The Il Nop

Anciety / 2018-09-26 23:57:09 / 浏览数 2365 技术文章 技术文章 顶(0) 踩(0)

原文链接

http://reversing.io/posts/the-il-nop/

The Il Nop 随便说点IL

什么是IL/IR?

IL是中间语言的缩写，IR则是中间表示的缩写。中间表示是我们在计算机程序从一个状态转移到另外一个状态时的存储方式，比如从源代码到编译后的二进制文件，中间语言

休斯顿总部！我们这出现了问题！

我对Falcon IL有个问题，是时候来做点改变了。

Falcon将二进制代码提升到一种叫做Falcon
IL的中间语言，这时一种比较简单的中间语言，不过倒是囊括了指令语义和控制流。我用Falcon来实现可以在x86和MIPS二进制上用的静态分析和符号执行工具。

在我打算给Falcon IL做点改变之前，我决定写一篇文章讲讲Falcon
IL的历史，我会讲讲我做的一些决定，哪些事情做起来效果比较好，哪些不太好，以及有哪些已经改变了。

在最开始的时候

Falcon
IL并不是我第一个实现的IL，我第一个IL几乎完全就是RREIL的复制品，主要包括了独立的指令语义，但是不包括控制流。Queso是另外一个二进制分析框架，不过现在放弃
Toolkit是一个实验性质的动态二进制插桩引擎，也实现了它自己的IL，叫做Bins，也就是Binary toolkit INStruction.

我其实还是比较喜欢Angr的pyvex里用的VEX IR的，以及一直很流行的LLVM IR，虽然LLVM
IR更多是为了表示源码级的程序而不是二进制级的，McSema和s2e找了一些方法来把LLVM IR用到二进制程序上。我用过BAP
IL，现在通过我在ForAllSecure的工作，以及我使用BAP IL的经验，我终于有了对Falcon的一个初步的设计决策。以及和以前一样，我也发现binary
ninja的LLIL和MLIL对于想快速开始工作还是很好的（我自己也几周前写了一个MLIL的工具）。

但是等到我自己来实现Falcon的IL的时候，我发现我还是需要仔细思考一下需要做什么，以及我到底需要什么。

通过指令数来看IL

IL可以通过他们实现了多少种类的指令来画个图,那么RREIL可能是最简单的一种了，它用了一种three form形式的IL，也就是说（几乎）所有的指令都是`OP, DST, LHS, RHS`的形式。比如一个RREIL的加法指令就是`ADD RAX, RAX, 7`。在RREIL里是没有表达式的，只有指令，它的IL本身也被设计为使得指令的数量越少越好。

与其相对的则是VEX IR。Vex ir并不是针对程序分析所设计，Vex ir被设计为使得valgrind可以快速运行。VEX
IR有上百条指令种类，允许valgrind去针对插桩代码用最优的指令，以求指令执行速度更快。VEX
IR的吸引力来源于它的完全性，以及价格（GPL），虽然要将其调整为便于程序分析的目的可能会比较麻烦，毕竟有上百条指令种类。

在中间的就有BAP IL，LLVM IR和Binary ninja的IL，这些指令都支持表达式，通过表达式我们就可以有`a = b * (c + 7)`，而没有表达式我们就需要: `temp0 = c + 7; a = b + temp0`，看起来就比较恶心了。在我看来，表达式用起来还是比较简单的，和three
form的IL一样简单，表达式的变形和简化可以一次性写完，并且到处使用。

所有的这些IL，从RREIL到Binary Ninja的IL，都被设计为保存同样的信息，他们只是表达信息的方式不同，比如一种IL可能会用`ADD DST, LHS, RHS`而另外一种用`DST = (LHS + RHS)`，但是他们的语义还是保持一样的。

所以在一种IL里哪种算是一条指令，以及哪个IL具有最多的指令数呢？我相信一个IL里的一条指令告诉我们数据会怎样被处理和移动，不过我也知道这并不是一个完全的定义

以赋值为例，比如`DST = LHS OP RHS`，其中`OP ∈ {ADD, SUB, MUL}`，其实与`ADD DST, LHS, RHS, SUB DST, LHS, RHS, and MUL DST, LHS, RHS`一样，他们总归都是赋值，但是一种形式允许我们编码很多种不同的操作类型，而另外一种就需要根据操作来选择不同的赋值指令类型，相对来说我更喜欢前面一种。

对Falcon来说，我想设计一种简单的，基于表达式的IL，我最开始设计出了5种操作类型：

- Assign { dst: Scalar, src: Expression }
- Store { index: Expression, src: Expression }
- Load { dst: Scalar, src: Expression }
- Branch { target: Expression }
- Raise { expr: Expression }

`Expression`一开始包括这些值:

- Scalar(scalar)
- Constant(constant)
- Add(lhs, rhs)
- Sub(lhs, rhs)
- Mul(lhs, rhs)
- Divu(lhs, rhs)
- Modu(lhs, rhs)
- Divs(lhs, rhs)
- Mods(lhs, rhs)
- And(lhs, rhs)
- Or(lhs, rhs)
- Shl(lhs, rhs)
- Shr(lhs, rhs)
- Cmpeq(lhs, rhs)
- Cmpneq(lhs, rhs)
- Cmplts(lhs, rhs)
- Cmpltu(lhs, rhs)
- Trun(bits, rhs)
- Sext(bits, rhs)
- Zext(bits, rhs)

scalar的类型是Scalar，constant的类型是Constant，lhs的类型是Expression，rhs的类型是Expression，bits的类型是usize。

每一个`Instruction`都保存了一个`Operation`，以及一些其他的信息，比如这个`Operation`在解码前的地址，可选的注释，一个在一个`Block`中的位置。`Block`和`Edge`组

这里是Falcon原始IL的引用

通过可读性看IL

Binary
Ninja是第一个我知道的可读的IL，如果你花点时间在逆向工程里看IL，这种感觉就很像你朋友的侄子上Java导论课上的很费劲，然后他很确定他写的找一个数组里最大值的

RREIL是不可读的，VEX IR也不可读，在我有了一些Binary Ninja IL的经验之后，我想要一个可读的IL，这是我现在Falcon IL的一些示例：

```
[ Block: 0x0 ]
804849B 00 exc:32 = (esp:32 + 0x4:32)
804849F 01 temp_0.0:32 = (esp:32 & 0xFFFFFFF0:32)
804849F 02 ZF:1 = (temp_0.0:32 == 0.0:32)
804849F 03 SF:1 = trun.1((temp_0.0:32 >> 0x1F:32))
804849F 04 CF:1 = 0x0:1
804849F 05 OF:1 = 0x0:1
804849F 06 esp:32 = temp_0.0:32
80484A2 07 temp_0.0:32 = [(ecx - 0x4:32)]
80484A2 08 esp:32 = (esp:32 - 0x4)
80484A2 09 [esp:32] = temp_0.0:32
80484A5 0A esp:32 = (esp:32 - 0x4:32)
80484A5 0B [esp:32] = ebp:32
```

不算太好，但是我还见过更差的，首先虽然这些对于不太熟悉Falcon IL的人来说看起来都很像是赋值，但是其实在里边是有`Assign`, `Store`,
和`Load`的操作类型的。以及由于表达式的存在，我们在这里可以把像`temp_0.0:32 = [(ecx - 0x4:32)]`或者`SF:1 = trun.1((temp_0.0:32 >> 0x1F:32))`编码到一个操作里，这样我觉得是会提高可读性的。

Falcon IL没有为了可读性变得像Binary Ninja的IL那么长，但是还是做了一些工作让他变得不那么不可读。

编码跳转

跳转指令可以分为几种类型，其中比较典型的是直接和间接，一个直接跳转是指目标地址直接被编码在指令里的，比如"跳到地址0x12345"，或者"给当前的指令指针加上-2

我们还有"call"和"非call"跳转指令，可能更好描述他们的用词应该是"过程间跳转"和"过程内跳转"。在大多数架构上这个信息都直接在指令助记符里，在x86里我们有`call`(
`jalr`和其他`lr`，在有的架构里确实会更奇怪一点，但是这里我们只讨论Falcon支持的架构。

最后跳转操作可以是带条件的，条件跳转就是一些像x86汇编里`je`一样的指令，当一些条件被满足，例如zero标志位为1的时候就进行跳转。

Falcon把过程间跳转，或者说"Call"跳转，以及间接跳转都编码为跳转指令。Falcon把过程内直接跳转，包括带条件的过程内直接跳转隐式的编码为了在控制流图里可选的受
IL里是没有指令的，而是会建立一条边到跳转目标处。

Falcon IL将过程内间接跳转（大多用于跳转表）简单编码为间接跳转。

一开始看起来很美好，之后有了一些改动

一开始Falcon工作的很好。系统调用指令，以及一些其他的有趣的指令都被实现为`Raise`操作，通过这个操作Falcon IL就可以去符号执行CGC里的一些回文问题，之后我还写了一些静态检测MIPS二进制程序里libc函数传参的一些常见错误。等到我需要lift浮点数指令的时候，我就简单的翻译

Raise变成了Intrinsic

最终，我在符号执行MIPS指令的时候，我碰到了`rdhwr`指令，这条指令在MIPS ABI里有特殊的处理方法，当指令的`rd`操作数，也就是源操作数是栈指针的时候，TLS的地址被放进了`rt`操作数，也就是目标操作数，这个情况不是我的`Raise`操作可以处理

最后我方法是把`Raise`操作换成了更长的，但是也更充分准备的Intrinsic操作，这个操作有他自己的操作数，还提供了甚至在操作语义没有完全被记录的时候的分析功能。可

现在我完全把所有`Raise`出现的地方都换成了intrinsic，然后一切工作正常，我很高兴有这样的一个切换。

If-Then-Else，或者说ite表达式

状态合并是个问题，这是一篇文章，这是另外一篇。给出两个具有路径约束的值，例如`A = 1 iff B = 2`和`A = 2 iff B != 2`，我们需要把值合成`A = 1 iff B = 2 else A = 2`，然后合并状态，在其他东西都没问题的时候的话，但是那就不是这篇文章需要讨论的问题了。看起来相当直接，当然，除非你不知道如何在你的IL里表示if-then-else。

我一开始有意识的在Falcon IL里剩下了`Ite`表达式，汇编指令里的条件赋值被翻译为了控制流图，条件则被转换为了图里的条件边，赋值指令只在条件保护的边执行为真的时候才会碰到，这样的做法一

经过了一段时间，我心软了，现在Falcon有`Ite`表达式了，这个表达式永远不会被lift出来，但是在之后的分析中可以被加上，比如符号执行里的状态合并（或者其他的情况

MIPS，你这个有delay-slot的小恶魔

为了理解这个程序，你需要了解一点关于Falcon IL和MIPS的知识

MIPS跳转有一个指令delay-slot，对于一个跳转指令，比如`beq v0, v1, label`，MIPS将会：

1. 计算是否跳转条件为真，以及是否会进行跳转
2. 执行跳转后的指令（以及delay slot）
3. 进行跳转，如果需要跳转的话

- Falcon IL的指令，在lift之后，包含lift之前的地址，主要是被翻译器用来计算CFG的边，以及分析找到间接跳转之后的目标

如果在你的IL里没有显式的delay slot的符号（关于这个问题我其实应该再写一篇文章），delay slot就会成为一个完美的大灾难。

我们从指令一开始是怎么实现的开始

假设我们有以下的代码:

```
1000: balr 0x123456:32
1004: addiu $v0, $zero, 1
```

非常简单，Falcon 以块为单位lift指令，而不是独立的指令，在我们碰到一个带有delay slot的跳转指令的时候，我们可以简单的把跳转指令lift了，再lift delay slot，之后把delay slot放在跳转指令之前，块结束的地方，非常简单，所以我们来看看大概长什么样：

```
1004 00 $v0:32 = (0x0:32 + 0x1:32)
1000 01 b 0x123456:32
```

牛逼极了，任务完成，只是，如果在分析过程中，有另外一个跳转指向0x1000，那么会发生什么问题？好，我们看下，我们就会找到0x1000，然后开始执行，然后忽略了d slot。这个问题会导致`ld.so`出现失败，然后让人花好多小时，甚至好多天，我自己都没数来debug么？是的。。所以应该怎么fix呢？

首先我们可以在1004指令前插入指令，这个指令什么也不干，但是分配给他地址1000。我们还是需要跳转指令的地址，因为这是现在lifter的工作方式，现在，Falcon大概会

```
1000 00 nop = 0x0:1
1004 01 $v0:32 = (0x0:32 + 0x1:32)
1005 02 b 0x123456:32
```

哇塞看啊，这样就好了，只是看起来不太漂亮是吧？有时候确实是。。

记住，条件跳转会在delay slot执行之前计算跳转条件，比如这样一个指令序列：

```
1000 beq $v0, $zero, label
1004 addiu $v0, $zero, 1
```

这就会变成：

```
1000 00 nop = 0x0:1
1004 01 $v0 = (0x0:32 + 0x1:32)
```

```
// outgoing condition on edges is ($v0 == $0x0:32)
```

我们就弄坏了这个条件跳转，我最后采用的解决方案是把nop换成一个到变量`branch_condition:1`的赋值，这样就成了：

```
1000 00 branching_condition:1 = ($v0:32 == 0x0:32)
1004 01 $v0 = (0x0:32 + 0x1:32)
// outgoing condition on edges is branching_condition:1
```

这样就好了，我觉得这个方法对于所有单指令delay slot的架构都适用。

缺少跳转指令的情况

我们之前提到了Falcon
IL是怎么完全忽略掉过程内直接跳转的，这些跳转会被隐式转换成CFG内的边。这就有了一个问题：有时候一个分析需要知道跳转指令的地址。最显然的例子就是一个跳转指

这个问题在MIPS里用上面的丑陋的hack方法解决了，但是在x86里还是存在。我可以继续往IL里添加没用的赋值，就为了解决在某个指令应该在的地址的占位问题，但是这样
code elimination（不活跃代码消除）的pass可能会消除一些指令，使得刚才的那些问题在之后又出现了。

我觉得是时候搞一个完备的nop操作了，因为我知道Nop操作没有有意义的语义，以及我在分析的时候可以跳过，我甚至可以在观察IL的时候忽略（保证了一定程度上的IL可读
code elimination的时候，或者其他的IL变形的时候，我可以不管nop，或者在移除指令后插入nop，保证未来需要依赖到地址的分析工作正常。

Falcon，以及继续Falcon IL

我不觉得有这么一种IL可以使得所有的分析都非常适合，一个智者曾经告诉我"你可以创建一个IL来适合这个分析，但是另外一种就不行了"。我们从一个提供了不错的基础的
IL依然是我最喜欢的IL，然后我会经常来保证他的完整性。

上一篇：提高AFL qemu模式性能 下一篇：利用ssh绕过macOS Maja...

1. 0 条回复
   - 动动手指，沙发就是你的了！