软件安全与漏洞分析

3.2 返回导向编程的发展（1）

Previously in Software Security

- 返回导向编程初步介绍
  - 渊源
  - 基本原理
  - 典型构造方式
- CISC指令构架对返回导向编程的意义

返回导向编程的发展

- 本节主题 – 失去CISC所带来的优势时，返回导向编程如何生存？
  - RISC以及针对RISC的返回导向编程
  - Smashing the stack vs. Smashing the gadgets
  - 不使用ret指令（0xC3）的返回导向编程

回顾：经典的CISC返回导向编程

- 将劫持后的控制流引导至**正常指令的内部（middle of an instruction）**
- 利用CISC的特点，形成**意外的指令片段（unintended instruction sequence）**
- Gadget资源的重要来源：因误解析而形成的“指令”（特别是0xC3字节）

- 那么，无法使CPU做出错误的解读，又如何？

精简指令集（RISC)

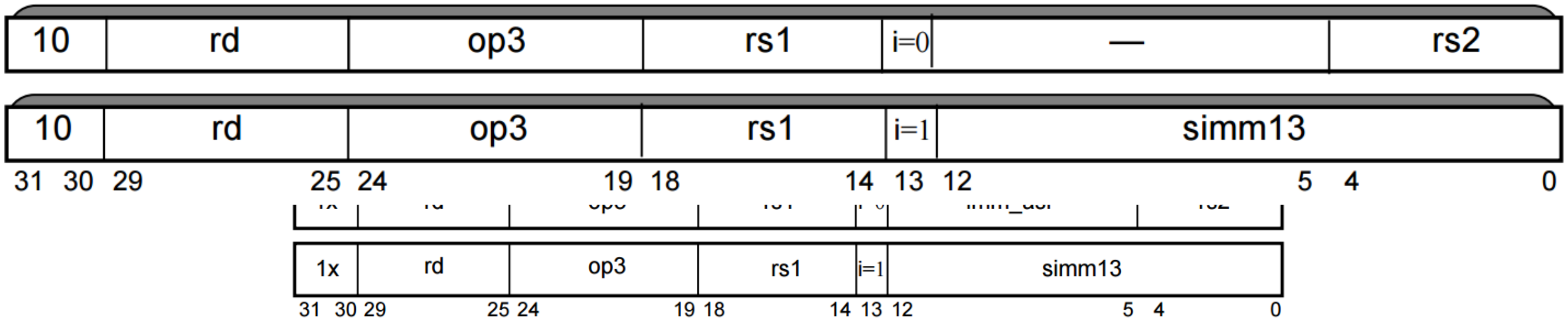
- 背景：CISC存在许多缺点
  - 各种指令的使用率相差悬殊，且微码串行执行让频繁使用的简单指令也效率低下
  - 复杂指令 → 复杂的硬件结构，CISC越来越难以集成在单一芯片上
  - 许多复杂指令需要极复杂的操作，多数已可视为某种高级语言的翻版，通用性差
- RISC的特点：
  - 统一指令编码，如所有指令长度相等、op-code位置相同等，可快速解译
  - 泛用的缓存器，单纯的寻址模式(用计算指令序列取代复杂寻址模式)
  - 硬件中支持少数数据型别（如区分整数/浮点数等）

精简指令集（RISC)

- 举例：“可 ……SPARC）

op = 00_2: *SETHI, Branches, and ILLTRAP*

Instruction	op3	Operation	Assembly Language Syntax		Class
ADD	00 0000	Add	add	reg_{rs1}, reg_or_imm, reg_{rd}	**A1**
ADDcc	01 0000	Add and modify cc's	addcc	reg_{rs1}, reg_or_imm, reg_{rd}	**A1**
ADDC	00 1000	Add with 32-bit Carry	addc	reg_{rs1}, reg_or_imm, reg_{rd}	**A1**
ADDCcc	01 1000	Add with 32-bit Carry and modify cc's	addccc	reg_{rs1}, reg_or_imm, reg_{rd}	**A1**



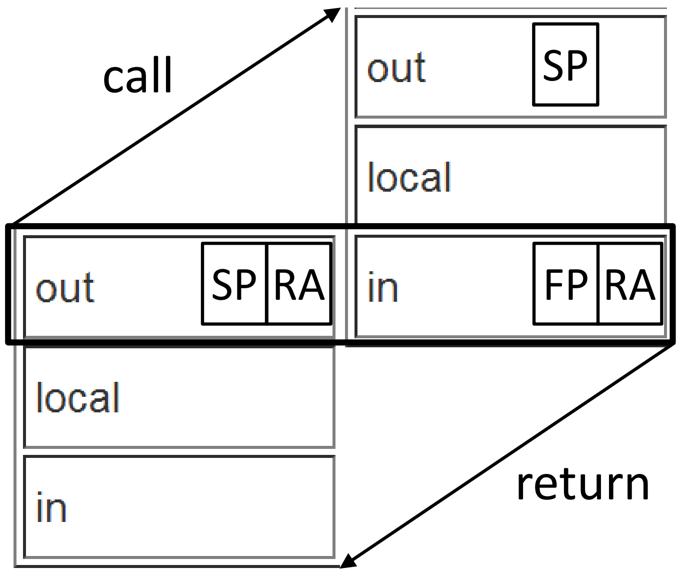
精简指令集（RISC）

- SPARC的寄存器“窗口”机制
  - 32个通用寄存器，其中8个全局寄存器和一个“窗口”（包含24个寄存器）
  - 支持2~32个“窗口”（取决于硬件实现），通常为7~8个——由此得名“可扩展的”
  - 在任何时候，只有一个寄存器窗口是可见的

Register Group	Mnemonic	Register Address
global	%g0-%g7	r[0]-r[7]
out	%o0-%o7	r[8]-r[15]
local	%l0-%l7	r[16]-r[23]
in	%i0-%i7	r[24]-r[31]

精简指令集（RISC)

- SPARC函数调用的栈结构



Address	Storage
Low Memory	
`%sp`	Top of the stack
`%sp - %sp+31`	Saved registers `%l[0-7]`
`%sp+32 - %sp+63`	Saved registers `%i[0-7]`
`%sp+64 - %sp+67`	Return struct for next call
`%sp+68 - %sp+91`	Outgoing arg. 1-5 space for caller
`%sp+92 - up`	Outgoing arg. 6+ for caller (*variable*)
`%sp+__` `%fp-__`	Current local variables (*variable*)
`%fp`	Top of the frame (previous `%sp`)
`%fp - %fp+31`	Prev. saved registers `%l[0-7]`
`%fp+32 - %fp+63`	Prev. saved registers `%i[0-7]`
`%fp+64 - %fp+67`	Return struct for current call
`%fp+68 - %fp+91`	Incoming arg. 1-5 space for callee
`%fp+92 - up`	Incoming arg. 6+ for callee (*variable*)
High Memory	

针对RISC的返回导向编程

- 对SPARC构造返回导向编程所面临的问题
  - 无法利用意外的指令序列（该种情况不可能发生）
  - x86下返回导向编程gadget的所有构造特点在RISC中均不存在
- 新的返回导向编程设计思路
  - 将函数的**后缀**作为gadget使用（利用其结尾处的ret-restore指令序列）
  - 利用结构化数据流使得gadget与SPARC的函数调用惯例相吻合
  - 构造内存-内存gadget（寄存器仅在gadget内部使用）

针对RISC的返回导向编程

- SPARC返回导向编程的图灵完整性 --- 读/写指针

Pointer Read (v1 = *v2)

Inst. Seq.	Preset	Assembly

%i0 = m[v2]	%i4 = &v2	ld [%i4], %i0
		ld [%i0], %i0
		ret
		restore

v1 = m[v2]	%i3 = &v1	st %o0, [%i3] ret restore

Pointer Write (*v1 = v2)

Inst. Seq.	Preset	Assembly
%i0 = v2	%l1 = &v2	ld [%l1], %i0 ret restore
m[v1] = v2	%i0 = &v1-8	ld [%i0 + 0x8], %i1 st %o0, [%i1] ret restore

针对RISC的返回导向编程

- SPARC返回导向编程的图灵完整性 --- 常/变量赋值

Constant Assignment (v1 = 0x********)

Inst. Seq.	Preset	Assembly
v1 = 0x********	%i0 = *Value* %i3 = &v1	st %i0, [%i3] ret restore

Constant Assignment (v1 = 0x00******)

Inst. Seq.	Preset	Assembly
v1 = 0xff******	%i0 = *Value* | 0xff000000 %i3 = &v1	st %i0, [%i3] ret restore
v1 = 0x00******	%i0 = &v1	clrb [%i0] ret restore ...

Variable Assignment (v1 = v2)

Inst. Seq.	Preset	Assembly
v1 = v2	%l7 = &v1 %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore

针对RISC的返回导向编程

- SPARC返回导向编程的图灵完整性 --- 算术运算（以加法为例）

Inst. Seq.	Preset	Assembly

v1++	%i1 = &v1	ld [%i1], %i0
		add %i0, 0x1, %o7
		st %o7, [%i1]
		ret
		restore

Inst. Seq.	Preset	Assembly
m[&%i0] = v2	%l7 = &%i0 (*+2 Frames*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%i3] = v3	%l7 = &%i3 (*+1 Frame*) %i0 = &v3	ld [%i0], %l6 st %l6, [%l7] ret restore
v1 = v2 + v3	%i0 = v2 (*stored*) %i3 = v3 (*stored*) %i4 = &v1	add %i0, %i3, %i5 st %i5, [%i4] ret restore

针对RISC的返回导向编程

- SPARC返回导向编程的图灵完整性 --- 逻辑运算（以逻辑与为例）

Inst. Seq.	Preset	Assembly
m[&%l3] = v2	%l7 = &%l3 (*+2 Frames*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%l4] = v3	%l7 = &%l4 (*+1 Frame*) %i0 = &v3	ld [%i0], %l6 st %l6, [%l7] ret restore
v1 = v2 & v3	%l3 = v2 (*stored*) %l4 = v3 (*stored*) %l1 = &v1 + 1 %i0 = -1	and %l3,%l4,%l2 st %l2,[%l1+%i0] ret restore ...

针对RISC的返回导向编程

- SPARC返回导向编程的图灵完整性 --- 移位运算

Inst. Seq.	Preset	Assembly
m[&%i2] = v2	%l7 = &%i2 (*+2 Frames*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%i5] = v3	%l7 = &%i5 (*+1 Frame*) %i0 = &v3	ld [%i0], %l6 st %l6, [%l7] ret restore
%i0 = v2 << v3	%i2 = v2 (*stored*) %i5 = v3 (*stored*) %l6 = -1	sll %i2,%i5,%l7 and %l6,%l7,%i0 ret restore
v1 = v2 << v3	%i3 = v1	st %o0, [%i3] ret restore

针对RISC的返回导向编程

- SPARC返回导向编程的图灵完整性 --- 控制转移

Inst. Seq.	Preset	Assembly
jump T1	%i6 = T1	ret restore

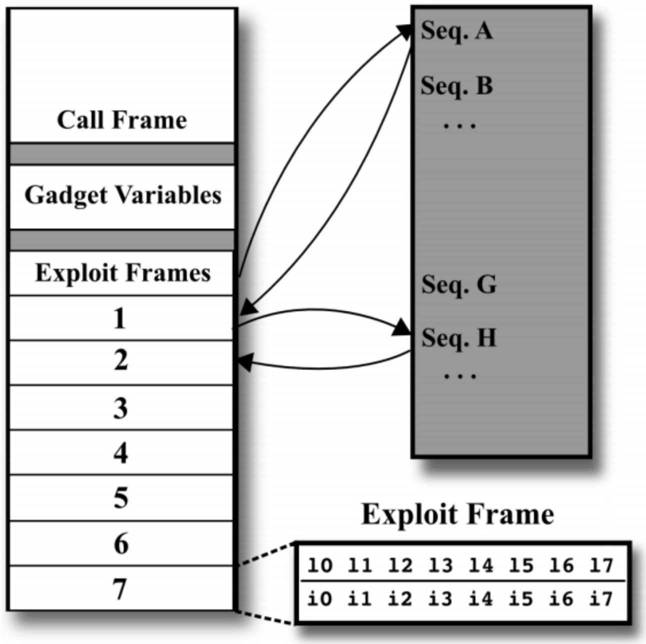
Inst. Seq.	Preset	Assembly
m[&%i0] = v1	%l7 = &%i0 (*+2 Frames*) %i0 = &v1	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%i2] = v2	%l7 = &%i2 (*+1 Frame*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
(v1 == v2)	%i0 = v1 (*stored*) %i2 = v2 (*stored*)	cmp %i0, %i2 ret restore

if (v1 == v2):
%i0 = T1
else:
%i0 = T2
	%i0 = T2 (NOT_EQ)
	%l0 = T1 (EQ) - 1
	%l2 = -1
		be,a *1 ahead*
		sub %l0,%l2,%i0
		ret
		restore

m[&%i6] = %o0	%i3 = &%i6 (*+1 Frame*)	st %o0, [%i3] ret restore
jump T1 or T2	%i6 = T1 or T2 (*stored*)	ret restore

针对RISC的返回导向编程

- SPARC返回导向编程的图灵完整性 --- 函数/系统调用



Inst. Seq.	**Preset**	**Assembly**
m[&%i6] = LastF	%i0 = LastF %i3 = &%i6 (safe)	st %i0, [%i3] ret restore
m[&%i7] = LastI	%i0 = LastI %i3 = &%i7 (safe)	st %i0, [%i3] ret restore
Optional: Up to 6 function arg seq's (v[1-6]).		
m[&%i_] = v_	%l7 = &%i[0-5] (safe) %i0 = &v[1-6]	ld [%i0], %l6 st %l6, [%l7] ret restore
Previous frame %i7 set to &FUNC - 4.		
call FUNC		ret restore
Opt. 1- Last Seq.: No return value. Just nop.		
nop		ret restore
Opt. 2 - Last Seq.: Return value %o0 stored to r1		
r1 = RETURN VAL	%i3 = &r1	st %o0, [%i3] ret restore

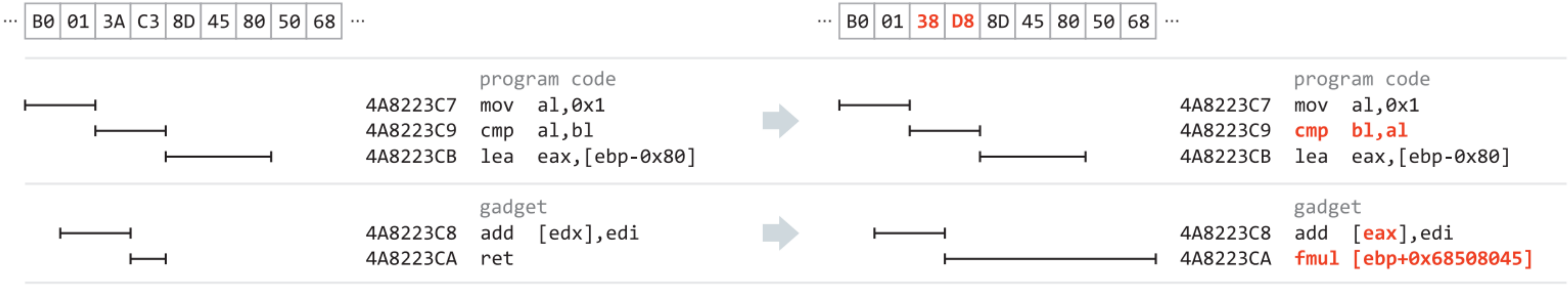
Inst. Seq.	**Preset**	**Assembly**
Write system call number to %i0 of trap frame.		
m[&%i0] = num	%l7 = &%i0 (trap frame) %i0 = &num	ld [%i0], %l6 st %l6, [%l7] ret restore
Optional: Up to 6 system call arg seq's (v[1-6]).		
m[&%i_] = v_	%l7 = &%i[0-5] (arg frame) %i0 = &v[1-6]	ld [%i0], %l6 st %l6, [%l7] ret restore
Arg Frame: Trap arguments stored in %i[0-5]		
nop		ret restore
Trap Frame: Invoke system call with number stored in %i0 with %o[0-5] as arguments.		
trap num	%i0 = num (stored) %o0 = v1 %o1 = v2 %o2 = v3 %o3 = v4 %o4 = v5 %o5 = v6	mov %i0, %g1 ta %icc, %g0+8 bcc,a,pt %icc, *4 Ahead* sra %o0,0,%i0 restore %o0,0,%o0 ba __cerror nop ret restore

针对RISC的返回导向编程

- 参考文献：Buchanan E, Roemer R, Shacham H, et al. When good instructions go bad: Generalizing return-oriented programming to RISC[C]//Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008: 27-38.
- 这里，针对RISC的返回导向编程设计反映出一个重要现象：
  - 指令集简单了，但是ROP实现反而更复杂了？
  - 原因：缺少x86上ROP所依赖的关键特性——这能否用于对ROP的防御？

Smashing the gadgets

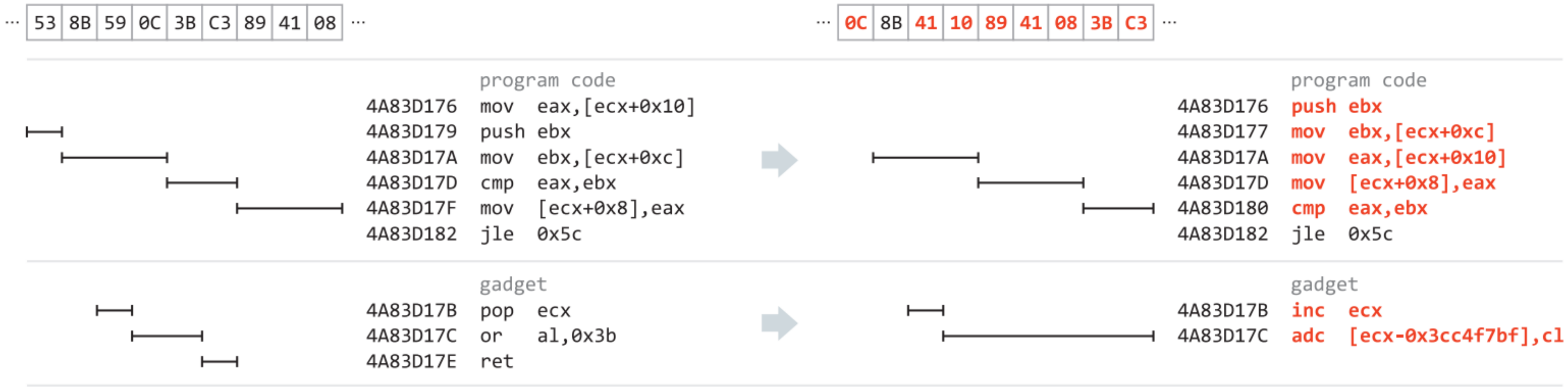
- 核心思路：阻止x86代码中出现意外的gadget



- 难点：x86编译器所生成的代码高度优化，难以随意改变指令长度
  - 因此，Smashing the gadgets必须是（in-place）的
  - 像上面这样抹掉0xC3字节仅仅是可用的手段之一

Smashing the gadgets

- 手段2：指令重新排序



Smashing the gadgets

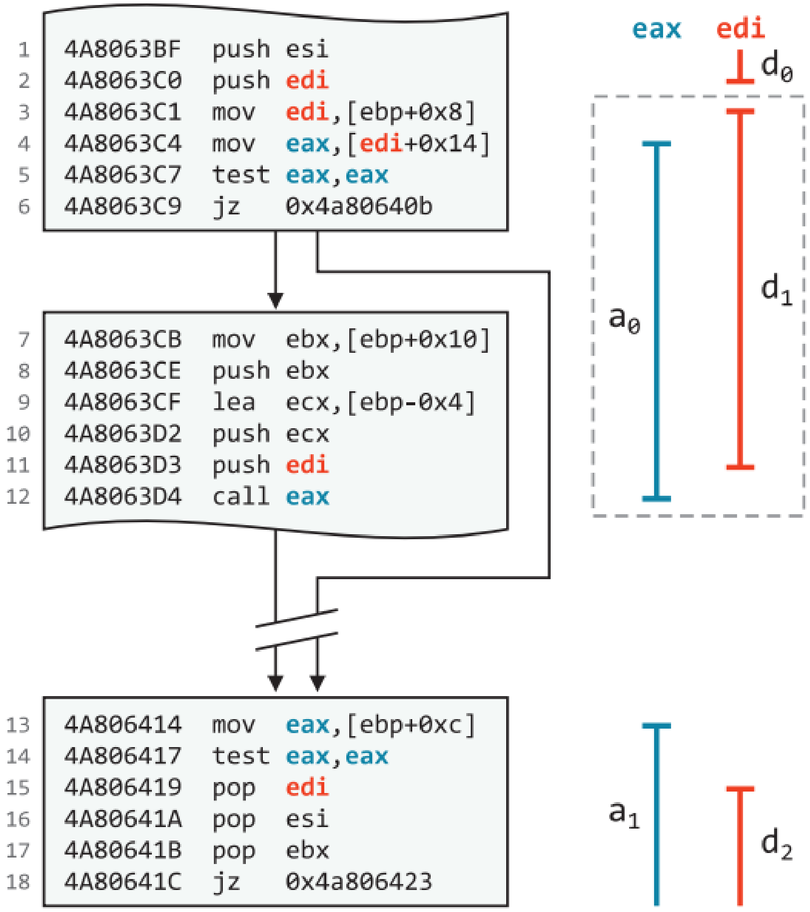
- 手段3：寄存器压栈顺序随机化

```
4A834B3B   0  push ebx
4A834B3C  -4  push esi
4A834B3D  -8  mov  ebx,ecx
4A834B3F  -8  push edi
4A834B40  -C  mov  esi,edx
...
4A834B7C  -C  pop  edi
4A834B7D  -8  pop  esi
4A834B7E  -4  pop  ebx
4A834B7F   0  ret
```

```
4A834B3B  push edi
4A834B3C  push ebx
4A834B3D  push esi
4A834B3E  mov  ebx,ecx
4A834B40  mov  esi,edx
...
4A834B7C  pop  esi
4A834B7D  pop  ebx
4A834B7E  pop  edi
4A834B7F  ret
```

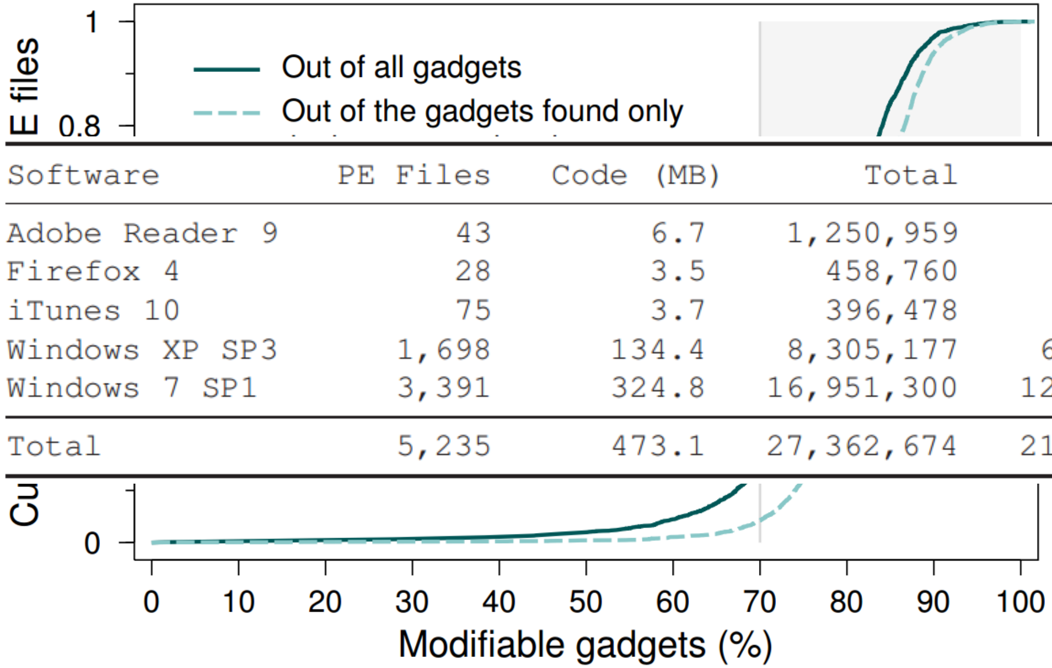
Smashing the gadgets

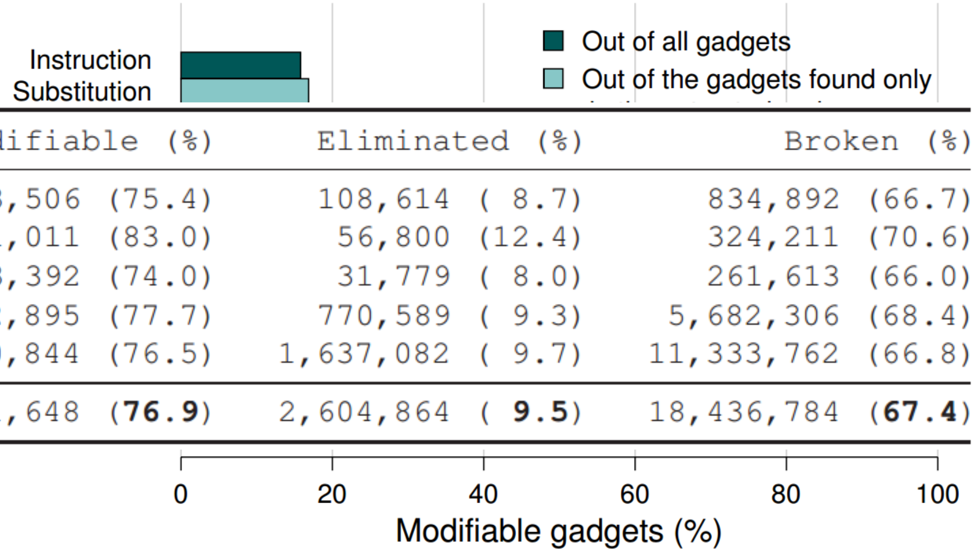
- 手段4：寄存器重分配



Smashing the gadgets

- Smashing the gadgets的防御效果





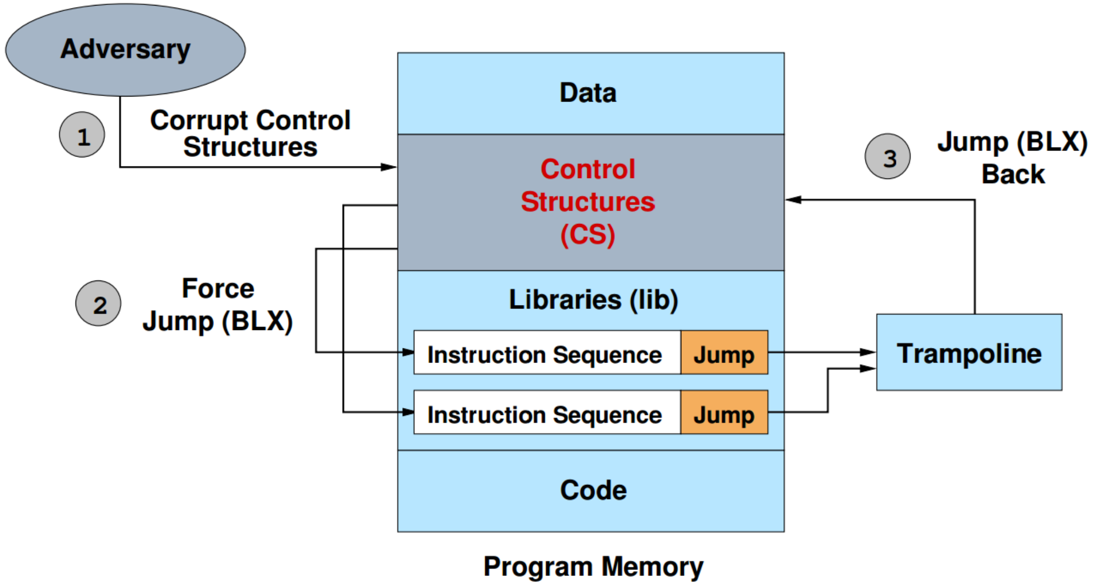
Software	PE Files	Code (MB)	Total	Modifiable (%)	Eliminated (%)	Broken (%)
Adobe Reader 9	43	6.7	1,250,959	943,506 (75.4)	108,614 (8.7)	834,892 (66.7)
Firefox 4	28	3.5	458,760	381,011 (83.0)	56,800 (12.4)	324,211 (70.6)
iTunes 10	75	3.7	396,478	293,392 (74.0)	31,779 (8.0)	261,613 (66.0)
Windows XP SP3	1,698	134.4	8,305,177	6,452,895 (77.7)	770,589 (9.3)	5,682,306 (68.4)
Windows 7 SP1	3,391	324.8	16,951,300	12,970,844 (76.5)	1,637,082 (9.7)	11,333,762 (66.8)
Total	5,235	473.1	27,362,674	21,041,648 (**76.9**)	2,604,864 (**9.5**)	18,436,784 (**67.4**)

Smashing the gadgets

- 参考文献：Pappas V, Polychronakis M, Keromytis A D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization[C]//Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012: 601-615.
- 类似的其他研究：Li J, Wang Z, Jiang X, et al. Defeating return-oriented rootkits with return-less kernels[C]//Proceedings of the 5th European conference on Computer systems. ACM, 2010: 195-208.
- 但是，这一切仍然并没有什么效（**luan**）果（**yong**）**……**

不“返回”的返回导向编程

- 核心思想：既然你们针对ret（0xC3），那么我想出不用ret也可以的办法
  - 利用update-load-branch指令序列“pop x; jmp *x”
  - 但是，update-load-branch序列并不像ret那样常见，因此要采用“蹦床”机制



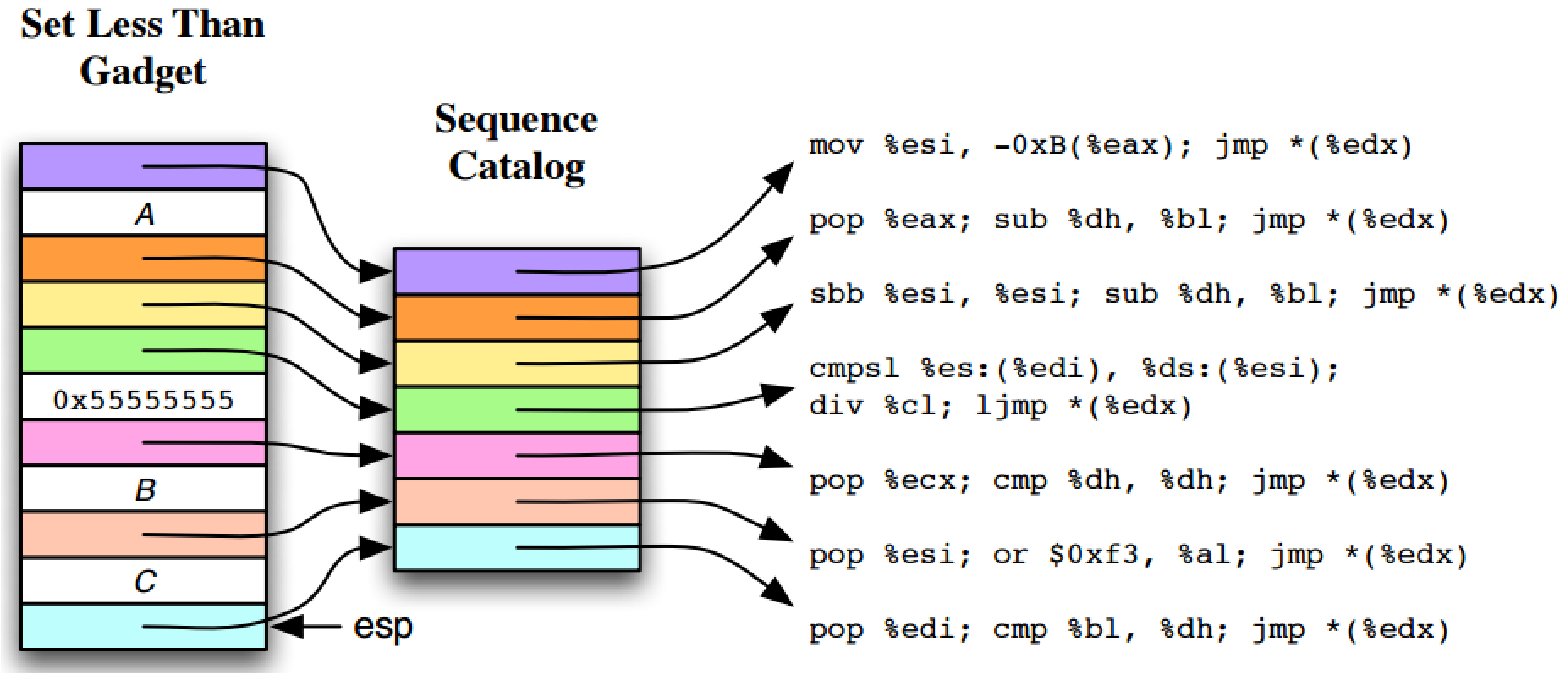
不“返回”的返回导向编程

- ROP without return之图灵完整性：可用资源及trampoline

pop %eax;	sub %dh, %bl;	jmp $*$(%edx)
pop %ecx;	cmp %dh, %dh;	jmp $*$(%edx)
pop %ebp;	or $0xF3, %al;	jmp $*$(%edx)
pop %esi;	or $0xF3, %al;	jmp $*$(%edx)
pop %edi;	cmp %bl, %dl;	jmp $*$(%edx)
pop %esp;	or %edi, %esi;	jmp $*$(%eax)
popad;	cld;	ljmp $*$(%edx)

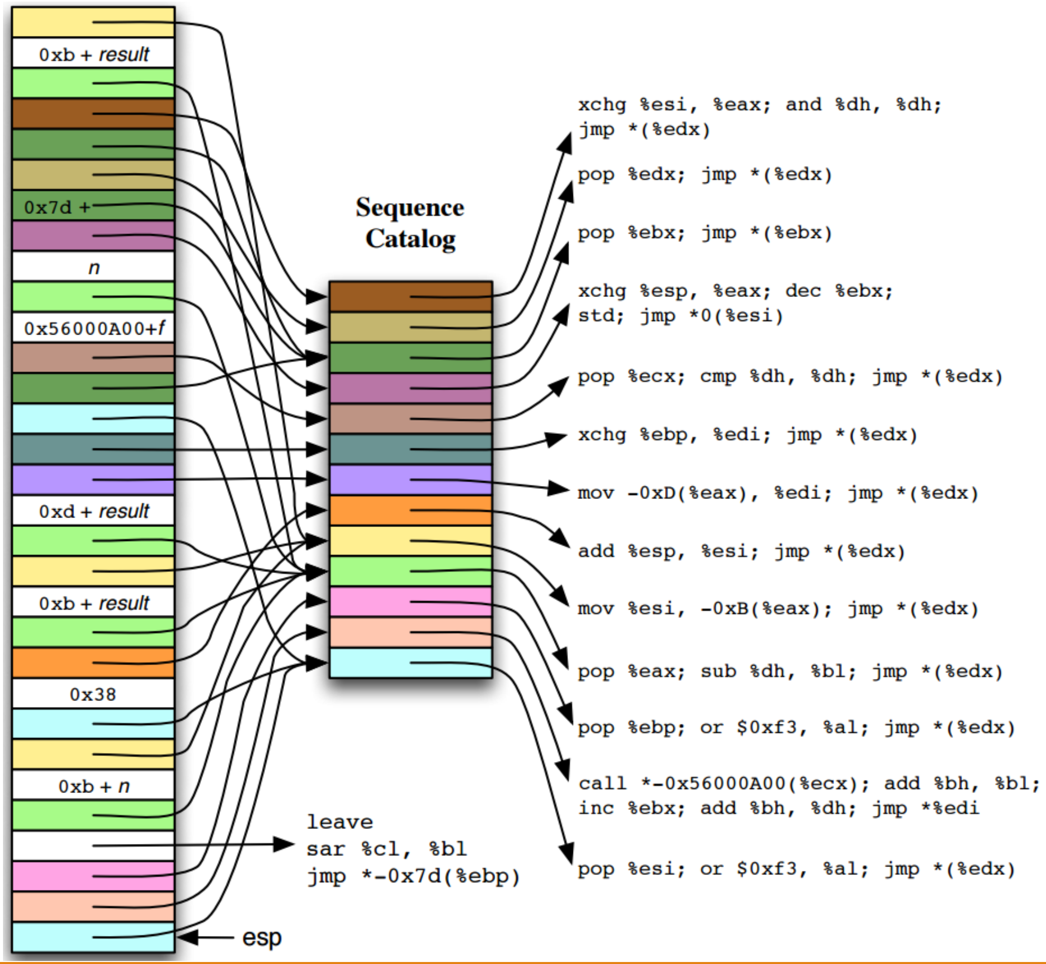
不“返回”的返回导向编程

- ROP without return之图灵完整性：条件分支



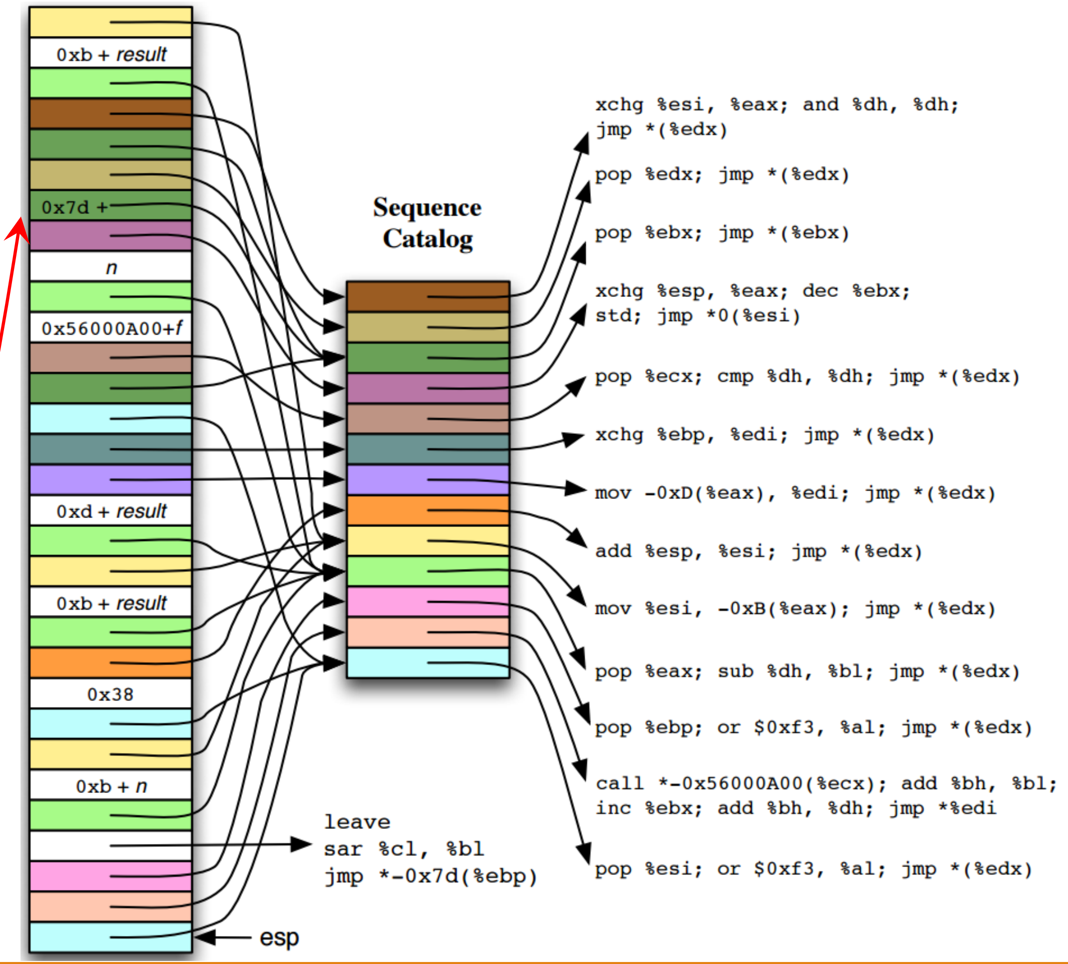
不“返回”的返回导向编程

- ROP without return之图灵完整性：函数调用
  - 在esi中载入call-jmp序列的地址
  - 在ebp中载入leave-jmp序列的地址
  - 在eax中载入n+偏移量
  - 将call-jmp序列的地址存储至地址n
  - 改写esi，使其存储“返回地址”
  - esi值写入result位置后，再读出至edi
  - 交换使返回值存入ebp，leave指令换入edi



不“返回”的返回导向编程

- ROP without return之图灵完整性：函数调用
  - 在esi中载入pop-jmp序列的地址
  - 在ecx中载入函数入口地址
  - 在eax中载入地址n
  - 交换esp和eax，栈指针指向n（函数地址）
  - edi处的leave指令将使函数“返回”至



What’s next?

- 控制流完整性保护技术
  - 理念和思路
  - 不同类型的设计方案