第46卷 第1期 计 算 机 学 报 Vol. 46 No. 1
2023年1月 CHINESE JOURNAL OF COMPUTERS Jan. 2023

基于软硬协同的程序运行时安全保护机制

李亚伟 章隆兵 张福新 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 内存篡改（Memory Corruption）是现代各类攻击的主要原因，通过修改内存中的数据，达到劫持控制流的目的. 使用不安全语言暴露内存细节给开发者，导致很多的敏感数据可以任意被修改. 现有的解决方案针对安全攻击主要包括两个方面，软件检查和硬件机制保护. 基于软件检查的机制虽然灵活，但是存在严重的性能问题. 基于硬件的方法可以大幅度解决性能问题，而且要比软件的方式安全性更高. 因此提出了很多的硬件相关的保护机制. 但现有的硬件机制大都仅仅针对单一的攻击，而且缺乏灵活性. 在本文中，我们提出了一种软硬件结合的解决方案，通过对程序运行时敏感数据进行加密隐藏，在访问这些敏感数据时进行解密，然后做安全检查，判断敏感数据是否被修改. 在硬件实现上，本文设计了安全的Load和Store类指令，以及硬件加密解密模块. 同时在软件编译器上对此类安全指令支持，针对不同的使用场景提出了两种安全策略：全局约束策略和上下文执行约束策略. 相比于前者，后者提供了更加严格的约束，可适用于安全度更高的程序保护. 本文的安全机制能够抵御多种攻击向量，比如针对CFI类攻击，最近的DOP攻击，GOT表和虚函数表指针感染攻击等. 还可以抵御缓冲区溢出类的攻击,支持信息隐藏等. 通过SPEC2006的测试程序表明,本文提出的安全机制性能损耗仅仅为4.5%.

关键词 内存篡改；敏感数据；加解密；运行时保护
中图法分类号 TP393 **DOI号** 10.11897/SP.J.1016.2023.00180

A security protection mechanism on program runtime based on software and hardware cooperation

LI Ya-Wei ZHANG Long-Bing ZHANG Fu-Xin 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 Memory corruption is the root cause of modern attacks. The purpose of hijacking the control flow is achieved by modifying the data in memory. Using unsafe languages to expose memory to developers results in a lot of sensitive data that can be modified arbitrarily. The existing solutions against security attacks mainly include two aspects, software-based and hardware mechanism protection. Although the software-based mechanism is flexible, it has serious performance overheads. The hardware-based method can greatly cut down the performance loss, and it is safer than the software method. Therefore, many hardware protection mechanisms have been proposed. However, most of the existing hardware mechanisms only target a single attack and lack flexibility. In this paper, we propose a solution that combines software and hardware by encrypting and hiding sensitive data when the program is running. The

收稿日期：2021-07-30；在线发布日期：2022-07-18. 本课题得到中国科学院先导课题“桌面处理器软硬件协同性能增强技术（No. XDC05020100）”资助. **李亚伟**，博士研究生，主要研究领域为操作系统、硬件安全、处理器架构和编译器优化. E-mail: liyawei19b@ict.ac.cn. **章隆兵**（通信作者），博士，副研究员，博士生导师，主要研究领域为计算机系统结构和微处理器设计. E-mail: lbzhang@ict.ac.cn. **张福新**，博士，博士生导师，主要研究领域为处理器微结构设计、性能分析和操作系统. **王 剑**，博士，研究员，博士生导师，主要研究领域为并行编译、嵌入式操作系统、处理器结构设计等领域.

mechanism decrypts these sensitive data when accessing, and then does security checks to determine whether the sensitive data has been modified. We design secure Load and Store instructions in terms of hardware implementation, as well as encryption and decryption hardware modules. Software compiler supports such security instructions. At the same time, two security strategies are proposed for different usage scenarios: global restriction strategy and context execution restriction strategy. Compared with the former, the latter provides more stringent constraints and can be applied to program protection with higher security priority. Our security mechanism can resist a variety of attack vectors, such as CFI attacks, recent DOP attacks, GOT table and virtual function table infection attacks, etc. It can also mitigate buffer overflow attacks and support information hiding. The experiments of SPEC2006 show that the performance overhead of our proposed safety mechanism is only 4.5%.

Keywords memory corruption; sensitive data; encryption/decryption; runtime protection

1 引 言

现代的底层软件栈，系统服务，高性能计算等大都是采用 C/C++编程语言. 而这些系统级的编程语言是非安全性的，它们暴露大量的底层相关的细节给开发者，这样的编程模式一方面增加了程序语言的灵活性，接近底层也带来了较高的性能. 但是更多底层细节的暴露给程序的安全带来了巨大的隐患. 尤其现在的高性能软件，比如 Linux 内核、基础软件设施等等. 大都是采用这些高性能语言编写. 因此在保持不改变编程语言特性和软件兼容性的前提下，采用底层硬件和编译器来增强程序安全性具有很大的实践性的意义.

然而这些非安全语言导致的内存篡改是现代各种安全类攻击产生的主要原因. 为了抵御这些攻击（Attacks）或者漏洞（Bugs），学术界提出了许多应对的机制和保护手段. 主要分为两类：一类是基于使用编译器动态的生成检测代码，在运行时检测比对是否被修改，比如 StackGuard[1]、Soft-Bound[2] 、文献[3]和在商业编译器上实现的地址清理（Address Sanitizer）[4]机制等. 另一类则是通过底层硬件来防御攻击，比如基于硬件的监视器（Monitor）研究[5-8]和基于硬件策略(Policy-Based)的机制[9-13]. 软件的实现方式虽然灵活，但是存在严重的性能开销[2,4]. 硬件能够克服软件上的性能问题，但是同样存在一些比较明显的缺点. 比如策略的方法需要硬件管理元数据(Metadata)，开销相对较高，设计较为复杂. 其次现有提出的硬件机制大都是应对单一的攻击[14-17]，比如缓冲区溢出，返回地址保护等等，无法抵御更多的攻击向量.

为了在利用硬件机制优势的同时，克服硬件的局限与挑战，本文提出了一种基于程序运行时的保护机制，在小程度修改硬件的基础上，能够提供更加细粒度的内存字级（Machine Word）的保护. 我们采取了多种方式解决硬件上的问题. 第一，为了减少元数据管理带来的负担，我们直接在数据所在的内存位置上，采用两种不同的安全策略，来满足不同的安全约束场景. 一种是较为宽松的策略（全局约束），使用每个进程全局的 Key（Kg）来加解密敏感数据（包括返回地址，指针，涉及控制流转移的临时数据和全局数据）. 这个 Key（Kg）在进程初始化的时候设置，对用户程序来说，是透明的无法读取与修改. 另一种策略是强安全策略（上下文执行约束），在每个调用的函数过程中，会有不同的 Key（Kc）产生. 而在当前函数的执行环境中，涉及到的敏感数据都是使用 Kc 来加操作（除过特殊情况，文章中相关章节有详细说明）. 第二，在数据的所在的内存位置操作. 这样设计的好处既避免使用影子栈带来的额外的存储空间，也较少了在多线程程序中数据共享而引起的影子数据拷贝带来的开销. 第三,减少了硬件设计上带来的复杂度. 本文的设计很大程度上与原来处理器硬件结构上解耦合，只需要添加相关的译码控制逻辑和数据通路实现安全模块与处理器的结合. 不用为处理元数据而增加处理器寄存器的位宽[18,19]和修改处理器的 Cache 存储层次结构[20,21]，也不用增加额外的数据结构和硬件维护机制[9,18,22-24]. 本文提出的保护机制在并不需要修改源代码，只需要重新编译即可.

我们提供了一套完整的解决方案，包括添加补丁的 Clang/LLVM 编译器，支持前端敏感数据的识别与后端代码的生成；C/C++运行时库和 Linux 内核的支持. 本文的设计的主要优势在于：（1）不用复

杂的元数据管理，减少性能损耗和编译器设计复杂度；（2）不需要开辟大块的专用数据空间（没有影子栈）；（3）尽可能的与处理器结构设计解耦合，易于与现有的设计集成；（4）能够抵御多种攻击向量，比如 CFI，ROP，DOP，缓冲区溢出，信息隐藏等；（5）提供指令字级粒度的硬件保护机制.

我们在模拟器 Gem5[25,26]上实现了设计的原型，采用 AArch64 架构. 编译器使用 Clang/LLVM,以及相关的一系列工具. C 库使用的是 Musl libc, 简单易用，而且可移植性较好. C++库使用的是与 Clang 配套的 libc++. 由于 GNU 的 glibc 与 GCC 的相关度较高, 因此我们不使用 GNU 的一些运行时库, 而是采用新的一套软件方案.

为了评估设计的性能开销，我们使用了 SPEC 2006 测试程序来分析使用安全机制带来性能上的损耗. 我们也在第六部分详细的评估了此安全机制的在微结构上产生的性能影响. 评估结果显示本文提出的保护机制性能损耗仅仅只有 4.5%.

本文的贡献如下所述：

（1）设计（Design）. 详细的阐述了我们的安全机制原理和动机，提供了完整的一套解决方案，包括不同场景下提供的两种策略，以及如何更好的利用我们提出的保护机制.

（2）实现（Implementation）. 我们在 Gem5 平台上实现了底层硬件，增加相关指令和安全模块. 扩展了 Clang/LLVM 编译器，增加了编程接口，能够产生特定的指令和程序运行时库的支持.

（3）安全分析（Security Analysis）. 为了全面的评估我们设计方案的安全性，首先我们分析了如何具体地抵御各种相关攻击. 使用不同的安全约束策略，我们也给出了一些针对敏感数据的保护方案. 其次我们移植了 Juliet C++和 RIPE 安全测试样例来评估设计的安全性.

（4）评估（Evaluation）. 我们使用 SPEC 2006 测试集评估性能开销，从底层的微架构方面，分析了安全机制带来的性能损耗.

（5）编程指导（Programming Guide）. 为了更好的使用安全机制，我们也提供了一些编译选项和相关说明，提供了较为完整的编程指导，以方便开发者能够容易地移植和使用安全机制.

文章剩余的章节如下组织. 第二章比较了现有的相关实现技术. 第三章节定义了攻击模型和相关的假设. 第四章节详细地阐述了设计的动机，敏感数据的分类以及机制的整体的框架. 第五章节主要描述设计具体的实现细节，包括底层硬件指令集的支持，上层的软件的支持. 第六章节给出了如何抵御攻击的具体例子，对安全性做了详细地分析. 第七章节评估安全测试和性能开销.

2 相关工作

2.1 基于纯软件的方法

软件的实现方式动态的利用编译器的灵活性，在程序的编译期间，产生保护的相关代码. 在运行过程中会检测这些保存属性信息来判断程序是否恶意的修改了安全区域. 比如最早期为了防止栈内容的可执行，提出了 NX(non-executable)技术，现在的大多数体系架构都沿用这项保护措施. 为了保护缓存区溢出采用软件设置简单的标志来检查程序是否越界篡改了其他内存. StackGuard[1]利用编译器生成代码时，在返回地址（Return Address，RA）相邻的区域加入一个加入随机字符（Canary Word），攻击代码修改返回地址会导致随机字符更改，在函数返回时检查是否修改. Soft-Bound[2]也是使用编译器为每个指针产生相关的元数据，元数据保存着基址和边界信息. 在每次使用指针的时候都要访问元数据. 它有个必须要满足的条件是：指针传递的时候，元数据也需要传播. 这就带来了性能上的损失. 文献[3]提出了新的内存错误的检错方式，取得了较先前研究更低的性能损耗. 还有其他的研究如 Valgrind[27],主要采用动态二进制插桩的方式保证内存免于恶意修改. 而较新的应用于商业软件 GCC/Clang 的保护机制地址清理（AddressSanitizer）[5]，采用一种特殊的内存分配器（Memory Allocator）和代码植入的方式，能够检测针对于数据越界访问（Out-of-Bound）的安全问题，但是取得灵活性的同时也带来了 73%的性能损耗.

2.2 基于硬件支持的机制

先前基于软件的保护方式具有较强的灵活性，大多不需要修改源代码就可以检测抵御某些安全攻击，但是有两个重要的问题就是较高的性能损耗和软件的可绕过性. 因此提出了较多的基于硬件的保护方式. 如基于硬件的监视器（Monitor）研究[5-8]，主要借助硬件监督模块，检查非安全的修改. 这种方式追踪指令级粒度的操作，利用在流水线阶段集成或者提供对外的监视器接口的方式. 而 SDMP 等相关研究[9-13]采用较为复杂的基于策略的保护机制，编译器按照规则生成相关的元数据，硬件根据这些属性策略来确定访问的合法性. 这些研究方法需要保存和维护较为复杂的策略. 最近的一些相关

研究[14-17]提供基于域隔离保护的硬件机制，借助 Intel MPK，MPX[17,22]等技术，实现安全区与非安全区的隔离．但是保护的粒度较大，很难提供更加细粒度的安全保护．

表 1 展示了已有相关技术的各种细节比较．软件的方式一般采用动态插桩的方式，在大多数情况下不用修改代码，使用打上相关补丁的编译器重新编译就可以使用，如文献[1,2,5,28-31]等．这种方式支持抵御的攻击也比较多像 CFI、ROP、BO、BC、UAF（Use-after-free）等．但是存在一个比较明显的问题就是性能开销比较大．硬件的方式，抵御的攻击相对较少，但是有个明显的优势就是性能开销较小，如文献[20,21,32-36]等．

虽然基于硬件的方式有很大的优势，但是还是有较多的局限．基于策略（Policy-Based）的硬件保护机制[9,18]虽然使用硬件来加速元数据的分析(读取和策略选择)，但是在策略规则的生产和管理上较为复杂，带来了很大的不确定性．另外也要注意策略的保存，以免恶意攻击程序篡改．而文献[20,21]等通过对内存的细粒度的标记，达到追踪内存是否被修改的目的．但同时也要修改复杂的 Cache 层次结构来支持内存标记(Memory-Tagged)．基于 MPX[22] 和 MPK[14-17]的商业解决方案，较多的用于应用程序的安全隔离，应对的攻击有限，大多数无法提供细粒度的保护．而影子栈类[32]的保护，只是针对抵御较单一的 ROP，CFI 等攻击．

表 1　存在的安全机制比较

分类	机制名称	架构(1)	主要目的(2)	多线程(3)	元数据(4)	信息隐藏(5)	细粒度保护(6)	自我保护(7)	影子数据(8)	性能开销(9)
基于软件的保护机制研究(Software-Based)	Randomization[31]	ALL	ROP	低	否	否	否	否	否	0.28%
	StackGuard[1]	ALL	BC	低	否	否	否	否*	否	2.8%
	CPI/CPS[30]	X86/64	CFI，ROP	较高	是	否	是	否*	是*	8.4%
	BGI[28]	X86	Isolation	中等	否	是	是	NA	否	接近 16%
	Soft-Bound[2]	x86-64	BC	中等	是	否	是	NA	否	67%
	Address Sanitizer[4]	ALL	BC	高	是	否	是	NA	是	73%
	XFI[29], SOFIA[34]	X86	CFI	低	否	否	否	NA	否	高
基于硬件的保护机制研究(Hardware-Based)	Shadow Stack[32]	X86	CFI，ROP	较高	否	否	否	是	是	3.5%
	HardBound[37]	X86	BC	低	是	否	是	NA	否	5%~9%
	MPX [22]	X86	BC	中等	是	否	是	NA	否	接近 50%
	MPK[14-17]	X86	Isolation	中等	否	是	否	NA	否	较低
	HE[36]	SPARC	Memory-Tagged	低	是	否	否	是	否	较低
	HDFI[21]	RISCV	Isolation	低	否	是	是	NA	否	接近 2%
	WatchDog[18]	X86-64	EMM*	高	是	否	是	是	否	29%
	文献[12,13]	X86	Metadata	低	是	否	否	NA	否	33%
	SDMP[9]	Alpha	Stack Protection	高	是	否	否	是	否	接近 5.7%
	STT[33]	—	SE	中等	否	是	否	否	否	接近 14.5%
	HAFIX[35]	SPARC	CFI，ROP	低	否	否	是	否*	否	接近 2%
	ADI[20]	SPARC	Memory-Tagged	中等	否	是	是	否*	否	低*
	AOS[24]	ARM64	BC	高	是	否	是	NA	否	8.4%
	本文	ARM64	Encryption	低	否	是	是	是	否	接近 4.5%

注：（1）架构是指支持的平台，ALL 表示在软件编译器实现，可支持多种的方式，-表示文献中未提及．（2）控制流完整性攻击（Control-Flow Integrity，CFI），ROP（Return-Oriented Programming）．隔离（Isolation）主要指进程内安全数据的隔离防御．边界检查（Bound Checking，BC），主要是防御缓冲区溢出．推测执行（Speculative Execution，SE）攻击可以推断程序中的敏感数据． 强制内存管理（Enforce Memory Management，EMM）对内存的访问设置较强的限制．（3）表示在多线程中共享数据和进程切换的代价．（4）表示是否需要管理元数据．（5）表示是否支持隐藏信息．（6）表示保护的粒度．（7）表示有无针对自身的安全保护，NA 表示未提及．*表示大多数文献并没有具体的分析其自身的安全性，只是假设不考虑某种情况，本文认为是不够安全．（8）表示是否使用影子栈或影子数据区．（9）表示带来的性能开销，有些文献没有给出具体的数值，本文沿用文献中的表述．

2.3 基于加解密算法的机制

与本文设计方案类似的安全机制有 ARM 的 PA 和 CCFI[38]. ARM64 提供了一种新的安全机制称为指针验证(Pointer Authentication, PA). 主要是检查指针的完整性. 当处理器执行 AArch64 状态时, PA 机制的底层硬件指令会产生授权码(Pointer Authentication ,PAC)来保证指针的完整性, PAC 的实现底层也是使用相关加密算法完成. 在使用返回地址保护时, 我们的设计方案可以达到几乎和 PA 近似的性能损耗[24].

与 PAC 相似的机制还有文献[38]提出的 CCFI, 它和 ARM64 PA 的实现方式一样, 主要是保护返回指定和控制流控制, 不同于 PAC 的是, 它使用硬件加速 AES 算法来产生 MAC. SPEC2006 的测试结果展示, 其平均带来了 52%的性能损耗. 在使用加速指令时也有接近 18%较高的性能代价.

我们的实现机制在针对指针保护时, 类似于 PAC 机制. 但我们的机制相对于 PAC 有较大的不同. 首先, 我们提出了针对非控制流数据的保护, 这并不仅仅是对指针的保护,以及针对 ROP 变体的保护. 其次, 我们提出了两种不同的安全保护策略, 针对不同的场景, 使用不同的安全约束. 另外, 我们的机制还有针对缓冲区溢出, 信息隐藏的安全应用场景. 相对而言, 使用的范围更加的广泛, 应用更加的灵活.

3 攻击模型和假设

我们涉及的攻击类型主要是软件层面相关的攻击, 而使用物理的攻击方式, 比如侧信道攻击, "熔断"与"幽灵", 硬件木马等不在本文的涉及范围. 在软件与硬件的约定上, 我们假定, 底层的硬件是安全的, 不存在可利用的漏洞.

我们假设攻击者拥有可以任意地读写修改内存的能力, 这些内存标志为可读可写的. 攻击者无法修改只读和可执行的内存(比如代码段). 假设攻击者不能够获得系统的更高级权限, 无法读写某些特权级的寄存机和数据区. 其次我们提出的数据保护的策略在多线程共享的环境中同样适应. 我们目前的叙述只是以单线程的视觉角度解释原理和讨论情况.

为了更好的演示安全测试用例, 我们只是模拟主要的攻击过程, 并不是接近真实的安全攻击. 我们手动的在代码中注入非安全的代码或者数据, 重新编译, 然后测试我们的安全机制是否能够抵御攻击. 包括对二进制可执行文件的修改, 为了能够使用安全测试样例, 我们也修改了测试文件中和架构相关的部分. 其他的系统级、libc 库、执行环境我们都认为是安全的.

4 设 计

内存篡改相关的漏洞其实质主要是针对一些敏感的数据像指针类, 控制流转移类数据没有保护. 已有提出的相关安全机制也都是针对此方面设计特殊的保护措施. 如软件的动态插桩审查, 硬件的一致性比对等. 我们在处理器中增加了安全模块, 这个安全模块在动态过程中对这些敏感数据在内存中加密, 在读取的时候再进行相关的解密, 通过检查设置的相关属性是否被修改, 进而达到保护的作用. 下面我们首先会阐述设计的相关挑战与目标, 以及如何克服这些问题. 其次描述总体的设计架构, 包括软件层面与底层的硬件设计.

4.1 设计动机

为了保证程序动态执行过程中, 执行上下文的完整性(Integrity), 避免被恶意的修改, 达到安全保护的效果. 同时确保我们设计本身的安全性与完备性. 我们主要面临以下三方面的挑战:

挑战 1. 如何区别程序中的敏感性数据.

挑战 2. 如何保证执行过程中敏感数据的一致性.

挑战 3. 如何安全的管理加解密过程中的秘钥 Key.

下面我们会逐一详细的说明我们的设计是如何克服和解决这些问题的.

4.2 程序的敏感性数据

程序在执行过程中需要保存变量和存放运行时的一些重要数据结构, 这些数据结构可以影响程序的控制流转移. 目前很多较为先进的攻击都是利用修改这些关键的敏感数据. 比如 ROP 攻击, 通过覆写返回地址, 构造一些列的执行链, 然后跳转执行到攻击代码. 另外比较新的 DOP(Data-Oriented Programming)[39,40]的攻击则是利用临时变量, 不用改写指针就可达到攻击目的, 这类攻击称为非控制流数据攻击(Non-Control-Data Attacks). 如图 1 所示.

这是常见的用于更新链表中成员变量的代码示意. 提供给 foo_req 函数的参数指针 req_ptr, 未被劫持. 但是它保存在了栈内(第 6 行). 很可能被攻击者劫持(比如缓冲区溢出等手段)覆盖了 req 的内容. 之后将 req 的地址重新付给了 ptr. 假如攻击者伪造一些列的 Req 数据链. 在第 9 行, 控制循环的

次数和重新赋值指针更新其所指向的内容. 首先，循环次数可以控制. 其次，条件变量 if 判断控制是否修改内存. 即使 size 是很小的值，通过控制循环次数，也可以修改为任何想要的值. 因此攻击者可以利用上述代码可以向任意的地址上写任意的值. DOP 证明是图灵完整的[35].

```
1:   struct req { struct req* req;  int count, type; }
2:   void foo_req ( struct req *req_ptr, int size )
3:   {
4:   struct req req;
5:   struct req *ptr;
6:   req = *req_ptr;
7:   ptr = &req;
8:   ...
9:    for (; ptr != NULL; ptr = ptr->req){
10:       if ( ptr->type == ADD )
11:              ptr->count += size;
12:       if ( ptr->type == NONE )
13:              break;
14:    }
15:    ...
16:    return;
17:  }
```

图 1　DOP 攻击代码片段（Gadgets）示例

还有比如函数的返回地址，全局函数的 GOT 表，指针变量，虚函数表等控制程序执行的改变. 因此我们定义敏感数据为影响控制流转移的内存对象（Memory Object），以下我们简称敏感内存对象. 敏感内存对象主要分为以下几大类：

（1）部分全局的数据变量，主要包括一些指针变量和一些非控制流数据. 这类敏感内存对象主要是执行过程中能够影响函数的执行流转移（如条件判断）.

（2）与栈相关的部分局部数据，包含临时的非控制流数据、指针类数据，返回地址、帧指针等.

（3）全局偏移表（GOT），保存全局函数指针.

（4）虚函数表（Vtable），存放 C++虚函数指针相关结构.

这几类敏感数据是造成程序感染和安全性问题的主要原因，很多文献提出了相应的解决方案，但大部分都是针对某一类单一对象，并不能保护更多的敏感数据分类. 因此我们的目的是尽可能多的保护这些对象免于篡改.

4.3　执行上下文一致性

对于挑战 2，保证数据一致性有较多的实现方式. 比如借助编译器将数据备份保存，然后执行完再检查，比对异常则检查失败，触发异常[3,5]. 这种方式灵活，易于实现，但是存在严重的性能问题. 正常情况下，对于一个数据的访问，需要多执行保存、读取、检查这三个步骤. 同一个变量的读取需要访存两次. 这样的方式带来较大的性能开销.

其次是使用类似于影子栈的结构. 这种做法是将上述的变量读取过程中多余三个步骤用硬件来实现. 这样减少了软件指令的指令数，如果对于频繁访问的变量，这样的做法还是比较可观的. 虽然减少了指令，但是实际的操作由硬件实现. 还是需要额外的空间存储，在访问变量的时候，还是要读取，虽然没有了指令，但还是增加了底层的内存带宽.

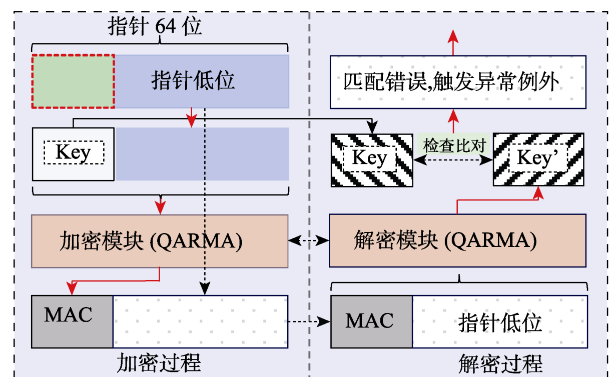

图 2　指针加密示意图

我们的采用加密的方式来隐藏（Cloaking）数据，分别检查加密解密后数据所带的标签属性（Taging），来判断数据是否被其他的函数更改. 这样做的优点在于减少访存的同时能够借助编译器的灵活性，实现抵御多种类型的攻击. 但是同样存在两方面的问题需要解决. 敏感内存对象包含一般的非控制流数据和指针类，这是两类不同的内存对象.

4.3.1　指针数据对象

对于指针，在 64 位系统中，由于现有的体系结构无法全部使用所有的地址空间. 因此指针中有些位是没有用到的，为了节省内存空间，我们将加密后的属性值，保存在最高位的保位置中. 这些加密后的属性的我们简称为 MAC（Message Authentication Code）. 具体的执行流程如图 2 所示.

下面图 3 显示了使用该机制后，对虚实地址转换的影响. 在 ARM64 架构中，地址位宽为 64 位. 其中最高位 VA[63]指示选择内核页表还是用户页表. 在 Linux 中，通常配置页大小为 4KB，3 级页表，39 位虚拟地址，48 位物理地址. 虚拟地址 VA[62:39]实际在虚实地址转换中没有用到. 即使使用 16KB 和 64KB 页大小时，也有 VA[62:47]的位置可以使用. 因此可以将保存在保留位，只需要在转换时，将保

留位置补全，以满足 ARM64 架构的虚拟内存系统规范[40].

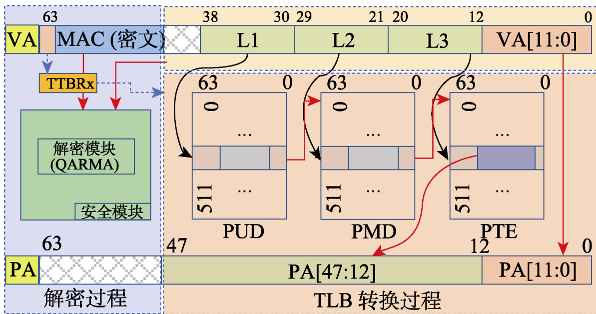

图 3 虚实地址转化图

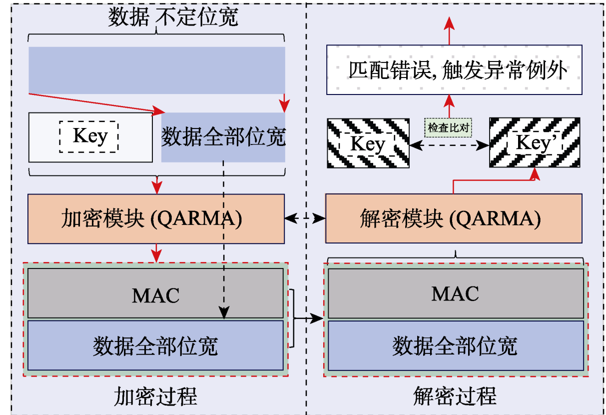

图 4 数据加密示意图

4.3.1 非控制流数据对象

对于一般的非控制流数据，情况稍微有些复杂. 不同于指针，我们不能保证数据的哪些位宽是保留的没有意义的. 为了普适性，我们并不规定数据存储的位宽，不局限于特定的数目. 我们假定数据的所有位宽是都有用的. 因此为了减少设计的复杂度，我们采用扩充数据位宽的方式，保存加密后的属性值. 我们按照表 2 的规则增加位宽.

表 2 数据位宽扩充规则

数据类型	原始所占内存字节数	扩充后所占内存字节数	特殊情况说明
Bool	1	2	MAC 占 1 个字节
Char	1	2	MAC 占 1 个字节
Short	2	4	MAC 占 2 个字节
Int	4	8	MAC 占 2 个字节
Pointer	8	16	MAC 占 2 个字节
Long	8	16	MAC 占 2 个字节
Array*	-	-	后面章节特殊说明

注：Array 的 MAC 保存按照程序的实际进行保存.

需要说明的情况是，对于 Bool 类型的数据，在判断控制转移的情况时（if-else 条件中），结果只有两种，要么 0，要么 1，这样的话，攻击者可能会采用猜测的方式覆写加密属性，这样有 50%可能性推测正确.

因此我们在动态中增加 7-bit 的信息扩展成 char 的方式来实现. 这 7 位是来自程序运行中函数的 ID，因此也不会被准确的推测. 而 Array 的 MAC 的保存主要有两种，一种是数组中的每一个元素按照上面的规则扩充. 或者是采用统一的 MAC，保存在内存中. 由编程者根据提供的编译选项来控制采用那种方式. 执行流程如图 4 所示，编译器在编译代码时，会自动地分析和扩充位宽，预留多余的位置.

4.4 安全策略

我们使用 MAC 码来隐藏保护内存对象的属性. 它采用 QARMA[41]方式产生（见第四章节讨论），在执行过程需要指定一个特定的 Key. 在读取相关数据也就是解密的过程，需要恢复和比对，来判断是否被修改.

由上面针对不同数据和指针的分析可见，如何避免攻击者恶意的窃取关键的 Key 是本设计自身安全的核心. 这也是挑战 3 需要解决的问题.

首先我们从考虑 Key 的保存和恢复. 通常的做法是将其产生的 Key 保存起来，然后再在需要的时候读取出来. 但是这样做存在以下几方面的问题：（1）空间、带宽增加：Key 的保存需要申请额外的空间，这就增加了 CPU 带宽、访存时间以及内存空间. 这对于嵌入式环境来说，代价相对较大. （2）安全性降低：保存和读取 Key 需要指令来操作，但是这些指令攻击者也可以使用（比如 Code-Reuse 类攻击）. 这样的话，程序没有任何的安全性. 造成这样的其实质问题就是操作没有绑定特定的执行上下文，没有确定执行的权限. 这导致任何的函数可以访问和修改. 另一种方式是按照一定的方式动态的生成 Key，这样做的好处在于不需要保存 Key，只需要在需要的时候按照一定的策略产生. 这些策略需要和函数的执行上下文关联，确保执行的唯一性. 根据不同的需求，我们提供两种不同的安全策略. 下面我们具体讨论这两种安全策略.

（1）全局约束

最简单直接的方式就是所有的加密都使用一个全局的执行策略. 也就是说，在函数执行初始化之前，设置全局的安全 key，然后在后期的函数执行中，所有的加密解密都使用它. 这样做的好处是实现简单，而且在涉及 setjmp/longjmp 或者 C++的异

常时，解栈（Stack Unwind）需要处理的情况单一，不需要编译器插入复杂的判断逻辑．在每个程序初始化的时候，由内核的程序加载器（ELF Loader）为每个进程创建一个 GKey．为了保证其运行时的安全性，我们设置以下执行约束：

约束 1．由程序加载器初始化，这是随机的，独立的，与其他进程不一致．

约束 2．GKey 保存在一个 CPU 内部寄存中（非通用寄存器）GKCR（GKey Context Register）中．

约束 3．只有特权级才能访问和修改 GKCR 寄存器，一般的用户指令无法读取和修改此寄存器．

（2）上下文执行约束

另外一种策略比全局约束更加的严格，它的主要思想是将 Key 的产生与函数的执行上下文绑定，只有处于当前的函数执行上下文中，安全模块（QARMA）的解密操作才能解析出正确的值，以及相关的 MAC 内容．

其操作流程如图 5 右部分所示．为了保证策略的正常执行，需要满足以下约束：

约束 1．当前函数 CFID 寄存器用户不可读不可写，对用户而言是透明的．在程序加载的时候由加载器（ELF Loader）初始化设置，特权级可读可写．

约束 2．CCID 寄存器在使用前必须由加载器初始化，后期由硬件指令（Call 和 Ret 指令）自动维护更新管理，用户无特权修改．特权级可读可写．

约束 3．进入函数和退出函数的明确界限．使用 Clang/LLVM 编译器时，在 ARM64 平台，以 BLR 或者 BL 指令为进入函数，以 RET 为退出函数．编译器产生的代码规整．如果用户手动编写汇编代码时，也需要按照此约束．在后面第七节有详细的编程说明指导．

约束 4．上下文执行约束主要是针对用户级代码，使用编译器生成规则代码．在底层系统内核的时候，通常不满足上述约束，因此尽可能少使用此规则，避免产生意想不到的结果．

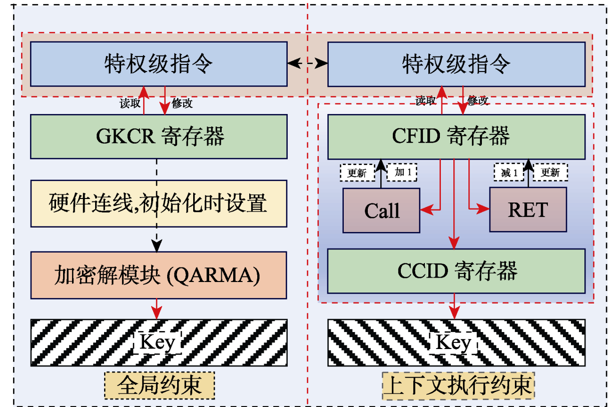

图 5　两种策略约束示意图

4.5　总体架构

系统的总体架构如图 6 所示，主要包括前端编译器对源程序的分析编译，然后硬件 CPU 安全模块支持生成的这类指令．包括运行时库以及 Linux 内核的支持．

（1）Clang/LLVM

我们扩展了 Clang/LLVM，增加了相关的编译指示参数．用户无需要修改源代码，只需要增加安全属性重新编译即可．Clang 在解析 C/C++源代码时，按照我们在 4.2 节中讨论的情况，自动的解析

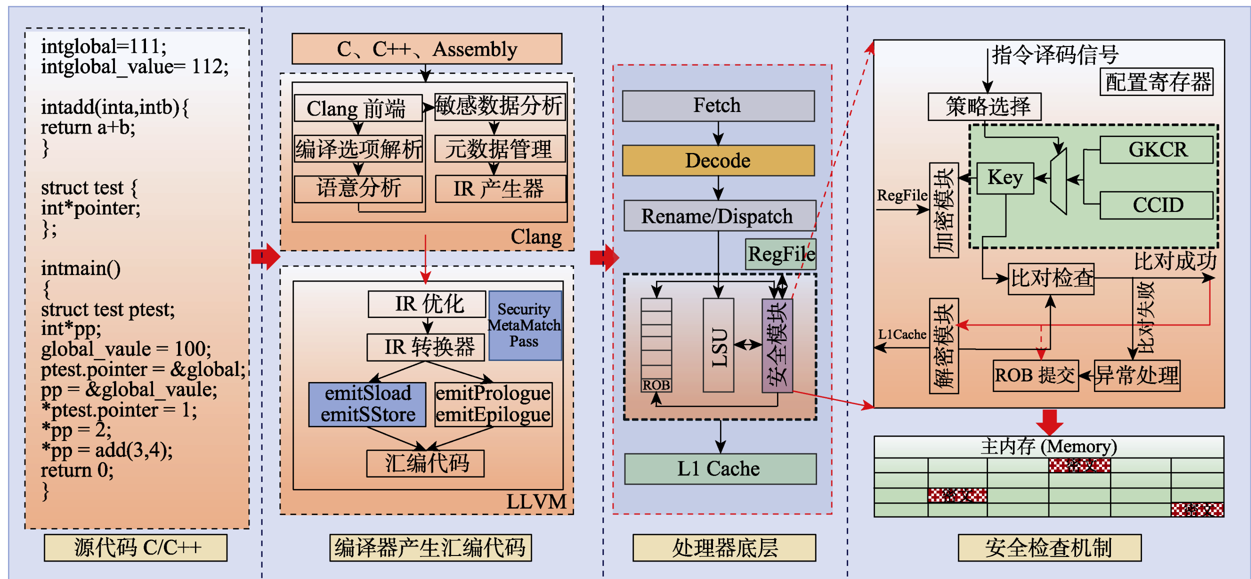

图 6　总体架构图

敏感数据，然后在生成的 LLVM IR 代码中带有安全的元数据标记.

在后端执行优化的阶段后，我们提供了 LLVM Transform Pass 的 Security Meta Match Pass，这个 Pass 处理前面对 IR 指令做的标记，在启用后端 Target Backend 生产目标指令时，按照用户提供的安全策略，如图 5 中所示，由 emitSload 和 emitSStore 函数用我们的新的安全类指令替换原来的指令.

表 3 Clang 编译器编译选项

选项	类型*	选项说明
-sc-ra	数据类型	对返回地址安全检查
-sc-if	数据类型	对 if 条件判断检查
-sc-sw	数据类型	对 switch 类型判断
-sc-pointer	数据类型	对指针安全检查
-sc-global	数据类型	对全局数据检查
-sc-policy-global	策略选择	使用全局约束策略
-sc-policy-context	策略选择	使用上下文执行约束策略
-aarch64-security-check	使能检查	打开安全检查机制

为了增加编程灵活性，我们提供了编译选项，指定对那些数据的访问需要安全的处理，以及采用哪一种安全策略. 如表 3 所示. 在一些特殊情况下，为了保障程序的正确执行，我们强制使用了全局约束策略，即使是编译器指定，编译器也会强制选择全局约束的安全类指令. 下文的安全分析章节中对这种特殊情况有具体的说明.

（2）安全模块

主要负责对需要安全检查的数据，执行相关的操作. 主要是实现了文献 AQRMA[41]的加解密算法. 根据前端的译码的指令，在数据从通用寄存器中写回到内存时，先经过安全模块执行加密，然后将加密后的数据写会到内存. 当访问数据时，从内存中加载出来，执行解密，然后写到寄存器. 安全模块也负责对相关寄存器，包括 GKCR，CCID，CFID 等的管理和维护.

（3）策略检查与异常处理

策略选择主要在编译的时候确定，由相关的指令附带策略信息，然后在指令译码时去查找相关寄存器组，再提供给安全模块，执行相应的操作. 安全模块如果检查匹配，则继续执行. 如果匹配不成功，则直接产生用户数据访问异常，交由操作系统处理，以避免内存敏感数据修改和泄漏.

5 实 现

这章节主要讨论实现的具体细节. 首先描述底层硬件实现的相关细节，包括指令集的支持、流水线的设计、Key 的产生和管理、加解密的具体实现. 其次软件如何支持这些硬件的操作，包括编译器的支持、运行时环境和底层内核对安全检查的支持.

5.1 硬件设计

（1）指令集支持

为了满足设计需求，我们增加了两类指令，特权级和普通用户级指令. 详细的说明如下.

① 特权级指令

特权级指令如附录 2 所示. 在程序加载时，根据 ELF 文件格式的标志符号(EF_ARM_ABI_SEC)，在进程的 task_struct 结构体中置位 SE_check 标志，同时内核会在加载完成后使能检查机制. 在执行 exit 系统调用后，关闭检查机制.

需要特殊说明的是，在进程调度的时候，我们会根据当前进程的 task_struct 判断，是否需要关闭安全机制，在切换到下一个进程时，检查是否需要打开此机制. 全部由此类指令实现. 用户无法使用此指令.

② 普通级用户指令

普通用户指令附录 2 所示. 我们沿用 ARMv8 指令架构中，同指令分类保留的指令编码. 需要解释的是，新增加的指令，除了上述的特权级指令外，剩下的都是 Load 和 Store 类指令（下文称为 Sload 类和 SStore 类）. SLoad 类指令在加载数据到内存时，按照安全策略，使用不同的 Key，解密后，将数据保存到通用寄存器中,SStore 指令操作类似. 因此，对程序员来说，一旦 SLoad 类指令执行完毕，通用寄存器中保存的就是对象的值，不许额外的操作. SStore 类指令执行完成后，内存中保存的就是加密之后的数据.

需要说明的是，SLoad 和 SStore 类指令有两位（不同类别的指令，这两位在不同的位置. 典型的是，Load/Store Pair 类指令在 Inst[11,10]，下面以这类指令为例），表示使用的是哪一种安全策略. Inst[11,10]=2’b00 时，使用全局安全策略. Inst[11,10]= 2’b11 时，表示使用上下文执行策略. 其余的保留做以后扩展. 编译器在编译源代码的时候，会按照指定的策略，在相应的指令位置硬件编码相应的策略.

（2）流水线设计

本文的硬件实现都是基于目前流行的 Gem5 模拟器[38,39]，它能够精确的模拟底层架构. 我们使用它来模拟底层 CPU 架构，选择 O3CPU 模型. 其主要的硬件架构如下图 7 所示.

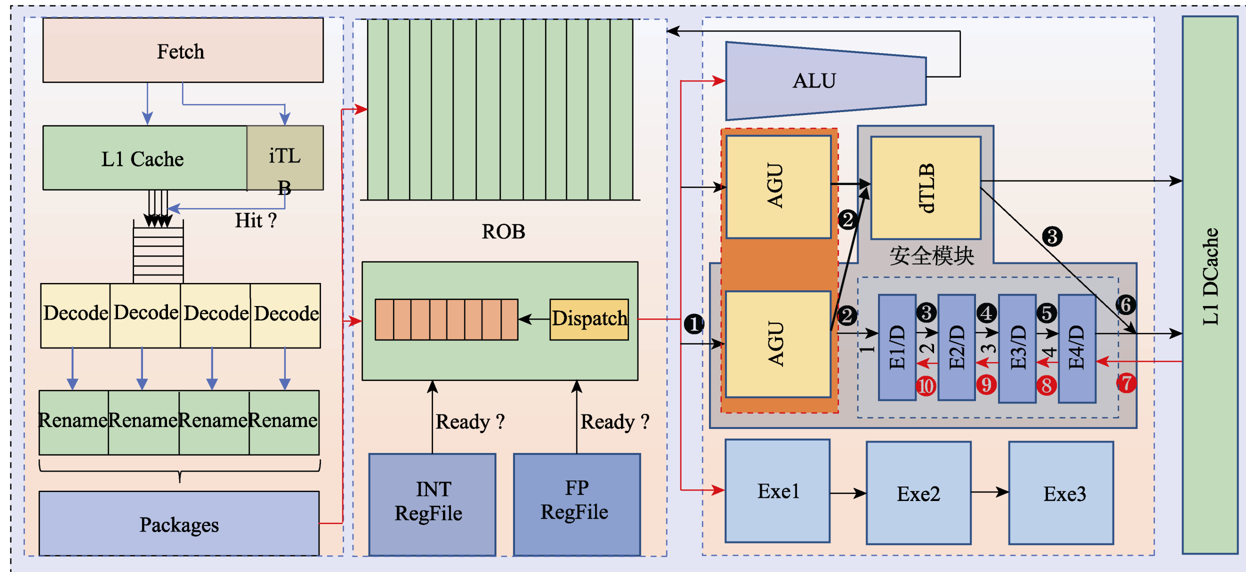

图 7　底层实现示意图

为了减少与其他部分的耦合度，我们尽可能提供统一的接口，来自其他部件的硬连线都在安全模块（图中灰色的部分）中使用. 根据前面的说明，SLoad 类和 SStore 类指令底层实质上主要分两个步骤：第一、类似于正常的 Load 和 Store 指令，将数据从内存加载到寄存器或者将寄存器的值写回到内存. 我们沿用 Gem5 的访存通路. 指令在译码之后，发射进入 ROB（Reorder Buffer，重定向缓存队列）等待指令执行完毕. 指令在译码阶段完成的时候，会产生安全信号 Security_Signals 和策略信号 Policy_Signals（这些信息保存在 ROB 中），然后进入发射队列. 等待发射到 LSU（Load Store Unit，访存单元）. 第二、前面获得数据后进入安全模块，执行具体的加密解密操作. 在本文的设计中这部分主要分 4 个流水线完成，如图中安全模块所示. 结束此阶段后，将 ROB 的相关标志清除，通知指令可以退出执行队列.

下面我们具体的分析安全类指令的关键执行流水线. 如图 7 所示：❶指令从发射队列获得所需操作数后，发射到 LSU，首先根据其操作码产生地址. ❷接着进入 TLB（Translation Lookaside Buffer，转移后备缓冲器），将虚拟地址转换成物理地址. 为了加速执行，在 SStore 类指令进行 TLB 转换的同时，将指令的源寄存器数据送入加解密模块. ❸到❺按照 Policy_Signals 的信息，生成不同的加密 Key，然后获得密文，将其送入 ROB 中. ❻在指令退出时写回到 Cache 中.

如图 7 所示，SLoad 类指令经过 TLB 后，向 Cache 获得数据，一般的 Load 类指令直接写回到目的寄存器，但是安全类 SLoad 指令要经过解密的阶段，此时根据 ROB 中此指令的 Policy_Signal 信息. ❼到❿选择相应的策略解密，然后将解密后的数据送回到寄存器. 为了实现精确异常捕获，此时指令还不能直接退出，在安全模块内解密之后，还要和具体的内容匹配，如果不匹配则产生异常，通知 ROB，执行异常流程.

（3）Key 的产生和管理

来自译码阶段的安全策略 Policy_Signal 信号，确定 Key 是来自于 GKCR 寄存器还是 CCID 寄存器. GKCR 寄存器由特权级指令 GKCRS 初始化，后期重新设置会触发 Configure Error 异常. 保证初始化一次. 但是在指令 GKCRC 之后，可使用 GKCRS 重新设置其值. 这样做的目的是为了保证其安全性，避免被程序恶意的重新设置.

CCID 寄存器的值由 CFID 寄存器硬件设置. CFID 寄存器类似 GKCR 寄存器，初始化时设置. 之后执行 Call 类（BL，BLR）指令时，使 CFID 的值增加 1，当执行 RET 指令时减 1. 具体的执行流程如上面图 4 所示.

有两点需要特别注意：（1）更新 CFID 寄存器时，需要 Call 和 Ret 信号的同时，还要满足 Privileged 信号（图中未画出）为 0. 这是因为考虑到一般的特权级代码（某些核心代码为汇编代码）Call 和 Ret 指令并不是成对出现. 因此我们只在用户态下使用.（2）在超标量指令乱序执行情况下，如果使用上下文执行策略时，更新 CCID 寄存器，需要判断后面执行的安全类指令是否与当前更新 CCID 的指令间有没有其他的更新 CCID 寄存器的指令. 在推测执行

时需要特别注意，避免使用旧值产生错误.

（4）安全模块的实现

加解密的实现采用 QARMA[41]的实现方法. 这是相对轻量级的加解密算法，灵活配置，可输出较短位宽的密文. 在不同的使用场合能够均衡延迟、面积和功耗，广泛应用于硬件实现[42,43]. QARMA 需要两个输入数据产生一个输出数据. 加密阶段，不同策略的 key 和敏感对象（数据，地址，指针）输入，输出一个不同位宽的密文.

在解密阶段，输出 Key′，比对当前的 Key. 如果匹配异常，则产生数据访问异常.

在片上资源或者功耗等严苛的条件下，上述的加密过程也可换成简单的函数. 此时加密解密都走统一的通路，如上面的灰色部分. 只是原先对 Key 的比对替换成对密文的比对. 因此来判断加密数据是否和原始的相同.

5.2 软件支持

为了有效的使用底层硬件，我们从下面三方面软件层做了适配，保障可靠的运行和提供统一的编程模型.

（1）编译器的支持

如章节 5.1 所述,我们在编译器 Clang/LLVM 前端增加对敏感数据的判断解析. 在 LLVM 后端添加指令替换 Pass，同时增加新安全类指令的支持.

（2）运行时环境

像 C/C++这样的编程语言，全局变量的初始化是由编译器直接写入 ELF 二进制可执行文件. 主要体现在 ELF 文件中的.data 段、.got 段等. 在读取这些变量时，往往使用的是解密类的指令，但是由于加密操作是在运行时决定的，因此编译器不能在生成二进制文件的保护时直接对其加密. 比如函数指针类，初始化的全局变量等. 为了满足程序的正确性，我们需要在主函数 main 执行前，增加对这些全局变量的初始化.

实现步骤如上面的图 8 所示，将全局数据用一般的普通加载指令读取到寄存器，然后强制性使用全局约束的安全策略，将敏感对象重新加密，然后写回到内存中. 下次访问时就直接按照正常的流程使用. 具体的实现是，在程序编译成可重定位文件的时候，会将全局变量的初始化代码写入.security.init.global 段中，然后 LD 链接器将这些段的入口地址指针统一放入.sec_ctor 段中，在 init 函数初始化的时候，执行这些初始化代码.

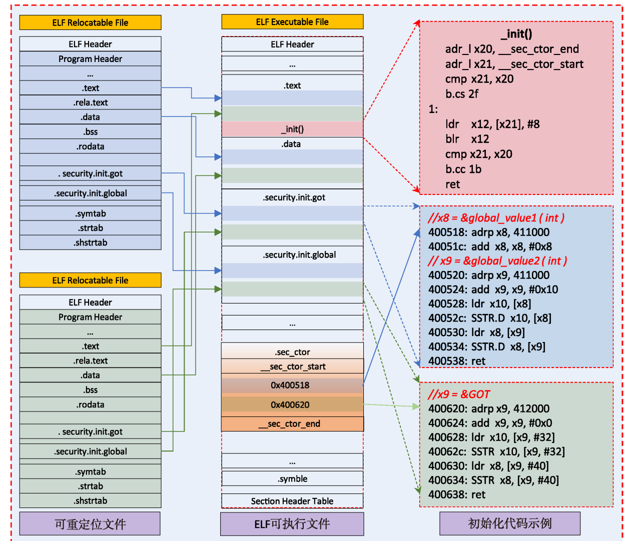

图 8 可执行 ELF 文件生成流程图

（3）内核支持

① 程序加载

为了简化程序启动流程，Clang/LLVM 编译器生成二进制 ELF 文件时，会在文件的 e_flags 标志中增加 EF_ARM_ABI_SEC（值为 0x06000000）标志，表示此二进制文件使用了安全检查机制. 因此 ELF 程序加载器在加载此可执行文件时，判断是否有此标志置位. 然后在内核在 start_thread_ common 函数中初始化 GKCR 和 CFID 寄存器，使能安全检查机制. 同时设置当前进程 task_struct 中的标志. 完成初始化，等待程序执行.

② 进程切换

进程切换时，判断当前进程 task_struct 的 SE_check 标志是否置位，如果当前进程使能了安全检查，①执行 GKCRR/CFIDR，保存寄存器的值. ②执行 GKCRC/CFIDC 指令，清空相关寄存器的值. ③判断下一个进程是否是使能安全机制. 如果不是，则执行 SECD 指令. ④调用指令 GKCRS/CFIDS 恢复当前进程相关寄存器.

③ 异常处理

我们在硬件上增加了安全异常的支持. 如果在安全模块检查失败，触发数据访问异常，硬件设置异常状态寄存器 ESR_ELx 中的 EC=0b111001. 执行硬件同步异常，然后交付给 Linux 内核中的通用异常处理函数 do_mem_abort 中判断，给进程发送信号，终止用户进程，回收资源等.

6 安全性分析

本章节主要分析安全机制如何能够抵御常见的

攻击．我们从以下六个方面分析其安全性，并结合相关示例代码做进一步的讨论．

6.1　ROP

在许多的安全场景中，通过破坏或者篡改指针成为首选的攻击向量．比如常见的 ROP 和 JOP[44,45]，通过挟持执行控制流，组建一系列的代码片段(Gadgets)，这些代码片段可以执行用户任意的操作．

根据我们提供的安全机制，我们在保存函数返回地址时，将其根据提供的策略加密．返回时从栈中读取时解密，根据高位保存的标签（ Tag ）判断是否被修改．这是最简单的实现方式．在函数进入时保存帧（ FP ）指针和返回地址（ LR ）指针，退出函数时检查．

但是针对一些 ROP 的变体，很多的方案则无法抵御．比如文献[43]中所叙述的．这类变体并不是简单的使用栈保存的返回地址，然后使用 ret 指令返回．如图 9 下半部分所示，它们的功能类似 RET 指令．常用的 ROP 抵御手段对这种攻击捉襟见肘．但是如果使用我们提出的安全机制，使用 SLDR 指令加载函数指针，则可以解决上述问题．

```
1: stp   x29, x30,  [sp,  #48]      // save fp and lr
   ...
2: ldp   x29, x30,  [sp,  #48]      // load fp and lr
2: ret                              // return (lr)

3: SSTP  x29, x30,  [sp,  #48]  // security save fp and lr
   ...
4: SLDP  x29, x30,  [sp,  #48]  // security load fp and lr
5: ret                          // return (lr)
            ROP 代码示意

1: ldr     x1,  [sp,  #30]     // load address
2: br/blr  x1                  // jump to target

3: SLDR  x1,  [sp,  #30]     // use SLDR inst instead
4: br/blr  x1                // jump to target
            ROP 变体代码
```

图 9　ROP 及变体代码示意图

6.2　DOP

攻击者通过修改控制转移数据能够改变程序的执行流，造成权限提升和信息泄漏．有许多的安全防御机制保证控制流完整性（ CFI ）[21,30,36]．然而一直被忽视的非控制流数据在内存漏洞方面也有卓越的表现．在文献[39]中提出的 DOP 攻击（ Data-Oriented Programming)，能够利用非控制流数据(非用于控制流转移的内存变量)，不用事前知道内存地址而可以轻松地绕过 ASLR（ Address Space Layout Randomization ）防御机制．甚至，攻击者可以利用其在内存中执行任意的计算．

CFI 机制难以抵御这种非控制流数据的攻击．而 PCA[42,43]机制针对返回地址和代码指针的保护．接下来我们分析安全检查机制如何保护数据修改，避免 DOP 攻击．

我们还是沿用章节 3.2 中的例子来说明．下面是使用安全机制生成的代码，编译器打开了安全检查选项．代码如下面图 10 所示．假设攻击者修改了 req 内容，在第 6 行从内存中加载到了 ptr（ 此时 ptr 等于 req 的值）．在第 9 行，也就是执行图 1 中的 C 代码 ptr->type==ADD 或者执行 ptr->type=NONE 时，需要加载 type 的值，本来指令读取 ldr w9, [x8,#12]，但是我们使用安全检查指令 SLDR.D x9, [x8,#16]，此时的 type 还保存着在 3.3 章节中说明的数据扩展的内容．右边的第 9 行代码会做安全检查，如果一旦检查失败，会触发异常，程序终止．这样篡改的数据无法执行．在 12 行加载 ptr->req 指针，第和 13 行重新赋值 ptr，也会做安全的检查． 第 14 行将按照我们之前叙述的安全规则，将数据保存到内存，一旦指针或者数据被篡改，则不安全检查不满足，导致失败．按照这样的规则，在访问变量数据和指针时能够保证上下文完整性．值得说明的是，使用我们的安全机制，很难构造 Code-Reuse 类的攻击．因此不会使用这类安全指令构造数据．如果在安全场景比较苛刻的情况时，可以使用安全性约束更加强的执行上下文一致性约束，能够防御这类 DOP 攻击．

原始代码示意	安全机制保护代码示意
1: foo_req:	foo_req:
2: add x8, sp, #0x10 // x8=ptr	add x8, sp, #0x10 // x8=ptr
3: str x8, [sp, #8] // save ptr	SSTR x8, [sp, #8] // save ptr
4: ...	...
5: 0x400400:	0x400400:
6: ldr x8, [sp, #8] // load ptr	SLDR x8, [sp, #8] // load ptr
7: cbz x8, 400450	cbz x8, 400450
8: ...	... 必须按照扩展规则64位对齐
9: ldr w9, [x8, #12] //ptr->type	SLDR.D x9, [x8, #16] // ptr->type
10: cmp w9, #0x3ff	cmp w9, #0x3ff
11: ...	...
12: ldr x8, [sp, #8] // load ptr	SLDR x8, [sp, #8] // load ptr
13: ldr x8, [x8] // ptr = ptr->req	SLDR x8, [x8] // ptr = ptr->req
14: Str x8, [sp, #8] // save ptr	SSTR x8, [sp, #8] // save ptr
15: b 0x400400	b 0x400400
16: ...	...
17: 0x400450:	0x400450:
18: ldr x30, [sp, #48]	ldr x30, [sp, #48]
19: add sp, sp, #0x40	add sp, sp, #0x40
20: ret	ret

图 10　DOP 保护代码示意图

6.3 GOT

GOT 表是在程序动态链接过程中，保存外部定义的符号的特殊数据结构，在 ELF 的.got 段中. GOT 表保存着全局函数的地址. 通过修改它能够改变函数的执行流. 修改 GOT 表的攻击[46]，采用非暴力破解，可以绕过 ASLR 和 W⊕X 防御机制，执行各种易受感染的栈相关攻击，危害较大. 为了保护 GOT 表免于此类攻击，我们将在程序链接编译的时候，如果需要访问 GOT 表的函数，则使用 SLDR 指令，如下图 11 所示：第 2 行是原来的代码，直接使用 ldr 加载地址. 如果是修改的攻击者的目标地址，则可以正常执行. 但是如何使用 SLDR 指令（第 3 行所示）. 则在读取的时候，会检测安全检查.

```
1:  adrp   x16,  0x414000       // Address of GOT
2:  ldr    x17,  [x16, #80]     // address of printf in GOT
3:  SLDR   x17,  [x16, #80]     // security check it
4:  add    x16,  x16, #0xd0
5:  br     x17
```

图 11 GOT 表函数访问代码示意图

使用我们提出的保护机制时，有三点需要注意：第一、在采用对 GOT 保护时，默认使用的是全局安全策略，如果编译选项指定-sc-policy-context 时，则默认忽略. 第二、在整个程序执行 main 函数之前，在 init 初始化函数中，需要对 GOT 表项进行初始化，流程和初始化全局变量相似. 第三、ARM64 平台在编译时默认启用 PLT（Procedure Linkage Table，延迟绑定技术），在程序需要的时候，在解析 GOT 表项的地址. 为了减少设计的复杂度，降低程序的不可预期的问题，在编译的时候，我们关闭 PLT 选项.

6.4 指针完整性保护

（1）虚函数表

现有的系统关键性软件由于性能的原因大都采用 C++语言编写，而虚函数表（Virtual Function Table，VFT）是实现 C++函数多态特性的关键. 因此很多的虚函数表成为攻击的考虑的目标. 文献[47]分析了常见的感染虚函数表的方式. 在这些常见的攻击中，虚函数表指针（VFPTR）成为常用的攻击手段，它指向程序的虚函数表，而虚函数表中保存着该类的函数指针.

针对虚函数的保护，也提出了较多的应对机制. 接下来我们详细说明我们的解决方案如何保护虚函数劫持攻击的. 如图 12 所示，中间汇编代码第 3 行调用类构造函数，执行初始化任务. 在第 12 行，X1 保存虚函数表的地址 Vptr. 将其写入类的所在内存空间. 然后，第 5 行读取虚函数表，第 7 行调用函数. 编译器编译类时对虚函数表设置为只读的属性，因此在程序执行过程中无法修改. 但是，如果攻击者攻击第 14 行，修改虚函数表指针，指向伪造的函数表，则执行流被攻击者控制. 按照我们的保护机制，函数在第 14 行和第 21 行保存虚函数表指针 Vptr 时,使用 SSTR 指令保存，一旦 X0 指向内存被修改，则用 SLDR 指令加载时安全检查失败，进入异常处理. 这样就保护类的虚函数表. 在编译 C++的虚函数表时，默认强制使用全局约束的策略.

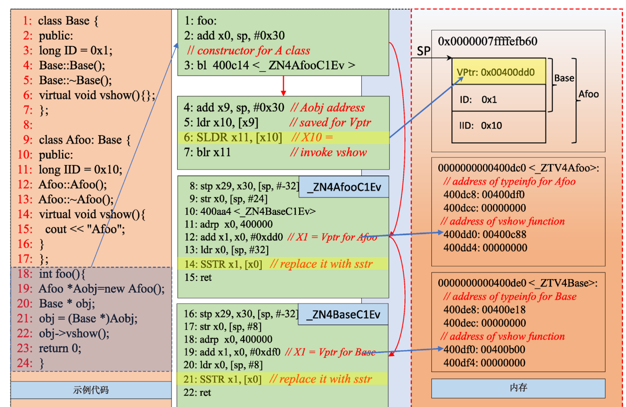

图 12 保护虚函数代码示意图（左边是 C++示例函数，右边是汇编代码）

（2）指针分离

控制流劫持的实质在于攻击者可以利用其他的内存漏洞修改代码指针，而执行流无法检查指针是否被修改，指针是否按照正常的意图执行. 基于这个想法，提出了很多保护机制. 文献[30]提出的代码指针隔离（Code Pointer Separation，CPS）将代码指针保存到一个隔离的安全空间，防止被攻击者篡改. 类似于增加一块影子内存，专门存储指针. 然后编译器自动插入保存和检查代码，验证其完整性. 这样做有两个弊端，第一增加了额外的内存空间. 而且并不能完全保证其安全性，因为编译器插入的代码也要手动写入这个安全区（Safe Regions）. 如果攻击者攻击这块代码，则安全性也难以保证. 第二，这样保存检查带来了 5%的性能损耗[30]. 使用我们的安全机制，在保存指针时使用 SSTR 指令，用 SLDR 指令加载. 则可以判断指针是否被修改过.

6.5 Bufferflow

缓冲区溢出是最常见利用的漏洞，它可以覆写与其申请的空间相邻的变量，典型的应用就是修改

了栈中保存的返回地址和帧地址. 也有很多的文献[1,3,28,29]提出防御溢出的机制. 软件的做法通常是插入标签，再执行的时候再比对，比如 StackCanary 方式. 这种方式防御能力较弱，插入的标签也容易被伪造，因此安全度并不高. 硬件的实现抵御缓冲区溢出的方式如文献[1,3]，针对的攻击向量单一，实现相对复杂. 我们的实现如下所示.

如图 13 中代码所示. 主要分两个步骤，①插入标签，以缓冲区的结束地址为需要加解密的对象. 使用 STTR 将其存入缓冲区尾端. 此步骤一般在 buffer 初始化的时候，由编译器动态的插入. ②检查标签. 使用 SLDR 指令读取结束地址的内容，然后比对两次的地址是否相同. 此步骤，在需要访问 buffer 的内容或者敏感数据时读取比对.

```
insert_canary:
   SSTR  x1,  [x1, #0]      // insert a tag
 ...
check:
   SLDR  x2,  [x1, #0]
   cmp   x1,  x2            // check address tag
   b.ne  exit
continue:
 ...
exit:
   syscall  exit
```

图 13　缓冲区溢出保护示例

6.6　信息泄漏

从我们提出解决方案的初衷来看，我们的目的是为了保存程序的敏感数据. 因此在这方面，我们也可以使用其加密内存秘钥和密码等对象. 例如漏洞 HeartBleed[48]，攻击者利用在 OpenSSL 库中一个读取超过 Buffer 指定长度的数据的漏洞，可以泄漏随网站证书的秘钥. 比如实现 HDFI[21]，将内存中的数据按照字节的粒度做一个标签，然后通过相关指令检查. 我们的实现要简单的多，只需要将其使用 SSTR.D 类指令保存到内存，然后在需要访问的函数中，使用 SLDR.D 指令读取. 攻击者就算获得相应内存的数据，也并不是容易得到其原始的内容. 如果是针对安全性更强的应用场景，建议采用约束更强、更安全的上下文执行策略.

7　评　估

本章节我们主要讨论如何评估我们的解决方案. 我们从下面三个方面来分析：

（1）安全性. 主要通过一些安全测试程序，来评估我们的设计是否能够保证其安全性.

（2）性能评估. 通过 SPEC2006 性能测试集，评估我们的安全机制带来的性能损耗.

（3）易用性. 我们讨论安全机制的适用场景，并且给出编程指导和一些使用建议.

7.1　环境搭建

（1）硬件平台

我们选择 Gem5 模拟器[25,26]作为硬件模拟的平台. 使用 ARM64 架构，CPU 模型选择 O3 模型，具体的配置信息如下面的表 4 中所示.

表 4　Gem5 模拟器配置信息

参数	配置及其说明
架构	ARM64，DerivO3CPU 模型
核心	频率 2GHz，取指宽度 8 指令，取指队列 32 项，发射宽度 8 指令，提交队列 8 指令；64 项 Load/Store 队列，256 个整数物理寄存器；192 项 ROB 项
安全模块	MAC 可选位，加密解密: 4-cycles
L1 指令 Cache	32KB, 4 路相连，1-cycle，Cache line 为 64 字节
L1 数据 Cache	32KB, 8 路相连，1-cycle，Cache line 为 64 字节
L2 共享 Cache	4MB, 16 路相连，8-cycles，Cache line 为 64 字节
DRAM*	4GB，从 L2 读取延迟为 30ns

注：* 主内存选用单通道 DDR3-1600 X64.

（2）软件环境

编译器我们使用 Clang/LLVM-9.0.0，这个版本支持 Linux 内核的编译，利于后期实验. 实验使用和 Glibc 兼容度较高的 musl-libc 库，版本是 1.2.2. C++库使用的是和 Clang 配套的 libcxx、libcxxabi 和 libunwind 库. 为了适配我们的安全机制，我们对 musl-libc 做了部分修改. 包括增加链接时 crtdenctor.o，修改部分启动初始化代码. Linux 内核使用的版本是 4.20.12. 其他部分的修改在文章中涉及到的部分有特殊说明.

7.2　安全评估

本小节主要分析我们提出安全机制的安全性，使用现有的安全测试集测试. 下面我们从下面三方面说明.

（1）RIPE 测试集

为了验证我们设计的有效性，我们使用 RIPE 测试集[49]验证. RIPE 测试集主要是为了测试控制流完整性(CFI)防御机制，它广泛用于安全类测试[21,50]. RIPE 测试集提供了五个测试维度，包括 Buffer 的位置(Location)、溢出的技术(Overflow Technique)、目标代码指针(Target Code Pointer)、攻击代码(Attack Code)和滥用函数(Function Abused)这五大类.

它主要是为 X86 平台而设计,为了适应 ARM64 平台，我们需要做一些移植工作. 由于一些原因，我们无法全部的移植 RIPE 的所有测试集. RIPE 总共覆盖 112 个攻击. 很多的攻击特性并不是符合 ARM 架构. 除去针对 X86 平台，以及在 ARM64 编译时不使用的代码生成规则. 我们总共移植了 52 类攻击. 包括修改了其中的部分代码，如 ROP 攻击的代码片段（Gadgets），帧指针的使用等等. 实验显示能够检测所有的异常攻击. 具体的测试通过情况在附录 1 中所示.

（2）NIST Juliet 测试集

Juliet 1.3 测试集是由 NIST（美国国家标准技术研究院）[51]收集的一系列由 C/C++语言开发的测试集. 最新的版本 1.3 总共包含 118 类 CWEs（Common Weakness Enumeration，通用缺陷枚举）. 为了测试通过情况，我们移植了 CWE121 和 CWE122 两类. 主要是针对基于栈的缓冲区溢出和基于堆的缓冲区溢出. 由于我们目前安全机制的移植和适配只对库函数 memcpy 函数做了修改. 如在 6.5 节讨论的方法，在缓冲区分配时插入标签，在 memcpy 后使用缓冲区变量前检查标签. 因此我们测试和此函数相关的 82 个样例，结果显示全部通过测试.

（3）伪造的测试样例

我们也手动编写了一些测试样例. 比如为了测试 DOP 类似的攻击，我们伪造了类似章节 6.2 中的示例漏洞，检测是否能够抵御. 同样我们为了测试虚函数表指针劫持，GOT 表相关的安全，也编写了类似的攻击代码. 需要说明的是，编写的测试样例并不是仅仅针对我们的安全机制，而是具有一定的普遍性.

7.3 性能评估

为了全方面的评测我们设计的带来的性能损耗，我们主要从以下三个方面评估. 第一，新增的 Sload 和 SStore 指令对流水线的影响. 第二，与其他的安全机制的性能对比. 第三，从宏观的角度分析保护不同的敏感数据对性能产生的影响.

根据前面章节的讨论，在处理全局变量时（GOT 表也可认为由编译器产生的全局变量），为了保证程序的正确执行，在 main 函数执行之前，要初始化这些变量. 由于此部分只执行一次，并不频繁的调用，因此初始化的性能损耗可以忽略不计.

7.3.1 流水线

特权级指令可以单周期完成，因此我们可忽略这类指令. 我们重点分析分 SLoad 类和 SStore 类指令. 如图 7 所示，SLoad 类指令在发射后，进入 LSU 单元，然后执行地址产生，dTLB 转换以及 Cache 的访问. 这是通常 SLoad 类指令所经过的流水线. 但是安全类的 SLoad 指令，需要将 Cache 读出的数据加载到安全模块，执行完之后才能将结果写入寄存器，进入 ROB 队列. 因为这是强的数据依赖操作. 因此，相比于正常的 Load 指令，需要多执行加密的模块的延迟，我们的设计中是 4 个时钟周期.

SStore 指令在发射单元数据准备好时，按照两条流失线执行. 一条按照正常的 Store 类指令执行地址产生，TLB 以及 Cache 的访问；第二条发射单元所需要的源数据准备好直接将其送入安全模块推测执行. 安全模块的输出数据在写会到 ROB，等待指令的提交. 这样可以节省 2 个周期的延迟.

7.3.2 与其他机制性能对比

为了更加清楚地展示与其他安全机制的不同与优势，我们对比了与本文设计方案比较相似的其它四个安全机制. 我们使用 SPEC CPU 2006 整型测试集[52]评估性能. 编译器使用 Clang-9.0.0，采用-O0，没有优化选项. 总共执行了 9 个测试程序，其他的测试程序由于在其文献没有记录，因此我们去掉冗余的. 其他安全方案的相关说明如下所示：

• AOS：边界检查的安全机制[24].
• CPI：基于软件方式的代码指针完整保护[30].
• CCFI：使用 x86 加密指令来保证完整性[38].
• PA：ARM64 的指针验证[42,53].

图 14 展示了不同解决方案执行 SPEC CPU2006 整形测试集的规范化时间. AOS 大约有 8.2%的性能损耗，虽然使用更加精简的指令数目，但是由于每次保护完整性都需要 ARM64 PA 操作以及较为复杂的元数据索引，而且还要处理相同 PAC 值的特殊情况.

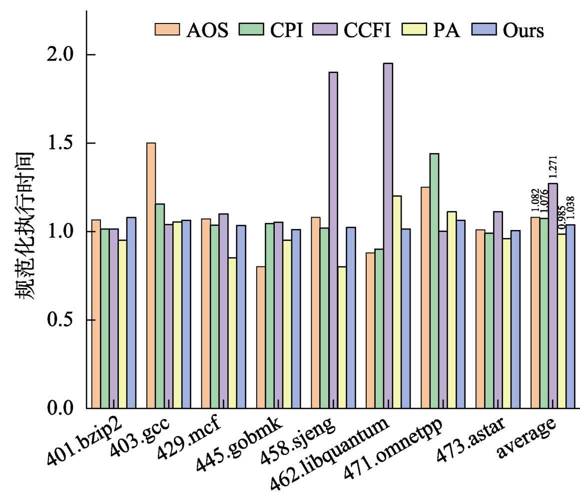

图 14 不同机制的执行时间

软件的 CPI 方式，监管程序内所有的指针，阻止控制流劫持类攻击. 每个指针都有相应的元数据属性操作，每次操作指针都有首先访问元数据. 因此平均损耗大约为 7.6%. 471.omnetpp 测试程序由 C++编写，由大量的指针类数据，频繁地调用，因此几个安全机制的损耗都相对较高.

CCFI 使用 x86_64 的加密指令，使用 AES 算法，加密的宽度达到了 128 位. 由于 Intel 加密指令执行拍数较长，因而在较多的测试集中，执行时间明显高于基准时间. PA 我们使用 parts-llvm[52]编译，在只保护返回地址的情况下，性能损耗最低，效果也最为明显.

7.3.3 性能分析

（1）不同保护对象下的执行时间

所有的测试都使用-O0 选项. 测试程序 输入使用 ref 数据集. SPEC2006 整型测试集共有 13 个测试样例，其中 456.hmmer 和 464.h264ref 程序崩溃，我们共测试了剩余的 11 个测试样例. 我们首先执行 2 百万条指令热启动，然后统计执行 2 亿条指令的运行情况.

图 15 展示了程序执行的时间情况. 我们以不使用保护作为基准，然后分别统计了使用两种不同保护机制的执行情况. 分别是全保护（使能所有的保护选项），指针保护（指针和返回地址）方案. 情况如图 15 所示：使用全保护方案时，性能的损耗为 4.53%，而仅保护指针时，性能损耗只有 2.03%. 由我们在前面章节的分析，影响性能的主要是两方面：1）流水线安全类指令的流水线延迟. 2）访存指令的数量. 由前面的章节的分析可得，虽然我们的安全模块的执行为 4 个时钟周期，但是安全 SStore 类的指令仅有 2 个周期，而 SLoad 指令由于数据依赖必须多额外 4 个周期完成.

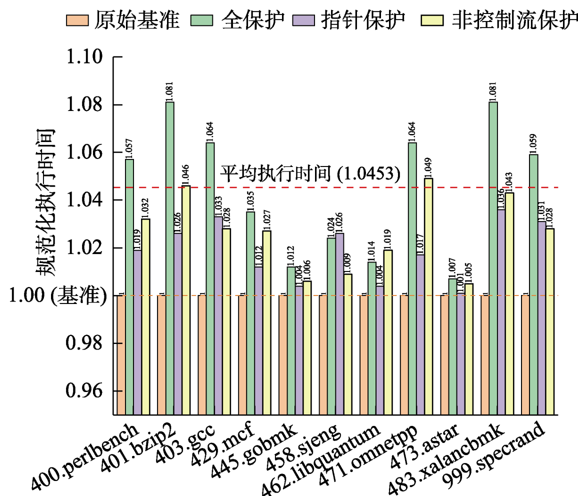

图 15 SPEC2006 性能测试结果

测试样例 445. gobmk, 462.libquantum, 473.astar 等性能损耗相对较低，在平均执行时间之下. 而 401.bzip2, 403.gcc, 483.xalancbmk 等执行时间高于平均值. 性能损耗最高达到 8.1%. 主要原因是此类应用主要是访存密集型程序，频繁的读取内存，同时也有较多的安全类指令. 如图 16 所显示各个程序的访存指令与安全类 SLoad/SStore 指令的占比情况. 比如 401.bzip2 程序，总的访存类指令占将近 70% 左右，而安全类指令占据将近 20%. 在这些安全类指令中相对耗时的 SLoad 类指令几乎占据了整个的安全访存类指令，因此相比于基准，性能损耗最大. 而 458 程序，虽然访存指令占比较高，但是安全类指令的占比较低仅有 5%，因此性能损耗也较低只有 2.6%.

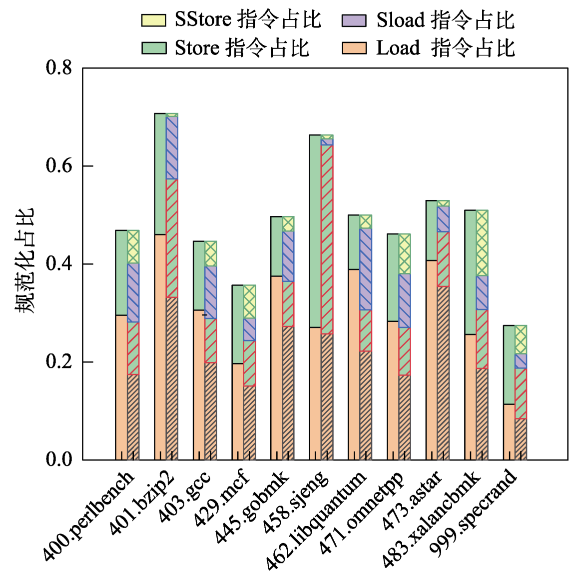

图 16 SPEC2006 安全指令和访存指令与总指令数的占比情况

（2）SLoad 和 SStor 的对比

图 17 展示了安全指令与访存指令的占比情况. 483.xalancbmk 程序的 Sload 和 SStore 指令占据基准访存达到 80%，因此在性能测试中图 7 中，执行时间增加了 8.1%，程序 401.bzip2 和 403.gcc 也是同样的情况. 458.sjeng 的占比最低，因此相对执行时间增加的也比较少 2.6%. 与我们前面的性能评估时得出的结论一致.

表 5 展示的是在执行过程中正常的访存指令与安全指令 SLoad 和 SStore 的访存带宽数据情况. 此表也展示了各个程序的访存密集型特性. 其中 Load 统计数据包括 SLoad 类指令，Store 包括安全执行指令 SStore. 而 SLoad 和 SStore 表明在程序执行中解密

和加密的数据大小. 程序 400.perlbench, 401.bzip2 和 403.gcc 等相对较其他应用带宽更高, 它们的执行时间也最高, 超过了平均执行时间.

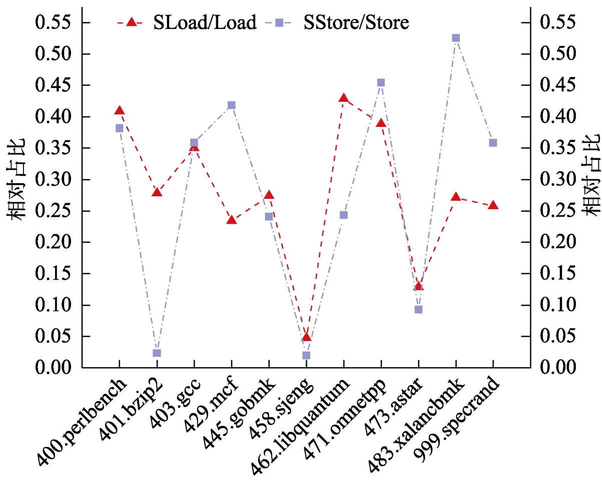

图 17 SPEC2006 中安全指令 Sload、SStore 在 Load、Store 指令中的占比情况

表 5 安全检查时内存带宽

测试集	Load(MB)	Store(MB)	Sload(MB)	SStore(MB)
400.perlbench	494.96	290.74	202.22	111.01
401.bzip2	771.43	415.83	214.58	9.80
403.gcc	513.28	236.08	179.79	84.75
429.mcf	330.43	267.92	77.47	112.13
445.gobmk	629.80	203.96	172.61	49.11
458.sjeng	453.66	659.67	21.48	12.80
462.libquantum	652.73	186.50	279.74	45.33
471.omnetpp	474.61	299.68	184.36	136.09
473.astar	682.72	205.61	87.75	19.08
483.xalancbmk	123.10	121.91	33.36	64.08
999.specrand	87.49	123.86	22.55	44.39

（3）Cache 缺失率

使用安全策略保护非控制流数据时, 我们使用特定的扩展规则来保存数据的 MAC 信息. 因此对 Cache 产生一定的影响. 图 18 展示了程序中的数据增加占比以及对 L1、L2 Cache 的影响. 我们增加了相关统计指令（对程序产生的影响很小, 可忽略不计）, 在程序运行过程中, 动态的收集数据增加的情况. 相比于静态的算法, 动态更加能够体现数据增多对程序的影响. 使用此机制后, 程序平均的数据增加在 5.3%. 程序 483.xalancbmk 异常, 未计算在内. 而 Cache Miss 都几乎保持在 1%~2.5%之间. 可见数据增加对程序的影响相对较小. 因为数据的扩充都和数据相邻, 因此在访问数据和 MAC 元数据时, 都在同一个 Cache Line 内. 而且这些非控制流数据比较分散, 并不是集中出现. 只是相比未增加 MAC 时, 同一个 Cache 行容纳的数据变少. 这体现在有 2%左右的 Cache Miss 增加.

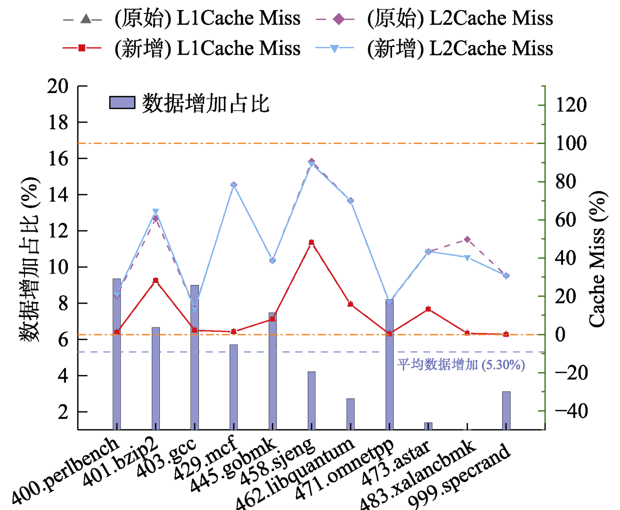

图 18 数据增加及 Cache Miss 情况

7.4 实践指导

我们从下面三个方面来说明安全机制在实践中使用的编程和移植建议.

（1）编译选项

根据我们第三章设计部分的讨论, 我们对不同的敏感内存对象采取安全保护的机制. 不同的场景可以选择不同的保护对象. 针对控制流相关的保护返回地址等; 针对 DOP 相关的攻击我们可以选择使用对非控制流数据加密. 以及针对指针完整性, 采用对指针的保护, 如 GOT, 虚函数劫持等等. 我们提供编译选项来支持上述不同的应用对象加密. 但是在默认的情况（使用-aarch64-security-check）, 我们仅仅打开对返回地址, 帧指针进行保护.

其次, 我们也提供了针对非控制流保护的更加灵活的选择方式. 对需要保护的数据 Clang 提供了 __attribute__((security)) 的属性选项. 需要说明的是, 如果变量按照上述申明, 则在生产具体的底层代码时, 强制使用相关的指令. 而不需要手动的指定针对非控制流-sc-*的相关选项. 在使用时需要特殊的注意.

（2）策略选择

我们提供了全局约束和执行上下文约束, 这两种约束中, 后者需要满足的约束条件更加的苛刻, 因此保护的安全程度也越高. 在某些特殊场合下, 可以对这些关键代码进行执行上下文策略保护. 默

认使用的是全局约束. 其次在我们有些特殊的保护，比如 GOT，虚函数指针等保护，即使指定执行上下文策略时，为了保证程序的正确执行，也会强制使用全局策略.

（3）移植指导

对于一些库函数，使用这类安全指令时需要特殊的注意，比如 memcpy，memmove 等，不建议使用安全机制，因为这些函数经常被其他的程序使用，会导致程序执行错误. 建议针对不同的项目使用对这类函数的包装函数. 很多在 C/C++实现的项目中，可能使用此类语言提供的小技巧，比如使用内嵌汇编修改控制流等方式. 建议不要使用，因为可能会产生意想不到的错误. 原则上不需要修改源代码，只需要重新编译即可，但是如果项目中有上述的特殊注意，还需要手动的修改部分源码，更好的适配安全检查机制.

8 结 论

本文中我们提出了一种软硬件协同保护程序运行时的安全解决方案. 可以应对控制流劫持攻击，如 ROP，CFI 类攻击等. 通过对非控制流数据的保护，可以抵御类似 DOP 的攻击. 我们提出的解决方案同时也可以保护运行时 ELF 文件中的 GOT 表和虚函数表指针. 相比于其他的安全防御机制，我们的解决方案在支持抵御多种公积向量的同时，也具有较小的性能损耗. SPEC2006 的结果显示，仅有 4.5%的性能损耗.

参 考 文 献

[1] Cowan C, Pu C, Maier D, et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks//USENIX security symposium. 1998, 98: 63-78

[2] Nagarakatte S, Zhao J, Martin M, et al. Spatial memory safety for C//Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009

[3] Hasabnis N, Misra A, Sekar R. Light-weight bounds checking //Proceedings of the 10th International Symposium on Code Generation and Optimization. San Jose, USA, 2012: 135-144

[4] Serebryany K, Bruening D, Potapenko A, et al. Addresssanitizer: A fast address sanity checker//2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12). San Jose, USA, 2012: 309-318

[5] Chen S, Kozuch M, Strigkos T, et al. Flexible hardware acceleration for instruction-grain program monitoring. ACM SIGARCH Computer Architecture News, 2008, 36(3): 377-388

[6] Deng D Y, Suh G E. High-performance parallel accelerator for flexible and efficient run-time monitoring//IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). Boston, USA, 2012: 1-12

[7] Fytraki S, Vlachos E, Kocberber O, et al. FADE: A programmable filtering accelerator for instruction-grain monitoring//2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). Orlando, USA, 2014: 108-119

[8] Wahab M A, Cotret P, Allah M N, et al. ARMHEx: A hardware extension for DIFT on ARM-based SoCs//2017 27th International Conference on Field Programmable Logic and Applications (FPL). 2017: 1-7

[9] Roessler N, DeHon A. Protecting the stack with metadata policies and tagged hardware//2018 IEEE Symposium on Security and Privacy (SP). San Francisco, USA, 2018: 478-495

[10] Azevedo de Amorim A, Collins N, DeHon A, et al. A verified information-flow architecture//Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego, USA, 2014: 165-178

[11] Dalton M, Kannan H, Kozyrakis C. Raksha: a flexible information flow architecture for software security. ACM SIGARCH Computer Architecture News, 2007, 35(2): 482-493

[12] Liu Z, Criswell J. Flexible and efficient memory object metadata. ACM Sigplan Notices, 2017, 52(9): 36-46

[13] Shroff P, Smith S, Thober M. Dynamic dependency monitoring to secure information flow//20th IEEE Computer Security Foundations Symposium (CSF'07). Venice, USA, 2007: 203-217

[14] Vahldiek-Oberwagner A, Elnikety E, Duarte N O, et al. ERIM: Secure, efficient in-process isolation with protection keys (MPK)//28th USENIX Security Symposium (USENIX Security 19). Santa Clara, USA, 2019: 1221-1238

[15] Hedayati M, Gravani S, Johnson E, et al. Hodor: Intra-process isolation for high-throughput data plane libraries//2019 USENIX Annual Technical Conference ({USENIX}{ATC} 19). Renton, USA, 2019: 489-504

[16] Wang Z, Wu C, Xie M, et al. Seimi: Efficient and secure smap-enabled intra-process memory isolation//2020 IEEE Symposium on Security and Privacy (SP). 2020: 592-607

[17] Proskurin S, Momeu M, Ghavamnia S, et al. xMP: selective memory protection for kernel and user space//2020 IEEE Symposium on Security and Privacy (SP). 2020: 563-577

[18] Nagarakatte S, Martin M M K, Zdancewic S. Watchdog: Hardware for safe and secure manual memory management and full memory safety//2012 39th Annual International Symposium on Computer Architecture (ISCA). 2012: 189-200

[19] Nagarakatte S, Martin M M K, Zdancewic S. Watchdoglite: Hardware-accelerated compiler-based pointer checking// Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. Orlando, USA, 2014: 175-184

[20] Aingaran K, Jairath S, Konstadinidis G, et al. M7: Oracle's next-generation sparc processor. IEEE Micro, 2015, 35(2): 36-45

[21] Song C, Moon H, Alam M, et al. HDFI: Hardware-assisted data-flow isolation//2016 IEEE Symposium on Security and Privacy (SP). San Jose, USA, 2016: 1-17

[22] Oleksenko O, Kuvaiskii D, Bhatotia P, et al. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. Proceedings

of the ACM on Measurement and Analysis of Computing Systems, 2018, 2(2): 1-30

[23] Criswell J, Dautenhahn N, Adve V. KCoFI: Complete control-flow integrity for commodity operating system kernels//2014 IEEE Symposium on Security and Privacy. Berkeley, USA, 2014: 292-307

[24] Kim Y, Lee J, Kim H. Hardware-based always-on heap memory safety//2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 2020: 1153-1166

[25] Binkert N, Beckmann B, Black G, et al. The gem5 simulator[J]. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1-7

[26] Lowe-Power J, Ahmad A M, Akram A, et al. The gem5 simulator: Version 20.0+. arXiv preprint arXiv:2007.03152, 2020

[27] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Notices, 2007, 42(6): 89-100

[28] Castro M, Costa M, Martin J P, et al. Fast byte-granularity software fault isolation//Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. Koblenz, Germany, 2009: 45-58

[29] Erlingsson U, Abadi M, Vrable M, et al. XFI: Software guards for system address spaces//Proceedings of the 7th Symposium On Operating Systems Design and Implementation. Berkeley, USA, 2006: 75-88

[30] Kuznetzov V, Szekeres L, Payer M, et al. Code-pointer integrity//The Continuing Arms Race: Code-Reuse Attacks and Defenses. 2018: 81-116

[31] H Koo, Y Chen, L Lu, V Kemerlis, et al. Polychronakis, compiler-assisted code randomization//2018 IEEE Symposium on Security and Privacy (SP), San Francisco, USA, 2018. 461-477

[32] Dang T H Y, 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. New York, USA, 2015: 555-566

[33] Yu J, Yan M, Khyzha A, et al. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data//Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. New York, USA, 2019: 954-968

[34] De Clercq R, Götzfried J, Übler D, et al. SOFIA: software and control flow integrity architecture. Computers & Security, 2017, 68: 16-35

[35] Davi L, Hanreich M, Paul D, et al. HAFIX: Hardware-assisted flow integrity extension//2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). San Francisco, USA, 2015: 1-6

[36] Zeldovich N, Kannan H, Dalton M, et al. Hardware enforcement of application security policies using tagged memory//Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. San Diego, USA,. 2008

[37] Devietti J, Blundell C, Martin M M K, et al. Hardbound: Architectural support for spatial safety of the C programming language. ACM SIGOPS Operating Systems Review, 2008, 42(2): 103-114

[38] Mashtizadeh A J, Bittau A, Boneh D, et al. CCFI: Cryptographically enforced control flow integrity//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, USA, 2015: 941-951

[39] 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 (SP). San Jose, USA, 2016: 969-986

[40] Chen S, Xu J, Sezer E C, et al. Non-Control-Data Attacks Are Realistic Threats//USENIX Security Symposium. 2005, 5

[41] Avanzi R. The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. IACR Transactions on Symmetric Cryptology, 2017: 4-44

[42] ARM Connected blog. ARMv8-A architecture – 2016 additions. https://www.community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions, 2016,10

[43] Qualcomm Product Security. Pointer Authentication on ARMv8.3–Design and Analysis of the New Software Security Instructions. https://www.qualcomm.com/documents/whitepaper-pointer-authentication-armv83, 2017,1

[44] Buchanan E, Roemer R, Shacham H, et al. When good instructions go bad: Generalizing return-oriented programming to RISC//Proceedings of the 15th ACM conference on Computer and communications security. Denver, USA, 2008: 27-38

[45] Checkoway S, Davi L, Dmitrienko A, et al. Return-oriented programming without returns//Proceedings of the 17th ACM Conference on Computer and Communications Security. Chicago, USA, 2010: 559-572

[46] Roglia G F, Martignoni L, Paleari R, et al. Surgically returning to randomized lib//2009 Annual Computer Security Applications Conference. 2009: 60-69

[47] Zhang C, Song C, Chen K Z, et al. VTint: Protecting Virtual Function Tables' Integrity//Network & Distributed System Security Symposium. San Diego, USA, 2015

[48] Codenomicon and N Mehta, The Heartbleed Bug. http://heartbleed.com/, 2014

[49] Wilander J, Nikiforakis N, Younan Y, et al. RIPE: Runtime intrusion prevention evaluator//Proceedings of the 27th Annual Computer Security Applications Conference. New Orleans, USA, 2011: 41-50

[50] Ding R, Qian C, Song C, et al. Efficient protection of path-sensitive control security//26th {USENIX} Security Symposium ({USENIX} Security 17). Berkeley, USA, 2017: 131-148

[51] Kass M J. NIST Software Assurance Metrics and Tool Evaluation (SAMATE). 2005

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

[53] https://github.com/pointer-authentication/parts-llvm

LI Ya-Wei, Ph.D. candidate. His main research interests include operation system, hardware security, processor architecture and compiler optimization.

ZHANG Long-Bing, Ph.D., professor. His research interests include computer architecture and microprocessor design.

ZHNAG Fu-Xin, Ph.D., professor. His research interests include processor architecture, performance analysis and operation system.

WANG Jian, Ph.D., professor. His research interests include parallel compilation, embedded operation system and processor architecture.

Background

This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDC05020100. The main purpose of this project is to improve the key issues of the processor, including high CPU frequency, design virtualization expansion, security enhancement, high-performance CPU design method, etc.

Memory corruption is currently the root security vulnerability. It occupies more than 70% of all computer security issues. Many mechanisms have been proposed to increase security. It includes two aspects. One method is based on software. During the code produced by the compiler, the verification code is dynamically inserted on program, such as AddressSanitizer and StackGuard mechanisms. Although the software implementation mechanism is flexible, the dynamic execution of the inserted code increases the overall performance overheads. On the other hand, hardware-based mechanism also gives a promising direction, such as the current commercial enhancement technologies Intel's MPX and MPK. Although the hardware implementation can solve the problem of performance loss, it can defend against limited attacks, high design complexity, and so on.

In this paper, we propose a security check mechanism based on software and hardware collaboration. It can deal with multiple attack vectors while maintaining the high performance of the hardware design. It also can provide more fine-grained security protection (Machine Word). At the same time, two security strategies for different scenarios are proposed, global constraints and execution context constraints. Different strategies can be used according to different security requirements. We implemented the prototype on the hardware simulator Gem5. Through the SPEC2006 experiment, the security mechanism we proposed has only a 4.5% performance overhead. We also analyzed 6 series of safety application examples in detail. Meanwhile, we give a set of solutions including the compiler, runtime and libc library. To accurately analyze the performance loss of the security module, we microscopically elaborate the impact of the hardware structure. In addition, it is also compared with other types of security mechanisms. The results show that our designed scheme can provide a secure defense mechanism with low loss.

附录 1：

RIPE 测试程序情况：

表 6 RIPE 测试集通过情况

序号	Overflow Technique	Target Code Pointer/(Location)	Function Abused	Attack Code	通过	序号	Overflow Technique	Target Code Pointer/(Location)	Function Abused	Attack Code	通过
1	Indirect	ret	memcpy	createfile	√	27	Indirect	ret	memcpy	returnintolibc	√
2	Indirect	funcptrstackvar	memcpy	createfile	√	28	Indirect	funcptrstackvar	memcpy	returnintolibc	√
3	Indirect	funcptrheap	memcpy	createfile	√	29	Indirect	funcptrheap	memcpy	returnintolibc	√
4	Indirect	funcptrbss	memcpy	createfile	√	30	Indirect	funcptrbss	memcpy	returnintolibc	√
5	Indirect	funcptrdata	memcpy	createfile	√	31	Indirect	funcptrdata	memcpy	returnintolibc	√
6	Indirect	structfuncptrstack	memcpy	createfile	√	32	Indirect	structfuncptrstack	memcpy	returnintolibc	√
7	Indirect	structfuncptrheap	memcpy	createfile	√	33	Indirect	structfuncptrheap	memcpy	returnintolibc	√
8	Indirect	structfuncptrdata	memcpy	createfile	√	34	Indirect	structfuncptrdata	memcpy	returnintolibc	√
9	Indirect	structfuncptrbss	memcpy	createfile	√	35	Indirect	structfuncptrbss	memcpy	returnintolibc	√
10	Indirect	ret	homebrew	createfile	√	36	Indirect	ret	homebrew	returnintolibc	√
11	Indirect	funcptrstackvar	homebrew	createfile	√	37	Indirect	funcptrstackvar	homebrew	returnintolibc	√
12	Indirect	funcptrheap	homebrew	createfile	√	38	Indirect	funcptrheap	homebrew	returnintolibc	√
13	Indirect	funcptrbss	homebrew	createfile	√	39	Indirect	funcptrbss	homebrew	returnintolibc	√
14	Indirect	funcptrdata	homebrew	createfile	√	40	Indirect	funcptrdata	homebrew	returnintolibc	√
15	Indirect	structfuncptrstack	homebrew	createfile	√	41	Indirect	structfuncptrstack	homebrew	returnintolibc	√
16	Indirect	structfuncptrheap	homebrew	createfile	√	42	Indirect	structfuncptrheap	homebrew	returnintolibc	√
17	Indirect	structfuncptrdata	homebrew	createfile	√	43	Indirect	structfuncptrdata	homebrew	returnintolibc	√
18	Indirect	structfuncptrbss	homebrew	createfile	√	44	Indirect	structfuncptrbss	homebrew	returnintolibc	√
19	Direct	stack	memcpy	createfile	√	45	Direct	stack	memcpy	returnintolibc	√
20	Direct	heap	memcpy	createfile	√	46	Direct	heap	memcpy	returnintolibc	√
21	Direct	bss	memcpy	createfile	√	47	Direct	bss	memcpy	returnintolibc	√
22	Direct	data	memcpy	createfile	√	48	Direct	data	memcpy	returnintolibc	√
23	Direct	stack	homebrew	createfile	√	49	Direct	stack	homebrew	returnintolibc	√
24	Direct	heap	homebrew	createfile	√	50	Direct	heap	homebrew	returnintolibc	√
25	Direct	bss	homebrew	createfile	√	51	Direct	bss	homebrew	returnintolibc	√
26	Direct	data	homebrew	createfile	√	52	Direct	data	homebrew	returnintolibc	√

注：我们移植了能在 Arm64 平台正常运行的 52 个样例．使用本文设计的安全机制，都能到捕获到异常．

附录 2：

新增特权级指令、用户级指令以及用于统计信息的辅助指令如下表：

表 7　新增指令列表及说明

指令	指令类型	指令描述
特权级指令		
SECE（SEcurity Check Enable）	使能特权类	打开安全检查机制
SECD（SEcurity Check Disable）	使能特权类	关闭安全检查机制
GKCRS（GKCR Set）	设置寄存器类	设置全局策略 GKCR 寄存器
GKCRR（GKCR Read）	设置寄存器类	读 GKCR 寄存器
GKCRC（GKCR Clear）	设置寄存器类	清空 GKCR 寄存器
CFIDS （CFID Set）	设置寄存器类	设置上下文 CFID 寄存器的值
CFIDR （CFID Read）	设置寄存器类	读 CFID 寄存器的值
CFIDC （CFID Clear）	设置寄存器类	清空 CFID 寄存器
用户级指令		
SLDP	Load Pair 类	从内存加载数据（解密）到寄存器对
SSTP	Store Pair 类	存储寄存器对（加密）到内存中
SLDPpre	Load Pair 类	加载数据（解密）到寄存器对,并切更新地址
SSTPpre	Store Pair 类	存储寄存器（加密）对到内存中,更新地址
SLDPpost	Load Pair 类	更新地址，加载数据（解密）到寄存器对
SSTPpost	Store Pair 类	更新地址，存储寄存器（解密）对到内存
SLDR	Load 类（指针）	从内存中加载**指针**到寄存器中
SSTR	Store 类（指针）	存储**指针（加密）**到内存中
SLDx*.D	Load 类（数据）	从内存中加载**不同的类型**数据到寄存器中
SSTx*.D	Store 类（数据）	存储**不同的类型**数据到内存中
SCKE	ALU 类	将源寄存器的值加密后，保存到目的寄存器
SCKD	ALU 类	将源寄存器的值解密后，保存到目的寄存器
STAT_pt	统计信息辅助类	统计被加密的指针
STAT_all	统计信息辅助类	统计内存分配情况（堆、栈）
STAT_obj	统计信息辅助类	统计非控制流对象

注：x*表示不同的类型，字节 Byte，8 位；半字 Half-word，16 位；字 Word，32 位；双字 Double，64 位.