doi:10.3772/j.issn.1002-0470.2024.01.004

基于软硬件协同的细粒度安全域隔离机制①

李亚伟② 章隆兵③ 王 剑

(计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)

(中国科学院计算技术研究所 北京 100190)

(中国科学院大学 北京 100049)

摘 要 基于内存篡改的攻击能够恶意地修改程序执行环境的关键数据,给程序提供一个安全可信的执行环境是抑制恶意软件的有效手段。本文提出了一种基于软硬件协同的解决方案,能够以函数调用为粒度,为程序执行提供相对隔离的安全执行环境。为了配合软件,在底层提供了 2 大硬件支撑:load/store 指令在访存时都要进行地址检查,同时也设置了访问属性;在硬件页表上增加了函数调用隔离域(CFID),在 TLB 转换时进行安全隔离的检查。提供了 2 种不同场合的数据共享访问策略,在 GEM5 上实现了原型系统,通过运行安全测试集,能够有效地隔离非安全环境。相比于虚拟机和特权级切换的方法,本文的硬件实现几乎没有切换损耗。在 SPEC CPU 2006 的测试集中,本文提出的硬件隔离机制总体性能损耗低于 3%。

关键词 细粒度;隔离执行;硬件安全;软硬件协同

现在的基础软件栈主要采用 C/C ++ 这种非安全语言,这些语言为了方便灵活地操作底层的硬件,给程序员暴露了很多的细节,比如可以直接操作指针、随意修改栈中的内容、无限制地使用内联汇编等。这些编程语言的设计初衷是更加容易地与硬件交互,但是带来易用的同时对系统造成了很大的安全性问题[1]。

对内存任意修改的非安全操作是造成现代各类安全问题的根源。目前各类攻击(attacks)或漏洞(bugs),如 Heartbleed[2] 等,大都是内存安全相关的。为了解决这些安全问题,学术界提出了很多应对机制,主要有基于软件的方法和基于硬件的解决方法。基于软件的实现方式,如 AddressSanitizer[3],使用编译器在解析源程序时,动态地分析相关的敏感性操作(如指针的访问、内存空间的分配等),增加程序对内存访问的监管,减少容易产生安全问题的操作。StackGuard[4] 和 SoftBound[5] 都是采用软件来防护缓冲区溢出(buffer overflow)和边界检查(bound checking)的相关问题。基于软件的方式主要是灵活性比较大,但其最大的问题在于该方式有严重的性能损耗[4-7]。

相比于软件方式的不安全性,基于硬件的方式能够最大程度地弥补软件方式带来的性能上的损耗。硬件的实现方式主要是利用底层硬件提供的模块,配合相关的指令来完成安全性的检查。如 NX[8](no-execute),在页表上增加不可执行位。Intel MPK(memory protection keys)[9] 使用 4 bit 的标记,将进程的内存空间分成了 16 个相毗邻的区域,来区分不同的安全域。基于 SMEP(supervisor mode execution protection)[10-11] 的方式是借助虚拟机,切换不同的特权级,从而达到隔离执行环境的目的,比如文献[12,13]中的实现机制。而传统的 POSIX(porta-

① 国家重点研发计划(2022YFB3105104)资助项目。

② 男,1992 年生,博士生;研究方向:操作系统,硬件安全,处理器架构和编译器优化;E-mail: liyawei19b@ict.ac.cn。

③ 通信作者,E-mail: lbzhang@ict.ac.cn。

(收稿日期:2022-09-15)

ble operating system interface)方式就是采用操作系统提供的 mprotect 系统调用,通过陷入内核特权,增加或者屏蔽页面的属性。

虽然增加硬件的支持能够提供相对安全的执行环境,但还是存在一定的局限性。首先,基于硬件的方法不能细粒度地对程序进行有效的隔离。比如 Intel 的 MPK(memory protection keys)[9] 和 ARM 的 TrustZone[14] 只能提供少数的隔离空间,如果需要隔离较多的空间时,只能借助于来回切换空间,这造成了一定的损耗。其次是借助权限切换的方式,比如基于虚拟机和系统调用的方式,在一些应用场景比较快速时,延迟较大,性能下降明显,如 mprotect(20-50X)[15] 和 MPK(3-13.5X)[16]。

本文提出了一种更加细粒度的隔离方法,以函数调用为隔离边界,能够提供多达 4 096 个相对独立的安全空间,这些空间能够毗邻,也可以横跨多个空间域,应用灵活。本文增加了新的用户态 ICall 与 IRet 指令,这 2 条指令除了正常的函数调用与返回的同时,还能够切换执行环境。相比于其他的隔离方式,该方法切换迅速,不需要陷入特权态,从而有较高的性能。为了提供独立的安全环境,在页表项增加了 12 bit 的隔离标志 GFID(global function identifier)。每次内存分配时,都会根据全局的 GFIDR(GFID register)寄存器设置页表,执行流如果没有权限则无法访问其他的隔离空间。

为了能够在隔离空间共享数据,本文提出了 2 种安全的策略,保证隔离域与非可信区的数据安全。这些策略都有硬件、编译器以及辅助指令的支持,避免软件恶意地修改,从而提高安全性。同时也严格地硬件隔离了程序执行的栈空间,防止指令流随意修改其他栈中的敏感数据。用户不需要大幅度修改源程序,只需要在程序函数调用时显式地加入编译属性标志,编译器会自动插入相关的指令流,以满足不同的安全需求。

在模拟器 Gem5[17] 上实现了本文的原型设计,采用 RISC-V 架构,处理器选择 O3CPU。为了评估本文的设计方案,主要做了安全评估与性能评估。安全评估采用 NIST 测试样例以及手动设计的关于跨域访问的不安全操作代码,实验显示本文方法能够完全终止程序的执行。性能方面的评估,主要是采用 SPECCPU2006 测试集,实验结果显示,本文的设计仅有 3% 的性能损耗。

1 相关工作

1.1 基于内核的技术

最开始提供隔离技术的是操作系统提供的进程隔离技术,进程之间彼此相互独立,但这类实现有较高的延迟。比如基于轻量级上下文的实现 lwCs(light-weight contexts)[18]、SMV(secure memory view)[19] 以及嵌套内核的方法[20]。Mimosa[11] 利用 Intel 的 TSX(transactional synchronization extensions)技术来保护密钥,避免程序窃取和冷启动攻击。这些技术采用类似内核保护的技术切换私有的数据或者安全域。

1.2 基于虚拟化的方法

Dune[21] 利用 Intel 的 VT-x X86 虚拟化技术实现进程内的隔离。SIM(secure in-VM monitoring)[22] 在非安全的客户端虚拟机中使用 VT-x 来隔离安全的监视器。文献[23]使用虚拟化技术提供隔离的沙箱机制,保证云端程序的安全性。这些方法都使用了相关的虚拟化技术,主要的代价开销来源于频繁的系统调用(陷入虚拟机)以及 TLB(translation lookaside buffer) Miss。总体来说基于虚拟化的方式实现的代价比较高。

1.3 基于可信执行环境的方法

这类的实现主要是 Intel 的 MPK 技术以及 ARM 的 TrustZone。比如 IMIX(in-process memory isolation extension)[24] 和文献[11,15],都是利用 MPK 技术,扩展 load/store 指令访问安全区域。MPK 技术与本文的实现类似,都是通过增加页表的一些保留位来区分不同的空间。但本文的设计是以函数的粒度来划分和使用可信的空间,这与 MPK 技术有着本质上的不同,具体的分析在第 2 节详细介绍。

2 设 计

首先需要解释的是进程内隔离的概念,这里所

关注的隔离是指单个进程内部,有些函数的执行结果情况未知,比如调用了某个不安全的库(在 Linux 中典型的 .so 文件),或者执行某个有恶意篡改核心数据的代码片段,本文需要提供一个相对安全的环境,保证调用不安全的函数之后,不影响本进程原有的敏感数据。下面详细论述本文提出设计的目的、动机以及如何解决遇到的挑战。

2.1 设计动机

为了保证执行流(代码片段)拥有隔离的独立环境,首先需要给代码提供内存操作的能力,保证能够将执行的结果保存。其次就是解决各个执行流之间的交互性问题,也就是如何实现数据的可访问性与不可访问性。因此为了保证程序的正确性与安全性,本文需要解决以下 3 个主要的问题。问题 1:如何定性地区分安全区与不可信区;问题 2:如何保证隔离空间的独立性;问题 3:如何处理安全区与非安全区的数据传递问题。

2.2 可信区间的切换

为了保证执行的安全,需要严格管理非安全的操作权限。对于问题 1,首先要确定程序遵守 2 点规范:(1)可信区间需要谨慎使用不可信区间返回的数据;(2)不可信区间不能访问可信区间的数据。这 2 个规范保证程序之间执行的安全特性。其次是如何进入和退出可信区间。通常进入可信区间的方式主要有以下 2 种。第 1 种是通过权限切换,然后进入较高特权级,执行完相应的操作之后,再次返回。这样做的好处是权限明显,非安全区无法直接操作安全区的数据,设计上也比较容易实现。但存在的主要问题就是由于要进行权限切换,借助操作系统发生系统调用,频繁的调用会使得系统的性能较差、开销较大。

另外一种是构造一个虚拟的执行环境,然后通过调用 VMcall 等相关指令进入虚拟机。好处是能够独立出一个进程并行执行,缺陷就是难以实现数据的共享,比如虚拟机需要返回数据,通常要借助其他的方式数据共享,每次调用都是一个比较耗时的处理。对于少数场景不太频繁的切换来说,选择类似虚拟机的方式比较明智。

为了实现细粒度的切换,使用更加简单的方式,类似于普通的函数调用 Call 指令与返回 Ret 指令。进入和退出抽象成函数调用的方式,进入函数,则进入另一个隔离域;函数返回,则退出当前隔离域。不同于普通的系统调用或者虚拟机调用指令,这些都是切换特权的指令,需要陷入内核执行较为复杂的情况判断。而 ICall 和 IRet 指令在执行函数调用的同时硬件也会准备隔离环境,仅需简单的数条指令就可以完成。

2.3 独立的执行空间

问题 2 是实现隔离空间独立的关键步骤,按照程序执行时内存的使用情况依次保证函数执行互不干扰。

首先,函数执行需要保证 3 个区域的数据存取。第 1 个区域是全局数据变量,这个保存在可执行文件(ELF)的数据段,因此在程序初始化的时候保证这部分数据可以访问。第 2 个主要的区域就是函数执行的栈空间,这个需要保存函数执行中的参数传递、临时变量的存储、运行中由于指令集寄存器有限而需要暂时保存在栈内存中的数据,以及比较关键的栈指针和函数返回地址也需要保存在栈中。第 3 个区域是在执行过程中动态地申请内存,比如通过 malloc 等相关 API(application programming interface)分配,最终还是系统调用 mmap 或者 brk 分配内存。

针对全局的数据访问,处理的方式比较容易,就是在程序初始化的时候,设置可见的全局数据访问范围,底层硬件访问方式在第 3 节实现中具体说明。对于栈数据的访问,本文的设计思路是:当前执行的执行流无法操作父栈中的数据,除非使用 2.5 节中设定的数据共享策略,如果一个访问不能满足这 2 个条件,则需要触发访问异常。

动态内存分配为了更好地满足对于即时编译(just-in-time)这类应用的支持,本文设计了更加灵活的处理方式。这类应用需要提供独立的执行环境,运行完即时销毁。在通过 malloc 分配内存时,会根据当前的 GFIDR 寄存器的值来判断是否需要分配新的页。这个寄存器保存着全局隔离域的值,是一个 64 位的值,这个值按照如下的方式更新:当使用 2.2 节中的 ICall 指令时,GFIDR 会自增 1;当

使用 IRet 时,则自减 1。这个寄存器是在程序初始化的时候,由操作系统提供的随机值,无法用其他的指令读取。在操作系统分配一个新的页时,内核会在设置页表时,在页表的[61:50](这个位段称为 IPSD, isolation and protection status domain)保存 GFIDR 寄存器的低 12 位。因此,动态内存区分成了 4 096 个区域。

按照本文的设计思路,当访存指令 load、store 访存时,在 TLB 中做地址转换的同时,还要判断是否符合本文设计的实现规则。表 1 是做转换时主要的判断原则。

表 1　TLB 转换时相应的权限

情况	Delta	访问权限	解释
1	大于 0	N	当前函数调用子函数的内存
2	等于 0	A	当前函数调用当前分配的内存
3	小于 0	A/N	当前函数调用父函数的内存

Delta 表示 IPSD 减去 GFIDR[11:0]的值;N 表示不可访问;A 表示可以访问;A/N 表示访问需要按照具体的共享策略决定。

第 1 种情况,访问子函数产生内存的情况。由于子函数可能返回未知安全性的数据,强制无法访问子函数申请内存的数据。如果需要访问,则可以在传递函数的时候,直接传递有当前函数分配的内存的指针。此时的情况就转化成了第 3 种情况。对于第 2 种情况,本文没有任何的限制,这也是保证库函数的主要原因。当前函数可以任意分配和使用当前函数的内存,无任何限制。第 3 种情况涉及到数据的共享,通过 2.4 节中的具体策略来约束数据的访问权限。

上述针对全局数据、栈内数据以及动态申请内存的保护都在硬件上做了限制,防止执行流通过 ROP(return-oriented programming)攻击任意地被修改,这也是保证安全可靠的前提。具体的安全性将在安全示例中说明。

2.4　数据的共享

问题 3 的处理是设计实现隔离环境的关键。针对上面的设计理念,如何在父函数给子函数传递数据的同时保证相对的独立性是设计本身最主要的安全性之一。针对数据传递的特性,本文提出了 2 个主要的约束,下面依次说明。

策略 1　局部约束:父函数传递单一指针。

单一指针就是这个指针所指向的内存对象中都是元数据,没有指针这类数据类型,这也是比较普遍的调用方式。如图 1 所示。

```
1:  struct  foo_obj  { int RTID;  int  size };
2:  void  func ( foo_obj*  ptr,  int  size)
3:  {
4:      if (ptr->type){
5:          ptr->size = size;      //通过指针访问内存对象
6:      }
7:  }
8:  void  main()
9:  {
10:     struct  foo_obj  local_obj;
11:     struct  foo_obj*  lptr;
12:     lptr  =  malloc ( sizeof ( foo_obj ) );
13:     ...
14:     func( &local_obj,  sizeof ( local_obj ) );
15:     ...
16: }
```

图 1　单一指针传递数据

图 1 中的结构体 foo _ obj 中只包含基础数据,可以有嵌套的数据结构,但是这些数据结构中不能有指针变量。在被调用时,如图中第 14 行,传递给子函数时只需要传递指针。正确情况处理这种情况比较简单,编译器只需要在执行调用的时候,先将指针赋给 a0 寄存器(以 RISC-V 架构为例),再以 a0 为指针空间,产生 ciprii 指令,确保将地址、空间大小及属性在执行 ICall 指令前生效,将这种情况认为是使用局部的约束,元数据全部是数据,不包含指针的情况。在子函数中,无法通过指针来访问 foo _ obj 之外的地址空间,这就保证了与其他的地址空间的独立性。

策略 2　全局约束:父函数传递指针的指针。

当父函数传递给子函数的指针对指向的内存中还保存有指针的类型时,处理的情况有所不同。如图 2 中代码所示。

结构体 foo _ obj 中包含有指针数据,或者嵌套的结构体中包含有这类数据时,处理的方式有所不同。主要不同在于,当没有指针的时候,在函数调用前只需要插入一条 ciprii 指令。但是,有指针的时候,为了保证子函数能够正常访问父函数传递给它的数据,在产生 ciprii 指令的同时,也要保证子函数能够处理ptr- > nextptr- > data这类数据。因为编译器

```
1:  struct foo_obj {  int RTID; int size;
2:                    foo_obj* nextptr;
3:                    long obj_metadata;
4:                    char* data ;
5:                    long data_metadata;  };
6:  void func ( foo_obj* ptr, char* data)
7:  {
8:      ...
9:      ptr->data = data;
10:     // 编译器隐式的自动添加
11:     ptr->data_metadata = sizeof(data) | attr0;
12:     ptr->nextptr = ptr;
13:     // 编译器隐式的自动添加
14:     ptr->obj_metadata = sizeof(foo_obj* ) | attr1;
15:     ....
16: }
17: void main()
18: {
19:     struct foo_obj* lptr ;
20:     char stack_data [32] ;
21:     lptr = malloc ( sizeof ( foo_obj ) );
22:     ...
23:     func(lptr , sizeof ( lptr ) );
24:     ...
25: }
```

图 2　父函数传递指针的指针

处理的时候,首先要把 nextptr 的值加载到寄存器,然后再根据结构体 foo _ obj 的空间排布,再去访问 data 这个成员变量。在这个过程中,要通过 nextptr 这个指针来访问。但是在调用 ICall 指令前就只确定了子函数能够访问父函数的空间,在处理这种情况前,首先要遍历结构体,然后对于这种指针,需要额外产生相关的 ciprii 指令,保证程序的正确性,这种约束称为全局共享约束。

在保证安全性的同时,为了支持数据的共享策略,底层硬件需要提供约束检查。函数调用时,主要是通过参数传递给子函数,然后子函数按照传递的参数来处理。根据前面的说明,在传递指针参数的时候,相当于为子函数提供可访问的内存地址空间。按照 2.3 节所述,需要硬件上提供可访问的内存范围。因此在底层提供上下文约束窗 CSW (context strains windows),每次在函数调用时,如果函数传递的参数中有指针(内存空间)传递,需要有调用者建立可以访问的地址窗口。除此之外,所有的访问都是视为非法访问。

需要注意的是 CSW 和栈的行为相似,随着函数的调用,当父函数进入子函数时,父函数的 CSW 内容需要保存。当子函数退出到父函数时,需要恢复父函数的 CSW 内容。进入和退出时的 CSW 严格相同,这部分由硬件自行保存且读取其他指令无法修改,这就保证了安全性。具体的硬件实现将在下一节中详细地描述。

3　实　现

本节主要讨论实现的具体细节。首先描述软件方面的设计,包括指令集的设计、编译器的支持以及动态运行库的相关支持。其次详细地阐述底层设计实现的细节,包括内存检查单元等。

3.1　指令集扩展

底层选用 RISC-V 架构,使用 32 bit 指令,主要包括以下 3 类:

(1)进入和退出隔离域:ICall 和 Iret;

(2)设置全局访问和 CSW 约束的指令类;

(3)打开和关闭,配置相关的指令类。

上述的 3 类指令中,只有第 3 类是特权级指令,用户态无法使用。其他 2 类都是用户级指令。

上述第 2 类,设置 CSW 指令是实现数据共享的关键。它的指令格式如下所示:

ciprii a0, imm _ attri;

ciprr a0, t0;

其中 *a0* 是传递子函数的参数,通常是指针。imm _ attri 是 14 位的立即数,前 10 位是 *a0* 指向的地址空间大小,后 2 位是属性。属性包括只读、可读可写。第 2 条和第 1 条相似,只是将其中的立即数先保存到寄存器中。这样做的目的是,使其能够处理超过 10 位的地址空间,这主要用于提供更大的地址空间。

3.2　编译器支持

采用 Clang/LLVM 编译组件,编译器主要支持以下 2 个方面。

(1)完成 3.1 节中指令集的扩展,包括 LLVM 后端支持代码的生成以及汇编器对特权级指令的支持,提供 ICall 与 IRet 的切换功能。

(2)由 2.4 节所述,Clang 需要分析函数在调用时的行为,分析参数的类型。生成传递内存空间的大小以及相关的属性。在传递指针的指针时,格外读取指针相关元数据属性,分析程序是否执行符合

编程的安全性,反馈警告消息给开发者改进。

实现上在 Clang 集成了 CallIsolationGuardPass,主要是负责程序行为的分析、参数判断,包括生成 LLVMIntriscs、辅助后期程序进一步分析以及生成 CSW 设置指令。其次增加了 SwitchDomainPass,主要是根据调用点函数的属性,生成相关的切换域指令。在函数 emitPrologue 和 emitEpilogue 中,设置栈的约束空间,保证子函数不会随意访问父类函数的内存空间,增加设计的安全性。

3.3　硬件支持

3.3.1　CSW 表

上下文约束窗 CSW 主要是安全域之间交互的访问窗口。每次函数切换域时提供的指针参数以及相关的空间大小、属性值都保存在这里。其具体的结构如图 3 所示。

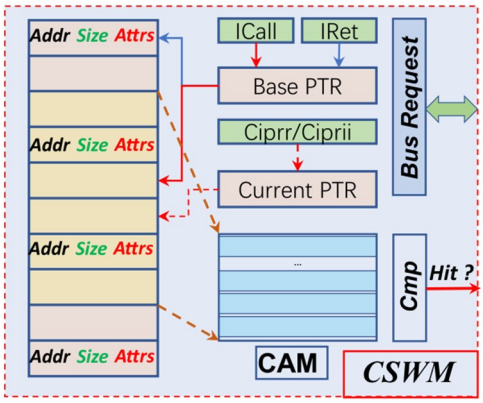

图 3　CSW 表的底层结构

当函数调用 ICall 切换安全域之前,编译器会分析进入函数时所需要的参数。假设有一个参数是传递指针,首先 Clang 会产生一条指令:

*ciprii*a0, imm _ attri;

这条指令会在表项中生成一个项目,相关寄存器设置如图 3 所示。如果并非单一的指针参数,则还需要加载指针的指针相关的元数据,和上述基本相同。当编译器分析完所有的参数时,释放 ICall 指令,使 CSW 的指针指向新的表项。当使用 IRet 时,则恢复原先的指向。考虑到设计最大支持 4 096 个隔离域,而且传递的参数不等,因此选择表的大小为 128 项。如果 CSW 表存满,需要将之前的表项写入内存中,当 CSW 空的时候,从内存中读取。

每次 CSW 请求内存时,都会按照一个完整的栈帧保存。比如当前进入域切换时,传递了 5 个参数,则需要一次性将 5 个表项全部写入或者从内存读出。为了支持不等量的参数,存入内存时保证第 1 项必须是当前 CSW 的个数,这可以在后期 CSW 内存请求时进行预取。

为了保证设计的安全性,CSW 表在内存中保存的位置用户态无法获取,也无法修改。在内核创建进程的时候,会分配这样的内存空间,这是保留的空间,仅内核可修改可读取。如果 CSW 内存请求时,发现此空间已满,则会触发例外程序,然后再分配内存。

3.3.2　内存检测单元

内存检测单元(memory check unit,MCU)主要负责访存是否符合设计的安全约束,一旦出现违例,就会触发异常处理。MCU 主要由 4 个部分组成。

(1)GDM(global domain manager)单元,主要负责 LSU(load store unit)单元产生虚拟地址时,将虚拟地址 IPSD 域和当前的 GFIDR 比较,结果按照表 1 的约束规则处理,其次负责初始化 GFIDR 寄存器。

(2)CSWM(context strains windows manager)单元,主要负责维护 CSW 表,如果需要内存请求,则向下一级 Cache 发起访存请求。其次维护 CSW 指针,也负责监视 ICall 和 IRet 指令。内置 CAM 表存储当前 CSW 的内容,比较是否命中。

(3)G&PM(global data and stack pointer manager)单元,主要负责监视当前栈帧的信息,同时维护全局地址空间可访问表,这个表主要是用于给全局数据变量提供可访问窗口,也可用于整个系统的地址空间信息隐藏。

(4)CCM(compare and check manager)单元,主要负责判断是否满足约束,如果出现违例,则触发异常,这个单元也负责 CSW 表的异常管理。

当 load 或者 Store 指令生成虚拟地址时,将此地址发送到 GDM、CSWM 和 G&PM 单元,如果符合其中的一项检测,有些地址访存会通过其中多项检查。将比对结果发送到 CCM 单元,如果最后符合约束规则可通过,通知 ROB(reorder buffer)单元可以进行指令提交,否则产生异常,通知用户。

3.3.3　检测状态机

每条指令进入 LSU 单元时,也会进入 MCU 的检查队列。每条访存指令都有一个状态的有限状态机,配合完成安全检查,如图 5 所示。

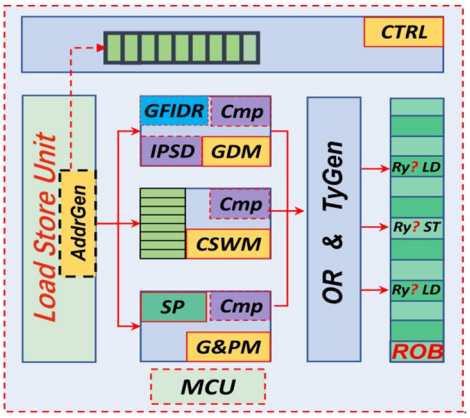

图 4　MCU 单元底层结构示意图

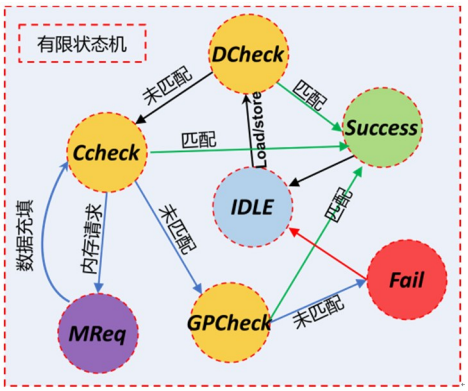

图 5　安全检查状态转换示意图

(1)Idle 状态:load/store 指令进入 MCU 单元队列时,如果当前没有其他检查指令,则进入 TLB 中的 GDM 单元,状态进入 Dcheck 状态。

(2)Dcheck 状态:此时比对 IPSD,如果成功则直接返回 Success 状态。否则进入 Ccheck。

(3)Ccheck 状态:接着会比对 CSW 中的内容。如果命中,则返回 Success 状态,结束比对。如果比对失败,则进入 GPcheck 状态。如果此时比对的内容不在内存中,则需要向内存发起访问请求,进入 Mreq 状态。

(4)GPcheck 状态:进入 G&PM 单元进行检查。此时,如果比对成功,则进入 Success 状态返回,没有成功,则进入 Failed 状态。

(5)Mreq 状态:这时等待 CSW 内存请求,一直等待内存数据响应。读完当前调用的 CSW 后,返回 Ccheck 状态。

(6)Failed 状态:到达此状态说明前面的检查都已经失败,则发起异常请求给 ROB。通知用户这是一条违例的访存指令。

(7)Success 状态:表明此条指令是通过了检查,结束检查。将信息更新到 ROB 中。

3.3.4　硬件优化

由于安全检查模块在访存步骤的关键路径上,模块设计的优劣直接导致整个程序的性能下降。为了提高流水线的利用率主要做了以下的优化措施。

(1)流水线的优化

设计整体流水线如图 6 所示,load/store 流水线的主要步骤为:1)地址产生;2)访问 TLB 单元和 Cache;3)数据对其检查;4)写回。

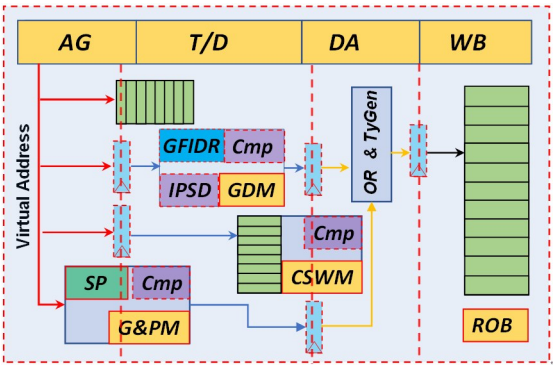

图 6　安全检查底层流水线结构图

如图 6 所示:1)在 AG 阶段输出虚拟地址后,在进入队列的同时,进入 GDM、CSWM、G&PM 单元,将临时信息保存;2)控制逻辑开始上节的状态机控制逻辑,同时将各阶段的信息传递给相关的单元;3)CSWM 单元由于是采用 CAM(content-addressable memory)表来比较,为了减少单周期执行的时间,将其分为 2 个流水阶段,在 TC 和 DA 阶段后产生结果;4)将各个单元比对的结果进入 CCM,最后根据检查结果继续队列中下一个指令操作。

(2)预取的优化

由于 CSW 有可能需要在内存中保存,但是如果等到 CSW 空或者满的时候,将内容读取或者写入内存。由前面的分析可得,有限状态机一直等到数据

的读写完成,这样会导致访存指令一直在 ROB 中等待完成,造成关键路径的等待,因此采用延迟触发的方式来处理。当快要满或者将要空的时候,触发内存逻辑,更早地完成操作。

为了配合延迟触发,在保存 CSW 项的时候做了一定的优化,保存的第 0 项是 CSW 项的数目以及参数掩码,表示是哪几个指针参数的相关属性,之后通知请求逻辑完成数据的读取。

3.3.5　内核和运行时的支持

为了辅助完成安全检查,内核以及库需要做以下的支持。

(1)内核支持:当发送 PageFault 异常时,请求内存分配,而在调用 mmap 或者 brk 的时候会将 GFIDR 寄存器的信息传递给底层内核 vm _ struct。当设置页表表项(page table entry,PTE)时,将其保存在 IPSD 域。

(2)调用系统 malloc 等相关 API 时,分配内存时,需要按照当前的 GFIDR 寄存器来判断是否需要合并分配的内存空间。如果不是,则重新分配新的分配槽。

(3)当程序在加载的时候,Ld. so 处理动态库的依赖时,需要将动态库可读可写的数据段设置成全局内存可见可访问,这保证其在之后的执行时避免非法访存。

4　安全示例

以常用的例子来演示安全机制是如何防护非安全域操作的,如图 7 的代码片段所示。

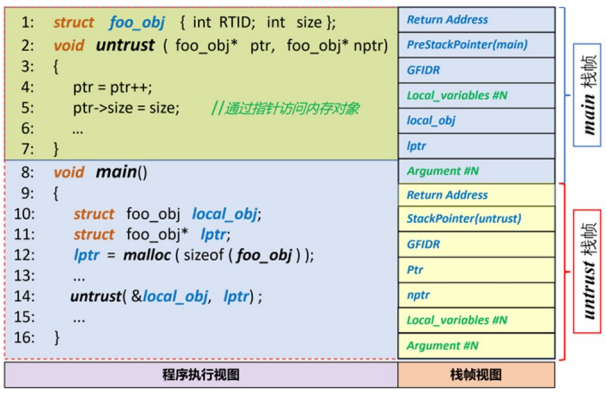

图 7　安全示例代码及栈帧视图

假设 main 函数是安全可信的,且在自己的栈内部保存了局部变量 local _ obj 以及申请的动态内存对象 lptr 的指针。main 函数需要在调用 untrust 时,在传递指针的同时,也切换了安全域。假设 untrust 函数是未知安全性的指令流,可能被劫持。图 8 的代码段是正常的 RISC-V 的反汇编代码(省区无关代码),包括安全防护后的汇编代码。untrust 指令流通过操作 local _ obj 的指针,间接修改了 main 栈内的数据,如图 7 中代码所示。同理,假设 foo _ obj 附近有重要的数据,通过 foo _ obj 的指针操作周围的内存区,造成数据泄露。

如果使用了使用安全防护,如图 8 所示,首先假设 untrust 强制通过栈指针操作 main 中保存的返回地址(图 7 中的栈排布),此时 MCU 单元的 G&PM 模块不能通过检查,报告异常。假设 untrust 想通过 foo _ obj 的指针越界操作 main 的栈数据(图 7 中的 10400 处的代码),此时安全检查会在 CSWM 内发现越界操作,将栈越界信息传递给 CCM。如果通过 lptr 的指针操作动态内存,则会在 TLB 和 GDM 中处理异常。最后综合后将结果传递给 ROB。

```
00000000000104f0 <untrust>:
  104f0:    addi      sp, sp, -32
  104f4:    sd        ra, 0(sp)
  104f8:    isd       gfidr, 8(sp)  // Save GFIDR Register
  10400:    addi      a0, a0, 24  // ptr++, 恶意修改指针
  10518:    sd        a4, 8(a0)
  ...
  10520:    ld        s0, 24(sp)
  10524:    addi      sp, sp, 32
  10528:    iret      // Exit Security Domain
000000000001052c <main>:
  1052c:    addi      sp,sp,-64
  10530:    sd        ra,0(sp)
  10534:    isd       gfidr, 8(sp)  // Save GFIDR Register
  ...
  10550:    mv        a1,a5
  10554:    ld        a0,-24(s0)
  10558:    ciprii    a0, 386       // Update CSW item0
  1055C:    ciprii    a1, 386       // Update CSW item1
  10560:    ildt      gfidr, 8(sp)
  10564:    ICall     ra,104f0 <untrust> // Enter UnSecurity Domain
  ...
  10580:    ld        ra,56(sp)
  10584:    ld        s0,48(sp)
  10588:    addi      sp,sp,64
  1058c:    ret
```

图 8　汇编代码示意图

下面假设赋予 untrust 更加极端的可能性,拥有 Code-Reused 攻击能力[21]。假设它可以跳转到代码 10564(ICall 指令之前),此时 a0 和 a1 都被设置成

目标的内存地址空间,Size 和属性设置成攻击范围。此时如果跳转到地址去,程序会再设置 CSW 表,接着可以跳转到攻击者设计的位置去执行相关非法操作。但是,关键点在于,在执行函数 main 的第 1 条指令时,会在栈内保存当前函数的 GFIDR,如图中 104f8 处指令代码。然后在代码 10560 处比对保存的是否和当前 GFIDR 值相等,如果不相等则触发 CRA 异常,通知上层用户非法操作。

需要注意的是,ildt 指令和 ICall 指令必须成对出现,并且 ildt 指令不能出现在 ciprii 类指令的前边。如果检查到恶意提前执行,也会报 CRA 异常。ildt 指令执行完后,会在底层硬件置标值位。ICall 指令执行前要检查此标志,如果没有也会报告硬件 CRA 异常情况。

假设 untrust 函数再试图去操作 main 栈中的数据,来修改之前 main 保存的 GFIDR 值。但是通过 local _ obj 和 lptr 指针越界处理 main 栈中的数据都会被 MCU 单元捕获,然后会进行异常处理。因此 untrust 都是无法修改安全域保存的 GFIDR 值。

5 评 估

这部分讨论如何全面评估安全机制。主要从下面 3 个方面来分析。

(1)安全性:通过安全测试集验证本文解决方案是否能够保证数据的安全。

(2)性能评估:通过分析 SPECCPU2006 来评估使用安全机制带来性能上的损耗。

(3)实用性分析:主要是提供一些编程上的规范,指导开发者更加容易地使用机制。

5.1 实验环境

5.1.1 硬件平台

采用 Gem5 模拟器作为硬件的实验平台,使用 RISC-V64 位架构。中央处理器(central processing unit,CPU)模型选择 O3CPU,支持乱序执行。具体的配置选项如表 2 所示。

5.1.2 软件环境

编译器使用 Clang/LLVM-12.0.1 版本[25],C ++ 库使用配套的版本,C 库使用 musl-libc 1.22 版本[26]。本文修改了其中 malloc 的相关代码,以支持安全机制。也修改了 ld. so 程序,保证动态库在加载的时候,能够正常地初始化执行。本文并未在 SPEC 中使用,只是测试了其基础功能。内核使用 Linux-5.6 版本,修改了异常处理部分代码,能够在 PTE 设置的时候更新 IPSD 域。增加了对 MCU 产生异常的处理程序。其次在加载程序的时候,能够随机设置 GFIDR,支持进程切换和开启、关闭安全机制的功能。

表 2 Gem5 模拟器配置

参数	说明
架构	RISCV64,超标量处理器 O3CPU 模型
核心参数	基本频率 2 GHz,取指宽度 8 指令,发射宽度 8 指令,提交宽度 8 指令,ROB 队列 64 项,Load/Store 队列各有 32 项,128 整数物理寄存器堆
L1 指令 Cache	大小 32 kB,2 路组相联,读取延迟 1 cycle,CacheLine 大小 64 字节,4 项 MSHRMiss 请求
L1 数据 Cache	大小 64 kB,4 路组相联,读取延迟为 1 cycle,CacheLine 大小为 64 字节,4 项 MSHRMiss 请求
L2 共享 Cache	大小 2 MB,8 路组相联,读写延迟为 20 cycles,CacheLine 为 128 字节,24 项 MSHRMiss 请求

5.2 安全性评估

对于安全的评估在第 4 节中做了相关的说明,本节主要是通过一些有异常的程序来测试机制的安全性。

5.2.1 Juliet 安全测试集

Juliet 测试集是由 NIST(美国国家标准技术研究院)收集的一些安全测试集,用 C/C + + 语言编写,多架构支持。总共有 118 类安全问题,本文选取了以下几类测试,结果如表 3 所示。

表 3 测试的 Juliet 分类

编号	说明
CWE121	基于栈的缓冲区溢出
CWE122	基于堆内存的缓存区溢出
CWE124	缓冲区向下溢出写,向缓冲区写入数据
CWE126	缓冲区向上溢出读,从缓存区读取敏感数据
CWE127	缓冲区向下溢出读,从缓冲区读取敏感数据
CWE469	修剪指针读写,指针重赋值后判断大小及引用
CWE467	传递 0 大小的类型,将小的值强制转换
CWE562	返回栈变量指针,操作子函数的栈数据

将上述的 8 类测试,逐个选取出来,编译成独立的 ELF 可执行文件。测试结果中,关于缓冲区溢出的部分测试未通过,主要是因为这些测试缓冲区溢出只是针对当前的栈内溢出,并未涉及到新的函数调用,也就是说测试只在一个安全域中进行,因此实验结果会全部通过。其他的关于函数调用(安全域切换)的测试,全部显示捕获了异常,程序中止。

5.2.2 伪造的测试集

手动编写了几大类的测试程序,都是按照不同的攻击类型实现的非法访问代码,具体测试的细节如表 4 所示。

表 4 不同类型攻击测试情况

类型	说明	结果
栈缓冲区溢出	溢出,修改父函数中的返回地址	截获
	溢出修改保存的栈指针	截获
	修改任意的栈数据	截获
	修改保存的 GFIDR 值	截获
指针参数	强制改变指针 size	截获
	强制数据类型转换	截获
	指针的指针没有元数据	截获
	指针的内存空间属性修改	截获
	指针为 0	截获
	指针强制整数指向	截获
	指针为 char *, Void *	截获
	指针 free 后重新传递	截获
Code Reuse Attack	修改返回地址 1	截获
	修改返回地址 2(返回不同的执行点)	截获
	修改条件判断	截获
堆内存相关	任意修改 malloc 的内容	截获
	任意修改 free 后的内容	截获

5.3 性能评估

本节评估设计的性能损耗。首先微观上分析可能存在损耗的方面,其次通过 SPECCPU 2006 来宏观测试性能损耗,最后分析本设计与其他安全机制的性能对比。

5.3.1 微观底层硬件分析

由前面的分析可以看出,存在性能损耗的有以下几个部分。

(1)本文设计中新增了的指令,主要包含 ciprii 类指令在建立 CSW 的时候,会有多余的指令向其中写入约束属性,为安全域切换准备。

(2)CSW 表满的时候,需要向内存写入;当表空的时候,需要从内存中读取。在此期间需要流水线等待完成才能写回提交指令。

为了保证安全域 CSW 表的设置不被恶意地修改,抵御 Code-Reused 攻击,首先要在函数的最开始保存 GFIDR 寄存器的值,然后在 ICall 指令之前重新读取判断是否遭受 ROP 攻击。

上述 3 部分是底层可能带来多余执行时间的环节。在 3.3 节中,在每次访存操作时,MCU 的安全检查流水化在执行 load/store 时同时执行,因此不会产生多余的流水线停顿。剩下的配置指令,保护使能和关闭安全机制这部分代码做了进一步的安全防护,但是这部分代码执行次数较少,因此损耗可忽略不计。

5.3.2 宏观测试集

用 SPECCPU 2006 整形测试集[27]评估安全机制带来的性能损耗。编译器使用 Clang-12.01,C ++ 库使用配套版本。编译选项使用-O2 优化选项,将 SPEC 测试集编译静态可执行文件。为了避免手动的设置安全域切换函数,随机选择函数作为切换点。这部分由动态编译器确定,选取大概 20% 的函数作为测试函数。总共选取了 11 个测试程序,400. perlbench 执行结果异常,在统计过程中去掉。采用热启动的方式,让程序先执行 200 万条指令,然后再统计执行 2 亿条指令的执行情况。

图 9 展示了使用安全机制后与原始执行的时间对比,实验结果全部采取归一化处理。从图中可以看出,实现的安全机制在 SPEC2006 的平均性能损耗为2.83%,这些性能损耗的增加主要在于多余的管理指令和 ciprii 指令,这些指令流水化后的执行只需要 1 个时钟周期。其次是为了保存设计的安全性,在防御 Code-Reused 攻击的时候,首先要在函数开始处产生一个类似 store 的指令,在执行敏感程序的时候,类似 load 的操作,然后执行对比判断,相当于多增加了 3 条指令:load、store 和 bneq。综上所述,这些都会使性能降低。

实验中 445. gobmk 这个测试程序性能损耗最大,达到了 6.55%。分析发现,这个程序中大量使用了在2.4节中的全局共享数据,主要是传递了指针的

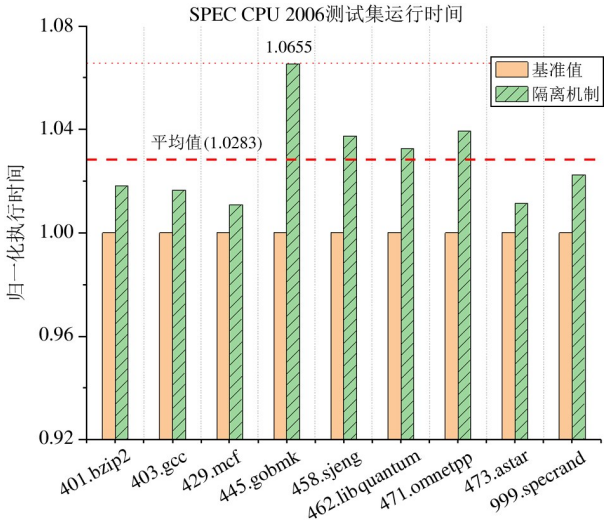

图 9　SPE CCPU2006 执行时间情况

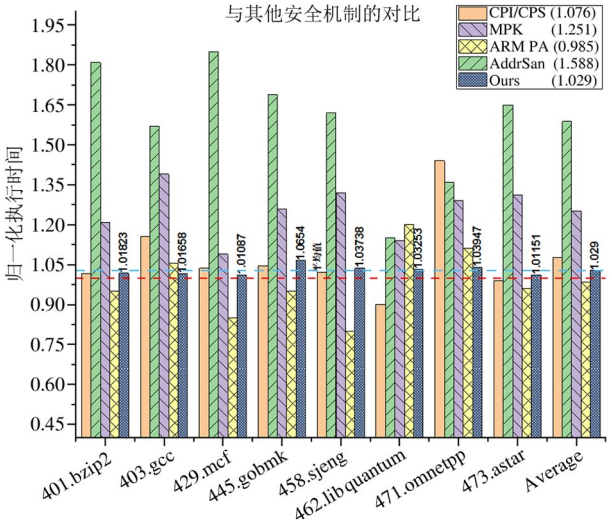

图 10　与其他机制的性能损耗比较

指针这类变量。传递的元数据中每一个指针变量,会产生一个 load 指令,并且也会产生一个 ciprii 指令。当大量使用的时候会造成上述情况,导致程序的性能损耗加大。

为了对比提出的安全机制设计与其他设计的性能损耗情况,选取了与本文设计相似的几个安全机制,在同样的测试集 SPEC2006 下,对比性能损耗情况,具体如下所示。

(1) AddressSanitizer[6]:Clang 的安全保护扩展。

(2) CPI/CPS(code-pointer integrity/code-pointer separation)[28]:用于保护指针完整性,还提供了安全栈保护敏感数据。

(3) MPK[29]:Intel 用于安全隔离的机制。

(4) ARM PA:ARM 的指针认证机制。

图 10 展示的是与其他安全机制的性能对比。可以看出,在使用软件的方式 AddressSanitizer 时,性能损耗最大,将近 60%。这主要是 Clang 在每次进行访存的时候,都要进行一次检查访存是否符合,这样一来每次访存相当于变成了好几条指令,性能损耗最大。其次 CPI 也是软件的方式实现,但是它只是监管程序中所有的代码指针(比如函数指针、返回地址等),因此损耗比较低。IntelMPK 机制的性能损耗仅有 25% 左右,它只提供了一套能够隔离的机制,但还是要用其他的辅助手段保证它的安全性,比如保护敏感数据,抵御 Code-Reused 类的攻击,这样性能损耗也提高了。ARM的指针认证只是针对于返回地址的,保护返回地址不被恶意修改,因此性能损耗最低。

5.4　可用性指导

(1)灵活易用的编译选项。在 Clang 中提供了编译属性_ attribute _((calliso)),在声明函数的时候显式地指出,在需要切换安全域的地方直接调用,编译器会自动分析参数,生成合适的指令。本文还增加了其他的辅助性编译选项,用于增加设计的安全性。

(2)可信空间的选择问题。一般如果使用第三方库的时候不确定其安全性,可以采用此套安全机制。但是以下场景不建议使用,比如使用库函数 malloc 分配内存的时候,因为返回时子函数返回的内存空间已经被 CSW 回收,如果此时再使用则会导致访问异常。

6　结　论

本文提出了细粒度安全域隔离硬件机制,提供了一套完整的解决方案,能够为可信环境提供安全有效的保障,底层硬件能够提供多达 4 096 个独立的隔离域。同时为了防止威胁性最大的 Code-Reuse 的攻击,本文提供了有效的防护机制,能够有效阻止这类攻击,使得隔离机制更加安全可靠。在 SPEC CPU 2006 上的测试结果显示,本文的安全机制性能损耗仅有 3%。

参考文献

[1] Google. Google queue hardening[EB/OL]. [2022-10-21]. https://security.googleblog.com/2019/05/queue-hardening-enhancements.

[2] MEHTA N. The heartbleed bug[EB/OL]. [2022-10-21]. http://heartbleed.com/.

[3] SEREBRYANY K, BRUENING D, POTAPENKO A, et al. AddressSanitizer: a fast address sanity checker[C]// 2012 USENIX Annual Technical Conference. Boston, USA: USENIX, 2012: 309-318.

[4] COWAN C, PU C, MAIER D, et al. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks[C]// USENIX Security Symposium. San Antonio, USA: USENIX, 1998: 63-78.

[5] NAGARAKATTE S, ZHAO J, MARTIN M, et al. Spatial memory safety for C[C]// Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. Dublin, Ireland: Association for Computing Machinery, 2009:15-21.

[6] AddressSanitizer. Clang 16.0.0git documentation [EB/OL]. [2022-10-21]. https://clang.llvm.org/docs/AddressSanitizer.html.

[7] 吴小王, 方勇, 贾鹏, 等. 基于动态分析的控制流劫持攻击检测[J]. 四川大学学报(自然科学版), 2021,58(3):73-79.

[8] WERNER J, BALTAS G, DALLARA R, et al. No-execute-after-read: preventing code disclosure in commodity software[C]// Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. Xi'an, China: Association for Computing Machinery, 2016:35-46.

[9] PARK S, LEE S, XU W, et al. Libmpk: software abstraction for Intel memory protection keys (Intel MPK) [C]// 2019 USENIX Annual Technical Conference. Renton, USA: USENIX, 2019:241-254.

[10] LEE H, SONG C, KANG B B. Lord of the x86 rings: a portable user mode privilege separation architecture on x86[C]// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto, Canada: Association for Computing Machinery, 2018:1441-1454.

[11] LIU Y, ZHOU T, CHEN K, et al. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, USA: Association for Computing Machinery, 2015:1607-1619.

[12] 王丽娜, 周伟康, 刘维杰, 等. 面向云平台的硬件辅助 ROP 检测方法[J]. 清华大学学报(自然科学版), 2018,58(3):237-242.

[13] 廖荣江. 基于隔离技术的防病毒存储设备的设计与实现[J]. 现代信息科技, 2021,5(6):161-163.

[14] AZAB A M, NING P, SHAH J, et al. Hypervision across worlds: real-time kernel protection from the ARM TrustZone secure world[C]// Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. Scottsdale, USA: Association for Computing Machinery, 2014:90-102.

[15] HEDAYATI M, GRAVANI S, JOHNSON E, et al. Hodor: intra-process isolation for high-throughput data plane libraries [C]// USENIX Annual Technical Conference. Renton, USA: USENIX, 2019:489-504.

[16] VAHLDIEK-OBERWAGNER A, ELNIKETY E, DUARTE N O, et al. ERIM: secure, efficient in-process isolation with protection keys (MPK) [C]// The 28th USENIX Security Symposium. Santa Clara, USA: USENIX, 2019: 1221-1238.

[17] LOWE-POWER J, AHMAD A M, AKRAM A, et al. The Gem5 simulator: version 20.0 + [EB/OL]. (2020-07-07) [2022-10-21]. https://research.vmware.com/files/attachments/0/0/0/0/1/1/6/2007.03152.pdf.

[18] LITTON J, VAHLDIEK-OBERWAGNER A, ELNIKETY E, et al. Light-weight contexts: an OS abstraction for safety and performance[C]// The 12th USENIX Symposium on Operating Systems Design and Implementation. Savannah, USA: USENIX, 2016:49-64.

[19] HSU T C H, HOFFMAN K, EUGSTER P, et al. Enforcing least privilege memory views for multithreaded applications[C]// Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria: Association for Computing Machinery, 2016:393-405.

[20] DAUTENHAHN N, KASAMPALIS T, DIETZ W, et al. Nested kernel: an operating system architecture for intra-kernel privilege separation[C]// Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems. Salt Lake City, USA: Association for Computing Machinery, 2015: 191-206.

[21] BELAY A, BITTAU A, MASHTIZADEH A, et al. Dune: safe user-level access to privileged CPU features[C]// The 10th USENIX Symposium on Operating Systems De-

sign and Implementation. Hollywood, USA: USENIX, 2012:335-348.

[22] SHARIF M I, LEE W, CUI W, et al. Secure In-VM monitoring using hardware virtualization[C]// Proceedings of the 16th ACM Conference on Computer and Communications Security. Chicago, USA: Association for Computing Machinery, 2009:477-487.

[23] 赵广强. 基于虚拟化的沙箱防御技术的研究与实现[D]. 广州: 广东工业大学, 2015.

[24] FRASSETTO T, JAUERNIG P, LIEBCHEN C, et al. IMIX: in-process memory isolation extension [C]// The 27th USENIX Security Symposium. Baltimore, USA: USENIX, 2018:83-97.

[25] ClangLLVM. Clang: a C language family frontend for LLVM [EB/OL]. [2022-10-21]. https://clang.llvm.org:LLVM.

[26] MUSLLIBC. Musllibc[EB/OL]. [2022-10-21]. https://musl.libc.org/: musl-libc.

[27] HENNING J L. SPEC CPU2006 benchmark descriptions [J]. ACM SIGARCH Computer Architecture News, 2006,34(4):1-17.

[28] KUZNETZOV V, SZEKERES L, PAYER M, et al. Code-pointer integrity[M]. New York: Association for Computing Machinery, 2018:81-116.

[29] OLEKSENKO O, KUVAISKII D, BHATOTIA P, et al. Intel MPX explained: a cross-layer analysis of the intel mpx system stack[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2018, 2(2):1-30.

A fine-grained security domain isolation mechanism based on software and hardware cooperation

LI Yawei, ZHANG Longbing, WANG Jian

(State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

(University of Chinese Academy of Sciences, Beijing 100049)

Abstract

Attacks based on memory tampering can maliciously modify key data in the program execution environment. Providing a safe and reliable execution environment for programs is an effective means of suppressing malware. In this paper, a solution based on software-hardware collaboration is proposed, which can provide a relatively isolated and secure execution environment for program execution with function calls as granularity. In order to cooperate with the software, two hardware supports are provided at the bottom layer. First, the load/store instruction must perform address check when accessing memory, and the access attribute is also set. Second, add function call isolation domain (CFID) on the hardware page table, which is checked for security isolation during TLB conversion. Sharing access strategies are provided in two different occasions. The prototype system on GEM5 is implemented, which can effectively isolate the non-secure environment by running the secure test set. Compared with virtual machine and privilege level switching methods, the hardware implementation has almost no switching overhead. In the test set of SPEC CPU 2006, the overall performance loss of the hardware isolation mechanism proposed in this paper is only 3%.

Key words: fine-grained, isolation execution, hardware security, software and hardware cooperation