MODERN
MEMORY DEFENSES

GRAD SEC

SEP 14 2017

TODAY'S PAPERS

Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

James Newsome
jnewsome@ece.cmu.edu
Carnegie Mellon University

Dawn Song
dawnsong@cmu.edu
Carnegie Mellon University

Abstract

1. Introduction

Control-Flow Integrity
Principles, Implementations, and Applications

Martín Abadi
Computer Science Dept.
University of California
Santa Cruz

Mihai Budiu Úlfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Princeton University

ABSTRACT

Categories and Subject Descriptors

General Terms

Keywords

1. INTRODUCTION

2. RELATED WORK

CONTROL FLOW INTEGRITY

Fundamentally, code injection attacks altered the target program's control flow

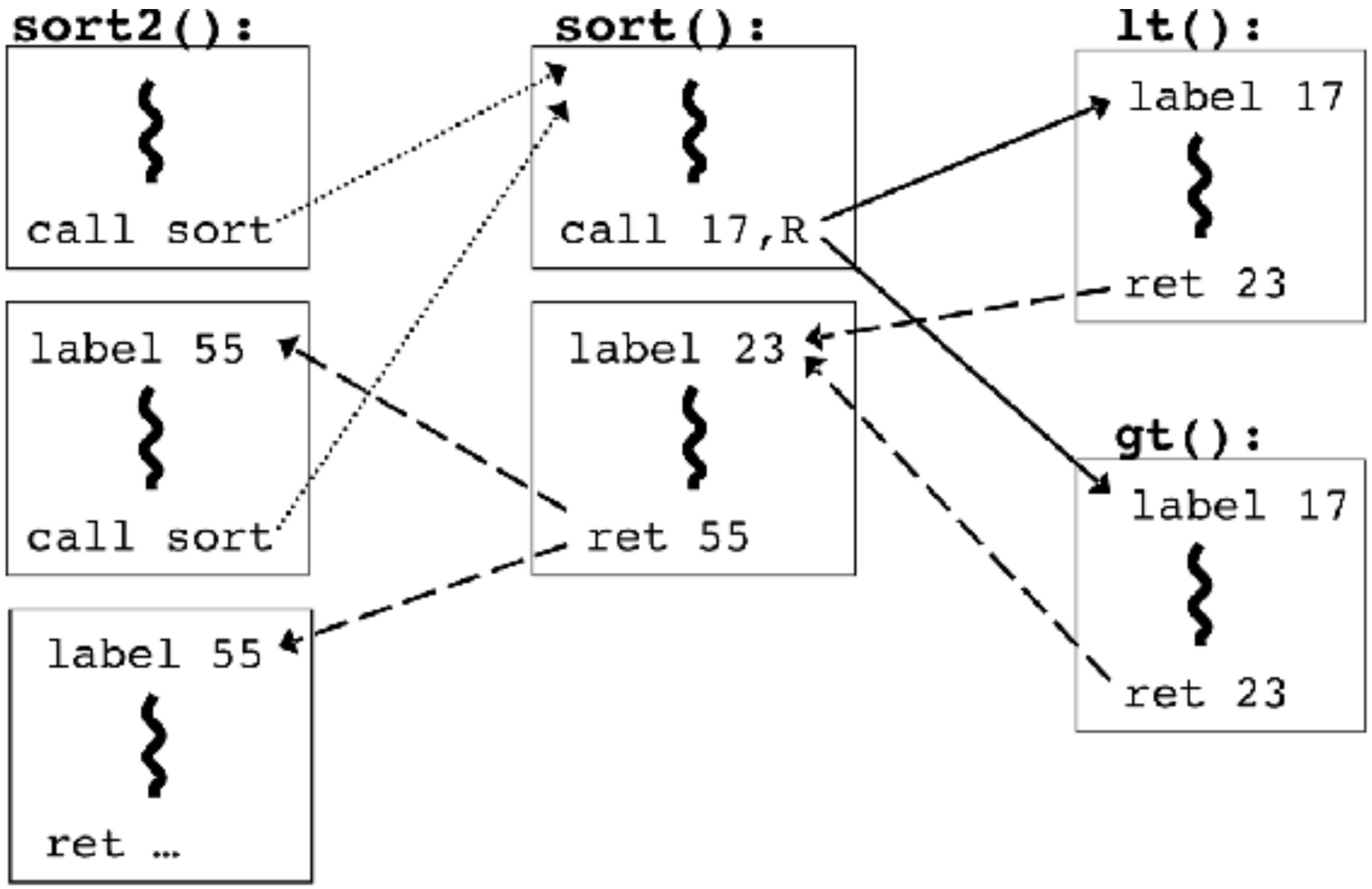
Recall: Confidentiality, Integrity, Availability

Most integrity defenses seek to detect

Typically they are unable to outright prevent

CONTROL FLOW GRAPH

```
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 );
}
```



Code injection, return to libc, ROP… all of them alter where one of the "ret"s points

REFERENCE MONITORS

Code or system responsible for checking whether data/execution matches some policy

File permissions, password checker, airline employees checking tickets…

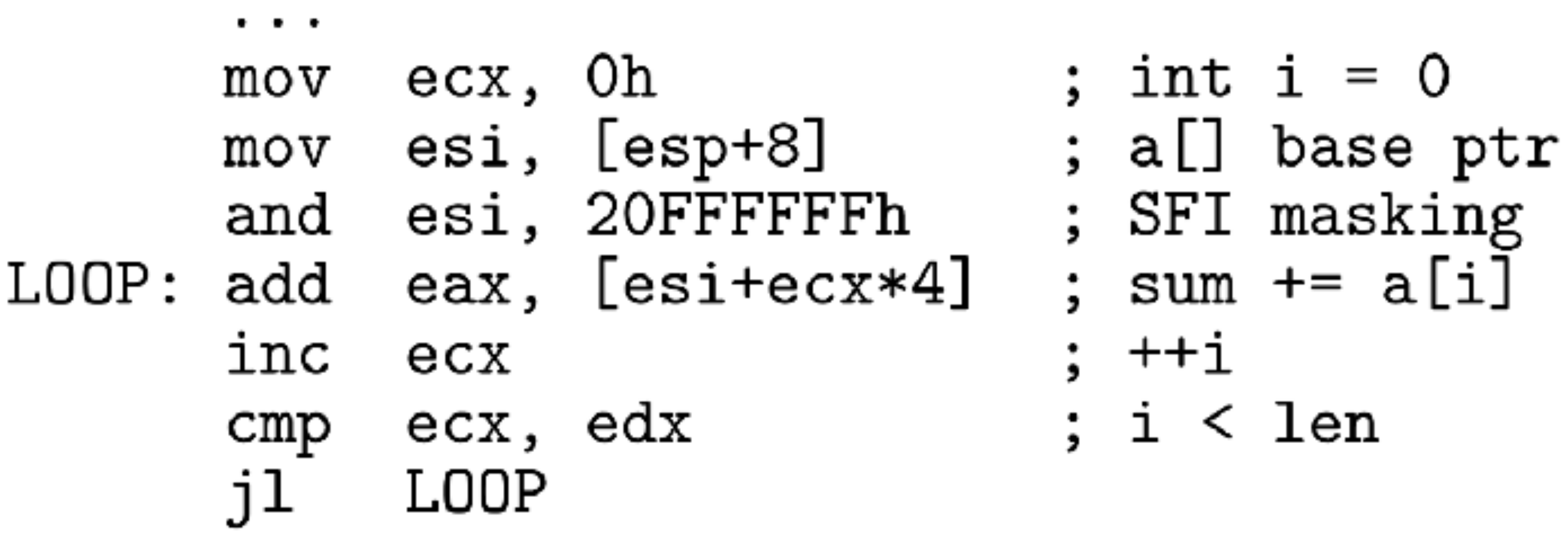
Mediates between user and sensitive resource

CFI is an *inline* reference monitor

ENSURE COMPLETE MEDIATION

SOFTWARE FAULT ISOLATION (SFI)

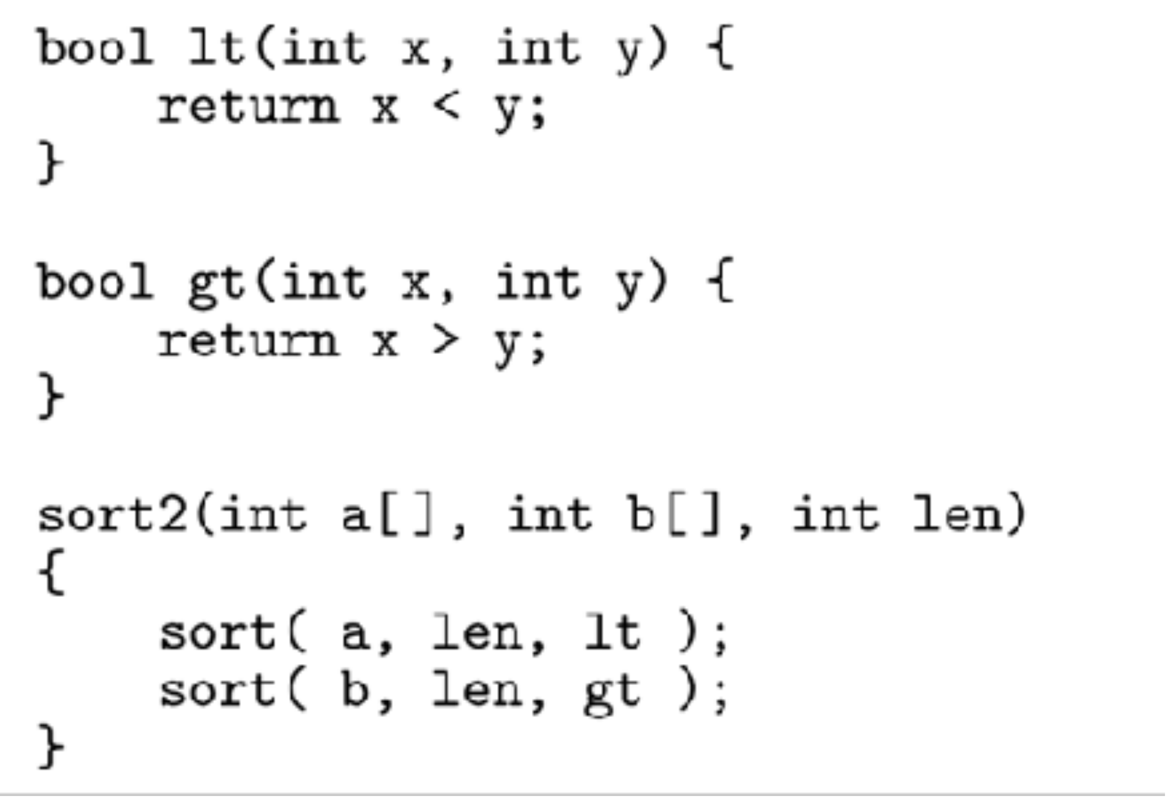
Insert code at each machine code instruction to ensure that the target memory region lies within some bounds



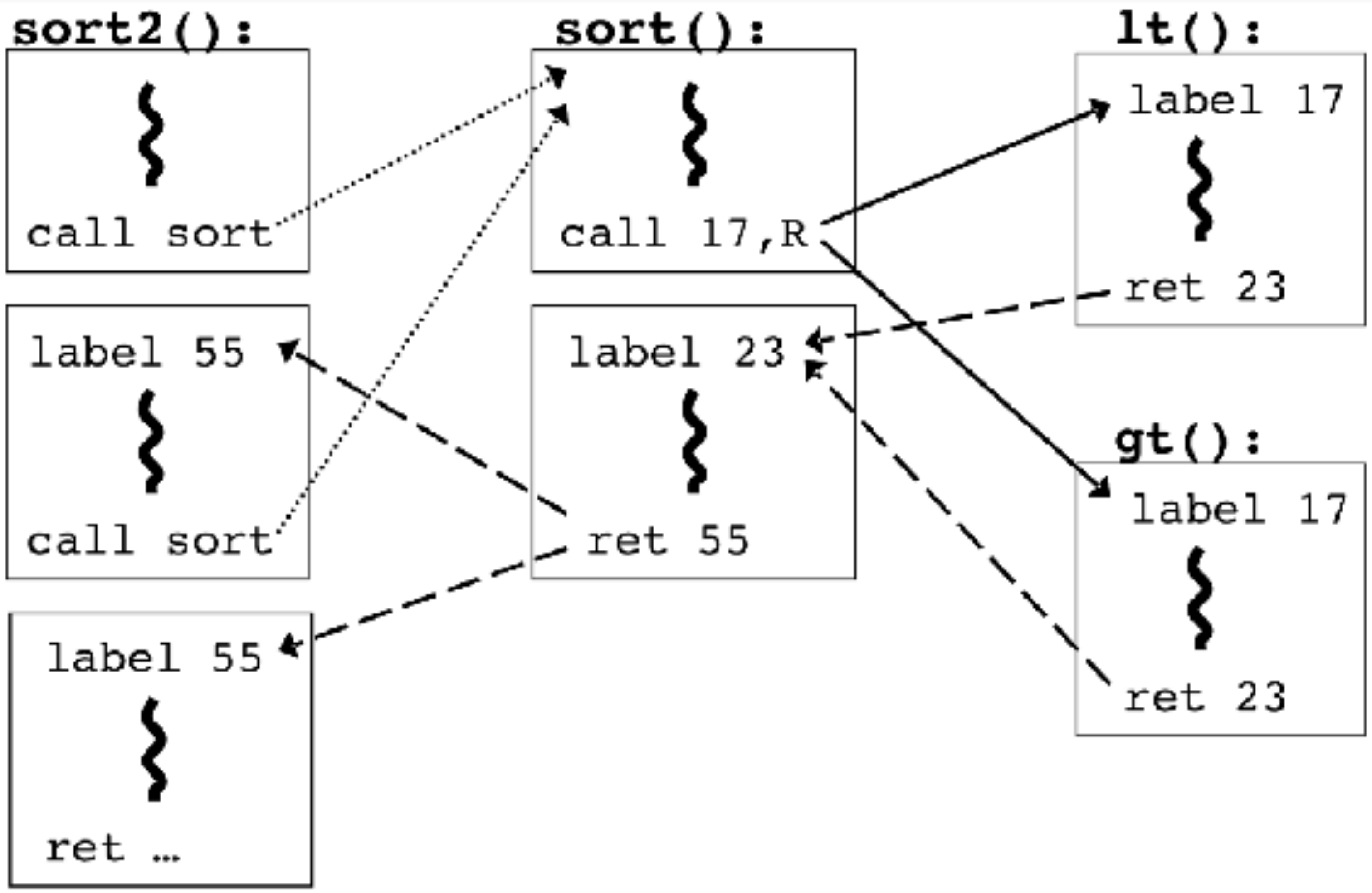
```
        ...
        mov   ecx, 0h               ; int i = 0
        mov   esi, [esp+8]          ; a[] base ptr
        and   esi, 20FFFFFFh        ; SFI masking
LOOP:   add   eax, [esi+ecx*4]      ; sum += a[i]
        inc   ecx                   ; ++i
        cmp   ecx, edx              ; i < len
        jl    LOOP
```

Keep only the LSBs (zero with ‘and’ then add the target memory region’s MSBs

INTEGRITY WITH LABELS



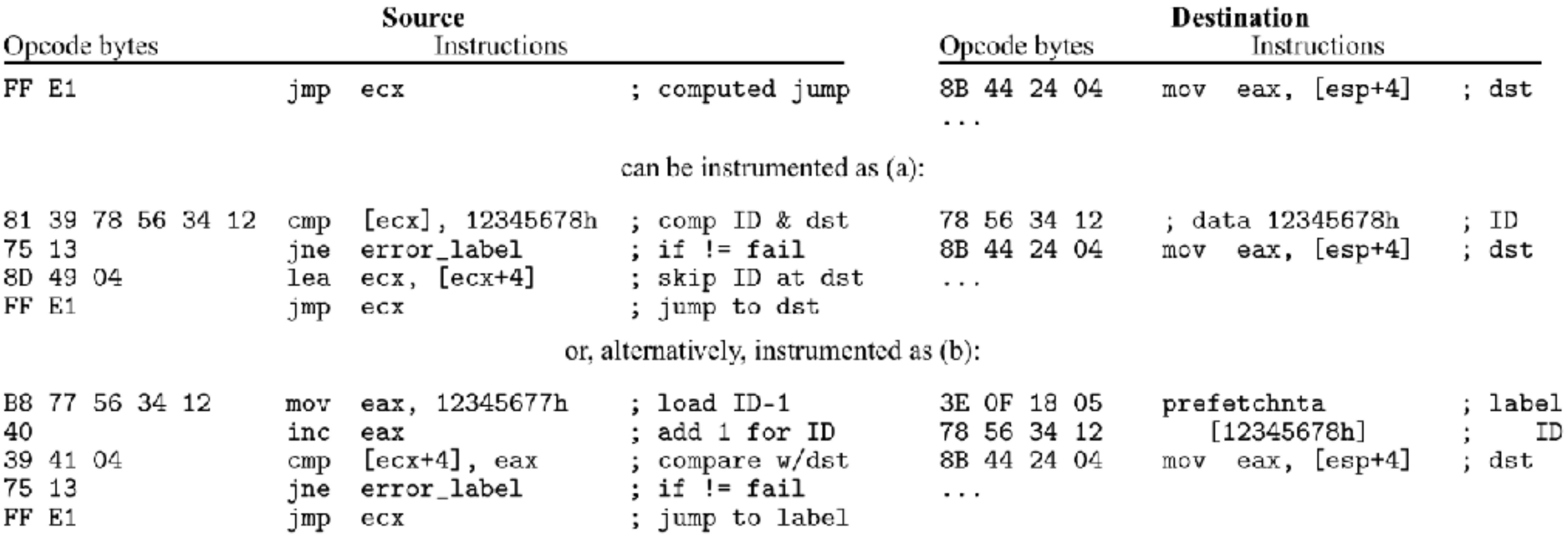
```
bool lt(int x, int y) {
    return x < y;
}

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

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



Note that we start in the trusted code.

The goal is to make sure we never ret somewhere we shouldn't

INLINING CFI



```
                         Source                                                    Destination
Opcode bytes                  Instructions                          Opcode bytes           Instructions
FF E1              jmp   ecx                   ; computed jump      8B 44 24 04     mov   eax, [esp+4]    ; dst
                                                                    ...
                                       can be instrumented as (a):
81 39 78 56 34 12  cmp   [ecx], 12345678h      ; comp ID & dst      78 56 34 12     ; data 12345678h      ; ID
75 13              jne   error_label           ; if != fail         8B 44 24 04     mov   eax, [esp+4]    ; dst
8D 49 04           lea   ecx, [ecx+4]          ; skip ID at dst     ...
FF E1              jmp   ecx                   ; jump to dst
                                   or, alternatively, instrumented as (b):
B8 77 56 34 12     mov   eax, 12345677h        ; load ID-1          3E 0F 18 05     prefetchnta           ; label
40                 inc   eax                   ; add 1 for ID       78 56 34 12        [12345678h]        ;     ID
39 41 04           cmp   [ecx+4], eax          ; compare w/dst      8B 44 24 04     mov   eax, [esp+4]    ; dst
75 13              jne   error_label           ; if != fail         ...
FF E1              jmp   ecx                   ; jump to label
```

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

Will only jump to a part of the code with the label 0x12345678

SECURITY GUARANTEES

Attack model: arbitrary control over the data portion of memory

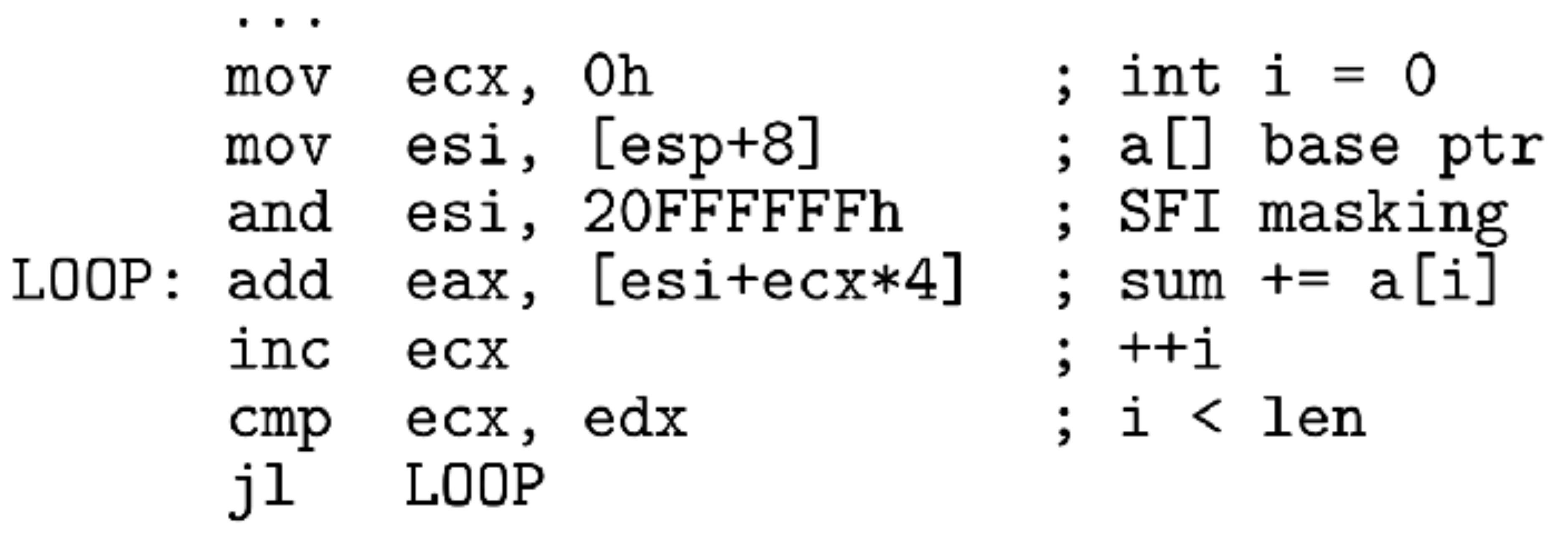
UNQ: No label appears elsewhere in code

NWC: Code segment is not writable

NXD: Data segment is not executable

SOFTWARE FAULT ISOLATION (SFI)

Insert code at each machine code instruction to ensure that the target memory region lies within some bounds



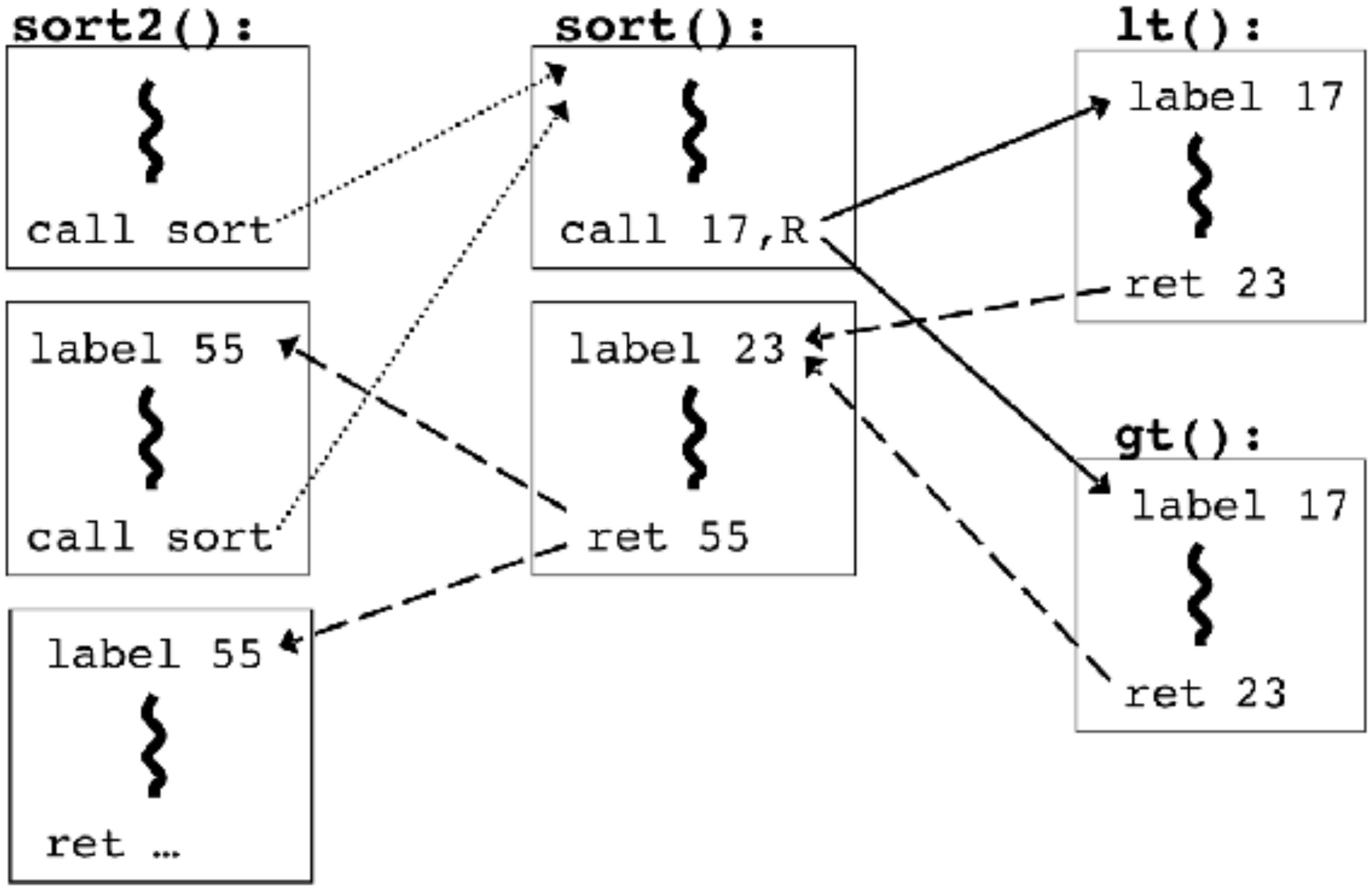
```
       ...
       mov   ecx, 0h             ; int i = 0
       mov   esi, [esp+8]        ; a[] base ptr
       and   esi, 20FFFFFFh      ; SFI masking
LOOP:  add   eax, [esi+ecx*4]    ; sum += a[i]
       inc   ecx                 ; ++i
       cmp   ecx, edx            ; i < len
       jl    LOOP
```

Normally you want the ‘and’ in the loop,

But CFI ensures no jumps into the loop

LABELS ARE NOT UNIQUE

```
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 );
}
```



Attacker could potentially cause sort() to return to either of the memory locations labelled 55

LABELS ARE NOT UNIQUE

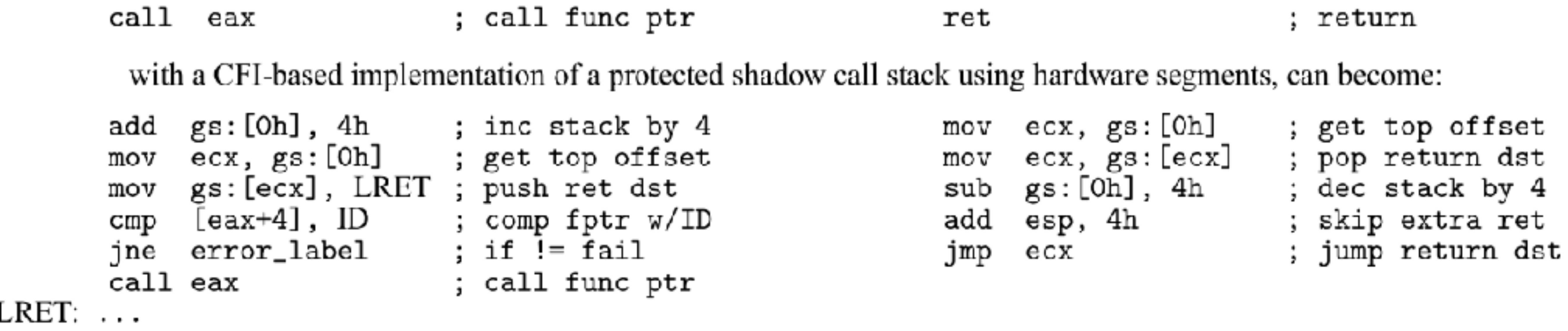
Code duplication

Shadow stack

SHADOW CALL STACKS

One possibility: SFI to maintain a region of memory (e.g., 0x1*) specifically for the shadow call stack

Hardware support: x86 offers memory segments



```
call  eax                 ; call func ptr           ret                      ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
add   gs:[0h], 4h       ; inc stack by 4          mov   ecx, gs:[0h]     ; get top offset
mov   ecx, gs:[0h]      ; get top offset          mov   ecx, gs:[ecx]    ; pop return dst
mov   gs:[ecx], LRET    ; push ret dst            sub   gs:[0h], 4h      ; dec stack by 4
cmp   [eax+4], ID       ; comp fptr w/ID          add   esp, 4h          ; skip extra ret
jne   error_label       ; if != fail              jmp   ecx              ; jump return dst
call eax                ; call func ptr
LRET:  ...
```

%gs always points to shadow stack segment

Protected by CFI + static analysis of code

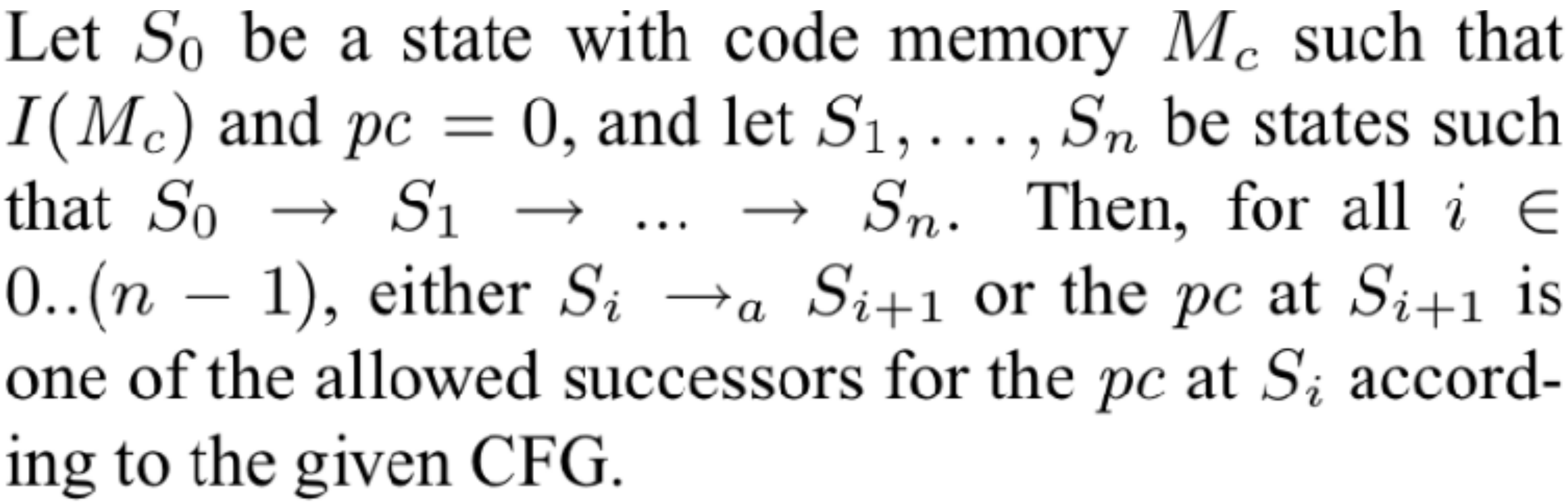
SECURITY GUARANTEES

Attack model: arbitrary control over the data portion of memory

UNQ: No label appears elsewhere in code

NWC: Code segment is not writable

NXD: Data segment is not executable



Let S_0 be a state with code memory M_c such that $I(M_c)$ and $pc = 0$, and let $S_1, \ldots, S_n$ be states such that $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$. Then, for all $i \in 0..(n-1)$, either $S_i \rightarrow_a S_{i+1}$ or the pc at S_{i+1} is one of the allowed successors for the pc at S_i according to the given CFG.

EVALUATION

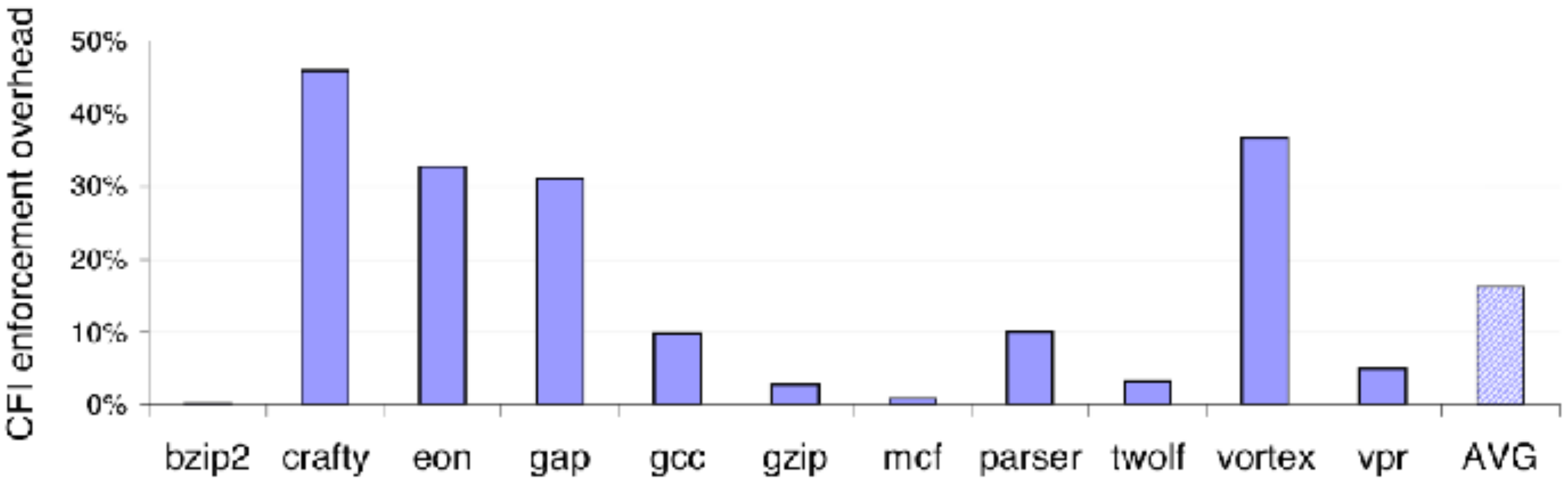


Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

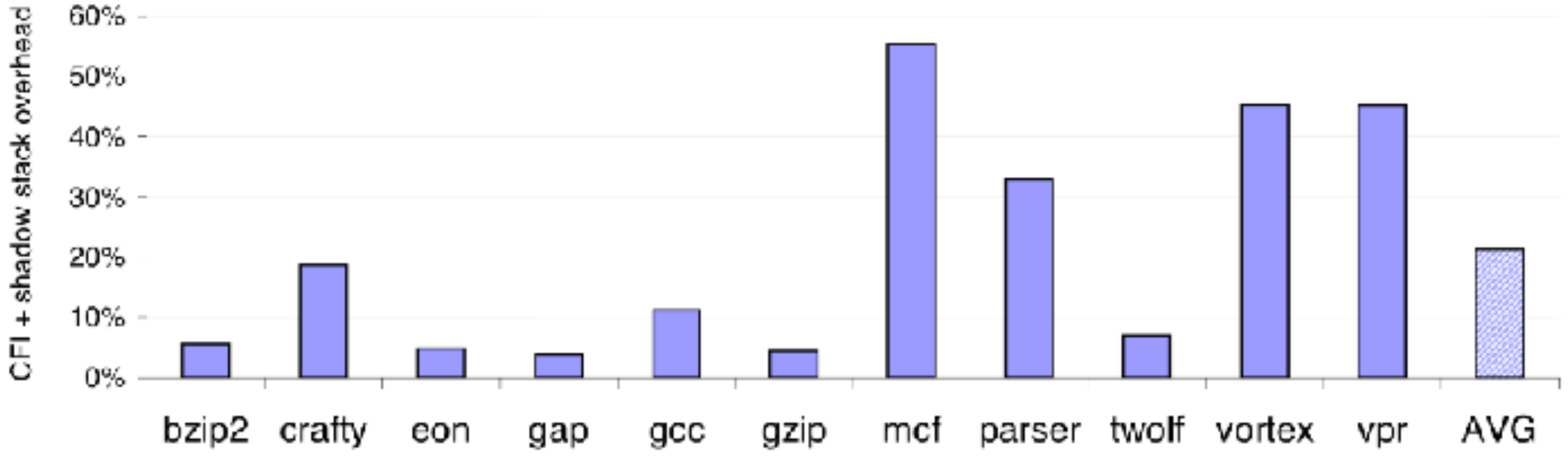


Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.

Shadow stack reduces some unnecessary ID checks during returns

CFI: SHORTCOMINGS

CFI: SHORTCOMINGS

No dynamically generated code (functional programming?)

Requires recompiling the code

TODAY'S PAPERS

Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

James Newsome
jnewsome@ece.cmu.edu
Carnegie Mellon University

Dawn Song
dawnsong@cmu.edu
Carnegie Mellon University

Abstract

1. Introduction

Control-Flow Integrity

Principles, Implementations, and Applications

Martín Abadi
Computer Science Dept.
University of California
Santa Cruz

Mihai Budiu
Úlfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Princeton University

ABSTRACT

Categories and Subject Descriptors

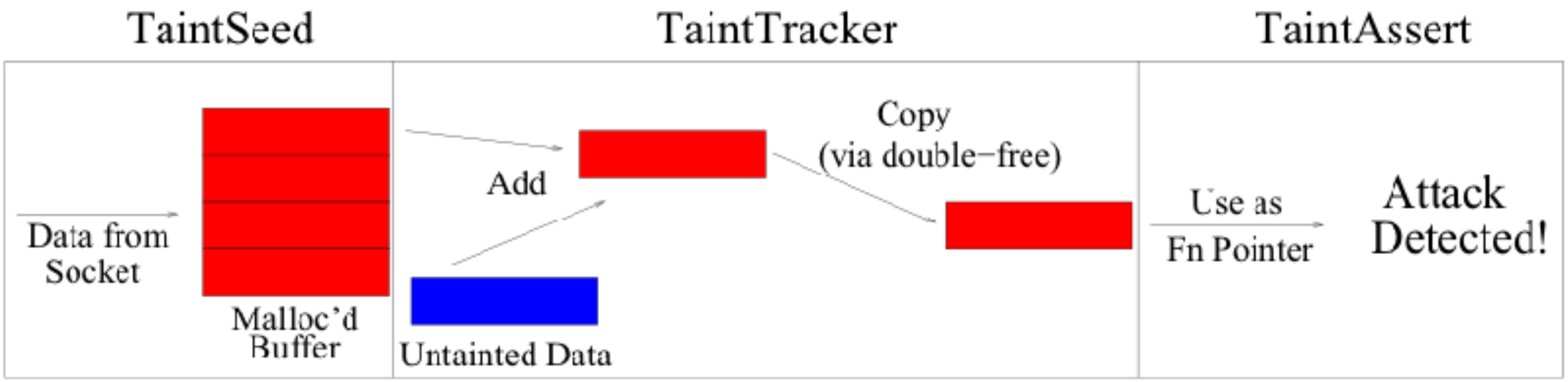
General Terms

Keywords

1. INTRODUCTION

2. RELATED WORK

TAINT TRACKING: HIGH LEVEL IDEA



Potentially malicious input "taints" memory

Track what gets tainted

Enforce that some operations only work on untainted data

TAINT TRACKING: CHALLENGES

How do we track memory accesses?

How do we keep track of what's tainted?

How do we protect the taint info?

How do we “propagate” taint?

TAINT PROPAGATION (TAINTDROID)

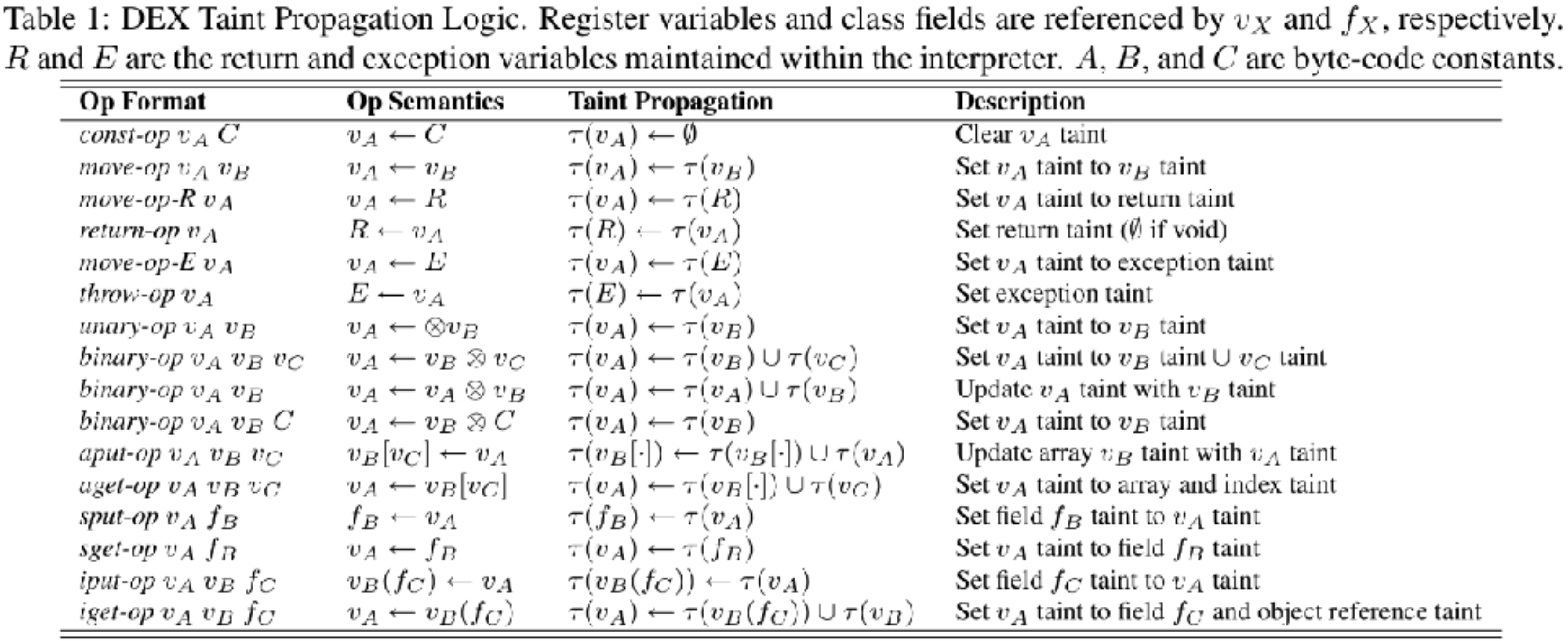


Table 1: DEX Taint Propagation Logic. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter. A, B, and C are byte-code constants.

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

Define what propagation rules for all operations

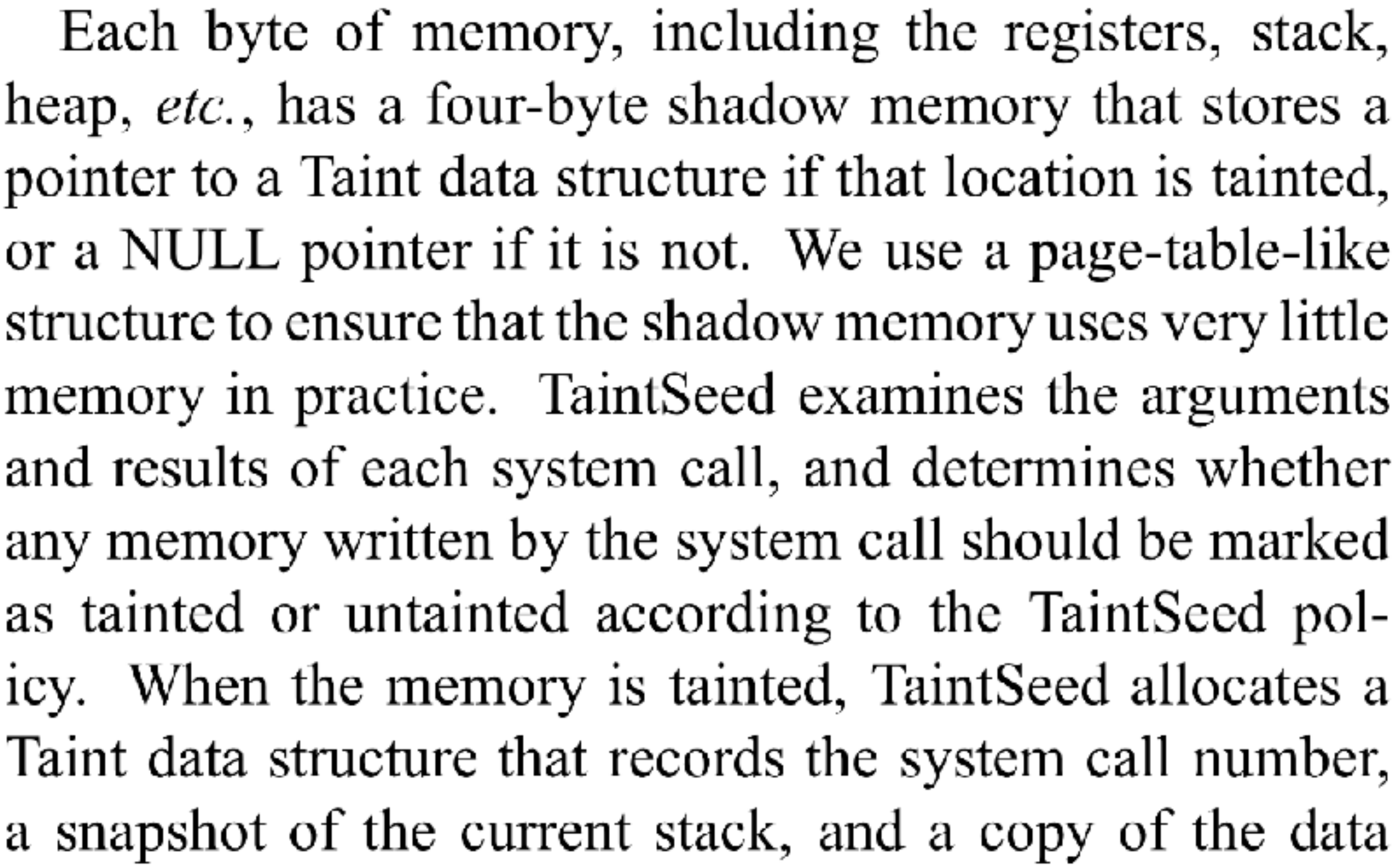
TAINT TRACKING

Instrument every (relevant) operation

Mechanism: Valgrind
  Translates x86 into its own instruction set
  Passes these to TaintCheck
  TaintCheck passes back modified instructions
    Add code to update taint info

TAINT STORING: RETURN OF THE SHADOW

1 byte memory -> 4 byte pointer -> taint data structure



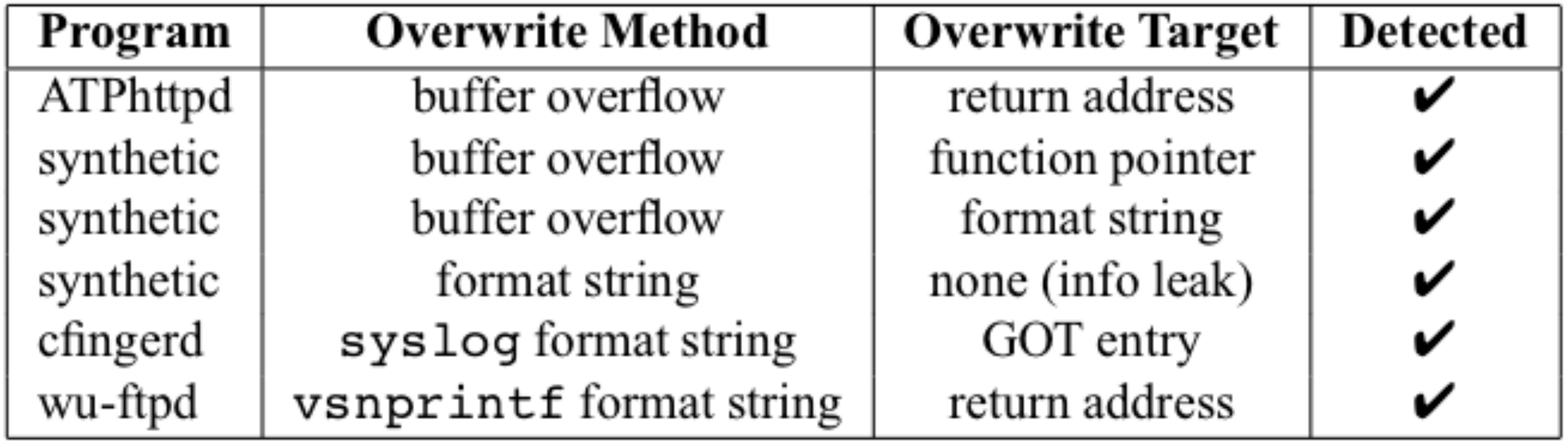
Each byte of memory, including the registers, stack, heap, *etc.*, has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted, or a NULL pointer if it is not. We use a page-table-like structure to ensure that the shadow memory uses very little memory in practice. TaintSeed examines the arguments and results of each system call, and determines whether any memory written by the system call should be marked as tainted or untainted according to the TaintSeed policy. When the memory is tainted, TaintSeed allocates a Taint data structure that records the system call number, a snapshot of the current stack, and a copy of the data

POLICY CHECKING

Must specify what operations aren't permitted on tainted data

EVALUATION

Has the possibility for false positives, false negatives



Program	Overwrite Method	Overwrite Target	Detected
ATPhttpd	buffer overflow	return address	✔
synthetic	buffer overflow	function pointer	✔
synthetic	buffer overflow	format string	✔
synthetic	format string	none (info leak)	✔
cfingerd	`syslog` format string	GOT entry	✔
wu-ftpd	`vsnprintf` format string	return address	✔

Table 1. Evaluation of TaintCheck's ability to detect exploits

EVALUATION

Has the possibility to adversely affect performance

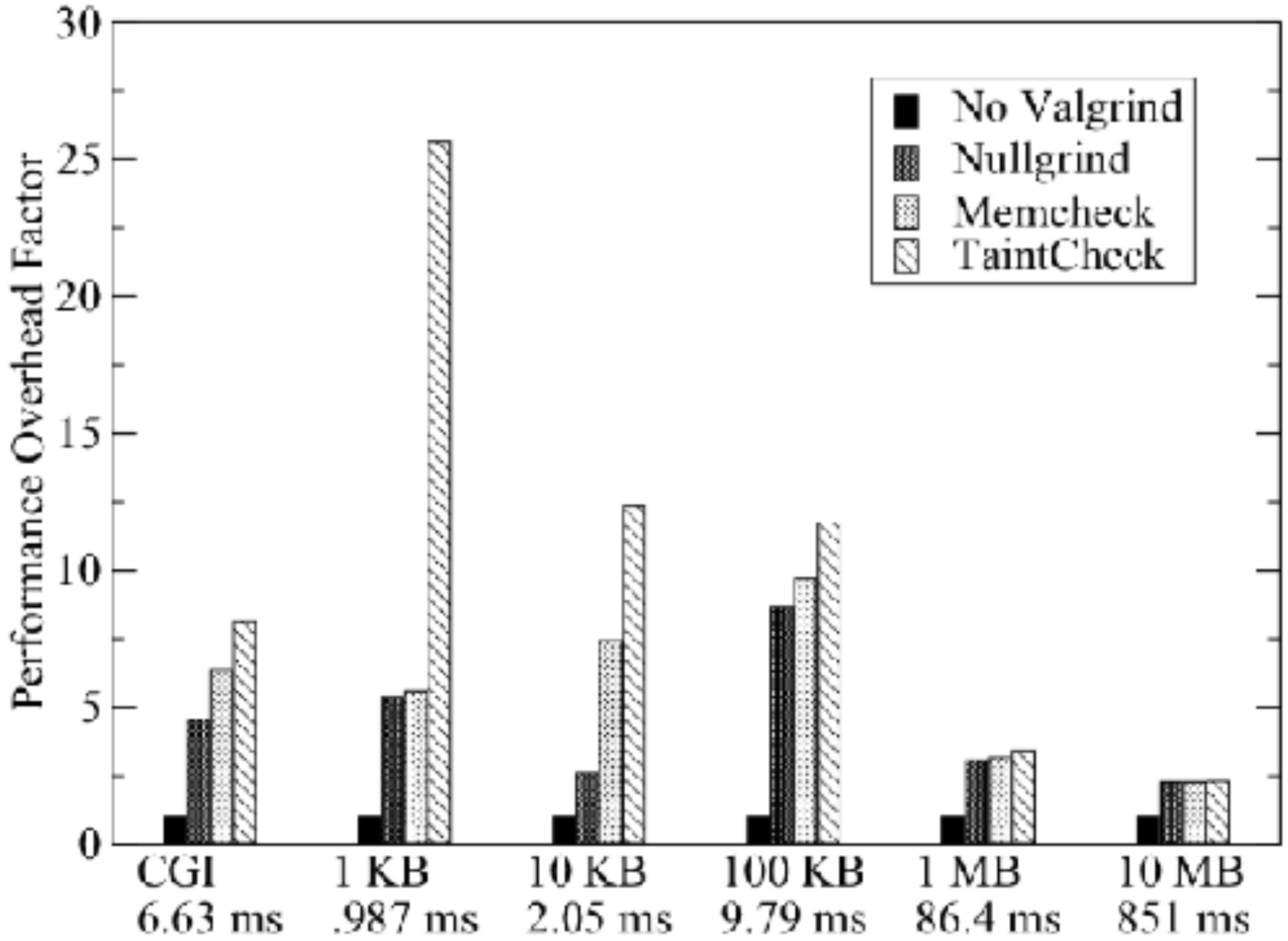


Figure 3. Performance overhead for Apache. Y-axis is the performance overhead factor: execution time divided by native execution time. Native execution times are listed below each experiment.

EVALUATION

Has the possibility to be overtrained to known vulnerabilities

TAINTDROID

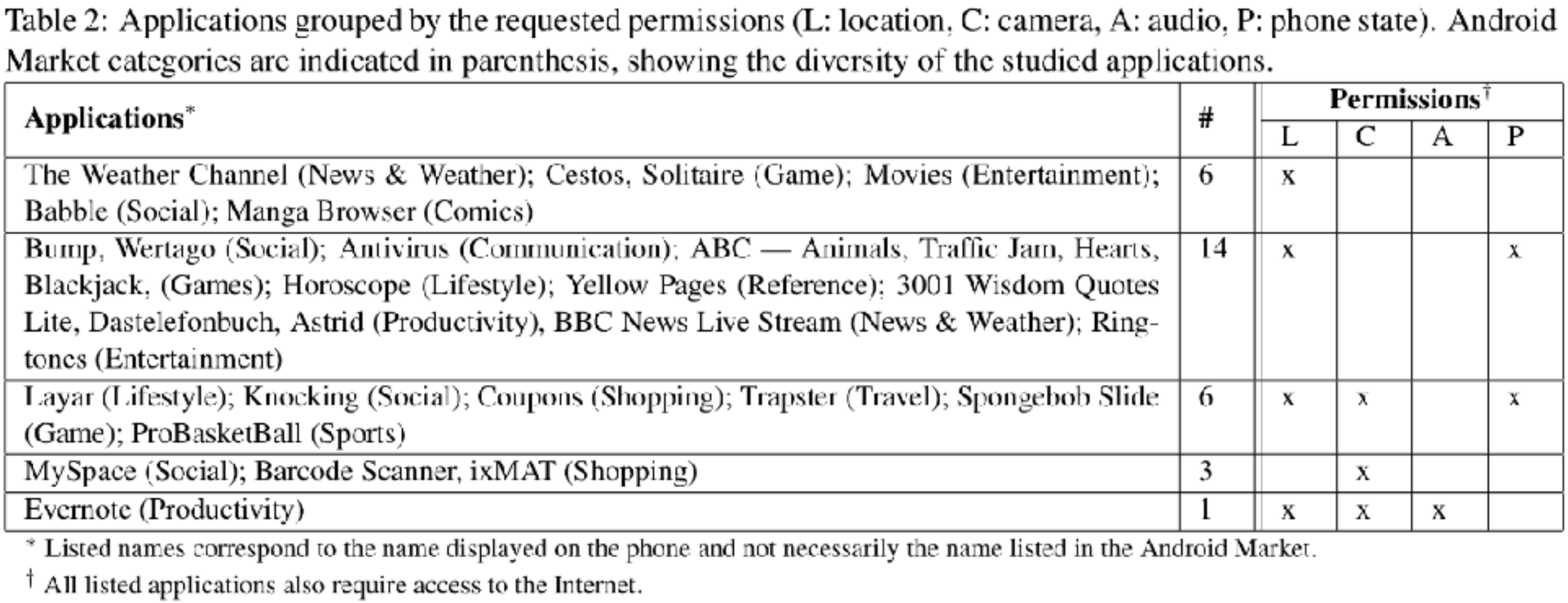


Table 2: Applications grouped by the requested permissions (L: location, C: camera, A: audio, P: phone state). Android Market categories are indicated in parenthesis, showing the diversity of the studied applications.

Applications*	#	**Permissions†**			
		L	C	A	P
The Weather Channel (News & Weather); Cestos, Solitaire (Game); Movies (Entertainment); Babble (Social); Manga Browser (Comics)	6	x			
Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic Jam, Hearts, Blackjack, (Games); Horoscope (Lifestyle); Yellow Pages (Reference); 3001 Wisdom Quotes Lite, Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment)	14	x			x
Layar (Lifestyle); Knocking (Social); Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports)	6	x	x		x
MySpace (Social); Barcode Scanner, ixMAT (Shopping)	3		x		
Evernote (Productivity)	1	x	x	x	

* Listed names correspond to the name displayed on the phone and not necessarily the name listed in the Android Market.

† All listed applications also require access to the Internet.

TAINTDROID

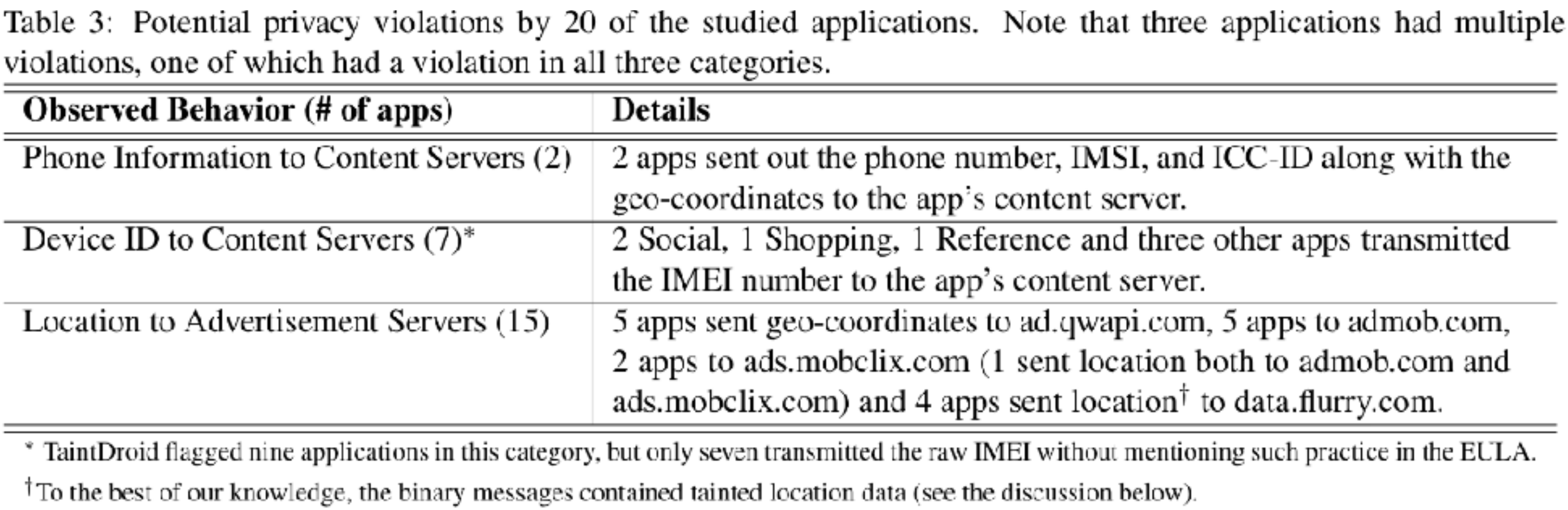


Table 3: Potential privacy violations by 20 of the studied applications. Note that three applications had multiple violations, one of which had a violation in all three categories.

Observed Behavior (# of apps)	**Details**
Phone Information to Content Servers (2)	2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server.
Device ID to Content Servers (7)*	2 Social, 1 Shopping, 1 Reference and three 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.

* TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

† To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).