Defending Against Return-Oriented Programming

Michalis Polychronakis

joint work with Vasilis Pappas and Angelos Keromytis

Columbia University

DIMVA ’12 – July 27, 2012

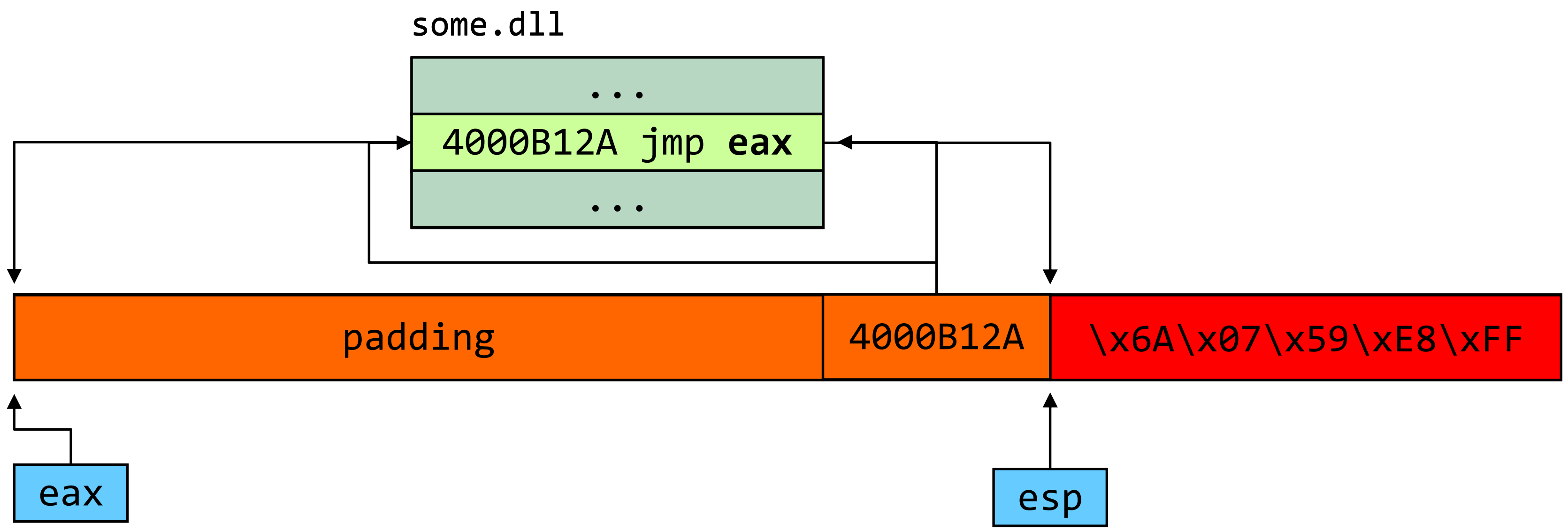
Outline

From shellcode to ROP

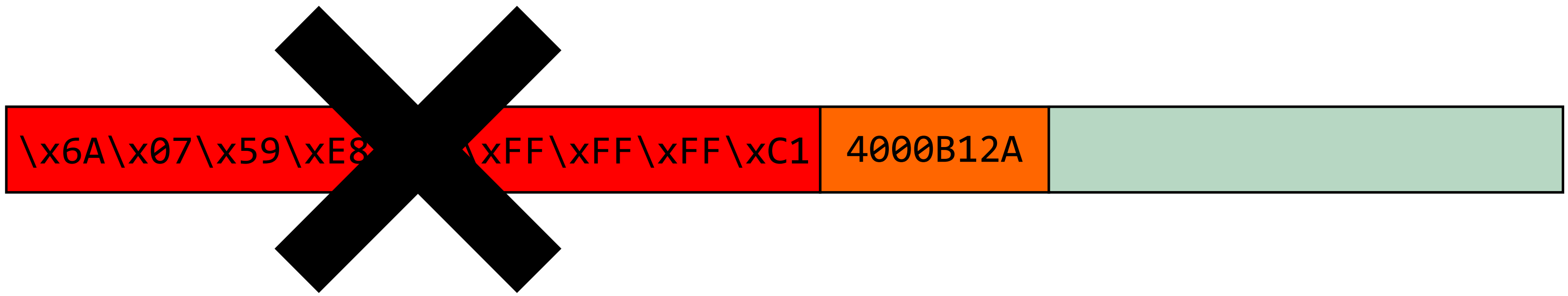
ROP payload detection

In-place code randomization

Code Injection



NX



W^X, PaX, Exec Shield, DEP

x86 support introduced by AMD, followed by Intel

- Pentium 4 (late models)

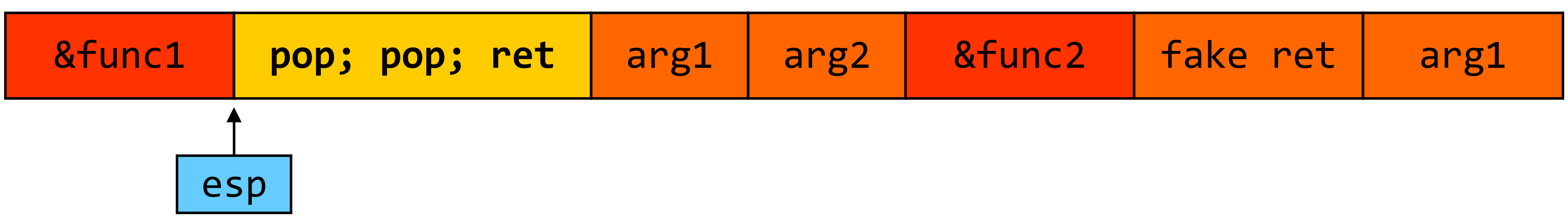
DEP introduced in XP SP2 (hardware-only)

- Applications can opt-in (SetProcessDEPPolicy() or /NXCOMPAT)

Ret2libc → ROP

ret2libc [Solar Designer '97]

ret2libc chaining [Nergal '01]



Ret2libc → ROP

Borrowed code chunks technique [Krahmer '05]

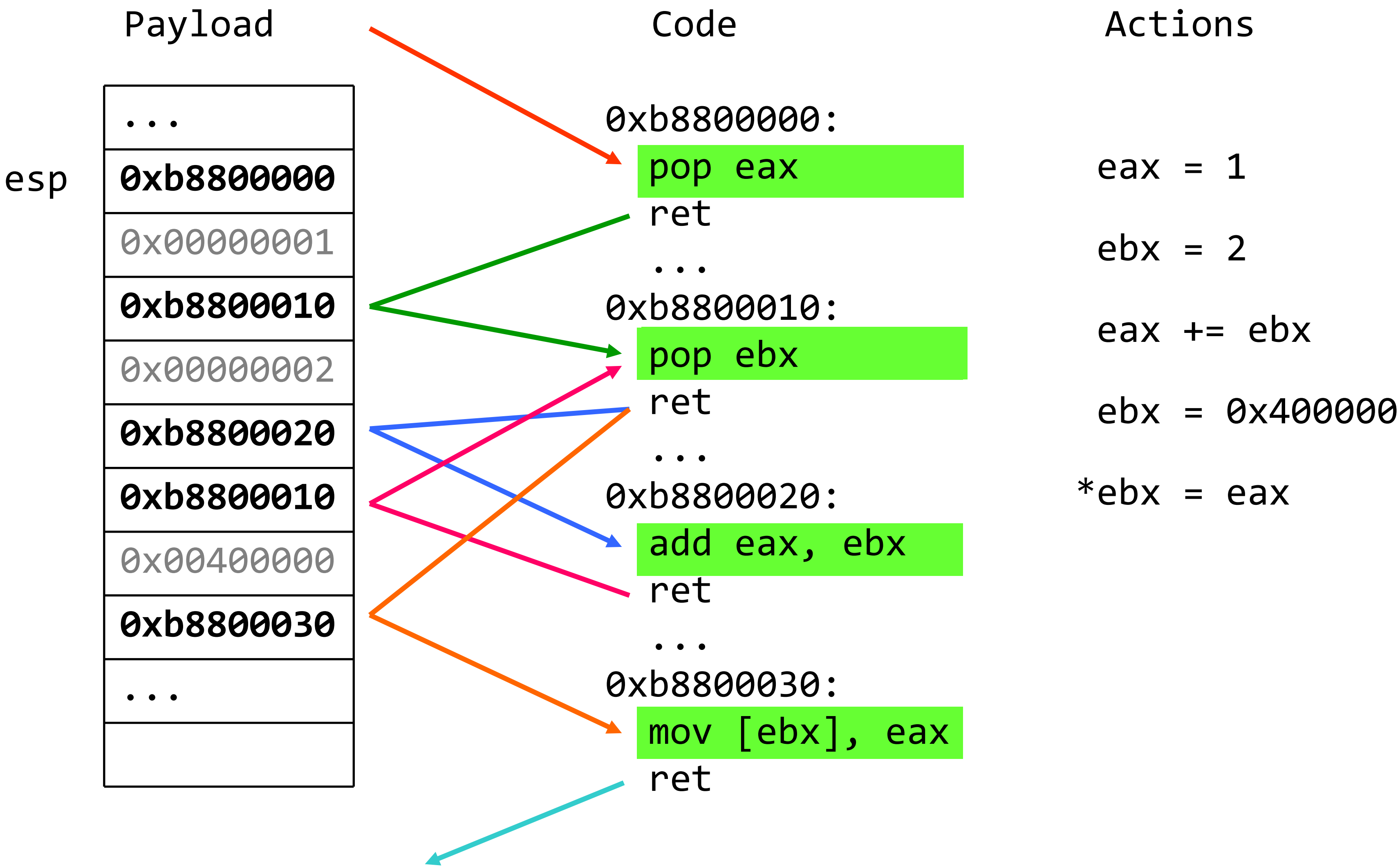
Pass function arguments through registers (IA-64)

```
0x0000000000400a82:   pop %rbx
0x0000000000400a83:   retq

0x00002aaaaac743d5:   mov %rbx,%rax   → &system
0x00002aaaaac743d8:   add $0xe0,%rsp
0x00002aaaaac743df:   pop %rbx
0x00002aaaaac743e0:   retq

0x00002aaaaac50bf4:   mov %rsp,%rdi   → /bin/sh
0x00002aaaaac50bf7:   callq *%eax
```

Return-oriented programming [Shacham '07]

Turing-complete return-oriented "shellcode"

Jump-oriented programming [Shacham '10]


Payload
Code
Actions
...
esp
0xb8800000
0x00000001
0xb8800010
0x00000002
0xb8800020
0xb8800010
0x00400000
0xb8800030
...
0xb8800000:
pop eax
ret
...
0xb8800010:
pop ebx
ret
...
0xb8800020:
add eax, ebx
ret
...
0xb8800030:
mov [ebx], eax
ret
eax = 1
ebx = 2
eax += ebx
ebx = 0x400000
*ebx = eax

Current State of ROP exploits

First-stage ROP code for bypassing DEP

- Allocate/set W+X memory (`VirtualAlloc`, `VirtualProtect`, …)
- Copy embedded shellcode into the newly allocated area
- Execute!

The complexity of ROP exploit code increases…

- New anti-ROP features in EMET
- ROP exploit mitigations in Windows 8

The embedded shellcode can be concealed

- ROP-based unpacker [Lu ’11]

How does ROP change attack detection?

Shellcode Detection Not Enough

```
push byte 0x7f
pop ecx
call 0x7
inc ecx
pop esi
add [esi+0xa],0xe0
xor [esi+ecx+0xb],cl
loop 0xe
xor [esi+ecx+0xb],cl
loop 0xe
xor [esi+ecx+0xb],cl
...
```

```
070072F7
00010104
070015BB
00001000
0700154D
070015BB
7FFE0300
07007FB2
070015BB
...
```

ROPscan

Detect ROP payloads in arbitrary inputs

Speculative execution

- Consider each 4-byte value as the beginning of a ROP payload

Code emulation

- Initialize VM with real (non-ASLR) process image(s)
- Speculatively execute gadgets based on valid memory addresses found in the scanned input

Identify the execution of multiple unique gadgets

... 41 41 41 F7 72 00 07 04 01 01 00 BB 15 00 ...

...	
070072F7	pop eax
070072F8	ret
...	

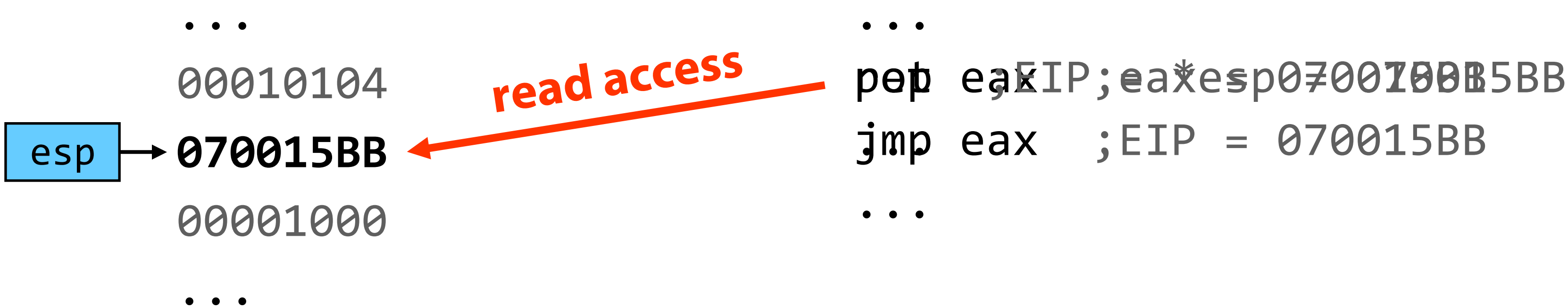
Runtime Heuristic

Benign inputs may trigger the execution of valid instruction sequences

Gadgets are peculiar:

End with an indirect control transfer instruction

that uses control data derived from the input buffer



Runtime Heuristic

An executed instruction sequence is a gadget if it

- **Reads** a valid destination address from the input payload
- **Transfers** control to that location

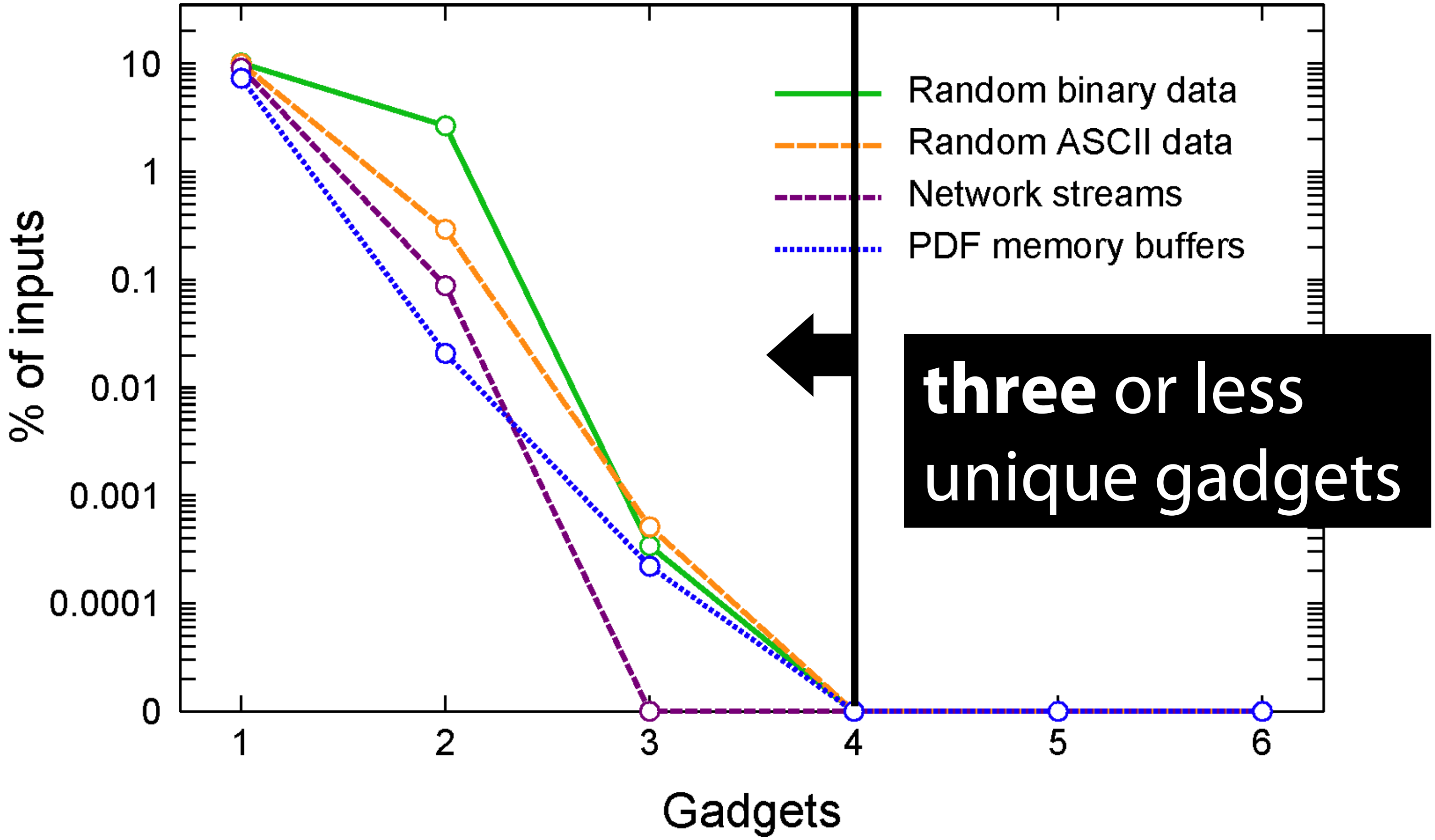
Gadget space for a typical process is very small

- A few MB vs. 4GB

Accidental gadget execution rare, but still possible!

- How often does it happen?
- To what extent?

Random Gadgets in Benign Inputs



```
starting exec - len: 0x1000 pos 0x11: PC: 0x773E274B
773e274b  D6                  salc
773e274c  83F8FF              cmp eax,0xffffffff
773e274f  75DB                jnz 0x773e272c
773e2751  EB77                jmp 0x773e27ca
773e27ca  83C8FF              or eax,0xffffffff
773e27cd  5F                  pop edi
773e27ce  5E                  pop esi
773e27cf  5B                  pop ebx
773e27d0  C9                  leave
773e27d1  C21000              retn 0x10
-------------- end of gdgt 773E274B
283b413f  FFCD                dec ebp
283b4141  FF8D4D00E985        dec [ebp-0x7a16ffb3]
283b4147  25CDFF8D4D          and eax,0x4d8dffcd
283b414c  58                  pop eax
283b414d  E948A0D1FF          jmp 0x280ce19a
280ce19a  E906FFFFFF          jmp 0x280ce0a5
280ce0a5  56                  push esi
280ce0a6  8BF1                mov esi,ecx
280ce0a8  8B06                mov eax,[esi]
280ce0aa  FF4804              dec [eax+0x4]
280ce0ad  8B4004              mov eax,[eax+0x4]
280ce0b0  85C0                test eax,eax
280ce0b2  7F1C                jnle 0x280ce0d0
280ce0d0  5E                  pop esi
280ce0d1  C3                  ret
-------------- end of gdgt 283B413F
282f2d6b  7402                jz 0x282f2d6f
282f2d6d  8919                mov [ecx],ebx
282f2d6f  885831              mov [eax+0x31],bl
282f2d72  C6403001            mov byte [eax+0x30],0x1
282f2d76  5B                  pop ebx
282f2d77  C3                  ret
-------------- end of gdgt 282F2D6B
283f0913  F4                  hlt
```

Sequence of random gadgets in benign input

```
starting exec - len: 0x128 pos 0x0: PC: 0x070072F7
070072f7  58              pop eax
070072f8  C3              ret
-------------- end of gdgt 070072F7
070015bb  59              pop ecx
070015bc  C3              ret
-------------- end of gdgt 070015BB
0700154d  8908            mov [eax],ecx
0700154f  C3              ret
-------------- end of gdgt 0700154D
070015bb  59              pop ecx
070015bc  C3              ret
-------------- end of gdgt 070015BB
07007fb2  8B01            mov eax,[ecx]
07007fb4  C3              ret
-------------- end of gdgt 07007FB2
070015bb  59              pop ecx
070015bc  C3              ret
-------------- end of gdgt 070015BB
0700a8ac  8901            mov [ecx],eax
0700a8ae  33C0            xor eax,eax
0700a8b0  C3              ret
-------------- end of gdgt 0700A8AC
070015bb  59              pop ecx
070015bc  C3              ret
-------------- end of gdgt 070015BB
0700a8ac  8901            mov [ecx],eax
0700a8ae  33C0            xor eax,eax
0700a8b0  C3              ret
-------------- end of gdgt 0700A8AC
070072f7  58              pop eax
070072f8  C3              ret
-------------- end of gdgt 070072F7
070052e2  FF10            call [eax]
0700d731  8B45DC          mov eax,[ebp-0x24]
0700d734  C3              ret
                          ...
```

Real ROP code

benign inputs ROP exploits

3 < detection threshold < **?**

Number of unique gadgets in the same execution chain

Tested ROP Exploits/Generic ROP Payloads

benign inputs

ROP exploits

3 < detection threshold < 8

Exploit/Payload	EDB-ID	Tested Platform	Gadget Space	Executed Gadgets	Unique Gadgets
Adobe Reader v9.3.0	16670	WinXP SP3	17.7MB	47	8
Adobe Reader v9.3.0	16687	WinXP SP3	17.7MB	60	12
Adobe Reader v9.3.4	16619	Win7 SP1	864KB	33	10
Adobe Reader v[illegible].4	1666[illegible]	[illegible]	17.7MB	60	12
Winamp v5.572	14068	Win7 SP1	[illegible]MB	126	21
Integard Pro v2.2.0	1501[illegible]	[illegible]	724KB	165	16
Mplayer Lite r33064	17124	Win7 SP1	6.4MB	179	16
All to MP3 Conv. v2.0	17252	WinXP SP3	9.4MB	388	16
msvcr71.dll (sayonara)	-	Win7 SP1	228KB	11	9
msvcr71.dll (corelan)	-	Win7 SP1	228KB	12	11
mscorie.dll (sayonara)	-	Win7 SP1	28KB	9	9
mfc71u.dll (corelan)	-	Win7 SP1	872KB	15	10

Use Cases

Network-level detection

- Incorporated ROPscan in Nemu, a shellcode detector
- Detection of both layers: ROP code and legacy shellcode

PDF scanning

- Incorporated ROPscan in MDScan, a PDF file scanner
- Scan newly allocated memory by the JS interpreter

Easy to incorporate into existing detectors

- Scan arbitrary data
- At least an order of magnitude faster than shellcode detection

How does ROP change system protection?

ASLR is not Fully Adopted

Executable programs in Ubuntu Linux

- Only 66 out of 1,298 binaries in /usr/bin [SAB11]

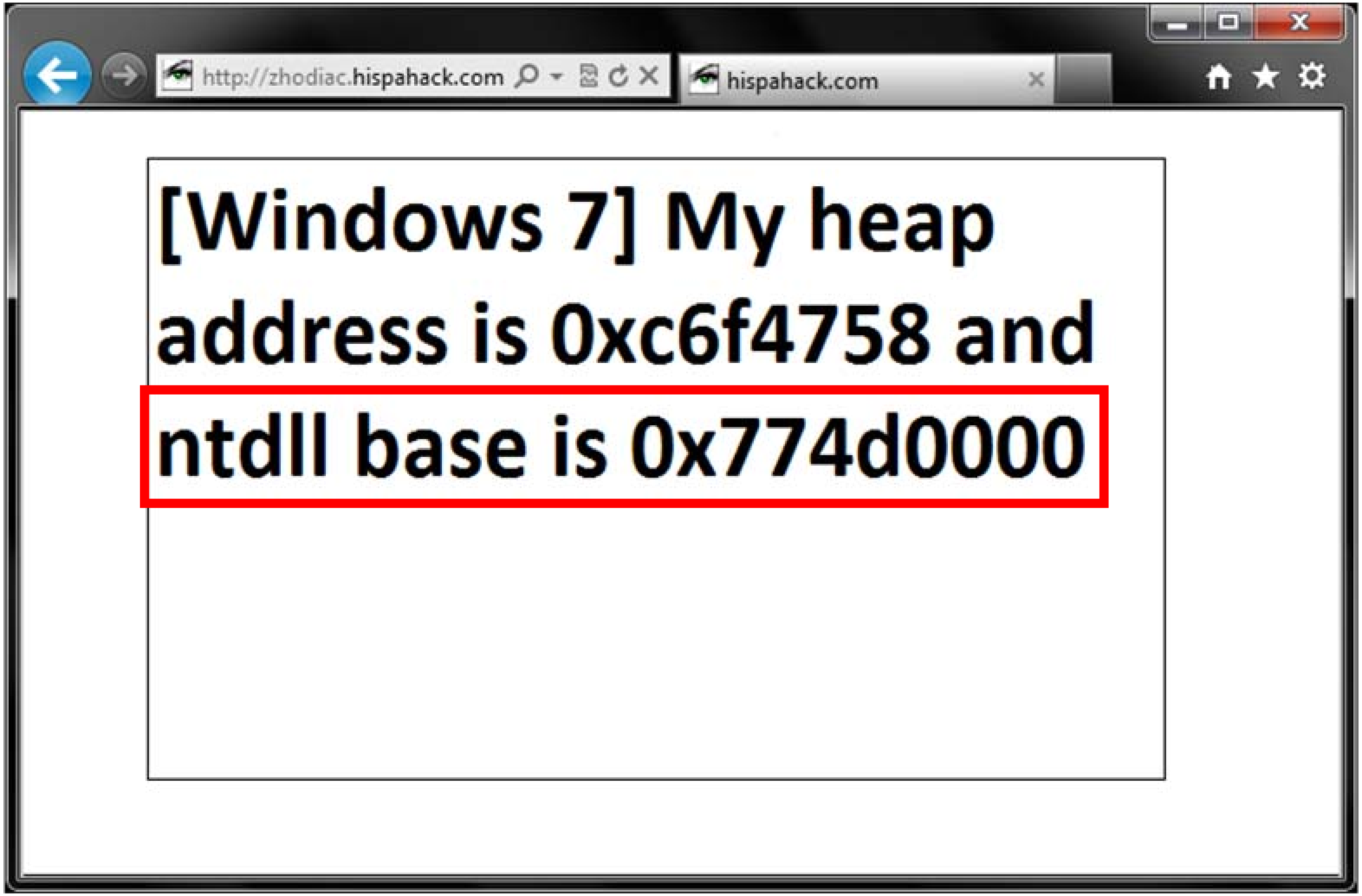
Popular third-party Windows applications

- Only 2 out of 16 [Pop10]

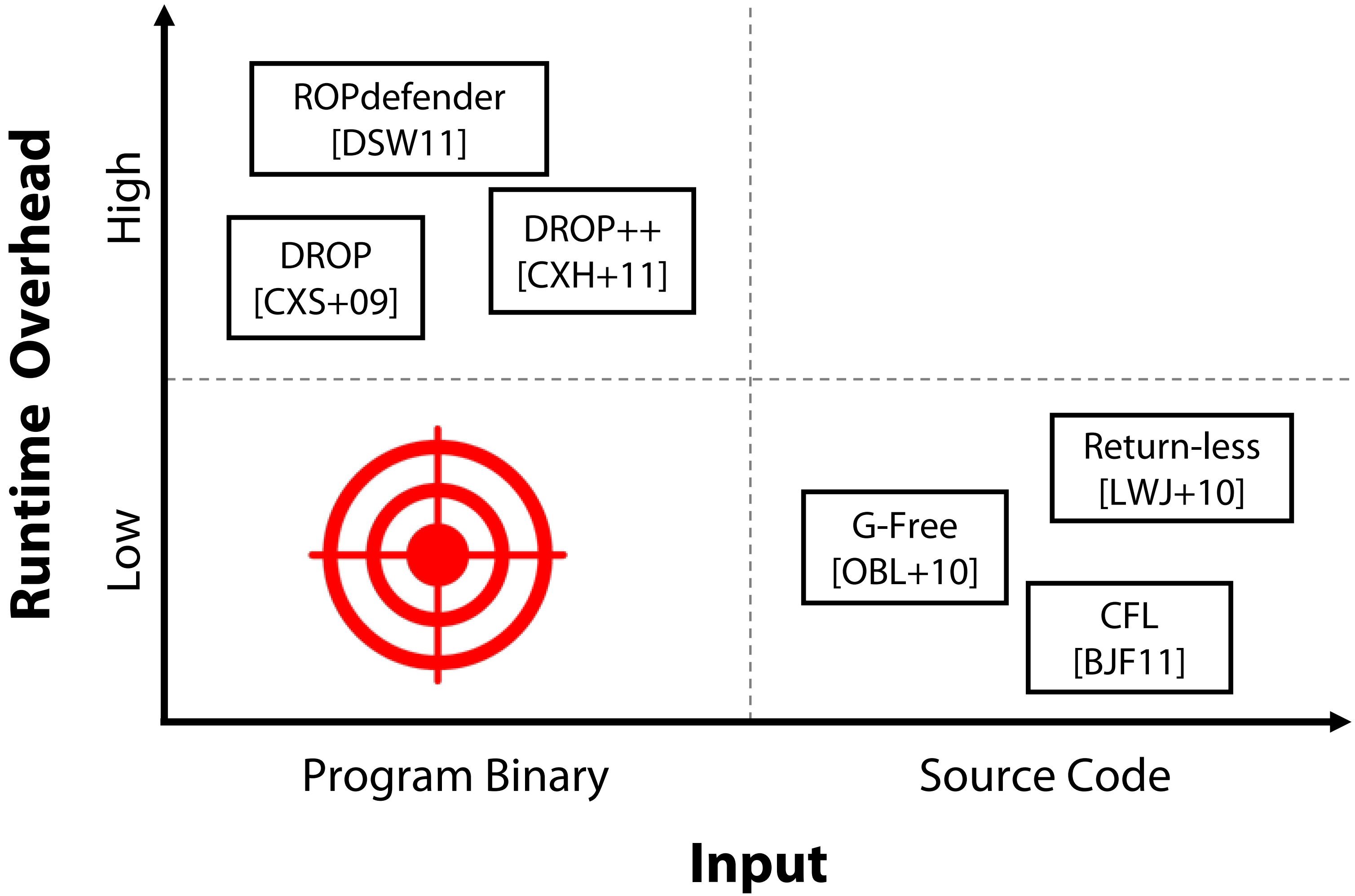
Even applications that enable ASLR sometimes have statically mapped DLLs

- EMET forced randomization

Information Leaks Break ASLR [Ser12]



ROP Defenses



In-Place Code Randomization

Software diversification

Applicable on third-party applications

Zero (non-measurable) performance overhead

Why In-Place?

Randomization usually changes the code size

- Need to update the control-flow graph (CFG)

Accurate disassembly of stripped binaries is hard

- Incomplete CFG (data vs. code)
- Code resize not an option

Must randomize in-place!

Code Transformations

Instruction Substitution

Instruction Reordering

- Intra Basic Block
- Register Preservation Code

Register Reassignment

Instruction Substitution

```
    add [edx],edi
          ret
B0 01 3A C3 8D 45 80 50 68
mov al,0x1
     cmp al,bl
          lea eax,[ebp-0x80]
```

```
    add [eax],edi
          fmul [ebp+0x68508045]
B0 01 38 D8 8D 45 80 50 68
mov al,0x1
     cmp bl,al
          lea eax,[ebp-0x80]
```

Instruction Reordering (Intra Basic Block)

```
8B 41 10   mov eax,[ecx+0x10]
53         push ebx
8B 59 0C   mov ebx,[ecx+0xC]
3B C3      cmp eax,ebx
89 41 08   mov [ecx+0x8],eax
7E 4E      jle 0x5c
```

```
59         push ebx
0C 3B      or al,0x3B
C3         ret
```

Instruction Reordering (Intra Basic Block)

```
8B 41 10   mov eax,[ecx+0x10]
53         push ebx
8B 59 0C   mov ebx,[ecx+0xC]
3B C3      cmp eax,ebx
89 41 08   mov [ecx+0x8],eax
7E 4E      jle 0x5c
```

```
41         inc ecx

10 89 41 08 3B C3
  adc [ecx-0x3CC4F7BF],cl
```

Register Preservation Code Reordering

Prolog

```
push ebx
push esi
mov  ebx,ecx
push edi
mov  esi,edx
     ...
```

```
push edi
push ebx
push esi
mov  ebx,ecx
mov  esi,edx
     ...
```

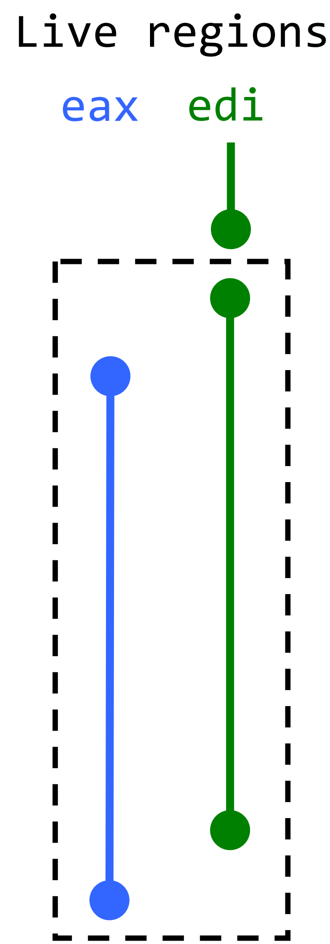
Epilog

```
pop  edi
pop  esi
pop  ebx
ret
```

```
pop  esi
pop  ebx
pop  edi
ret
```

Register Reassignment

```
function:
 push esi
 push edi
 mov  edi,[ebp+0x8]
 mov  eax,[edi+0x14]
 test eax,eax
 jz   0x4A80640B
 mov  ebx,[ebp+0x10]
 push ebx
 lea  ecx,[ebp-0x4]
 push ecx
 push edi
 call eax
 ...
```




```
function:
 push esi
 push edi
 mov  eax,[ebp+0x8]
 mov  edi,[edi+0x14]
 test edi,edi
 jz   0x4A80640B
 mov  ebx,[ebp+0x10]
 push ebx
 lea  ecx,[ebp-0x4]
 push ecx
 push eax
 call edi
 ...
```

Implementation: Orp

Focused on the Windows platform

- Could be integrated in Microsoft's EMET

CFG extraction using IDA Pro

- Implicitly used registers
- Liveness analysis (intra and inter-function)
- Register categorization (arg., preserved, …)
- Randomization
- Binary rewriting (relocations fixing, …)

Evaluation

Correctness and performance

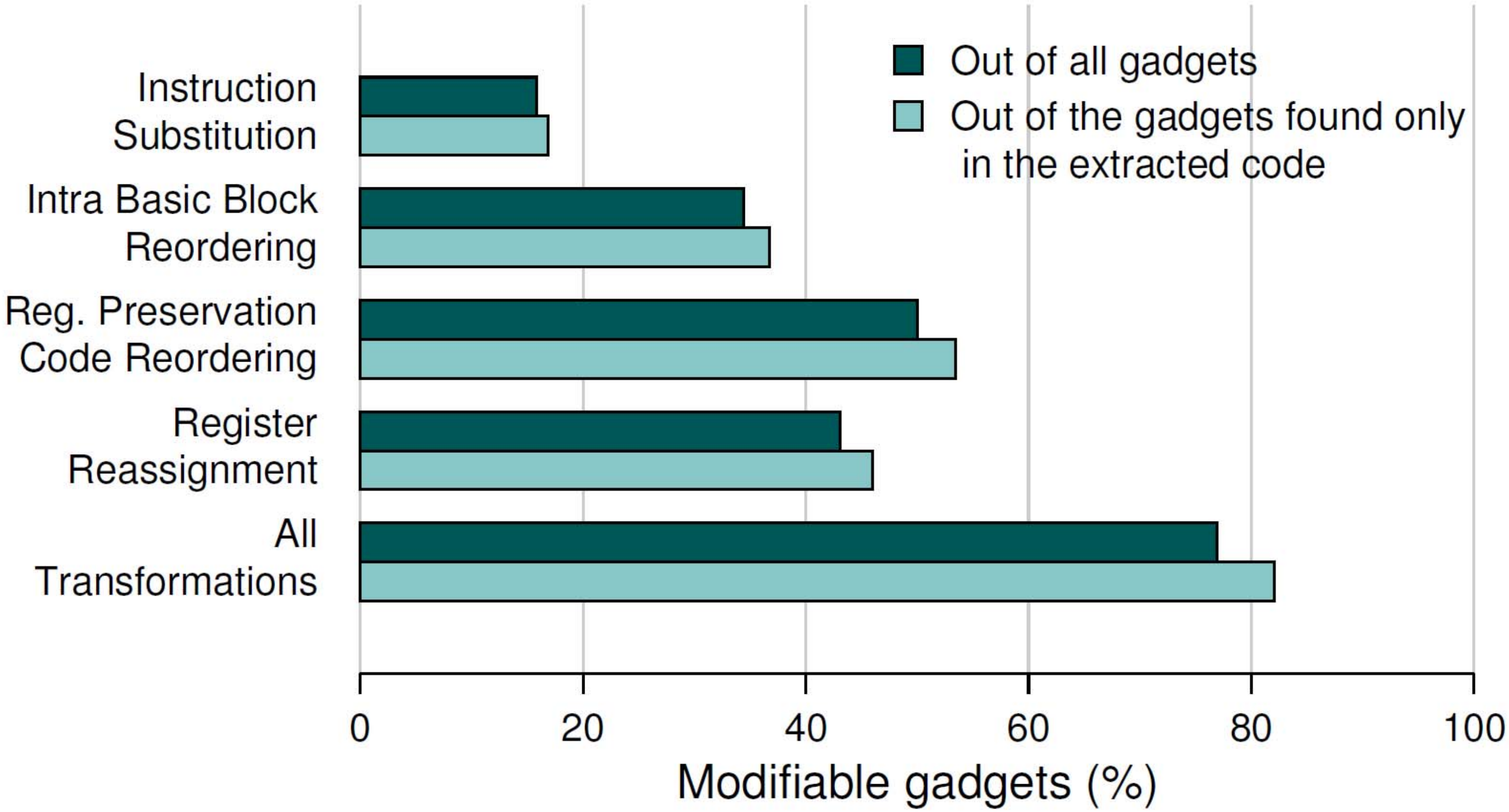
- Used Wine's extensive test suite with randomized versions of Windows DLLs

Randomization Coverage

Effectiveness against real-world exploits

Robustness against ROP Compilers

Randomization Coverage



Dataset: 5,235 PE files (~0.5GB code) from Windows, Firefox, iTunes, Reader

Real-World Exploits

Exploit/Reusable Payload	Unique Gadgets	Modifiable	Combinations
Adobe Reader v9.3.4	**11**	**6**	287
Integard Pro v2.2.0	**16**	**10**	322K
Mplayer Lite r33064	**18**	**7**	1.1M
msvcr71.dll (While Phosphorus)	**14**	**9**	3.3M
msvcr71.dll (Corelan)	**16**	**8**	1.7M
mscorie.dll (White Phosphorus)	**10**	**4**	25K
mfc71u.dll (Corelan)	**11**	**6**	170K

Modifiable gadgets were not always directly replaceable!

ROP Compilers

Is it possible to create a randomization-resistant ROP payload?

Using only the remaining non-randomized gadgets

Tested two ROP payload construction tools

mona.py: constructs DEP+ASLR bypassing code

Allocate a WX buffer, copy shellcode, and jump to it

Q: state-of-the-art ROP compiler [SAB11]

Designed to be robust against small gadget sets

ROP Compiler Results

Non-ASLR Code Base	Mona		Q	
	Original	Rand.	Original	Rand.
Adobe Reader v9.3.4	✓	✗	✓	✗
Integard Pro v2.2.0	✗	✗	✓	✗
Mplayer Lite r33064	✓	✗	✓	✗
msvcr71.dll	✗	✗	✓	✗
mscorie.dll	✗	✗	✗	✗
mfc71u.dll	✓	✗	✓	✗

Both tools failed to construct ROP payloads using non-randomized code!

Summary

Return-Oriented Programming is increasingly used in real-world exploits

ROP payloads can be detected using speculative code execution

In-place code randomization prevents real exploits and ROP compilers

Get the code (Python)
http://nsl.cs.columbia.edu/projects/orp

References

[Ser12] Fermin J. Serna. The case of the perfect info leak, 2012.
http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[SAB11] Edward J. Schwartz et al. Q: exploit hardening made easy. USENIX Security, 2011.

[Pop10] Alin Rad Pop. Dep/aslr implementation progress in popular third-party windows applications, 2010.
http: //secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). CCS, 2007.

[CDD+10] Stephen Checkoway et al. Return-oriented programming without returns. CCS, 2010

[BJFL11] Tyler Bletsch et al. Jump-oriented programming: a new class of code-reuse attack. ASIACCS, 2011.

[LZWG11b] Kangjie Lu et al. Packed, printable, and polymorphic return-oriented programming, RAID, 2011.

[DSW11] Lucas Davi et al. Ropdefender: a detection tool to defend against return-oriented programming attacks. ASIACCS, 2011

[CXS+09] Ping Chen et al. Drop: Detecting return-oriented programming malicious code, ICISS, 2009.

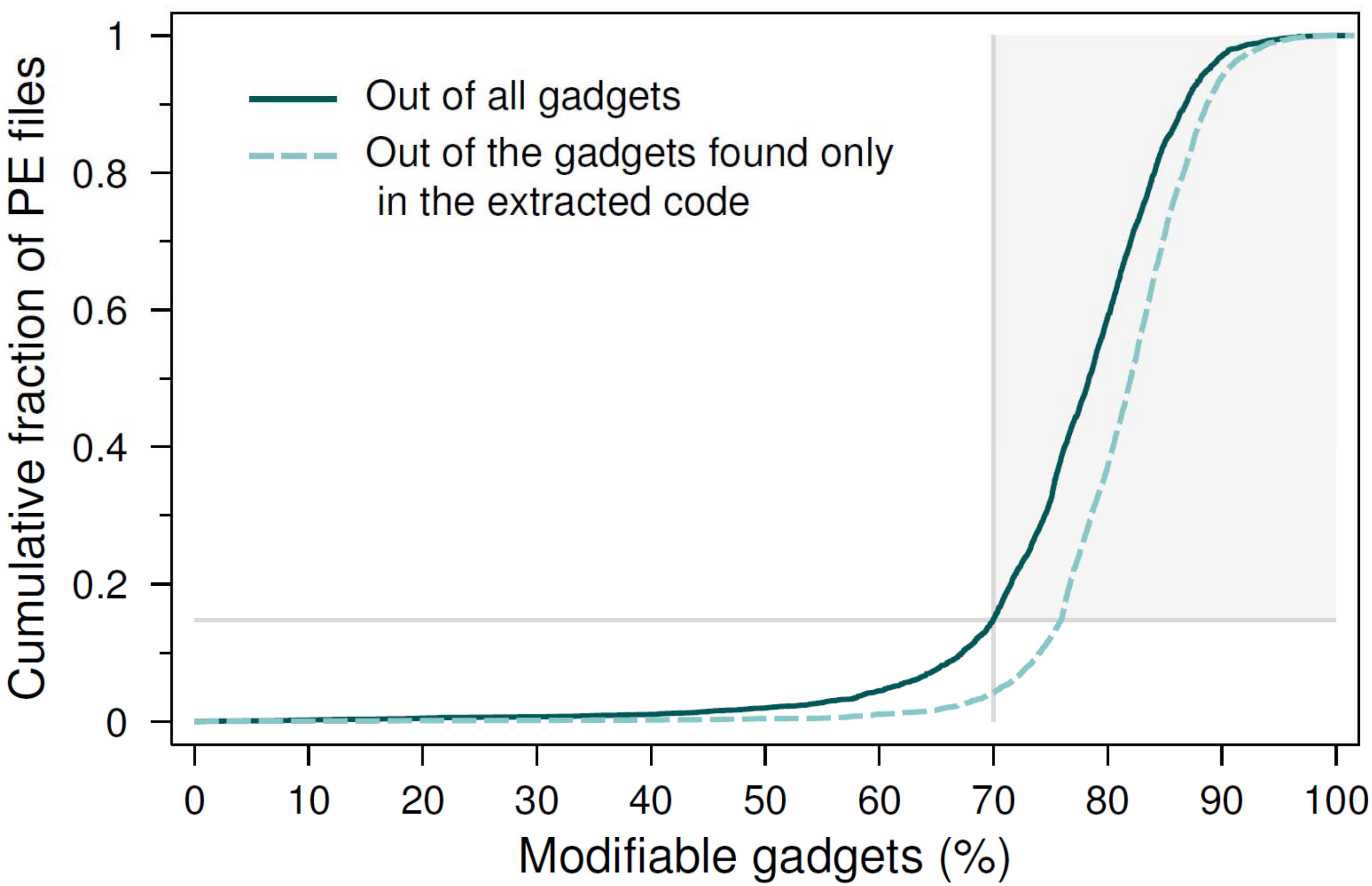
[CXH+11] Ping Chen et al. Efficient detection of the return-oriented programming malicious code, ICISS, 2011.

[OBL+10] Kaan Onarlioglu et al. G-free: defeating return-oriented programming through gadget-less binaries. ACSAC, 2010.

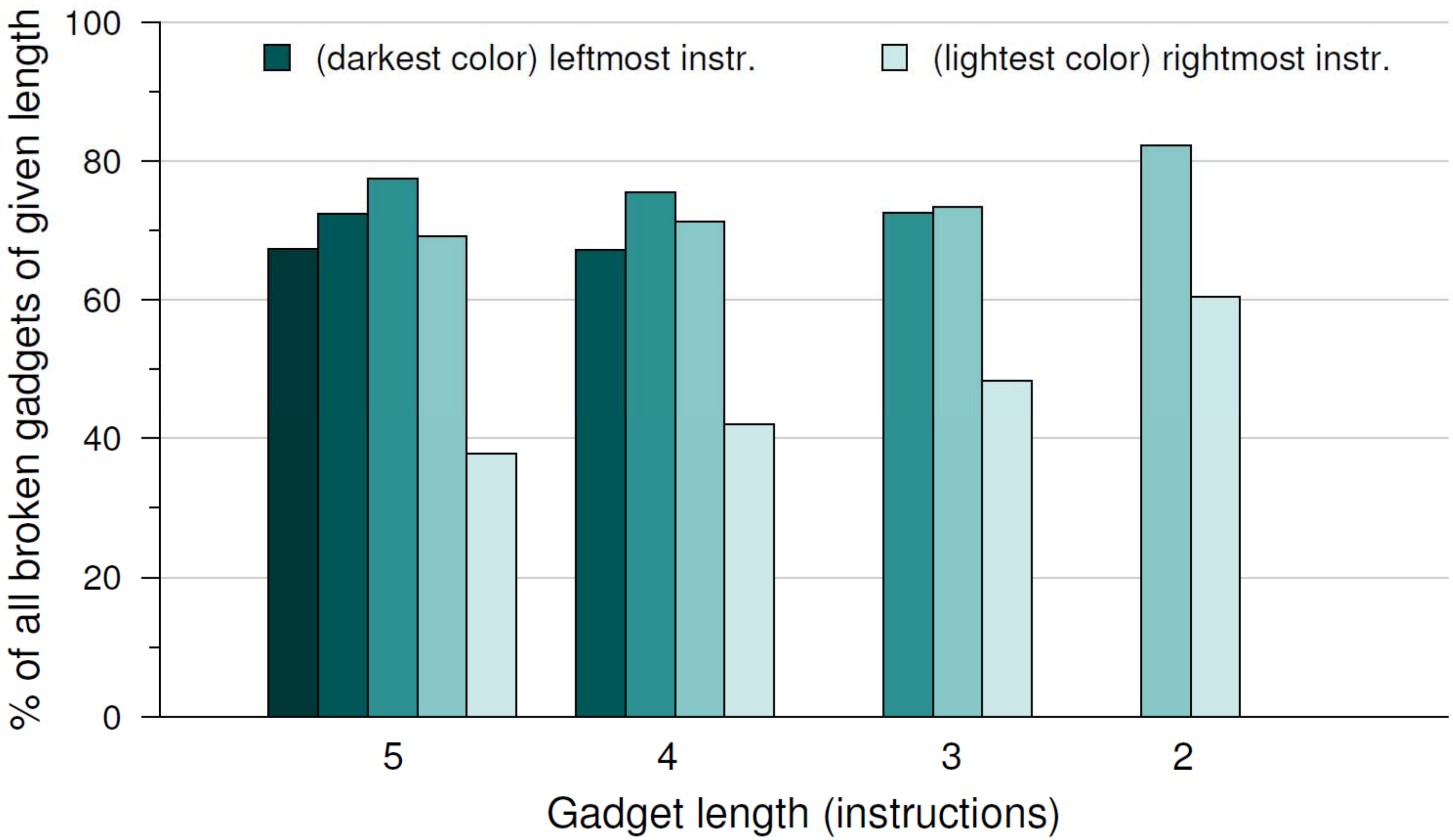
[LWJ+10] Jinku Li et al. Defeating return-oriented rootkits with “return-less” kernels. EuroSys, 2010.

[BJF11] Tyler Bletsch et al. Mitigating code-reuse attacks with control-flow locking. ACSAC, 2011.

Modifiable Gadgets



Impact on Broken Gadgets' Instructions



Randomization Entropy for Broken Gadgets