# 静态反汇编

孙聪

网络与信息安全学院

2020-09-21

# 课程内容

- 概述
- x86与x64体系结构
- 静态反汇编
- PE/ELF文件格式
- Windows系统编程基础
- DLL注入
- API钩取
- 调试工具与调试技术
- 混淆技术

# 提要

# 提要

# IA-32指令编码格式

| | Inst pre-fix | Opcode | Mode R/M | SIB | Displace-ment | Data Element |
|---|---|---|---|---|---|---|
| 长度(字节) | 0-4 | 1-3 | 0-1 | 0-1 | 0-4 | 0-4 |
| 解释 | 指令前缀，对指令补充说明，可选 | 指令操作码 | 操作数类型 | 辅助Mode R/M，计算地址偏移 | | 立即数 |

- 指令前缀：用于REP，跨段指令等情况
- 指令操作码：定义指令行为。指令助记符与操作码一一对应
- Mode R/M：操作数类型。R：寄存器；M：内存单元
- 立即数：用于操作数为常量值的情况

# IA-32指令编码格式

| | Inst pre-fix | Opcode | Mode R/M | SIB | Displace-ment | Data Element |
|---|---|---|---|---|---|---|
| 长度(字节) | 0-4 | 1-3 | 0-1 | 0-1 | 0-4 | 0-4 |
| 解释 | 指令前缀，对指令补充说明，可选 | 指令操作码 | 操作数类型 | 辅助Mode R/M，计算地址偏移 | | 立即数 |

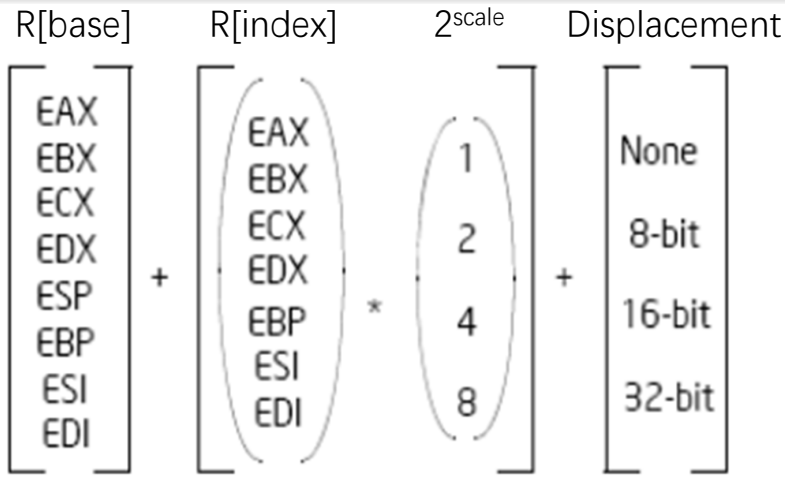
- SIB: Scale-Index-Base
- Displacement: 用于辅助SIB
  - Address = Reg[Base] + Reg[Index]$\times 2^{\text{Scale}}$ + Displacement

# IA-32指令编码格式

## 典型内存寻址方式

- Address in register
- Address = displacement
- Address = R[base] + displacement
- Address = R[base] + R[index]$\times 2^{scale}$, (scale = 0, 1, 2, or 3)
- Address = R[base] + R[index]$\times 2^{scale}$ + displacement

例: mov eax, [esi + ecx*4 + 4]



# 反汇编器

## 功能

将二进制文件中的机器码转化为汇编代码或等价的中间语言代码

## 分类

- 动态反汇编器
  - 执行二进制码，记录执行路径，并对记录的执行路径进行解码
  - 精确，可以对混淆的二进制进行反汇编
  - 记录路径需要时间，一次只能反汇编一条执行路径
- 静态反汇编器
  - 不执行二进制文件而直接对其反汇编
  - 优点：快，能够覆盖多于一条执行路径
  - 缺点：复杂，难以处理代码混淆

# 反汇编的基本操作单元

## 解码函数（decode）

- decode(code, offset) ⇒ (inst_str, inst_len)
- 参数
  - code：二进制码字节队列
  - offset：下一条汇编指令在此字节队列中的起始位置偏移量
- 返回值
  - inst_str：当前解码出的汇编指令
  - inst_len：当前解码出的汇编指令的长度

例，二进制码code=“6A 03 83 C4 0C B8 CC CC CC CC”，
初始offset=0

- decode(code, 0) ⇒ (“push 3”, 2)，解码“6A 03”
- decode(code, 2) ⇒ (“add esp, 0x0C”, 3)，解码“83 C4 0C”
- decode(code, 5) ⇒ (“mov eax, 0xCCCCCCCC”, 5)，解码“B8 CC CC CC CC”

# 提要

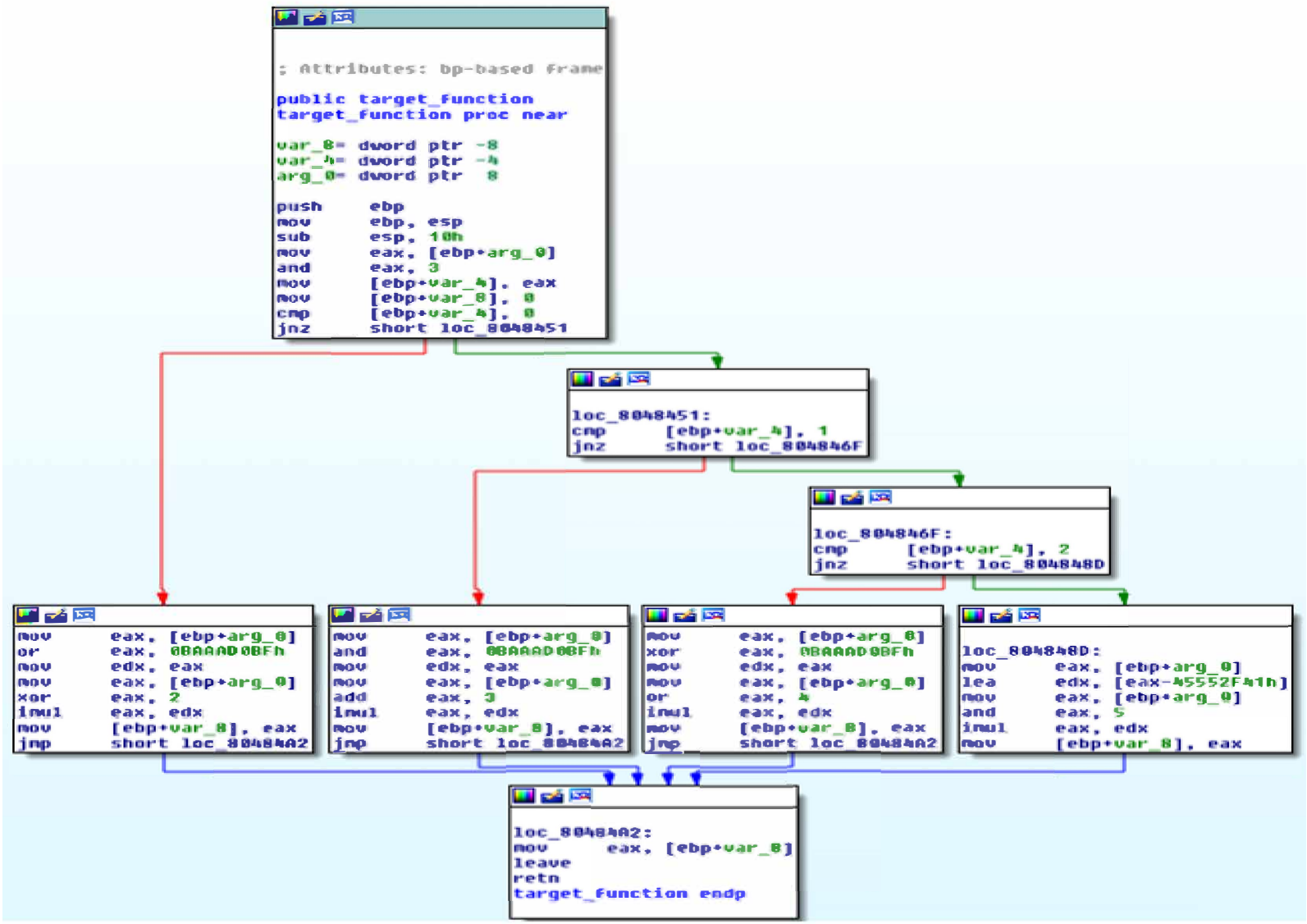
# 静态反汇编器

- 输入：一个二进制文件
- 目标：对于该二进制文件中的可执行节区(sections)进行反汇编
  - 可以使用该二进制文件中的其他信息，如符号表（symbol tables），如果符号表可用
- 输出：一个控制流图（Control-Flow Graph，CFG）

# 控制流图

- 节点：由一系列汇编指令组成的基本块
  - 一个基本块由一系列顺序执行的汇编代码组成，其间没有跳转指令，也没有其他外部跳转指令以其间为跳转目标
- 有向边：连接各基本块
  - 从基本块b1到基本块b2的有向边的含义是：在执行完b1后，有可能开始执行b2
- 从一个基本块可以发出多条有向边
  - 例如，当该基本块的最后一条指令为条件跳转指令时

# 控制流图示例


; Attributes: bp-based frame
public target_function
target_function proc near
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr 8
push ebp
mov ebp, esp
sub esp, 10h
mov eax, [ebp+arg_0]
and eax, 3
mov [ebp+var_4], eax
mov [ebp+var_8], 0
cmp [ebp+var_4], 0
jnz short loc_8048451
loc_8048451:
cmp [ebp+var_4], 1
jnz short loc_804846F
loc_804846F:
cmp [ebp+var_4], 2
jnz short loc_804848D
mov eax, [ebp+arg_0]
or eax, 0BAAAD0BFh
mov edx, eax
mov eax, [ebp+arg_0]
xor eax, 2
imul eax, edx
mov [ebp+var_8], eax
jmp short loc_80484A2
mov eax, [ebp+arg_0]
and eax, 0BAAAD0BFh
mov edx, eax
mov eax, [ebp+arg_0]
add eax, 3
imul eax, edx
mov [ebp+var_8], eax
jmp short loc_80484A2
mov eax, [ebp+arg_0]
xor eax, 0BAAAD0BFh
mov edx, eax
mov eax, [ebp+arg_0]
or eax, 4
imul eax, edx
mov [ebp+var_8], eax
jmp short loc_80484A2
loc_804848D:
mov eax, [ebp+arg_0]
lea edx, [eax-45552FA1h]
mov eax, [ebp+arg_0]
and eax, 5
imul eax, edx
mov [ebp+var_8], eax
loc_80484A2:
mov eax, [ebp+var_8]
leave
retn
target_function endp

# 静态反汇编的挑战

- 变长指令集
  - 对于精简二进制（stripped binaries），不知道其指令边界
  - 精简二进制：指不包含调试符号的二进制文件，相比非精简（unstripped）二进制更小，执行性能更高
- 数据嵌入在代码中
  - 例：编译器可能将跳转表（jump table）嵌入到代码节区
  - 注意：现代编译器已极少这样做，但obfuscator仍有可能这样做
- 间接跳转/调用（indirect jumps/calls）的目标需要静态分析
  - 例如“jmp 16[ebp]”，“call eax”

# 静态反汇编算法

## 主流算法

- 线性扫描（Linear sweep）
- 递归遍历（Recursive traversal）
- 混合方法

## 考量因素

- 从何处开始反汇编
- 如何选择下一条反汇编的指令
- 如何区分代码和数据
- 何时终止反汇编

# 线性扫描

## 原理

- 从代码节区的入口点开始扫描
- 顺序逐一解码指令，直到代码节区末尾；如果此过程中遇到解码出非法指令的情况，则停止

例： gdb，objdump使用线性扫描

# 线性扫描算法

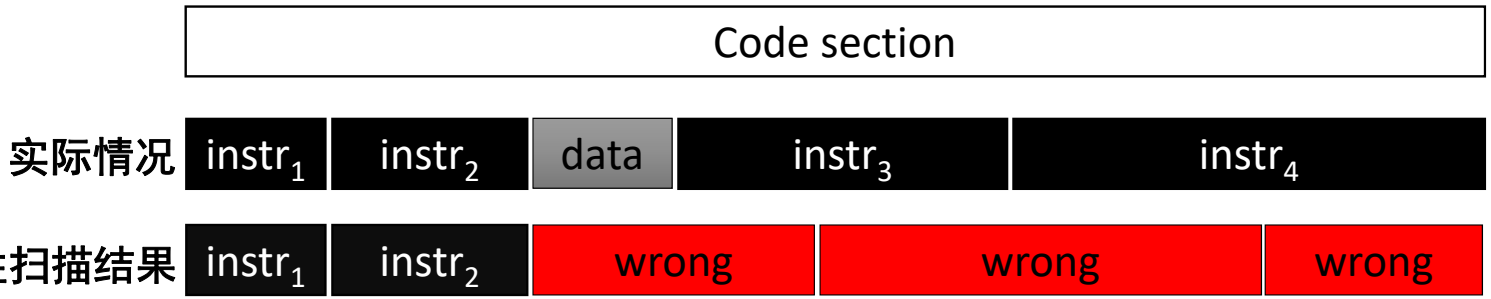
- 输入
  - code: 代码节区中的字节
  - codeSize: 代码节区的大小

```
currOffset=0;
instrSet=∅;
while (currOffset < codeSize) {
    (instr, size) = decode(code, currOffset);
        // 假定decode失败(或到达代码缓冲区结尾)时抛出异常
    instrSet = instrSet ∪ {(currOffset, instr, size)};
    currOffset += size;
}
buildCFG(instrSet); //构建基本块、加边, 从而生成CFG
```

# 线性扫描的特点

- 优点：简单，易实现
- 缺点：如果代码节区中有数据，会将数据误识别为代码

- 例：




- $inst_2$可能是一个jump指令，跳到下一条指令；或是一个ret

# 递归遍历

## 基本思想

- 沿着反汇编过程中生成的控制流图对指令进行反汇编

# 递归遍历算法

- 输入：code（代码节区中的字节）

```
init_offset = 0;
worklist = {init_offset}; processed = ∅;
while (worklist ≠ ∅) {
    offset = removeOneTarget(worklist);
    processed = processed ∪ {offset};
    (instr, size) = decode(code, offset);
    switch (instr) {
        case non-control-flow-instr: add_target(offset+size);
        case unconditional-jmp(dest): add_target(dest);
        case cond-jmp(dest_1, dest_2): add_target(dest_1); add_target(dest_2);
        case func-call(dest_1, ..., dest_k):
                                    add_target(dest_1); ... add_target(dest_k);
        ...
    }
}
procedure add_target(offset):
    if (offset ∉ processed) then worklist = worklist ∪ {offset};
```

# 递归遍历的特点

- 优点：能够容忍数据嵌入在代码节区中的情况
- 缺点：对于间接跳转/调用(indirect jumps/calls)，很难决定控制流边到底指向哪里

- IDA Pro使用递归遍历
  - 商业级的反汇编器
  - 生成不完整的控制流图
  - 控制流图不完整的原因是对于间接跳转/调用语句不存在控制流图边

# 提要

# 对混淆代码的反汇编

"Static Disassembly of Obfuscated Binaries" by Kruegel et al at 2004 Usenix Security. https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/full_papers/kruegel/kruegel_html/disassemble.html

- 线性扫描与递归遍历结合
- 使用启发式方法(heuristics)从初始CFG中移除伪节点

混淆二进制(obfuscated binaries)

- 无符号信息
- 经过混淆(例如向代码节区插入不可达的垃圾数据)
- 例如, "ins1; ins2" $\Rightarrow$ "ins1; mov eax, someConst; jmp eax; junk bytes; ins2"

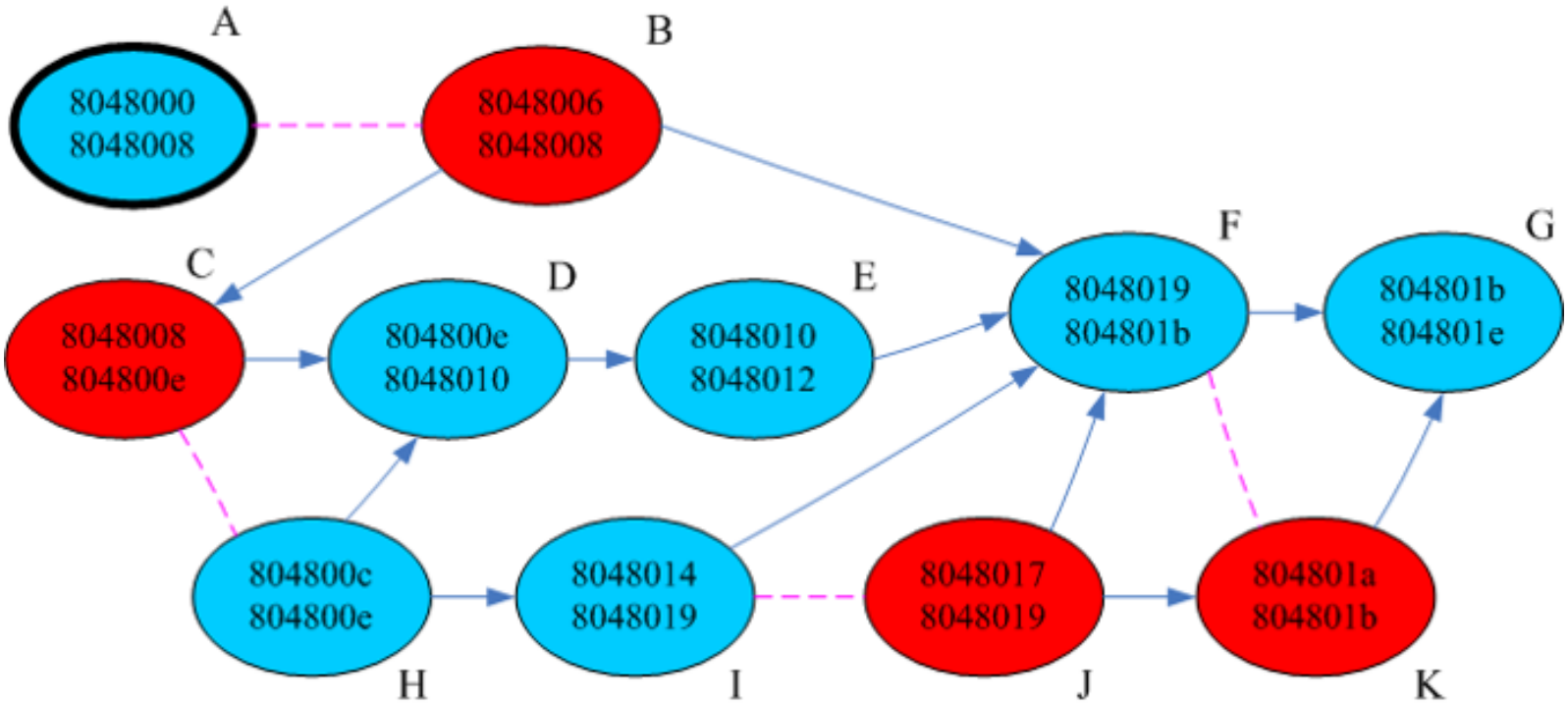
# 算法

- 识别函数
  - 在二进制代码中匹配“常用序言”(common prologs)语句
  - 常用序言语句：“push %ebp; mov %esp, %ebp”，编码为0x55 89 e5
- 构造过程内控制流图(intra-procedural CFGs)
  - 从每一个地址开始解码；丢弃非法指令
    (目的是为了容忍变长指令集，可能导致互相重叠的指令)
  - 识别函数中的所有直接跳转(direct jump)指令
  - 选出跳转候选指令：a)跳转目标在函数内部的直接跳转指令；
    b)直接条件分支指令
  - 通过将入口指令和跳转候选指令看作起始点，使用递归遍历方法，构造出初始CFG
- 在初始CFGs中解决块冲突
  - 通过5个步骤移除伪节点

# 示例程序

| | | | | Valid | Candidate |
|---|---|---|---|---|---|
| 8048000 | 55 | push | %ebp | x | |
| 8048001 | 89 e5 | mov | %esp, %ebp | x | |
| 8048002 | e5 e8 | in | e8,%eax | | |
| 8048003 | e8 00 00 74 11 | call | 19788008 <obfuscator> | x | |
| 8048004 | 00 00 | add | %al, %eax | | |
| 8048005 | 00 74 | add | | | |
| 8048006 | 74 11 | je | 8048019 | | x |
| ... | | | | | |
| 804800c | 75 06 | jne | 8048014 | x | x |
| ... | | | | | |
| 8048010 | eb 07 | jmp | 8048019 | x | x |
| ... | | | | | |
| 8048017 | 74 01 | je | 804801a | | x |
| 8048018 | 01 89 ec 5d c3 90 | add | %dh,ffffff89(%ecx,%eax,1) | | |
| 8048019 | 89 ec | mov | %ebp, %esp | x | |
| 804801a | ec | in | (%dx), %al | | |
| 804801b | 5d | pop | %ebp | x | |
| ... | | | | | |

# 初始控制流图

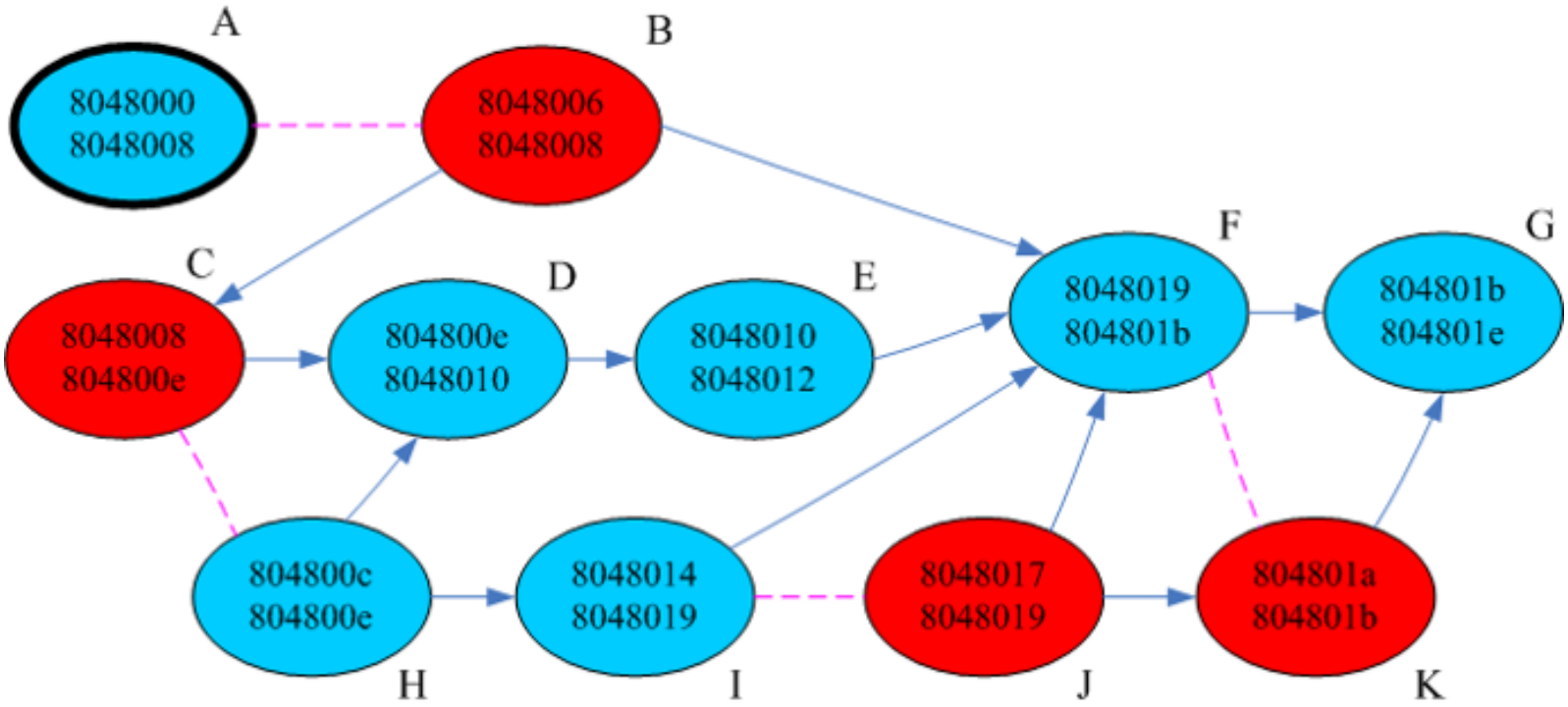



- 蓝色节点： 实际CFG的节点
- 红色节点： 伪节点
- 节点A: 入口节点
- 粉色虚线： 节点间存在冲突
- 实线箭头： 初始CFG中的边

# 解决块冲突

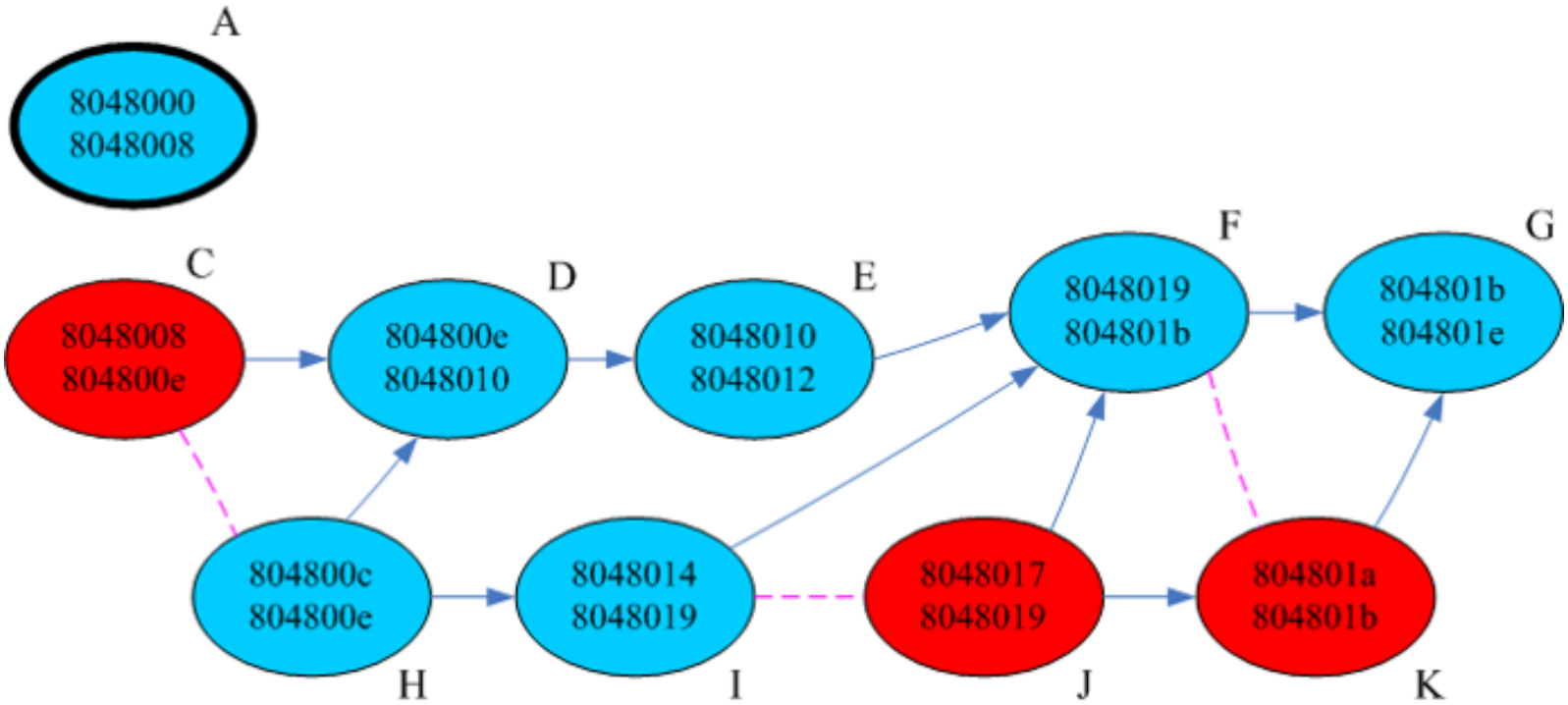
- 第一步：移除与合法节点冲突的节点
  - 入口节点(Entry node)必为合法节点
  - 从合法节点可达的节点必为合法节点
  - 与合法节点冲突的节点必为非法节点
- 第二步：移除冲突节点的祖先节点
  - 假定：合法节点的地址范围不会重叠
  - 如果两个相互冲突的节点有共同的祖先节点，则祖先节点必为非法
- 第三步：移除具有更少前驱节点的冲突节点(启发式方法)
  - 假定合法节点更紧密地集成在CFG中
  - 节点的前驱节点更多，说明与CFG结合得越紧密
- 第四步：移除具有更少直接后继节点的冲突节点(启发式方法)
  - 假定合法节点更紧密地集成在CFG中
  - 节点的直接后继更多，说明与CFG结合得更紧密
- 第五步：随机地移除互相冲突的节点
  - 随机地从两个互相冲突的节点中取出一个
  - 最差的情形

# 解决块冲突



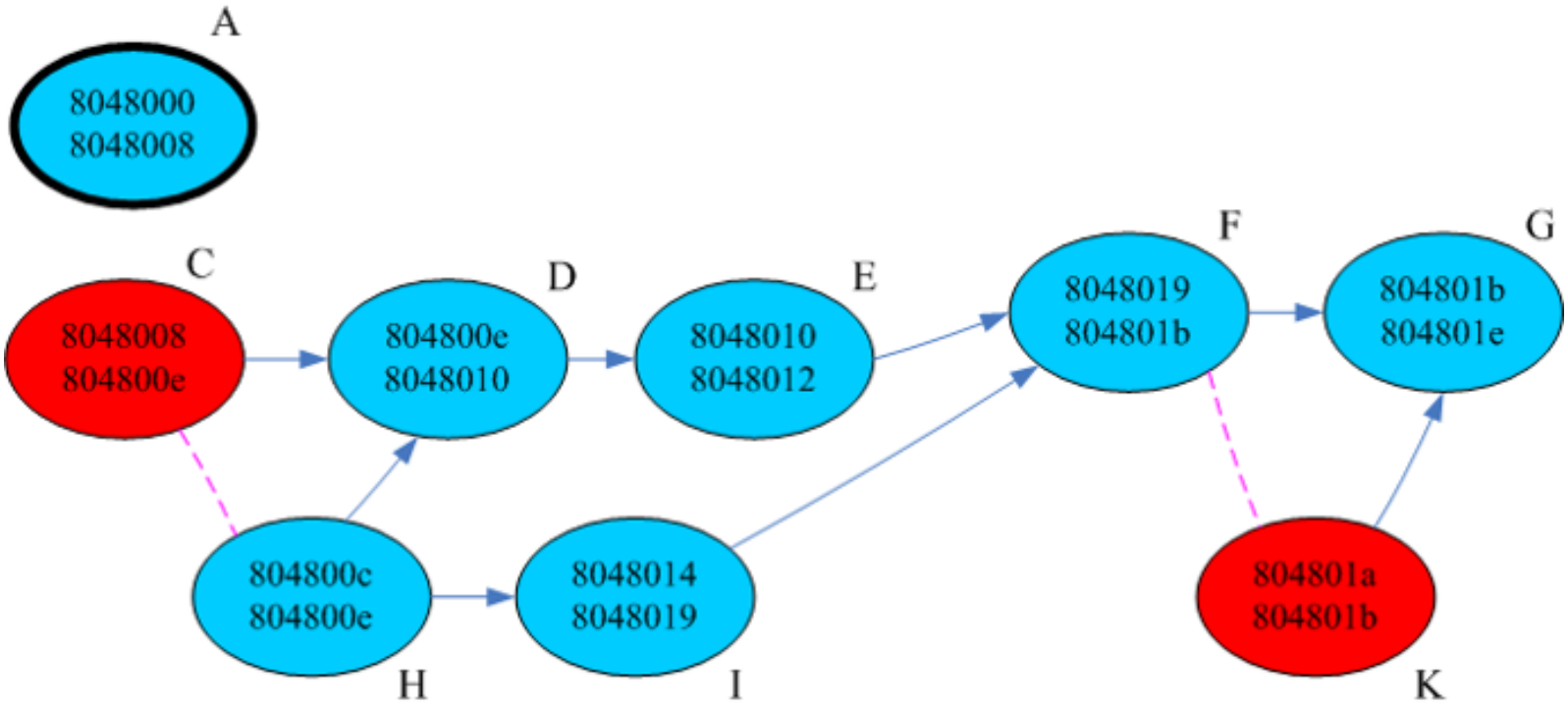

- 初始控制流图

# 解决块冲突



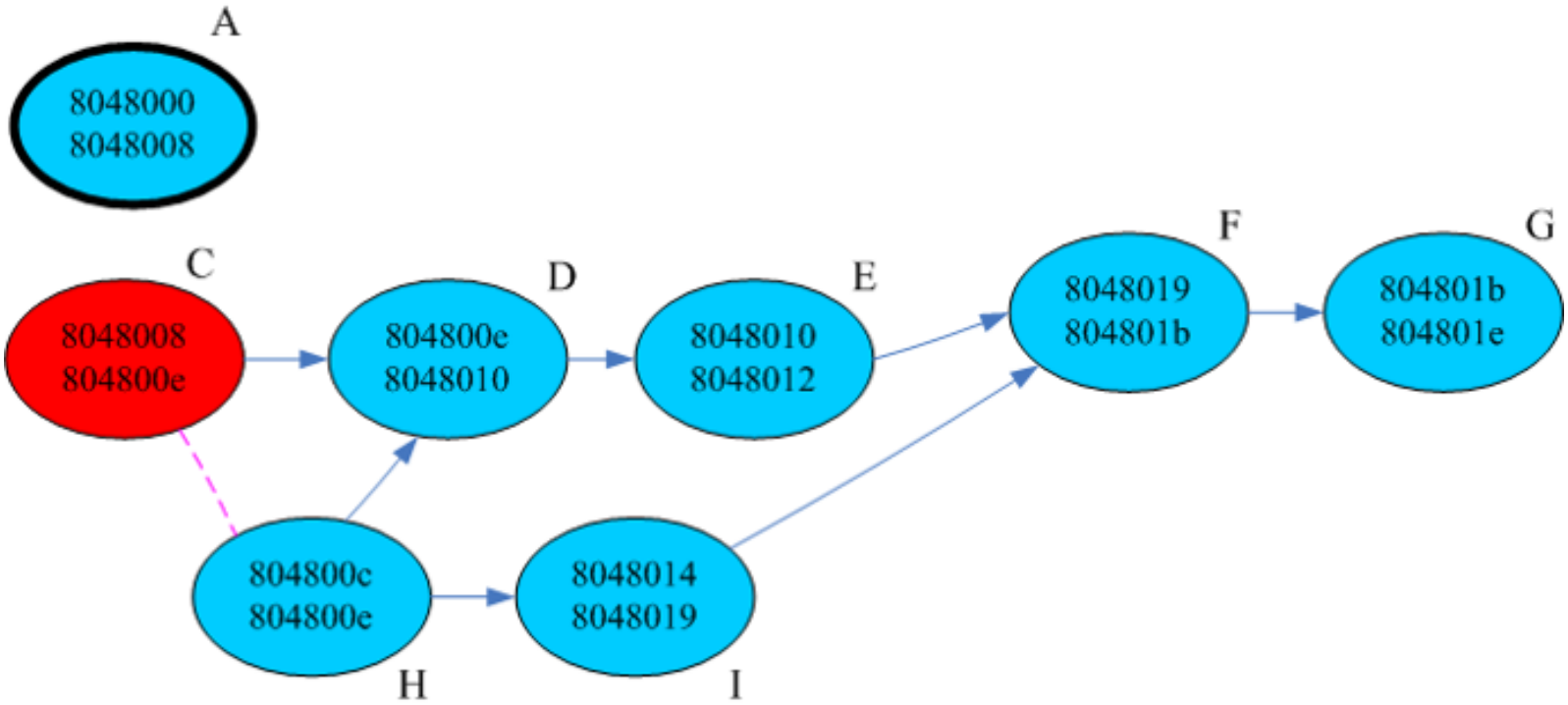

- 第一步后的CFG（移除了节点B）

# 解决块冲突



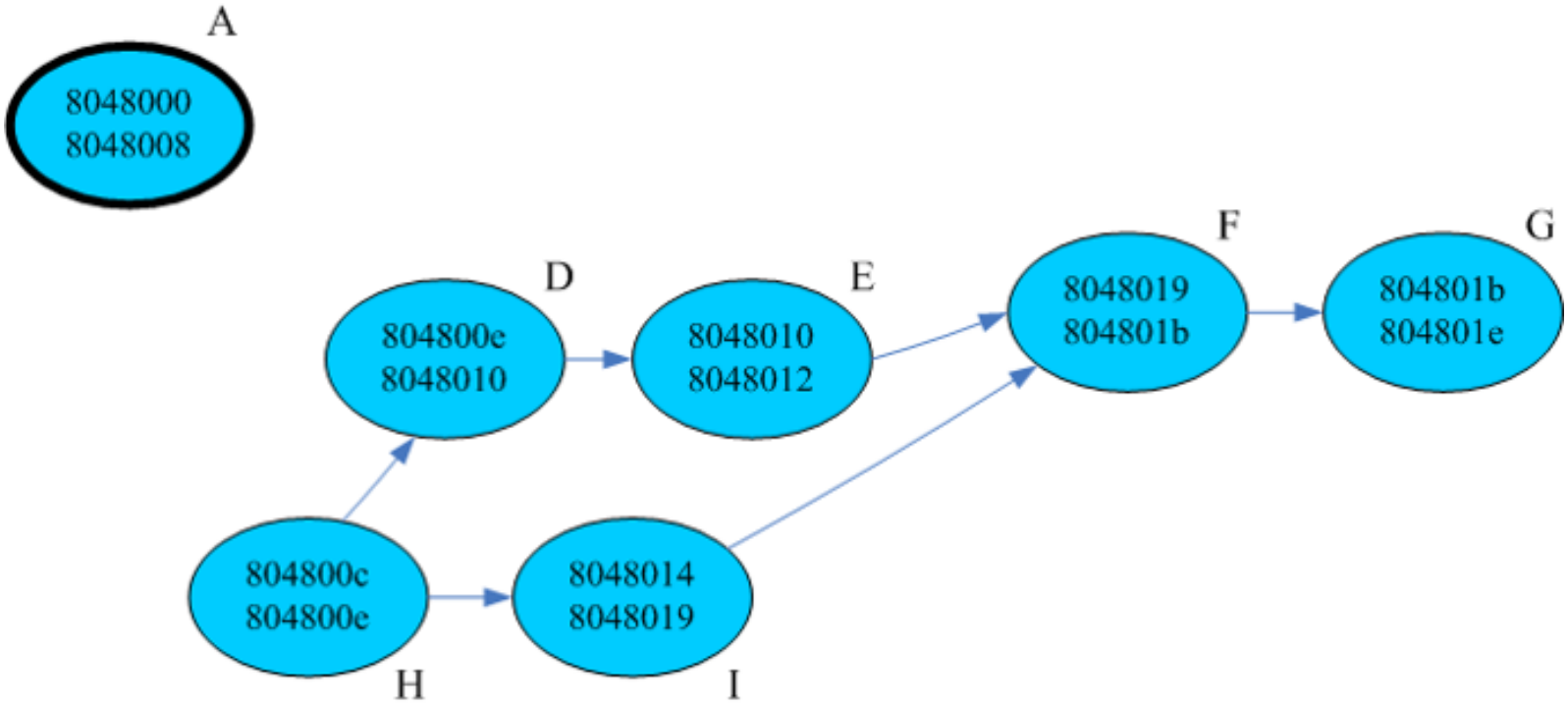

- 第二步后的CFG（移除了节点J）

# 解决块冲突




- 第三步后的CFG（移除了节点K）

# 解决块冲突




- 第四步后的CFG（移除了节点C）

# 该反汇编方法的精确性

| Program | Objdump | Linn/Debray | IDA Pro | This paper |
|---|---|---|---|---|
| compress95 | 56.07 | 69.96 | 24.19 | 91.04 |
| gcc | 65.54 | 82.18 | 45.09 | 88.45 |
| go | 66.08 | 78.12 | 43.01 | 91.81 |
| ljpeg | 60.82 | 74.23 | 31.46 | 91.60 |
| li | 56.65 | 72.78 | 29.07 | 89.86 |
| m88ksim | 58.42 | 75.66 | 29.56 | 90.39 |
| perl | 57.66 | 72.01 | 31.36 | 86.93 |
| vortex | 66.02 | 76.97 | 42.65 | 90.71 |
| Mean | 60.91 | 75.24 | 34.55 | 90.10 |

- 所有程序都经过一个混淆工具的混淆
- 被每个工具正确反汇编的指令的百分比

# 延伸阅读


- Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu: Shingled Graph Disassembly: Finding the Undecideable Path. PAKDD (1) 2014: 273-285
- Erick Bauman, Zhiqiang Lin, Kevin W. Hamlen: Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. NDSS 2018
- Matthew Smithson, Khaled Elwazeer, Kapil Anand, Aparna Kotha, Rajeev Barua: Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. WCRE 2013: 52-61
- Eui Chul Richard Shin, Dawn Song, Reza Moazzezi: Recognizing Functions in Binaries with Neural Networks. USENIX Security Symposium 2015: 611-626