

结构化签名和签名的结构化

funnywei@xfocus.org

2005-08-19



X'con 2005

内容

- ❖ 二进制比较中的图论知识
- ❖ 补丁代码对汇编结构的影响
- ❖ 三个层次的同构
- ❖ 处理编译优化
- ❖ **Demo**
- ❖ 一些问题



二进制比较中的图论知识

◆ 同构

- ◆ 与图论中的同构不太一样

◆ 结构化汇编代码的图

◆ 相关

- ◆ 控制流图分析

- ◆ 数据流分析



补丁代码对汇编结构的影响

- ◆ 增加限制函数
 - ◆ MS04-022 Task Schedule BOF
- ◆ 增加一个“if”来判断字符串的长度
 - ◆ MS04-011 IIS PCT BOF



三个层次的同构

- ◆ 三个层次

- ◆ 函数调用关系图级

- ◆ 控制流图级

- ◆ 指令级

- ◆ 两个阶段

- ◆ 初始化固定点

- ◆ 传播固定点



函数调用关系图级

◆ 签名

◆ 基本信息

- ◆ 基本块数
- ◆ 链接数
- ◆ 子调用数

◆ 结构信息

- ◆ 所有模式的素数乘积

◆ 限制属性

- ◆ 相同名字
- ◆ 签名唯一性
- ◆ 相同入度
- ◆ 相同字符串引用
- ◆ 递归函数
- ◆ 相同素数乘积



相同名字

- ❖ 如果两个函数具有相同的名字，可以作为一个匹配对。
- ❖ 我们必须排除：
 - ❖ `Unknow libname`
 - ❖ `Loc_XXXXXXXX`



签名的唯一性

- ◆ 计算两个函数签名的欧几里德距离
- ◆ 对于 \mathbf{x} 属于图 \mathbf{B} ，如果
$$\exists a \in A, \forall b \in A, b \neq a, 0 = |x-a| < |x-b|$$
- ◆ 增加 (\mathbf{x}, \mathbf{a}) 作为一个匹配对



相同入度

- ◆ 当两个函数具有相同的入度
- ◆ 计算它们签名的欧几里德距离
- ◆ 当它们满足

$$x \in B, \exists a \in A, \forall b \in A, b \neq a, 0 = |x - a| < |x - b|$$

- ◆ 增加 (x, a) 作为匹配对。



相同字符串引用

- ❖ 费时！
- ❖ 可以将字符串转换成**MD5**值再进行处理



相同素数乘积

费时！怎么办？

函数A的素数乘积可以表示为 $\Pi A = K \cdot 2^{64} + b$

函数B的素数乘积可以表示为 $\Pi B = j \cdot 2^{64} + c$

若 $b \neq c$, $\Pi A \neq \Pi B$

若 $b = c$, 我们可以认为 $\Pi A = \Pi B$, 正确？

总数 $l = C_{n+m-1}^n$, 这里 $\Pi c_1 \neq \Pi c_2 \dots \neq \Pi c_l$

$$\Pi A \equiv \Pi B \pmod{2^{64}} \text{ 小于等于 } \left(\frac{P_m^n}{2^{64}} - 1 \right) / C_{n+m-1}^n$$



结构化信息

◆ 结构化分析

◆ 结构模式

◆ 顺序模式

◆ 条件模式

◆ If-then, if-then-else, switch-case

◆ 循环模式

◆ Self loop, While loop, Endless loop, multiexit loop

◆ 非结构模式

◆ 非结构循环

◆ 非结构条件

◆ 增加结构信息到签名中



控制流图级

◆ 签名

◆ 基本信息

- ◆ 从入口到自身的最短路径上基本块个数
- ◆ 从自身到出口的最短路径上基本块个数
- ◆ 子调用数

◆ 结构信息

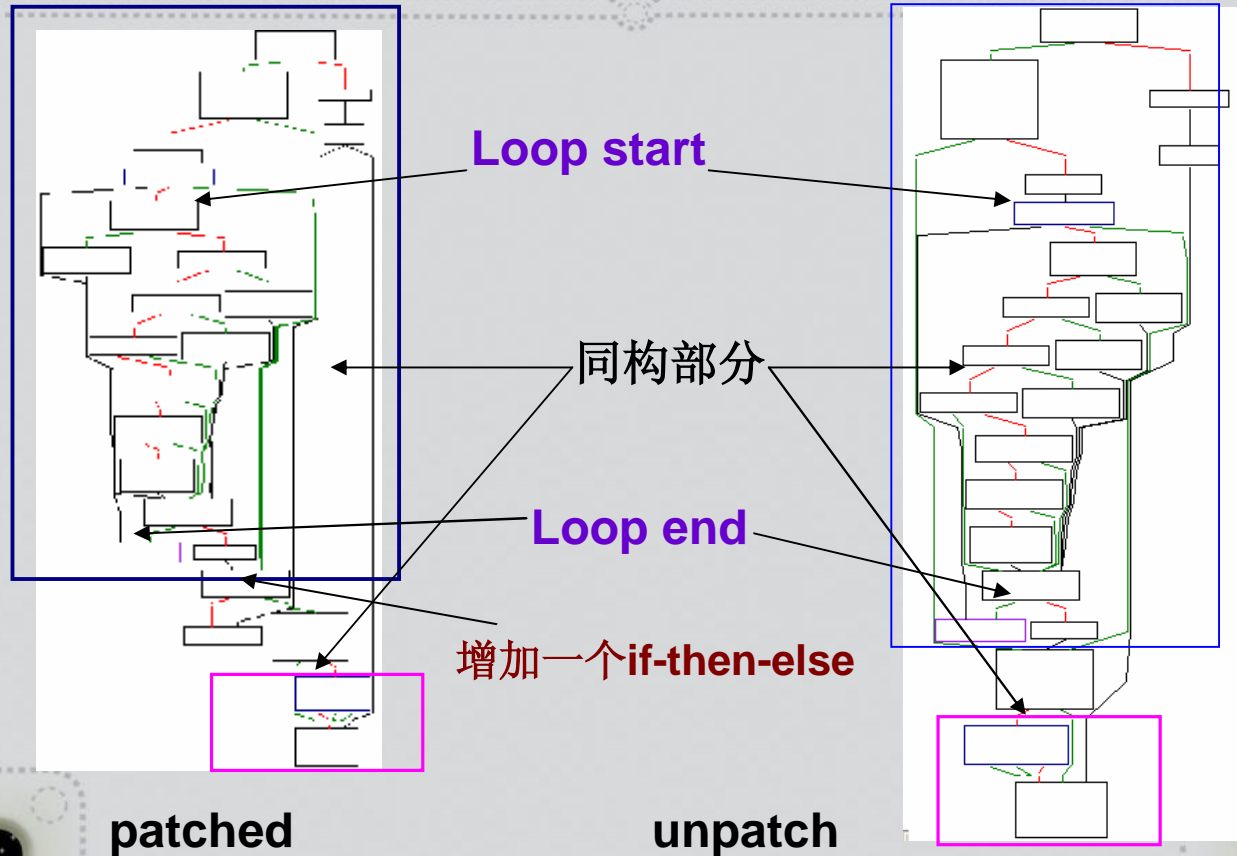
- ◆ 结构层次

◆ 限制属性

- ◆ 相同素数乘积
- ◆ 相同字符串引用
- ◆ 相同子调用



MS04-011 Pct1SrvHandleUniHello



patched

unpatch

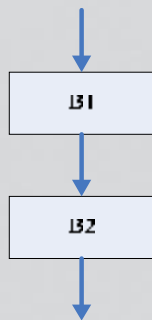


结构分析

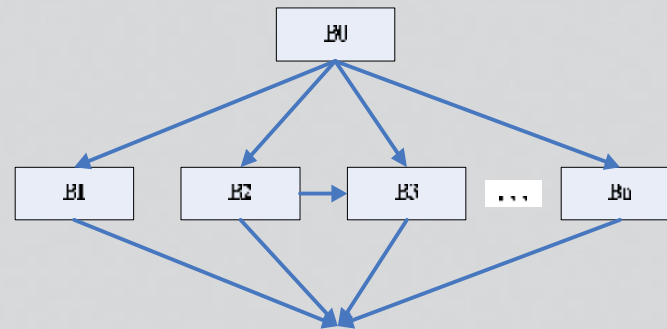
- ❖ 费时的!
- ❖ 但当我们需要“**visual diff** 两个函数流图的时候”，会非常的有用。
 - ❖ 在控制流图一级的同构。



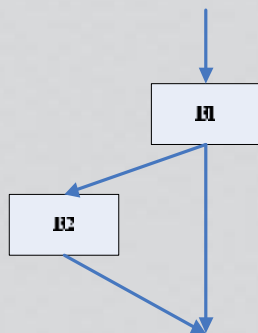
顺序和条件模式



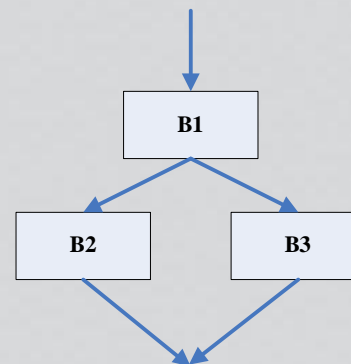
顺序，可以被忽略



Switch-case



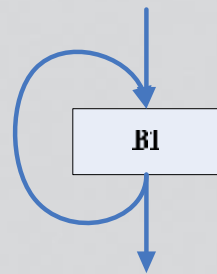
If-then



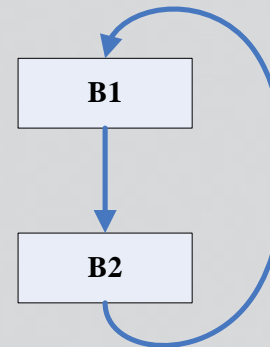
If-then-else



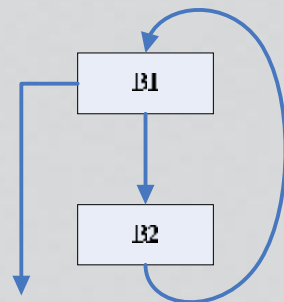
循环模式



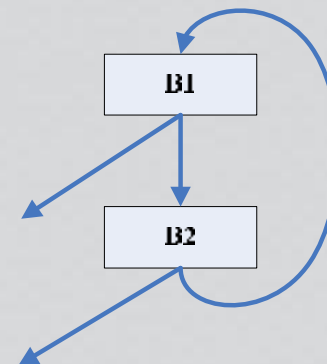
Self loop



Endless loop



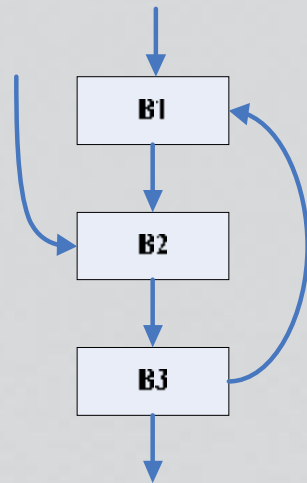
While loop



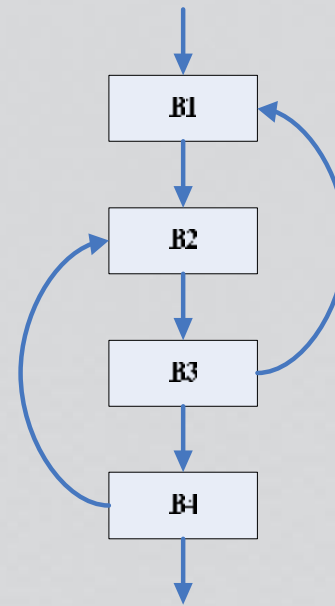
Multi exit



非结构循环



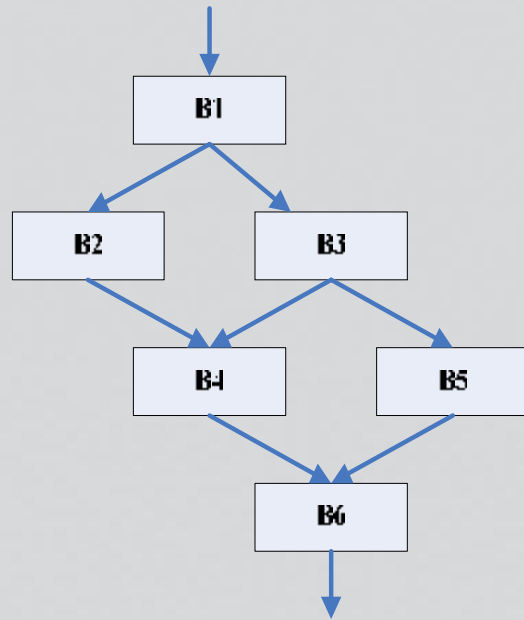
Multi entry



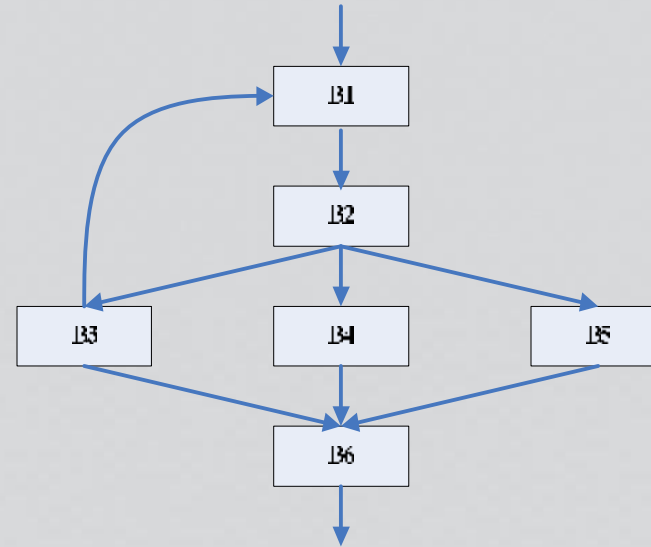
Overlapping



非结构条件 (1)



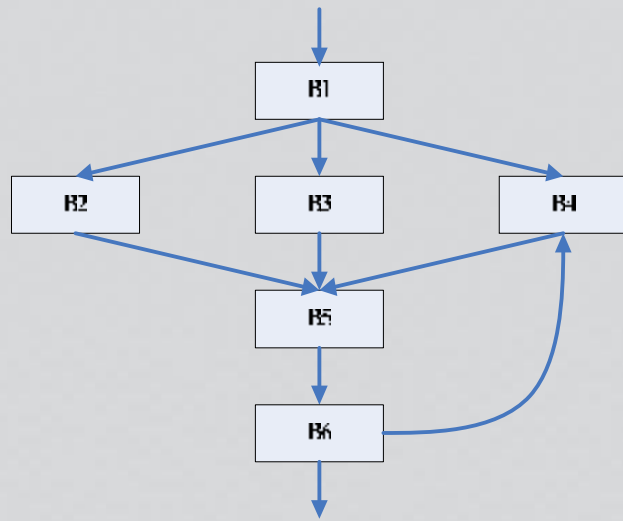
非结构 2-出度条件



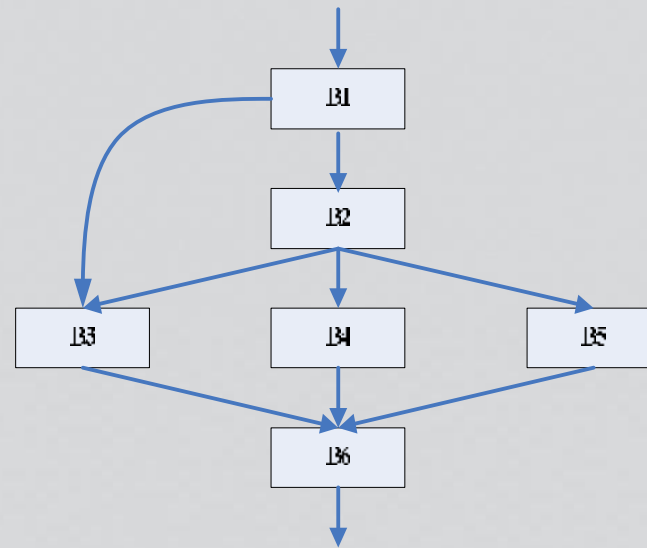
非结构 N-出度条件(1)



非结构条件 (2)



非结构N-出度条件 (2)

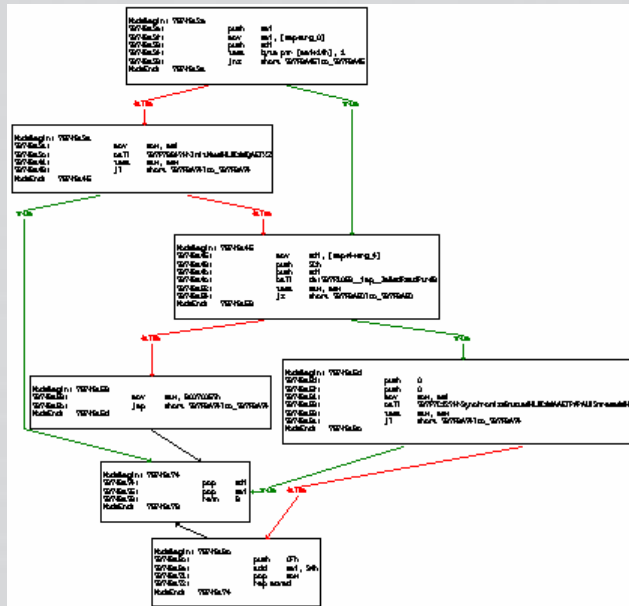


非结构N-出度条件(3)

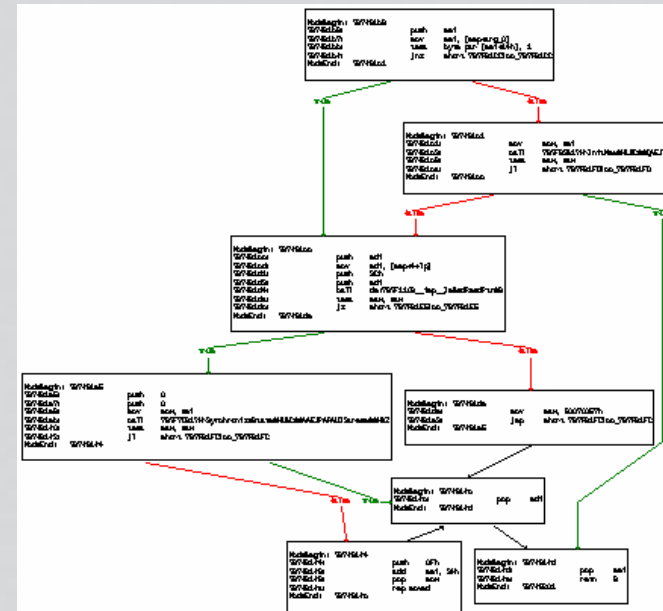


例子

MS 05-011 HLBC::GetBrowseWindowInfo



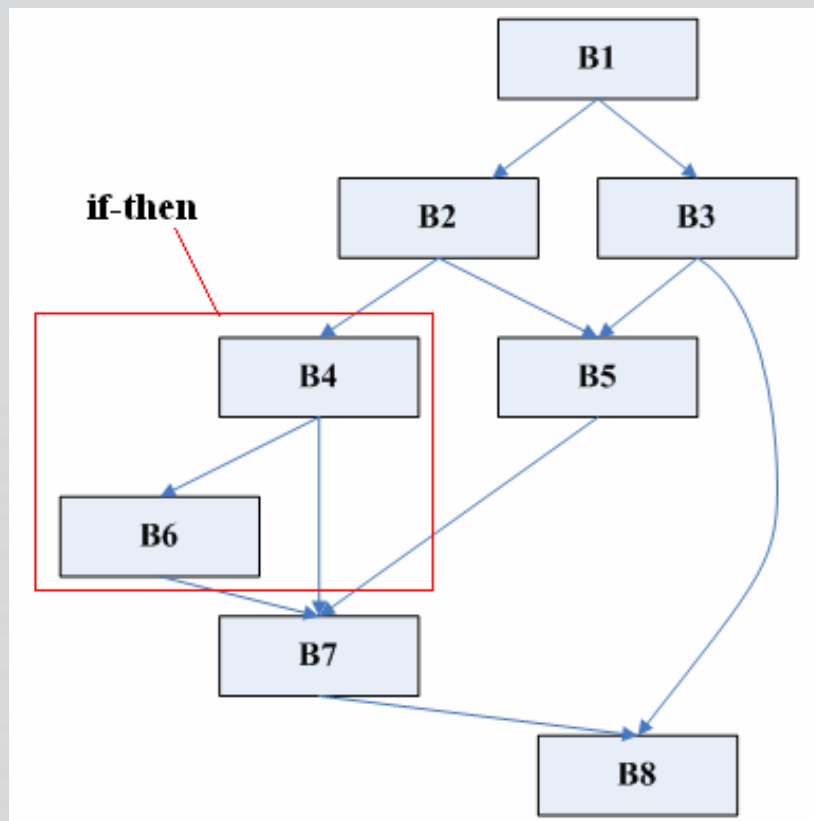
Unpatch Sig = 3::6::3



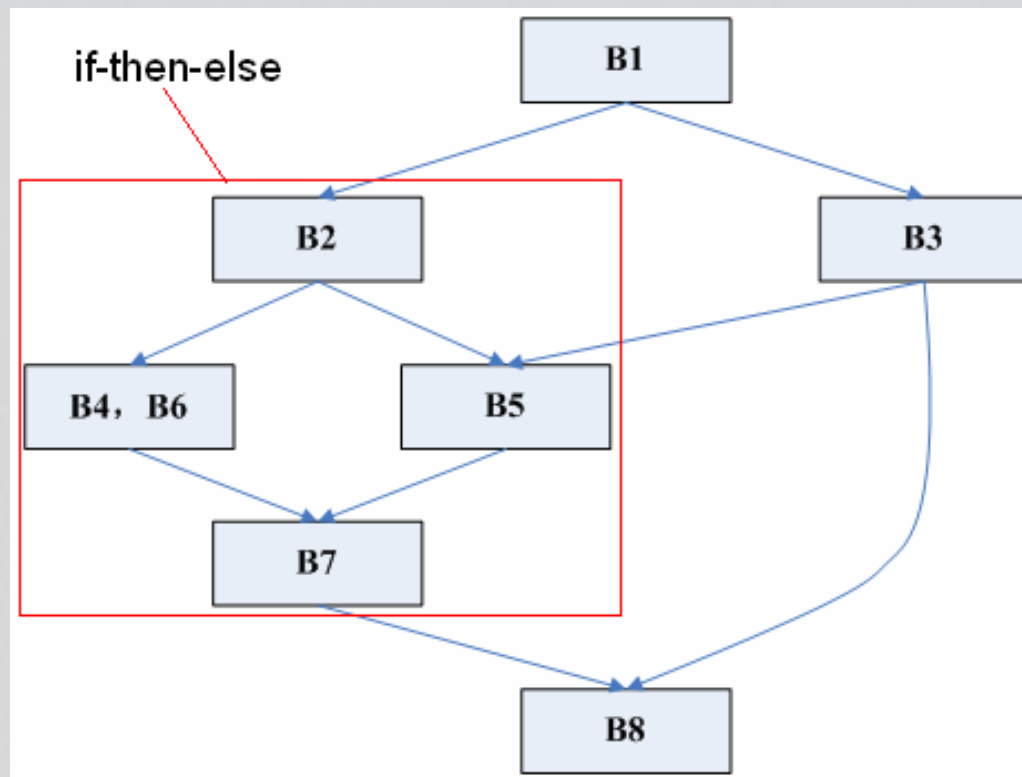
Patched Sig = 4::7::3



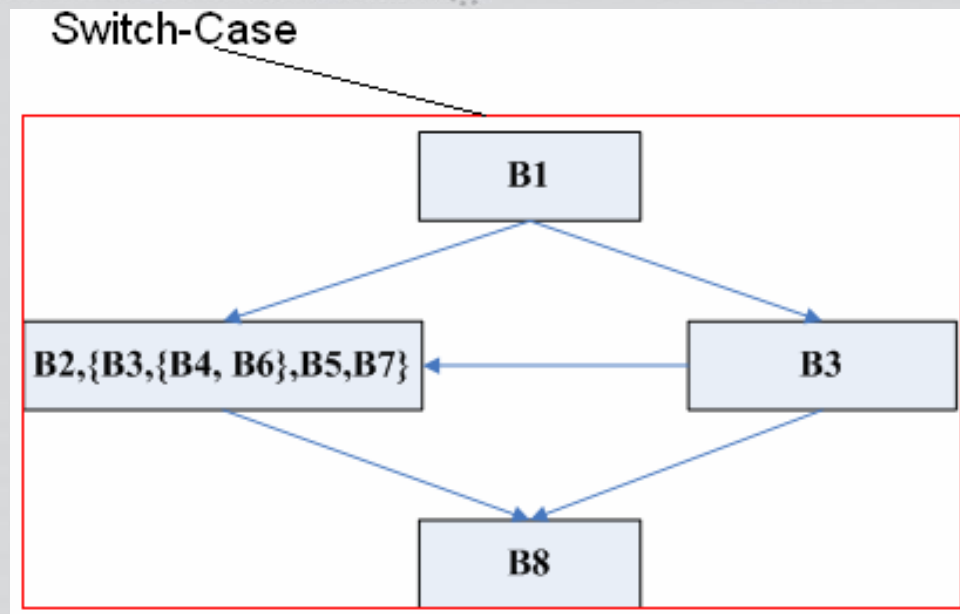
If-then 归约



If-then-else 归约



Switch-case 归约



另外一个的归约过程是相同的。



所以这两个函数相同

假设:

If-then := 素数 2

If-then-else := 素数 3

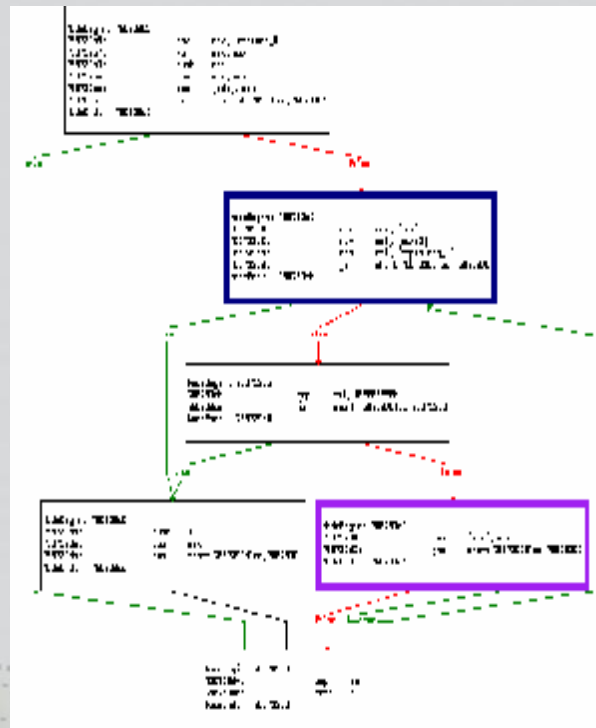
Switch-case := 素数 11

这两个的结构信息签名值 = 66

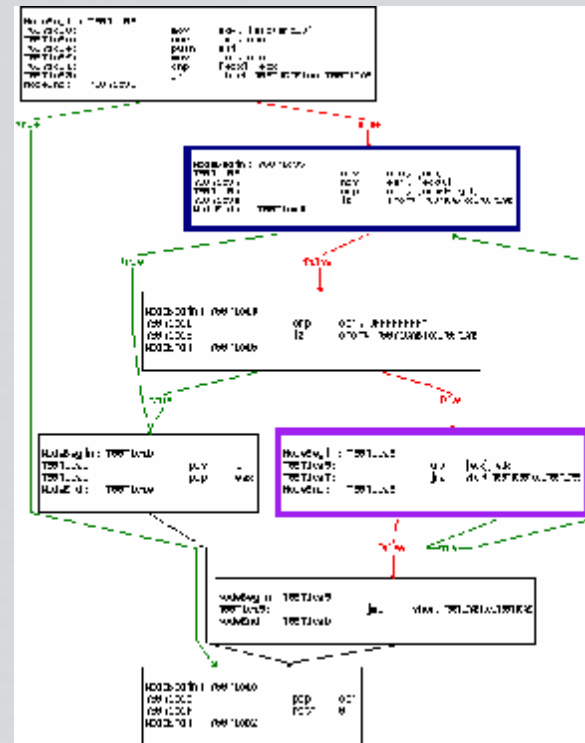


处理编译器优化

例子1: MsglsSessionInList



Sig = 6::9::0



Sig = 7::10::0



优化处理算法 1.0

对于控制流图G1，每个入度为1的节点，假设它的入边是 (y, x) ，出边是集合：

$$j = \{ \langle x, z_1 \rangle, \langle x, z_2 \rangle, \dots, \langle x, z_n \rangle \}$$

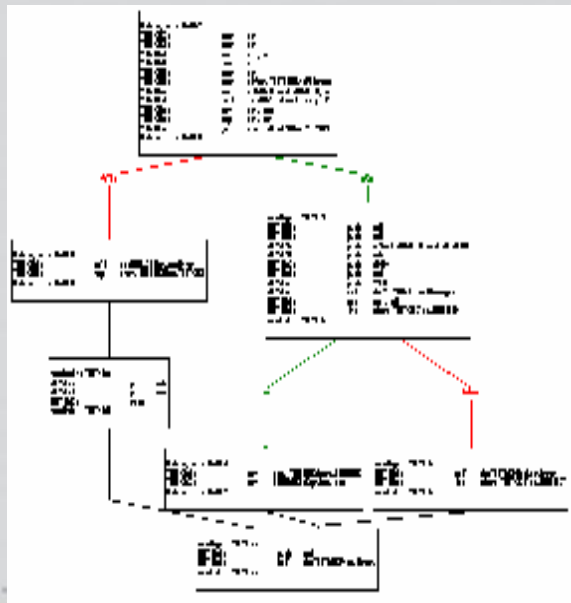
删除节点 x ，边 (y, x) 和所有节点 x 的出边，增加从 y 到 x 孩子的边。

使用1.0算法，例子1的两个签名都变成了 4::7::0



超过一个return节点，
在使用了算法1处理后

例子 2



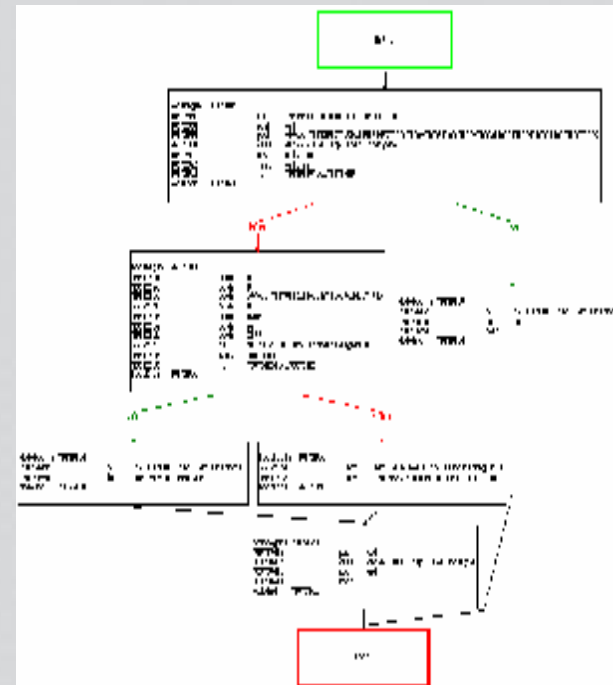
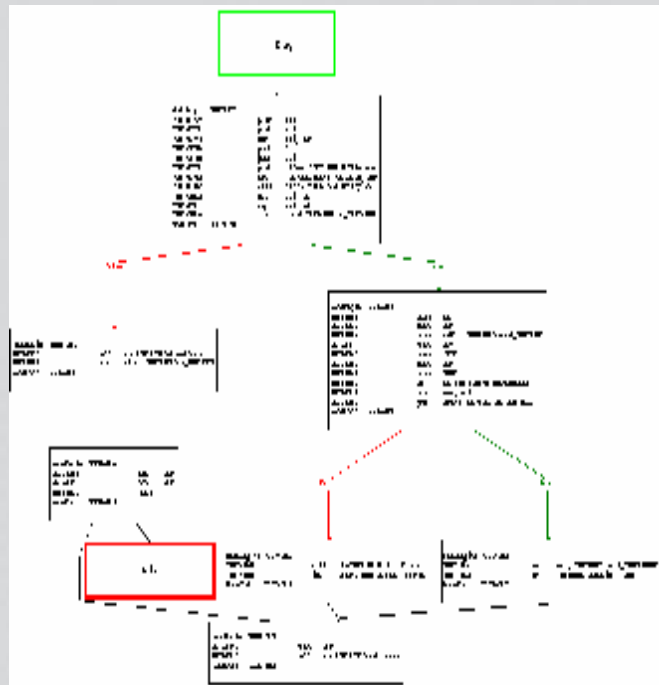
`sig3 = 3::4::5`



`Sig = 3::3::5`



增加虚拟的Entry和Exit节点



Sig = 3::4::5

- 优点：
- 1、可以统一优化处理过程。
 - 2、在loop的检测的时候有用。



X'con 2005

Demo



一些问题

- ❖ 完全调用图的问题
- ❖ 结构化比较的问题
- ❖ 指令比较的问题



完全调用图的问题

- ◆ 当我们遇到
 - ◆ `call esi`
 - ◆ `call [ebx+0x4]`
- ◆ 有些情况是可以解决
 - ◆ 最后的定义在同一个函数内
 - ◆ **Vtbl**函数和动态绑定
 - ◆ 函数指针模拟表
- ◆ 有些时候很难
 - ◆ 定义来自于调用另外一个模块的函数
 - ◆ 定义来自于全局未初始化变量
- ◆ 在循环中的**call**指令，每次获得不同值



最后的定义在同一个函数内

```
7C80B7FC ; BOOL __stdcall UnmapViewOfFile(LPCVOID lpBaseAddress)
7C80B7FC         public UnmapViewOfFile
7C80B7FC UnmapViewOfFile proc near           ; CODE XREF: su
7C80B7FC                                         ; BaseInitAppco
7C80B7FC
7C80B7FC lpBaseAddress = dword ptr 4
7C80B7FC arg_4         = dword ptr 8
7C80B7FC
7C80B7FC         mov     edi, edi
7C80B7FE         push  ebp
7C80B7FF         mov     ebp, esp
7C80B801         push  esi
7C80B802         mov     esi, ds:NtUnmapViewOfSection
7C80B808         push  edi
7C80B809         push  [ebp+arg_4]
7C80B80C         push  0FFFFFFFFh
7C80B80E         call  esi ; NtUnmapViewOfSection
7C80B810         mov     edi, eax
```

通常，我们使用函数指针或者动态加载dll的函数...



Vtbl 函数和动态绑定

```

0040100F      mov     dword ptr [eax]. offset base_vtbl
00401015      mov     esi, eax
00401017      jmp     short loc_40101B
00401019      ; 明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明明
00401019      loc_401019:                               ; CODE XREF: _main+
00401019      xor     esi, esi
0040101B      loc_40101B:                               ; CODE XREF: _main+
0040101B      mov     eax, [esi]
0040101D      mov     ecx, esi
0040101F      call   dword ptr [eax]
00401021      mov     edx, [esi]
00401023      mov     ecx, esi
00401025      call   dword ptr [edx+4]

```

```

004050B0 base_vtbl      dd offset demo_1
004050B4      dd offset demo_2
004050B8 unk_4050B8      db 0FFh

```



函数指针模拟表

```
00401020 Demo          proc near          ;  
00401020  
00401020 arg_0             = dword ptr 8  
00401020  
00401020 push    esi  
00401021 mov     esi, [esp+arg_0]  
00401025 call   dword ptr [esi]  
00401027 call   dword ptr [esi+4]  
00401028  
---
```

```
00401030 _main             proc near          ; (  
00401030 push   offset simul vtble  
00401035 call   Demo  
0040103A pop    ecx
```

```
00407030 simul_vtble      dd offset Demo_1  
00407034           dd offset Demo_2  
00407038 aDemo1         db 'Demo 1',0Ah,0  
00407040 aDemo2         db 'Demo 2',0Ah,0
```



定义来自于调用另外一个模块的函数

```
7C862CC1                                     ; UnhandledExceptionFilter+9B7j
7C862CC1      push      Target
7C862CC7      call     RtlDecodePointer
7C862CCC      cmp     eax, edi
7C862CCE      jz     short loc_7C862CE5
7C862CD0      push   ebx
7C862CD1      call   eax
7C862CD3      cmp   eax, 1
7C862CD6      jz    loc_7C863458
```

Eax由另外一个模块的函数的返回值定义。



定义来自全局未初始化变量

```
77E9BDB1 loc_77E9BDB1: ; CODE X
77E9BDB1 mov     eax, dword 77EC144C
77E9BDB6 cmp     eax, ebx
77E9BDB8 jz      short loc_77E9BDC7
77E9BDBA push   esi
77E9BDBB call   eax
77E9BDBD cmd   eax, 1
```

SetUnhandledExceptionFilter UnhandledExceptionFilter

dword_77EC144C: dd 0

```
77E6BC57 SetUnhandledExceptionFilter proc near
77E6BC57
77E6BC57 lpTopLevelExceptionFilter= dword ptr 4
77E6BC57
77E6BC57 mov     ecx, [esp+lpTopLevelExceptionFilter]
77E6BC5B mov     eax, dword_77EC144C
77E6BC60 mov     dword 77EC144C, ecx
77E6BC66 retn   4
```



在循环中的Call指令每次获得不同值

```
_main:  
sub    esp, 0Ch  
push  esi  
push  edi  
mov   [esp+14h+var_C], offset loc_401000  
mov   [esp+14h+var_8], offset loc_401010  
mov   [esp+14h+var_4], offset loc_401030  
lea  esi, [esp+14h+var_C]  
mov  edi, 8
```

```
loc_401066:  
call  dword ptr [esi]  
add  esi, 4  
dec  edi  
jnz  short loc_401066
```

true false

```
0040106E:  
pop  edi  
pop  esi  
add  esp, 0Ch  
retn
```

循环三次

首先我们需要追踪edi的D-U chain

每次，call都是不同的调用

为了方便固定点传播，我们可以在调用图上增加三个函数作为其孩子。



结构化比较的问题

- ❖ 错误匹配
- ❖ 深度优化
- ❖ 安全函数对非安全函数的替换
- ❖ 常量变化
- ❖ 内联函数
- ❖ 编译器的不同版本



错误匹配

- ❖ 我们无法证明错误匹配不会发生
- ❖ 多种因素造成



安全函数替换非安全函数

MS04-011 LSASS BOF

unpatch

```
785A0B58      push    [ebp+arg_8]
785A0B5B      mov     eax, 7FFh
785A0B60      sub     eax, esi
785A0B62      push    [ebp+arg_4]
785A0B65      push    eax
785A0B66      lea    eax, [ebp+esi+Buffer]
785A0B6D      push    eax
785A0B6E      call   ds:vsprintf
785A0B74      add     esp, 10h
```

patched

```
7859EE5B      push    [ebp+arg_8]
7859EE5E      lea    eax, [ebp+esi+Buffer]
7859EE65      push    [ebp+arg_4]
7859EE68      push    eax
7859EE69      call   ds:vsprintf
7859EE6F      add     esp, 0Ch
```



内联函数

会改变调用者的子调用数。

1、Inline前缀

```
__inline int max(int a, int b)
{
    if (a > b) return a;
    return b;
}
```

2、成员函数的实现在类定义中



指令级同构的问题

- ◆ 指令重排序
- ◆ 寄存器重分配
- ◆ 数据流分析



 X'con 2005

The End



 XFOCUS TEAM

BEIJING.CHINA

2002-2005