CE 874 - Secure Software Systems

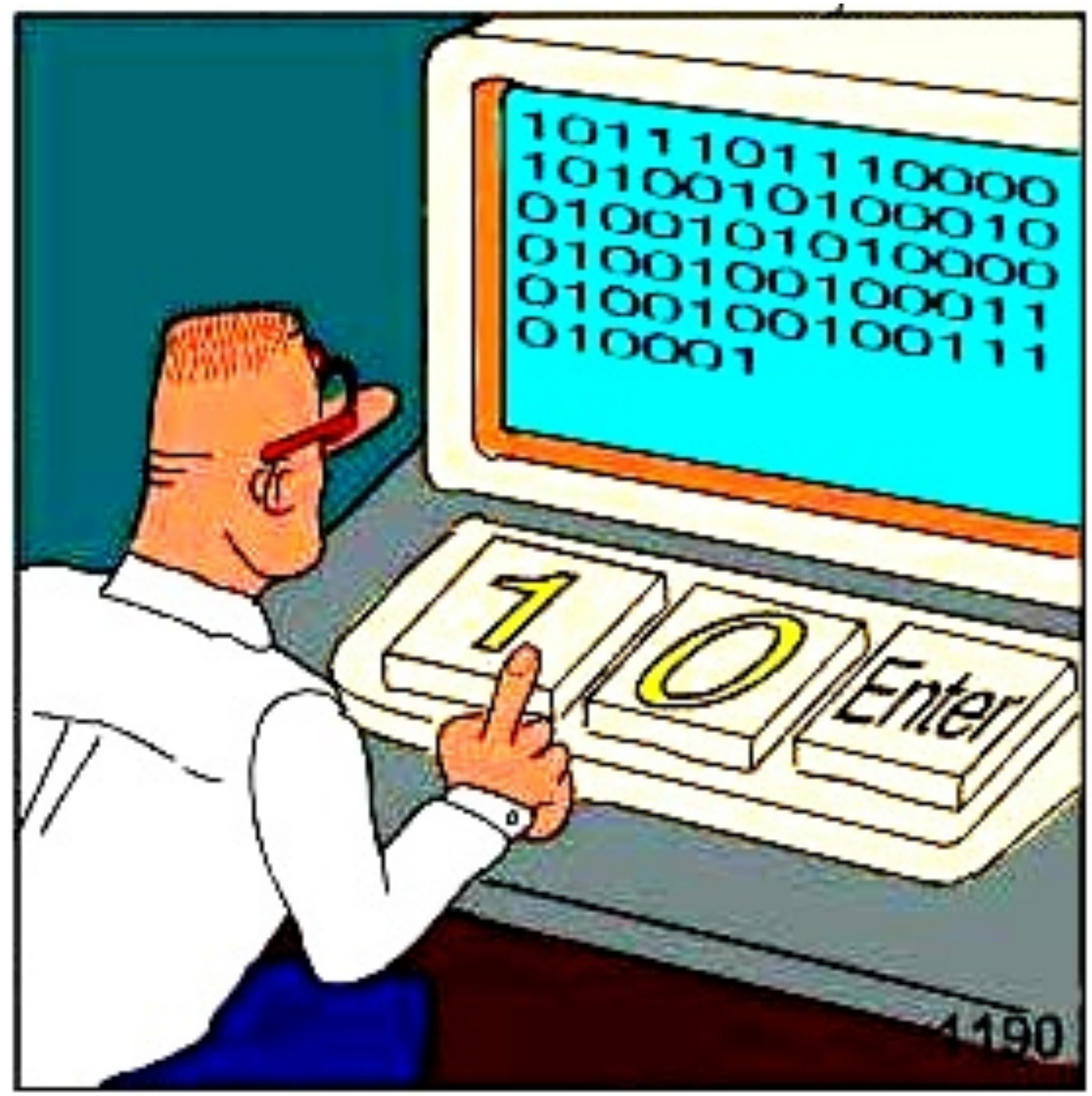
Reassembly

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. Reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Run-Time protection/enforcement

- In many instances we only have access to the binary
- How do we analyze the binary for vulnerabilities?
- How do we protect the binary from exploitation?
- This would be our topic for the next few lectures



REAL Programmers code in BINARY.

Why Binary Code?

- Access to the source code often is not possible:
  - Proprietary software packages
  - Stripped executables
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- Binary code is the only authoritative version of the program
  - Changes occurring in the compile, optimize and link steps can create non-trivial semantic differences from the source and binary
- Worms and viruses are rarely provided with source code

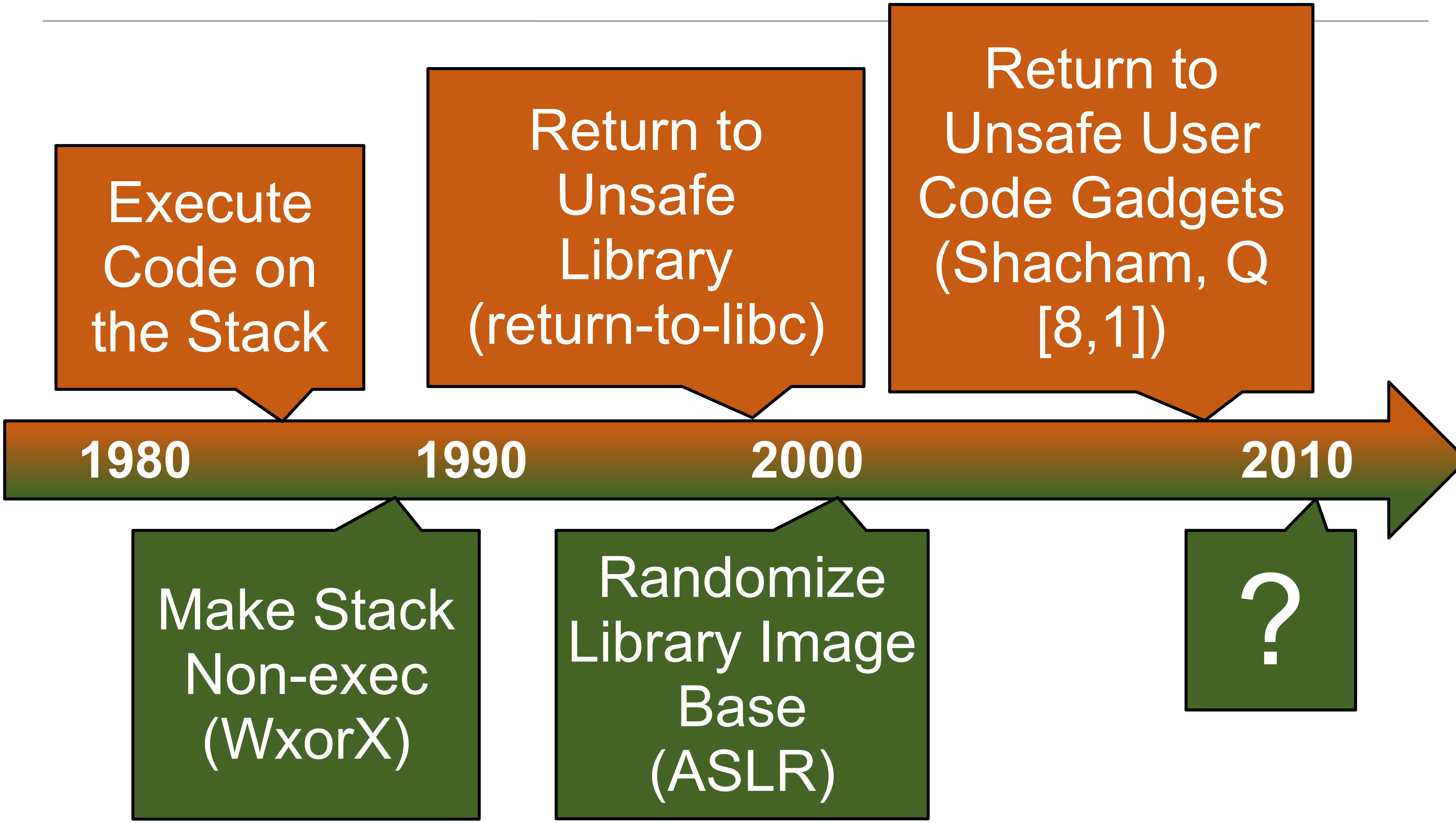
Goals for the day

- Last time we discussed binary analysis
  - Binary Analysis
  - Binary patching/rewriting
  - Binary instrumentation
    - Very short discussion of CFI
    - Taint analysis
- Today we want to discuss:
  - another use case for binary patching
  - why reassembly (i.e. binary re-writing) is hard?

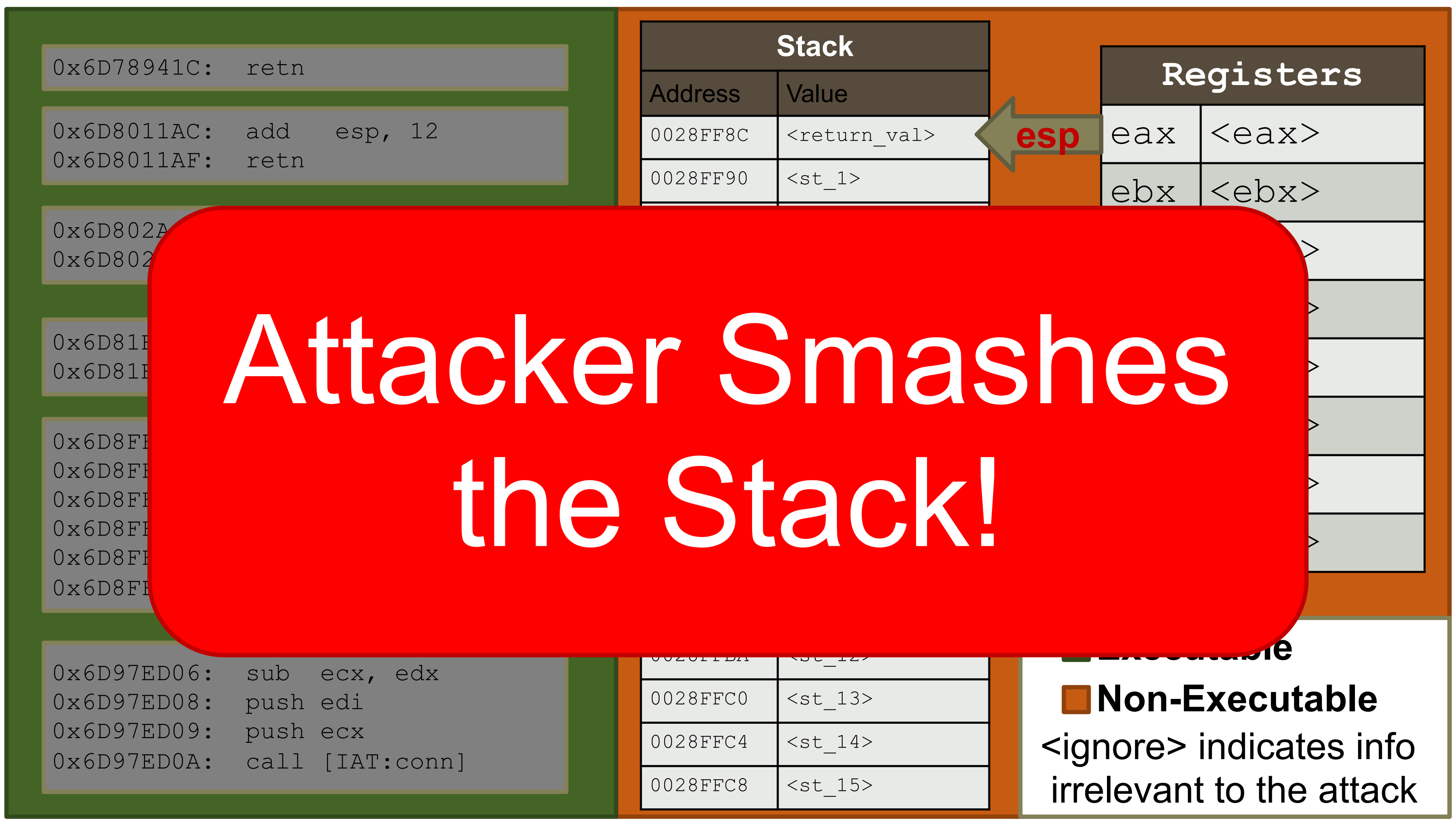
Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code

R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin.CCS 2012

Attacks Timeline

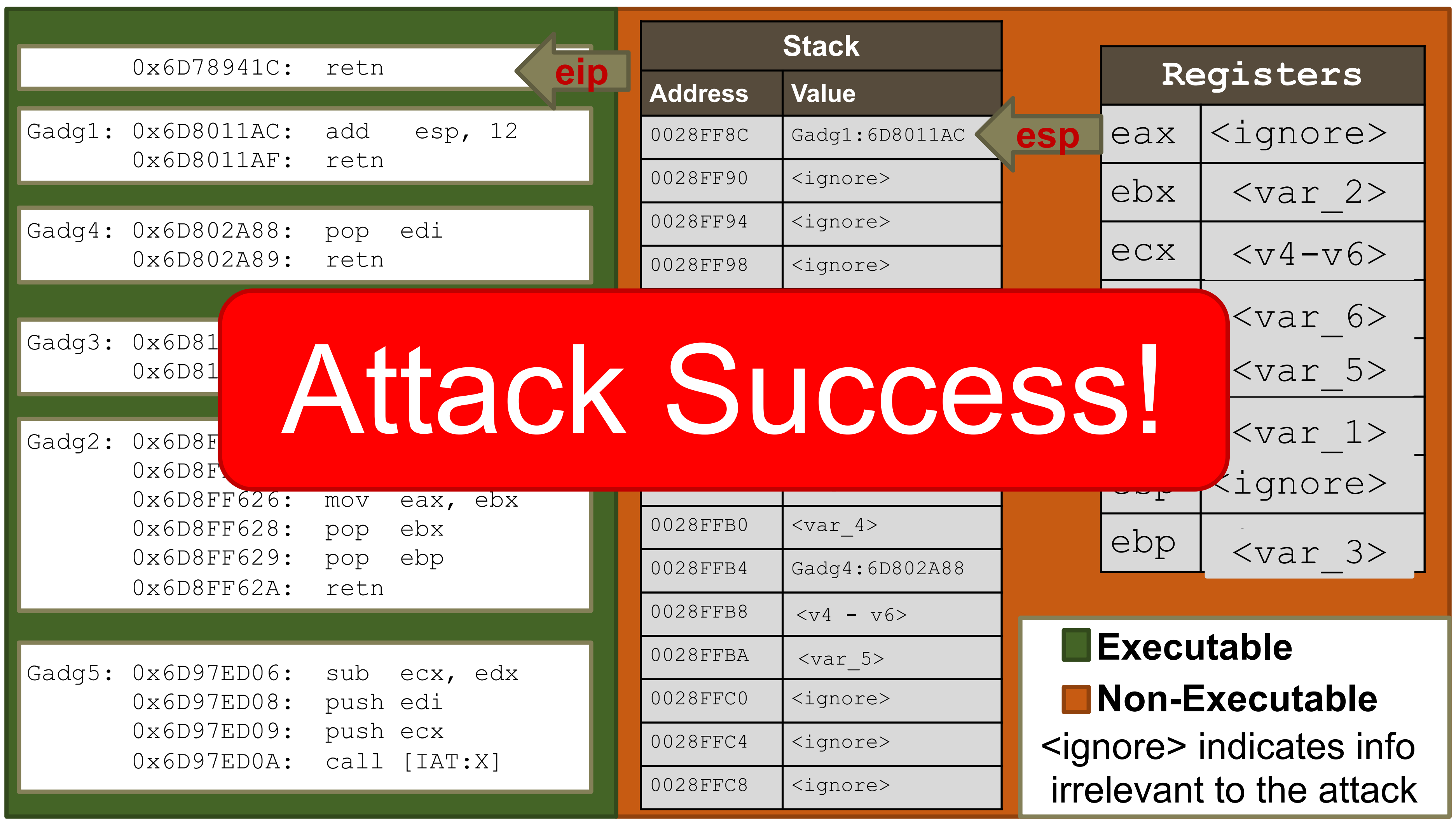


RoP Attack



0x6D78941C: retn
0x6D8011AC: add esp, 12
0x6D8011AF: retn
0x6D97ED06: sub ecx, edx
0x6D97ED08: push edi
0x6D97ED09: push ecx
0x6D97ED0A: call [IAT:conn]
Stack
Address
Value
0028FF8C
<return_val>
0028FF90
<st_1>
0028FFC0
<st_13>
0028FFC4
<st_14>
0028FFC8
<st_15>
esp
Registers
eax
<eax>
ebx
<ebx>
Non-Executable
<ignore> indicates info irrelevant to the attack
Attacker Smashes the Stack!

RoP Attack



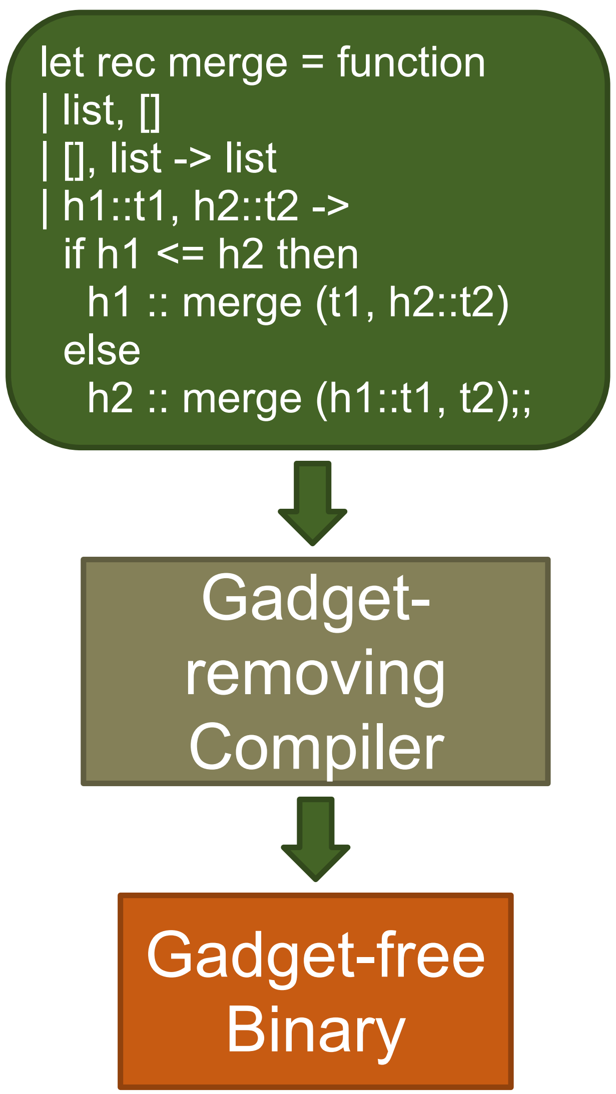
Action: Store `<var_5>` in `edi` for later use

RoP Defense Strategy

- RoP is one example of a broad class of attacks that require attackers to know or predict the location of binary features

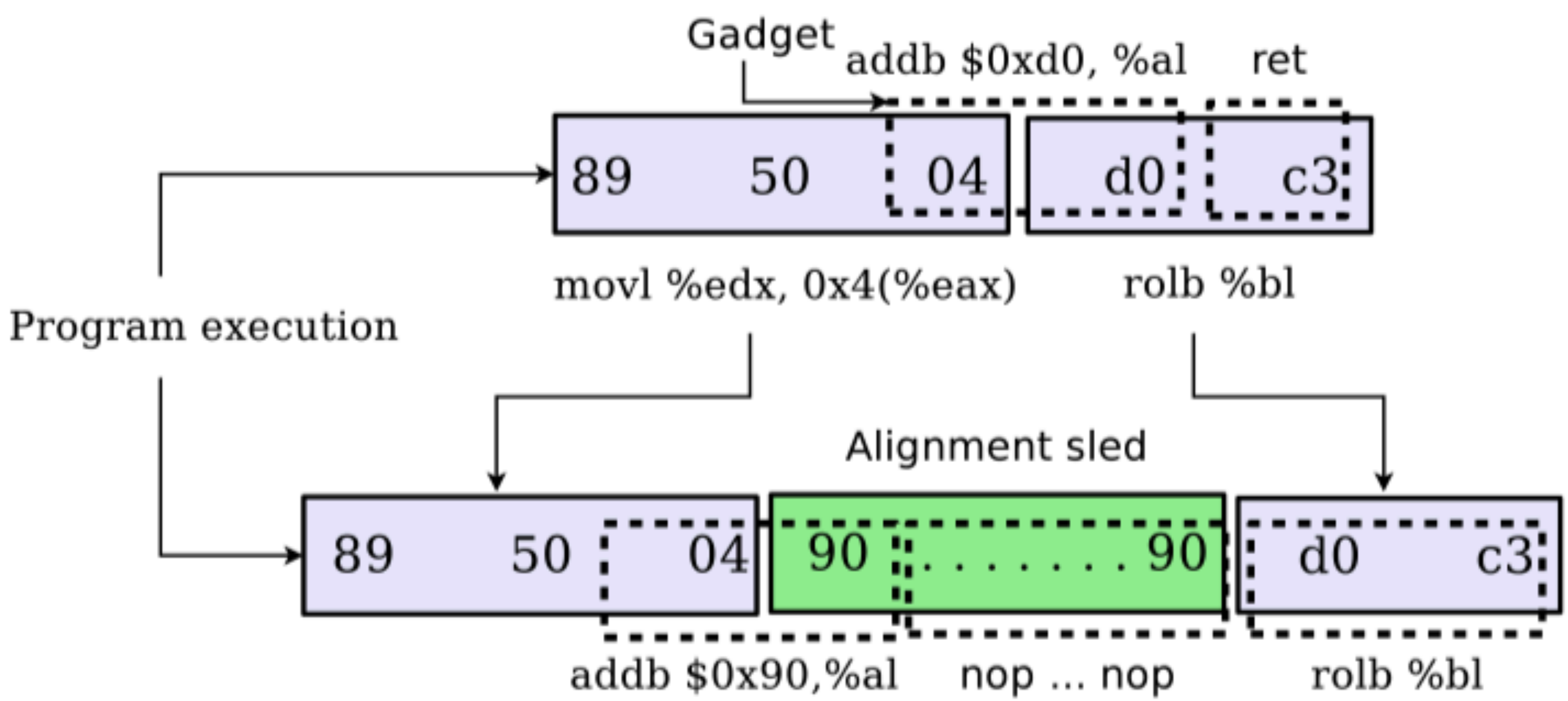
<u>Defense Goal</u>
Frustrate such attacks by randomizing feature space or removing features

RoP Defenses: Compiler-based

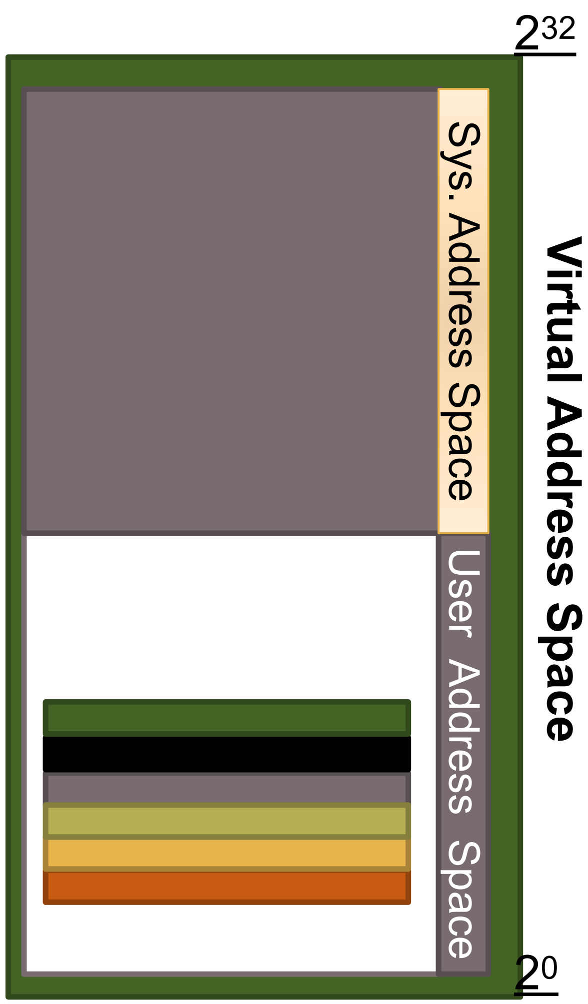


- Control the machine code instructions used in compilation (Gfree [2] and Returnless [3])
  - Use no return instructions
  - Avoid gadget opcodes
- Hardens against RoP
- Requires code producer cooperation
  - Legacy binaries unsupported

GFree Alignment Sled

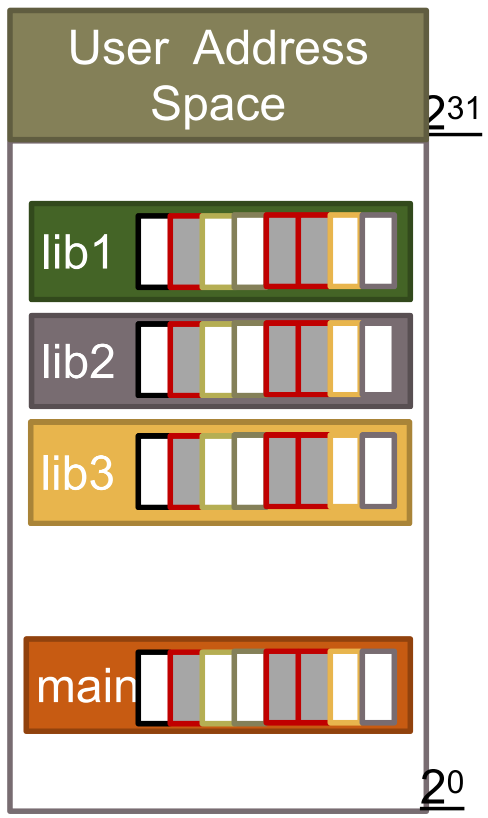


RoP Defenses: ASLR



- ASLR randomizes the image base of each library
  - Gadgets hard to predict
  - Brute force attacks still possible [4]

RoP Defenses: IPR / ILR



- Instruction Location Randomization (ILR) [5]
  - Randomize each instruction address using a virtual machine
  - Increases search space
  - Cannot randomize all instructions
  - High overhead due to VM (13%)
- In-place Randomization (IPR) [6]
  - Modify assembly to break known gadgets
  - Breaks 80% of gadgets on average
  - Cannot remove all gadgets
  - Preserves gadget semantics
  - Deployment issues

Our Goal

- Self-randomizing COTS binary w/o source code
  - Low runtime overhead
  - Complete gadget removal
  - Flexible deployment (copies randomize themselves)
  - No code producer cooperation

Challenge: Binary Randomization w/o metadata

- Relocation information, debug tables and symbol stores not always available
  - Reverse engineering concerns
- Perfect static disassembly without metadata is provably undecidable
  - Best disassemblers make mistakes (IDA Pro)

Program	Instruction Count	IDA Pro Errors
mfc42.dll	355906	1216
mplayerc.exe	830407	474
vmware.exe	364421	183

 [Wartell'12]

Unaligned Instructions

- Disassemble this hex sequence
  - Undecidable problem

```
FF E0 5B 5D C3 0F
88 52 0F 84 EC 8B
```

Valid Disassembly	
FF E0	jmp eax
5B	pop ebx
5D	pop ebp
C3	retn
0F 88 52 0F 84 EC	jcc
8B …	mov

Valid Disassembly	
FF E0	jmp eax
5B	pop ebx
5D	pop ebp
C3	retn
0F	db (1)
88 52 0F 84 EC	mov
8B …	mov

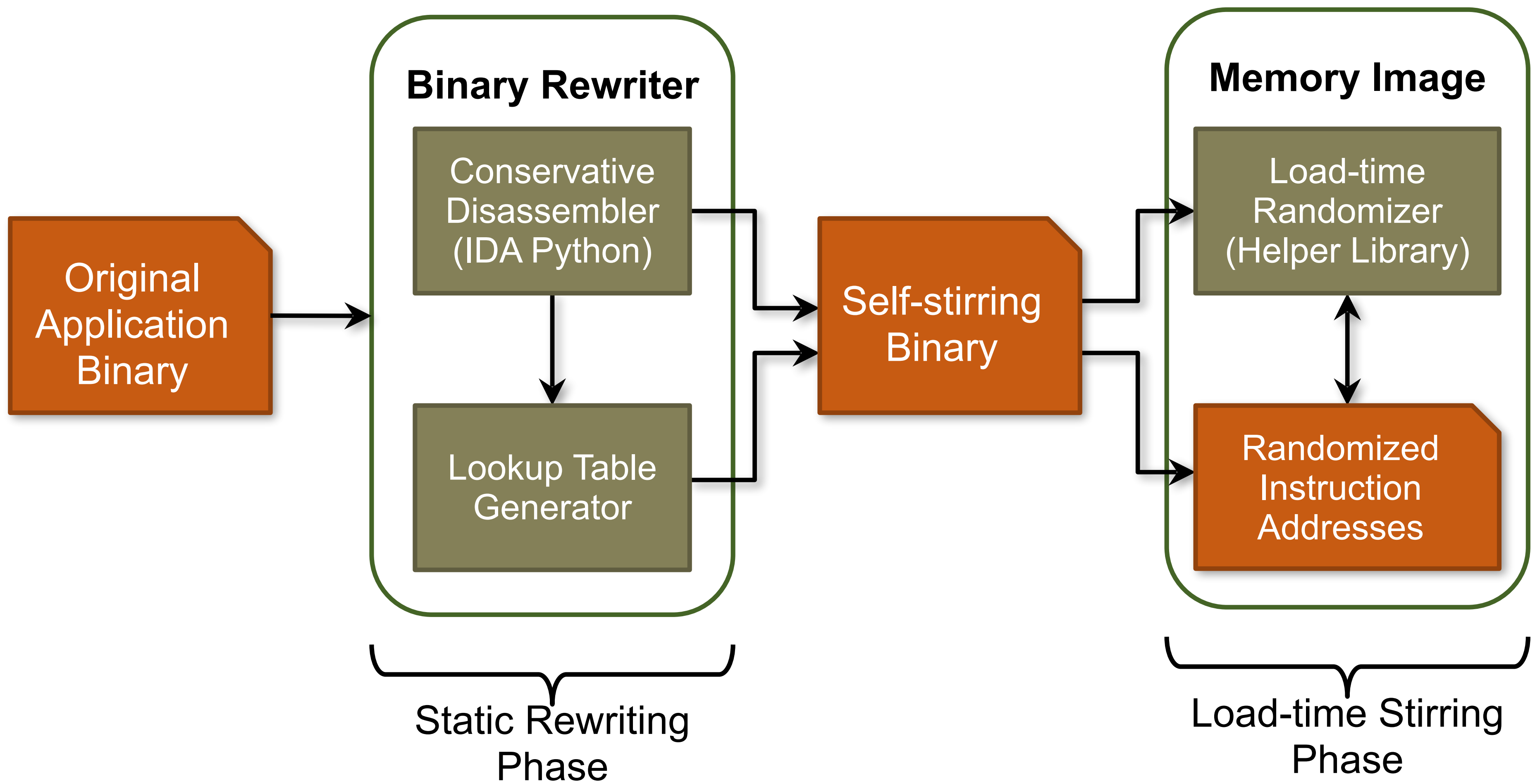
Valid Disassembly	
FF E0	jmp eax
5B	pop ebx
5D	pop ebp
C3	retn
0F 88	db (2)
52	push edx
0F 84 EC 8B …	jcc

 [Wartell'12]

Our Solution: STIR (Self-Transforming Instruction Relocation)

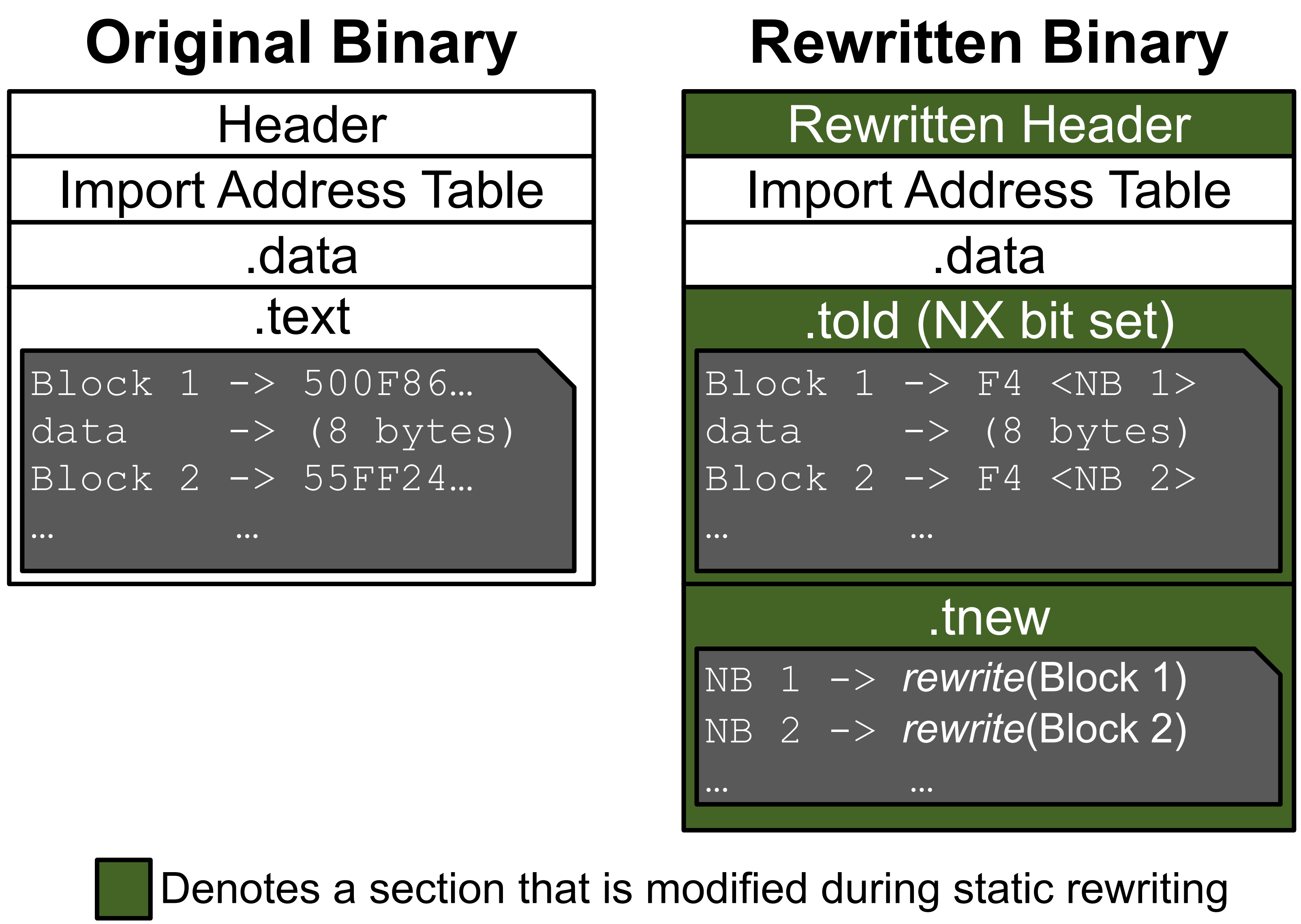
- Statically rewrite legacy binaries to re-randomize at load-time
  - Greatly increases search space against brute force attacks
  - Introduces no deployment issues
  - Tested on 100+ Windows and Linux binaries
  - 99.99% gadget reduction on average
  - 1.6% overhead on average
  - 37% process size increase on average

STIR Architecture

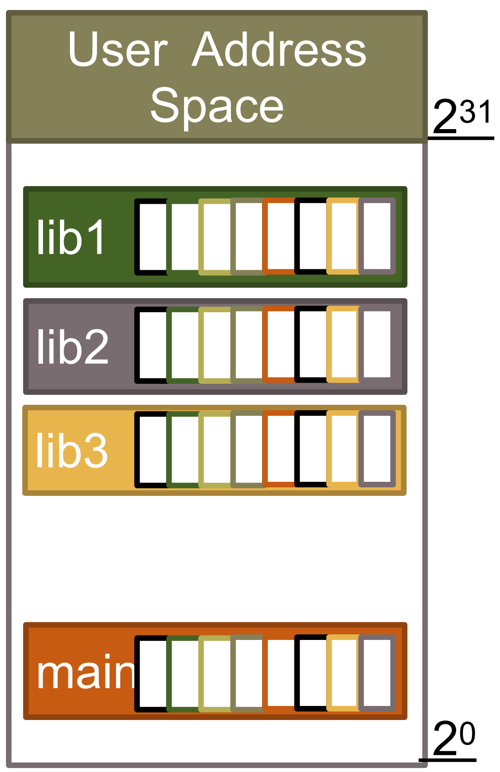


 [Wartell'12]

Static Rewriting



Load-time Stirring



- When binary is loaded:
  - Initializer randomizes .tnew layout
  - Lookup table pointers are updated
  - Execution is passed to the new start address

Computed Jump Preservation

Original Instruction:

eax = 0x411A40

.**text**:0040CC9B	FF DO	call eax

Original Possible Target:

.**text**:00411A40	5B	pop ebp

Rewritten Instructions:

eax = 0x411A40
eax = 0x534AB9

.**tnew**:0052A1CB	80 38 F4	cmp byte ptr [eax], F4h
.**tnew**:0052A1CE	0F 44 40 01	cmovz eax, [eax+1]
.**tnew**:0052A1D2	FF D0	call eax

Rewritten Jump Table:

.**told**:00411A40	F4 B9 4A 53 00	F4 dw 0x534AB9

Rewritten Target:

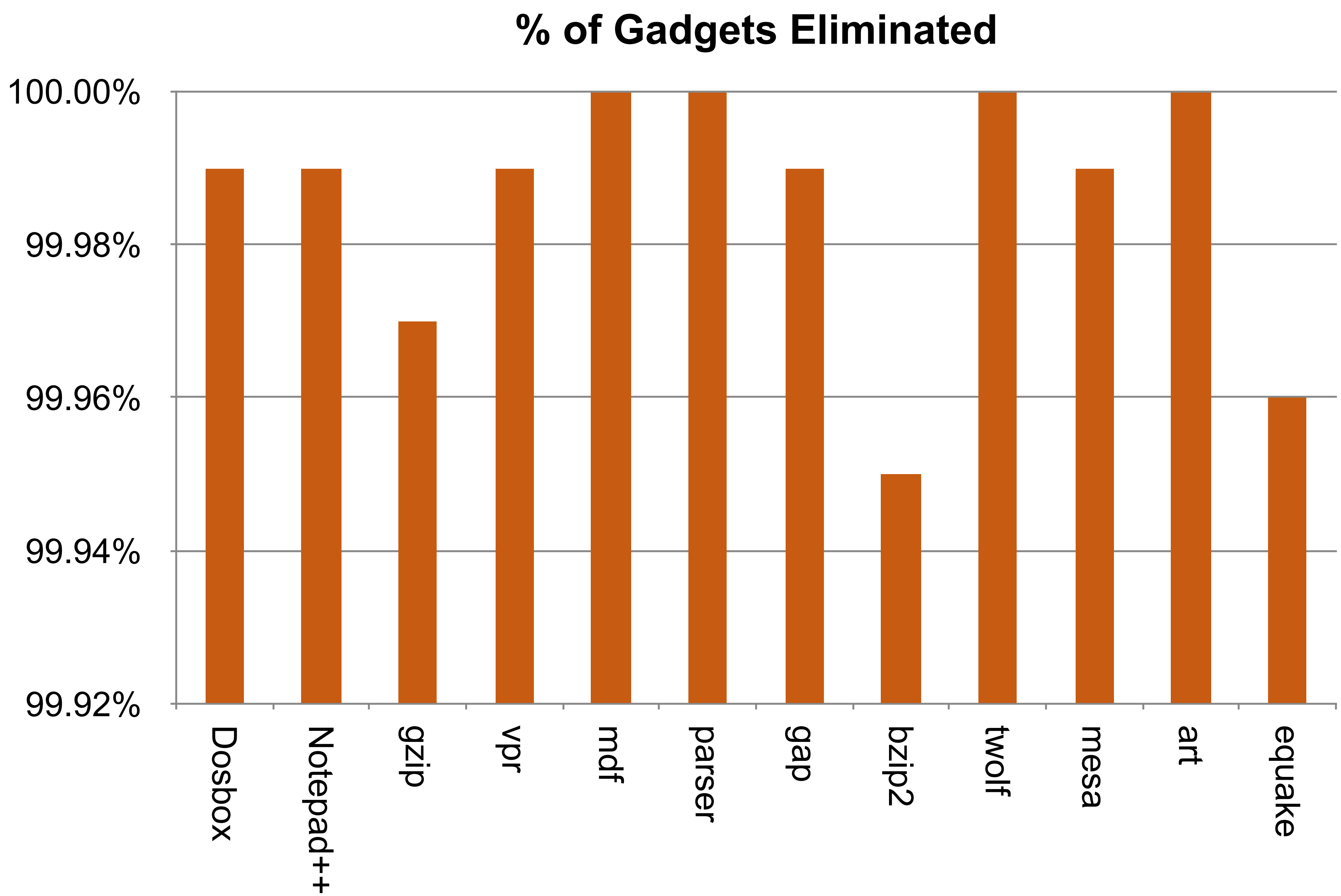
.**tnew**:00534AB9	5B	pop ebp

Entropy Discussion

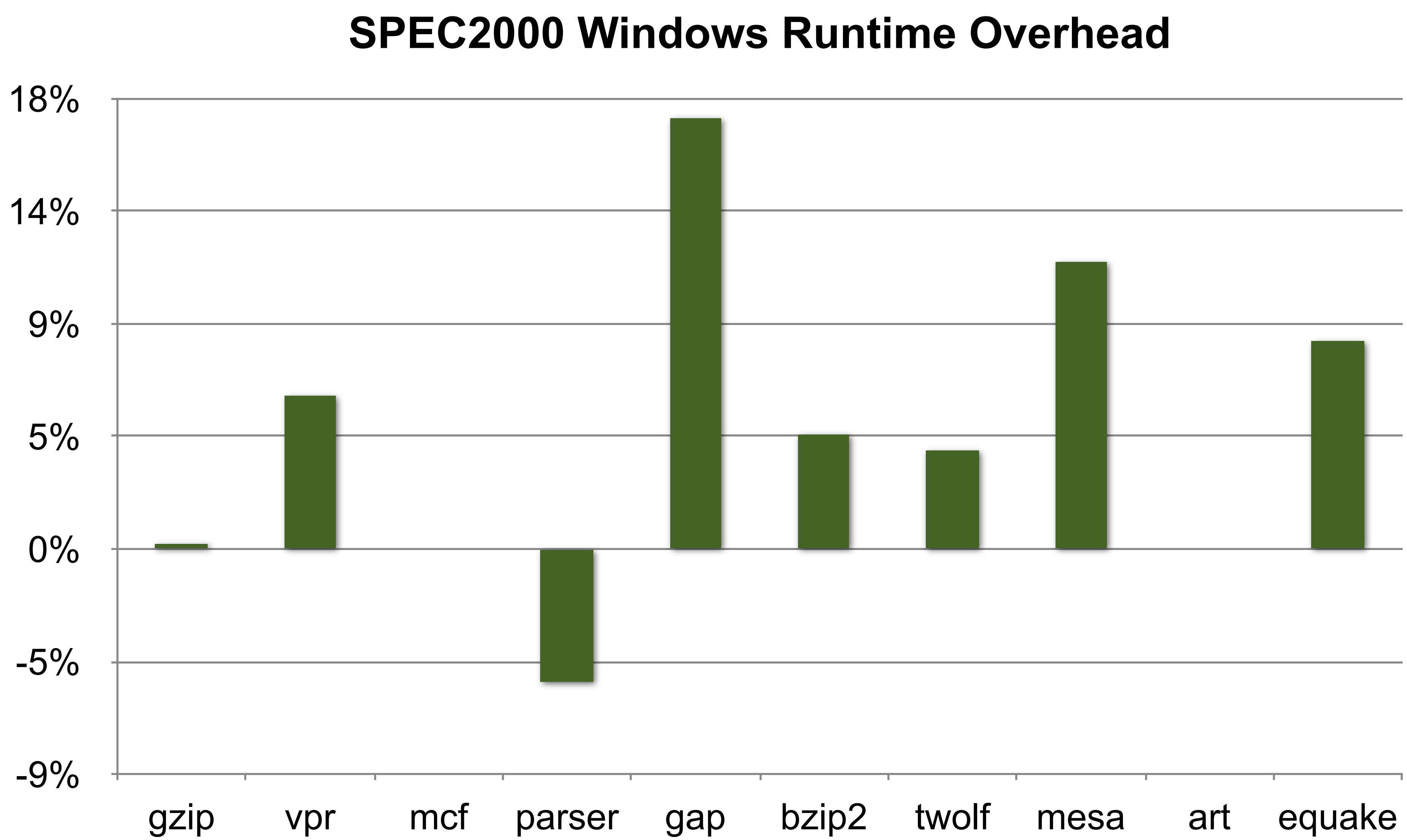
- ASLR
  - 2^{n-1} probes where n is the number of bits of randomness
- STIR
  - $(2^n)! / (2(2^n - g)!)$ probes where g is the number of gadgets in the payload
    - Must guess each where each gadget is with each probe.

- On a 64-bit architecture, the expected number of probes for a g=3-gadget attack is therefore over 7.92×10^{28} times greater with STIR than with re-randomizing ASLR.

Gadget Reduction

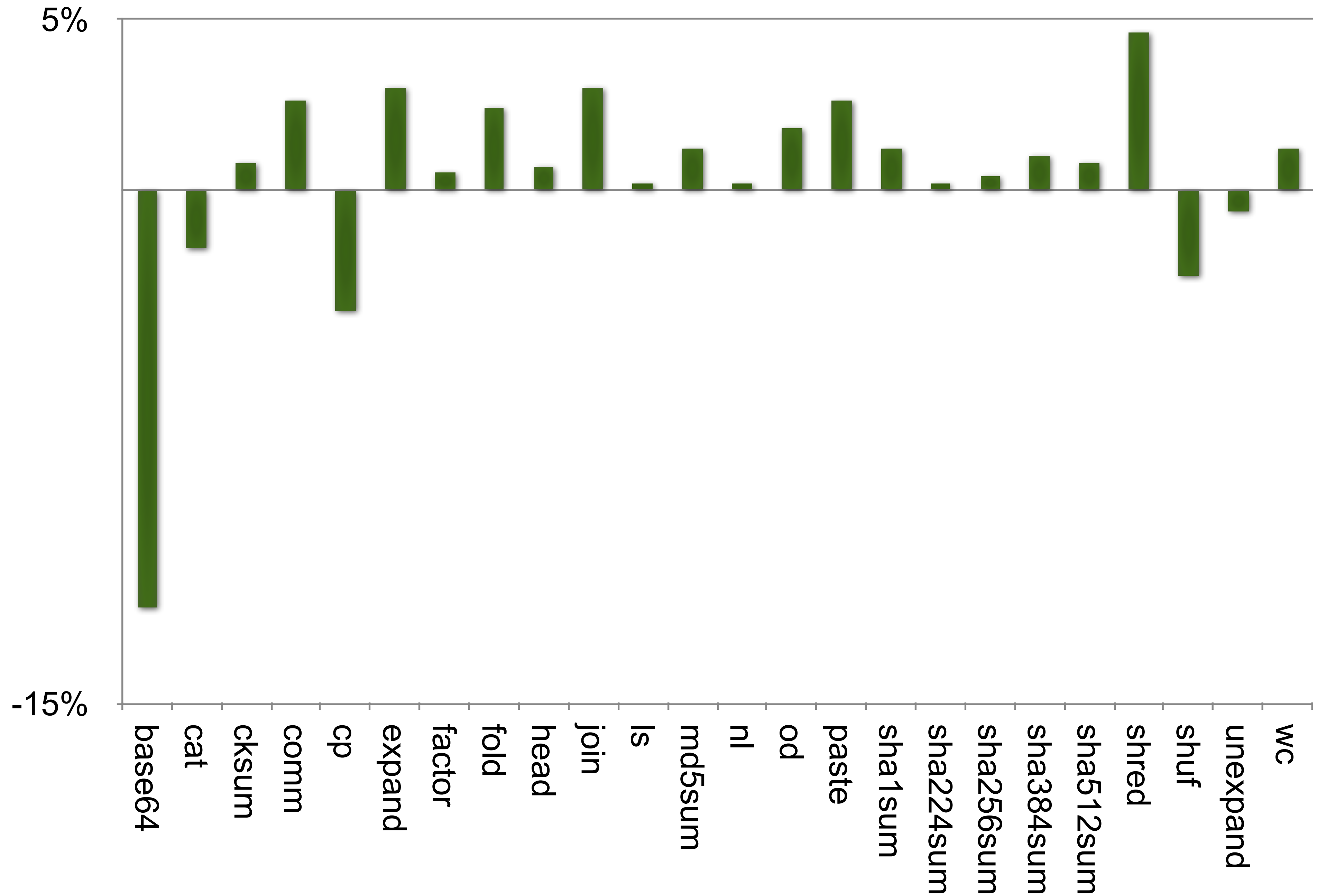


Windows Runtime Overhead



 [Wartell'12]

Linux Runtime Overhead



Conclusions

- First static rewriter to protect against RoP attacks
  - Greatly increases search space
  - Introduces no deployment issues
  - Tested on 100+ Windows and Linux binaries
  - 99.99% gadget reduction on average
  - 1.6% overhead on average
  - 37% process size increase on average

Problems with Binary Stirring

- Binary Stirring employs heuristics, which work on simple binaries
- Dynamic libraries are not considered in the evaluation
  - hence symbolization problem not addressed

Reassembleable Disassembling

Shuai Wang, Pei Wang, and Dinghao Wu, Usenix Security 2015

Motivation

- Analyzing and retrofitting COTS binaries with:
  - software fault isolation
  - control-flow integrity
  - symbolic taint analysis
  - elimination of ROP gadgets
- Binary rewriting comes with major drawbacks/limitations
  - runtime overhead from patching due to control-flow transfers
  - patching requires PIC if code is relocated
  - instrumentation significantly increases binary size
  - binary reuse only works for small binaries (coverage)

Goal

Produce reassembleable assembly code from stripped COTS binaries in a fully automated manner.

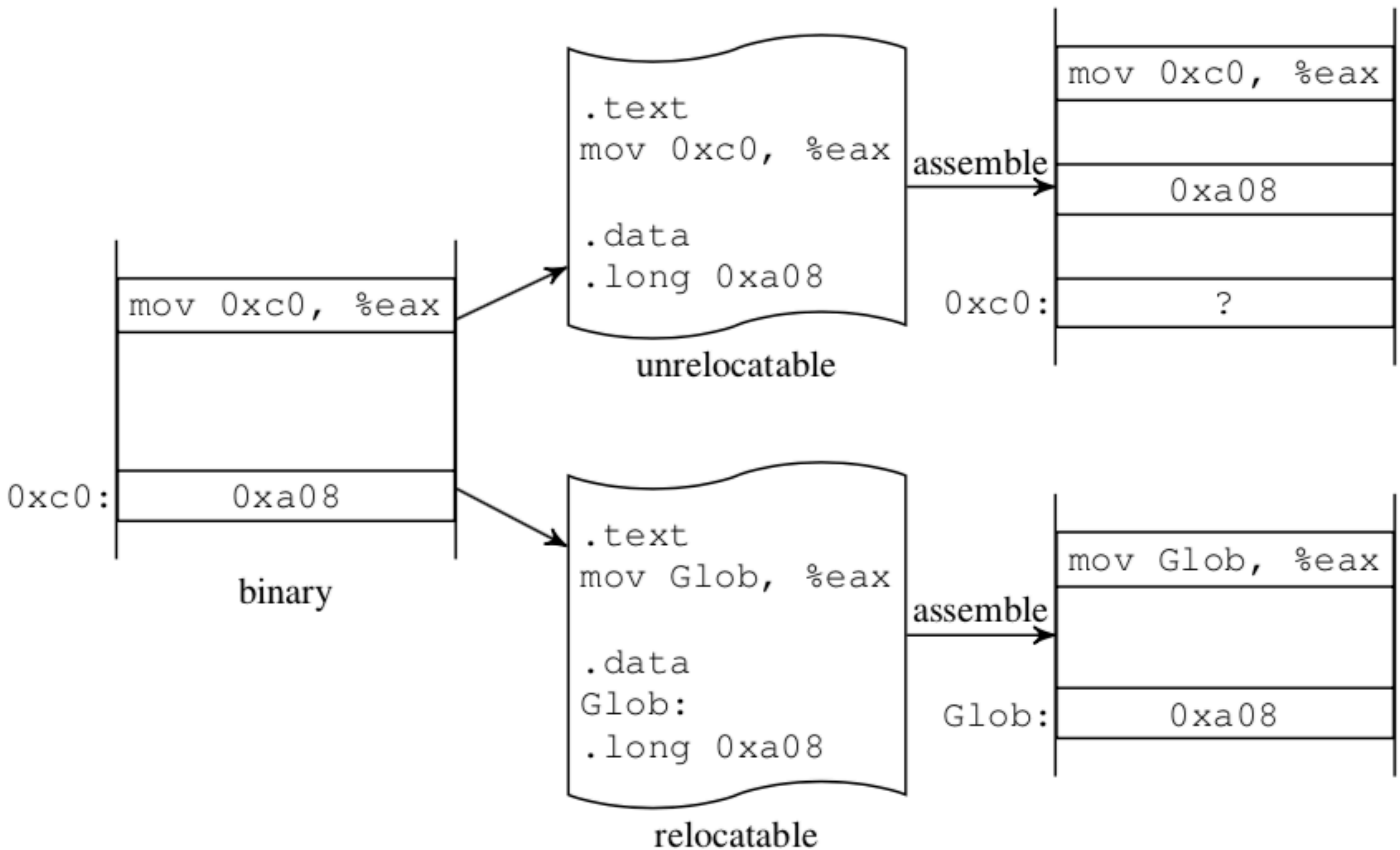
- Allows binary-based whole program transformations
- Requires relocatable assembly code→symbolization of immediate values
- Complementary to existing work

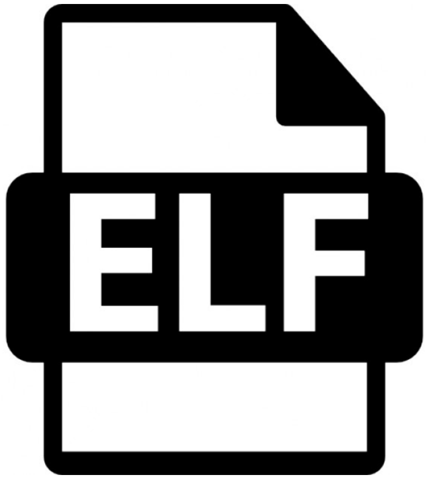
Symbolization

Given an immediate value in assembly code,
is it a constant or a memory address?

- Reassembling transformed program changes binary layout
- Address changes invalidate memory references
- x86
  - No distinction between code and data
  - Variable-length instruction encoding

(Un)Relocatable Assembly Code





Disassemble

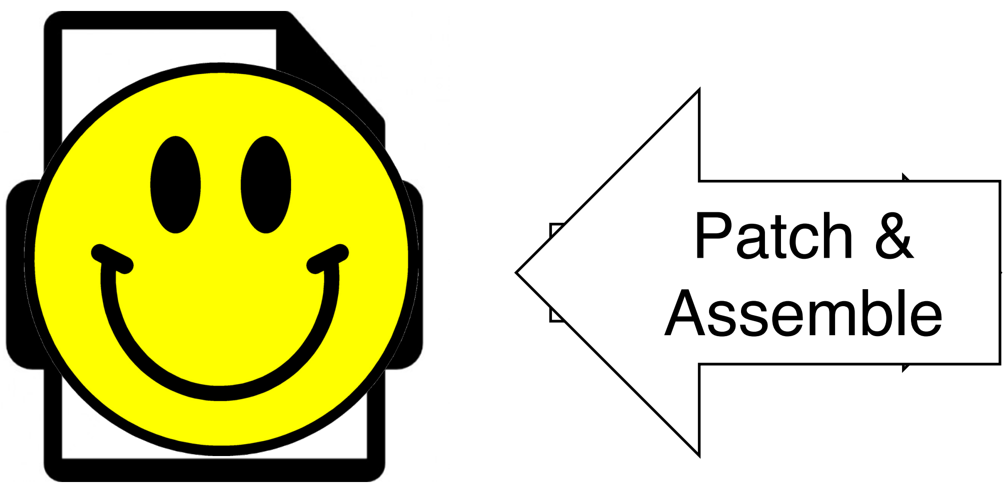
	.text
400100	mov [6000a0], eax
400105	jmp 0x40020d
	...
40020d	mov [6000a4], 1
	.data
6000a0	.long 0xc0debeef
6000a4	.long 0x0

	.text
	mov [*data_0*], eax
	jmp *target*
	...
target	mov [*data_1*], 1
	.data
data_0	.long 0xc0debeef
data_1	.long 0x0

Patch &
Assemble

	.text
400100	mov [6000a0], eax
400105	jmp 40020d
40020d	CRASH!
40020f	mov [6000a4], 1
	.data
6000a0	“cat\x00”
6000a4	.long 0x0
6000a8	

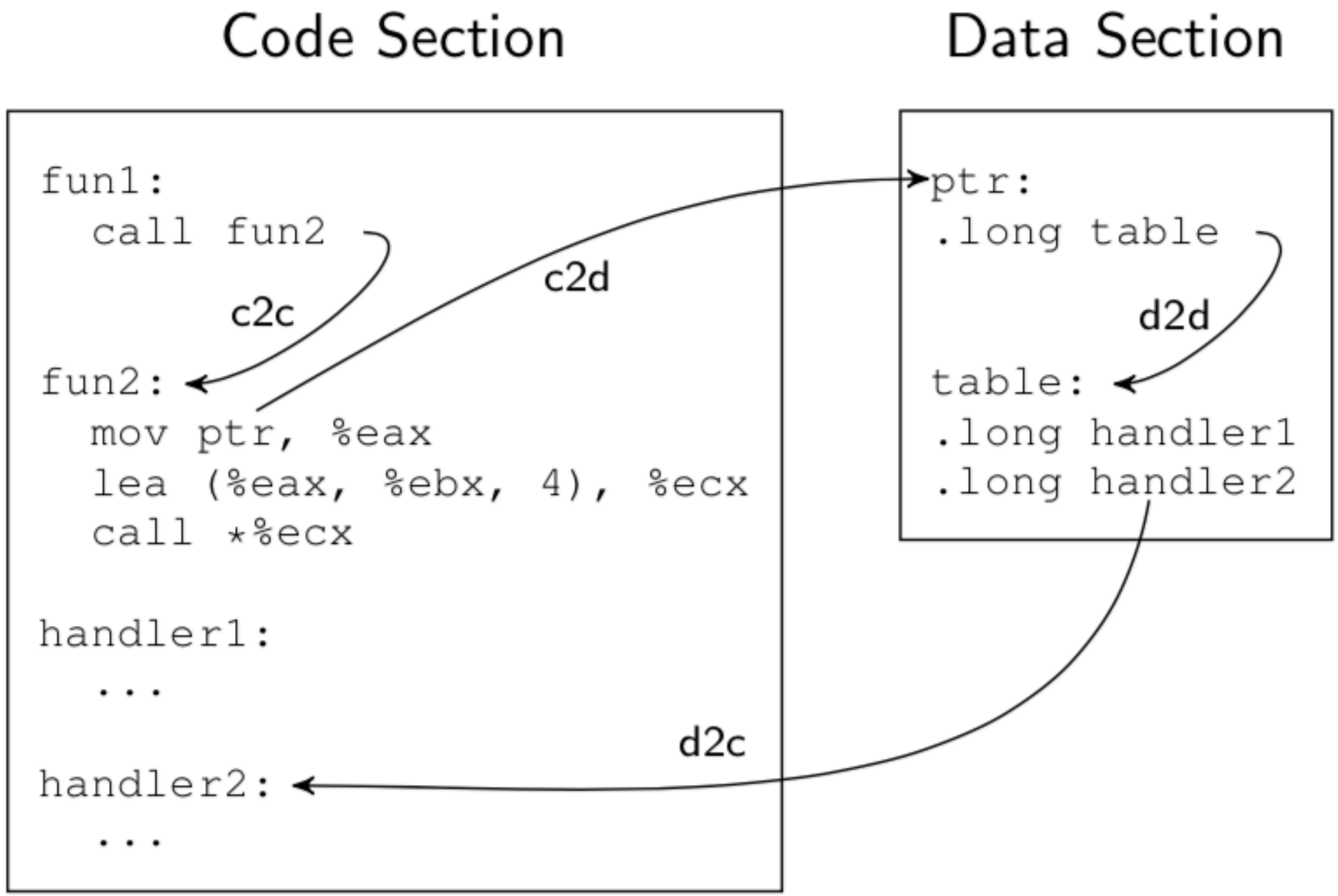
Non-relocatable Assembly



	.text
	mov [*data_0*], eax
	jmp *target*
	...
	CRASH!
target	mov [*data_1*], 1
	.data
data_0	.long 0xc0debeef
new	"cat\x00"
data_0	.long 0xc0debeef
data_1	.long 0x0

Relocatable Assembly

Types of Symbol References



Symbolization of c2c and c2d References

- Valid memory references point into code or data section
- Assume all immediates to be references and filter out invalid ones

Symbolization of d2c and d2d References

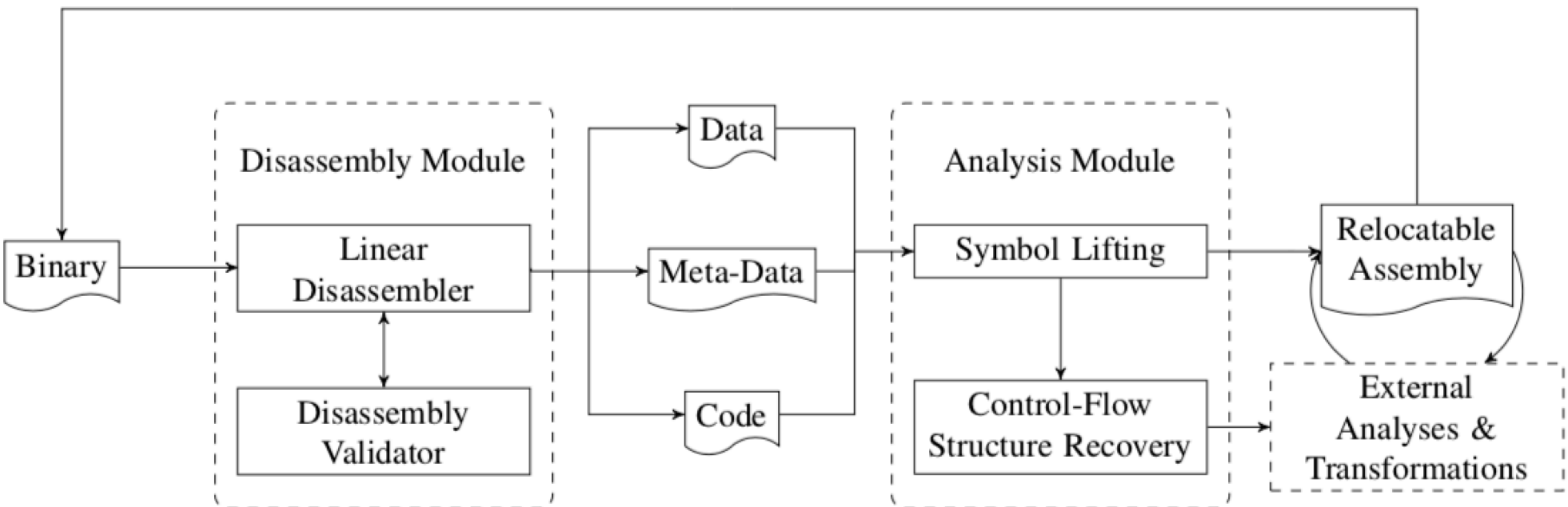
- Assumption 1
  - “All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.”
  - → Consider only n-byte values which are n-byte aligned
- Assumption 2
  - “Users do not need to perform transformation on the original binary data.”
  - → Keep start addresses of data sections during reassembly and ignore d2d references
- Assumption 3
  - “d2c symbol references are only used as function pointers or jump table entries.”
  - → References need to point to start of a function or form a jump table

Evaluation

- Uroboros: 13,209 SLOC in OCaml and Python; works with x86/x64 ELF binaries
- Intel Core i7-3770 @ 3.4GHz with 8GiB RAM running Ubuntu 12.04
- 122 programs compiled for 32- and 64-bit targets
- gcc 4.6.3 with default configuration and optimization of each program
- stripped before testing

Collection	Size	Content
COREUTILS	103	GNU Core Utilities
REAL	7	bc, ctags, gzip, mongoose, nweb, oftpd, thttpd
SPEC	12	C programs in SPEC2006

Architecture of Uroboros



Correctness

- Test input shipped with programs or custom test of major functionality (some of REAL)

Assumption Set	Binaries Failing Functionality Tests	
	32-bit	64-bit
{}	h264ref, gcc, gobmk, hmmer	perlbench, gcc, gobmk, hmmer, sjeng, h264ref, lbm, sphinx3
{A1}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A2}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A3}	gobmk	gcc, gobmk
{A1, A2, A3}	gobmk	

Symbolization Errors

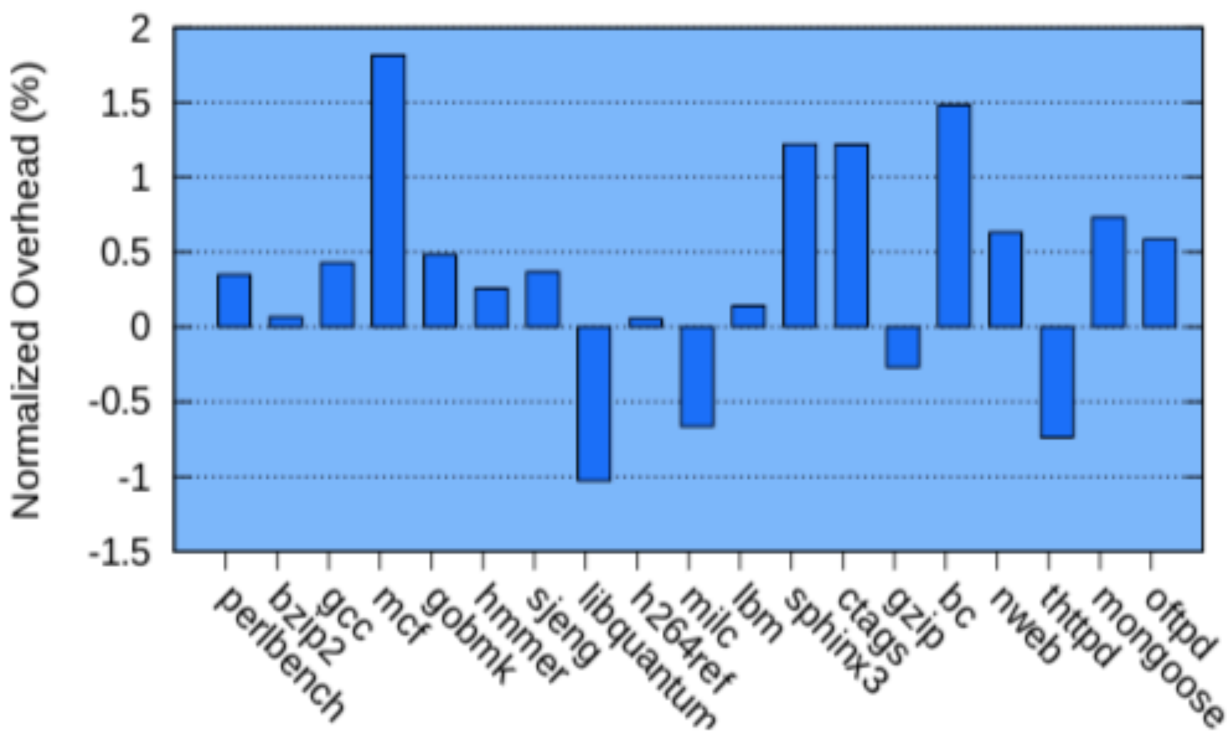
Table 4: Symbolization false positives of 32-bit SPEC, REAL and COREUTILS (Others have zero false positive)

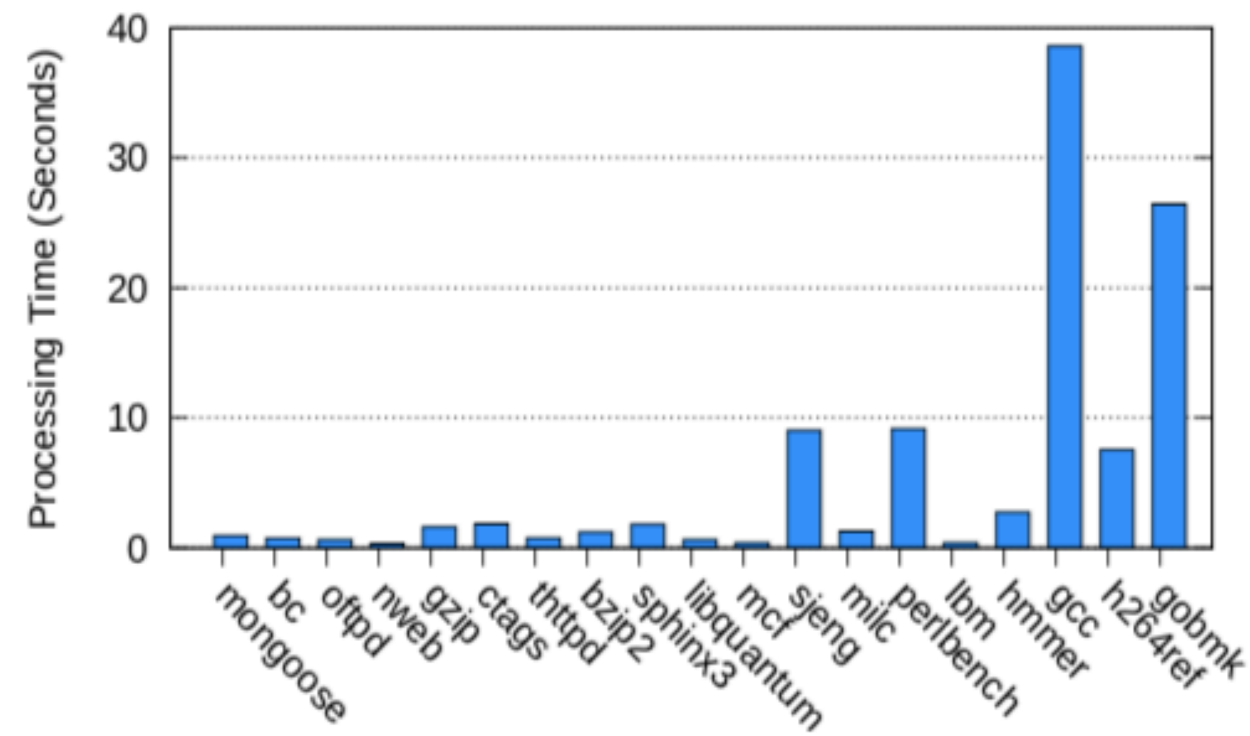
Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	1	0.049‰	1	0.049‰	0	0.000‰	0	0.000‰
gcc	262698	49	0.187‰	32	0.122‰	32	0.122‰	0	0.000‰	0	0.000‰
gobmk	65244	1348	20.661‰	985	15.097‰	912	13.978‰	78	1.196‰	5	0.077‰

Table 5: Symbolization false negatives of 32-bit SPEC, REAL and COREUTILS (Others have zero false negative)

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gcc	262698	11	0.042‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gobmk	65244	86	1.318‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰

Overhead for REAL and SPEC





- No increase in binary size after first disassemble-assemble cycle

Conclusion

- Heuristic-based symbolization of memory references
- Uroboros provides re-assembleable disassembly
  - Available at https://github.com/s3team/uroboros
- Assumes availability of raw disassembly and function starting addresses
- Tested with gcc and Clang compiled binaries
- Limited support for C++ (need to parse DWARF)

Ramblr: Making Reassembly Great Again

Ruoyu “Fish” Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna, NDSS 2017

Disassemble

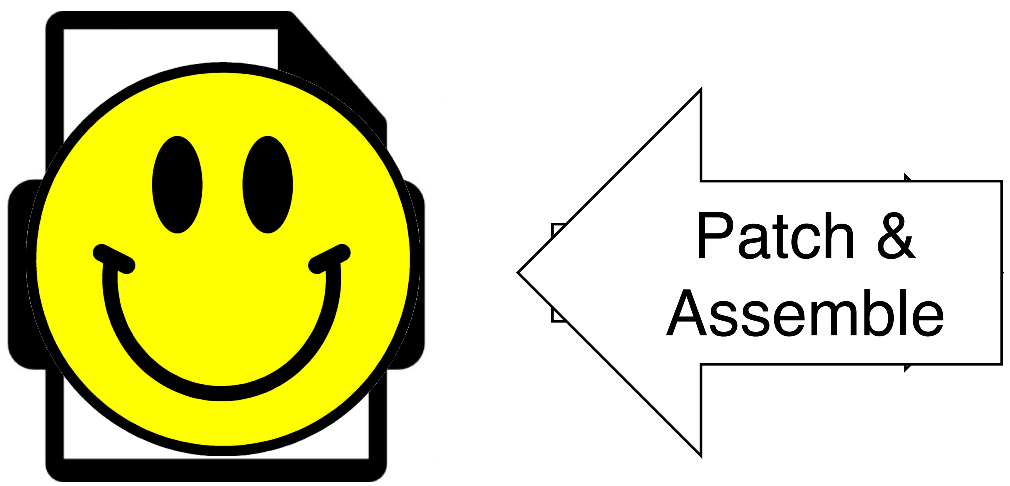
	.text
400100	mov [6000a0], eax
400105	jmp 0x40020d
	...
40020d	mov [6000a4], 1
	.data
6000a0	.long 0xc0debeef
6000a4	.long 0x0


```
        .text
        mov   [data_0], eax
        jmp   target
        ...
target  mov   [data_1], 1

        .data
data_0  .long 0xc0debeef
data_1  .long 0x0
```

Patch & Assemble

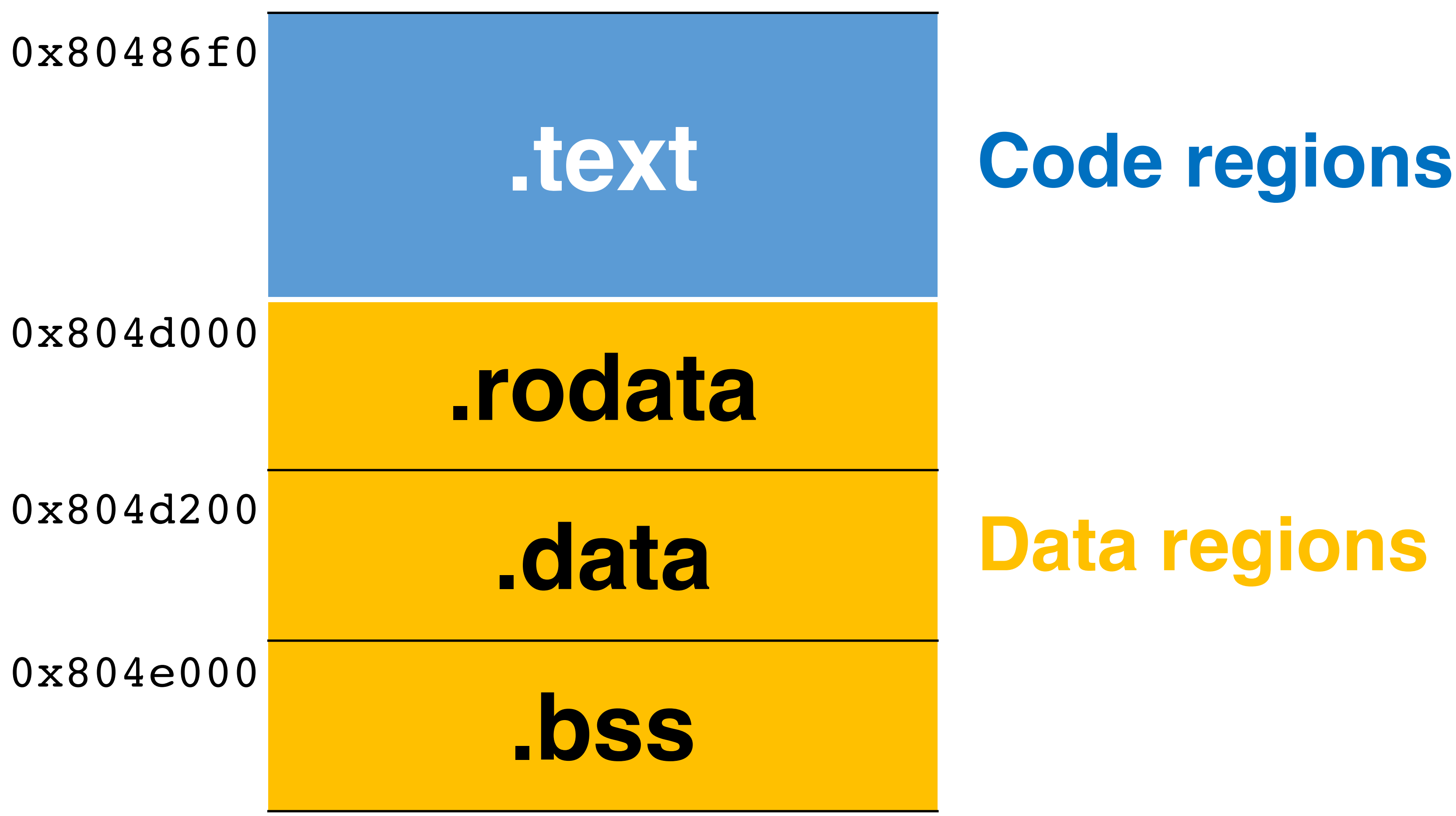
	.text
400100	mov [6000a0], eax
40020d	CRASH!
40020f	mov [6000a4], 1
	.data
6000a0	"cat\x00"
6000a4	.long 0x0
6000a8	

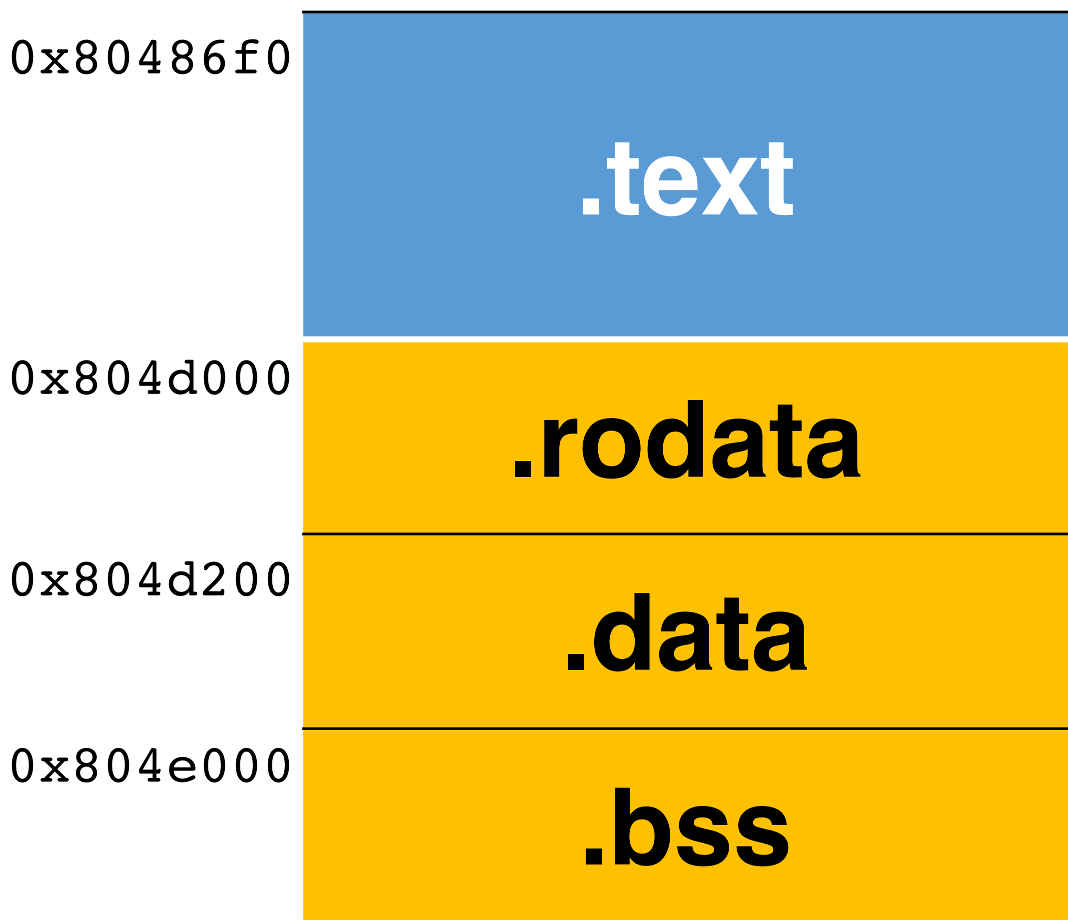
Non-relocatable Assembly



	.text
	mov [data_0], eax
	jmp target
	...
	mov [data_1], 1 CRASH!
target	mov [data_1], 1
	.data
data_0	.long 0xc0debeef
data_1 new	.long 0x0 "cat\0"
data_0	.long 0xc0debeef
data_1	.long 0x0

Relocatable Assembly


0x80486f0
.text
Code regions
0x804d000
.rodata
0x804d200
.data
Data regions
0x804e000
.bss

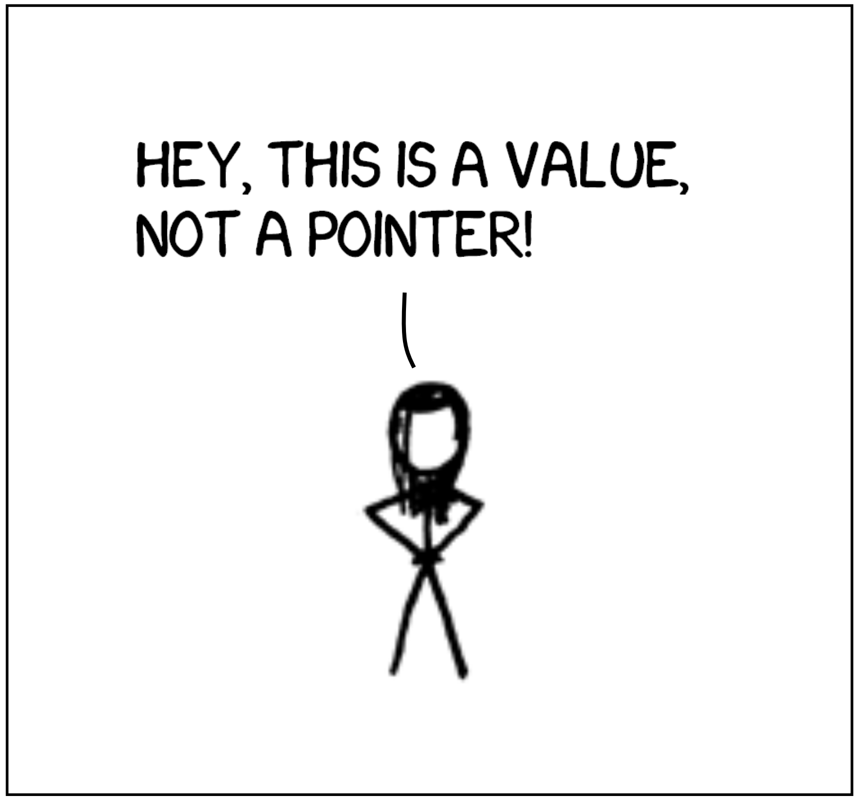


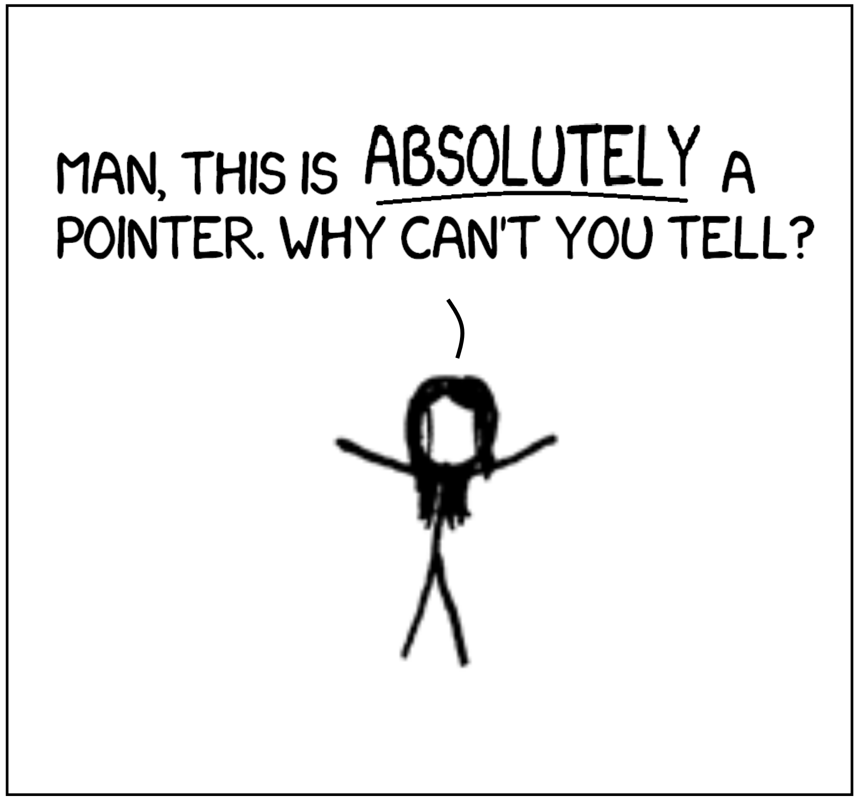

```
push    ebp
mov     ebp, esp
sub     esp, 0x48
mov     DWORD PTR [ebp-0x10], 0x0
mov     DWORD PTR [ebp-0xc], 0x0
mov     DWORD PTR [ebp-0xc], 0x80540a0
mov     eax, 0xfb7
mov     WORD PTR [ebp-0x10], ax
mov     eax, ds:0x805be60
test    eax, eax
jne     0x804895b
mov     eax, ds:0x805be5c
...

.data:
804d538:    0x8048eec
804d53c:    0x8048f05
804d540:    0x8048f1e
```

Uroboros

USENIX Sec '15

Problem





False Positives

False Negatives

[Fish'17] Thanks **xkcd** :-)

Problem: Value Collisions

False Positives

```
/* stored at 0x8060080 */
static float a = 4e-34;
```

A Floating-point Variable *a*

```
8060080        .db 3d
8060081        .db ec
8060082        .db 04
8060083        .db 08
```

Byte Representation

```
8060080        label_804ec3d
```

Interpreted as a Pointer

Problem: Compiler Optimization

False Negatives

```
int ctrs[2] = {0};

int main()
{
    int input = getchar();
    switch (input – 'A')
    {
        case 0:
            ctrs[input – 'A']++;
            break;
        ...
    }
}
```

A code snippet allows **constant folding**

Problem: Compiler Optimization

False Negatives

```
int ct                                                    04a034

int ma
{                                                       4], 1
    in
    sw
    {
```

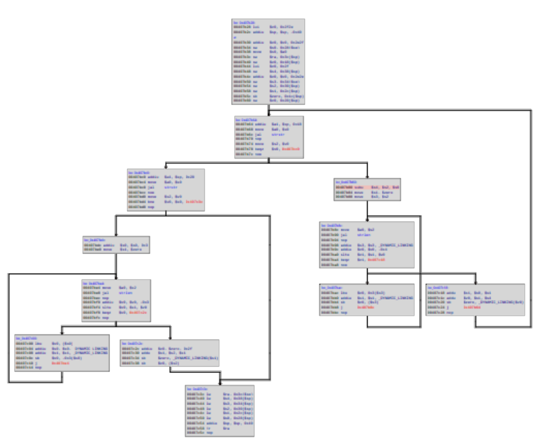
`0x804a034 – 'A' * sizeof(int) =`
`0x8049f30`

```
    }
}
```

not
tion

A code snippet allows **constant folding** Compiled in Clang with –O1

Pipeline



CFG Recovery

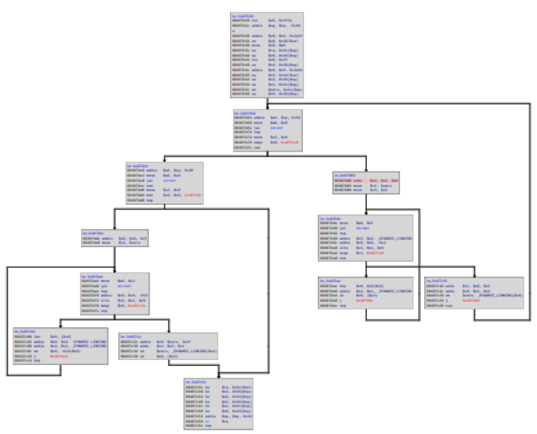
0x804850b	Pointer
0xa	Integer
0xdc5	Integer
63 61 74 00	String
0x80484a2	Pointer
0x804840b	Pointer
0xa0000	Integer

Content Classification

```
push    offset label_34
push    offset label_35
cmp     eax, ecx
jne     label_42

.label_42:
mov     eax, 0x12fa9e5
...
```

Symbolization
&
Reassembly

Pipeline



CFG Recovery

0x804850b	Pointer
0xa	Integer
0xdc5	Integer
63 61 74 00	String
0x80484a2	Pointer
0x804840b	Pointer
0xa0000	Integer

Content Classification

```
push    offset label_34
push    offset label_35
cmp     eax, ecx
jne     label_42

.label_42:
mov     eax, 0x12fa9e5
...
```

Symbolization
&
Reassembly

CFG Recovery

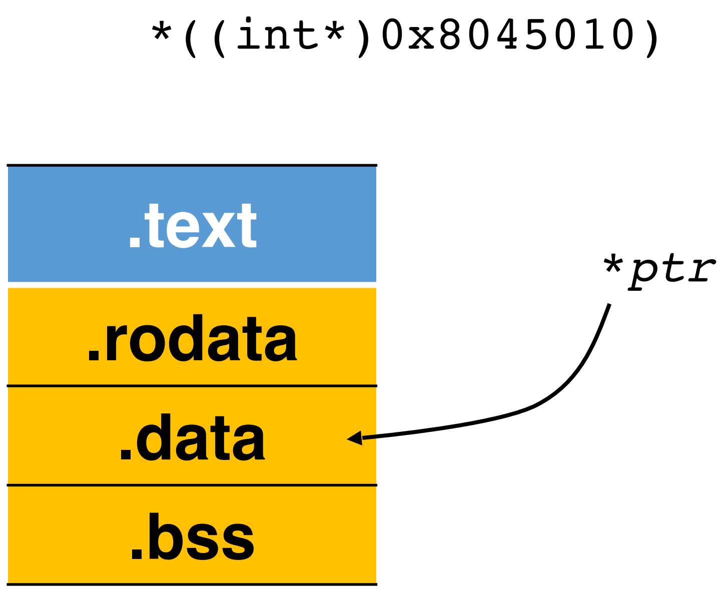
```
31 ed 5e
89 e1 83
e4 f0 50
54 52 68
00 25 05
08
```

```
0x80486f0:
xor  ebp, ebp
pop  esi
mov  ecx, esp
and  esp,
0xfffffff0
push  eax
push  esp
push  edx
...
```

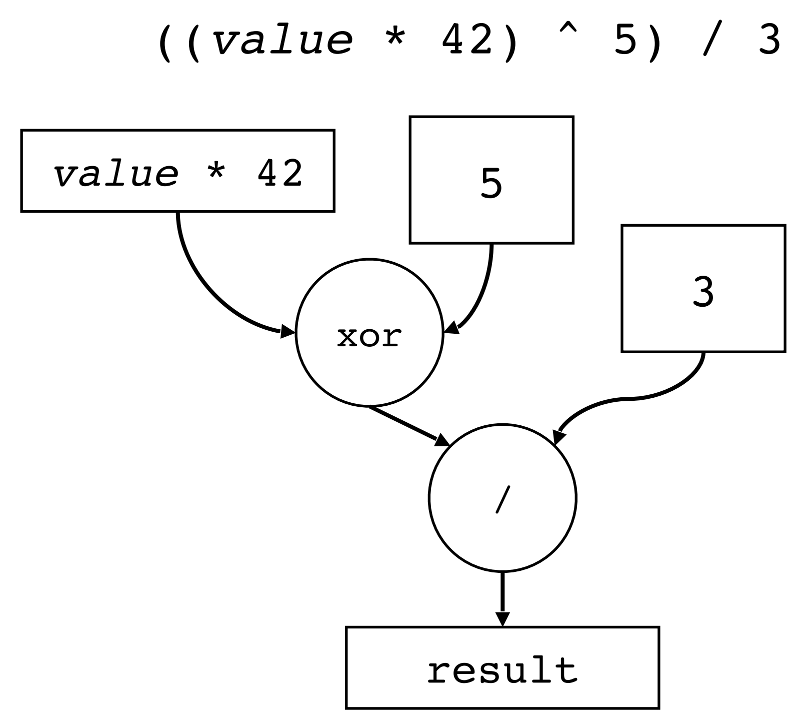
Recursive Disassembly

Iterative Refinement

Content Classification



A Typical Pointer



A Typical Value

Content Classification

Type Category	Examples
Primitive types	Pointers, shorts, DWORDs, QWORDs, Floating-point values, etc.
Strings	Null-terminated ASCII strings, Null-terminated UTF-16 strings
Jump tables	A list of jump targets
Arrays of primitive types	An array of pointers, a sequence of integers

Data Types that Ramblr Recognizes

Content Classification

MOVe **S**calar **D**ouble-precision
floating-point value

```
movsd   xmm0, ds:0x804d750
movsd   xmm1, ds:0x804d758
```

Two floating-points	
804d750	Floating point integer
804d758	Floating point integer

Recognizing Types during CFG Recovery

Content Classification

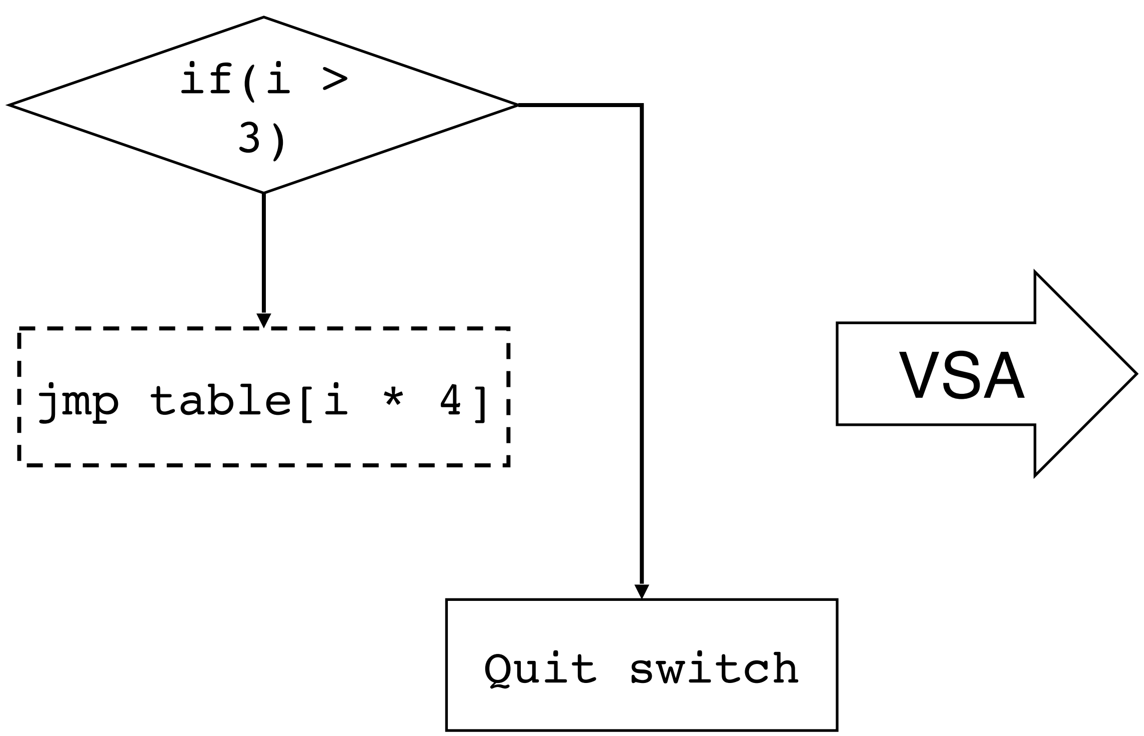
```
chr = _getch();
switch (i)
{
    case 1:
        a += 2;
break;
    case 2:
        b += 4;
break;
    case 3:
        c += 6;
break;
    default:
        a = 0; break;
}
```

Slicing

```
switch (i)
{
    case 1:
        ...
    case 2:
        ...
    case 3:
        ...
    default:
        ...
}
```

Recognizing Types with Slicing & VSA

Content Classification



`i = [0, 2] with a stride of 1`

A jump table of 3 entries	
`table[0]`	Pointer, jump target
`table[1]`	Pointer, jump target
`table[2]`	Pointer, jump target

Recognizing Types with Slicing & VSA

Base Pointer Reattribution

False Negatives

```
int ctrs[2] = {0};

int main()
{
    int input = getchar();
    switch (input – 'A')
    {
        case 0:
            ctrs[input – 'A']++;
            break;
        ...
    }
}
```

A code snippet allows **constant folding**

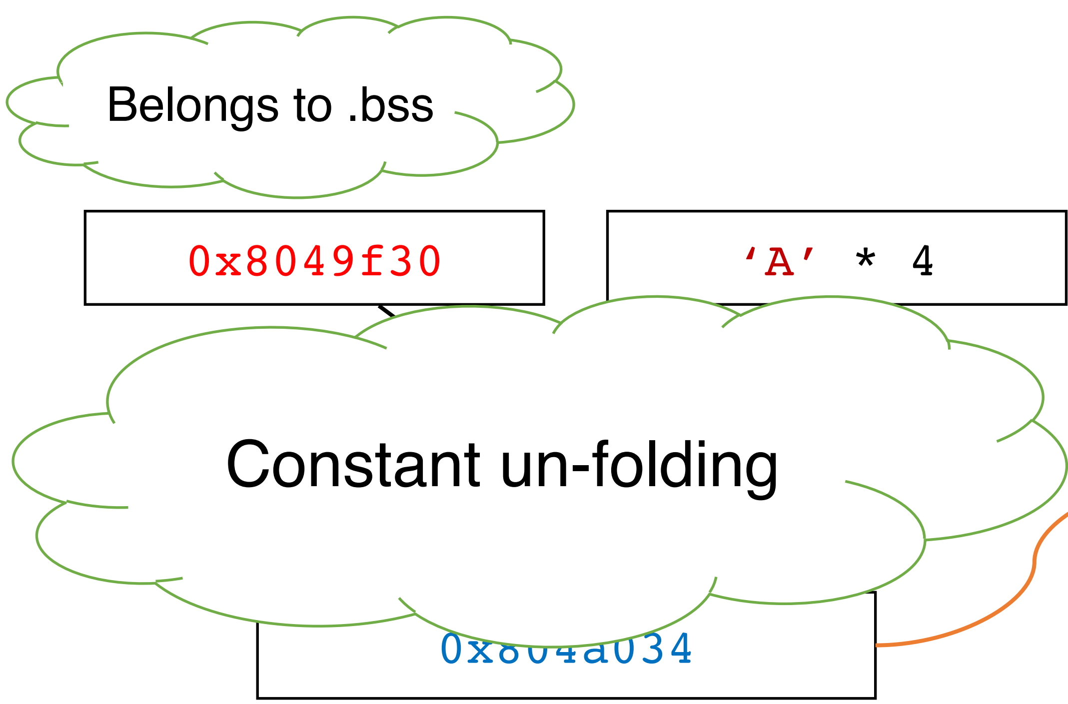
```
; Assuming ctrs is stored at 0x804a034
; eax holds the input character
; ctrs[input – 'A']++;
          add    0x8049f30[eax * 4], 1
...

.bss
804a034:  ctrs[0]
804a038:  ctrs[1]
```

0x8049f30 does not belong to any section

Compiled in Clang with –O1

Base Pointer Reattribution

False Negatives



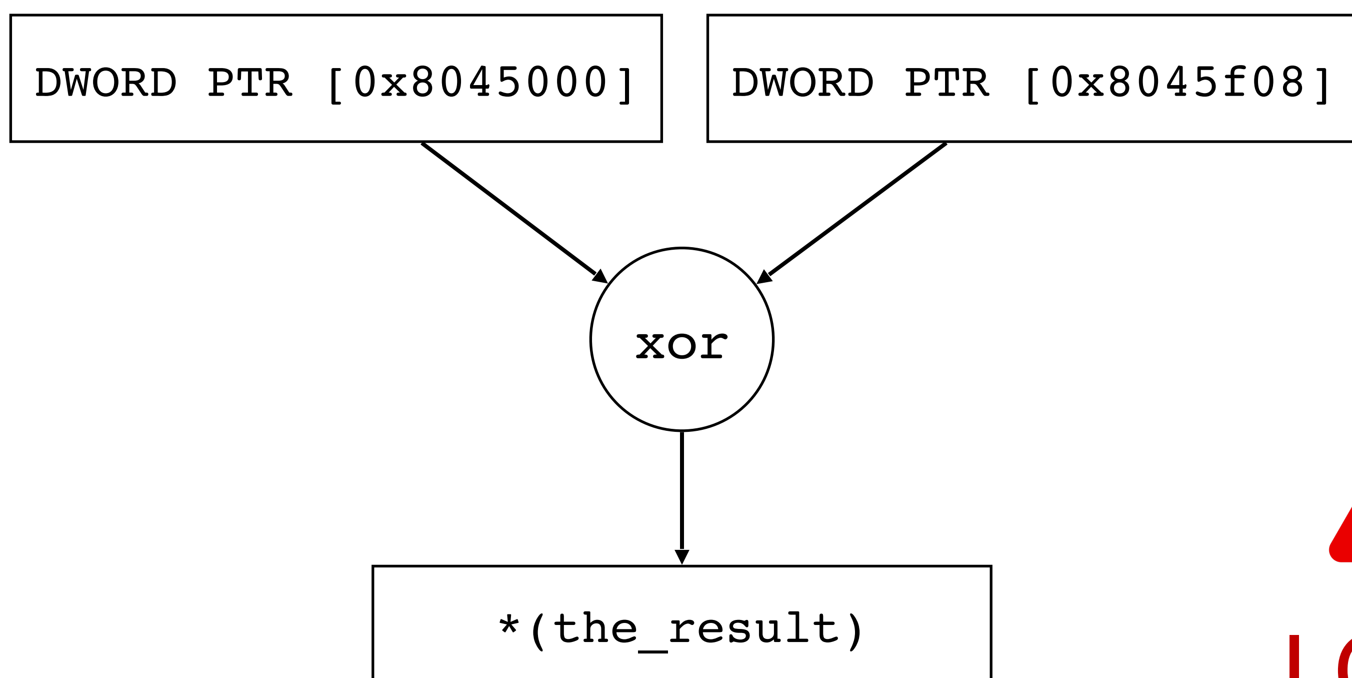
The Slicing Result

```
; Assuming ctrs is stored at 0x804a034
; eax holds the input character
; ctrs[input – 'A']++;
          add    0x8049f30[eax * 4], 1
...

.bss
804a034:  ctrs[0]
804a038:  ctrs[1]
```

0x8049f30 does not belong to any section

Compiled in Clang with –O1

Safety Heuristics: Data Consumer Check



I GIVE UP

Unusual Behaviors Triggering the Opt-out Rule

Symbolization & Reassembly

0x400010	➡	label_34
0x400020	➡	label_35
0x400a14	➡	label_42
	...	
0x406000	➡	data_3

Symbolization

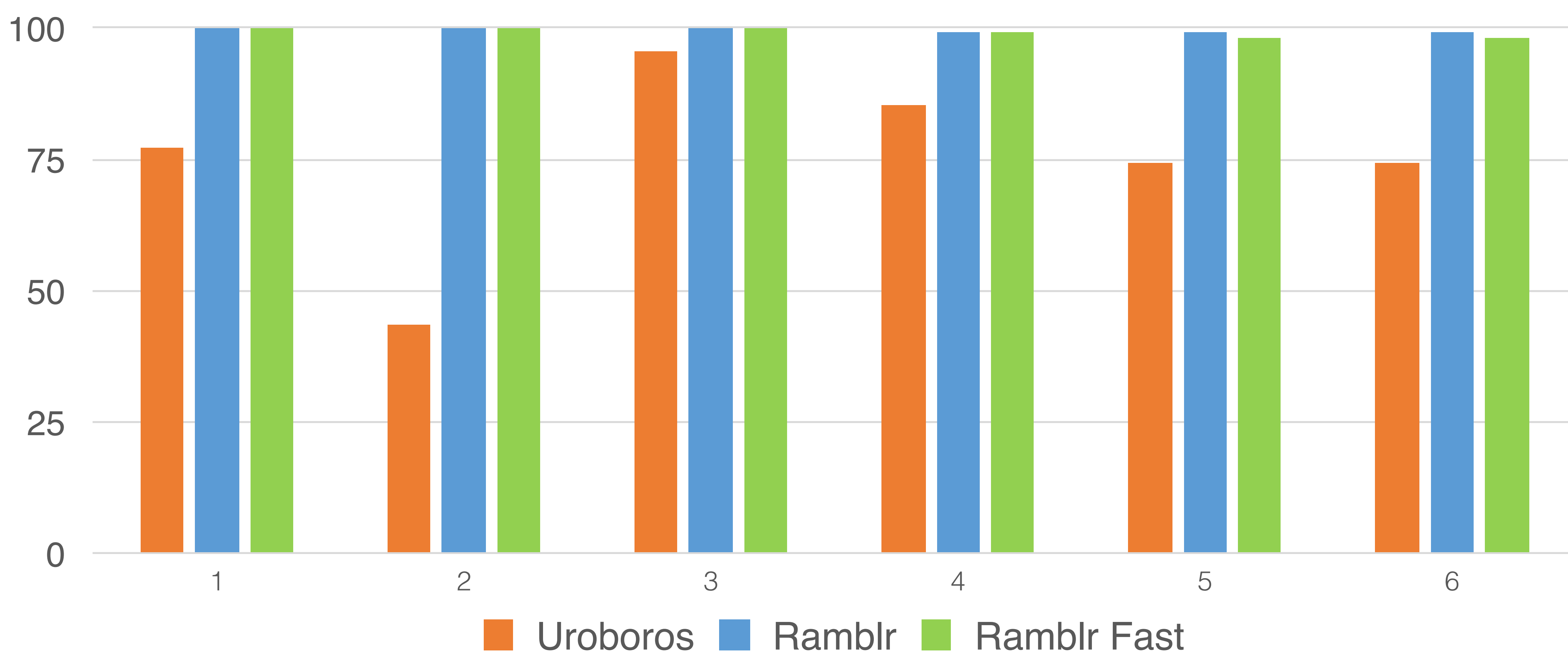
```
push    offset label_34
push    offset label_35
cmp     eax, ecx
jne     label_42

.label_42:
mov     eax, 0x12fa9e5
...
```

Assembly Generation

Data sets

	Coreutils 8.25.55	Binaries from CGC
Programs	106	143
Compiler	Clang 4.4	CGC 5
Optimization levels	O0/O1/O2/O3/Os/Ofast	
Architectures	X86/AMD64	X86
Test cases	Yes	Yes
Total binaries	**1272**	**725**

Brief Results: Success Rate



Ramblr is the foundation of ...

- Patching Vulnerabilities
- Obfuscating Control Flows
- Optimizing Binaries
- Hardening Binaries

Another related work

- Superset Disassembly: Statistically Rewriting x86 Binaries Without Heuristics, Eric Bauman, Zhiqiang Lin, Kevin Hamlen, NDSS 2018.

Acknowledgments/References (1/2)

- [B. P. Miller'06] A Framework for Binary Code Analysis, and Static and Dynamic Patching, Barton P. Miller, Jeffrey Hollingsworth, February 2006
- [Wartell’12] Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code (Slides), R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin.CCS 2012
- [Onarlioglu’10] G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries, K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, E. Kirda, ACSAC 2010
- [Wang’15] Reassembleable Disassembling (Slides), Shuai Wang, Pei Wang, and Dinghao Wu, Usenix Security 2015
- [Fish’17] Ramblr: Making Reassembly Great Again (Slides), Ruoyu “Fish” Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna, NDSS 2017

Acknowledgments/References (2/2)

- [CS-6V81] System Security and Malicious Code Analysis, S. Qumruzzaman, K. Al-Naami, Spring 2012. Based on “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software”, J. Newsome and D. Song, NDSS 2005.
- [Salwan’15] Dynamic Binary Analysis and Instrumentation Covering a function using a DSE approach, J. Salwan, Security Day, January 2015.