tag directly stored
  - Large tag stored indirectly in memory
- Tagged memory, cache, register, PC,…
- Tag **cannot** be addressable
  - Only updated by PUMP rules

PUMP Rule

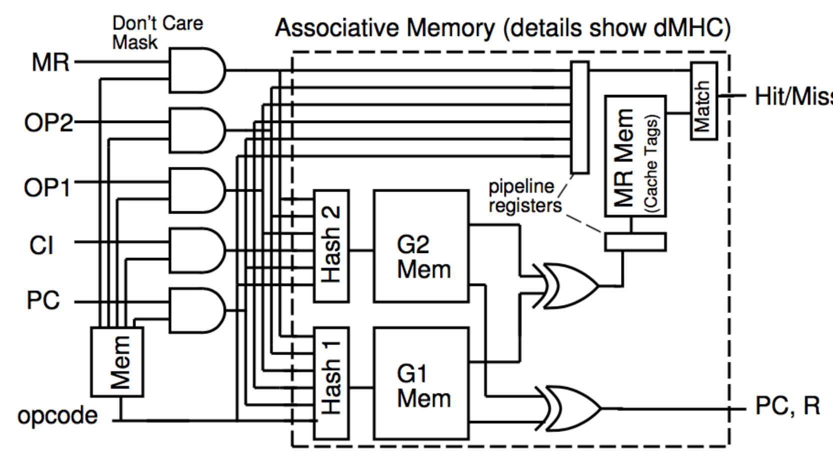
- Define tag-propagation policy
- *opcode : (PC, CI, OP_1, OP_2, MR) -> (PC_{new}, R)/Fail*
  - opcode : opcode of current instruction
  - PC : tag of PC
  - CI : tag of current instruction
  - OP_1/OP_2: tag of operator (register)
  - MR: tag of value read from memory
  - PC_{new} : update tag of PC
  - R : update tag of result

PUMP Rule - CFI

- $return$: (empty, -, -, -, -) -> (check, -)
- $\overline{return}$: (check, tgt, -, -, -) -> (empty, -)
- $\overline{return}$: (empty, -, -, -, -) -> (empty, -)
- $return$: (check, tgt, -, -, -) -> (check, -)

Rule Cache

- Provide single-cycle common-case computation on metadata
- A hardware-cache for most recently used rules
  - Cache-hit does not add extra cycle
- Associative mapping
  - Opcode and 5 input tags
  - 2 output tags
  - Compare tag-pointer
  - Failure cases are not cached



Miss Handler

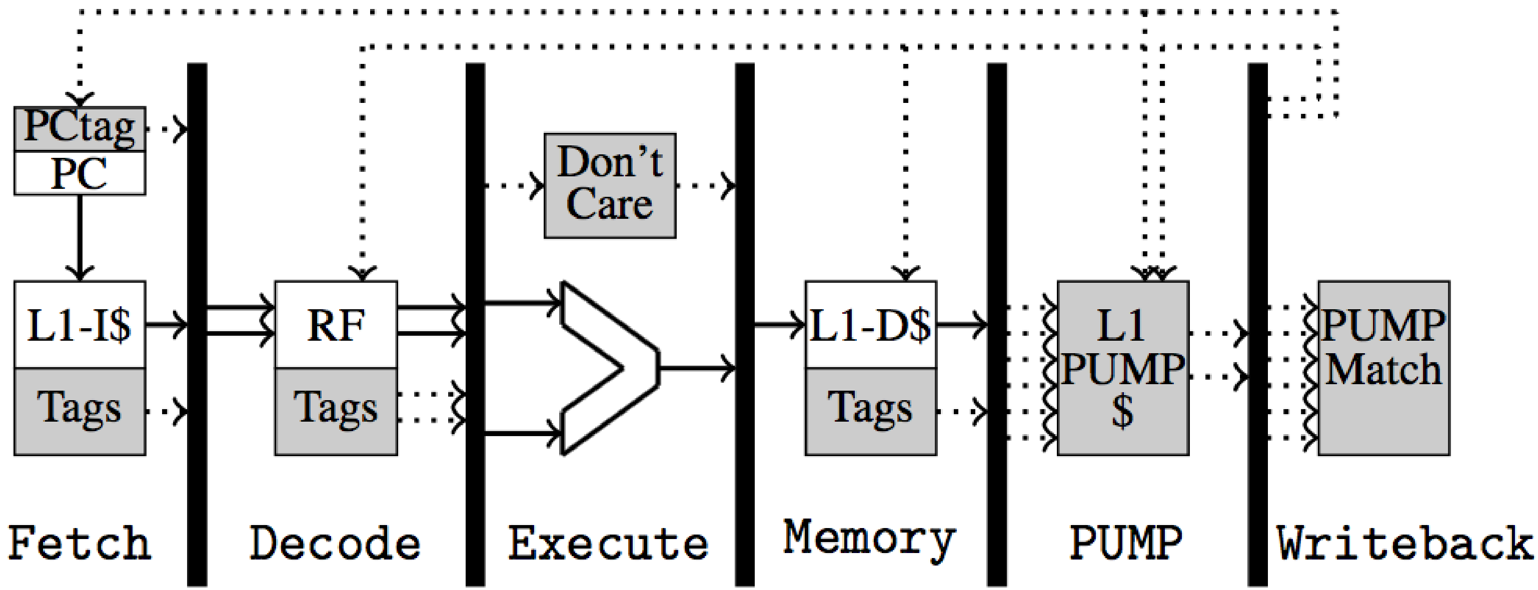
- **Save** current opcode and tags in new registers
- **Transfer control to handler** running in a special mode
- Miss handler **decides if the operation is allowed**
  - Yes, generates an rule, when return
  - No, invokes a suitable security fault handler
- Miss handler **installs** this rule into caches and **re-issues** the faulting instruction
  - *Miss-handler-return* instruction

Miss Handler

Algorithm 1 Taint Tracking Miss Handler (Fragment)

switch (op)
case `add`, `sub`, `or`**:**
  $PC_{new} \leftarrow PC$
  $R \leftarrow$ canonicalize$(CI \cup OP1 \cup OP2)$
(... cases for other instructions omitted ...)
default: trap to error handler

Pipeline Integration



Design Problem

- Large memory costs
  - +190% memory area overhead
- Performance overhead
  - Fast single policy (average 10%)
  - Slow composite policy (worst case 780%)
- Energy overhead
  - 400% for single policies (worst case)
  - 1600% for the Composite policy (worst case)

BOUND CHECKING

Bounds Checking

- Tracking bounds information
- Check bounds before memory accesses

- Challenges
  - How to record the bounds information
  - How to efficiently check bounds

Bounds Checking

```c
int foo(const char * str) {

    char buf[1024];
    if (strlen(str) >= sizeof(buf)){
        /*str too long*/
        exit(1);
    }
    strcpy(buf, str);
    ...
}

```

AddressSanitizer: A Fast Address Sanity Checker (USENIX ATC'12)

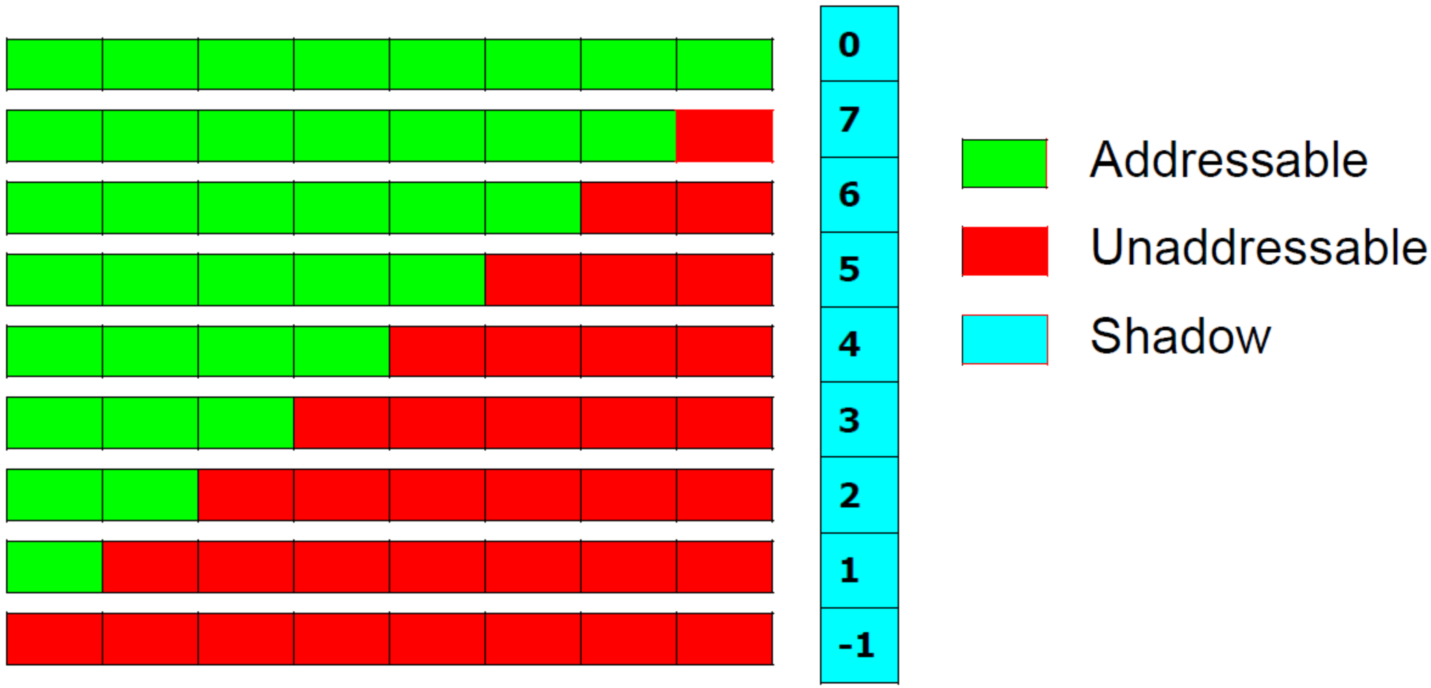
ADDRESS SANITIZER

Overview

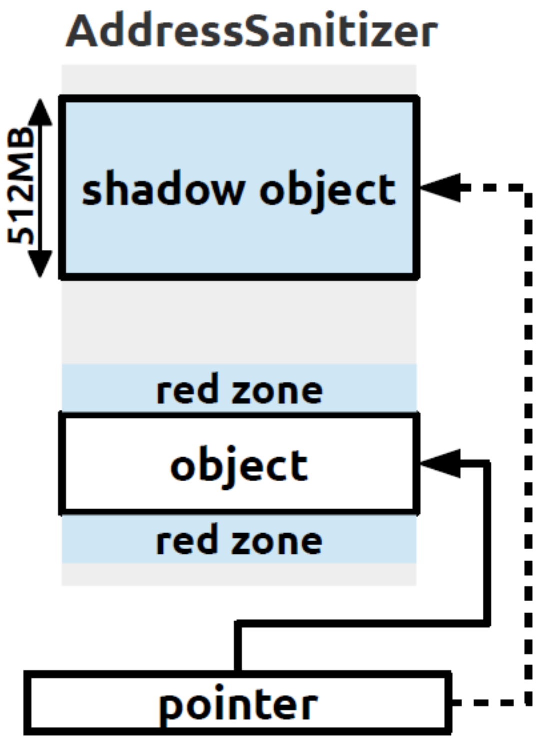
- Compile-time instrumentation module
  - LLVM, platform independent
  - ~1 KLOC
- Run-time library
  - Supports Linux, MacOS, Android
  - ~9 KLOC
- Released in May 2011
- Part of LLVM since November 2011

Design

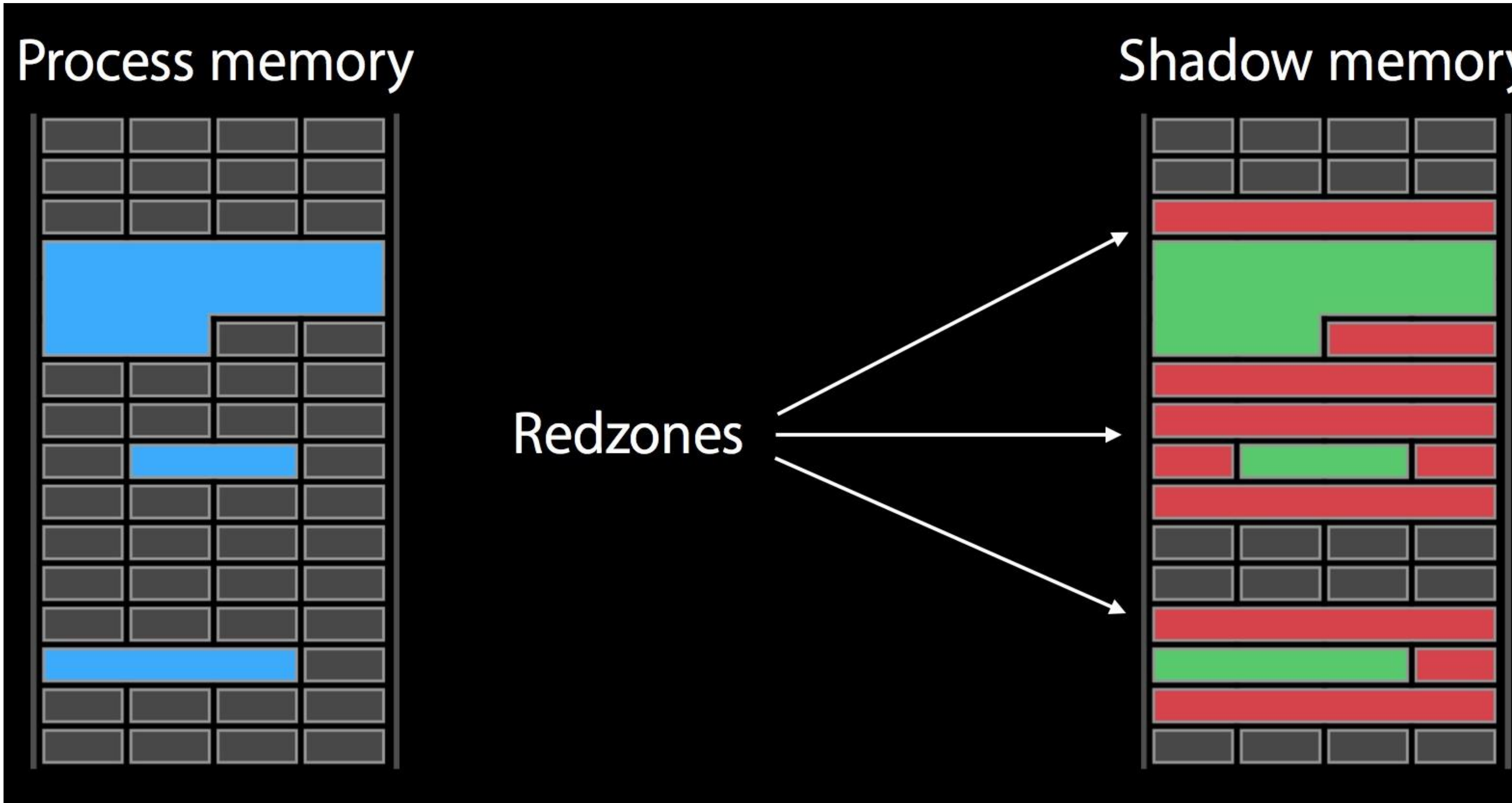
- Shadow Byte



Design



- 1/8 of the memory are use to save the shadow object.


Process memory
Shadow memory
Redzones

Instrumentation: 8 bytes access

```
*a = ...
```

```
char *shadow = (a>>3)+Offset;
if (*shadow)
        ReportError(a);
*a = ...
```

Instrumentation: N bytes access(N=1,2,4)

```
*a = ...
```

```
char *shadow = (a>>3)+Offset;
if (*shadow  && *shadow <= ((a&7)+N-1))
        ReportError(a);
*a = ...
```

Performance Evaluation

1.73x slowdown (reads & writes) 1.26x slowdown (writes only)