摘要

文中提出了一种新的防御思想：CPI，保证所有代码指针（函数指针，保存的返回地址）的完整性。同样可以抵挡ROP等控制流劫持的攻击，并且针对于年初的几篇专门针对CFI的攻击，也能成功防御住。同时文中还提出了一种相对条件更宽松的CPS，更加高效，不过是通过放松了条件限制实现的，能达到类似的防御效果。CPI和CPS防御比CFI更加高效，可实践性更强，作者已经把原型放到网上，并集成到LLVM中作为一个编译选项，还计划将这个防御提交到llvm的upstream上去。

CPI是精确的，对于程序中所有的代码指针都能确保内存的安全性。核心思想是将进程内存划分为安全区域和常规区域。通过静态分析的方法找出程序中所有需要保护的内存对象（所以这是需要在有源码的基础上，在编译时对指针访问进行插桩，尽管不需要对源码进行修改），借此来达成保证所有代码指针的内存安全的目的。

- 安全区：只有编译时证明安全或者运行时经过动态安全监测的内存操作，才能对安全区域进行访问。
- 常规区：无需任何运行时动态检查，只需像未添加保护时那样访问内存即可，故不会带来额外开销。

攻击模型

只关心控制流劫持的攻击，修改或泄露未受保护的数据等仅针对数据的攻击不在本文的考虑范围内。

假设现实攻击者已经有足够强大的能力：

- 控制了进程所有内存，但不能修改代码段。
- 攻击者可通过控制输入来对任意地址进行读写。
- 攻击者无法控制程序加载进程（program loader process)

这些假设保证了编译时进行插桩的完整性，保证程序的加载进程能够安全地将安全区和常规区独立开来。

设计

CPI Property相关定义

- 内存对象(memory object)：关于内存分配的，语言相关的内存单元：如全局或局部变量，动态分配的内存块，或者一个大的内存对象的子对象（结构体的成员）。同时也是与特定程序相关的，比如程序定制了自身的内存分配方法。
- 控制流目标（Control flow destination）：即位于代码段中的位置，如函数头或返回地址。
- 目标对象（target object）：内存对象或控制流目标。
- 指针解引用(pointer dereference)：通过指针来访问内存目标
    - 数据指针的读写
    - 将控制流转移到代码指针所指向的位置上去
- 一个指针是基于（based on）一个目标对象X的⇔为保证一个指针最多基于一个对象，这个指针在运行时通过以下方式获得：
    - 在堆上为X分配一块内存
    - 如果X是静态分配的，显示声明让指针获得X的地址(比如通过C/C++的&)，比如一个静态或全局变量，一个控制流目标（包括在调用一个函数时隐式放置在栈上的返回地址）
    - 获取X的一个子对象y（如结构体的成员）
    - 计算一个指针表达式（指针运算，数组下标引用，简单地复制一个指针），操作数可以不全是指针，也可以都是基于对象X的指针。

- 对一个指针的解引用是安全的⇔对于这个指针解引用所访问的内存落在目标对象的范围内（这里的目标对象就是CPI要保护的目标）。
- 程序的一次执行是内存安全的⇔执行过程对所有指针的解引用都是安全的
- 一个程序是内存安全(memory-safe)的⇔对所有输入，程序的所有可能执行过程都是内存安全的。
- 敏感指针: Sensitive Pointer are code pointers and pointers that may later be used to access sensitive pointers.
    - 对于一个指针是否敏感的定义是动态的，一个void*指针上一秒指向一个整数时还不是敏感的，但可能下一秒将其指向另一个敏感指针后，这个指针也变成了敏感指针。
    - 只有在运行时才能精确地确定一个指针是否是敏感的
- 一个程序的执行过程满足CPI属性⇔无论是解引用还是访问敏感指针都是安全的。
    - 由于CPI是在编译时通过静态分析来进行的，故通过CPI得到的应该是所有敏感指针的一个近似解（超集）

方法

CPI静态分析

基于类型的静态分析

如果一个指针的类型是敏感的，这个指针就是敏感的。

敏感类型包括:

1. 指向函数的指针
2. 指向敏感类型的指针

3 指向有一或多个成员是敏感类型的复合类型（结构体或数组）
4 通用指针(void, *char*, 在定义struct或class前就声明的指针)
5 也可以自定义其他的敏感类型
6 所有在编译或运行是隐式生成的代码指针（返回地址，C++虚函数表，setjmp缓存）

识别相关指针

通过静态分析从程序中找出所有处理这些指针的指令：

1 指针解引用
2 指针运算
3 内存分配或释放
    - 相关标准库函数
    - C++的new/delete操作符
    - 自实现的分配算法

C/C++中char* 这样的通用指针被识别为敏感指针，可能永远不会指向敏感数据，民启发式地减少敏感指针：

1 假设传给libc标准库字符串处理函数的指针不是敏感指针
2 指向一个常量字符串的指针不是敏感指针

CPI插桩

目标

1 保证所有敏感指针存储在安全区
2 在运行时能够创建和传递这类指针的相关元数据
3 在对这类指针进行解引用时，检查元数据

细节

CPI在安全区和常规区为敏感指针都分配了存储空间，但二者同时只能有一个是有效的。（通用指针在运行时的敏感性可能发生变化，同时也可以避免因为改变了内存空间分布而产

生的一些兼容性问题）。CPI也将指针在常规区的地址来计算指针在安全区的相应地址。

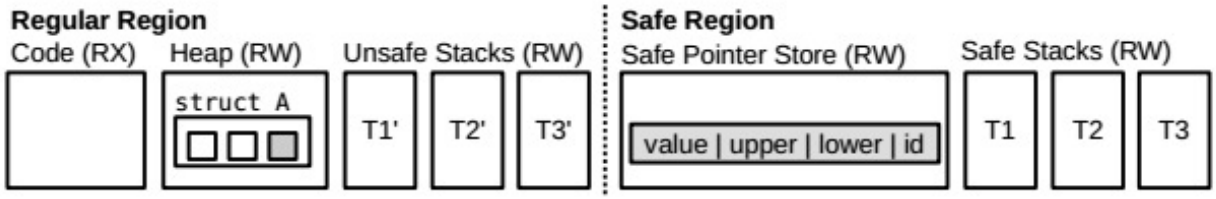

Figure 2: CPI memory layout: The safe region contains the safe pointer store and the safe stacks. The location of a sensitive pointer on the left (shaded) remains unused, while the value of this pointer and its metadata are stored in the safe pointer store. The safe stacks T_1, T_2, T_3 have corresponding stacks T_1', T_2', T_3' in regular memory to allocate unsafe stack objects.

metadata描述了指针指向的目标对象(target object)：对象的内存地址上下界，指针的值，时序ID。这样CPI就能在无需到所有指针访问处都插桩的前提下，保证全内存中所有敏感指针的数据安全。

插桩即为直接改变在第一步中通过CPI静态分析找到的操作敏感指针的指令，根据前面的定义直接创建或传递metadata。主要包括以下几类指令

1. 显示获取静态分配的内存对象或函数的地址，在堆上分配一块新的内存，子对象是需要被插桩的敏感指针的对象
2. 计算指针表达式的指令，需要插入传递metadata的部分
3. 敏感指针与内存读写相关的指令由CPI内联指令取代，这样在读写内存时，会同时将指针值和指针相关的metadata读取或写入安全区。
4. call和ret指令也需要插桩，不过CPI使用safe stack来保护的。

通过插桩在运行查看与该指针相关的metadata，检查所有敏感指针的解引用是否安全。与对安全区的限制访问结合起来，就能保证所有敏感指针的精确的内存安全。
通用指针在运行时同时存在安全区或常规区，但同时只有一个是有效的。若不属于安全区，则将lowbound设置成大于highbound，访问metadata时若检测到这个不合法，则去访问常规区。

安全区隔离

x86-32

依赖于硬件段保护，安全区只能通过特定段寄存器访问（这个段寄存器不会再有其他用途），这里CPI将这个段寄存器也看做Program loader，CPI也将其他通过syscall来设置段寄存器的方式禁掉了。（这里感觉有点勉强，实验中测试的程序有限，如果真的应用到工业实践中，不能保证所有的程序和系统都不会用到这个段寄存器）

x86-64

尽管仍提供了两个段寄存器，X64中不再受到段限制。与x86类似，CPI使用了其中一个寄存器，同时还为安全区随机选择一个基址。
这里是基于这样一个事实（假设）：在常规区中，没有任何地址是指向安全区的（信息隐藏，前面提到，通过指针在常规区的地址来计算其在安全区中的地址）。并且64位系统下48bit的地址空间也可以做到防止暴力破解，攻击者在多尝试几次之后程序可能就会崩溃。（这里没有指出是在编译时随机选取一个基址还是程序加载时动态指定一个基址，猜测应该是后者吧。。。）

其他架构

地址随机化或者SFI。SFI需要对程序中的所有内存操作进行轻量级插桩。实验中显示，使用了SFI后额外开销大约为5%。

由于敏感指针只占一少部分，且空间分布高度分散，所以在实现时为了节省内存，采用hash table, 多级查找表或者最简单的以地址作为下标的数组。作者最后三种方法都实现了（如果要推送到官方，效率非常重要，需要多用几种不同的模型来做测试）。
最后还提到准备用Intel 的MPX来实现安全区，这只是Intel那边的一个测试套件。
安全栈
程序运行时需要频繁地访问栈上的数据（返回地址，寄存器溢出等），为了降低性能开销和复杂性，CPI对栈进行特殊处理。
在编译时，能够静态确定大多数对栈上对象的访问是否安全，故这一部分对象就没有必要在运行时对其进行检查并生成对应的metadata。栈上大多数对象都是通过esp加上一个偏移来访问的，因此CPI将这些可以证明其访问安全性的对象放置在安全区中的一个安全栈上，访问安全栈上的数据无需任何检查，减少了CPI的开销。
如果有函数中的局部变量在栈上，并且需要对齐进行检查，则在常规区给他们分配一块独立的栈帧。
安全栈同样由三个阶段组成：

1. 静态分析：
    - 识别出每个函数中可保证被安全访问的栈上局部变量，将其放在安全栈上。
    - 返回地址和溢出的寄存器也满足这条规则，放到安全栈上

- 如果有函数中存在需要对其进行检查的变量，则需要添加指令，为其在常规栈上分配出一个栈帧。

2 插桩：

3 runtime support

与CFI相比，CPI安全栈的优势在于：

1 可以保护所有返回地址和大多数局部变量，然后CFI的shadow stack只能保护返回地址

2 安全栈能够与使用常规栈的未插桩代码完全兼容，并直接支持异常，尾调用和信号处理

3 安全栈几乎是零开销，因为只有一小部分函数需要额外栈帧，但是CFI的shadow stack为每个函数调用都分配了一个栈帧。

可能会取代栈cookie，它的安全性更强，性能开销和部署复杂度也更低。

CPS(Code-Pointer Separation)

CPS相对CPI而言，改动如下：

1 静态分析识别敏感指针的规则有所放松：

- 限制只保护代码指针，指向代码指针的指针则不受保护（不再具有propagate这个传递性，由于C++里每个类对象都有一个虚函数表，如果加上传递性，则CPI的开销会增大）。
- based-on定义放宽，只要代码指针指向控制流目标即可。
  - 这样可以防止攻击者通过一个其他类型的数据伪造代码指针
  - 但仍允许攻击者迫使程序读写代码指针

2 CPS不需要任何metadata，因为Control flow destination必须是精确的，也就无需上下界，并且都是

静态的，也就无需与时间相关的metadata（也存在例外，比如卸载共享库，这里进行了独立处理）。

- 这样就减少了安全区的大小，以及读写代码指针的访存次数