Modular Control-Flow Integrity

MAR 29TH, 2016

论文下载

摘要

过去CFI不支持单独编译，影响了CFI技术的推广。
MCFI支持模块独立插桩、静态和动态链接。组合多个模块，用每个小模块的CFG生成一个新的CFG。一个难点是在多线程的动态链接中如何保证从旧的CFG安全地转换到新CFG去。文中使用一个动态的数据结构来表示CFG，并封装相应的对数据结构进行读写的事务来保证线程安全。

引言

分离编译，编译器能独立地编译应用程序的每个模块，并连接这些模块，添加CFI规则时，无需考虑其他模块，独立地插桩，并将插桩好的模块连接到一个工作可执行程序中。
MCFI中，一个app被分成了多个模块，每个模块都有代码，数据以及帮助与其他模块进行链接和生成模块CFG的辅助信息。当一个模块与其他模块连接后，应该生成一个新的CFI规则，新规则允许原来的ICF能够跳到更多的target上去。故在连接时CFI规则发生了变化。
MCFI在链接lib时，两个挑战是：

1 在多线程中如何在运行时安全有效地更新规则
    - MCFI在代码段之后，用一个独立地表来表示CFG。为保证表的平稳更新，MCFI设计了table-check和

table-update 事务，使用了一个轻量级的STM（software Transactional Memory）算法

2 在模块进行组合是，如何有效地生成精度高的规则
    - 使用辅助类型信息来生成cFG

贡献：

1 MCFI是第一个支持单独编译的有效CFI插桩方法，很大改善了CFI实践性
2 基于类型匹配的C程序CFG生成的方法，高效且可在动态链接中使用，C程序无需或只需做小的修改即可与CFG生成过程兼容
3 实现了一个编译工具链，可对X86的C程序进行插桩。

概述

MCFI是一个细粒度的CFI。

威胁模型

1 并发程序中攻击者有个独立地线程，可任意读写内存，假设一个线程的寄存器不可以直接被攻击者的线程修改，但可通过修改内存的方式来间接修改寄存器。
2 不允许任意代码执行，开启NX，code可读可执行不可写，data可读可写不可执行

ID Tables

ECN(Equivalences-Class Number)从代码段剥离，存在一个由两个独立的表组成的动态数据结构中。这两个表都是一个从地址到ID的映射。

- branch ID Table — Bary Table
    - 分支指令地址到其对应ID的映射
- target ID Table — Tary Table
    - target地址到其对应ID的映射

将ID从代码中分离出来有一下好处：

1. 代码段对应表中的ID可重复(IDs in the tables can overlap with the numbers in the code section)，消除了传统CFI的ID全局唯一性假设
2. 间接分支跳转前插桩代码可根据ID表来参数化，一旦载入就不再改动，因此程序的代码页面和库都在在进程间共享，节省了内存和应用程序的启动时间
3. 中心化的ID表为并行化的CPU内存复制机制带来了良好的内存cache访问和快表更新。

Table access transactions

ID表支持多线程并发访问。链接新模块时需要更新生成新CFG，这时若有其他线程需要读表，则需要相应的同步机制。简单的方式是锁机制，但这带来的开销比较大。
文中使用STM的通用模式，将表操作封装成事务，来保证安全性和效率。

1. Check transaction (TxCheck)
    - 在执行间接跳转前执行该事务，读取branchID和TargetID并比较，这个事务只需要读表
2. Update transaction (TxUpdate)
    - 在动态链接过程中执行该事务，在连接一个新的Lib进来后，从新CFG中生成新ID，该事务更新 bary和tary两个表。

这个更高效的原因是 check transac 执行推测性的读表操作，假设没有其他线程同时在执行并发写操作；若假设不成立，则停止并retry。这个技术符合上下文，并且足够高效。

Module Linking

一个MCFI模块不仅包含code和data,还有辅助信息。一个模块包含的辅助信息越多，生成的CFG精度就越高，MCFI接收类

型信息，能够表示函数类型和函数指针类型。类型信息用来生成模块的CFG，一个间接调用能够通过函数指针来调用一个函数，只要这个函数的类型与函数指针的类型匹配。这能够生成相当精确的CFG，但需要有源码的配合。

ID Tables and Transaction

支持X86-32和x86-64

ID Tables

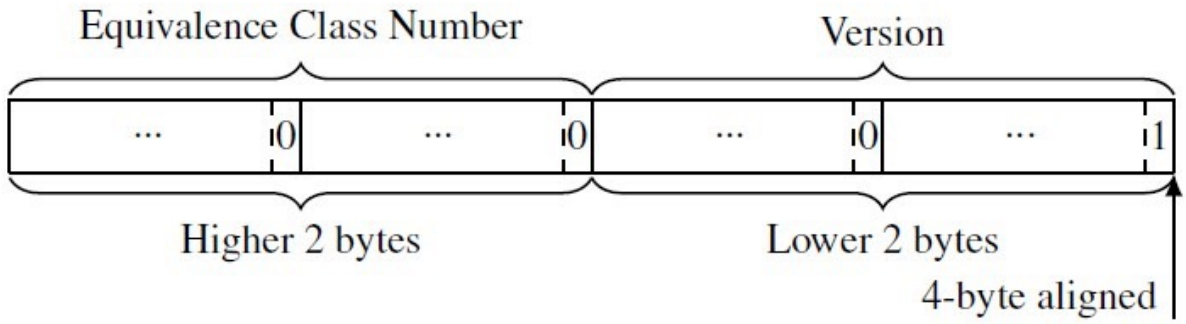

Figure 2. MCFI's ID Encoding.

ID 表示

一个ID 4Byte，每个Byte的最低位是确定的，从高到底依次为 0,0,0,1。这些保留比特能够防止查找时指向ID中间。
高的2个BYTES表示ECN，一共可表示2^{14}个等价类，这对大程序足够了。
低2个BYTES表示事务中的版本号，用来检测是否一个check transaction需要被中止和retry.同样也允许 2^{14}个不同的版本号。

两个表

文中使用数组来存储这两个表，用代码地址作为数组下标。
tary表中，不可能的target addr在数组中对应的值为0，否则则是对应的ID，这样在查找和更新时都非常高效。

- tary表的大小是代码大小的4倍（每个id 4字节）。

- 优化：插入nop指令来使间接跳转的target地址是4字节对齐的，因此表中只需要4字节对齐的代码地址入口。这样表大小就与代码大小一样大了。

bary表也可优化：因为bary表将间接跳转地址对应bID,而指令地址在载入到内存时就知道了，在对应的bID读取的指令处直接插入对应的bary表下标，这样就不需要没有间接跳转指令的代码地址了。只要装载器在bID读取指令处嵌入了正确的表下标，那么所有bary表中的BID都是合法的ID。

表的保护

在程序运行时需要对表进行保护，这样应用程序就不能直接修改表内容。文中使用MIP来限制代码可写的内存区域，故不能直接修改表。

- X86-32：保留1GB用来运行程序代码，1GB用来保存表区域
- X86-64：4GB用来存储表内容，对内存写操作进行了插桩，使其范围限制在[0,4GB)之间。
- 为有效访问表，用了一个额外的段寄存器存储表内存区域的基址： %fs对应x86-32, %gs对应x86-64.

Table Transactions

Update Transaction

```
1  void TxUpdate () {
2    acquire(updLock);
3    globalVersion = globalVersion + 1;
4    updTaryTable();
5    sfence;
6    updBaryTable();
7    release(updLock);
8  }
9  void updTaryTable() {
10   // allocate a table and init to zero
11   allocateAndInit(newTbl);
12   for (addr=CodeBase;addr<CodeLimit;addr+=4) {
13     ecn=getTaryECN(addr);
14     if (ecn >= 0) {
15       entry=(addr - CodeBase) / 4;
16       newTbl[entry]=0x1; // init reserved bits
17       setECNAndVer(newTbl, entry,
18                    ecn, globalVersion);
19     }
20   }
21   copyTaryTable(newTbl, TaryTableBase);
22   free(newTbl);
23 }
```

Figure 3. Pseudo code for implementing update transactions.

- 在MCFI动态链接器中使用。
- 考虑到时间中，很少进行更新，故使用updLock串行化该更新操作，但这个全局更新锁不会阻止 upd和chk这两个事务的并发性。

更新过程分为两步：

1 更新tary 表

2 在tary表更新的内容全部写入到内存后，再开始更新bary表

- 两个表的更新不能交错，这样在更新事务的一些中间状态，可能表中一些ID是新版本，一些ID是旧版本，这样check时可能对不同的ICF会使用不同的

CFG。如果先更新完一个再更新另一个，check 中要么用old CFG,要么用新CFG。

代码中第21行是critical 。

更新表可并行化，只需要每个ID更新是原子的，可借助movnti指令，直接将数据写入内存，无需通过cache，更快地并行拷贝。

Check Transaction

```
1  TxCheck {
2    popq  %rcx
3    movl  %ecx, %ecx
4  Try:
5    movl  %gs:ConstBaryIndex, %edi
6    movl  %gs:(%rcx), %esi
7    cmpl  %edi, %esi
8    jne   Check
9    jmpq  *%rcx
10 Check:
11   testb $1, %sil
12   jz    Halt
13   cmpw  %di, %si
14   jne   Try
15 Halt:
16   hlt
17 }
```

Figure 4. Implementation of check transactions for x86-64 return instructions.

2,9表示原始的ret指令。 5读取bID,6读取tID。bID和tID的比较有以下四种情况：

1 esi和edi相等，执行 7,8,9，符合CFI，ret正常执行

2 target addr不是4字节对齐的，或者对应的tart ID是0，则tID不合法。执行 7,8,11,12,16

3 tID合法，但bID与tID的版本号不同，retry，check transaction等待update transaction完成。执行7,8,11,12,13,14

4 tID合法，ECN不同，版本号相同，违反了CFI规则，执行 7,8,11,12,13,14,16

可串行化

upd 串行化点在第5行，在此之前chk用旧CFG，在此之后，chk用新CFG。chk串行化点在第6行读取tID后。

The ABA Problem

攻击者可能载入2^{14}个模块，穷尽所有版本号。这在实践中是不太可能能，即使是对JIT编译的代码。安全性只有在程序至少执行了2^{14}次更新后才会被破坏。

MCFI可以维护一个已更新模块的计数器，保证不会到2^{14}。也可以为版本号提供一个更大的空间，比如在64位机器上用8字节ID。

Module Linking

MCFI将类型信息附加到模块上，并在快速CFG生成过程中使用类型匹配。一个有函数类型信息和函数指针类型信息的MCFI模块能够生成比粗粒度的CFI更精确的CFG。每个模块的类型信息可通过扩充编译工具链来生成。文中修改了LLVM，将类型信息从源码层传到了底层。

类型匹配的CFG生成

这里一个模块可能是多个小的模块连接而成的，并且假设当前模块是由C代码生成的。

直接跳转的边可直接计算出来，这里只考虑间接跳转：

- 通过函数指针的间接调用
    1 函数地址在代码中被使用

2. 函数类型结构等价于函数指针类型

- 间接跳转可分为两类：过程内跳转和过程间跳转
  1. 过程内跳转：通常是 switch, goto，target通常被硬编码到程序中，是一个只读的跳转表。
  2. 过程间跳转：间接尾调用(jmp func)，使用同样的类型匹配方法。一个过程间间接跳转能够跳到任何类型等价于间接跳转对应的函数指针的函数。
- 返回指令
  1. 构造一个函数调用图，标明函数通过直接或间接被调用。
  2. 尾调用处理：f中调用g，g通过尾调用调用h，则f中调用点到h也存在一条边。
  3. 通过调用图，能够生成控制流边：如果存在一条边从一个调用点到一个函数f，则函数f中的返回指令能够返回到调用点后的返回地址。
- 特殊情况：
  1. longjmp和setjmp：longjmp返回到每个setjmp的返回地址
  2. 变长参数：只允许返回地址类型和固定参数类型匹配的函数
  3. 信号处理函数：信号处理函数不会返回到应用程序代码，而是返回到一个小的代码片段，这个片段调用sigreturn系统调用。文中将这个小的代码片段内联到信号处理函数中，消除了ret
  4. 内联汇编：需要开发者为汇编中用到的函数和函数指针添加类型注解。

生成类型匹配的CFG的条件

首先假设输入的C程序满足两个条件：

- C1：不允许类型转换为函数指针，或者从函数指针进行类型转换
- C2：不允许汇编

C1不允许与函数指针相关的显式或隐式类型转换。

- 隐式类型转换：若union中包含一个函数指针的成员，或者一个结构体转换为另一个带有函数指针成员的结构体，但二者的函数指针类型是不兼容的

C2则需要向汇编代码中加入类型注释，这样才能用相同的类型匹配方法。

文中对SPEC 2006做了实验，检测其中的C程序是否满足这两个条件：违背C1很容易被检测，因为LLVM内部表示使得所有类型转换都是显示的，违背C2，分析器只报告出现了内联汇编（实验中没有发现违背C2，对于libc手动添加了类型注释）。

SPECCPU2006	SLOC	VBE	UC	DC	MF	SU	NF	VAE
perlbench	126,345	2878	510	957	234	633	318	226
bzip2	5,731	27	0	0	6	4	0	17
gcc	235,884	822	0	0	15	737	27	43
mcf	1,574	0	0	0	0	0	0	0
gobmk	157,649	0	0	0	0	0	0	0
hmmer	20,658	20	0	0	20	0	0	0
sjeng	10,544	0	0	0	0	0	0	0
libquantum	2,606	1	0	0	0	0	0	1
h264ref	36,098	8	0	0	8	0	0	0
milc	9,575	8	0	0	3	0	0	5
lbm	904	0	0	0	0	0	0	0
sphinx3	13,128	12	0	0	11	1	0	0

Table 1. C1 violations in SPECCPU2006 benchmarks.

上图中误报主要有以下5种模式：

1 Upcase(UC):向上类型转换，有时结构体之间的类型转换用到了如参数多态和继承这样的特性。一个抽象类型可

能有几个具体的子类型，他们之间有些成员是一样的。一个函数若接受了抽象类型的参数，则这个函数也是多态的。调用这个函数的caller可能会进行类型转换，这些类型转换是向上的，误报就是因为一个具体的struct中可能有一些类型转换后所没有的成员(extra field)

2. Safe downcast(DC)：从抽象类型向下转换为具体类型通常是不安全的，然而通常抽象结构中有个一个类型标记来标注运行时该结构的一个具体类型。如果一个type tag和具体的结构类型正确关联，那么这也可能是误报。这时就要手动修改
3. Malloc and free (MF): malloc通常返回void*，若malloc用来分配一个包含函数指针的结构体，这也可看做误报。free同理。
4. Safe Update(SU): 用常量来初始化函数指针也被视为误报，比如把函数指针初始化为NULL，则表示将整型转换为函数指针
5. Non-function-pointer access(NF):有一些类型转换涉及了函数指针，但转换结束后这个函数指针再也没有被用过。

在处理完这5种误报后，仍有5个存在违背condition的情况，进一步细分为以下两类：

- K1: 用一个类型与函数指针类型不匹配的函数地址来初始化函数指针
- K2: 一个函数指针被转换为另一个函数指针，随后又转换回来

	perlbench	bzip2	gcc	libquantum	milc
K1	4	0	36	1	0
K1-fixed	4	0	22	1	0
K2	222	17	7	0	5

Table 2. Numbers of cases for the two kinds of violations.

K1-fixed标书需要修改源码来使用类型匹配方法生成CFG的数量，K2的修复不需要修改源码

- 多数K1需要认为修改源码，因为不匹配的函数指针和函数可能导致生成的CFG精度下降。修改方法通常是给函数加个wrapper，功能等价于原函数，但签名与函数指针相同。所有K1-fixed都是用这种方法修改的。
- K2可能是由于向下类型转换时没有进行动态类型检查（这样在上轮第一次修复误报时没有修复成功），这种情况下，开发者认为这是安全的（可能通过代码审查），不需要动态检查。K2不需要修改源码来生成CFG，只需要提供额外的数据集即可（就是加入特例吧？）

静态和动态链接

MCFI静态链接器用组合模块阶段的辅助信息改变了标准静态链接器，也改变了标准链接器的PLT入口模板，用MCFI插桩的PLT入口代替原来的不安全版本。

MCFI也允许多线程程序动态装载新的lib并将控制流转移到Lib 代码去。动态链接器自身也被MCFI插桩，在一个沙盒中运行，就像其他程序模块一样。在程序模块装入前，动态链接器首先被装载到内存中，程序模块的GOT入口被设置为动态链接器的入口点。

动态链接一个lib可分为以下几步：

1 Module preparation:

- 一个运行的程序通过跳到PLT或用dlopen调用MCFI的动态linker来载入新的lib，动态链接器将lib载入沙盒中，并将lib code设置为可写不可执行，然后链接器分析lib并生成新的PLT target地址

2 新CFG生成:

- 链接器调用CFG生成器生成新CFG，PLT入口通过名字匹配与对应函数相关联，为bary和tary生成新的ID；为沙盒中的lib代码嵌入读取bID的指令
- 代码页设置为只读，静态验证是否遵守CFI规则
- 代码设置为可执行不可写

3 ID表更新:

- 链接器将新的PLT target地址传送出去，执行一次update transaction，调整表中的ID，修改GOT的入口，使用新的PLT target address

MCFI’s Toolchain

工具链包括:

- rewriter: 执行程序插桩
- static linker：组合模块，PLT插桩
- CFG generator：收集辅助模块信息，构建CFG
- verifier：验证MCFI模块的插桩是否遵循CFG
- runtime system：载入和执行插桩的程序
- dynamic linker：运行时调用，动态载入lib