

DOI:10.3969/j.issn.1671-0673. 2026. 01. 016

面向数据攻击的 ARM MTE 自动防御实现

梁思远¹, 刘铁铭¹, 张媛媛²

(1.信息工程大学,河南 郑州 450001; 2.32738 部队,河南 郑州 450001)

摘要:针对基于软件的防护机制面临性能开销偏高且难以防御部分高级攻击的问题,提出一种使用 ARM 内存标记扩展(MTE)来增强内存安全性的方法。该方法使用底层虚拟机(LLVM)编译器进行堆栈调用分析,并使用 MTE 机制进行标签管理。通过使用基准测试和 50 个易受攻击的程序进行实验评估,该方法显示出针对数据导向攻击显著的防御能力,运行时开销平均为 52.8%,文件大小开销为 17.7%。该研究在对抗数据导向攻击方面取得了进展,在安全性和性能效率之间取得了较好的平衡。

关键词:内存错误;数据导向攻击;ARM 内存标记扩展;自动防御;底层虚拟机

中图分类号:TP309.2

文献标识码:A

文章编号:1671-0673(2026)01-0120-07

Automatic Defense Implementation of ARM MTE Against Data Attacks

LIANG Siyuan¹, LIU Tieming¹, ZHANG Yuanyuan²

(1.Information Engineering University, Zhengzhou 450001, China; 2.Unit 32738, Zhengzhou 450001, China)

Abstract: To address the challenges of high performance overhead and the difficulty of defending against certain advanced attacks in software-based protection mechanisms, a method for enhancing memory security is proposed using ARM memory tagging extension (MTE). The low level virtual machine (LLVM) compiler is leveraged for stack-call analysis, and the MTE mechanism is employed for tag management. Experimental evaluation using benchmark programs and 50 vulnerable applications demonstrates significant defensive capabilities of the method against data-oriented attacks, with an average runtime overhead of 52.8% and a file size overhead of 17.7%. This research has made advances in the defense against data-oriented attacks, achieving a favorable balance between security and performance efficiency.

Key words: memory error; data-oriented attacks; ARM MTE; automatic defense; low level virtual machine (LLVM)

随着控制流完整性(Control-Flow Integrity, CFI)技术的广泛部署,传统劫持程序控制流的内存破坏攻击得到有效遏制^[1-3]。然而这种防御机制的普及使得数据导向攻击(Data-Oriented Attacks)快速发展^[4-5]。此类攻击仅操作非控制流数据而不篡改代码指针,遵循合法的控制流图路径,通过破坏栈变量等数据结构实现信息泄露^[6]、权限提升^[7-8]乃至任意代码执行^[9]。以 WU-FTPD 服务器的格式字符

串漏洞为例,攻击者可在不破坏控制流的前提下,篡改 setuid 系统调用使用的 pw→pw_uid 参数值以获得更高权限,如下列代码。由于不扰乱程序正常执行流程,数据导向攻击具有更强的隐蔽性,使得传统防御措施难以检测与预防。

```
1. struct passwd { uid_t pw_uid; ... } *pw; ...  
2. int uid = getuid();  
3. pw->pw_uid = uid;
```

收稿日期:2024-03-12;修回日期:2025-12-17

作者简介:梁思远(1998—),男,硕士,主要研究方向为嵌入式安全。E-mail:siyuanliang98@163.com

通信作者:刘铁铭(1977—),男,副教授,博士,主要研究方向为嵌入式安全。E-mail:fxliutm@163.com

```

4. ...           //format string error
5. void passive (void) {
6.     ...
7.     setuid(0);
8.     setuid(pw->pw_uid);
9.     ...
10. }

```

针对上述威胁,研究人员提出数据流完整性、SoftBound^[10]和HardScope^[11]等以非控制数据保护为核心的防御方案。这些方法主要依赖影子栈等软件机制实现^[12],普遍存在性能开销偏高、适用场景受限等问题,软件实现自身也可能引入新的漏洞^[13]。同时,实现复杂度与运行时开销的显著增加也限制了这些方案在实际部署与规模化应用的效果。

硬件辅助内存标记技术为缓解上述问题提供了新思路。ARM内存标记扩展(Memory Tagging Extension, MTE)利用硬件功能为每个内存块及其对应指针分配唯一标签,建立双层引用机制^[14-15]。内存访问时,MTE验证指针标签与内存位置标签的匹配性,不匹配则触发安全违规响应。现有研究已将MTE用于增强内存安全防御,但主要集中在堆内存保护,且依赖手动源代码分析进行标记插入。此外,MTE技术集成至现有系统需进行大量代码修改,工程实现难度较大;技术尚属新兴,缺乏充分的测试验证与案例研究。

近些年在利用ARM MTE增强内存安全方面的研究主要有以下3类代表性的工作。1)通过静态分析将内存分配划分为安全类与危险类,并实施编译时混淆保护,从而实现内存防护^[16];2)通过MTE在实际系统中的通用缺陷,针对4位标签导致的碰撞概率高、防护弱问题,提出基于BTree压缩的内存高效标签存储方案;3)基于对象级与页面级多粒度标签融合架构实现,提升检测概率,增强防护能力^[17]。

这些工作也存在共同的局限性。1)防护重心集中于堆内存,对栈内存的自动化分析缺失;2)所有方法均需要人工介入源代码标记过程;3)缺乏兼顾安全性与工程可行性的自动化方案。

本文提出一种面向栈内存保护的自动化MTE防御方法。该方法通过底层虚拟机(Low Level Virtual Machine, LLVM)编译器传递系统性扫描程序,自动识别栈内存分配点并完成标记插入,降低人为错误风险。针对标签执行结束后的清除问题,设计运行时监控组件持续检查标签协议合规性,

在检测到违规时进行告警;同时实现标签清理机制,在函数或进程不再需要特定内存段时回收标记,防范内存泄漏等安全漏洞。实验基于基准测试集和50个含已知漏洞的程序展开,运行时开销为52.8%,文件大小增加17.7%,并在含有CVE漏洞的Linux程序上验证有效性。该方法将保护重心从手动、以堆为中心的模式转向自动化栈内存防护,以更少人工干预实现内存安全的系统性增强。

1 线程模型与系统概述

1.1 线程模型

在该威胁模型中,假设存在一个攻击性极强的对手,该对手完全控制目标程序的数据内存。该模型涵盖涉及内存损坏漏洞的各种场景,例如缓冲区溢出。假设已经实施诸如CFI^[1-3]、数据执行保护(Data Execution Prevention, DEP)^[18]和地址空间布局随机化(Address Space Layout Randomization, ASLR)^[19-20]等众所周知的保护措施,以防御控制数据攻击。

1.2 系统概述

本文提出的自动化保护系统旨在保护程序中的所有栈内存空间,利用ARM MTE机制来增强内存安全性。系统的基本工作流程如图1所示。

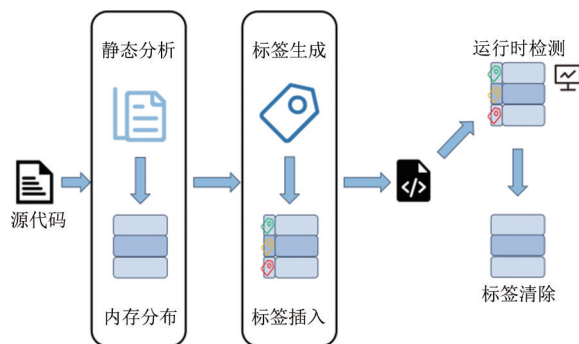


图1 系统架构

该系统始于对每个程序函数的分析,重点是识别与栈内存分配相关的指令;为每个分配的内存块生成随机标签,并将其插入到这些指令中;在系统运行时,不断监视内存访问,并将其与分配的标签进行验证,确保内存安全;当内存块不再使用或函数完成时,从内存块中清除标签,保持系统的完整性和效率。

2 系统设计

通过利用ARM MTE,自动化保护系统在各种

编程的上下文中增强了栈内存的安全性。自动化保护系统的具体实现过程,包括系统设计的核心组件和方法论,如内存分配识别、标签生成和插入、运行时监控和执行,以及关键的标签清除过程通过一个代码示例展示,如图2所示。

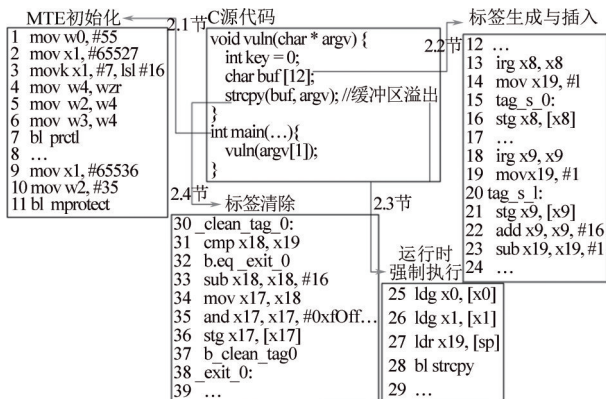


图2 方法演示

2.1 MTE启动和内存分配提取

系统从一个程序分析阶段开始,自动分析程序中的每个功能。该步骤在研究函数代码的同时,也关注在堆栈上分配内存的指令。分析确定并提取出负责在每个函数中分配堆栈内存的特定指令。通过这一方法,系统能够确定代码中分配堆栈内存的确切段。这些点通常对程序的正确操作至关重要,并且容易受到内存漏洞的影响。这种集成的方法确保堆栈内存使用中关键区域的精确和有针对性的识别,为内存保护的后续阶段奠定基础。

在识别函数中的所有堆栈内存分配点后,系统自动插入按分配大小配置的 `mprotect` 指令,该指令通过调用 `PROT_MTE` 参数激活指定内存区域的 MTE 机制,确保各栈内存在受到保护的同时,具备标记能力。

2.2 标签生成与插入

内存分配点确定后,系统开始根据 MTE 协议为每个内存块生成随机标签。该协议为每个内存段分配一个唯一标识符,确保标签随机和不可预测,且符合 MTE 协议规范。这个过程需要重点考虑内存对齐,这是 MTE 机制的操作细节而导致的一个关键问题。

MTE 机制的工作原理是通过向每 16 B 的物理内存添加 4 bit 元数据来标记内存位置,称为标记颗粒。在根据 MTE 协议标记堆栈分配的过程中,如何在遵守 MTE 内存对齐要求的同时,使系统适应并处理不同大小的堆栈分配问题至关重要。对于小于 16 B 的堆栈分配,系统将其填充至 16 B 块,

以确保符合 MTE 的 16 B 结构,并为整个颗粒插入 1 个标签。

对于大于 16 B 的分配,系统将其对齐至 16 B 的整数倍,分配足量颗粒并赋予相同标签标识。此统一标记流程确保了跨不同分配规模的防护一致性与机制有效性。

2.3 运行时监控和实施

在系统中,对程序执行过程期间的内存访问监控是保障内存安全的基础。使用根据 MTE 协议生成和插入的标签,系统不断验证对特定内存块每次访问的合法性。检查访问是否与相应的标签一致,并遵守内存安全原则。如果系统检测到任何未经授权或不正确的内存访问,例如对带有不匹配标记的内存块的访问,将通过引发异常或触发警报来作出适当的响应。

为实现这种程度的监视和保护,系统对栈内存标签进行特殊处理。MTE 架构假定栈指针的可靠性,任何不由栈指针管理的栈内存访问都需要系统从内存地址读取相应的标签。这种做法可以有效防止标签不匹配和潜在的系统错误。

两种关键场景需要对标签的特殊处理:第 1 种涉及函数调用参数的管理。根据 ARM 架构的调用约定,该方法提取所有系统调用参数,并分析它们是否指向栈内存。如果参数指向栈内存,系统将从该内存中读取标签。由于该设计依赖于静态分析,确定一个参数是否指向栈内存具有挑战性。为解决这一问题,在每次参数调用之前执行一段汇编代码。这段代码确定参数的值是否落在有效的栈内存范围内,并在必要时检索其标签。第 2 种涉及栈指针的栈内存访问。系统会分析指令是否是栈内存的读取或写入操作。如果涉及栈内存且操作数不是栈指针,系统会读取该内存段的标签。

通过这些专门的进程,系统可以更好地确保所有堆栈内存访问都是合法和安全的,从而保持程序的完整性和可靠性。

2.4 标签清除

标签清除子系统在全部控制流场景下均满足 MTE 的机密性与完整性要求。该子系统对函数返回语义、非局部跳转、中断/异常入口与返回等事件进行统一建模,并在栈内存生命周期结束时执行同步的标签清除 `tag-eviction` 操作,确保任意栈区间在重新分配前其 `logical tag` 已被回退为 `0x0` (ARM 规范定义的“默认标签”),从而消除元数据残留导致的重用侧信道风险。

2.4.1 常规函数返回路径

编译器在函数 prologue 中记录初始 SP 快照 (SP_0)。epilogue 执行前,通过 $(SP_1 - SP_0)$ 得出当前帧的栈字节数 N ,随后以 MTE 粒度 (16 B) 为步长,依次对区间 $[SP_0, SP_0 + N)$ 内的每一 granule 发出 STG 指令,将其 logical tag 重置为 0x0。该过程与返回地址的 atomical 更新在同一异常级别完成,保证 tag 清除与栈回收的瞬时一致性。

2.4.2 非局部退出

非局部退出包括 longjmp、throw、setjmp 伪造跳转等。编译器在中间表示 (IR) 层为每个可能触发非局部退出的调用点插入 jmp_buf 元数据,并在后端生成栈展开描述符 (.eh_frame)。运行时若检测到控制流经 Unwind_ForcedUnwind 或 longjmp 离开当前栈帧,unwinder 在第 2 遍扫描中回溯所有已访问的栈区间,并立即调用 STG 序列执行 tag 擦除;该操作发生在任何潜在栈内容复用之前,防止“tag 残留→类型混淆”攻击面。

2.4.3 中断/异常与信号处理程序

内核在 entry.S 中将 SP_ELO 切换至独立中断栈之前,通过 IRG 为新栈分配随机 logical tag;在退出路径 (eret) 中,利用 MSR SPSel 恢复线程栈后,以异常发生时的 SP_ELO 快照为基址,对中断处理期间可能污染的区间 $[SP_0, SP_0 + \text{max_irq_stack})$ 执行批量 STG 清零。用户态信号机制沿用同一范式,rt_sigreturn 系统调用在恢复 ucontext_t 之前,由 libc 的 signal trampoline 完成 tag 擦除。

整个过程系统严格遵守 MTE 的操作参数和约束。标记清理必须符合 MTE 的体系结构细节,确保安全维护和符合系统体系结构。这种管理标签的方法不仅维护了内存的完整性,并为将来的安全利用做好了准备,符合当代计算环境中内存安全的严格标准。

3 系统实现

3.1 环境设置

在系统的实现阶段,选择 QEMU 来模拟 Aarch64 架构上的 ARM MTE。为建立相应的虚拟环境,通过使用特定命令行选项,对 QEMU 进行配置,启用 MTE 功能。这些选项包括 -machine virt, mte=on 和 -cpu max。这种设置能够彻底测试和评估 MTE 在各种场景下的性能和有效性。该系统运行在 Linux Ubuntu 22 版本 6.0.1,使用 Clang 版本 17.0.0。

3.2 实现方法

该系统使用 LLVM 的 Clang 编译器将 C 源代码转换为 LLVM IR。通过自定义 LLVM Pass,对 IR 文件进行分析并插入 MTE 标签操作。生成一个集成 MTE 功能的可执行二进制文件。处理流程包括从编译到 LLVM IR,应用 LLVM Pass,生成启用 MTE 汇编代码,并最终编译成可执行文件的全过程。确保 MTE 的集成,提高内存安全性。有关实现步骤,参考算法如下,演示参见图 2。

算法 1 LLVM Pass Workflow for MTE Instrumentation

```
/*Initiate MTE*/
1. for Each I in entryBlock do
2.   If I is AllocaInst then
3.     lastAlloca = &I;
4. If lastAlloca is not null then
   /*Insert a new basic block*/
5.   BasicBlock (MTE_init_BB);
   /*Set the new insertion point*/
6.   SetInsertPoint (MTE_init_BB) → getFirstNon-
      PHI();
   /*Insert the inline assembly code*/
7.   getFirstNonPHI ← initTagAsm;
   /*Memory Allocation Extraction*/
8. For Each BB in F do
9.   For Each I in BB do
10.    If I is AllocaInst then
       /*Get the allocated memory size*/
11.      allocaSize();
       /*Resize and align the memory*/
12.      newSize = (allocaSize + 15) & 15;
       /*Tag Generation and Insertion*/
13.      SetInsertPoint(Alloca →
getNextNode());
14.      insertTagAsm();
15.      getNextNode ← buildtagAsm;
       /*Runtime Monitoring and Enforcement*/
16. For Each BB in F do
17.   ForEach I in BB do
18.     If I is CallInst then
       /*Load tags for arguments*/
19.   For Each &arg in CallInst do
       /*Insert the inline assembly code*/
20.     SetInsertPoint(CI);
21.     checkTagAsm();
22.     CI ← checkTagAsm;
       /*Tag Cleanup*/
23. For Each BB in F do
```

```

24.   For Each I in BB do
25.       If I is Return or LongJump or Interrupt then
26.           SetInsertPoint(i);
/*Insert the inline assembly code*/
27.           cleanTagAsm();
28.            $i \leftarrow \text{cleanTagAsm};$ 

```

3.2.1 MTE 初始化

在 LLVM Pass 中,通过将特定的汇编指令插入主函数指定位置,为 ARM64 架构初始化 MTE。这种集成是激活和配置程序启动时 MTE 的关键。

在该过程当中,使用的汇编指令包括 `pretl` 和 `mprotect`。`pretl` 指令是启用带标签地址 ABI 的重要指令,允许同步 MTE 标签检查故障,并在随机生成的集合中使用所有非零标签。`mprotect` 指令用于在之前创建的匿名内存映射上启用 MTE。

这些指令在 IRBuilder 框架中通过内联汇编的方式实现。通过将指定内容嵌入主函数序言位置,确保 MTE 能够高效地集成到程序中,为整个程序执行期间的内存安全打下坚实基础。

3.2.2 内存分配提取

在 LLVM Pass 的设计当中,需要对每个函数进行详细分析,以确定堆栈内存分配指令。利用 LLVM 工具集中的编译器遍历函数中的每条指令。在迭代的过程中,系统会专门查找 `AllocaInst` 实例,这是指示堆栈内存分配的指令。通过确定这些分配点,为内存标记奠定基础。

3.2.3 标签生成与插入

该关键阶段,LLVM 专注于为先前已识别的每个栈内存分配生成和嵌入标签。这里的主要任务包括修改 `AllocaInst` 指令。这些修改确保分配对齐到 16 B 边界。

为解决这一技术难点,使用内联汇编,将其集成到 LLVM IR 中。这种集成使得系统能够无缝插入特定的存标记指令,这些指令的目的是为栈上分配的每个内存块分配一个随机标签。

3.2.4 运行时监控和实施

在系统的设计中,运行时监控和强制执行功能在确保程序执行期间的内存安全方面起着重要作用。这个过程被分为两个主要场景,每个场景都通过不同的方法来维护内存完整性。

场景 1:即系统调用参数可能来自栈分配的情况,LLVM Pass 会介入以确保安全性。在分析每个系统调用的 LLVM IR 时,LLVM Pass 会检查调用的参数是否来自栈分配的内存。如果是,则会插入内联汇编代码来验证内存访问与其对应标签的匹配性。

场景 2:处理不涉及栈指针的栈内存访问。在这种情况下,由于 LLVM Pass 是一个前端编译器,因此在编译期间无法直接确定这些情况。为克服这一问题,通过开发一个使用正则表达式的 Python 脚本来识别符合此场景的汇编语言指令。该脚本通过分析每一行汇编代码,找到所有栈内存操作。如果某个操作不涉及栈指针,脚本会修改代码以包含 `ldg` 指令,强制程序读取内存的标签。

3.2.5 标签清除

在 LLVM Pass 结束阶段使用标签清理程序。当检测到函数返回,包括涉及非标准退出(如 `longjmp` 或异常)的场景时,该程序将被触发。在这个节点上,LLVM Pass 会计算函数所使用的栈空间量,并根据计算结果系统遍历已分配栈空间。在遍历过程中,同时清除所有内存块中的标签。这项关键任务使用内联汇编来完成,它允许精确且高效地移除标签。

清理过程对于确保内存重用或释放的安全性至关重要。通过移除标签,LLVM Pass 有效减少函数执行完成后分配内存可能遗留的安全风险。

4 实验与评估

对 ARM MTE 框架在增强数据导向攻击的内存安全性方面的有效性进行评估。通过对 GNU Core Utilities^[21]、SPEC CPU 2017^[22]以及 50 个已知漏洞程序的全面测试,分析 MTE 保护对性能和文件大小的影响。并通过考察 MTE 框架在 DNStracer 和 Wget 这两个已有记录 CVE 漏洞且广泛使用的 Linux 程序中的应用,评估其在现实场景中的有效性。

4.1 性能评估

为衡量系统效率,测试实验使用两个基准测试套件,分别来自 GNU Core Utilities 和 SPEC CPU 2017。使用 Clang 17.0.0 将所有程序编译为 64 bit 可执行程序。

1)运行时执行开销。使用 GNU Core Utilities 和 SPEC CPU 2017 基准测试套件,通过比较实施 MTE 前后的基准运行时间,评估 MTE 保护对系统性能的影响。图 3 直观地展示了这些比较结果。值得注意的是,由于两个基准测试套件在执行时间上存在显著差异,为了清晰起见,对 SPEC CPU 基准测试时间进行了缩放处理。

通过分析显示,平均运行时执行开销为 52.8%。需要强调的是,由于实验是在完全虚拟仿真的环境中进行的,与真实实现存在一定的差距,因此这种仿真环境可能会引入比硬件实现更多的延迟。

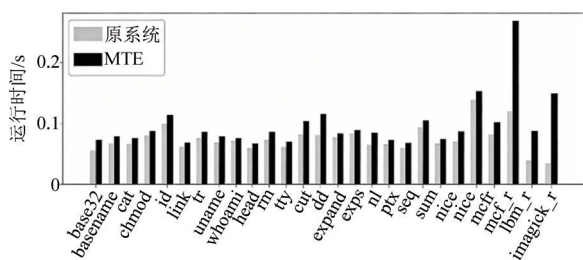


图3 运行时开销

根据ARM的设计原则,基于硬件的MTE将运行得更快,在真正的硬件实现中,观察到的开销会显著降低。

2)文件大小开销。为了评估MTE保护带来的文件大小开销,在GNU核心实用程序和SPEC CPU 2017上进行了测试。该方法将原始文件的大小与受MTE保护的文件进行比较,结果如图4所示。统计得到平均文件大小开销为17.7%,这一增长主要归因于为MTE保护插入了汇编语言代码。

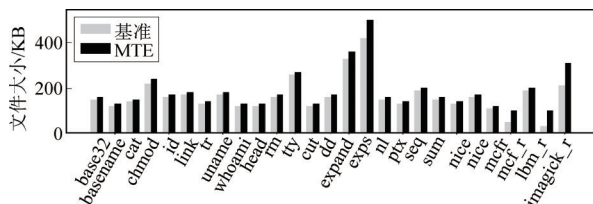


图4 文件大小变化

4.2 安全性分析

为评估该自动化防御方法的实际有效性,选取真实漏洞CVE-2017-9430^[23]作为验证案例,如下所示。

```
// dnstracer_broker.h
#ifndef NS_MAXDNAME
#define NS_MAXDNAME 1024
#endif
// dnstracer.c
int main(int argc, char **argv)
{
    ...
    char * server_name = "127.0.0.1";
    char * server_ip = ...;
    char ipaddress[NS_MAXDNAME];
    char argv[NS_MAXDNAME];
    int server_root = 0;
    ...
    strcpy(argv0, argv[0]);
    ...
}
```

该漏洞存在于DNS追踪工具DNStracer中,对strcpy函数处理长参数时缺乏边界校验机制,导致

栈缓冲区溢出风险。当DNStracer接收超长的命令行输入时,main函数中的argv0缓冲区(固定大小1 024 B,由NS_MAXDNAME定义)会被溢出,进而引发栈内存破坏。在未部署MTE机制的场景下,server_root等后续栈变量面临被覆盖的风险,可能导致程序行为失控。而传统防御机制对此类破坏难以有效检测。本方法通过在栈缓冲区中嵌入内存标签,使strcpy越界写入触发段触发错误并终止程序运行,从根源上阻止栈破坏的发生,显著降低缓冲区溢出风险。

在MTE机制部署开销方面,MTE机制部署后的DNStracer可执行程序文件大小为124 184 B,较原始版本的118 064 B仅增加约5.18%,证明该自动化防御方法在提升安全性的同时对资源消耗的可控性。

5 结束语

本文通过利用ARM MTE增强内存安全性,从而解决面向数据的攻击的挑战。该系统通过LLVM编译器传递和高效的MTE机制自动实现堆栈内存保护,大幅减少了对手动安全措施的依赖。对各种基准测试和脆弱程序的测试验证了方法的有效性,在安全性增强和性能之间取得了平衡。在系统架构中,通过遍历的方式对所有内存空间建立了全面保护机制。然而,在日常场景中,没有必要屏蔽程序中的每个内存空间。一般来说,只有特殊的地方需要设防。保护所有空间的方法需要相当大的成本,大大增加了系统的操作负担。在未来的工作中,将设计一个用来增强与内存空间结构有关的识别过程。这一举措包括识别方法的优化,主要集中在缩小内存保护空间。最终目标是将系统开销缩减到尽可能小的程度。

参考文献:

- [1] ZHANG M W, SEKAR R C. Control flow integrity for COTS binaries[C]//Proceedings of the USENIX Security Symposium. Berkley, USA: USENIX Association, 2013: No.177449371.
- [2] ABADI M, BUDI M H, ERLINGSSON Ú, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Transactions on Information and System Security, 2009, 13(1):No.4.
- [3] ZHANG C, WEI T, CHEN Z F, et al. Practical control flow integrity and randomization for binary executables[C]//Proceedings of the 2013 IEEE Symposium on Secu-

- ity and Privacy. Piscataway, USA: IEEE, 2013:559-573.
- [4] CASTRO M, COSTA M, HARRIS T. Securing software by enforcing data-flow integrity [C]//Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, USA: USENIX Association, 2006:147-160.
- [5] PINCUS J, BAKER B. Beyond stack smashing: recent advances in exploiting buffer overruns [J]. IEEE Security & Privacy, 2004,2(4):20-27.
- [6] HU H, SHINDE S, ADRIAN S, et al. Data-oriented programming: on the expressiveness of non-control data attacks [C]//Proceedings of the 2016 IEEE Symposium on Security and Privacy. Piscataway, USA: IEEE, 2016: 969-986.
- [7] DING R, QIAN C, SONG C, et al. Efficient protection of path-sensitive control security [C]//Proceedings of the 26th USENIX Security Symposium. Berkeley, USA: USENIX Association, 2017:131-148.
- [8] HU H, CHUA Z L, ADRIAN S, et al. Automatic generation of data-oriented exploits [C]//Proceedings of the 24th USENIX Security Symposium. Berkeley, USA: USENIX Association, 2015:969-986.
- [9] CHEN S, XU J, SEZER E C, et al. Non-control-data attacks are realistic threats [C]//Proceedings of the 14th USENIX Security Symposium. Berkeley, USA: USENIX Association, 2005:177-192.
- [10] NAGARAKATTE S, ZHAO J Z, MARTIN M M K, et al. SoftBound: highly compatible and complete spatial memory safety for c [J]. ACM SIGPLAN Notices, 2009,44(6): 245-258.
- [11] NYMAN T, DESSOUKY G, ZEITOUNI S, et al. HardScope: hardening embedded systems against data-oriented attacks [C]//Proceedings of the 2019 56th ACM/IEEE Design Automation Conference. Piscataway, USA: IEEE, 2019:1-6.
- [12] BUROW N, ZHANG X P, PAYER M. SoK: shining light on shadow stacks [C]//Proceedings of the 2019 IEEE Symposium on Security and Privacy. Piscataway, USA: IEEE, 2019:985-999.
- [13] CHENG L, AHMED S, LILJESTRAND H, et al. Exploitation techniques for data-oriented attacks with existing and potential defense approaches [J]. ACM Transactions on Privacy and Security, 2021,24(4):No.26.
- [14] SEREBRYANY K. ARM memory tagging extension and how it improves C/C++ memory safety [J]. Login USENIX Mag, 2019,44:No.195804188.
- [15] PARTAP A. Memory tagging: a memory efficient design [DB/OL]. (2022-09-01) [2023-12-16]. <https://arxiv.org/abs/2209.00307>.
- [16] LILJESTRAND H, CHINEA C, DENIS-COURMONT R, et al. Color my world: deterministic tagging for memory safety [C]//Proceedings of the 2016 IEEE Symposium on Security and Privacy. Piscataway, USA: IEEE, 2016: 93-109.
- [17] UNTERGUGGENBERGER M, SCHRAMMEL D, NASAHL P, et al. Multi-tag: a hardware-software co-design for memory safety based on multi-granular memory tagging [C]//Proceedings of the ACM Asia Conference on Computer and Communications Security. New York, USA: ACM, 2023: 177-189.
- [18] Legacy Update. Changes to functionality in microsoft windows XP service pack 2-microsoft download center archive [EB/OL]. (2004-09-21) [2023-12-13]. <https://legacyupdate.net/download-center/download/10418/changes-to-functionality-in-microsoft-windows-xp-sp2>.
- [19] BHATKAR S, DUVARNEY D C, SEKAR R C. Address obfuscation: an efficient approach to combat a broad range of memory error exploits [C]//Proceedings of the USENIX Security Symposium. Berkeley, USA: USENIX Association, 2003:No.989324.
- [20] SNOW K Z, MONROSE F, DAVI L, et al. Just-In-time code reuse: on the effectiveness of fine-grained address space layout randomization [C]//Proceedings of the 2013 IEEE Symposium on Security and Privacy. Piscataway, USA: IEEE, 2013:574-588.
- [21] Gnu's Not Unix. Coreutils-GNU core utilities [EB/OL]. (2025-12-05) [2024-03-12]. <https://www.gnu.org/software/coreutils/>.
- [22] SPEC. SPEC CPU 2017 [EB/OL]. (2017-06-20) [2024-03-12]. <https://www.spec.org/cpu2017/>.
- [23] National Vulnerability Database. NVD-CVE-2017-9430 [EB/OL]. (2017-06-05) [2024-03-12]. <https://nvd.nist.gov/vuln/detail/CVE-2017-9430>.

(编辑: 冯 春)