

ARM 指针认证机制研究综述

张 军^{1,2}, 侯 锐², 李小馨², 王兴宾², 赵路坦², 国冰磊¹

¹湖北文理学院 计算机工程学院 襄阳 中国 441100

²中国科学院信息工程研究所 信息安全国家重点实验室 北京 中国 100093

摘要 内存错误漏洞是以不安全语言编写软件系统中安全性和可靠性问题的主要原因。这些漏洞常被用来将代码执行重定向到攻击者控制的位置。诸如代码复用攻击这样的内存错误漏洞利用的流行促使主要处理器制造商设计基于硬件的防御机制。一个例子是 ARMv8.3 中引入的指针认证(PAuth)机制。PAuth 使用签名密钥和指针上下文信息对指针进行签名,上下文信息是缩小保护范围和开发不同类型安全机制的关键元素。通过使用轻量级分组密码算法 QARMA64 并将指针认证码(PAC)存储在指针未使用位中, PAuth 可以较小的性能和存储开销检查指针的完整性。当前一些研究使用 PAuth 降低内存安全机制的性能开销,还有一些研究基于 PAuth 提高控制流完整性的保护精度。虽然 PAuth 受到越来越多的关注,但它仍然遭受暴力攻击和 PAC 伪造攻击。因此,很有必要对当前基于 PAuth 的安全应用进行总结,并分析其存在的问题。本文首先介绍内存错误漏洞利用的相关背景和相应的保护机制。然后,我们详细介绍了 PAuth 机制的详细信息,包括硬件支持、加密算法和密码密钥管理,及其潜在的安全问题。然后,我们总结了当前基于 PAuth 的内存安全和控制流完整性的研究,特别是指针上下文的选择方法。最后,基于我们的调查,讨论和展望 ARM PAuth 未来可能的研究方向。未来的研究方向可能包括以下几个方面: PAuth 密钥管理和上下文选择、针对推测攻击的防御,以及 PAuth 与其他 ARM 安全机制的结合使用。

关键词 指针认证; 内存错误漏洞; 控制流完整性; 指针完整性

中图法分类号 TP309.2 DOI号 10.19363/J.cnki.cn10-1380/tn.2023.11.08

A Survey for ARM Pointer Authentication

ZHANG Jun^{1,2}, HOU Rui², LI Xiaoxin², WANG Xingbin², ZHAO Lutan², GUO Binglei¹

¹ Department of Computer Engineering, Hubei University of Arts and Science, Xiangyang 441100, China

² State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

Abstract Memory corruption vulnerabilities are the primary cause of security and reliability issues in software systems written in unsafe languages. These vulnerabilities are often exploited to redirect code execution to a location controlled by the attacker. The prevalence of memory corruption exploitation like code reuse attacks has prompted major processor manufactures to design hardware-based countermeasures. An example is the pointer authentication (PAuth) introduced in ARMv8.3. PAuth signs a pointer with a signing key and a context as a nonce. The context is a critical element to narrow down the protection domain and develop different kinds of security mechanisms. By using the lightweight block cipher algorithm QARMA64 and storing the pointer authentication code (PAC) in unused bits of the pointer, PAuth can check the integrity of pointers with much less performance and storage overhead. Currently, some studies use PAuth to reduce the performance overhead of memory security mechanism, and several PAuth based research ideas have been proposed to improve the protection accuracy of control flow integrity. Although PAuth attracts more and more attention, it still suffers from brute force attacks and PAC forging attacks. Therefore, it is necessary to summarize the current security studies based on PAuth and analyze their existing problems. In this paper, we first introduce the relevant background of memory corruption exploitations and the corresponding protection mechanisms. We then present the details of PAuth mechanism, involving the hardware support, encryption algorithm and cryptograph key management, and its potential security issues. After that, we summarize the current security studies for memory security and control flow integrity based on PAuth, especially the selection methods for pointer's context. Finally, we look forward to the future research directions based on our investigation. The future research directions may include the following aspects: key management and context selection for PAuth, defense against speculative attack, and the combination of PAuth and other ARM security mechanisms.

Key words pointer authentication; memory corruption vulnerabilities; control-flow integrity; pointer integrity

通讯作者: 张军, 博士, 副教授, Email: zhangjun@hbuas.edu.cn

本课题得到“新能源汽车与智慧交通”湖北省优势特色学科群项目, 中国科学院信息工程研究所信息安全国家重点实验室开放课题(No. 2021-ZD-06)、湖北省自然科学基金(No. 2022CFB325, No. 2022CFB805)、国家自然科学基金(No. 62272459)资助。

收稿日期: 2022-03-03; 修改日期: 2022-08-25; 定稿日期: 2023-09-04

1 引言

因 C/C++等底层语言提供对内存的显式和细粒度的控制,为了保证软件系统的效率,大多数底层系统软件(如操作系统、设备驱动程序和编译器)由这类语言编写。不幸的是,直接控制内存的能力在提高软件运行效率的同时,增加了系统安全风险。当用 C/C++开发程序时,边界检查与存储管理均由程序员负责,这个过程极易出现内存错误,如缓冲区溢出(buffer overflow)和释放后重用(use after free)等^[1]。

内存错误的形式有两种:空间内存错误与时间内存错误。当程序读或写的存储区域超过了分配对象的边界时,便产生空间内存错误,如缓冲区溢出。当在对象创建前或释放后对其访问,则会产时间内存错误,这类对象的指针被称为野指针(wild pointers)或悬空指针(dangling pointers)。内存错误漏洞是软件系统可靠性和安全漏洞的根本原因,是当前主要的攻击方法之一,占总漏洞数量的四分之一左右^[2]。基于内存错误漏洞,攻击者不仅能实施信息泄漏攻击^[3-4],还能劫持程序的控制流^[5-7]。例如,代码复用攻击将系统中已有代码片断串联起来执行,实现图灵完备的计算。劫持程序控制流是内存错误漏洞利用最主要目的之一^[1]。

鉴于防御内存错误漏洞利用的重要性,几十年来,学术界和工业界提出大量防止基于内存错误漏洞利用的方法。防止内存错误的漏洞利用的方法可分为以下三类:(1)将安全敏感的数据和指针放在只读的内存区域。这种方法对函数指针的静态表很有用,但对诸如返回地址或包含函数指针的动态分配对象不适用^[1]。(2)使攻击者更难找到跳转目标。例如:地址空间布局随机化(Address Space Layout Randomization, ASLR)采用随机化的方法使程序以不同形态(每次执行随机改变代码位置)出现,使攻击者对于确定跳转目标位置,从而提高内存错误漏洞利用难度^[8]。但通过信息泄露攻击仍能绕过 ASLR 的防护^[3-4]。(3)通过指针认证检测攻击者是否基于内存错误改变程序控制流。软件栈保护(Stack Smashing Protection, SSP)^[9]、内存保护扩展(Memory Protection Extension, MPX)^[10-11]、代码指针完整性(Code Pointer Integrity, CPI)^[12]、控制流完整性(Control Flow Integrity, CFI)^[13-14]等均属于此类。因为劫持程序控制流是基于内存错误漏洞利用的最主要的目的,本文主要聚焦于基于指针认证的控制流劫持防御方法。

ARM 于 2016 年发布的指针认证机制(Pointer Authentication, PAuth)^[15-17]采用密码学方式认证指针

完整性,被研究者用于控制流劫持攻击防御。为认证指针完整性, PAuth 基于指针上下文信息,用密码学算法为指针计算指针认证码(Pointer Authentication Code, PAC)。因 64 位架构中实际的地址空间小于 64 位,指针值的高位没有实际意义,可以忽略。利用这一高位字节忽略(top-byte-ignore, TBI)的架构特点, PAuth 将 PAC 保存在指针值未使用的高位上。我们可以在指针写入内存之前为其插入一个 PAC,在使用指针之前基于 PAC 对其完整性进行认证。与加密方式相比,基于认证的指针完整性认证方式有以下优点:(1)真实的指针值依然存在,指针值仍可用于分支预测和调试;(2)可以准确知道系统崩溃的位置,而不是在系统崩溃后跳转到一个随机的位置。

目前支持 ARMv8.3-A 架构的处理器较少,仅苹果在其商业处理器中使用 PAuth 机制^[17-19],相信随着支持 PAuth 机制的处理器越来越多, PAuth 机制的研究与应用会越来越受到关注。ARM PAuth 机制的设计初衷是为以较小的性能开销认证指针完整性^[15]。近年来,研究者提出基于 PAuth 的内存安全机制^[20-25]和控制流完整性保护机制^[26-31]。PETS^[21]、PTAuth^[22]和 PACSafe^[25]利用 PAuth 基于指针上下文计算 PAC 的特点实现保障时间内存安全的 lock-key 机制; PACSan^[23]与 AOS^[24]基于 PAC 保存在指针高位的特点,将 PAC 作为内存安全机制元数据表索引,降低元数据表查找和传递引入的性能开销;基于 PAuth 的控制流完整性保护机制^[26-31]通过 PAuth 指令上下文信息的选择,提高控制流完整性机制的精度,同时基于硬件辅助降低保护机制的性能开销。尽管基于 PAuth 实现的安全机制有诸多优点,但仍存在一些问题,研究表明攻击者可基于暴力破解攻击^[32]和伪造签名组件攻击^[33]绕过 PAuth 的保护。因此,很有必要对 ARM PAuth 机制进行梳理,对当前基于 PAuth 机制实现的安全应用进行总结,分析其存在的安全问题,并讨论和展望 ARM PAuth 机制未来可能的研究方向。

本文的组织结构如下:第 2 节简述内存错误漏洞利用防御机制研究背景;第 3 节对 PAuth 机制进行梳理;第 4 节对 PAuth 的安全性进行分析;第 5 节对基于 PAuth 机制实现的防御机制进行总结;第 6 节讨论并展望未来的研究方向;第 7 节对全文进行总结。

2 相关研究背景

为了防御基于内存错误的控制流劫持攻击,我们需要理解如何实施控制流劫持攻击。如图 1 所示为控制流劫持攻击实施步骤,及针对各步的防御策略。

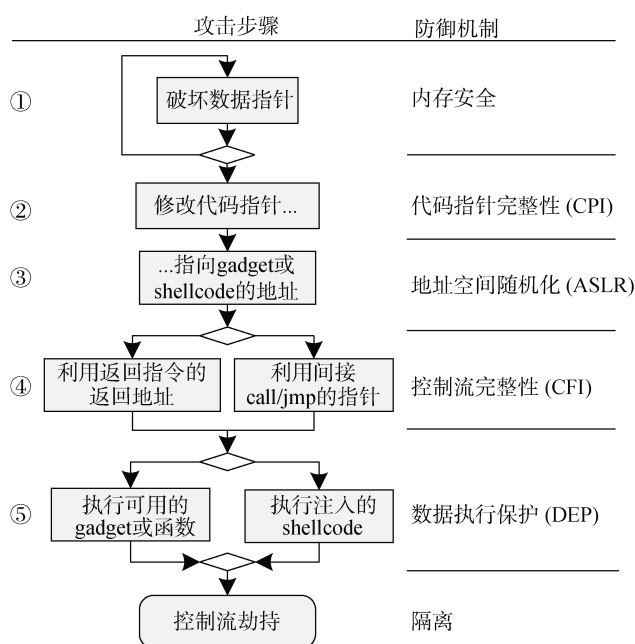


图 1 控制流劫持攻击实施步骤及各步的防御机制^[1]

Figure 1 Summary of control-flow hijack defense mechanisms aligned with individual steps that are necessary for a successful attack^[1]

实施控制流劫持攻击的根源在于系统中存在内存错误漏洞^[1]。攻击从触发内存错误开始,例如:使一个指针超出边界或悬空。若能彻底消除代码中的内存错误,则能从根源上防止控制流劫持攻击。然而为了追求高效,当前的系统程序均采用 C/C++ 这类低层次语言编写。因为这类语言缺少安全检查,很容易出现内存错误。若使用诸如 Cyclone^[34]与 CCured^[35]等具有安全检查的语言,则会带来很大性能开销。因此完全消除内存错误不现实,只能采用额外的机制在程序运行时防止基于内存错误的漏洞利用,即实现内存安全。可以通过检查指针引用的正确性发现内存错误。为了检查指针引用的正确性,研究者将指针绑定一个元数据(meta-data),元数据中包括指针的边界信息与标记等。边界信息用于检查空间上(如缓冲区溢出)的内存错误,标记用于检查时间上(如释放后利用)的内存错误。Intel 于 2013 年将内存安全保护增加到其指令集架构中,即 MPX 机制^[10-11]。目前的存储安全机制存在两方面的问题: (1) 性能开销较大,如 MPX 在硬件的支持下,其平均性能开销仍高达 50%,很难在实际中应用; (2) 元数据的存储仍存在安全性和扩展性问题。目前 Linux 内核和 GCC 编译器已停止支持 Intel MPX^[36]。

在攻击的第二步,攻击者利用内存错误将一个指针重写。例如使用缓冲区溢出重写程序栈上保存的返回地址,或利用释放后重用攻击将函数指针指

向其他位置。为了保护与控制流相关的指针, Kuznetsov 等提出 CPI 机制^[12]。CPI 通过静态分析识别代码中安全敏感指针,将安全敏感指针保存在安全区域。在这些指针引用时,通过检查指针对应的元数据判断指针引用的有效性。但 CPI 的安全性依赖于安全区域的安全性, Evans 等^[37]与 Göktaş 等^[38]通过侧信道攻击,成功找到了 CPI 的安全区域。CCFI^[39]是一种特殊的 CPI,其元数据为指针对应的信息校验码(Message Authentication Code, MAC)。CCFI 采用加密算法计算 MAC,可有效防止控制流挟持攻击。CCFI 利用 Intel 处理器的加密指令(AES.NI)加速加密算法,但 CCFI 的性能开销仍超过 50%。

在攻击的第三步,攻击者需要将代码指针指向特定的代码片段或 shellcode。针对这一步,可以采用地址空间布局随机化(ASLR)的方法防御^[8]。ASLR 采用随机化的方法使程序以不同的形态(每次执行随机改变代码位置)出现,使攻击者难于确定可复用代码的位置,从而提高攻击难度。但通过信息泄漏攻击仍能绕过 ASLR 的防护^[3-4]。

在攻击的第四步,攻击者使用重写的返回地址或函数指针将控制流转移到攻击者指定的位置。CFI 被认为是检测异常控制流转移的通用方法^[13],这种方法保证程序的控制流沿着应用的控制流图(Control Flow Graph, CFG)确定的路径执行。LLVM 编译器从 2014 年开始支持 CFI^[14]。微软从 2014 年开始,在 Windows 中增加 CFI 的保护 CFG(Control Flow Guard)^[40]。CFG 在间接函数调用前检查目标地址是否是进程位图上的合法地址。

在攻击的第五步,按攻击者的意图执行恶意注入的代码或复用已有的代码。DEP(Data Execution Prevention)能有效防止注入代码的执行,但不能防止执行复用的代码。若攻击者成功实现了前五步,他已经成功挟持了控制流。在第六步,可通过隔离(isolation)限制攻击者的能力。例如使用软件错误隔离(Software Fault Isolation, SFI)限制带有攻击的代码影响其他地址空间^[41]。Intel 的 SGX^[42]则防止攻击者破坏安全敏感地址空间的存储。

3 ARM PAuth 机制分析

PAuth 是 ARMv8.3-A 及之后 ARM Cortex-A 架构的新硬件安全特性^[15-16,43-46],该机制在 ARM 64 位(AArch64)状态下有效。增加 PAuth 的主要目的为检查关键指针的完整性。PAuth 的基本思想是,先用密码学哈希算法为指针计算指针授权码(PAC),然后将 PAC 储存在 64 位指针高位未使用的填充位中。程序

开发者或编译器可用 PAuth 指令为特定指针生成签名,并在被签名指针使用前对其进行认证^[43-46]。

3.1 PAuth 机制使用示例

在详细介绍 PAuth 实现之前,我们通过图 2(a)所示例子简要介绍 PAuth 机制的应用。图 2(a)所示函数前言将上一个函数的堆栈基地址(寄存器 x29)和该函数的返回地址(寄存器 x30)保存到堆栈上,并为该函数的参数和变量预留存储空间。函数尾声则将上一级函数使用的堆栈基地址和返回地址从堆栈中恢复出来,同时恢复上一级函数的堆栈指针现场。若该函数中存在基于堆栈的缓冲区溢出,堆栈中保存的返回地址可能被修改,使该函数返回到攻击者指定的位置。

(a) 原始代码	(b) 软件堆栈保护	(c) PA实现的堆栈保护
<pre>STP X29, X30, [SP, #-32] MOV X29, SP ... LDP X29, X30, [SP], #32 RET</pre>	<pre>STP X29, X30, [SP, #-48] MOV X29, SP ... ADRP X0, 488000 ADD X0, X0, #0x9B0 LDR X1, [X0] STR X1, [SP, #40] MOV X1, #0X0 ... ADRP X0, 488000 ADD X0, X0, #0x9B0 LDR X2, [SP, #40] LDR X3, [X0] SUBS X2, X2, X3 MOV X3, #0X0 BEQ 4007DC BL 420010 ... IDP X29, X30, [SP], #48 RET</pre>	<pre>PACIASP STP X29, X30, [SP, #-32] MOV X29, SP ... IDP X29, X30, [SP], #32 AUTIASP RET</pre>

图 2 PAuth 实现堆栈保护与软件堆栈保护的比较^[15,46]

Figure 2 The comparison between software stack protection and stack protection based on PAuth^[15,46]

为了检测堆栈中保存的返回地址是否被修改,图 2(b)所示的软件堆栈保护(Stack Smashing Protection)机制在返回地址与缓冲区间存放一个特定信息(canary),在函数返回时进行校验。如果 canary 保持不变,则程序继续进行;若被改变,则终止执行。GCC 和 LLVM 通过添加 -fstack-protector-all 选项使能软件堆栈保护^[47-48]。从图 2(b)中可以看出软件堆栈保护在函数前言增加 5 条指令,在函数尾声中增加 8 条指令。软件堆栈保护对于防御基于堆栈的缓冲区溢出是有效的,但不能防止信息泄露攻击,也不能防止攻击在不修改 canary 的情况下修改返回地址。

图 2(c)所示为基于 PAuth 实现的堆栈保护,目前 GCC 和 LLVM 均增加这项功能^[49],编译时通过添加 -mbranch-protection=pac-ret 选项使能^[46,50]。其中 -PACIASP 指令用堆栈指针作为链接寄存器(x30)的上下文,为保存在链接寄存器中的返回地址计算并插入 PAC。相应地, AUTIASP 指令则重新计算从堆

栈取回返回地址的 PAC,并与返回地址高位地址保存的 PAC 比较,进而判断返回地址是否被修改,实现对堆栈的保护。从图 2(c)可以看出, PAuth 实现的堆栈保护分别在函数的前言与尾声增加了一条指令,与软件堆栈保护机制相比,可有效降低保护机制的性能开销和存储开销。

3.2 PAuth 机制的硬件支持

除了上述的 PACIASP 与 AUTIASP 指令外, PAuth 还包括表 1 所示的指针签名与指针认证指令。为了降低 PAuth 机制的性能开销,该机制将 PAC 保存在指针值未使用的位上。为了提高 PAuth 机制的安全性,该机制采用密码学算法实现 PAC 的计算。本节从(1) PAC 的存储方式, (2) 新增 PAC 指令, (3) PAC 密码学算法与密钥管理三方面对 PAuth 机制进行详细介绍。

表 1 PAuth 相关指令^[44]
Table 1 PAuth instructions^[44]

指令	密钥	指针类型	用途
PACIAx	APIAKey	代码	签名
PACIBx	APIBKey	代码	签名
PACDAx	APDAKey	数据	签名
PACDBx	APDBKey	数据	签名
AUTIAx	APIAKey	代码	认证
AUTIBx	APIBKey	代码	认证
AUTDAx	APDAKey	数据	认证
AUTDBx	APDBKey	数据	认证
PACGA	APGAKey	通用	内存块签名
XPAC			去除 PAC

3.2.1 PAC 的存储方式

因 64 位架构中实际使用的地址空间小于 64 位,指针值的高位并没有实际意义。利用这一特点, PAuth 将 PAC 保存在指针值未使用的位上,包含 PAC 的指针格式如图 3 所示。因 PAC 保存在未用的指针扩展位上, PAC 的大小由虚拟地址空间的大小 VA_BIT 及是否使用地址标签(address tagging)^[46]决定。VA_BIT 通常为 39 位或 48 位,指针的其他位并不用于地址转换,因此可以用于存储 PAC。需要注意的是,当 ARM 内存标签扩展(Memory Tagging Extension, MTE)使能时,指针的高 8 位用于存储内存标签。操作系统通常包括用户空间和内核空间两个区域。用户空间与内核空间的虚拟地址的高 12 位分别为 0x000 和 0xFFFF。当 PAuth 使能时,虚拟地址的高位被用来存储 PAC,位 55 被保留,用于表示是用户空间的操作还是内核空间的操作。

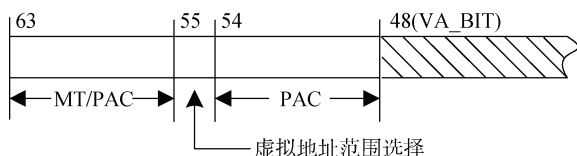


图3 包含 PAC 的指针格式

Figure 3 Pointer layout with PAC

将 PAC 放在指针未使用的位上有以下好处: (1) 通过将 PAC 直接嵌入到相应指针, 避免额外寄存器的使用, 避免增加寄存器的压力; (2) ARM 的 PAuth 机制利用底层架构的特点实现, PAC 的计算与认证均可由一条指令完成, 避免了较大的性能开销, 同时也不需要额外指令传递 PAC; (3) 与相关内存安全机制相比, 在不需要改变硬件结构的情况下, 可在现有 ARM 处理器上实现多种安全机制^[15,46]。

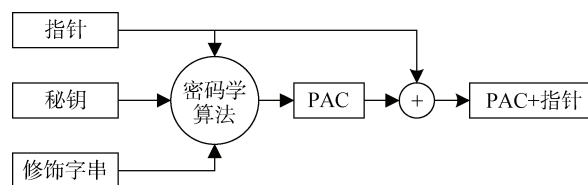
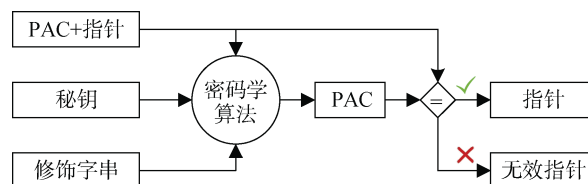
3.2.2 PAuth 指令

如表 1 所示, PAuth 有两类主要的操作: ①计算并添加 PAC; ②基于 PAC 认证指针并恢复指针值。这两个操作分别由 PAC*和 AUT*指令完成。为了防止经过认证的指针值彼此之间任意互换, PAC*和 AUT*指令通过密钥和修饰符(modifier)对的组合区分不同类别的指针。PAuth 提供了 5 个密钥, 其中 4 个密钥分配给 PAC*和 AUT*指令(指令/数据和 A/B 密钥的组合), 第 5 个密钥分配给 PACGA 指令。修饰符包括通用寄存器中的值、程序堆栈指针、零等, 这些信息为 PAC 的计算提供指针上下文信息, 防止指针间的互换, 提高 PAuth 机制的安全性。在下文中, 我们也将修饰符所包含信息称为上下文信息。例如, 堆栈指针(Stack Pointer, SP)可以在每次调用函数时具有不同的值, 但在给定函数调用的开始和结束时具有相同的值。使用 SP 作为修饰符得到的 PAC 仅对该函数的调用有效。这是因为 SP 在以后的函数调用中位于不同的位置。由此也可以看出, 生成和认证 PAC 的指令指定的修饰符必须一致, 否则会产生错误。

PAC*指令执行的过程如图 4 所示。PAC*指令被用于生成 PAC, 并将其嵌入到指针中。例如, 指令 PACIA x0, x1 将寄存器 x0 中的值作为代码指针, 将寄存器 x1 中的值作为上下文信息, 所用的密钥为 APIAKey, 通过密码学算法计算 PAC 并将其嵌入到代码指针高位中, 最后保存在寄存器 x0 中。

相应地, AUT*指令基于指针包含的 PAC 对指针的完整性进行认证。其执行过程如图 5 所示, 例如, 指令 AUTIA x0, x1 将寄存器 x0 中的值作为代码指针, 将寄存器 x1 中的值作为上下文信息, 用密钥 APIAKey 重新计算指针的 PAC, 如果指针所包含的

PAC 与重新计算的 PAC 匹配, 则对指针的认证成功, 寄存器 x0 中所保存的指针将被变成普通的指针。如果认证不成功, 则 PAC 字段中扩展位次高位(62 位)与第三高位(61 位)用错误码代替, 使指针值变成无效地址^[33,44]。

图4 PAC*指令的执行过程^[46]Figure 4 Execution process of PAC*^[46]图5 AUT*指令的执行过程^[46]Figure 5 Execution process of AUT*^[46]

在大多数情况下, 认证失败生成的无效地址在 AUT*指令后会被很快使用。实际的指针认证错误便是通过该无效指针引用时的地址转换异常检测出。处理器状态寄存器提供的信息不足以判断是否存在指针认证失败。因此, 低权限级应用很难分辨异常是源于针对指针认证的攻击, 还是应用中无害的存储操作错误。这种设计将错误处理与指针分开, 避免增加错误处理的指令。但在有些情况下, 认证失败生成的无效地址并没有立即使用, 给攻击者提供了攻击的机会^[33,51]。为了处理这种情况, ARM8.6-A 增加了两个特征: EnhancedPAC2 与 FPAC^[43]。与 ARMv8.3-A 中用 PAC 替换指针高位不同, EnhancedPAC2 将 PAC 与指针高位异或, 这样便可以防御指针重用攻击。FPAC 则用于防止攻击者基于 AUT*指令返回的地址猜测正确的 PAC 值。FPAC 的实现使 PAC 不正确时, 使 AUT*指令的执行产生异常, 防止攻击者为某个地址猜测正确的 PAC。然而, PACMAN^[32]采用推测执行攻击猜测 PAC, AUT*指令执行异常因预测执行错误而被清除。攻击者通过这种方式仍然可以猜测出正确的 PAC。

图 2 中指令 PACIASP 用 x30 作为指针, 用堆栈指针 SP 作为上下文信息生成 PAC, x30 寄存器默认为链接寄存器, 保存函数调用的返回地址, 因此, 该指令在返回地址保存在堆栈前为其增加 PAC。相应地, AUTIASP 指令用 x30 作为指针, 用栈指针 SP 作为上

下文信息认证 PAC。

除了 PAC* 与 AUT* 指令, PAuth 还提供除去 PAC 的指令 XPAC* 和为两个 64 位输入计算认证码的指令 PACGA。XPAC* 可以在未认证的情况下将签名指针中的 PAC 去除, 因此, XPAC* 指令不需要使用密钥。PACGA 指令可用于为小内存块提供保护, 在保护内存中安全敏感数据结构时很有用。PACGA 还可以链接起来保护任意大小的块, 例如, 通过调用一系列的 PACGA 指令对保存的线程状态进行签名和认证^[51-52]。

除上述基本的 PAC 生成与认证指令, ARMv8.3 PAuth 还提供一些组合指令, 这些指令将已有的操作与指针认证相结合。基于函数指针的分支跳转通常由 BLR 指令完成, 该指令跳转到函数指针指定的位置, 并用正确的返回地址更新链接寄存器。PAuth 现在提供 BLRAA 指令, 它在分支跳转到函数指针之前首先对函数指针进行身份认证。类似地, 对于函数返回, PAuth 提供 RETAA, 它在返回到函数调用位置之前认证返回地址。组合指令可以防止 TOCTOU(time-of-check-to-time-of-use) 攻击^[53]。例如, 图 2(c) 中 AUTIASP 指令先认证从栈上取回的返回地址, 再执行 ret 指令返回到函数被调用的位置。当指令 AUTIASP 执行完成后发生中断时, 寄存器 x30 中的指针已通过认证, 该值在中断处理时将保存在栈中。攻击者可重写栈中保存的寄存器 x30 的值, 然后在函数返回时可跳转到任意位置。当用 RETAA 代替这两条指令时, 指针认证与跳转由同一条指令实现, 可有效防止 TOCTOU 攻击。

3.3 密码学算法与密钥管理

由图 4 与图 5 可以看出, 密码学算法是 PAC* 和 AUT* 指令执行过程的重要部分。若采用标准的密码学算法(如 AES 等), PAC* 和 AUT* 指令的执行具有较多的执行周期延迟, 引起处理器流水线停顿。这类指针通常在代码的关键位置(函数序言和尾声), 使 PAuth 机制引入较大性能开销。另一方面, PAC 是截取密码学算法计算的信息认证码得到, 所以密码学算法的安全性要足够强。为了解决上述两个问题, 高通采用轻量级的分组密码算法 QARMA 实现 PAC 的计算与认证^[54]。

QARMA 是专为 PAuth 设计的轻量级可调整分组密码算法。QARMA 有两个版本, 其中分组长度为 64 位的版本记为 QARMA-64, 分组长度为 128 位的版本记为 QARMA-128。密钥长度为分组长度的 2 倍, 调整值长度与分组长度相同。PAuth 机制采用 QARMA-64。可调整是指, 除了密钥和明文/密文外, 代表上下文信息的调整值(tweak, 即上文介绍的 modifier)作为第

三个输入。调整值与密钥一起选择密码算法计算结果, 其中调整值是公开的, 即使攻击者完全控制调整值, QARMA 算法也是安全的。

用户进程(EL0)PAuth 密钥由内核(EL1)管理^[33,55-56]。内核为每个用户进程分配 5 个 128 位密钥, 其中 APIAKey 与 APIBKey 用于指令指针, APDAKey 与 APDBKey 用于数据指针, APGAKey 为通用认证密钥, 用于对较大的数据块进行签名^[33,44]。提供多个密钥可以对指针替换攻击提供基本保护。每个密钥对应一组指令, 例如, 指令 PACIAx 与 AUTIAx 用 APIAKey 实现代码指针的签名与认证。AArch64 架构用 10 个 EL1 寄存器保存这些密钥, 每两个寄存器分别保存一个密钥的高 64 位与低 64 位^[57]。这些寄存器只能被 EL1 及以下的程序访问, 因此, 用户空间进程不能通过访问寄存器读取或更改密钥。至 2020 年 3 月开始, Linux 内核将 PAuth 保护扩展到内核中, 可通过在编译内核时选择 CONFIG_ARM64_PTR_AUTH_KERNEL 选项使能内核 PAuth 保护, 内核代码只使用 APIAKey, 该密钥与用户 APIAKey 分时复用 APIAKeyHi_EL1 寄存器和 APIAKeyLo_EL1 寄存器。

内核与用户进程 PAuth 密钥均保存在内核 thread_info 结构体中, 该结构体是进程描述符 task_struct 结构体的一部分^[56]。需要注意的是, 内核代码使用自己的 APIAKey, 该密钥在内核启动时分配, 每次进入内核态时将内核 APIAKey 更新到 APIAKeyHi_EL1 和 APIAKeyLo_EL1 寄存器; 为降低共享库代码指针保护的性能开销, 所有用户进程共享同一个 APIAKey, 当从内核态退出时, 将其更新到寄存器中; 同一进程内的线程共享 APIBKey、APDAKey、APDBKey、APGAKey 密钥, 当用 fork() 创建新的进程时, 父进程密钥仍保存在 thread_info 结构体中。当新进程调用 exec() 函数时, 内核使用 get_random_bytes() 函数初始化 PAuth 密钥, 并用 APIBKey、APDAKey、APDBKey、APGAKey 更新相应寄存器。当 exec() 函数没有被用于启动新进程时, 用户程序可通过调用 prctl() 函数重置密钥, 从而保证不同进程使用不同 APIBKey、APDAKey、APDBKey、APGAKey 密钥。

苹果最早将 PAuth 用到产品中, 表 2 所示为 iOS 中 PAC 密钥使用情况。从表中可以看出, APIAKey 用于全局代码指针, 如函数指针、C++ 虚表项(虚函数方法)等。APIBKey 则用于线程局部代码指针, 如函数返回地址等。这是因为在 iOS 用户空间, 各进程间共享 A 簇密钥, 但拥有不同的 B 簇密钥。与 APIAKey 和 APIBKey 相似, APDAKey 用于全局数据指针的保

护, 如 C++虚表指针, APDBKey 用于线程局部数据指针保护。APGAKey 主要用于线程状态(如 PC、LR、CPSR 等)的保护。除此之外, iOS 计划将保护扩展到系统管理数据结构的成员^[58]。

表 2 iOS 中指针认证机制使用方法^[58]
Table 2 The PAuth mechanism used in iOS^[58]

签名对象	密钥类型	上下文信息
函数返回地址	APIBKey	存储地址
Objective-C 方法缓存	APIBKey	存储地址+class+selector
函数指针	APIAKey	0
块调用函数	APIAKey	存储地址
C++虚表项	APIAKey	存储地址+方法名哈希
goto 标签	APIAKey	函数名哈希
用户线程状态	APIAKey	存储地址
C++虚表指针	APDAKey	0
内核线程状态	APGAKey	*

4 安全分析

尽管 PAuth 的使用提高了攻击难度, 已有研究表明, 攻击者可在 iOS14.3 和 A14 芯片上利用漏洞, 完全绕过 PAC 保护, 执行任意 shellcode^[33,59]。因此, 很有必要对 PAuth 存在的安全问题进行分析。

4.1 暴力破解攻击

暴力破解通过大量猜测和穷举的方式尝试获得目标指针 PAC 值, 暴力破解的难度依赖于 PAC 的位数 b 及程序运行的场景。在典型的 AArch64 Linux 系统中, b 在 16~24 之间^[33,46,51]。在类 UNIX 系统中, fork() 系统函数调用能够创建一个完全相同的进程, 这意味着子进程与父进程有完全相同的 PAC 密钥^[60]。另一方面, 多线程程序中, 各线程的 PAC 密钥也相同。在 Linux 5.0 中, 内核使用 get_random_bytes() 函数初始化新进程的 PAuth 密钥, 使子进程的密钥与父进程不同。在这种情况下, 假设攻击者猜测 b 位 PAC 成功概率为 p , 并且在猜测 PAC 认证失败后使程序结束执行, 攻击需要平均尝试 $\frac{\log(1-p)}{\log(1-2^{-b})}$ 次。当 b

为 16 位, 猜测成功概率 p 为 50% 时, 攻击需要尝试 45425 次^[26-27]。

在某些情况下, 通过预先派生子进程(preforking)的方式创建一系列子进程, 达到减少频繁创建和销毁进程的开销^[61-62]。例如: prefork 模式下的 Apache 服务器在启动之初, 就预先派生一些子进程, 然后等待请求进来, 并且总是试图保持一些备用的子进程。在这些情况下, 子进程与父进程仍共享 PAC 密

钥, 攻击者可通过请求远程服务的方式获得不受限的尝试猜测 PAC 值。若失败的指针认证检查没有结束所有共享 PAC 密钥的进程, 共享 PAC 密钥不会被重新设置, 攻击者平均需要 2^{b-1} 次猜测获得 b 位 PAC^[26-27], 即当 PAC 是 16 位时, 攻击者需要进行约 32768 次尝试, 也就是说暴力破解的难度降低了。Azad 只用了 15min 便暴力破解 PACIZA 指令 24 位 PAC^[63]。

Ravichandran 等^[32]在其 2022 年 ISCA 论文中提出基于推测执行的暴力破解 PAC 的方法 PACMAN, 并实现控制流劫持攻击。PACMAN 攻击是内存破坏攻击和推测执行攻击的组合, 其中内存破坏攻击用于重写 PAuth 签名的指针和控制分支跳转方向, 推测执行攻击则基于微架构侧信道信息, 在不引起系统崩溃的情况下获得 PAC 认证结果。本节以如图 6 所示数据 PACMAN 代码片断及其执行流程为例介绍 PACMAN 攻击实施过程。在时间 t_1 前, 攻击者通过训练, 使分支跳转 BR1 发生, PAuth 签名的指针认证通过。在时间 t_1 , 攻击者使分支跳转条件不满足, 并重写 PAuth 签名的指针 guess_ptr。在 t_2 时刻推测执行指针认证指令 AUT, 如果认证通过, 则产生正确指针 verified_ptr, 否则, 产生错误指针。在时刻 t_3 , 如果 verified_ptr 有效, 则 load 指令正常执行, 通过微架构信息可观察到, 如果 verified_ptr 无效, 则产生推测异常。在时刻 t_4 , 处理器清除推测执行指令。因此, 即使猜测的 PAC 不正确, AUT 指令产生异常, 程序也不会崩溃。通过不断重复这个过程, 便能获得重写指针的正确 PAC。论文结果表明, XNU 内核包含 55159 个可能的 PACMAN 代码片断; 对于 16 位 PAC, 平均 2.94s 便能遍历所有可以有 PAC 值。由此可见, PACMAN 使针对 PAC 的暴力破解攻击更容易实施。如何防止 PACMAN 攻击是亟待解决的问题。

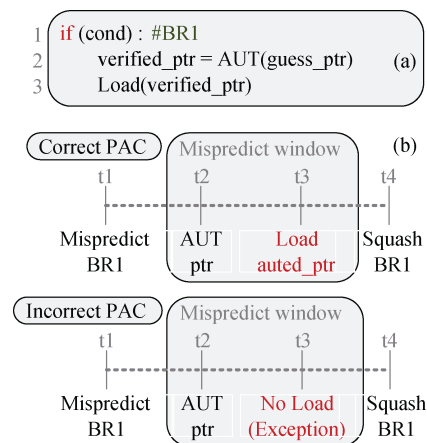


图 6 (a)数据 PACMAN 代码片断及其(b)执行流程^[32]
Figure 6 (a) A data PACMAN gadget and (b) its execution process^[32]

4.2 信息泄露与指针替换攻击

PAuth的设计旨在抵抗信息泄露攻击^[15]。PAC是通过密码学算法计算得到的,从内存中读取任意多的授权认证指针也无助于攻击者伪造授权指针。另一方面,PAuth密钥保存在处理器特定寄存器,并且这些寄存器不能被用户态的程序读取。因此,用户态的信息泄露也无助于获得计算PAC的密钥。尽管信息泄露对获得授权密钥和伪造授权指针没有帮助,但能够获得足够多被授权的指针,便可用于实施指针替换攻击。

指针替换攻击是指攻击者用一个签名指针替换另一个签名指针。例如:攻击者基于内存错误漏洞使某个puts()函数输出其提供的字符串。当用指向system()函数的被授权指针替代指向puts()函数的被授权指针时,攻击者可在特权级下执行其提供的命令^[15,46]。

为了防止指针替换攻击,PAuth机制设计时用不同类型的密钥区分不同类型的指针。除此之外,在PAC计算时用上下文信息进一步将指针进行区分,以限制哪些指针可以相互替换。例如:在GCC返回地址签名机制中,通过将当前的SP作为PAuth计算的上下文信息,使返回地址与栈帧位置绑定。但是,SP的值不唯一,不能表示特定函数的调用。当一个受攻击函数的SP与另一个函数匹配时,该函数被授权的返回地址便可以重用。对于内核中的任务堆栈,这个问题更为突出,给定用户线程发出的系统调用使用相同的内核堆栈(16KB)运行。此外,每个用户线程都有自己的内核堆栈,该堆栈在4KB边界上对齐,因此SP的低12位在线程之间重复^[29]。

另外,攻击者还可以在相同的程序栈上触发不同功能,然后基于信息泄露攻击读取程序栈上的内容,进而收集绑定不同栈地址的签名返回地址。特别是上述的预先派生进程或多线程程序共享PAuth密钥,攻击者可以在更大的搜索空间寻找有用的授权指针^[15,46,51,64]。这些收集的返回地址可以用于构造实施ROP攻击的代码串。这说明PAuth机制并不是万能的,需要进一步研究防止指针替换攻击的方法。

4.3 伪造签名组件攻击

基于伪造签名组件可以绕过PAC保护^[33,51,65]。伪造签名组件指的是一组可用于对任意指针签名的指令序列。如果攻击者可以触发一个函数的执行,伪造签名组件从内存中读取指针、添加PAC并将其写回,这样攻击者就可以利用这类指令序列伪造任意

指针的PAC。

谷歌的Project Zero 互联网安全团队提出基于PACIZA和PACIA/PACDA的伪造签名组件^[33,51]。基于PACIZA的签名组件有两种形式。第一种为如图7所示基于AUTIA+PACIZA的签名组件。Brandon Azad^[33]发现当一个指针签名不合法时,AUTIA指令执行时不仅不会产生异常,还会在合法签名中间插入错误代码并返回,即AUTIA指令返回内容包含了大部分合法签名。另一方面,PACIZA为指针添加PAC时,它实际上用还原后的扩展位对指针签名,如果扩展位本来无效,则会通过翻转PAC的一个位来破坏具有无效扩展位指针的PAC,也就是说PACIZA也会返回一部分合法签名。将PACIZA与AUTIA两个返回的部分合法签名对照,便可恢复出全部的合法签名。由AUTIA和PACIZA组成的序列可以作为签名组件,只需要在伪造的PAC中翻转一位。ARM8.6-A EnhancedPAC2特征将PAC与指针高位异或,使攻击者不能通这种方法恢复合法签名^[43]。

```

sysctl_unregister_oid:
...
LDR      X10, [X9,#0x30]!
CBNZ     X19, loc_FFFFFFFF007EBD330
CBZ      X10, loc_FFFFFFFF007EBD330
MOV      X19, #0
MOV      X11, X9
MOVK     X11, #0x14EF,LSL#48
AUTIA    X10, X11
PACIZA   X10
STR      X10, [X9]

```

图7 基于AUTIA+PACIZA的伪造签名组件^[51]

Figure 7 Signing gadget based on AUTIA+PACIZA^[51]

第二种基于PACIZA的签名组件由Samuel Groß提出^[65]。在其iMessage攻击演示中用到了两个基于ObjC方法的特殊组件:①[CNFileServices dlsym::]:该组件是对dlsym函数的包装,等价于dlsym。当PAC使能时,dlsym返回的函数指针使用进程共享的APIAKey密钥和0作为上下文进行签名,通常用作将函数指针当作回调的场景;②[NSInvocation invokeUsingIMP::]:该组件为NSInvocation的私有用法,可以接受基于APIAKey密钥和0作为上下文签名的函数指针,实现参数可控的函数调用。这两个组件相结合,可以调用任意能被dlsym找到的函数。

基于PACIA/PACDA的伪造签名组件以如图8所示的JOP方式实现^[33]。在图中寄存器X2被设置为一个可写内存区域,将X9设置为要签名的指针,将X10设置为签名的上下文。

Loc_1:	MOV	X0, X4
	BR	Loc_2
...		
Loc_2:	MOV	X9, X0
	BR	Loc_3
...		
Loc_3:	MOV	X10, X3
	BR	Loc_4
...		
Loc_4:	PACIA	X9, X10
	STR	X9, [X2, #0x100]

图 8 基于 JOP 的伪造签名组件^[33]Figure 8 Signing gadget based on JOP^[33]

5 PAuth 研究进展

选取不同的上下文信息, 可实现多种基于 PAuth 的安全机制, 本节先介绍基于 PAuth 的内存安全机制和控制流完整性保护机制, 然后介绍基于 PAuth 与 MTE 的隔离方法。

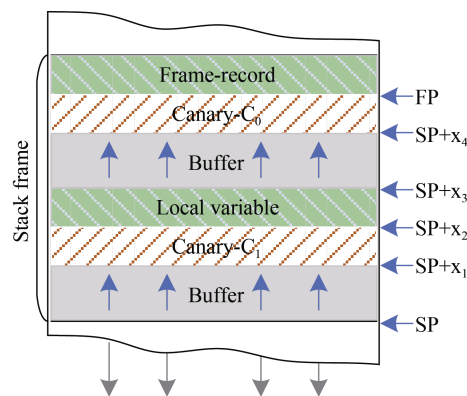
5.1 基于 PAuth 的内存安全机制

内存安全仍是当前漏洞利用的根源之一。为了实现空间内存安全, 研究者通常用包含对象基地址和大小的元数据检验指针引用是否超过对象边界; 若实现时间内存安全, 还需要在元数据中增加表示对象是否有效的信息, 如为对象分配 ID。当前内存安全机制亟待解决的问题是: 如何降低元数据传递和内存检验引入的性能开销。基于 PAuth 的内存安全机制将元数据编码到指针中, 并利用 PAuth 的硬件特征降低元数据传递和内存检验引入的性能开销。

PCan^[20] 基于 PAuth 实现栈空间内存安全, 主要解决传统单一 canary 机制依赖 canary 保密性、存在逐字节暴力破解风险及不能检测仅针对局部变量的缓冲区溢出问题。PCan 的实现方式如图 9 所示, 通过在函数序言添加指令, 在有缓冲区溢出风险的位置添加 canary, 在函数尾声添加基于 canary 检查是否存在缓冲区溢出的指令。PCan 使用 PACDA 指令计算 canary, 并将前一次计算得到的 canary(C_{i-1})作为下次计算 canary(C_i)的上下文信息。因每次执行时产生不同的 canary, 并且不用在内存中设置参考 canary, 可避免泄露或修改内存中参考 canary 的攻击。PCan 的性能开销为 0.3%, 可作为当前 GCC 和 LLVM/Clang 基于 PAuth 返回地址保护的补充。

PETS^[21]与 PTAAuth^[22]基于 PAuth 实现 lock-key 机制, 实现堆时间内存安全。二者均将 lock 值保存在对象基地址位置, 并用 PACDA/PACIA 指令对指针签名, 将对象 lock 值和对象基地址编码到指针 PAC

中, 不仅使指针指向关系能够被认证, 还能防止针对元数据的重用攻击。二者的区别在于元数据的传递方式及指向对象内部指针认证方式不同。PTAAuth 默认指针为对象的基地址, 在处理指针算术运算时, 只需对指针值进行算术运算, 不需要额外操作; 通过算术运算得到的指针往往指向对象内部, 即指针值并不是对象的基地址。当指针不是对象基地址时, PTAAuth 基于对象 16 字节对齐的特点, 从指针指向的位置向后寻找对象的基地址。PETS 采用 low-fat 内存分配器, 通过逻辑移位和与操作便能很快找到对象基地址。因此, 对指针进行算术运算操作后, PETS 便先计算对象基地址并取出对象 lock, 用 PACDA 指令对新指针进行签名。从表 3 可以看出, PTAAuth 的性能开销比 PETS 少 20% 左右, 说明基于对象基地址计算 PAC 可有效降低元数据传递引入的性能开销。PTAAuth 的实验结果表明向后寻找对象的基地址的操作可高达 10% 以上, 若 PTAAuth 采用 low-fat 内存分配器, 可快速得到对象基地址, 进一步降低性能开销。

图 9 检测所有缓冲区溢出的多 canary 机制^[20]Figure 9 Multi-canaries used to detect all overflows^[20]

PACSan^[23]与 AOS^[24]基于 PAuth 实现时间和空间内存安全, 前者内存安全保护范围涵盖堆、栈和全局变量, 后者侧重于堆内存安全。在对象分配后, 这两种方法为对象生成元数据, 并利用 PAuth PACDA 指令为对象指针签名。与 MPX 等内存安全机制相似, AOS 与 PACSan 将元数据表存储在影子存储区域。与 MPX 等内存安全机制使用对象指针作为元数据表索引不同, 这两种方法用对象指针签名作为元数据表的索引, 这样不仅简化了元数据管理, 加快元数据的存取速度, 还能减少元数据表的存储开销, 并且因对象指针签名保存在指针高位, 指针赋值、算术运算和作为函数参数传递也不需要额外的操作。因 AOS 与 PACSan 的元数据仅包含对象基地址和对象大小, 二者均不能检测出针对子对象的溢出攻击。

表 3 基于 PAuth 的内存安全机制对比

Table 3 Comparison of memory safety mechanisms based on Pauth

	PCan ^[20]	PETS ^[21]	PTAuth ^[22]	PACSan ^[23]	AOS ^[24]	PACSafe ^[25]
保护范围	栈空间内存安全	堆时间内存安全	堆时间内存安全	完整内存安全	完整内存安全	完整内存安全
PAC 作用	canaries	Key	Key	元数据表索引	元数据表索引	Key
上下文信息	栈指针与函数 ID/ 上一级签名指针	Lock	Lock	时间有效性标记和 对象大小	栈指针	0
对象分配	生成 canaries	生成 lock 和 key	生成 lock 和 key	生成有效性标记, 计算 PAC, 基于 PAC 将元数据保存 到元数据表	对象指针签名, 计算边界并保存 到元数据表	为对象分配 ID, 并将对象对应影子存储区间用 ID 填充
指针引用	/	指向授权	指向授权	指针边界检查	指针边界检查	指向授权
指针运算	/	为新指针计算 key	无操作	无操作	无操作	取原指针 PAC 作为新指针 PAC
对象释放	/	使 lock 无效	使 lock 无效	清除边界表项, 认证 heap 对象指针	清除边界表项,	将对象对应影子存储区间的 ID 置为 0x1
性能开销	0.03%	57.12%	26%	83.87%	8.4%	59.7%
内存开销		1.9%	2%	191.99%	大部分<10%	100%
元数据安全性	✓	✓	✓	×	×	✓

在实现方式上, AOS 与 PACSan 有以下不同: ①PACSan 用 PACGA 指令和随机数生成对象时间有效性标记, 并将其与栈指针作为对象指针签名上下文信息, 这样 PACSan 元数据表项对应唯一对象。AOS 仅将栈指针作为对象指针上下文信息, 使得多个对象对应同一个元数据表项。当元数据表项被填满后, 需要对元数据表项进行扩展; ②当某个对象被 realloc 函数重新分配时, 新生成对象可能与其有相同的基址、对象大小和栈指针。因 AOS 对象指针的签名不包含时间有效性信息, 不检测 use-after-alloc 漏洞; ③AOS 对 PAuth 的指针签名与认证指令进行扩展, 增加了元数据存储与清除指令, 还在处理器流水线增加内存检查单元。而 PACSan 仅利用已经有 PAuth 硬件特征, 不需要修改处理器硬件结构。

与 AOS 与 PACSan 相似, PACSafe^[25]基于 PAuth 和影子存储区间的元数据实现时间和空间内存安全。与二者不同的是, PACSafe 的元数据并不包含对象的边界信息。PACSafe 在分配对象时, 为对象分配一个 ID, 并用 ID 填充影子存储区间中与对象对应且与对象大小相同的区域。与 AOS 与 PACSan 签名对象指针并用签名指针作为元数据索引不同, PACSafe 用对象指针从影子存储区间取出对象 ID, 用 0 作为上下文信息为对象 ID 签名计算 PAC 值, 并将 PAC 值保存在对象指针的高位。对 ID 签名使指向对象的所有指针共用 PAC, 使指针赋值、算术运算和作为函数参数传递不需要额外操作, 与没有使能 PACSafe 保护的代码有较好的兼容性。因整个对象所有域共享 ID, 所以 PACSafe 与 AOS 和 PACSan 一样不能防

止对象内部的内存错误。另一方面, PACSafe PAC 中不包含对象指针信息, 攻击者可通过信息泄露攻击获得对象 PAC 值, 伪造指向对象的指针。

基于 PAuth 的内存安全机制对比如表 3 所示。从中可以看出, 基于 PAuth 实现的多 canary 栈时间内存安全机制的性能开销可忽略不计, 并且基于栈的时间内存安全漏洞较少, 因此, 内存安全机制的研究应集中在堆的保护。PACSan、AOS 与 PACSafe 均实现完全的内存安全保护, PACSan 与 PACSafe 的性能开销均超过 80%, 比基于 MPX 实现的完全内存安全保护机制 BOGO 高, 说明元数据表的管理是 PACSan 和 PACSafe 的性能瓶颈。AOS 采用 PAuth 指令和边界管理指令(与 MPX 边界管理指令类似)减少元数据表管理的开销, 其较低的性能开销从侧面说明了边界管理指令对减少堆内存安全机制性能开销的重要性, 这一假设有待在后序研究中进行验证。

5.2 基于 PAuth 的控制流完整性保护

控制流完整性保护是防止代码复用攻击的重要机制。如表 4 所示, 当前基于 PAuth 的控制流完整性保护机制^[26-31]通过 PAuth 指令上下文信息的选择, 提高控制流完整性机制的精度, 同时基于硬件辅助降低保护机制的性能开销。PAuth 指令上下文信息可分为静态和动态两类。常用的动态上下文信息为 SP, 静态上下文信息包括对象类型、函数 ID、函数地址和函数名子等。因函数返回和间接跳转引起的控制流改变特点不同, 基于 PAuth 的控制流完整性保护机制对函数返回控制流(后向控制)和间接跳转控制流(前向控制流)保护采用不同保护策略。本节先从后向控

制流完整性与前向控制流完整性保护两方面介绍基于 PAuth 的控制流完整性保护机制的研究进展, 着重介绍其上下文信息选择方法, 然后介绍基于 PAuth 与 MTE 的隔离方法。

5.2.1 后向控制流保护

高通在 PAuth 白皮书中第一次提出基于 ARMv8-A PAuth 的返回地址完整性保护机制^[15], 这种方法首先在 Linaro 的 GCC 工具链中实现, 目前 GCC^[66]和 LLVM^[67]编译器均支持返回地址签名。图 2 中用授权指令 autiasp 和返回指令 ret 分别实现返回地址的完整性认证和函数返回。若采用 retaa 指令先认证返回地址完整性, 然后返回到函数调用位置, 可防止 TOCTOU 攻击^[30]。上述方法均使用当前栈指针 SP 作为计算 PAC 的上下文信息, 将返回地址与函数栈帧绑定在一起, 防止攻击者篡改返回地址。

因为多个函数调用可能会使用相同的 SP 值, 上述用 SP 作为返回地址上下文信息的方法容易受到返

回地址替换攻击, 如 4.2 节所述, 重复的 SP 值在内核空间更加常见。为了防止返回地址替换攻击, 研究者在动态上下文信息 SP 的基础上增加静态上下文信息。如表 4 所示, PARTS^[27]将编译时分配的函数 ID 作为返回地址的静态上下文信息, 并将其与 SP 拼接作为上下文信息; PAL^[31]则将函数名的哈希值作为返回地址的静态上下文信息; Camouflage^[29]基于内核虚拟地址空间保护不变, 函数虚拟地址唯一的特点, 将函数地址作为内核中返回地址的静态上下文信息。但从函数序言到函数尾声, Camouflage 需要一个寄存器保存函数的地址。PACStack^[26]则用上一级函数调用的签名返回地址作为当前函数调用返回地址的上下文。这种方法将程序栈上返回地址签名串连在一起, 只要保证链首的安全性, 便能保证所有返回地址的安全性。PACTIGHT^[28]采用与 PACStack 类似的后向控制流保护方法, 并且在返回地址上下文信息中增加了函数 ID。

表 4 基于 PAuth 的控制流完整性保护机制对比
Table 4 Comparison of CFI mechanisms based on PAuth

	后向控制流保护		前向控制流保护	
	上下文信息	性能开销	上下文信息	性能开销
PACStack ^[26]	上一级函数调用签名返回地址	3%	—	—
PARTS ^[27]	SP[15:0] Function-ID[47:0]	<0.5%	Hash(ElementType)	19.5%
PACTIGHT ^[28]	上一级函数调用签名返回地址, Function-ID	0.64%	指针位置与指针标签	5%
Camouflage ^[29]	SP[31:0] FunctionAddress[31:0]	10%~28%	OwnerAddress[47:0] ObjectMember-ID	10%~30%
PACKER ^[30]	SP	2%~7%	PointerAddress	10%~20%
PAL ^[31]	SP[31:0] Hash(Function-name)[31:0]	0~1 μ s	Hash(PointerType)/Hash(OwnerType)/ObjectField/PointerAddress/CallSite	13%

表 4 还给出了各后向控制流保护机制的性能开销。PARTS 后向控制流保护机制的性能开销小于 0.5%, 其性能开销与 LLVM 基于 PAuth 的返回地址保护 (-mbranch-protection) 性能开销相当(0.43%)^[47], PACStack 基于 SPEC CPU2017 在 ARMv8-A FVP (Fixed Virtual Platform)评测的性能开销约为 3%, 其性能开销增加归因于将通用寄存器 X28 保留为签名链首的保护和增加的存操作。PACTIGHT 基于 SPEC CPU2006 在苹果 M1 处理器评测的性能开销约为 0.64%, 其与 PACStack 评测结果相差较大的原因可能有两点: ①二者所有测试程序不同; ②在 FVP 上的评测结果与苹果 M1 处理器上的评测结果差异较大。Camouflage、PACKER 和 PAL 是对内核返回地址的保护, PAL 最大增加了 1 μ s 的系统调用延迟; PACKER 的性能开销在 2%~7%之间; Camouflage 的性能开销则在 10%~28%之间。因这些方法采用的测

试程序和测试平台各不相同, 且评测结果差异较大, 需要统一的评测方法对基于 PAuth 的内核后向控制流保护机制的性能开销进行评估。

5.2.2 前向控制流保护

与后向控制流完整性只保护返回地址不同, 前向控制流完整性保护的实施需要定义和识别要保护的指针。与 CPI 类似, 基于 PAuth 的前向控制流完整性保护机制均基于 LLVM 编译器的中间表示识别函数指针, 并插入指针认证代码, 各种基于 PAuth 前向控制流保护机制采用的上下文信息如表 4 所示。

PARTS^[27]与 PACTIGHT^[28]通过保护局部指针、全局指针、静态指针及结构体中的指针实现代码指针完整性, 防止针对控制流和数据流的攻击。PARTS 将代码指针和数据指针类型作为上下文信息。为了处理指针类型转换, PARTS 在指针载入时用原指针类型对其进行认证, 并去掉 PAC; 在保存指针时用

转换后指针类型重新对指针签名。PARTS 提供的安全保护相当于细粒度的控制流完整性,即只允许基于函数指针跳转到具有相同类型的函数,因此,PARTS 仍存在指针替换风险。PACTIGHT 将指针位置与指针标签作为指针的上下文信息,并采用与 CPI 类似的元数据表管理上下文信息。PACTIGHT 元数据包含指针标签,指针类型大小和指针队列大小。其中指针标签与指针位置相异或作为指针签名的 modifier,指针类型大小和指针队列大小则用于支持指针运算和实现空间内存安全。PACTIGHT 同样面临元数据表泄露的问题,因指针上下文信息中包含指针位置信息,攻击者并不能将签名指针用在其他位置,但仍可能在相同位置实现签名指针重用攻击^[28],或基于泄露的指针标签伪造签名指针^[33]。

Camouflage^[29]、PACKER^[30]和 PAL^[31]实现内核控制流完整性保护。尽管内核中大多数函数指针位于只读段.rodata 中静态操作结构体(operation structures),但一些单独的指针及指向操作结构体的指针仍存在被修改的风险。为了避免保护所有内核指针增加的性能开销,Camouflage^[29]标记内核中需要保护的指针,包括复合类型申明中的指针成员。为了提高 CFI 保护精度,Camouflage 用对象的地址与类型作为指针动态与静态相结合的上下文信息。与 Camouflage 相似,PACKER^[30]将函数指针变量的虚拟地址作为计算 PAC 的上下文信息。PACKER 基于 LLVM IR,通过域敏感的分析方法识别内核中需要保护的函数指针。为了解决签名指针作为参数传递时上下文获取问题,PACKER 将函数指针地址编码到函数指针中。

在前述工作的基础上,PAL^[31]提出更细粒度,多层次 PAuth 上下文信息,前向 CFI 的保护精度可以根据性能要求进行调节。PAL 采用动态与静态相结合的多层次 PAuth 上下文信息。其中静态上下文信息包括两方面:①函数指针的类型特征,即函数申明类型的哈希,与 PaX RAP^[68]类似;②函数指针拥有者的类型。这两种上下文信息相结合,将内核中(void *) (void *)类型的函数分为 35 种不同的类,这实际上便是 MLTA^[69]限制间接函数调用目标地址的方法。动态上下文信息通过工具标注实现,也包括两种情况:①对象嵌入指针在其生命周期中保持不变,将指针与对象中某些域或地址绑定;②对于非嵌入指针,将其与指针调用的上下文绑定。为了防止攻击者通过修改因编译器优化或进程切换保存到内存的寄存器值实施 TOCTOU, PAL 用时间戳作为上下文信

息,并采用与 PACStack 相似的签名链结构对保存到内存的寄存器值签名和认证。PAL 的上下文分析工具基于输入的安全等级对源代码进行分析,并自动标注选择的上下文信息。

上述基于指针完整性的控制流保护机制只保护间接控制流改变,没有考虑基于异常的攻击。防异常攻击控制流完整性机制(FCFI)需检测程序是否按控制流图确定的基本块^①序列执行,实现时需要设置全局CFI状态,并在基本块转换时对其更新。FIPAC^[70]将ARM PAuth机制用于全局状态的更新。为了考虑控制流的历史,FIPAC用式(1)所示的状态更新函数将下一个控制流状态与之前的控制流状态累积在一起。然而,通过故障改变程序中的代码指针,使攻击者可以按照控制流图的边界改变间接分支的控制流。为了防止这种攻击,Nasahl等^[71]通过扩展FIPAC,基于ARM PAC机制保护所有间接跳转用到的地址。他们在编译时将这些地址替换为嵌入PAC的地址,将所有跳转指令替换为相应的PAuth指令。

$$S = \text{pacia}(S_p, PC, K_A) = \text{MAC}_{K_A}(S_p, PC) \oplus S_p \quad (1)$$

5.2.3 小结

从表4可以看出,动态上下文与静态上下文相结合可提高控制完整性保护的精度。前向控制流保护与后向控制流保护所用静态上下文信息相似,主要为对象/指针类型、函数名、函数ID、函数地址等。后向控制流完整性保护通常采用SP作为返回地址的动态上下文信息,而前向控制流保护的动态上下文信息则包括指针地址、指针所在对象其他域信息和函数调用位置等。如何识别安全敏感指针,并根据指针特点选择适合的上下文信息并平衡安全性和性能开销是值得研究的问题^[31]。

5.3 基于 PAuth 与 MTE 的隔离方法

为了防止控制流攻击和面向数据编程攻击,Giannaris等基于ARM PAuth与MTE实现区间隔离方法HAKC^[72]。MTE是ARMv8.5-A架构推出的内存标签扩展(Memory Tagging Extension)^[73],尝试从芯片架构设计层面来解决内存安全错误的技术难题。MTE用4位标签标记内存中16字节的数据,并将4位标签保存在指针的高字节上,相当于对内存“染色”。当存取标记内存区域时,MTE通过检查内存标签与指针高位标签是否一致,判断是否存在破坏内存安全的操作。

①基本块内指令顺序执行,没有控制流改变

若基于MTE实现区间划分, 只最多只能构成16个分隔区域。HAKC将MTE与PAuth相结合实现二级区间划分方法, 可提供更细粒度的分隔区域。HAKC区间划分实现方法如图 10所示, 图中矩形表示Compartment, 为一级区间划分, 矩形中包含的圆形表示Clique, 为二级区间划分, Clique的划分可用16种颜色表示。HAKC将Compartment号保存在指针的高位, 用MTE指令将Clique的颜色保存在内存标签中。在指针认证时, 将Compartment号与Clique颜色拼结成64位值, 作为指针认证的上下文信息, 从而达到区间隔离的目的。

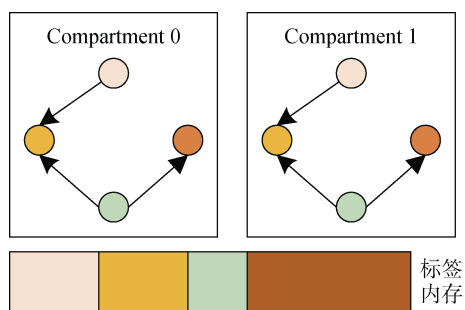


图 10 HAKC 区间划分实现方法^[69]

Figure 10 The method of HAKC compartmentalization^[69]

6 PAuth 应用面临挑战和研究方向

在之前的章节中, 对PAuth的应用情况与研究现状进行了介绍与总结。目前PAuth机制的应用还处于一个早期阶段, 依然存在一系列的挑战尚待解决, 本节对未来的研究方向进行讨论和展望。

(1) **PAuth 密钥管理。**PAuth 机制的安全性依赖于其密钥的安全性。因为PAuth的签名算法是公开的, 若签名密钥泄露, 攻击者可以伪造签名的函数指针。PAuth 密钥由操作系统内核管理, 而内存安全问题是当前操作系统主要安全问题之一, 仅在 2021 年, 208 个 CVE 披露的漏洞利用与 Linux 内核有关^[74]。这说明由操作系统管理PAuth密钥存在安全隐患, 另一方面也说明操作系统内核也需要基于PAuth的安全机制保护。目前Linux内核已使用PAuth保护代码指针完整性, 但像iOS等并不运行hypervisor (EL2)或安全监视器(EL3)的系统, 需要操作系统内核管理其自身的PAC 密钥。攻击者可基于内核内存错误漏洞读取内核PAC 密钥并替换EL0 用户态进程PAC 密钥, 然后通过用户在用户空间执行PACIA 指令伪造内核签名指针。因此, 很有必要研究与操作系统隔离的密钥管理机制^[75]。

另一方面, 用户空间进程使用相同的APIAKey

密钥, 加上有些PAuth 指令以零作为签名上下文信息, 攻击者可以在自己的进程中对指针签名。当该指针在其他进程中使用, 将通过BLRAAZ/BRAAZ 等指令的认证, 这意味着又可以实施JOP 攻击^[33,64]。用户进程共享APIAKey 密钥的根本原因在于共享库文件, 若不同的进程使用不同A族密钥, 则创建新进程时需要花费大量时间和空间复制共享库文件。因此, 我们需要研究更安全有效的共享库文件方法。

(2) **PAuth 上下文信息选择。**从上面的分析可知, 选择不同的上下文信息可以实现不同类型的安全机制, 同时上下文信息的选择也决定了安全机制的保护力度。当前上下文信息包括静态与动态两类, 静态上下文信息包括上文所述的指针类型^[27], MLTA^[69]提出的多层类型分析在指针类型上对上下文信息进一步细化。SP 则属于典型的动态上下文信息, 常用于返回地址的签名, 即后向控制流完整性保护。前向控制流完整性所使用的动态上下文信息则较难确定。PACKER^[30]将函数指针变量的虚拟地址作为计算PAC 的上下文信息, 并将函数指针虚拟地址编码到函数指针中, 使其能方便获得上下文信息, 这种方法改变了指针编码结构。PAL^[31]则基于指针类型不同, 采用标注方式为指针绑定上下文信息, 在签名指针传递时, 其面临上下文信息传递的问题。从以上分析可知当前动态上下文信息仍存在不足, 如何定义安全有效并便于获取的上下文信息是需要进一步研究的问题。

(3) **推测攻击防御。**当前基于PAuth的安全机制的威胁模型均没有考虑推测执行攻击。研究表明, 基于推测执行的暴力破解方法平均只需2.94s 便能遍历所有可能PAC 值, 使第5节所述的基于PAuth的安全机制安全性大打折扣。若限制包含基于AUT*指令认证指针的内存操作或分支跳转的推测执行, 则会引入较大性能开销。如何防止基于推测执行的暴力破解PAC 是亟待解决的问题。

(4) **多种安全机制综合应用。**除了本文重点介绍的PAuth 机制外, ARM 体系结构还包括TrustZone、MTE、BTI(Branch Target Identification)^[76]等安全机制。目前将这些安全机制综合应用于内存安全和控制流完整性保护的研究还较少, 如何将这安全机制综合应用, 提高内存安全和控制流完整性保护机制的安全性, 降低防御机制的性能开销值得尝试。

7 总结

内存安全问题是当前系统安全问题的根源之一, ARM PAuth 以硬件方式降低防御机制性能开销, 受

到广泛关注。本文详细介绍了 ARM PAuth 的硬件支持, 指令执行过程及其密钥管理方式; 从暴力破解、指针替换攻击、伪造签名组件三方面对 PAuth 的安全性进行分析; 最后, 对基于 PAuth 的内存安全机制和控制流完整性保护机制研究现状进行总结。通过以上分析, 我们认为以下几个方面可能会成为未来的研究方向: ①研究基于硬件隔离了 PAuth 密钥管理机制, 提高防御机制的安全性, 研究新的密钥分配方式, 实现更安全有效的共享库文件方法; ②研究如何定义安全有效并便于获取的动态上下文信息; ③研究防止基于推测执行暴力破解 PAC 的方法; ④研究如何综合利用多种安全机制, 提高内存安全和控制流完整性保护机制的安全性, 降低防御机制的性能开销。

致 谢 衷心感谢各位评审专家对本文提出的宝贵意见。本课题得到“新能源汽车与智慧交通”湖北省优势特色学科群项目, 中国科学院信息工程研究所信息安全国家重点实验室开放课题(No. 2021-ZD-06)、湖北省自然科学基金(No. 2022CFB325、No. 2022CFB805)、国家自然科学基金(No. 62272459)资助。

参考文献

- [1] Szekeres L, Payer M, Wei T, et al. SoK: Eternal War in Memory[C]. *2013 IEEE Symposium on Security and Privacy*, 2013: 48-62.
- [2] Vulnerabilities By Type. CVE Details: The ultimate security vulnerability data source. <http://download.cvedetails.com/vulnerabilities-by-types.php>. Dec. 2021.
- [3] Rudd R, Skowrya R, Bigelow D, et al. Address Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity[C]. *2017 Network and Distributed System Security Symposium*, 2017: 1-15.
- [4] Gras B, Razavi K, Bosman E, et al. ASLR on the Line: Practical Cache Attacks on the MMU[C]. *2017 Network and Distributed System Security Symposium*, 2017: 1-15.
- [5] Shacham H. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)[C]. *The 14th ACM conference on Computer and communications security*, 2007: 552-561.
- [6] Bletsch T, Jiang X X, Freeh V W, et al. Jump-Oriented Programming: A New Class of Code-Reuse Attack[C]. *The 6th ACM Symposium on Information, Computer and Communications Security*, 2011: 30-40.
- [7] Schuster F, Tendyck T, Liebchen C, et al. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C Applications[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 745-762.
- [8] Address Space Layout Randomization. IBM Documentation. <https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization>. Dec. 2021.
- [9] Security Feature Parity: GCC and Clang. Linux Plumbers Conference. <https://outflux.net/slides/2019/lpc/gcc-and-clang.pdf>. 2019.
- [10] Intel Memory Protection Extensions Enabling Guide. Intel. <https://www.intel.com/content/www/us/en/developer/articles/guide/intel-memory-protection-extensions-enabling-guide.html>. Jan. 2016.
- [11] Oleksenko O, Kuvaiskii D, Bhatotia P, et al. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack[J]. *The ACM on Measurement and Analysis of Computing Systems*, 2(2)Article No. 28, 2018.
- [12] Kuznetsov V, Szekeres L, Payer M, et al. Code-pointer integrity[C]. *USENIX 11th Symposium on Operating Systems Design and Implementation*, 2014: 147-163.
- [13] Abadi M, Budiu M H, Erlingsson Ú, et al. Control-Flow Integrity Principles, Implementations, and Applications[J]. *ACM Transactions on Information and System Security*, 2009, 13(1): 1-40.
- [14] Tice C, Roeder T, Collingbourne P, et al. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM[C]. *The 23rd USENIX conference on Security Symposium*, 2014: 941-955.
- [15] Pointer Authentication on ARMv8.3. Qualcomm Technologies, Inc. <https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf>. Jan. 2017.
- [16] Arm Architecture Reference Manual Armv8, for A-profile architecture. ARM Developer. <https://developer.arm.com/documentation/ddi0487/gb/>. July. 2021.
- [17] ARMv8.3 Pointer Authentication in xnu. Open Source at Apple. <https://opensource.apple.com/source/xnu/xnu-7195.50.7.100.1/doc/pac.md>. Dec. 2021.
- [18] Preparing Your App to Work with Pointer Authentication. Apple Developer. https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication. Dec. 2021.
- [19] Apple 平台安全保护. Apple. <https://support.apple.com/zh-cn/guide/security/welcome/web>. Dec. 2021.
- [20] Liljestrand H, Gauhar Z, Nyman T, et al. Protecting the Stack with PACed Canaries[C]. *The 4th Workshop on System Software for Trusted Execution*, 2019: 1-6.
- [21] Aweke Z B. Leveraging Processor Features for System Security[D]. Ann Arbor, MI, USA: University of Michigan, 2019.
- [22] Farkhani R M, Ahmadi M, Lu L. PTAAuth: Temporal Memory Safety via Robust Points-to Authentication[EB/OL]. 2020: arXiv: 2002.07936. <https://arxiv.org/abs/2002.07936>.
- [23] Li Y, Tan W D, Lv Z Z, et al. PACSan: Enforcing Memory Safety Based on ARM PA[EB/OL]. 2022: arXiv: 2202.03950. <https://arxiv.org/abs/2202.03950>.
- [24] Kim Y, Lee J, Kim H. Hardware-Based Always-on Heap Memory Safety[C]. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020: 1153-1166.
- [25] Hohentanner K, Zieris P, Horsch J. CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++[EB/OL]. 2022: arXiv: 2202.08669. <https://arxiv.org/abs/2202.08669>.
- [26] Liljestrand H, Nyman T, Ekberg J E, et al. PACStack: an Authenticated Call Stack[C]. *USENIX Symposium on Operating*

- Systems Design and Implementation*, 2021: 357-374.
- [27] Liljestrand H, Nyman T, Wang K, et al. PAC it Up: Towards Pointer Integrity Using ARM Pointer Authentication[C]. *The 28th USENIX Conference on Security Symposium*, 2019: 177-194.
 - [28] Ismail M, Quach A, Jelesnianski C, et al. Tightly Seal your Sensitive Pointers with PACTight[EB/OL]. 2022: arXiv: 2203.15121. <https://arxiv.org/abs/2203.15121>.
 - [29] Denis-Courmont R, Liljestrand H, Chinae C, et al. Camouflage: Hardware-Assisted CFI for the ARM Linux Kernel[C]. *2020 57th ACM/IEEE Design Automation Conference*, 2020: 1-6.
 - [30] Yang Y T, Zhu S B, Shen W B, et al. ARM Pointer Authentication Based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels[EB/OL]. 2019: arXiv: 1912.10666. <https://arxiv.org/abs/1912.10666>.
 - [31] Yoo S, Park J, Kim S, et al. In-Kernel Control-Flow Integrity on Commodity OSes Using ARM Pointer Authentication[EB/OL]. 2021: arXiv: 2112.07213. <https://arxiv.org/abs/2112.07213>.
 - [32] Ravichandran J, Na W T, Lang J, et al. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution[C]. *The 49th Annual International Symposium on Computer Architecture*, 2022: 685-698.
 - [33] Examining Pointer Authentication on the iPhone XS. Google Project Zero. <https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html>. Feb. 2019.
 - [34] Jim T, Morrisett J G, Grossman D, et al. Cyclone: A Safe Dialect of C[C]. *The General Track of the annual conference on USENIX Annual Technical Conference*, 2002: 275-288.
 - [35] Necula G C, Condit J, Harren M, et al. CCured: Type-Safe Retrofitting of Legacy Software[J]. *ACM Transactions on Programming Languages and Systems*, 2005, 27(3): 477-526.
 - [36] Intel MPX Support in the GCC Compiler. GCC Wiki. <https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler>. 2020.
 - [37] Evans I, Fingeret S, Gonzalez J, et al. Missing the Point(Er): On the Effectiveness of Code Pointer Integrity[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 781-796.
 - [38] Göktas E, Gawlik R, Kollenda B, et al. Undermining Information Hiding (and what to do about it)[C]. *The 25th USENIX Conference on Security Symposium*, 2016: 105-119.
 - [39] Mashtizadeh A J, Bittau A, Boneh D, et al. CCFI: Cryptographically Enforced Control Flow Integrity[C]. *The 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015: 941-951.
 - [40] Control Flow Guard. Microsoft. <https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard>. 2021.
 - [41] Castro M, Costa M, Harris T. Securing Software by Enforcing Data-Flow Integrity[C]. *The 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, 2006: 11.
 - [42] Costan V, Devadas S. Intel SGX Explained[EB/OL]. 2016: <https://eprint.iacr.org/2016/086.pdf>.
 - [43] Developments in the Arm A-Profile Architecture: Armv8.6-A. ARM Community. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-developments-armv8-6-a>. 2019.
 - [44] Arm Architecture Reference Manual Supplement. ARM. <https://developer.arm.com/documentation/ddi0568/latest>. 2019.
 - [45] Armv8.1-M Pointer Authentication and Branch Target Identification Extension. ARM Community. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension>. April. 2017.
 - [46] Learn the architecture: Providing protection for complex software. ARM Developer. <https://developer.arm.com/documentation/102433/0100/Stack-smashing-and-execution-permissions>. Jan. 2020.
 - [47] "Strong" Stack Protection for GCC. LWN.net. <https://lwn.net/Articles/584225/>. Feb. 2014.
 - [48] Implementing Stack Smashing Protection for Microcontrollers. Embedded Artistry. <https://embeddedartistry.com/blog/2020/05/18/implementing-stack-smashing-protection-for-microcontrollers-and-embedded-artistrys-libc/>. May. 2020.
 - [49] Pointer Authentication Support in LLVM Toolchain. ARM Developer. <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/llvm-toolchain/pointer-authentication>. 2021.
 - [50] A-Profile Architecture Support in LLVM Toolchain. ARM Developer. <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/llvm-toolchain/architecture-support>. Dec. 2021.
 - [51] iOS Kernel PAC, One Year Later. Google Project Zero. https://bazed.github.io/presentations/BlackHat-USA-2020-iOS_Kernel_PAC_One_Year_Later.pdf. 2020.
 - [52] Attacking iPhone XS Max. Wang T, Xu H. <https://i.blackhat.com/USA-19/Thursday/us-19-Wang-Attacking-iPhone-XS-Max.pdf>, 2019.
 - [53] Splitting atoms in XNU. Project Zero. <https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html>, 2019.
 - [54] Avanzi R. The QARMA Block Cipher Family [EB/OL]. 2016: <https://eprint.iacr.org/2016/444.pdf>.
 - [55] Pointer Authentication in AArch64 Linux. The Linux Kernel. <https://www.kernel.org/doc/html/latest/arm64/pointer-authentication.html>. July. 2017.
 - [56] Linux Kernel. Trovalds L, <https://github.com/torvalds/linux>.
 - [57] System Register Index by Functional Group. ARM Developer. <https://developer.arm.com/documentation/ddi0601/2022-03/Registers-by-Functional-Group?lang=en#Pointerauthentication>.
 - [58] Krstić I. Behind the Scenes of iOS and Mac Security: how the PAC is used. Blackhat, 2019.
 - [59] Zhou Z, Xie J. Hack Different: Pwning iOS 14 with Generation Z Bugz. Blackhat, 2021.
 - [60] Bovet D P, Cesati M. *Understanding the Linux kernel*[M]. 3rd ed. Beijing: O'Reilly, 2006.
 - [61] Apache MPM Prefork. APACHE http project. <https://httpd.apache.org/docs/2.4/mod/prefork.html>. 2021.
 - [62] Apache 的三种工作模式. 腾讯云. <https://cloud.tencent.com/developer/article/1667445>. 2020.
 - [63] A Study in PAC. Azad B. <https://bazed.github.io/presentations/MOSEC-2019-A-study-in-PAC.pdf>, 2019.
 - [64] HackPac: Hacking Pointer Authentication in iOS User Space, DEF

- CON 27. <https://doi.org/10.5446/48426>. Aug. 2019.
- [65] Remote iPhone Exploitation Part 3: From Memory Corruption to JavaScript and Back -- Gaining Code Execution. Samuel Groß. <https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html>. Jan. 2020.
- [66] A-Profile Architecture Support in GNU Toolchain. ARM Developer. <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/architecture-support>. 2021.
- [67] Pointer Authentication. LLVM Compiler Infrastructure. <https://llvm.org/docs/PointerAuth.html>. 2021.
- [68] RAP: RIP ROP. PaX Team. <https://pax.grsecurity.net/docs/PaX-Team-H2HC15-RAP-RIP-ROP.pdf>. Oct. 2015.
- [69] Lu K J, Hu H. Where does it Go? : Refining Indirect-Call Targets with Multi-Layer Type Analysis[C]. *The 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019: 1867-1881.
- [70] Schilling R, Nasahl P, Mangard S, et al, FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication[EB/OL]. 2017: ArXiv Preprint ArXiv:1901.09035.
- [71] Nasahl P, Schilling R, Mangard S. Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication[C]. *2021 IEEE International Symposium on Hardware Oriented Security and Trust*, 2022: 68-79.
- [72] Giannaris Y. Securing Operating Systems using Hardware-Enforced Compartmentalization[D]. Massachusetts Institute of Technology, 2021.
- [73] Memory Tagging Extension: Enhancing Memory Safety through Architecture. Steve Bannister. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety?_ga=2.73994258.1293509842.1644411524-555515125.1642119465. Aug. 2019.
- [74] Linux Kernel CVEs. All Indexed CVEs. <https://www.linuxkernel-cves.com/cves>. 2021.
- [75] Zhang J, Hou R, Song W, et al. RAGuard[J]. *ACM Transactions on Architecture and Code Optimization*, 2018, 15(4): 1-21.
- [76] Branch Target Identification. ARM Developer. <https://developer.arm.com/documentation/ddi0596/2021-09/Base-Instructions/BTI--Branch-Target-Identification--Sep.2019>.



张军 于 2018 年在中国科学院计算技术研究所获得博士学位。现任湖北文理学院副教授, 硕士生导师。研究领域为计算机体系结构, 系统安全。研究兴趣包括代码复用攻击防御。Email: zhangjun@hbuas.edu.cn



王兴宾 于 2021 年在中国科学院信息工程研究所获得博士学位。现任中国科学院信息工程研究所副研究员。研究领域为 AI 芯片安全架构设计、人工智能安全。Email: wangxingbin@iie.ac.cn



侯锐 于 2007 年在中国科学院计算技术研究所获得博士学位。现任中国科学院信息工程研究所研究员, 博士生导师。研究领域为计算机体系结构、数据中心服务器架构、处理器芯片安全、AI 芯片安全架构设计、人工智能安全。Email: hourui@iie.ac.cn



赵路坦 于 2021 年在中国科学院大学网络空间安全专业获得博士学位。现任中国科学院信息工程研究所副研究员。研究领域为计算机体系结构、处理器芯片安全。研究兴趣包括: 推测执行漏洞、分支预测器侧信道、缓存侧信道等处理器芯片安全漏洞防御。Email: zhaolutan@iie.ac.cn



李小馨 于 2022 年在中国科学院信息工程研究所获得博士学位。现在中国科学院信息工程研究所做博士后。研究领域为: 系统安全。研究兴趣包括内存安全保护, 操作系统安全。Email: lixiaoxin@iie.ac.cn



国冰磊 于 2020 年在新疆大学获得博士学位。现任湖北文理学院讲师。研究领域为数据库系统、云计算、节能计算与通信。Email: binglei@hbuas.edu.cn