二进制代码块: 面向二进制程序的细粒度控制流完整性校验方法

王明华[1;2], 尹 恒[2], Abhishek Vasisht Bhaskar[2], 苏璞睿[3;4], 冯登国[4]

[1] 百度 X-Lab
[2] 雪城大学
[3] 中国科学院软件研究所计算机科学国家重点实验室
[4] 中国科学院软件研究所可信计算与信息保障实验室

摘要 控制流完整性(CFI)是一种在程序中通过保护间接转移有效减少代码注入和代码重用攻击等威胁的技术。由于二进制程序缺少源代码级别的语义, CFI 策略的设定需要很谨慎。现有的面向二进制的 CFI 解决方案, 如 BinCFI 和 CCFIR, 虽然能够提供对二进制程序的防护能力, 但它们应用的策略过于宽松, 依然会受到复杂的代码重用攻击。本文提出一种新的面向二进制的 CFI 保护方案, 称为 BinCC。它可以通过静态二进制重写为 x86 下的二进制程序提供细粒度保护。通过代码复制和静态分析, 我们把二进制代码分成几个互斥代码块。再进一步将代码中的每个间接转移块归类为块间转移或块内转移, 并分别应用严格 CFI 策略来限制这些转移。为了评估 BinCC, 我们引入新的指标来评估每种间接转移中合法目标的平均数量, 以及利用 call-preceded gadgets 产生 ROP 漏洞利用的难度。实验结果表明与 BinCFI 比较, BinCC 显著地将合法转移目标降低了 81.34%, 并显著增加了攻击者绕过 CFI 限制实施复杂的 ROP 攻击的难度。另外, 与 BinCC 可以降低大约 14%的空间开销, 而只提升了 4%的运行开销。

关键词 控制流完整性
中图法分类号 TP309.7 DOI 号 10.19363/j.cnki.cn10-1380/tn.2016.02.006

Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries

Minghua Wang[1;2;4], Heng Yin[2], Abhishek Vasisht Bhaskar[2], Purui Su[1;3], Dengguo Feng[1]

[1]Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy ofSciences
[2]Syracuse University
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[4]University of Chinese Academy of Sciences

Abstract Control Flow Integrity (CFI) is an effective technique to mitigate threats such as code-injection and code-reuse attacks in programs by protecting indirect transfers. For stripped binaries, a CFI policy has to be made conservatively due to the lack of source code level semantics. Existing binary-only CFI solutions such as BinCFI and CCFIR demonstrate the ability to protectstripped binaries, but the policies they apply are too permissive, allowing sophisticated code-reuse attacks. In this paper, we propose a new binary-only CFI protection scheme called BinCC, which applies static binary rewriting to provide finer-grained protection for x86 stripped ELF binaries. Through code duplication and static analysis, we divide the binary code into several mutually exclusive code continents. We further classify each indirect transfer within a code continent as either an Intra-Continent transfer or an Inter-Continent transfer, and apply separate, strict CFI polices to constrain these transfers. To evaluate BinCC, we introduce new metrics to estimate the average amount of legitimate targets of each kind of indirect transfer as well as the difficulty to leverage call preceded gadgets to generate ROP exploits. Compared to the state of the art binary-only CFI, BinCFI, the experimental results show that BinCC significantly reduces the legitimate transfer targets by 81.34% and increases the difficulty for adversaries to bypass CFI restriction to launch sophisticated ROP attacks. Also, BinCC achieves a reasonable performance, around 14% of the space overhead decrease and only 4% runtime overhead increase as compared to BinCFI.

本文是 Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries(发表于 ACSAC 2015)的扩展。

通讯作者: 王明华, 于 2016 年在中国科学院软件研究所可信计算与信息保障实验室获得博士学位。现为 Baidu X-Lab 高级研发工程师。研究领域为信息安全。研究兴趣包括: 程序安全分析、软件漏洞分析。Email: wmhfzh@163.com。

本课题得到国家科学基金会资助# 1054605, 美国空军研究实验室资助# FA8750-15-2-0106, 中国的国家基础研究发展计划资助# 2012 CB315804, 中国的国家自然科学基金资助# 91418206, 以及中国国家留学基金委(CSC)等机构出资支持。
收稿日期: 2015-11-18; 修改日期: 2016-3-15; 定稿日期: 2016-4-23

Key words control flow integrity

1 前言

ASLR[22]和 DEP[2]能够缓解传统漏洞威胁。然而, 即使 ASLR 和 DEP 开启, 攻击者仍然能够通过代码重用实施攻击[4,21], 例如 ROP[20]就是这样一种代码重用技术。近年来, 一些研究工作[3,6-8,13,19,24]已经提出针对代码重用攻击的解决方案, 并且取得了一定的成果。

控制流完整性[1]在缓解控制流劫持攻击中扮演着重要的角色, 其主要思想是依据控制流图(CFG)来限制程序中的控制流转移。对于具有源代码的程序, CFG 图的构造趋于完整, 能够依次提出严格的 CFI 约束策略。但是, 对于二进制程序而言, 由于缺乏源代码或调试信息, 无法构造完整的 CFG 信息, 所以提出的 CFI 策略仅能是粗粒度的。尽管诸如 CCFIR 和 BinCFI 等 CFI 解决方案可以阻止绝大多数控制流劫持威胁, 但是它们仍然可能受到利用 call-preceded 实现的复杂 ROP 的攻击 [5,10]。

本文提出一个新的面向二进制程序的 CFI 保护方法——BinCC, 旨在为二进制程序提供更细粒度的保护。具体的方法是: 通过复制少量的代码, 并执行静态分析, 把二进制代码分成几个互斥代码块, 并将每个间接转移分为块间转移或块内转移。接下来提出严格的 CFI 策略来约束这两种控制流转移, 其中, 块内转移只允许到达当前块内的目标, 块间转移只允许到达代码块中特定类型的目标, 从而显著地缩减了控制流转移合法目标的数量。

为了评估 BinCC, 我们引入新的指标来评估每种间接转移中合法目标的平均数量, 以及利用 call-preceded gadgets 实现 ROP 利用攻击的难度。与 BinCFI 相比较, 实验结果表明 BinCC 在这两方面均有显著的提高。其中, BinCC 将合法转移目标降低了 81.34%。尤其, BinCC 对 returns 指令(ret)提供细粒度的保护, 将合法目标数量平均降低了 87%, 显著增加了通过 call-preceded gadget 实现的复杂 ROP 攻击的难度。除此之外, BinCC 还具有合理的性能。相比 BinCFI, BinCC 运行开销仅增加 4%, 而空间开销降低 14%。

总言之, BinCC 做出了如下贡献:

- BinCC 通过代码复制和代码块构造, 将程序中的间接控制流转移分类为块内转移和块间转移, 并分别实施了细粒度的 CFI 策略。
- 与 BinCFI 和 CCFIR 相比, BinCC 显著缩减了间接控制流转移(尤其是 returns 指令)的合法目标数量。
- BinCC 不仅可以缓解常见的控制流劫持威胁, 而且能够显著增加了利用 call-preceded gadget 实施复杂 ROP 利用攻击的难度。
- BinCC 加固的程序具有合理的性能。比起 BinCFI, BinCC 能够将加固的程序大小减少约 14%, 运行时间增加约 4%。

内容组织如下。在第二章中讨论背景和相关技术, 接着在第三章中介绍代码块的概念和 CFI 策略。在第四章中介绍代码块结构, 在第五章中介绍 CFI 的实施方法。第六章为实验评估。第七章探讨本文方法的局限性。第八章对本文进行总结。

2 背景和相关技术

CFI 解决方案包括基于源代码和基于二进制两类。由于在实践中我们获得的二进制程序大部分都是没有源码的, 所以我们将侧重点放在基于二进制的解决方案上。为了说明问题, 我们对 CCFIR 和 BinCFI 的实现方式, 以及针对它们的攻击方式进行讨论。

2.1 基于源代码的 CFI

许多研究工作[3,11,12,16,23]依赖于源代码来提出 CFI 的防护策略。研究工作[11,23]主要侧重于保护虚函数的调用。主要思路是利用类继承结构分析来识别合法目标, 并插入检查代码来实现类方法和 vtable 检查。这两种解决方案都可以为虚函数调用操作提供细粒度保护, 但无法对返回操作进行保护。CFL[3]为在每个间接转移前加一个锁操作, 并仅在合法地址处进行相应的解锁操作。这个方法与我们的方法在相关调用函数返回上类似, 但是它依赖于源码, 而源码在实际应用中并不容易得到, 而且这个方法缺乏对模块化支持。MCFI[15]是一个支持模块化的 CFI 解决方案。它使用几个地址表来存储间接转移的合法目标, 并在模块动态加载时使用辅助类型信息来更新目标。在运行时, MCFI 通过执行特定的校验例程, 对间接控制流转移的合法性进行校验。

RockJIT[16]是 MCFI[15]工作的扩展, 它可以缓解与 JIT 代码相关的控制流攻击, 可以阻止由 JITed 代码引起的控制流攻击。通过使用 JIT 编译器的源代码来计算程序的 CFG, 并通过运行时产生的动态代码来更新 CFI 策略。CPI[12]提出了指针代码完整性

和指针代码分离的思想。通过有选择地保护易受控制流劫持攻击的指针访问, 该方法可以保障程序控制流安全。

2.2 基于二进制的 CFI

研究工作[14,24,25]是基于面向二进制的解决方案。O-CFI 利用一个查询表来存放合法控制流转移目标地址, 辅助 O-CFI 在运行时对转移目标的校验。O-CFI 额外使用了随机化技术对程序中所有代码块进行了随机化处理, 减低了程序遭受信息泄露的风险。SFI[24]是一种沙箱技术, 用来实施控制流完整性保护。其基本思想是使不可信的模块在相同的程序地址空间内执行, 不允许其访问其他程序的数据和代码。PittSFIled[13]和 NaCl[25]是基于 SFI 实现的, 用于保护局部代码安全, 而且其限制间接转移的目标也需要满足内存对齐的要求。

Lockdown[17]使用影子栈对 ret 指令进行约束。然而, 影子栈可能引入高的运行开销, 同时为了维护 call/ret 对, 需要更多的内存读和写操作。此外, 影子栈需要存储在安全内存域中, 这需要由硬件或类似 SFI 的分离技术来提供支持。安全内存域也可能存在信息泄露的风险, 能够被攻击者利用, 如文献[9]中一个真实的攻击实例。

CCFIR[26]是一个面向 Windows x86 PE 二进制程序的 CFI 解决方案。它将间接转移的目标安排在一个称为“springboard”的新节中。不同类型的跳转目标在 springboard 按照不同的粒度对齐。在间接控制流转移时, 首先跳转至 springboard 中, 检测转移目标地址是否按照某种粒度对齐。如果对齐, 控制流会由 springboard 中对应 stub 函数接管, 继续执行; 否则, 转移目标被认为不合法。BinCFI[27]是另一种面向 Linux x86 ELF 二进制程序的 CFI 方案。BinCFI 通过反汇编二进制码, 改写间接控制流转移指令, 再将最终代码放入新代码段。对于每个指令, 它都维护一个原位置与新位置的映射表。当间接转移发生时, BinCFI 会使其跳转至相应的地址转换例程, 在地址转换例程中完成对目标地址的查找和转换。除此之外, BinCFI 还对模块化具有很好的支持。

CFG 依赖二进制程序自身, 因此无法精确表示, 所以提出的 CFI 策略都是粗粒度的。虽然 CFI 策略能够减少大量普通控制流攻击, 但允许的策略仍然可能被攻击者绕过。图 1 显示了针对 CCFIR 和 BinCFI 可能的攻击方式。实线表示运行时的执行流, 虚线表示可能被攻击的路径。对 CCFIR 来说, 如图 1(a)所示, springboard 中的目标, 只要满足对齐到常量 M_R, 任何调用地址都将被认为是合法的。同样, 对于 BinCFI, 一个 ret 能够返回到二进制程序中的任何调用点, 如图 1(b)所示。这些返回都是不受保护的, 因此攻击者能够利用 call-preceded gadget 实施 ROP 利用攻击。工作[10]已经证实了这种攻击的可能。

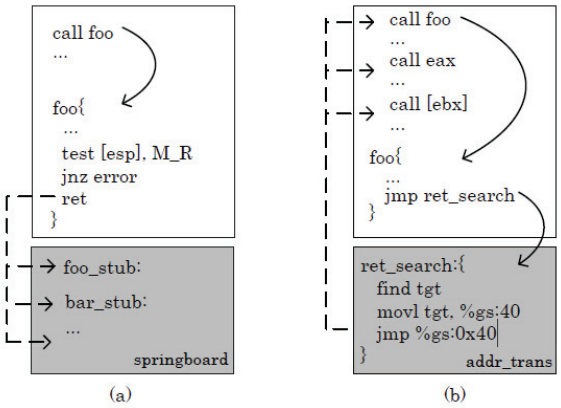


图 1 CCFIR 和 BinCFI 可能受到的攻击

3 方法框架

本文提出一种细粒度的 CFI 策略, 可以显著的改善间接控制流转移中的合法目标。通过复制一些必要代码和执行静态分析将二进制程序划分为一些独立代码部分, 并将每个间接转移定义为代码块内转移和代码块间转移。因此, 我们可以执行独立、严格的策略来达到细粒度保护。。下面, 我们将首先通过一个例子介绍代码块相关概念, 然后介绍 CFI 策略。

3.1 代码块

代码块是由函数的 Super-CFG(超级控制流图)构成的。对于一个函数, 其 Super-CFG 是由其 CFG(控制流图)构成的(Super-CFG 构造算法将在第四章讨论)。通过对函数的 Super-CFGs 进行合并, 构建每个代码块的有向图, 而这些代码块之间两两互斥。

我们通过图 2 的示例进行说明。示例中二进制代码原始函数包含: *main*, *foo*, *bar*, *qux* 和 *start*。*start* 是二进制程序的入口点。图中的节点代表代码中相应的指令。在图中, CC_1 表示由 *main* 函数的 Super-CFG 生成的代码块。由于 *foo* 函数在指令 3 处被直接调用, 所以, 由指令 5、6 和 7 组成的 *foo* 函数也被包含在 CC_1 中。CC_2 表示由 *bar* 函数的 Super-CFG 生成的代码块, 包含有四个结点。

我们将包含一个代码块中的结点分为三类: 根结点, 边缘结点和内部结点。根结点表示间接调用函数的入口, 在图中用灰色表示。例如, 1 和 5 是 CC_1 的根结点。边缘结点代表间接转移指令, 其目标无法在计算 Super-CFG 时识别, 在图中用条纹表示。例如, 2、4 和 6 都是 CC_1 的边缘结点。那些既不是根结点

也不是边缘结点的结点称为内部结点，在图中用白色表示。这些内部结点所表示的指令是那些非控制流转移指令或在 Super-CFG 中具有确定目标的控制流转移指令，例如 3 和 7。

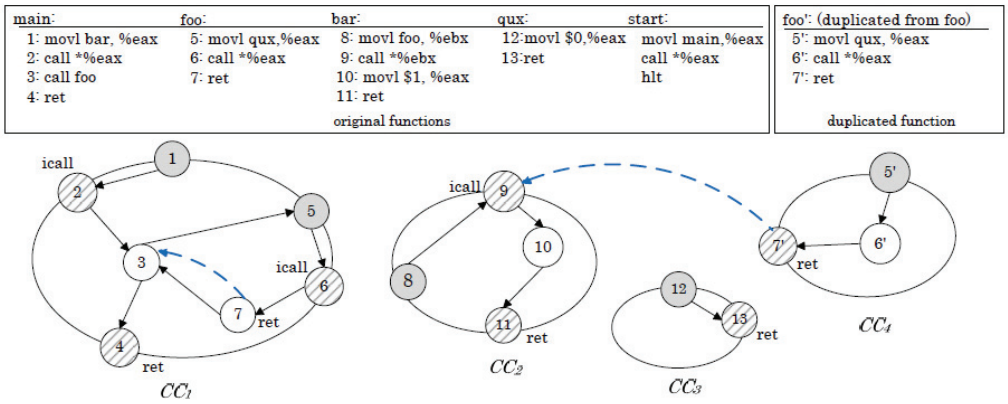


图 2　代码块示例

通过划分结点，我们可以将间接转移分为两类，即块内转移和块间转移。块内转移是由内部结点发起的间接转移，块间转移则是由边缘结点发起的间接转移。值得注意的是，在本文方法中，我们确保每个间接转移仅为块内转移或块间转移的一种，从而实施不同粒度的约束策略。为了达到该目标，我们在代码块构造前，首先需要完成对一部分代码的复制。

通常二进制代码中的函数分为两类，即间接被调函数(ICF, Indirect Called Function)和直接被调函数(DCF, Direct Called Function)。但可能存在一些函数既是 ICF 又是 DCF。我们将两类交集部分的函数进行复制，并将这部分新函数视为 ICF。这样，所有函数最终被归进两个互斥的集合中，如图 3 所示。当程序运行时，在一个间接调用位置，当原始函输被调用时，我们会实时调度来执行复制的函数。这样，在程序运行过程中，一个函数只会按照一种特定的方式被调用，即间接或直接。因此, ret 仅返回到一个特定类型的调用点。ret 也被分成两类：直接 ret 和间接 ret。其中直接 ret 返回到直接调用点；间接 ret 返回到间接调用点。

在图 2 示例代码中, ICF 由 *main*、*foo*、*qux*、*bar* 组成，而 DCF 由 *foo* 组成。只有 *foo* 函数被两种方式同时调用。假设从 *start* 开始执行，*foo* 函数首先在 9 处被间接调用，接着在 3 处被直接调用，因此返回指令 7 应该分别返回到这两个调用点。在 BinCC 中，*foo'* 是一个通过复制 *foo* 函数生成的新函数，它在指令 9 处被间接调用。这使得指令 7 变成块内转移，指令 7' 变成块间转移，也就是说 7 作为直接 ret 应该返回到 3 处，而作为间接 ret，则返回到 9 处，如两个虚箭头所示。

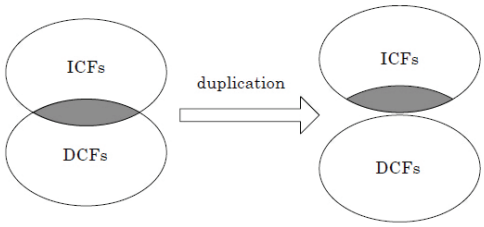


图 3　间接被调函数(ICFs)和直接被调函数(DCFs)。通过复制交集函数，函数被分为两个互斥的部分。复制的函数当作 ICFs。

3.2　CFI 策略

本小节，我们提出块内策略和块间策略来分别约束块内和块间的间接转移。

块内策略

这类策略是用来限制内部节点描述间接转移的，其转移目标是由静态分析确定的。这些目标通常存在于代码块的内部，并且在构成代码块的 Super-CFG 中确定。在一个代码块中，我们只关心两种间接控制流转移。一种是直接 ret，另一种是与 switch-case 跳转表相关的间接 jmp。对于每个直接 ret，合法的目标是处于当前代码块内的对应目标调用点。对于每个间接跳转而言，相应跳转表中所有条件分支都是合法的目标。跳转表中的条件分支可以通过静态分析确定。我们在构造 Super-CFG 时，将间接跳转与它的条件分支相关联来确定其目标。

如示例所示，这些代码块中有一个块内间接转移，即 CC_1 中 7 处的 ret，它只允许返回至调用点 3。

块间策略

这类策略用于限制边界结点，这些结点的转移目标不能从 Super-CFG 中静态确定。根据不同的转移类型，我们提出如下策略。

1. 间接 call 结点只能到达表示 ICF 入口点的根结点。

2. 间接 ret 结点只能回到表示间接 call 的边界结点。

3. 间接跳转结点, 不能通过静态分析得出其目标, 但可以达到根结点或表示间接 call 的边界结点。

这些策略的合理性在于: 在函数分成两个互斥集合之后, ICF 只能被间接调用指令调用, 因此, ICF 的间接返回只能回到间接调用点。间接 jmp 通常与跳转表相关, 这些跳转的目标可以通过静态分析得到。但是也有一些间接 jmp 无法通过静态分析得到, 它们的目标可能是 ICF 的入口点或调用点。

图 2 示例中, 根结点是 ICF(1,5,5',8 和 12)的入口点, 边界结点包含间接 call 结点(2,6 和 9)和间接 ret 结点(4,11,13 和 7')。在 BinCC 中, 间接 call 结点(2,6 和 9)可以调用 ICF(1,5,5',8 和 13), 间接 ret(4,11,13 和 7')可以返回到间接 call(2,6 和 9)。

4 代码块构造

为了构建代码块, 我们首先在二进制程序中识别所有的 DCF 和 ICF, 然后通过控制流分析来获得这些函数的 CFG, 进而计算 Super-CFG, 并基于 Super-CFG 构建代码块。下面我们将进行深入讨论。

4.1 DCF 识别

每个 DCF 是通过一个相关的 call 调用。函数的入口地址可以由 call 指令的地址和操作数计算得出。按照这种思路, 我们可以获得所有 DCF。

4.2 ICF 识别

程序中 ICF 地址与程序中存在的常量有关, 函数地址是常量自身, 或由常量计算而来。根据这一特点, 我们在程序中找出所有与函数地址相关的常量。

如果二进制程序中包含重定位表, 则所有被间接调用函数的地址信息都将在重定位表中。在这种情况下, 我们计算重定位表中的每个常量与代码基地址的和, 如果结果在此代码区中, 那么这个和即是一个 ICF 的地址。

对于不包含重定位表的二进制程序, 可能有 non-PIC 或 PIC 两种情况。在这些情况下, 我们采用同样的方法, 在二进制程序的常量中筛选 ICF。

对于 non-PIC 模块, 我们使用一个窗口值(例如: 在 32-bit 的系统中为 4 字节)在 data、.rodata、.init_array、exported symbol sections 和其他可能存在表示合法代码地址的程序段中扫描常量。同时, 我们也在代码域中收集常量。结合反汇编的结果, 如果此常量, 或常量与代码基址的和是有效代码地址, 且满足一个合法指令的边界, 我们就认为它是一个候选的 ICF。

对于一个 PIC 模块, 函数可以被 PC thunks 调用。因此, 除了与 non-PIC 执行相同的方法来收集常量之外, 我们还要识别出所有的 PC thunks, 并查看是否被用作函数调用。如果是, 那么也将这些计算出来的函数地址, 加入到候选 ICF 中。

需要注意的是, 最终得到的常量并不都是 ICF。一些常量实际上是 switch-case 跳转表中条件分支的地址或函数调用点的地址。这些显然不是函数入口点的地址, 所以我们去掉这两种候选常量, 剩下的则是最终的 ICF。

4.3 控制流分析

接下来我们通过静态控制流分析来计算所有函数的 CFG。我们尝试识别每个间接分支的所有可能的目标。这部分的主要困难在于如何分辨出间接跳转中所有可能的目标。在大多数常见的情况下, 间接 jmp 是用于跳转表的调度执行, 这个跳转的所有合法目标是对应的条件分支, 这些可以通过之前发现的常量识别出来。因此, 对间接 jmp 来说, 检查它是否与跳转表相关, 如果相关则将它与对应条件分支联系起来。

4.4 代码复制

如上所述, 为了实现本文提出的 CFI 策略, 我们需要复制 ICF 和 DCF 的交集。然而, 某些情况下, 由于编译器的优化, 一些 ICF 可能与 DCF 拥有相同的 ret, 并且这些 ret 无法明确地定义其转移类型(块内或块间)。为了解决这个问题, 如果 ICF 与 DCF 这两个函数拥有同样的 ret, 我们复制 ICF。这个复制的函数在二进制程序中会成为一个新的 ICF。

接下来我们复制函数 CFG 中所有指令。在添加 CFI 校验策略(在第五章讨论)之后, 所有代码被放置于新增的代码段中。源代码段被标记为不可执行, 而所有数据段都保持不变。被复制指令中的常量也没有变动, 所以这些指令在执行时, 仍然能够保证对数据段的正常访问。

4.5 代码块构造

代码块由函数的 Super-CFG 构成。算法 1 说明了如何计算一个函数的 Super-CFG。首先在函数中定位所有的直接调用, 然后通过 *AddSuperCFG* 在直接调用处加上被调用函数的 Super-CFG。这个函数引入了两条新的边, 一条由直接调用点(如, 结点 i)指向被调用点的入口, 另一条由被调用点返回到直接调用点。

算法 1. SuperCFG(CFG_{func}): 基于 CFG 计算 func 的 Super-CFG

输入: CFG_{func}: 计算 func 函数的 CFG

输出: sg: func 的 super-graph
1: $sg \leftarrow CFG_{func}$
2: FOREach$i \in Nodes(CFG_{func})$ DO
3: IF$i$is a direct call to fTHEN
4: $sg = AddSuperCFG(sg,i,SuperCFG(f))$
5: END IF
6: END FOR
7: RETURN sg

算法 2 显示了如何构造代码块和在代码块中划分结点。此算法的输入为所有 ICF。首先为每个 ICF 计算 Super-CFG, 然后基于共同边(例如, 共同的被调用点)合并这些 Super-CFG。这一过程通过 *MergeGraph* 实现。这个算法最终得到互斥的代码块集合, 每一个代码块包含了划分好的根结点、边界结点和内部结点。从此算法中可以看到, 根结点来自当前块中 ICF 的入口点。边界结点来自间接调用(*icall*)、间接返回(*iret*)和无法静态确定目标的间接跳转($ijmp_u$)的结点。内部结点则由表示当前块中非控制流指令和目标已确定的控制流转移结点组成。

我们还考虑了一种特殊的情况, 即可能存在无法进行静态控制流分析的孤立的代码块。例如, 一个间接跳转未的目标分支或是 dead code 等。考虑到这部分代码也可能执行, 所以也需要制定 CFI 约束。为此, 我们为每个孤立代码块生成了一个代码块。孤立代码块由 Super-CFG 组成, 此 Super-CFG 由算法 1 生成。其中, 代码块中的指令被看作"CFG"(记作 CFG_{func})。并且, Super-CFG 中的 *ibrnch*(如 *sg.ibrnch*), 是为边界结点, 它由 *icall*、*iret* 和 $ijmp_u$ 组成。Super-CFG 的 *inn*(如 *sg.inn*)为内部结点。考虑到孤立块不能作为函数, 因此不能被间接 call 调用。我们将孤立块中的入口当作边界结点, 而不是根结点。

算法 2. ConstructCC(ICFs): 将 ICFs 作为输入生成出代码块
输入: 所有 ICFs: ICFs
输出: 所有代码块: CC
1:$CC \Leftarrow \square$; $SG \Leftarrow \square$; $SG_{done} \Leftarrow \square$
2: FOR each$icf \in ICFs$DO
3: $sg = SuperCFG(CFG_{icf})$; $sg.ent = icf.entry$
4: FOR each $i \in Nodes(sg) - sg.ent$DO
5: IFi is aicall or a$ijmp_u$or a$iret$THEN
6: $sg.ibrnch = sg.ibrnch \cup \{i\}$
7: ELSE
8: $sg.inn = sg.inn \cup \{i\}$
9: END IF
10: ENDFOR
11: $SG = SG \cup \{sg\}$
12: ENDFOR
13: FOR$sg \in SG\text{-}SG_{done}$DO
14: $cc_{cur} = sg$; $SG_{done} = SG_{done} \cup \{sg\}$
15: $cc_{cur}.border = cc_{cur}.border \cup \{sg.ibrnch\}$
16: $cc_{cur}.inner = cc_{cur}.inner \cup \{sg.inn\}$
17: $cc_{cur}.root = cc_{cur}.root \cup \{sg.ent\}$
18: FOR$sg' \in SG - SGdone$DO
19: IF$HasCommonEdges(cccur, sg')$ THEN
20:$cc_{cur} = MergeGraph(cc_{cur}, sg')$
21: $cc_{cur}.border = cc_{cur}.border \cup \{sg'.ibrnch\}$
22: $cc_{cur}.inner = cc_{cur}.inner \cup \{sg'.inn\}$
23: $cc_{cur}.root = cc_{cur}.root \cup \{sg'.ent\}$
24: $SG_{done} = SG_{done} \cup \{sg'\}$
25: END IF
26: ENDFOR
27: $CC = CC \cup \{cc_{cur}\}$
28: ENDFOR
29: RETURNCC

5 CFI 实施

在构造代码块之后, 利用二进制重写方法来写入 CFI 约束规则。此部分主要通过对 BinCFI 进行扩展来实现。下面, 我们首先简要阐述 BinCFI 的基本架构, 然后再详细讨论为实施本文 CFI 策略所做的扩展和修改。

5.1 基础架构

BinCFI 基于反汇编结果对间接指令进行改写, 将最终的代码插入到新生成的代码段中, 并修改原代码段使其不可执行。对于原代码段中的每一条指令, BinCFI 使用形如<orig_addr, new_addr>的地址对, 将这条指令在原代码段中的地址关联到新代码段的地址。这些地址对用来进行从原代码段到新代码段中的地址转换。BinCFI 对所有间接转移目标产生地址对, 并用两个不同地址转换哈希表存放, 。其中, 一个哈希表用于维护 ret 指令的合法目标, 另一个用于维护间接 jmp 和 call 的合法目标。所有地址转换表都是只读属性。

BinCFI 对间接 call/jmp 和 ret 进行改写。对于与跳转表相关联的间接 jmp, 它们的操作数被改写为 $*(CE_1+Ind)+CE_2$ 这种形式, 其中 CE_1 和 CE_2 为常量, CE_1 表示跳转表相关, $*(CE_1+Ind)$表示所有可能的条件分支。另外, BinCFI 基于每个 CE_1 引入一个新的跳转表, 用来存放转换后的条件分支地址。对于剩下的间接转移, 改写过程如图 4 所示, 首先将运行目标(例如, %eax)存到一个线程局部变量中(例如, %gs:0x40), 接着执行地址转换例程 *addr_trans*。

```
call *%eax          movl %eax, %gs:0x40
                    jmp addr_trans
```

图 4　BinCFI 的间接转移测试，以间接调用为例。左侧是原指令，右侧是测试指令。

图 5 显示 *addr_trans* 的处理过程。首先检查转移是否违反 CFI 规则。如果没有，则进行地址转换。%gs:0x40 中存放的是原代码段的地址，即 orig_addr。BinCFI 通过相关地址转换表寻找对应的转换地址，即 new_addr。如果找到，则跳转到 new_addr 执行。否则，调用全局地址转换例程，来完成不同模块间的地址转换。

```
proc addr_trans:
        check_cfi_policy(orig_addr)
        if invald: trigger_alert()
        new_addr = find_trans_tgt(addr_trans_table, orig_addr)
        if found: goto new_addr
        else:  goto global_lookup_routine
```

图 5　BinCFI 中的地址转换例程。

全局地址转换例程通过查询 GTT(全局转换表)来进行地址转换。对于每个加载模块, GTT 记录了模块基地址和 *addr_trans* 地址之间的对应关系。在上述例子中，通过%gs:40 可以获知目标模块，如果 GTT 中没有找到该模块，则触发警告；如果找到，控制流则转移到目标模块的 *addr_trans* 例程，接着进行地址检测和转换操作。全局地址转换例程和 GTT 都被加入载入程序中，且每次加载到不同的内存地址。另外，当运行时加载一个新模块，也会更新 GTT，加入这个模块的相关信息。

5.2　扩展架构

为实现本文 CFI 策略，我们对 BinCFI 进行了扩展和修改。

扩展地址转换表

我们对地址转换表进行两项扩展。首先，通过代码复制，将新引入的间接转移目标引入新表项，其中包含复制的函数和返回调用点。接着，修改复制函数的有关表项。因此，如果原函数在运行时被调用，则实际执行的是对应的新函数。

我们以图 2 代码中的 foo 为例进行说明。图 6(a)展示了 BinCFI 记录的 foo 地址对，图 6(b)显示 BinCC 的扩展，在表中加入了函数入口<5’, 5’>与调用点<6’, 6’>。其中, 5_new 被修改成 5’。当 foo 在被间接调用时，将实际执行 foo’。

虽然代码复制会引入新的间接转移目标，但并不会影响保护效果。由于新加入目标的数量很少，因此仅需复制很小比例(见 6.1 节)的函数。此外，新的控制流转移也使用的相同的方法，实验结果表明，不会对保护效果造成影响。

5	5_new

(a)

5	5'
5'	5'
6'	6'

(b)

图 6　foo 的地址转换扩展表。间接转移目标由原目标替换, 5_new 即为 BinCFI 转换的 foo 的地址。

块内策略实施

这种策略用来约束代码块中代表间接转移的内部结点，如直接 ret 和跳转表相关的间接 jmp。它们的目标是确定的，可以从代码块构成的 Super-CFG 中获得。

为了约束直接 ret，我们为每个 ret 都准备了一个独立的地址转换哈希表来存储合法目标。对每个直接 ret，其目标为与 Super-CFG 相关联的调用点。对直接 ret 指令的改写过程如图 7 所示。插在两个 *prefetchnta* 指令中的 *start* 和 *size* 表明了对应地址转换表的具体位置。我们用另一个线程局部变量%gs:0x50 来存储第一个 *prefetchnta* 指令地址——*_addr*。转换例程 *addr_trans_dret* 在运行时使用%gs:0x50 来获取 *start* 和 *size* 以定位地址转换表，接着进行地址校验和转换操作。

```
_addr:  ret          _addr:  prefetchnta start
                             prefetchnta size
                             movl _addr, %gs:0x50
                             movl %(esp), %gs:0x40
                             jmp addr_trans_dret
```

图 7　BinCC 的直接返回测试。左侧是原指令，右侧是测试指令。

对与跳转表相关的间接 jmp，也可以使用相似的结构。然而, BinCFI 已经实现了对这类 jmp 转移的约束，所以，我们沿用 BinCFI 对这类 jmp 的改写方式。

块间策略实施

我们按照 3.2 节中所述的策略，对相应执行进行改写。改写方式与图 4 相似。每种控制流转移都有自己的地址转换表和地址转换例程来实现地址检查与转换。

我们需要对 PLT 条目中的间接 jmp 进行特殊处理。这类跳转用于动态符号解析。一般只有两种合法目标，一种是下一条指令的地址，也就是在符号解析之前初始化的值；另一种是在运行时确定的符号解析地址。在这些动态地址解析后，对这些 jmp 做出约束，使其只能跳转至对应的动态解析符号地址。

此外, 我们把间接跳转的所有目标安排在一个地址转换表中, 并将表放在新引入的只读数据域中。另外, 通过修改加载器使其能将数据域的属性改为可写, 然后用符号解析后的解析地址更新对应表项, 最后将可写属性改回。

除此之外, , 我们将孤立块中的间接 ret 定义为孤立 ret。由于不能通过静态分析识别孤立块的调用方, 因此, 允许孤立 ret 返回至任何函数调用点。

C++异常

我们还需要考虑 C++异常。在 C++程序中, 异常处理的必要信息存储在.*eh_frame* 域中。当异常被触发, 系统会使用当前执行的上下文来执行栈展开操作, 识别对应的 catch 分支。由于.*eh_frame* 中没有我们通过代码复制引入新代码的异常元数据, 当一个复制函数包含 C++异常逻辑且异常被触发时, 栈展开过程无法找到异常处理函数, 于是程序将会异常运行。为了避免这个问题, 我们不复制包含 C++异常逻辑的函数, 允许这些函数中的 ret 返回至任意调用点。

6 实验评估

我们通过从 SPEC CPU 2006 测试集编译得到的二进制程序对 BinCC 进行评估。编译器为 GCC 4.6.1, 编译优化级别为-O2。实验环境为 1.0GB 内存、20G 硬盘、单核的 Ubuntu 11.10 32 位虚拟机。

表 1 复制函数统计数据

程序	|ICFs|	|DCFs|	*#total*	*#dupl*	per%
lbm	5	41	45	1	2.22
gcc	6803	3287	9846	431	4.38
perlbench	2263	950	3185	83	2.61
libquantum	6	104	109	1	0.92
omnetpp	1362	727	1976	306	15.49
sjeng	142	147	287	2	0.70
gobmk	2126	731	2851	209	7.33
bzip2	55	76	130	1	0.77
milc	45	238	282	1	0.35
hmmer	249	349	597	1	0.17
povray	2011	1069	3037	136	4.48
sphinx	16	298	313	1	0.32
h264ref	133	473	598	16	2.68
astar	9	103	109	3	2.75
mcf	5	44	48	1	2.08
namd	54	76	129	1	0.78
soplex	636	487	1062	113	10.64
average	3539	1692	5637	183	3.42

6.1 代码复制评估

CFI 策略实施的一个关键操作是代码复制。我们需要评估一下复制的代码量。

表 1 列出了 ICF 和 DCF 的数量, 以及每个程序的所有函数和复制函数的总量。从中可以看到, 为了实现本文 CFI 策略, 我们仅需要复制少量的函数(约占所有函数的 3.4%)。

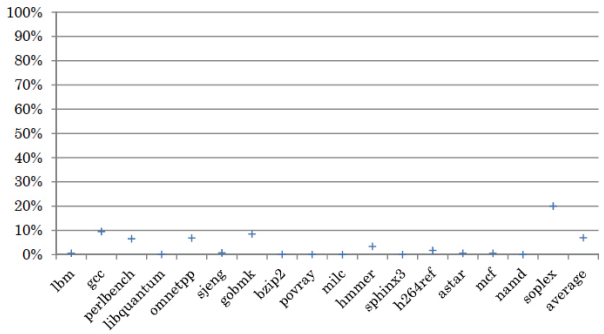


图 8 复制指令的百分数。

另外从表 1 中, 我们发现 C++程序(如 omnetpp 和 soplex)通常比 C 程序需要更多的函数复制。在这些 C++程序中, 大多数的复制函数是 C++虚拟函数。函数指针在虚拟表中被识别为 ICF, 它们既能被间接调用, 也能被直接调用(例如, 通过同一个类的继承函数), 所以我们对这些函数进行复制。

图 8 显示了示例中需要复制指令的百分比。平均而言, 只需要复制 7%的二进制指令。

6.2 间接转移的度量

BinCFI 引入平均间接目标缩减量(AIR)来评估 CFI 防护粒度。公式定义如下。

$$\mathrm{AIR}=\frac{1}{N}\sum_{j=1}^{N}\left(1-\frac{|T_j|}{S}\right)$$

在这个式子里, T_j 表示间接转换 i_j 的合法目标集。S 表示二进制编码的长度。对于 BinCC, 因为需要对代码进行复制, 因此, 编码的长度 S、间接转换的数目 N, 以及合法目标集 T_j 有可能会增加。考虑到这些因素, 计算 AIR 结果表明 BinCFI 为 98.86%, 而 BinCC 为 99.54%。

AIR 这个评估指标有失客观, 因为二进制中 S 要远高于 T_j, 这使得粗粒度的 CFI 方案都可以获得一个较高的 AIR 值。所以, 我们提出了一种新的度量方法 RAIR(Relative AIR)。RAIR 用来说明 BinCC 与 BinCFI 比较, 对间接控制流转移合法目标的缩减程度。

$$\mathrm{RAIR}=\frac{1}{N}\sum_{j=1}^{N}\left(1-\frac{|T_j|}{|T_j'|}\right)$$

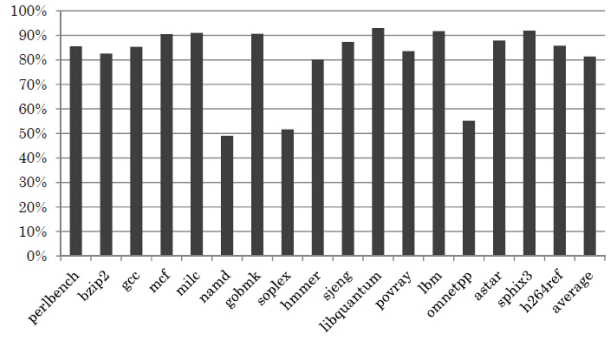


图 9　RAIR 测试示例。

其中, T_j'表示 BinCFI 中间接转换 i_j 的合理目标, T_j 表示 BinCC 中 i_j 的合理目标。图 9 列出了关于这个指标的一些统计数据。平均来说, BinnCC 比 BinCFI 减少了 81.34%的间接控制流转移目标。

相比 BinCFI, BinCC 缩减了每种间接转换的合理目标。我们用下面的表达式来评估每种间接控制流转移的合法目标的平均数量。

$$\mathrm{AVG}=\frac{1}{N}\sum_{j=0}^{N}|T_j|$$

在上式中, i_j 是某个特定种类的间接控制流指令; T_j 表示在 CFI 执行的二进制代码中 i_j 的合法目标集; N 表示二进制程序中所有间接控制流转移的数量。我们计算三种间接控制流转移的 AVG 值, 并计算 $\frac{AVG_{BinCFI}-AVG_{BinCC}}{AVG_{BinCFI}}$, 即 BinCC 与 BinCFI 相比, 对控制流转移合法目标缩减的百分比。图 10, 11, 12 分别展示了 BinCC 对间接 call, 间接 jmp 和间接 ret 缩减的百分比。

在图 10 中, 相比 BinCFI, BinCC 为每一个间接 call 缩减了 40%的合法目标。根据 BinCFI 实现, 所有可能的常量代码指针是间接调用的潜在目标。然而一些常量实际上是函数返回地址和跳转表中的分支地址, 并不是函数。我们在实现中已经把它们从目标集中移除。

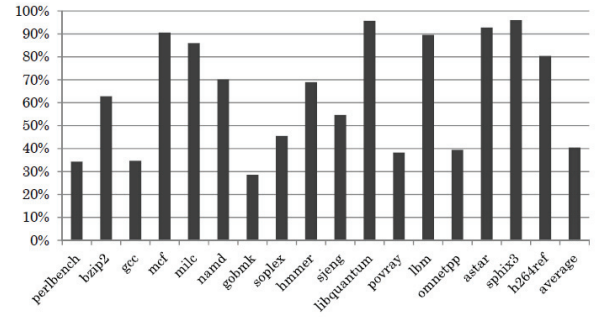


图 10　间接 call 中合法目标的减少百分比。

对于间接 jmp, 很多是在 PLT 中, 这些跳转有明确的目标, 即被解析的符号地址。因此, 相比 BinCFI, BinCC 为间接 jmp 减少了 35%的合法目标集, 如图 11 所示。

如图 12 所示, 相比 BinCFI, BinCC 为 ret 指令平均减少了 87%的合法目标。从图中可以看到, gcc 提升幅度最大。这是由于在这个二进制程序中, 直接 ret 比间接 ret 多, 且每一个直接 ret 的合法目标远少于间接 ret 的合法目标。

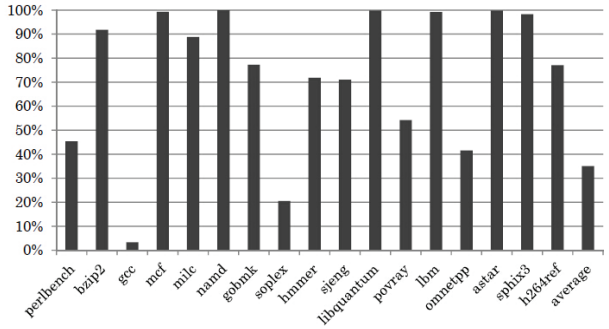


图 11　间接 jmp 的合法目标减少的百分比。

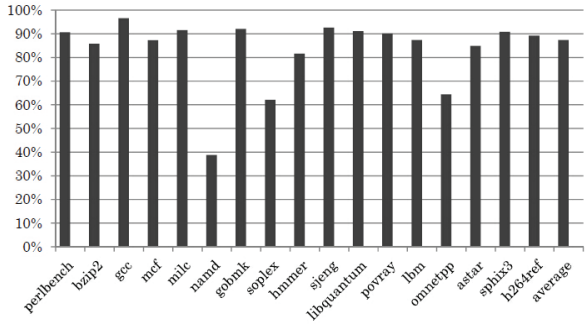


图 12　ret 的合法目标减少的百分比。

6.3　ROP 攻击评估

由于 ASLP 和 DEP 的引入, ROP 已成为攻击者常用的攻击手段。CCFIR 和 BinCFI 等 CFI 解决方案能够有效缓解传统 ROP 攻击, 然而, 研究工作[5,10]表明在一些情况下攻击者仍然能够通过 call-preceded gadget 绕过 CFI 的防护, 实施 ROP 攻击。

为了评估 BinCC 在抵御 ROP 攻击的能力, 我们提出评估指标——GS(GadgetSurvivability))来说明在 CFI 防护下, 利用 call-preceded gadget 实现 ROP 的难度。定义如下。

$$\mathrm{GS}=\frac{i}{|R|}\sum_{I=0}^{|R|}\frac{|C_i|}{|C|}$$

这个指标旨在评估: 假设在一个 CFI 加固的二进制程序中, 一条 ret 指令被攻击者完全掌控, 那么这个 ret 能够返回到一个 call-preceded gadget 的几率。

定义中 C_i 表示在 CFI 策略下 ret 指令 r_i 的合法目标集, R 表示所有的 ret 指令构成的集合, C 表示由所有 call-preceded gadget 构成的集合。对于 ret 指令 r_i, 能够达到$|C_i|$个 call-preceded gadget。所以, 它被控制并且能返回一个 call-preceded gadget 的概率是 $\frac{1}{R}\times\frac{|C_i|}{C}$。在考虑所有 ret 指令之后, 平均概率是 $\sum_{i=0}^{|R|}\frac{1}{|R|}\times\frac{|C_i|}{|C|}$或$\frac{1}{|R|}\sum_{i=0}^{|R|}\frac{|C_i|}{|C|}$。

对于 BinCC 来说, 对于每一个 r_i, $|C_i|$等于$|C|$, 所以这个指标值是 100%。这符合实际情况, 即: 攻击者能够在 CFI 下返回到任一 call-preceded gadget。

表 2 BinCC 与 BinCFI 中可用的 gadget 比例对比

程序	BinCC	BinCFI
Lbm	2.554%	100%
gcc	0.321%	100%
perlbench	0.530%	100%
libquantum	0.210%	100%
omnetpp	1.145%	100%
sjeng	0.299%	100%
gobmk	1.138%	100%
bzip2	0.474%	100%
povray	0.573%	100%
milc	0.283%	100%
hmmer	0.569%	100%
Sphinx3	0.444%	100%
h264ref	0.420%	100%
astar	0.387%	100%
mcf	0.355%	100%
namd	1.237%	100%
soplex	1.017%	100%
平均值	0.701%	100%

对于 BinCC 而言, GS 值将很小。因为 BinCC 对直接 ret 和间接 ret, 都显著缩减了它们的合法目标。当被控制的 ret 是一个间接 ret 时, 它可以返回到任一间接调用点; 如果 ret 是一个直接 ret, 则仅允许返回至以特定直接调用点起始的 call-preceded gadget。表 2 展示了 BinCC 和 BinCFI 对所有的测试用例求得的 GS 值。从表中可知, BinCC 显著降低了利用 call-preceded gadget 实现 ROP 攻击的可能。具体而言, 与 BinCFI 的 100%相比, BinCC 只有大约 0.7%的可能。

6.4 性能开销

我们对空间和时间的开销进行评估。对于空间开销, 主要比较 BinCC 和 BinCFI 地址转换表的增长量, 同时对增加的编码长度及所有文件比原始文件的增加量进行评估。对于时间的开销, 我们先执行 BinCC 和 BinCFI, 然后进行比较。

6.4.1 空间开销

由于采用相同的 BinCC 架构来改写二进制程序, 因此文件大小增加的原因和 BinCFI 的相同, 即新增的代码段和地址转换哈希表。对于 BinCC, 新代码段较原始代码长度增加了 1.4 倍; 对于 BinCFI, 这部分增加了 1.2 倍。BinCC 引入了四种表存储间接 call, 间接 jmp, 间接 ret 和直接 ret 的合法目标, 而 BinCFI 只使用两种表来存储间接 call、jmp 和 ret。总体而言, BinCC 引入的表的大小比 BinCFI 减小了 20%。这是由于哈希表大小是 2 的幂, 而 BinCC 显著缩减了间接控制流的合法目标, 使得表的大小小于 BinCFI 中表的大小。另外, BinCC 生成文件的大小比原始的二进制大小增加 125%, 比 BinCFI 低了 14%。

6.4.2 时间开销

图 13 展示了由 BinCC 和 BinCFI 加固的二进制程序的实际运行时间。在实验环境中, BinCC 加固的程序较原始程序在运行时间开销上增加 22%, 较 BinCFI 加固的程序仅增加 4%。这是因为 BinCC 在对间接控制流指令进行替换时, 增加了额外的指令, 来实现更严格的 CFI 约束策略。考虑到数据和指令的对齐会对程序执行效率产生影响, 所以, 我们在二进制重写操作前, 对一些数据和指令进行必要的对齐操作, 以此来提升访问数据和指令执行的效率, 减少时间开销。我们将在后续工作中进行完善。

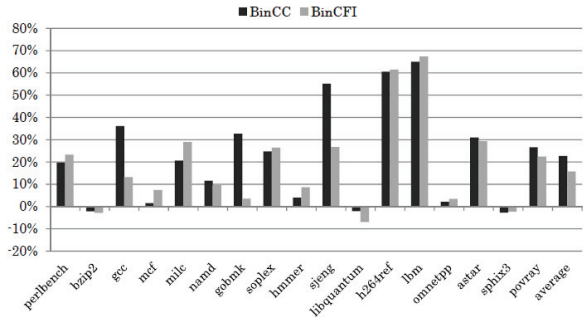


图 13 测试样例的运行开销

7 讨论

较已有的二进制 CFI 解决方案, BinCC 对间接控制流转移进行了更为细粒度的约束, 显著缩减了 ret 的合法目标, 使得可利用的 call-preceded gadget 显著减少。然而, 对于 indirect call 转移目标的约束, 尽管已经将转移目标限制在间接被调函数, 但是仍然存在被攻击者利用的可能。攻击者可以将 indirect call

操作数篡改为合法目标集合中的任何一个地址, 绕过对 indirect call 的 CFI 的检测, 可能造成程序运行崩溃或拒绝服务攻击等。这是针对二进制程序的 CFI 解决方案共同面临的问题。由于不依赖程序源代码和调试符号信息, 所以无法构造完整 CFG, 也无法准确确定 indirect call 的目标函数。研究工作[18]针对特定一类的间接函数(C++虚函数)的调用提出了约束和检验方法, 具体是对调用虚函数的 indirect call 进行加固, 仅允许调用目标为它所在类或其他同类族(class hierarchy)中的虚函数。在下一步工作中, 我们可以结合这种方法进一步提升 BinCC 对控制流指令的约束粒度。

另外, 当前 BinCC 不支持动态生成代码(例如 JIT 代码, 或者混淆代码等)。而这些问题也是面向二进制 CFI 解决方案存在的共同问题。我们将在下一步工作中对这部分内容进行系统的研究。

8 小结

对于面向二进制程序的 CFI 方法, 由于不依赖于程序源代码和调试符号, 所以已有解决方案采用粗粒度的 CFI 策略, 虽然能够防范普通控制流劫持攻击, 但是对于复杂 ROP 攻击防护能力有限。攻击者仍然能够构造利用绕过 CFI 规则的 ROP 链, 实现目的。

本部分提出了一种新的CFI解决方案BinCC, 为二进制程序实施细粒度的控制流完整性校验。该方法通过复制部分程序代码和静态分析, 将二进制程序划分为多个互斥的代码块。进一步将二进制程序中所有的控制流指令归类为代码块内转移指令和代码块间的转移指令, 并为这两种类型的控制流指令分别实施不同的转移约束条件。BinCC 严格地限定了间接控制流转移的合法目标, 显著提高控制流转移的防护效果。为了检验我们所提出的控制流完整性约束的策略, 我们提出若干新的评估指标, 并将 BinCC 和 BinCFI 进行了比较。实验结果表明, BinCC 具有较低的时间开销和空间开销。

致 谢 我们感谢帮助完成这篇论文的匿名评论者的反馈。这项工作是在 Minghua Wang 作为美国雪城大学的访问学生时完成的。本研究由国家科学基金会资助# 1054605, 美国空军研究实验室资助# FA8750-15-2-0106, 中国的国家基础研究发展计划资助#2012 CB315804, 中国的国家自然科学基金资助#91418206, 以及中国国家留学基金委(CSC)等机构出资支持。本材料中任何意见、发现和结论都属于作者, 不代表资金会机构的观点。

参考文献

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1): 4, 2009.

[2] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[3] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. *In Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. *In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[5] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. *In USENIX Security Symposium*, 2014.

[6] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng.Ropecker: A generic and practical approach for defending against rop attacks. *In Symposium on Network and Distributed System Security (NDSS)*, 2014.

[7] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. *In Proceedings of the 6th ACM Symposium on Information*, Computer and Communications Security, pages 40–51. ACM, 2011.

[8] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C.Necula. Xfi: Software guards for system address spaces. *In Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[9] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity1. *In Security and Privacy (SP)*, 2015.

[10] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. *In Security and Privacy (SP)*, 2014 IEEE Symposium on, pages 575–589. IEEE, 2014.

[11] D. Jang, Z. Tatlock, and S. Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. *In Symposium on Network and Distributed System Security (NDSS)*, 2014.

[12] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. *In USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[13] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. *In Usenix Security*, page 15, 2006.

[14] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. *In Symposium on Network and Distributed System Security (NDSS)*, 2015.

[15] B. Niu and G. Tan. Modular control-flow integrity. *In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 58. ACM, 2014.

[16] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. *In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328. ACM, 2014.

[17] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. *In Detection of Intrusions and Malware*, and Vulnerability Assessment, 2015.

[18] A. Prakash, X. Hu, and H. Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. *In Network and Distributed System Security Symposium*, NDSS, volume 15, 2015.

[19] A. Prakash, H. Yin, and Z. Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In Proceedings of the 8th ACM SIGSAC symposium on Information, *computer and communications security*, pages 311–322. ACM, 2013.

[20] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1): 2, 2012.

[21] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *In Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[22] P. Team. Pax address space layout randomization, 2003.

[23] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc&llvm. *In USENIX Security Symposium*, 2014.

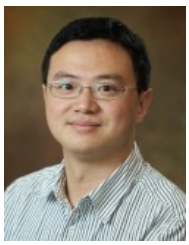
[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham.Efficient software-based fault isolation. *In ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216.ACM, 1994.

[25] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *In Security and Privacy*, 2009 30th IEEE Symposium on, pages 79–93. IEEE, 2009.

[26] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. *In Security and Privacy (SP)*, 2013 IEEE Symposium on, pages 559–573. IEEE, 2013.

[27] M. Zhang and R. Sekar. Control flow integrity for cots binaries. *In Usenix Security*, pages 337–352, 2013.

王明华 现为 Baidu X-Lab 高级研发工程师。2016 年 1 月在中国科学院大学计算机应用技术专业获得博士学位。研究兴趣包括: 程序安全分析、软件漏洞分析等。
Email: wmhfzh@163.com



Heng Yin 美国雪城大学副教授。研究领域包括恶意代码检测和分析、软件漏洞分析、移动应用安全性分析、虚拟化安全等。
Email: heyin@syr.edu

Abhishek Vasisht Bhaskar 现为美国雪城大学硕士研究生。研究兴趣包括:程序安全分析、虚拟化安全等。
Email: abhaskar@syr.edu

苏璞睿 中国科学院研究员、博士生导师, 研究方向为可信计算与信息保障。研究兴趣包括: 恶意代码动态逆向分析、移动终端应用安全、软件漏洞分析与评估等。 Email: supurui@tca.iscas.ac.cn

冯登国 中国科学院研究员、博士生导师, 研究方向为可信计算与信息保障。研究兴趣包括: 密码算法设计与分析、安全协议设计与分析、PKI 理论与关键技术、可信计算理论与关键技术、信息与网络安全等。
Email: feng@tca. iscas.ac.cn