计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(9):39−49 [doi: 10.15888/j.cnki.csa.008840]

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

跨平台内存安全测试集设计①

沈思豪[1,2], 解　达[1,2], 宋　威[1,2]

[1](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 100093)
[2](中国科学院大学 网络空间安全学院, 北京 101408)
通信作者: 宋　威, E-mail: songwei@iie.ac.cn

摘　要: 内存安全性是一项非常重要的性质, 但它很容易被攻击者利用. 过去针对内存安全问题提出的许多防御方案, 由于性能代价高昂, 很少能够部署在生产环境中. 最近, 随着 RISC-V 等开源处理器架构的兴起, 安全领域迎来了设计新的处理器硬件安全拓展的热潮, 硬件辅助防御方案的性能代价降低, 变得可以接受. 为了更好地支持内存安全处理器拓展的设计, 以及更好地评估处理器内存安全性能, 我们设计了一款兼具综合性、可移植性内存安全测试框架, 并开源了一个 160 测例大小的初始版本测试集, 覆盖了内存时空安全性、访问控制、指针和控制流完整性等方面, 并在 x86-64 和 RISC-V64 两种指令集架构的平台上进行了测试.

关键词: RISC-V; 内存安全; 测试集

引用格式: 沈思豪,解达,宋威.跨平台内存安全测试集设计.计算机系统应用,2022,31(9):39–49. http://www.c-s-a.org.cn/1003-3254/8840.html

Design of Cross-platform Memory Safety Test Suite

SHEN Si-Hao[1,2], XIE Da[1,2], SONG Wei[1,2]

[1](State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China)
[2](School of Cyber Security, University of Chinese Academy of Sciences, Beijing 101408, China)

Abstract: Memory safety is critical but vulnerable. In view of this, numerous defense countermeasures have been proposed, but few of them could be applied in a production-ready environment due to unbearable performance overhead. Recently, as open-sourced architectures like RISC-V emerge, the extension design of enhancing hardware memory safety has revived. The performance overhead of hardware-enhanced defense techniques becomes affordable. To support the extension design of enhancing memory safety systematically, this study proposes a comprehensive and portable test framework for measuring the memory safety of a processor. In addition, the study achieves an open-sourced initial test suite with 160 test cases covering spatial and temporal safety of the memory, access control, pointer and control flow integrity. Furthermore, the test suite has been applied in several platforms with x86-64 or RISC-V64 architecture processors.

Key words: RISC-V; memory safety; test suite

1　引言

内存安全问题一直是安全领域中经久不衰的问题[1]. 从缓冲区溢出漏洞的利用开始, 到现在返回导向编程技术 (return oriented programming, ROP)[1,2]、数据导向编程技术 (data oriented programming, DOP)[3] 攻击的应用, 基于内存安全问题的攻击手段在不断地更新迭代. 相应的, 也不断有内存安全防御方案被提出. 早期的内存安全防御方案大多基于软件, 如栈帧守护者

① 基金项目: 国家自然科学基金 (61802402, 62172406); 中国科学院率先行动“百人计划”青年俊才 (C 类)
本文由“RISC-V 技术及生态”专题特约编辑武延军研究员、宋威副研究员、张科高级工程师以及邢明杰高级工程师推荐.
收稿时间: 2022-03-24; 修改时间: 2022-04-25; 采用时间: 2022-05-16; csa 在线出版时间: 2022-07-22

(stack canary)[4], Ccured[5] 等, 这些方案虽然能够达到较好的防御效果, 但是性能开销较大. 所以这些方案鲜有被应用于商业处理器架构之上. 近几年来, 随着 RISC-V 等开源处理器架构的兴起, 陆续有新的硬件辅助内存安全方案被提出. 硬件支持使得防御方案的性能开销降低. 商业处理器架构逐渐产生了接纳硬件辅助安全方案的趋势.

然而目前, 无论是在工业界还是学术界, 都缺少对处理器内存安全性能进行评估的测试集. 现有的测试集, 如 RIPE[6], CBench[7] 等, 虽然能够在特定的内存安全性质上进行测试, 但是难以系统全面地反映处理器的内存安全状况; 此外, 对于 x86 系列以外的指令集架构的支持也不尽如人意.

因此, 设计一个系统的, 完善的, 跨平台的, 可拓展性强的硬件内存安全测试集就显得尤为重要. 为了解决这个问题, 我们提出了一个可拓展的内存安全测试集框架, 在该框架下给出了大小为 160 个测例的初始版本测试集. 受限于可用硬件资源, 我们仅在 x86-64 和 RISC-V64 两种指令集的不同平台上对测试集进行了测试. 初始版本的测试集涵盖了内存的时间、空间安全性, 内存访问控制, 指针完整性和控制流完整性等几个方面的安全特性.

本文工作可开源获取, 初始版本测试集可以在 GitHub 上找到: https://github.com/comparch-security/cpu-sec-bench.

2 研究背景

2.1 内存安全攻击防御技术概述

内存安全攻击和防御手段其实处于相互竞争和相互促进的关系中.

类似 C/C++的编程语言在底层实现中缺乏内存安全支持, 导致使用 C/C++语言编写的大量应用程序和动态库中包含缓冲区溢出、指针释放后使用 (use-after-free, UAF) 等内存时间和空间安全性漏洞.

代码注入攻击利用上述漏洞, 将预先构造好的恶意代码写入堆栈等用户数据区中, 将栈帧中保存的返回地址等代码指针的值覆盖为恶意代码起始地址, 从而实现恶意的代码执行.

代码注入攻击要求攻击者能够将恶意代码写入用户的可写数据区, 劫持程序的控制流指向恶意代码, 保证恶意代码能够执行. 针对这些前提条件, 一些经典的内存安全防御方案被提出: 栈帧守护者[8] 在函数返回时检测栈帧是否被缓冲区溢出覆盖, 阻止攻击者劫持栈帧中保存的返回地址; 数据执行保护 (data execution prevention, DEP) 设置页面可写不可执行属性, 将可执行页和可写页分开, 保证攻击者即使成功将恶意代码植入用户可写数据区, 也无法将其作为代码执行; 地址空间随机化 (address space layout randomization, ASLR) 通过使用位置无关代码 (place-independent code, PIC), 在程序加载时将加载基地址随机化, 达到将程序执行期间的数据和代码地址隐藏的目的.

由于上述方案特别是数据执行保护与地址空间随机化的性能开销低, 防御效果显著, 所以得到了大范围部署. 直接的代码注入攻击几乎失效了. 但攻击者仍然能够访问和调用程序自身和动态库提供的数据和代码. 更复杂的内存安全攻击手段被提出, 来绕过上述防御方案, 如返回导向编程技术、跳转导向编程技术 (jump-oriented programming, JOP)[9]、伪造对象导向编程技术 (counterfeit-object oriented programming, COOP)[10]、数据导向编程技术等, 这些攻击手段都有一个共同的特征, 即不注入外部代码, 而是直接复用受害者程序和标准库中的代码完成攻击. 此类攻击称为代码复用攻击.

返回导向编程技术是典型的代码复用攻击[11], 常用于构造指令执行序列, 完成对系统函数 mprotect 等的调用, 关闭数据执行保护. 跳转导向编程技术类似于返回导向编程记录, 但在技术细节上存在不同.

返回导向编程技术攻击会改变程序的正常控制流, 控制流完整性保护 (control flow integrity, CFI)[12] 通过在程序的间接跳转前插入验证代码, 保证程序的所有间接跳转都在程序编译时静态分析得到的控制流图的范围内. 根据控制流分析精度的不同, 控制流完整性保护具体可以分为细粒度和粗粒度两类. 前者使用不同的特征值区分不同的合法跳转地址; 后者则不对合法跳转地址进行区分.

传统的控制流完整性保护实现通过二进制分析完成, 重点保护间接跳转指令, 但是并不对类型进行检查, 无法对多态类对象的虚函数表指针提供保护. 伪造对象导向编程技术利用了这一点, 通过在用户数据区伪造对象, 修改虚函数表指针, 复用其它多态类提供的虚函数表, 通过调用一系列虚函数完成攻击.

伪造对象导向编程技术攻击无法通过简单的二进

制控制流分析实现的控制流完整性保护进行检测, 需要结合类型实现控制流完整性保护进行防御. 典型的防御方案如代码指针完整性保护 (code pointer integrity, CPI)[13], 指针标记 (pointer tagging), 指针审计 (pointer authentication) 等. 在 x86 架构下的 GCC 提供了虚函数表验证 (vtable verification, VTV)[14] 特性, 用于验证虚函数调用的正确性, 但是默认没有被 GCC 开启.

数据导向编程技术[3] 通过修改控制程序分支执行的关键数据来进行攻击, 达到劫持程序控制流的目的. 对于一些涉及敏感系统调用的程序, 数据导向编程技术攻击同样可以完成关闭数据执行保护的操作.

应对上述高级攻击的内存安全防御方案一般有两种实现方式. 一种使用纯软件方式实现, 主要在标准库、内核和编译器的层面进行安全增强, 很少或不依赖硬件提供支持, 导致性能开销过高, 难以被大范围部署. 例如, 代码指针完整性保护虽然能够防御大部分的代码复用攻击, 但是由于插入的验证代码过多, 导致性能开销过高, 一直没有被主流编译器 (GCC, LLVM) 采纳.

另一种方式使用硬件辅助的方法提供内存安全支持. 相较于纯软件的实现方式, 硬件辅助的实现方式能够对安全方案进行加速, 大幅降低安全方案实现的硬件开销. 典型的硬件辅助防御方案如 Intel 的 Intel 内存保护拓展 (memory protection extension, MPX), Intel 控制流保护拓展 (controlflow enforcement technology, CET), ARM 公司在 ARM v8.3-A 中增加的 Arm 指针审计 (pointer authentication, PA), ARM v8.5-A 中增加的内存标签拓展 (memory tagging extension, MTE).

2.2 学术研究中硬件安全测试现状

为了比较不同处理器提供的内存安全性, 我们需要一套测试集. 这套测试集: 1) 相对完善, 能够量化处理器的内存安全性; 2) 可移植性强, 能够在多个处理器平台上进行测试.

然而, 现在学术界普遍使用的测试集大多只关注内存安全的某个特定方面, 缺乏对内存安全的整体把握和评估. 以经典的内存安全测试集 RIPE 为例: RIPE 测试集[6] 是一系列缓冲区溢出攻击的集合. 它应用广泛, 常被用于测试控制流完整性保护防御方案. 每个测例都受 5 个变量控制: 缓冲区溢出位置、受攻击的代码指针类型、溢出攻击的类型、恶意代码和攻击使用的函数. RIPE 使用枚举穷尽各个变量组合的方法对缓冲区溢出做了相对完善的测试, 但是对于缓冲区溢出以外的漏洞涉及不多, 所以不适合作为一个完善的处理器内存安全测试集.

在上述观察的基础上, 我们提出了一种处理器内存安全测试框架, 并开源了一个基于该框架的内存安全测试集. 内存安全测试集的初始版本包括 160 项测例, 覆盖了内存的时空安全性 (spatial and temporal safety)、内存访问控制、指针完整性、控制流完整性等方面. 不同类型的漏洞及其相应的防御方案分别由对应的测例进行评测. 测试集已经在 x86-64 和 RISC-V64 两种指令集架构的几个不同平台上进行了评估.

2.3 安全假设

为了将测试范围限制在内存安全上, 我们作如下假设: 1) 攻击者能够控制用户程序的输入, 恶意利用程序的内存安全漏洞如注入恶意代码; 2) 用户程序包含可以被攻击者所利用的内存漏洞, 攻击者可以利用漏洞实现对程序任意地址的读写; 3) 我们还假设攻击者的目的是攻击用户程序空间内的数据, 而不是内核数据; 4) 此外, 我们也不考虑侧信道攻击, 如缓存侧信道、瞬态执行攻击等.

3 测试框架设计

3.1 测试对象

处理器内存安全测试集以平台为对象进行测试. 平台定义为测试集可以运行的环境. 它包括被测处理器以及运行于其上的操作系统. 操作系统包括一个内核以及若干运行时库.

3.2 测试范围

测试集假设: 内存安全是一系列内存安全性质的集合; 所有的内存漏洞和漏洞利用都是由于对某些内存安全性质缺少检查, 使得内存中的值被恶意泄露或篡改. 根据上述假设, 对内存安全的评估可以被细化为一系列对内存安全性质的测试, 测试这些内存安全性质是否能够被恶意利用. 如果一项测例成功完成执行, 说明平台对该测例对应的内存安全性质缺少检查.通过的测例数和被安全检查拦截的测例分布, 可以反映系统整体的内存安全性.

测试集主要关注那些能够被硬件辅助安全方案保护的内存安全性质. 对于利用内存漏洞展开的攻击, 我们分析该攻击破坏或利用了哪些内存安全性质, 而不关注攻击本身.

以缓冲区溢出攻击为例. 缓冲区溢出攻击是一种越过缓冲区边界对值进行修改的行为. 该行为破坏了缓冲区不应该越界访问的内存安全性质. PUMP[15], AArch64 Address Sanitizer 等硬件辅助的安全机制能够对缓冲区越界访问提供防护. 所以, 对于缓冲区溢出, 测试集测试几类常见的访问越界行为, 这些行为可能不构成完整的攻击. 这些行为如果被拦截, 则说明平台保护了缓冲区不应越界访问的性质.

然而, 由于测试集本身也是软件, 也需要在系统环境下运行, 所以单凭测试集本身难以区分一个内存检查到底是通过硬件方式还是纯软件方式实现的. 所以需要保证测试覆盖的内存检查是由平台而不是第三方安全软件实现的. 例如, 二进制翻译技术和专用的内核安全补丁也能提供内存安全防护, 但由于它们使用纯软件方式实现, 所以应当排除在测试范围之外.

3.3 测例分布

测例主要覆盖内存的空间安全性、时间安全性、访问控制、指针完整性、控制流完整性等 5 个方面.

3.3.1 空间安全性

空间安全性指内存访问总是落在正确的数据边界和程序的可见域内的性质. 任何数据边界外或是可见域外的访问都是不安全的. 典型的破坏空间安全性的攻击是缓冲区溢出攻击. 它是对缓冲区的越界访问. 除了缓冲区以外, 栈帧、动态分配对象、全局变量、只读数据等均存在越界访问的风险.

缓冲区溢出按照溢出方向可以分为上溢和下溢; 按照缓冲区位置可以分为堆上溢出和栈帧溢出. 喷射攻击是溢出攻击的一种特殊应用, 它将溢出位置和目标位置之间的全部内存都进行填充, 插入指向目标位置的跳转代码, 降低控制流劫持的难度.

对于缓冲区溢出的检测和防御方法包括内存检测器 (address sanitizer, Asan)[16]、内存标记[17] 和重型指针 (fat pointer)[18]; 对于栈帧越界访问, 使用重型指针在栈帧粒度上保持数据完整性[19]、在栈帧之间填充字节[20]、在栈帧粒度进行数据隔离[21] 都是有效的防御手段; 对于堆越界访问, 部分防御方案将陷阱数据填充在对象之间[22], 或者在对象粒度上进行边界检查.

测试集的空间安全性测例共 98 项, 主要使用两种方式构造缓冲区越界访问: 1) 通过合法的缓冲区指针和越界的地址偏移; 2) 修改合法的缓冲区指针指向界外位置. 测试集对栈上、堆上、全局变量、只读数据中的越界访问都进行了测试.

3.3.2 时间安全性

时间安全性指内存中的数据访问只发生在数据的生命周期之内. 任何发生在程序生命周期之前 (未初始化数据) 和之后 (释放后使用) 的访问都是不安全的. 时间安全性的测例只关心一个生命周期外的访问是否会发生, 不会针对具体的内存分配算法进行测试.

释放后使用相关的漏洞主要包括空悬指针 (dangling pointer), 未初始化变量等. 在堆上和栈上的空悬指针都会为程序带来较大的安全隐患.

应对空悬指针的防御方案包含空悬指针归零[23]、解引用前检查空悬指针[24]、阻止分配器在被释放对象地址进行重分配[25]、阻止未初始化数据访问、细粒度栈空间随机化[26]、内存标记等.

与时间安全性相关的测试共有 13 项, 主要检测栈上或堆上的数据在栈帧或对象被释放后能否被空悬指针继续访问. 此外, 还检查平台是否具有保证相同类型的对象在相同的内存地址不被重新分配、函数每次调用时动态变化栈帧结构等性质.

3.3.3 访问控制

访问控制性质指限制了攻击者内存访问能力的性质. 主要用来防御信息泄露攻击. 程序的函数体代码、全局偏移量表 (global offset table, GOT) 都是潜在的攻击目标. 攻击者在运行时读取程序的函数体代码, 检索可能成为 gadget 的代码片段, 用以构造代码复用攻击.

全局偏移量表用于程序动态链接共享库时检索符号. 由于全局偏移量表表项在运行时动态更新, 所以需要存储在可写页上. 攻击者可以通过读取全局偏移量表表项获取动态库函数在内存中的地址, 造成信息泄露. 攻击者也可以通过修改全局偏移量表来劫持库函数.

针对上述攻击, 主要的防御技术包括地址空间随机化, 防止攻击者读取可执行页的代码随机化、可读不可执行[27] 等. 测试集的访问控制测例共 3 项, 也围绕这些防御技术展开, 主要检查地址空间随机化是否有效, 函数体代码是否可读, 以及全局偏移量表特定表项是否可读等.

3.3.4 指针完整性

典型的控制流劫持攻击和防御手段经常围绕指针展开. 攻击的第 1 阶段修改保存敏感数据的指针, 破坏

了指针完整性; 第 2 阶段使用被修改的指针劫持控制流, 破坏了控制流完整性.

指针完整性测例主要关注保存敏感数据的指针的安全性. 敏感数据指针包括函数指针、虚函数表指针和全局偏移量表.

函数指针通常可拷贝但不可修改, 进行算数运算的情况非常罕见. 函数指针一般通过指针审计和指针标记[13] 进行保护. 虚函数表指针指向一张函数指针表, 其中每个表项指向类型对应的虚函数. 对虚函数表指针的保护方案包括代码指针完整性保护[13]、GCC VTV 等.

测试集的指针完整性测例共 5 项, 主要检测平台是否允许函数指针拷贝和算术运算, 是否允许对虚函数表指针进行读取和修改、是否允许对全局偏移量表进行修改等.

3.3.5 控制流完整性

控制流体现了程序动态执行时指令间逻辑上的先后顺序. 通过代码指针调用完成的控制流跳转称为前向控制流; 通过返回地址完成的控制流跳转称为后向控制流. 前向控制流完整性指保护代码指针解引用到合法的地址; 后向控制流完整性指保护返回地址不被恶意篡改.

控制流完整性相关的攻击方式包括代码注入攻击、代码复用攻击等, 对于使用多态的程序还包括虚函数表劫持攻击. 测试集的控制流完整性测例共 41 项, 主要围绕这些攻击的典型防御方案如数据执行保护, 控制流完整性保护等进行测试.

3.4 测例构造

测试集的整体架构如图 1 所示. 测试样例由两部分组成: 平台无关的测试逻辑, 与平台相关的支持库.

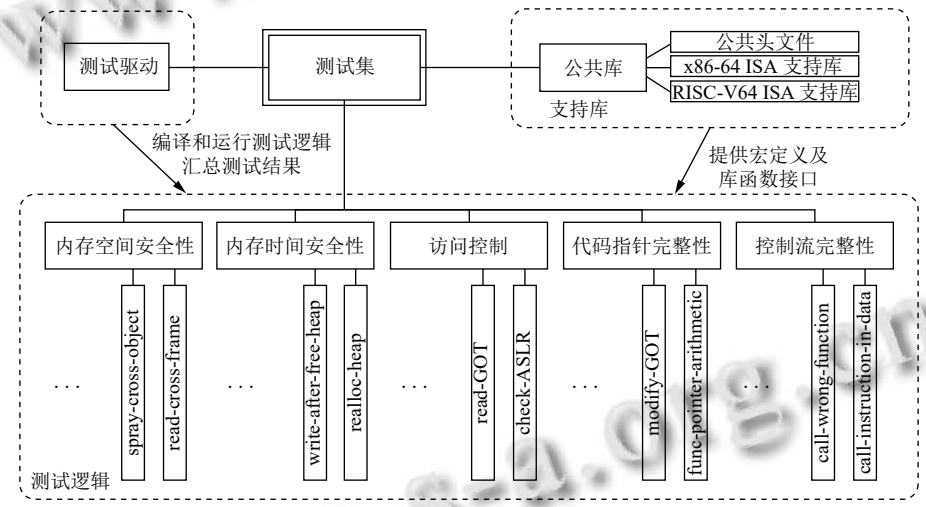

图 1 内存安全测试集整体框架图

每个测例为测试特定内存安全性质的 C++程序; 若某条性质对应的安全检查缺失, 测例可以利用该漏洞完成测试逻辑并返回零值, 表示漏洞被成功利用; 否则测例返回非零值并提示测试失败.

整个测试集的运行通过测试驱动控制. 测试驱动使用指定的编译选项对测例进行编译, 运行测例并统计测例的运行结果, 得到测例通过数量的量化数据.

测例中利用漏洞的恶意行为常常以汇编代码的形式实现. 这是为了防止编译器优化掉恶意行为. 这些汇编代码在平台相关的支持库中. 测试逻辑是使用平台无关的方式编写的; 需要使用恶意代码的部分通过调用平台支持库来完成.

测例的可移植性通过测试逻辑与支持库的划分实现. 二者之间通过宏的定义和调用产生联系. 支持库中的汇编代码使用宏定义的方式组织; 测试逻辑通过引用宏定义完成调用. 不同平台的支持库对相同的宏名称提供定义, 使用平台特定的汇编代码实现相同的动作. 每个测例都会引用公共头文件 (include/assembly.hpp), 该文件对不同平台支持库的头文件进行了包装, 通过不同架构的预定义宏进行区分 (如__x86_64、__riscv64), 编译测试集时编译器会根据预定义宏选择正确架构对应的支持库.

对于新增加的平台或指令集架构, 只需要和其他支持库对同样的宏名称进行定义即可, 不需要修改测试逻辑; 对于新增加的测例, 需要将测试使用支持库提供的宏实现, 如果需要新增宏定义, 则需要在所有的支持库中将对该宏进行定义. 对于新的指令集架构而言, 目前只有约 20 个宏名称需要被实现. 综上所述, 测试集具有良好的可拓展性.

下面以控制流完整性的测例 call-instruction-in-stack 为例, 描述测例的代码结构和测试流程.

测例 call-instruction-in-stack 用来测试将栈上地址作为目标地址的情况下, 函数调用是否能够成功执行. 测试逻辑的主要代码如代码清单 1 所示. 其中 assembly.hpp 和 signal.hpp 均为平台支持库的公共头文件. 头文件 assembly.hpp 负责提供 FORCE_NOINLINE、CALL_DAT、FUNC_MACHINE_CODE 等宏的宏定义. 头文件 signal.hpp 负责提供异常处理所需的接口代码. 部分平台在检测到违反内存安全规则的操作时会抛出异常, signal.hpp 提供将这些异常捕获并产生特定返回值的代码.

代码清单 1. call-instruction-in-stack 的测试逻辑

```
#include "include/assembly.hpp"
#include "include/signal.hpp"

int gv = 1;

int FORCE_NOINLINE helper(const unsigned char* m) {
  CALL_DAT(m);
  return gv;
}

int main()
{
  unsigned char m[] = FUNC_MACHINE_CODE;
  … //异常处理初始化代码
  int rv = helper(m);
  … //异常处理收尾代码
  exit(rv);
}
```

宏 FORCE_NOINLINE 的作用是使强制被修饰函数不生成内联代码, 原因是内联代码在部分平台上会影响测例测试逻辑的正确性; 宏 CALL_DAT(addr) 的作用是将 addr 作为目标地址, 进行函数调用; 宏 FUNC_MACHINE_CODE 的作用是模拟函数体代码, 使得 CALL_DAT 宏产生的函数调用一旦成功执行, 后续指令能够正常返回或是产生特定异常并被 main 函数中异常捕获逻辑捕获, 使得测例能够正常退出.

宏 CALL_DAT 的具体实现为 C 扩展内嵌汇编代码, 所以不同的指令集架构上, 实现各不相同. 在 x86-64 指令集架构上其实现如代码清单 2 所示, 在 RISC-V64 指令集架构上其实现如代码清单 3 所示.

代码清单 2. CALL_DAT 宏的 RISC-V64 架构实现

```
#define CALL_DAT(ptr)          \
  asm volatile(                \
    "jalr ra, %0, 0;"          \
    : : "r"(ptr)               \
    : "ra" )
```

代码清单 3. CALL_DAT 宏的 x86-64 架构实现

```
#define CALL_DAT(ptr)          \
  asm volatile(                \
    "call *%0;"                \
     : : "r" (ptr)  )
```

如果需要将本测例移植到 ARM AArch64 架构, 则只需要在 ARM AArch64 指令集架构的平台支持库和头文件中实现对 FORCE_NOINLINE、CALL_DAT 和 FUNC_MACHINE_CODE 这 3 个宏的定义即可.

测例测试逻辑的核心部分在 helper 函数. 如果 helper 函数中的 CALL_DAT 宏成功执行, FUNC_MACHINE_CODE 将会和 main 函数中的异常处理代码结合, 将 rv 的值设置为 0. 测例将以返回值 0 退出, 表示测试通过. 如果 CALL_DAT 宏的执行抛出异常, 则 main 函数中的异常处理代码会将抛出的异常转换为非零返回值−1 并退出程序.

4 测试结果及分析

4.1 测试平台

对于 x86-64 架构, 我们使用一台较旧的 Intel i7-3770 CPU 搭配 Ubuntu 16.04 操作系统, 和一台较新的 Intel Xeon 8280 CPU 搭配 Ubuntu 18.04 操作系统进行测试.

对于 RISC-V64 架构, 我们使用 SiFive 公司的 HiFive Unleashed 和 HiFive Unmatched 两款开发板进行测试, 两块开发板分别基于 SiFive 公司的 u540 和 u740 CPU, 操作系统均为 SiFive 公司提供的预编译 OpenEmbedded 操作系统.

我们在 x86-64 和 RISC-V64 两个 ISA 架构的 4 个平台上应用测试集进行了测试. 平台列表如表 1 所示.

表 1 内存安全测试集运行平台参数

平台	架构	处理器	操作系统	内核版本	编译器	C库
Intel i7-3770	x86-64	Intel i7-3770	Ubuntu 16.04	4.15.0	g++ 5.4.0	GLIBC 2.23
Intel Xeon 8280		Intel Xeon 8280	Ubuntu 18.04	5.4.0	g++ 7.5.0	GLIBC 2.27
HiFive Unleashed	RV64GC	SiFive u540	OpenEmbedded	5.8.2	g++ 10.2.0	GLIBC 2.32
HiFive Unmatched		SiFive u740	OpenEmbedded	5.13.19	g++ 11.2.0	GLIBC 2.28

4.2 测试结果

4.2.1 不同平台间安全性对比

为了保持一致性, 我们在不同平台上统一使用操作系统提供的 GNU g++编译器, 使用相同的编译选项 “-O2 -std=c++11 -Wall”进行编译. 测试集结果的概要如表 2 所示.

表 2 不同平台成功执行测试样例数

分类	测例总数	Intel i7-3770	Intel Xeon 8280	HiFive Unleashed	HiFive Unmatched
空间安全性	98	98	98	98	98
时间安全性	13	13	13	9	9
访问控制	3	3	2	2	2
指针完整性	5	5	4	5	5
控制流完整性	41	28	28	28	28
总计	160	147	145	142	142

时间安全性: Intel Xeon 8280、HiFive Unleashed、HiFive Unmatched 平台上, 部分堆上的释放后使用相关测例运行失败. Intel i7-3770 平台全部测例运行成功. 说明除了 Intel i7-3770 平台外, 其他各被测平台都具有一定的内存时间安全性防御能力. 通过调查原因发现, 这些平台使用了较新版本的 GLIBC. 后者采用了新的内存分配算法, 在同一片内存区域释放后和分配前插入了垃圾内容, 阻止了释放后信息泄露以及伪造未初始化变量对象攻击. 不过新的算法仍然能够被强制在同一块内存区域重新分配相同类型的对象, 一些使用空悬指针的释放后使用攻击仍然有效. 此外, 各个被测平台对栈上的释放后使用同样缺乏有效的安全检查.

指针完整性: 在各个被测平台上, 读写代码指针和虚函数表指针的测例全部成功执行. 虽然编译器在编译阶段给出了指针算数运算的警告, 但是指针算数运算测例在各个平台仍然能够成功执行. 修改全局偏移量表表项的测例在 Intel Xeon 8280 平台上执行失败, 但在其他平台上成功执行. 说明 4 个被测平台中, 只有 Intel Xeon 8280 平台默认提供了部分指针完整性检查. 该检查来自重定位只读保护 (relocation read-only, RELRO), 大多数 Linux 发行版都默认提供了部分重定位只读保护. 但是在 HiFive Unmatched 和 HiFive Unleashed 两个平台上重定位只读保护并没能覆盖库函数的入口.

访问控制: 检测发现 Intel i7-3770 平台的地址空间随机化测例成功执行. 其他各被测平台除了地址空间随机化测例以外其余测例都成功执行. 说明被测平台中大部分都默认开启了地址空间随机化保护, 但是缺乏对信息泄露的进一步防御. 分析原因发现, Intel i7-3770 平台编译器默认的编译选项不支持生成位置无关代码, 导致对用户程序的地址空间随机化无法使用. 增加“-pie -fPIE”选项后地址空间随机化相关测例执行失败, 地址空间随机化保护成功开启.

空间安全性: 在 4 个被测平台上, 所有 98 个测例都成功完成了测试. 说明被测平台默认提供的安全防护中缺乏对内存越界访问的安全检查. 软件上常使用 address sanitizer 来检测越界访问, 但是性能代价太高, 只适合在开发阶段使用, 无法部署到产品中. 硬件拓展如 CHERI[28]、PUMP[15] 等虽然实现了对越界访问的检查, 但是是以修改系统 ABI、增加硬件开销和性能开销为代价的.

控制流完整性: 对于后向控制流劫持相关的测例, 与返回导向编程技术相关的测例都成功执行; 代码注入攻击的测例悉数被数据执行保护拦截. 对于前向控制流劫持, 除了代码注入攻击的测例被数据执行保护拦截, 其他类型攻击相关的测例都成功执行. 对于虚函数表保护, 替换虚函数表、伪造虚函数表的相关测例均成功执行. 对部分使用新的内存分配算法的平台, 虚函数指针复用攻击相关的测例执行失败, 调查原因发现, 新的内存分配算法在释放对象时清零了虚函数表指针. 上述被测平台都具有一定的控制流完整性防御能力.

总的来说, 在各个被测平台上, 由默认配置提供的安全防护并无太大区别. 各平台默认都没有对空间安全性提供有效的保护; 在 HiFive Unleashed 和 HiFive Unmatched 平台上由于配套的工具链和运行时库增加了安全防护, 所以提供了更好的时间安全性保护. 地址

空间随机化和数据执行保护虽然为各平台提供了一定的内存安全防护能力, 但覆盖面较窄, 只能限制在特定的几项内存安全性质上.

4.2.2 不同编译器与编译选项间对比

编译器不同的编译选项也提供了部分安全防护. 在 Intel Xeon 8280 平台上, 使用 GCC 10.3.0 和 GLIBC 2.32 对不同的编译选项进行了测试. 为了测试 LLVM 提供的控制流完整性保护防御机制, 也将 LLVM13 在 Intel Xeon 8280 平台上进行了测试.

很可惜, 由于工具链移植仍然不完整, RISC-V 的 GCC 和 LLVM 没有提供对 VTV 和 CFI 的支持, RISC-V 架构的 address sanitizer 无法正常工作, 剩余的可用内存安全选项测试得到的结果差别不大, 对判断 RISC-V 架构平台的内存安全性意义不大, 所以我们将只对 Intel Xeon 8280 平台的测试结果进行讨论.

我们按照功能将编译器提供的安全方面的编译选项分为几组, 如表 3 所示.

表 3 内存安全相关不同编译选项组

选项组	编译选项	支持的编译器
默认	-O2	GCC 和 LLVM
RELRO	-pie -fPIE -Wl, -z, relro, -z, now	GCC 和 LLVM
栈保护*	-Wstack-protector -fstack-protector-all	GCC 和 LLVM
VTV*	-fvtable-verify=std	GCC
CFI*	-fvisibility=default -fsantinize=cfi -flto -fuse-ld=gold	LLVM
全部防护	上述所有	GCC 和 LLVM
Asan*	-fsanitize=address –param=asan-stack=1	GCC 和 LLVM
无防护	-z execstack -fno-stack-protector (sysctl -w kernel_randomize_va_space=0)	GCC 和 LLVM

注: *表示由于RISC-V架构的工具链原因, 这些编译选项在RISC-V架构的编译器下并不被支持或是实现不完整.

下面对表 3 中的选项组进行解释.

默认选项: 只要求-O2 优化, 其他为编译器默认选项; RELRO: 开启对全局偏移量表的全面保护; 栈保护: 通过在栈中插入 canary 实现栈覆写保护 (stack smashing protection); VTV: GCC 支持的虚函数表验证特性, 用于应对伪造对象导向编程技术攻击; CFI: LLVM 支持的前向控制流攻击防御机制; 全部防护: 对编译器应用上述支持的所有编译选项; Asan: 开启动态 address sanitizer; 无防护: 关闭包括数据执行保护在内的所有防护, 包括内核提供的地址空间随机化等.

在默认选项下, Intel Xeon 8280 平台使用 GCC 10.3 的通过测例数为 142, 与使用平台默认的编译工具相比, 全局偏移量表篡改可行性的测例通过, 但是有 4 个堆上释放后使用的测例失败, 原因是采用了新的 GLIBC 库. 使用 LLVM 通过的测例数同样为 142, 不过由于 LLVM生成的代码默认不开启 PIE 选项, 并且在编译时不允许代码指针算术运算, 所以具体成功执行的测例稍有区别.

在 RELRO 选项下, Intel Xeon 8280 平台下 GCC 编译通过测例数减少了 1, LLVM 编译通过测例数减少了 2. 可见开启 RELRO 选项对测试集涉及的内存安全漏洞并不敏感.

测试结果如表 4 所示.

开启栈保护选项下, Intel Xeon 8280 平台下对测试集通过测例数几乎没有任何影响, 因为大多数返回导向编程技术攻击都可以定位返回地址保存位置, 并在不触碰 canary 的情况下能够修改返回地址. 失败的测例为伪造栈帧攻击相关的测例.

表 4 Intel Xeon 8280 平台下不同编译选项组测试集编译运行通过测例数

编译器	默认	RELRO	栈保护	VTV	CFI	全部	Asan	无防护
GCC	142	141	141	136	N/A	134	8	155
LLVM	142	140	141	N/A	141	138	21	154

VTV 选项下, Intel Xeon 8280 平台下 6 项伪造对象导向编程技术相关测例都测试失败. 不过将虚函数表替换为子类、父类的行为仍然没有被拦截.

CFI 选项下, Intel Xeon 8280 平台下几乎没有提供任何安全增强. 可能的原因是 LLVM CFI 要求在链接期间对所有的类定义都可见. 这需要使用静态链接方式编译. 而为了应对编译器优化策略, 所有可执行文件均以动态链接方式链接. 这导致链接时分析将对虚函数表指针和函数指针的修改操作识别为了合法操作.

全部防护选项下, Intel Xeon 8280 平台下 GCC 编译测试集共有 26 项测例失败; LLVM 编译测试集共有 22 项测例失败.

开启 Asan 选项下, Intel Xeon 8280 平台下 GCC 编译测试集通过的测例数减少为 8. 通过的测例仅包括两项访问控制测试 (read-func 和 read-GOT) 以及 6 项栈上释放后使用攻击测例. LLVM 编译测试集通过的

测例数减少为 21. LLVM 编译测试集中, 返回导向编程技术和伪造对象导向编程技术攻击相关的测例都测试失败, 但是跳转导向编程技术攻击相关的测例仍然成功执行, 全局偏移量表表项修改可行性的测例也成功执行. 不过 LLVM 编译测试集中所有的释放后使用相关测例都测试失败, 包括被 GCC Asan 漏掉的栈上释放后使用攻击测例.

无防护选项下, Intel Xeon 8280 平台下 GCC 编译测试集仅有 5 项测例失败. 失败测例均为堆上释放后使用攻击测例. LLVM 编译测试集除了没有编译通过的代码指针算术操作测例之外, 结果与 GCC 编译测试集相同. 结合上述各点, GCC 编译器与 LLVM 编译器安全特性提供的内存安全检查大致相近, 只是在使用动态链接类型定义时 LLVM 的 CFI 安全特性未能发挥有效作用, 相比于 GCC 稍逊一筹. 不过, LLVM 测试集中关于代码指针算数运算的测例没有通过编译, 而 GCC 测试集中只是给出了警告, 这也说明两款编译器对于内存安全问题防护具有不同的侧重点. 两款编译器提供的 address sanitizer 拦截了绝大多数的内存安全恶意行为, 说明大多数的内存安全性质都依赖于内存的空间安全性.

5 讨论与展望

5.1 相关工作

关于测试集, 早期的测试集主要用于测试计算机的计算性能. 19 世纪 70 年代的 LINPACK 测试集用于测量计算机进行线性代数数值计算的性能, 至今还用于超算的性能衡量中. Dhrystone[29] 为衡量计算机普通整数运算提供了性能指标; CoreMark 专注于微控制器的性能测量; SPEC 测试集[30] 则用于性能更强的通用计算机. PARSEC[31] 则主要集中于衡量共享内存和多线程应用的性能.

在 2005 年, Kratkiewicz 等[32] 提出了使用构造的小型缓冲区溢出攻击测试现有的软件防御方案. 2006 年, BASS[33] 吸收了 SPEC 的思想, 将 7 个包含有不同种类内存漏洞的测例综合进行安全性验证, 同时提供了一个框架用于自动生成利用内存漏洞攻击. 据我们所知, BASS 是最早的尝试衡量计算机安全性的测试集; 然而该测试集的测试范围只限定在几个特殊的内存空间漏洞上. RIPE[6] 是当前内存安全领域应用最为广泛的安全测试集. 通过枚举几种攻击方式的组合, RIPE 能够覆盖 850 种缓冲区溢出攻击和返回导向编程技术攻击. 它也被用于衡量硬件辅助的控制流攻击防御方案. 但是按照 RIPE 的方法覆盖缓冲区溢出和返回导向编程技术攻击就需要 850 项测例, 要对内存安全进行较全面的覆盖可能难以实现.

最近几年出现了新的安全测试集设计. CONFIRM[34] 是最近提出的用于衡量不同控制流完整性防御方案的兼容性和可用性的安全测试集, 但是缺少对于安全性的评估. CBench[7] 对控制流完整性防御方案的实际效果进行评估, 采用与 BASS 类似的设计, 共使用 7 个大类共 18 个包含漏洞的程序. 与本文工作相比, CBench 使用完整的攻击进行测试, 而且集中在被测防御机制本身, 而不是实现这些机制的平台, 另外, CBench 也不支持跨平台, 只能在 x86-64 架构上运行.

5.2 对其他主流指令集的支持

由于目前我们可用的平台支持的指令集架构只包括 Intel x86-64 和 RISC-V64, 测试集目前仅在这两个指令集上进行了测试, 对于其他指令集架构的支持正在进行中. 未来计划增加对 ARM AArch64 和龙芯/MIPS 指令集架构的支持.

5.3 对测试环境的讨论

虽然主要测试目标是处理器及相应的指令集架构的内存安全水平, 但是测试集的执行并不能脱离测试环境. 这也导致在测例不通过时, 有时较难区分具体是处理器的硬件防御机制起了作用, 还是操作系统、编译器或标准库的软件防御机制起了作用.

上述问题向测试集引入了操作系统、编译器和标准库等无关变量. 一种消除这些无关变量的方法是将所有被测平台都强制安装特定的操作系统、编译器和相同版本的标准库. 这种方法虽然在理论上可行, 但是实践的难度很大. 如果将测试环境缩小到只包含内核与命令行工具的最小系统, 在嵌入式平台上比较容易实现, 但是在一般的服务器和 PC 机上安装最小环境则比较困难. 如果将测试环境规定为特定版本的操作系统发行版 (如 Ubuntu), 那么这一发行版并非能够被所有被测平台支持, 如 Mac M1 和其他众多嵌入式平台等.

基于上述原因, 我们不强制所有平台运行特定的操作系统、编译器和相同版本的标准库, 而是默认为某种发行版, 假定该发行版提供的测试环境足够小. 我们将被测目标的含义扩大为硬件平台及其支撑的运行环境. 受控变量除了处理器和硬件平台之外, 还包括平

台上运行的操作系统、编译器和标准库.

为了消除增加受控变量带来的影响, 保证能够正确的分析测试的结果, 在测试集的构成上, 测例尽量使用不同的非零返回值去标注不同位置和不同原因造成的测试失败, 从而为判断生效的防御类型提供线索.

此外, 我们也使用现有的平台对编译器提供的内存安全标志选项进行了讨论, 分析了主流编译器提供的内存安全防护的有效性. 操作系统和标准库这些变量对内存安全防御的影响也可以通过配置不同的内核安全功能、标准库版本进行评估. 不过限于篇幅和工作量的关系, 这些评估现在还没有展开.

6 结论

我们设计了一套兼具综合性和可移植性的内存安全测试集框架. 初始的测试集包含 160 项测例, 覆盖了内存时空安全性、访问控制、指针完整性和控制流完整性等几个方面. 每一类漏洞及其相关的防御方案都被若干测例评估. 为验证可用性, 我们将测试集在 Intel x86-64 和 RISC-V64 指令集架构上进行了评估. 我们的评估结果显示, 虽然地址空间随机化和数据执行保护等防御方案对被测平台提供了部分内存安全保护, 但大部分的内存漏洞在部分处理器的默认编译器配置下仍然能够被利用. 开启额外的编译器安全特性能够抵御特定类型的内存安全攻击. 尽管 address sanitizer 作为调试工具不能用于生产环境中, 它在捕获内存安全攻击上十分有效. 就相同平台上的编译器表现来看, LLVM 和 GCC 能够提供相近的内存安全保护, 两者对内存安全保护的侧重各有不同.

致谢

感谢郭雄飞提供的 HiFive Unleashed 开发板以及中国科学院软件研究所 PLCT 团队赠与的 HiFive Unmatched 开发板. 两套硬件设施对我们在 RISC-V 架构平台上的测试起到了很大帮助.

参考文献

1 Szekeres L, Payer M, Wei T, *et al*. SoK: Eternal war in memory. 2013 IEEE Symposium on Security and Privacy. Berkeley: IEEE, 2013. 48–62.

2 乔向东, 郭戎潇, 赵勇. 代码复用对抗技术研究进展. 网络与信息安全学报, 2018, 4(3): 1–12. [doi: 10.11959/j.issn.2096-109x.2018017]

3 Hu H, Shinde S, Adrian S, *et al*. Data-oriented programming: On the expressiveness of non-control data attacks. 2016 IEEE Symposium on Security and Privacy. San Jose: IEEE, 2016. 969–986.

4 Cowan C, Wagle P, Pu C, *et al*. Buffer overflows: Attacks and defenses for the vulnerability of the decade. DARPA Information Survivability Conference and Exposition. Los Alamitos: IEEE, 2003. 227–237.

5 Necula GC, Condit J, Harren M, *et al*. CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems, 2005, 27(3): 477–526. [doi: 10.1145/1065887.1065892]

6 Wilander J, Nikiforakis N, Younan Y, *et al*. RIPE: Runtime intrusion prevention evaluator. Proceedings of the 27th Annual Computer Security Applications Conference. Orlando: ACM, 2011. 41–50.

7 Li Y, Wang MZ, Zhang C, *et al*. Finding cracks in shields: On the security of control flow integrity mechanisms. Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Online: ACM, 2020. 1821–1835.

8 Dang THY, Maniatis P, Wagner D. The performance cost of shadow stacks and stack canaries. Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. Singapore: ACM, 2015. 555–566.

9 Bletsch T, Jiang XX, Freeh VW, *et al*. Jump-oriented programming: A new class of code-reuse attack. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. Hong Kong: ACM, 2011. 30–40.

10 Schuster F, Tendyck T, Liebchen C, *et al*. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. 2015 IEEE Symposium on Security and Privacy. San Jose: IEEE, 2015. 745–762.

11 陈林博, 江建慧, 张丹青. 利用返回地址保护机制防御代码复用类攻击. 计算机科学, 2013, 40(9): 93–98, 102. [doi: 10.3969/j.issn.1002-137X.2013.09.019]

12 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.

13 Kuznetsov V, Szekeres L, Payer M, *et al*. Code-pointer integrity. Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 147–163.

14 Tice C, Roeder T, Collingbourne P, *et al*. Enforcing forward-

edge control-flow integrity in GCC & LLVM. Proceedings of the 23rd USENIX Security Symposium. San Diego: USENIX Association, 2014. 941–955.

15 Dhawan U, Vasilakis N, Rubin R, *et al*. PUMP: A programmable unit for metadata processing. Proceedings of the 3rd Workshop on Hardware and Architectural Support for Security and Privacy. Minneapolis: ACM, 2014. 8.

16 Serebryany K, Bruening D, Potapenko A, *et al*. AddressSanitizer: A fast address sanity checker. USENIX Annual Technical Conference. Boston: USENIX Association, 2012. 309–318.

17 Qin F, Lu S, Zhou YY. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. Proceedings of the 11th International Symposium on High-performance Computer Architecture. Washington: IEEE Computer Society, 2005. 291–302.

18 Kwon A, Dhawan U, Smith JM, *et al*. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. Berlin: ACM, 2013. 721–732.

19 Das S, Unnithan RH, Menon A, *et al*. SHAKTI-MS: A RISC-V processor for memory safety in C. Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems. Phoenix: ACM, 2019. 19–32.

20 Bhatkar S, Duvarney DC. Efficient techniques for comprehensive protection from memory error exploits. Proceedings of the 14th USENIX Security Symposium. Baltimore: USENIX Association, 2005. 255–270.

21 Nyman T, Dessouky G, Zeitouni S, *et al*. HardScope: Hardening embedded systems against data-oriented attacks. Proceedings of the 56th Annual Design Automation Conference. Las Vegas: ACM, 2019. 63.

22 Devietti J, Blundell C, Martin MMK, *et al*. Hardbound: Architectural support for spatial safety of the C programming language. Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. Seattle: ACM, 2008. 103–114.

23 Lee B, Song CY, Jang Y, *et al*. Preventing use-after-free with dangling pointers nullification. Proceedings of the 22nd Annual Network and Distributed System Security Symposium. San Diego: NDSS, 2015. 1–15.

24 Nagarakatte S, Martin MMK, Zdancewic S. WatchdogLite: Hardware-accelerated compiler-based pointer checking. Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. Orlando: ACM, 2014. 175–184.

25 Akritidis P. Cling: A memory allocator to mitigate dangling pointers. Proceedings of the 19th USENIX Security Symposium. Washington: USENIX Association, 2010. 177–192.

26 Aga MT, Austin TM. Smokestack: Thwarting DOP attacks with runtime stack layout randomization. 2019 IEEE/ACM International Symposium on Code Generation and Optimization. Washington: IEEE, 2019. 26–36.

27 Pomonis M, Petsios T, Keromytis AD, *et al*. Kernel protection against just-in-time code reuse. ACM Transactions on Privacy and Security, 2019, 22(1): 5.

28 Woodruff J, Watson RNM, Chisnall D, *et al*. The CHERI capability model: Revisiting RISC in an age of risk. 2014 ACM/IEEE 41st International Symposium on Computer Architecture. Minneapolis: IEEE, 2014. 457–468.

29 Weicker RP. Dhrystone: A synthetic systems programming benchmark. Communications of the ACM, 1984, 27(10): 1013–1030. [doi: 10.1145/358274.358283]

30 Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1–17. [doi: 10.1145/1186736.1186737]

31 Bienia C, Kumar S, Singh JP, *et al*. The PARSEC benchmark suite: Characterization and architectural implications. 2008 International Conference on Parallel Architectures and Compilation Techniques. Toronto: IEEE, 2008. 72–81.

32 Kratkiewicz K, Lippmann R. Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. Workshop on the Evaluation of Software Defect Detection Tools. Chicago: PLDI, 2005. 1–10.

33 Poe J, Li T. BASS: A benchmark suite for evaluating architectural security systems. ACM SIGARCH Computer Architecture News, 2006, 34(4): 26–33. [doi: 10.1145/1186736.1186739]

34 Xu X, Ghaffarinia M, Wang W, *et al*. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. Proceedings of the 28th USENIX Security Symposium. Santa Clara: USENIX Association, 2019. 1805–1821.

(校对责编: 孙君艳)