计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(11):11−19 [doi: 10.15888/j.cnki.csa.008351]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

PEC-V: 基于 RISC-V 协处理器的内存溢出防御机制①

张雨昕[1,2], 芮志清[1], 李威威[3], 张　画[1,4], 罗天悦[1], 吴敬征[1]

[1](中国科学院 软件研究所 智能软件研究中心, 北京 100190)
[2](伊利诺伊大学香槟分校 The Grainger College of Engineering, Urbana-Champaign 61820)
[3](中国科学院 软件研究所 PLCT 实验室, 北京 100190)
[4](北京航空航天大学 高等理工学院, 北京 100191)
通讯作者: 吴敬征, E-mail: jingzheng08@iscas.ac.cn

摘　要: 内存溢出攻击是计算机系统中历史悠久且依旧广泛存在的攻击手段, 而指针加密技术可以有效阻止此攻击. 通过软件手段实现这一技术的方式将导致程序运行效率的显著降低并且产生额外的内存开销. 所以本文基于 RocketChip 的 RoCC (Rocket Custom Coprocessor) 接口实现一个加解密指针的协处理器 PEC-V. 其通过 RISC-V 的自定义指令控制协处理器加解密返回地址和函数指针等值达到阻止溢出攻击的目的. PEC-V 主要使用 PUF (Physical Unclonable Function) 来避免在内存中储存加密指针的键值, 所以此机制在保证了加密键值的随机性的同时也减少了访问内存的次数. 实验结果显示, PEC-V 能够有效防御各类缓冲区溢出攻击, 且程序平均运行效率仅下降 3%, 相对既往方案显著提高了性能.

关键词: 溢出攻击; 指针加密; RISC-V; RocketChip; PUF; PEC-V

引用格式: 张雨昕,芮志清,李威威,张画,罗天悦,吴敬征.PEC-V: 基于 RISC-V 协处理器的内存溢出防御机制.计算机系统应用,2021,30(11):11–19. http://www.c-s-a.org.cn/1003-3254/8351.html

PEC-V: Memory Overflow Defense Mechanism Based on RISC-V Coprocessor

ZHANG Yu-Xin[1,2], RUI Zhi-Qing[1], LI Wei-Wei[3], ZHANG Hua[1,4], LUO Tian-Yue[1], WU Jing-Zheng[1]

[1](Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](The Grainger College of Engineering, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL 61820, USA)
[3](PLCT Lab, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[4](SHENYUAN Honors College, Beihang University, Beijing 100191, China)

Abstract: In computer systems, the memory overflow attack is a long-existing security problem and is still common nowadays, which can be effectively hindered by pointer encryption. Nevertheless, the implementation of the technique by software significantly lowers the program running efficiency and leads to additional memory overhead. In this study, we develop an encrypted/decrypted pointer coprocessor PEC-V based on the Rocket Custom Coprocessor (RoCC) interface of RocketChip. The overflow attack can be prevented through the control of encryption/decryption of the return address and function pointer by the coprocessor under the user-defined instruction of RISC-V. PEC-V mainly depends on Physical Unclonable Function (PUF) to avoid storing the key value of the encrypted pointer in memory. Thus, this mechanism not only ensures the randomness of the key value, but also reduces the times of accessing memory. The experimental results

① 基金项目: 中国科学院战略性先导科技专项 (C 类)(XDC05040100); 国家自然科学基金 (61772507); 2020 年工业互联网创新发展工程 (TC200H030)
Foundation item: Chinese Academy of Sciences Strategic Priority Program (Category C)(XDC05040100); National Natural Science Foundation of China (61772507); Year 2020, Industrial Internet Innovation Development Project (TC200H030)
本文由“RISC-V 技术与生态”专题特约编辑武延军研究员、李玲研究员以及邢明杰高级工程师推荐.
收稿时间: 2021-04-29; 修改时间: 2021-05-21, 2021-06-08; 采用时间: 2021-06-15; csa 在线出版时间: 2021-10-22

show that PEC-V is defensive against various buffer overflow attacks while the program running efficiency is only reduced by approximately 3% on average, which is better than previous mechanisms.

Key words: overflow attack; pointer encryption; RISC-V; RocketChip; Physical Unclonable Function (PUF); PEC-V

1 引言

缓冲区溢出攻击是计算机系统安全领域的重要攻击手段之一. 攻击者通过对未做边界检查的缓冲区写入恶意数据, 达到控制程序甚至获取整个系统操作特权的目的. 然而因为以 C 语言为代表的被广泛使用的编程语言并不具有强制进行缓冲区边界检查的功能, 所以众多使用此类语言的系统中存在着诸多溢出漏洞易被攻击者利用. 如今, 每年此类漏洞被检测到的数量在各类关于计算机安全的漏洞依旧中占据前列[1]. 虽然随着技术的进步与发展, 操作系统、应用软件和硬件层面都发展出许多预防此类攻击的安全机制, 例如内存地址空间随机化 (ASLR)[2], StackGuard[3], 数据不可执行 (DEP)[4] 等. 与此同时, 攻击的方式也随之复杂, 发展出各类变种, 例如 Return-to-libc Attack (Ret2libc) 和 Sigreturn Oriented Programming (SOP) 等[5,6], 以此绕过各类安全机制.

指针加密是一种通过保护指令地址的溢出攻击预防机制. 溢出攻击的主要原理是注入恶意数据, 修改指令寄存器指向的地址使控制流跳转到攻击者构造的代码中, 最终劫持程序或者获取更高的系统权限. 而指针加密的主要机制是将程序返回地址和函数指针等重要数据加密后再存储在内存中, 而在需要将此类数据加载到指令寄存器中时再解密使指令寄存器指向正确的目的地址. 通过此种方法, 指令的地址被加密隐藏, 即使攻击者通过各种方式改变了指令寄存器中的值, 也无法预测指令指针实际指向的位置, 更无法使指针指向恶意构造的代码处. 因此这一安全预防机制可以有效阻止各类溢出攻击.

PointGuard 是一种通过软件手段实现指针加密机制的技术[7], 此机制将密钥存储在内存中某一区域, 在加解密时取出密钥值将其与指针的值做异或运算. 但是, 此方式在加解密时均需要访问内存, 会增加时间开销以及内存占用的空间开销. 而且对于能够读取数据的攻击, 易泄露加密键值从而导致预防机制失效. 针对这些问题, 本文提出基于 RISC-V 自定义指令设计的指针加密协处理器 PEC-V (Pointer Enryption Coprocessor on RISC-V), 通过硬件手段实现这一机制.

RISC-V 作为一个新型开源指令集架构, 同样易受到缓冲区溢出攻击. Jaloyan 等在论文中阐明了此类攻击在 RISC-V 架构上的可行性, 并利用内核中的代码片段成功实现了 ROP (Return Oriented Programming) 攻击并绕过多种安全防御机制[5]. 因此, 针对 RISC-V 的安全性研究十分必要.

RISC-V 具有的显著特性之一便是开放的可扩展性, RocketChip 是一款基于 RISC-V 的 SoC (System-on-a-Chip) 生成器, 它基于 RISC-V 预留的自定义指令设计的 RoCC (Rocket Custom Coprocessor) 接口可以方便的扩展协处理器. 而协处理器 PEC-V 的内部设计主要通过使用 PUF (Physical Unclonable Function)[8] 加解密数据. PUF 对不同激励输入做出唯一且随机的响应, 但对于同样的输入总是输出固定值. 因此本文将被加密数据地址作为激励输入, 将 PUF 响应作为密钥加密数据以此减少访问内存的时间开销, 减少内存占用, 同时保障了安全性.

2 背景

此节主要介绍针对溢出攻击的其他安全预防机制和这些机制存在的问题, 以及介绍 RISC-V 自定义指令和 RoCC 接口的细节和选择这一架构的原因.

2.1 预防缓冲区溢出攻击的机制介绍

目前, 针对缓冲区溢出攻击的研究十分广泛, 研究者们提出了许多预防机制. 首先, ASLR[2] 和 DEP[4] 便是两种已经被广泛使用的预防机制. 其中, ASLR 机制在加载程序地址空间时, 通过在栈、堆、代码区和动态链接库等内存段的起始地址前加入一段随机偏移量达到地址随机化的目的. 此种方法可以加大攻击者定位漏洞跳转到目的地址的难度. 而 DEP 机制是在内存页添加额外标识字节表示此页是否可执行, 通过操作系统的控制区分内存为可执行段和不可执行段, 因此写在缓冲区中被攻击者恶意注入的代码将不再可以被执行, 从而阻止攻击. 然而面对更加具有针对性和复杂的攻击手段, 这两种预防机制变得十分有限. 对于 ASLR

机制, 攻击者通过在注入代码中添加 nop 指令的方式使程序控制流最终仍能被攻击者掌握. 并且在 32 位的系统架构上, ASLR存在内存地址空间小的时候随机化不足的问题[9]. 而对于 DEP 机制, 攻击者可以利用动态链接库或者程序中已经存在的代码片段构造栈中的返回地址和数据实现 Ret2libc 攻击[10].

而后 Stack Canary[3] 作为一种更加有效的预防机制被提出, 并被实际应用. Stack Canary 可以通过将 NULL 值插在缓冲区的末尾, 以此识别缓冲区中数据是否越界, 当发现 NULL 值被覆盖时说明缓冲区溢出, 则立刻终止程序. 然而如果使用 NULL 值插在缓冲区末尾, 会有一些系统函数允许写入 NULL 值导致溢出攻击依旧可行. 此问题的一个解决方式是使用随机数代替 NULL 值. 然而如果使用随机数, 那么此数值需要对攻击者保密, 如果使用软件实现此机制, 随机数的种子值需要存储在 TCB 中, 而这将成为一个易受攻击的薄弱环节. De 等[11] 在论文中使用 PUF 在 RISC-V 架构上设计了一个 Canary Engine. 然而此种硬件实现方式依旧不能避免一些攻击. Stack Canary 机制只能检测到改变了缓冲区和返回地址之间所有数据的攻击, 对于通过指针直接覆盖返回值的攻击, Stack Canary 无法避免, 例如 printf 格式化字符串溢出等. 此外, 对于堆上的溢出攻击, Stack Canary 同样无法阻止[12].

近年来另一种防御机制控制流完整性 (CFI)[13] 逐渐成为研究的主流, 针对此方面的研究主要分为粗粒度控制流完整性[14] 和细粒度控制流完整性[15]. 这些机制在编译阶段构建程序的控制流图并通过某种方案在程序执行时使其只能按照流图中的边跳转. 然而这些方案大部分并未投入实际使用, 它们依旧各自存在一定的问题. 由于静态分析并不能完全确定程序的控制流图, 例如函数指针指向的函数无法在编译阶段确定, 因此粗粒度的控制流完整性方案无法精准定位程序跳转的位置, 这导致此机制依旧存在漏洞可被攻击者利用, 而细粒度方案则会增添大量空间和时间的开销[16,17].

类似于 CFI 机制, 指针加密也通过保护程序控制流达到阻止攻击的目的. Cowan 等[7] 提出的 PointGuard 技术便是通过加解密指针的方式阻止溢出攻击. 此技术为了保障加密密钥的随机性, 每当开启一个新进程时都需要重置此密钥 (REF), 而此值将被存在进程的 TCB 中. 因此 TCB 可能会成为整个系统中易受攻击的环节, 对于有可能存在数据泄露或者存在可以修改 TCB 值的漏洞的程序, 攻击者可以破解或绕过此机制. 此外由于每一次加解密返回地址或者其他可改变指令寄存器值的数据时, PointGuard 都需要从内存中取出密钥的值, 将其与返回地址或者函数指针执行异或运算[18]. 从以上描述中可以看出在存储和使用此类数据时, PointGuard 都需要额外增添 1 条访存指令, 这将影响到程序的运行效率. 综上, 本文受 Canary Engine[11] 的启发, 提出基于 RISC-V 自定义指令设计的指针加密协处理器 PEC-V, 通过硬件手段实现这一机制, 可在增强安全性的同时有效解决运行效率低的问题.

2.2 RISC-V 自定义指令和 RoCC 接口介绍

RISC-V 预留了众多未被定义的编码空间[19], 其中, 为了便于非标准化的扩展, RISC-V 预留了 4 种自定义指令 custom0/1/2/3, 使用者可以将这 4 种指令扩展成为协处理器的控制指令. RocketChip 是一个开源的 RISC-V 的 SoC 生成器. RocketChip的代码实现了 RoCC 接口, 此接口支持 custom0/1/2/3指令, 可用于连接用户自定义的协处理器[20].

ROCC 接口支持的 custom 指令的格式和字段如图 1 所示, 其中 rd 字段为目的寄存器, rs1 和 rs2 为源寄存器, xs1 和 xs2 表示协处理器是否读取源寄存器中的值, xd 表示是否写回目的寄存器, opcode 为 custom 指令的操作码, funct7 的值可由用户自行扩展以控制协处理器完成不同操作. 本文使用 custom0 指令, 编码 funct7 字段使 custom0 扩展成为 4 条不同的协处理器控制指令.

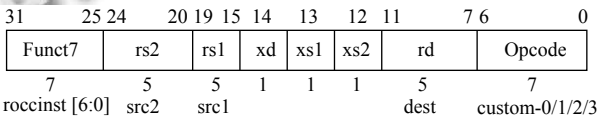


图 1 自定义指令格式

RoCC 接口的架构[21] 如图 2 所示, 其中 cmd 是散开的接口, 包括两个源寄存器的值和输入指令, 主处理器通过 cmd 将命令和数据传入协处理器中; resp 将协处理器中的值传递回 cmd 中 rd 字段指明的目的寄存器中; mem.req 和 mem.resp 分别用于协处理器发送内存读写请求和主处理器返回请求结果; busy 表示协处理器是否能继续接收指令, busy 值为真时表明协处理器处于堵塞状态.

从上文介绍中可以看出, RISC-V 指令集可扩展的

特性以及 RocketChip 已经实现的 RoCC 接口可以允许本研究方便地接入自定义的协处理器, 实现指针加密的机制. 因此本文选择使用此指令架构和接口设计和运行试验.

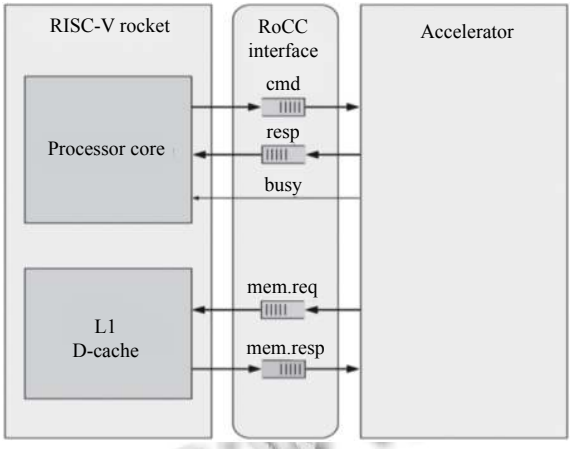


图 2 RoCC 接口架构细节

3 PEC-V 的内部设计和指针加密实现机制

3.1 PEC-V 内部构造

本文设计的用于指针加密的协处理器内部构造如图 3 所示. 其中, PUF 是一种硬件结构, 其利用硬件某些方面不可预测的随机性, 对某一输入可产生固定的随机响应. 一个输入和对应的响应被定义为一个 Challenge-Response-Pair (CRP), PUF 依据 CRPs 数量的多少被分为强 PUF 和弱 PUF, 其中, 强 PUF 的 CRPs 数量可以达到指数级[8], 足以满足指针加密的需要. 同时 PUF 成本低, 面积小, 耗能低, 且能一定程度上预防常见的对 IoT 的侧信道攻击[22]. 本文中使用的 PUF 为 SRAM-PUF. 图中, 每当 rs1 传入输入时, PUF 产生唯一对应的随机响应并传到异或门中.

TRNG 是真随机数发生器, 它利用机器产生的噪声例如热力学噪声、光电效应和量子现象等产生真正的随机数. 图中的 TRNG 在产生随机数后将其传入 secret register 中保存.

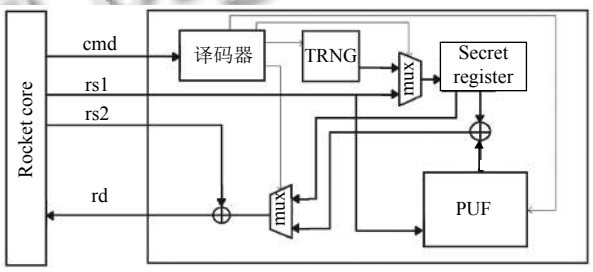


图 3 PEC-V 的电路逻辑图

3.2 自定义指令

本文对 RISC-V 自定义指令 custom0 的 funct7 字段进行编码, 在 custom0 的基础上扩展了 4 条协处理器控制指令. 协处理器中的译码器通过 funct7 字段的不同识别不同的指令. 4 条指令如表 1 所示.

表 1 增添的自定义指令

RoCC指令	Funct7	rs2	rsl	Funct3	rd	Opcode
指令1	0000000	—	—	000	—	0001011
指令2	0000001	—	Source register	010	—	0001011
指令3	0000010	—	—	100	Dest register	0001011
指令4	0000011	Source register2	Source register1	111	Dest register	0001011

指令 1 无输入和输出, 它的作用是命令协处理器中的 TRNG 产生 1 个真随机数并将值保存到 secret register 中. 每当建立 1 个新进程的时候, 操作系统都会发送指令 1 到协处理器, 这样每一个进程初始化后都会获得一个真随机数值.

指令 2 的作用是将协处理器 secret register 中存储的值取出存入 rd 字段指定的寄存器中. 每当 CPU 进行上下文切换将正在运行的进程切换出去的时候, 操作系统负责发送指令 2 到协处理器中取出此进程的随机数值并存入其 PCB 中.

指令 3 的作用是将 rs1 字段中指定的寄存器中的值传入协处理器的 secret register 中, 当 CPU 将某一进程切换进入的时候, 操作系统负责将 PCB 中存储的随机数值装载到常寄存器中并且发送指令 3 到协处理器, 协处理器再将随机值存入 secret register 中.

指令 4 是唯一由用户进程使用的指令, 其作用是将 rs1 中的值作为输入传入 PUF 中, PUF 产生的响应与 secret register 中的值进行异或后再与 rs2 中的值进

行异或, 最后传入 rd 中. 对于进程中的返回地址, 函数指针等需要加密的数据, 进程负责将需要加密的数据以及数据的存储地址分别传入 rs1 和 rs2 中并且发送指令 4 给协处理器, 等到协处理器返回值存入 rd, 进程再将加密后的数据存入内存. 而当进程需要使用此加密数据时, 同样将数据和地址存入 rs1 和 rs2 中, 发送指令 4 到协处理器, 余下过程与加密相同, 最后 rd 寄存器中即为解密后的数据.

通过上述的机制, 对于每一个进程都有唯一的真随机值, 这样不同的进程中即使是相同地址的数据, 加密后的密文将不会相同, 而对于同一进程, 在同一虚拟地址空间的数据, 由于地址不同, 地址作为 PUF 的输入得到的固定的随机响应也不相同, 所以 PUF 起到了在同一虚拟内存地址空间中随机化的目的. 综上, 此设计指针加密的键值的随机性更高, 所以此机制的安全性具有足够的保障.

3.3 汇编指令的修改

在 RISC-V 指令集增加自定义指令后, 为达到指针加密的目标, 还需在程序中插入自定义汇编指令. 本文首先将程序编译为汇编程序, 接着使用程序扫描编译后的汇编指令, 识别以下 4 处需要对指令做出修改的地方, 分别为: 函数指针赋值前, 函数指针解引用前, 被调用过程保存返回地址前和被调用过程返回之前. 之后自动在此些位置插入计算地址的指令和指令 4 进行加解密, 修改后的汇编指令再转换为二进制文件执行. 除了此插入方法之外, 还可通过使用 Clang/LLVM 编译器实现插入指令方式的方案. 首先将程序编译为 LLVM 的中间代码, 增加 pass 插入加解密的指令, 最后转换为可执行文件. 相对于本文的方法, 修改编译器的方案更为完善, 但为快速验证 PEC-V 的安全性, 本文未采取复杂的修改编译器的方案.

对于程序需要做的具体的改变本文通过图 4 中简单的示例程序进行说明.

图 4(a) 中的程序是使用函数指针的一个简单示例, 图 4(b) 中是针对返回地址加密的简单示例. 首先将程序编译成中间指令, 扫描定位到程序赋值和使用函数指针的位置以及调用过程时返回地址压栈以及出栈的位置, 接下来在这些位置插入需要的汇编指令进行加密和解密操作. 图中黑色方框中的汇编指令即为后期添加的加密和解密操作, custom_rd_rs1_r2 即为新增的指令 4.

对于函数指针, 在地址存入指针之前, 先将指针的地址通过指令 *addi a*4, *s*0, –24 放入常用寄存器 *a*4 中, *a*5 中本身存着函数指针的值, 接着 *a*4、*a*5 作为指令 4 中的 rs1、rs2 传入协处理器中, 协处理器进行加密后的返回值返回到 *a*5, 接着 *a*5 中加密后的值被存入指针中. 而当程序需要使用函数指针时, 同样将指针的值和指针的地址传入协处理器进行解密.

```
void printhello() {
    printf("hello\n");
}

int main() {
    void (*fun1)();
    fun1 = &printhello;
    (*fun1)();
    return 0;
}
```

```
<printhello>:
addi    sp,sp,-16
...
...
ret

<main>:
addi    sp,sp,-32
sd      ra,24(sp)
sd      s0,16(sp)
addi    s0,sp,32
lui     a5,0x10
addi    a5,a5,348 # 1015c <printhello>
#addi    a4,s0,-24
#custom0_rd_rs1_rs2 a5,a4,a5
sd      a5,-24(s0)
ld      a5,-24(s0)
#addi    a4,s0,-24
#custom0_rd_rs1_rs2 a5,a4,a5
jalr    a5
li      a5,0
mv      a0,a5
ld      ra,24(sp)
ld      s0,16(sp)
addi    sp,sp,32
ret
```

(a) 对函数指针的处理

```
void fun(){
    char buff[10];
    char str[20];
    scanf("%s",str);
    strcpy(buff, str);
}

int main() {
    fun();
    return 0;
}
```

```
<fun>:
addi    sp,sp,-64
sd      ra,56(sp)
#addi    a4,s0,-56
#custom_rd_rs1_rs2 ra,a4,ra
sd      s0,48(sp)
addi    s0,sp,64
addi    a5,s0,-56
mv      a1,a5
lui     a5,0x1f
addi    a0,a5,-176 # 1ef50 <__clzdi2+0x32>
jal     ra,102e6 <scanf>
addi    a4,s0,-56
addi    a5,s0,-32
mv      a1,a4
mv      a0,a5
jal     ra,10334 <strcpy>
nop
ld      ra,56(sp)
#addi    a4,s0,-56
#custom_rd_rs1_rs2 ra,a4,ra
ld      s0,48(sp)
addi    sp,sp,64
ret

<main>:
...
jal     ra,10160 <fun>
...
```

(b) 对返回地址的处理

图 4 对函数指针和返回地址的处理

而针对返回地址, *ra* 寄存器中存储的是返回地址, 图中 *jalra, offset* 指令执行的过程是:

$ra = pc + 2, pc = pc + offset;$

所以, 被调用函数保存返回地址之前, 通过 *add a*4, *s*0, –56 获得返回地址存在栈上的位置, 将 *ra* 和 *a*4 中的值都传入协处理器获得加密的地址. 当被调用函数返回时, 在返回地址出栈后, 插入同样指令进行解密.

从以上的例子中可以看出此机制的设计不会增加额外的内存访问, 因此减少了对程序运行效率的影响. 而从安全方面来看, 对于一个程序 *p*1 中一个需要加密的数据 *d*1, 它的密文是同时和 TRNG 产生的真随机数 r_{p1} 和 PUF 对其地址 *a*1 产生的响应 s_{a1} 异或的结果, 即密文 $m1 = d1 \oplus r_{p1} \oplus s_{a1}$. 所以一个程序中的某个需要加密的数据的键值将由 r_{p1} 和 s_{a1} 共同决定, 这在 PointGuard 的基础上更增添了安全性. 而当攻击者试图修改地址执行恶意代码时, 如果通过穷举的方式, 对于 64 位系统将有 2^{64} 种可能的加密键值. 当遭受溢出攻击, 程序指令寄存器的值被修改后, 虽然程序不会识别到值已经被改变, 但是解密后的地址会跳转到任意位置并造成程序崩溃, 阻止了攻击者试图控制程序的意图.

3.4 总结

根据以上的原理介绍可以看出, 此机制主要防御的攻击模式是, 利用缓冲区溢出改变指令寄存器指向的代码, 跳转注入代码或者库中以 ret 指令结尾的 gadgets, 最终实现攻击者想要的操作. 此机制实现安全性的前提是攻击者无法访问只有特权级进程才能控制的 PCB 区域, 无法获取 PCB 中存储的进程唯一的真随机键值. 在此前提下即使攻击者可在同一机器上运行自己的程序, 读取加密键值, 也无法从此数据中获取任何其他进程加密键值的信息, 而对于攻击者试图攻击的进程, 即使有几处内存位置的加密键值被读溢出漏洞所暴露, 其他位置的加密键值仍然无法被攻击者获取. 由于一般攻击都需要多处的代码段链接在一起执行, 因此, 攻击者能成功的概率将大大降低. 而如果通过暴力破解的方式, 需要的次数取决于 PUF 产生响应的字节数, 对于可产生 32 位响应的 PUF, 破解需要的时间将是指数级的.

4 实验与分析

本研究使用 Chisel 电路语言[23] 实现了 PEC-V 的逻辑电路, 接入 RocketChip 的 RoCC 接口, 接着使用 RocketChip 已实现的 Cycle-accurate C++ Emulator 进行实验. 实验的主要目的有二:

(1) 验证此机制的安全性.

(2) 测试此机制对程序运行效率的影响.

4.1 机制的安全性

测试程序. 首先, 本文设计了缓冲区溢出攻击的测试程序, 简单的程序示例如图 5 所示. 程序中 strcpy 函数将 str 中的值复制到 buff 中的时候, 由于没有边界检查, 栈中的返回值将被覆盖, 当函数执行完后指令指针指向到了 if 代码块之中, 然而 if 中指令在程序正常执行的情况下并不会被执行.

```
#include<stdio.h>
#include<string.h>
void fun2() {
        char buff[4];
        char str[24] =
        "\x41\x41\x41\x41\x41\x41\x41\x41
        \x50\x96\xee\x0f\x00\x00\x00\x00
        \xb0\x01\x01\x00\x00\x00\x00\x00";
        strcpy(buff,str);
}
int main() {
        int flag = 1;
        fun2();
        if (flag == 0) {
                printf("hello\n");
        }
        return 0;
}
```

图 5 简单的攻击代码

实验结果与分析. 当示例程序在仿真器上模拟运行的时候攻击者成功使得 if 中本不应被执行的代码执行, 而当 PEC-V 被使用时, 程序跳转到未知位置造成程序崩溃, 阻止了攻击者的目的.

除了自行设计的溢出攻击程序之外, 对于此机制的安全性, 进一步根据 Wilander 等设计的缓冲区溢出的测试集进行分析. Wilander 等罗列出了 20 余种针对缓冲区溢出的攻击[24], 主要分为 4 类, 分别为针对 return address, function point, old base point 和 longjmp buffer 的攻击. 因为本文的机制针对的是 return address 和 function point 的加解密, 所以在此处只讨论关于前两者的攻击. 攻击的方式又可分为栈溢出和堆溢出攻击, 攻击方式如表 2 所示.

因为此些攻击过程中都需要通过覆盖返回地址或函数指针指向目的地址, 然而由于此类数据已被加密, 在攻击者无法获得加密密钥的情况下, 覆盖后的值解

密后将指向随机地址造成程序崩溃, 因此攻击者的目的将无法实现.

表 2 Wilander 等[24] 研究中的攻击种类

溢出类型	攻击方式	攻击种类
直接溢出修改数据	栈溢出	针对返回地址
		针对作为局部变量的函数指针
		针对作为参数的函数指针
	堆溢出	针对作为局部变量的函数指针
间接溢出通过指针修改数据	栈溢出	针对返回地址
		针对作为局部变量的函数指针
		针对作为参数的函数指针
	堆溢出	针对返回地址
		针对作为局部变量的函数指针
		针对作为参数的函数指针

4.2 机制的效率

(1) 测试程序. 本实验首先使用 Juliet Test Suite[25] 这一常见漏洞测试集进行效率测试. 测试集中存在着几种针对缓冲区溢出漏洞的分类 (cwe121_stack_based_buffer_overflow, cwe122_heap_based_buffer_overflow 和 cwe124_buffer_underwrite, cwe680_integer_overflow_to_buffer_overflow), 选取这 4 种分类中的测试程序 *t*1, *t*2, *t*3 和 *t*4, 进行针对返回地址的防御实验. 其中, *t*1 为向栈上缓冲区复制数据越界, *t*2 为向堆上缓冲区复制数越界, *t*3 为数组下标越界, *t*4 为整数溢出导致数组越界. 由于 Juliet 测试集中缺少针对函数指针的攻击示例, 对于函数指针的效率测试使用 Wilander testbed 中的测试集, 包括栈上溢出修改函数指针 *w*1 和堆上溢出修改函数指针 *w*2 和 *w*3. 测试过程首先将 *t*1–*t*4 和 *w*1–*w*3 编译成原始的没有加密机制保护的版本, 在此基础上按照前文介绍的方式修改汇编代码, 增加使用 PEC-V 的防护机制. 接着在 C++仿真器上本研究使用 RISC-V 的 ProxyKernel 运行测试程序, 通过在测试程序开始前和结束后读取时钟寄存器的值来获得精确的运行周期.

除了使用已存在的测试集之外, 为了进一步探究此防御机制的效率问题, 本文对于栈上溢出修改函数指针和返回地址的测试程序进行修改, 增加对程序中嵌套调用函数和多次使用函数指针的情况的测试. test1 测试针对返回地址的加解密的效率, test1 中子函数迭代调用的次数从 1 到 6 增加. test2 测试针对函数指针保护的效率, test2 中使用的函数指针从 1 到 6 不断增多.

(2) 实验结果与分析. 对 Juliet Test Suite 和 Wilander Testbed 中的程序进行实验后结果如图 6 所示, 从中可以看出使用此机制对于常见的可利用漏洞进行防御, 程序效率的下降未超过 4%, 未造成严重影响. 而在对 test1 和 test2 加密前后两个版本测试后, 实验的运行结果分别如图 7(a) 和图 7(b) 所示. 从图中可以看出 test1 和 test2 在受到协处理器的加密机制保护后, 运行效率的下降在 1%–3% 之间.

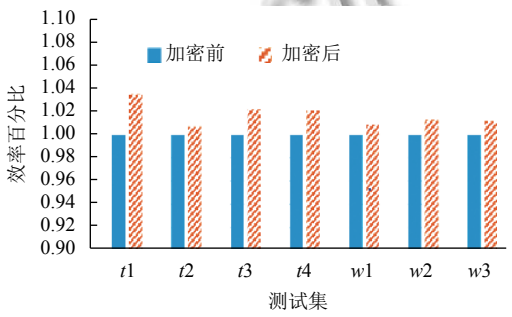
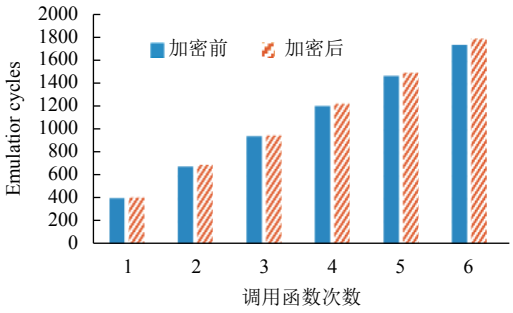
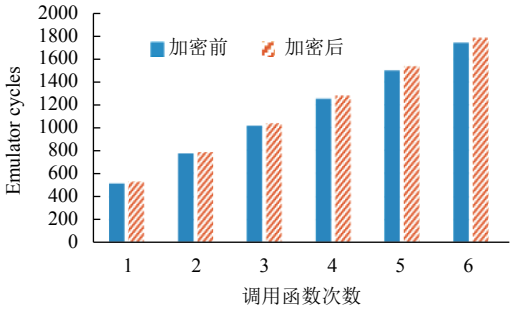


图 6 测试集运行效率



(a) 返回地址加密的性能



(b) 函数指针加密的性能比较

图 7 性能比较

从数据中可以看出, 对于一般测试程序而言, 硬件实施指针加密的方式并不会对程序的运行效率产生显著影响.

5 结论与展望

本文旨在介绍一种通过硬件方式实现的缓冲区溢出攻击保护机制. 通过 RoCC 接口接入指针加密协处理器, 加密返回地址和函数指针使得攻击者定位目标代码的难度大大增加, 同时使得对程序的运行效率的影响降到最小, 而 PUF 和 TRNG 在协处理器中的使用使得加密键值的随机性得到保障, 并且 PUF 的特性可以加大物理手段获取随机键值的难度. 实验结果表明, 未使用 PEC-V 机制时针对对 RISC-V 的溢出攻击能够成功, 然而增加这一防护机制后, 攻击的难度增加, 并且未对程序运行效率造成显著影响, 优于以往的实现方案.

此外本文主要针对返回地址和函数指针进行讨论, 而常见的溢出攻击针对的目标还有帧指针等, 此机制略作修改便可同样运用于帧指针值的保护. 而此预防机制存在一定的不足之处, 对于不需要返回到特定地址, 直接通过溢出改写重要变量或者泄露重要信息的攻击, 此机制暂时无法防御, 这方面的防御手段还需进一步研究.

参考文献

1 Vulnerabilities by date. https://www.cvedetails.com/vulnerabilities-by-types.php. [2021-04-09].

2 PaX-Team: PaX address space layout randomization (2003). http://pax.grsecurity.net/docs/aslr.txt. [2021-04-09].

3 Cowan C, Pu C, Maier D, *et al*. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. Proceedings of the 7th Conference on USENIX Security Symposium. Berkeley: USENIX Association, 1998. 5.

4 Ven A, Molnar I. Exec shield (2004). https://www.redhat.com/f/pdf/rhel/WHP0006USExecshield.pdf. [2021-04-09].

5 Jaloyan GA, Markantonakis K, Akram RN, *et al*. Return-oriented programming on RISC-V. Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS'20). Taipei: Association for Computing Machinery, 2020. 471–480.

6 Bosman E, Bos H. Framing signals—A return to portable shellcode. Proceedings of the 2014 IEEE Symposium on Security and Privacy. Berkeley: IEEE, 2014. 243–258.

7 Crispin C, Steve B, John J, Perry W. PointGuard: Protecting pointers from buffer overflow vulnerabilities. 12th USENIX Security Symposium (USENIX Security 03). Washington DC: USENIX, 2003, 8. 91–104.

8 Chang CH, Zheng Y, Zhang L. A retrospective and a look forward: Fifteen years of physical unclonable function advancement. IEEE Circuits and Systems Magazine, 2017, 17(3): 32–62. [doi: 10.1109/MCAS.2017.2713305]

9 Gisbert HM, Ripoli I. On the effectiveness of full-ASLR on 64-bit Linux. In-depth Security Conference 2014 (DeepSec). Vienna, 2014. 9.

10 Gisbert HM, Ripoll I. On the effectiveness of NX, SSP, RenewSSP, and ASLR against stack buffer overflows. 2014 IEEE 13th International Symposium on Network Computing and Applications. Cambridge: IEEE, 2014. 145–152. [doi: 10.1109/NCA.2014.28]

11 De A, Basu A, Ghosh S, *et al*. Hardware assisted buffer protection mechanisms for embedded RISC-V. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020, 39(12): 4453–4465. [doi: 10.1109/TCAD.2020.2984407]

12 潘亦, 吴春梅, 武港山. 防止缓冲区溢出攻击的增强编译技术分析. 计算机科学, 2005, 32(3): 156–158. [doi: 10.3969/j.issn.1002-137X.2005.03.041]

13 Abadi M, Budiu M, Erlingsson Ú, *et al*. Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security, 2009, 13(1): 4. [doi: 10.1145/1609956.1609960]

14 Davi L, Sadeghi AR, Lehmann D, *et al*. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. Proceedings of the 23rd USENIX Conference on Security Symposium. San Diego: USENIX, 2014. 401–416.

15 Payer M, Barresi A, Gross TR. Fine-grained control-flow integrity through binary hardening. In: Almgren M, Gulisano V, Maggi F, eds. Detection of Intrusions and Malware, and Vulnerability Assessment. Cham: Springer, 2015. 144–164. [doi: 10.1007/978-3-319-20550-2_8]

16 Carlini N, Wagner D. ROP is still dangerous: Breaking modern defenses. Proceedings of the 23rd USENIX Security Symposium. San Diego: USENIX, 2014. 385–399.

17 Zhang C, Wei T, Cheng ZF, *et al*. Practical control flow integrity and randomization for binary executables. 2013 IEEE Symposium on Security and Privacy. Berkeley: IEEE, 2013. 559–573. [doi: 10.1109/SP.2013.44]

18 Tuck N, Calder B, Varghese G. Hardware and binary modification support for code pointer protection from buffer overflow. 37th International Symposium on Microarchitecture (MICRO-37'04). Portland: IEEE, 2004. 209–220. [doi: 10.1109/MICRO.2004.20]

19 Waterman A, Lee Y, Patterson DA, *et al*. The RISC-V instruction set manual, Volume I: User-level ISA. http://www.icsi.berkeley.edu/pubs/arch/EECS-2011-62.pdf. [2021-04-15].

20 RISC-V International. RISC-V “RocketChip” tutorial. https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-tutorial-bootcamp-jan2015.pdf. [2021-04-15].

21 Martin J. RISC-V, Rocket, and RoCC. https://inst.eecs.berkeley.edu/~cs250/sp17/disc/lab2-disc.pdf. [2021-04-15].

22 Potkonjak M, Goudar V. Public physical unclonable functions. Proceedings of the IEEE, 2014, 102(8): 1142–1156. [doi: 10.1109/JPROC.2014.2331553]

23 Bachrach J, Vo H, Richards B, *et al*. Chisel: Constructing hardware in a Scala embedded language. Proceedings of the 49th Annual Design Automation Conference. San Francisco: Association for Computing Machinery, 2012. 1216–1225. [doi: 10.1145/2228360.2228584]

24 Wilander J, Kamkar M. A comparison of publicly available tools for dynamic buffer overflow prevention. Proceedings of the 10th Network and Distributed System Security Symposium. San Diego: Internet Society, 2003. 149–493.

25 Boland T, Black PE. Juliet 1.1 C/C++ and Java Test Suite. Computer, 2012, 45(10): 88 –90. [doi: 10.1109/MC.2012.345]