计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202440480
61(10): 2482−2500, 2024

基于动态完整性度量的机密计算运行时监控方案

李 为[1,3] 冯 伟[3] 秦 宇[3] 冯登国[2,3]
[1](中国科学院大学 北京 100049)
[2](计算机科学国家重点实验室(中国科学院软件研究所) 北京 100190)
[3](中国科学院软件研究所 北京 100190)
(liwei2018@iscas.ac.cn)

Scheme of Runtime Monitoring in Confidential Computing with Dynamic Integrity Measurement

Li Wei[1,3], Feng Wei[3], Qin Yu[3], and Feng Dengguo[2,3]
[1] (*University of Chinese Academy of Sciences*, *Beijing* 100049)
[2] (*State Key Laboratory of Computer Science* (*Institute of Software*, *Chinese Academy of Sciences*), *Beijing* 100190)
[3] (*Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)

Abstract Confidential computing (CC) is based on the hardware TEE, which protects the confidentiality and integrity of data in use through isolation, integrity measurement, and remote attestation, and is protected from attacks by privileged adversaries. However, the measurement and attestation mechanisms of existing CC platforms focus on the launch time integrity instead of the runtime. When potential memory vulnerabilities in user workloads are exploited by adversaries, it can lead to attacks such as control flow hijacking. The existing CC platforms that only protect integrity at startup cannot effectively prevent or detect such runtime attacks. To address that, we propose a scheme of runtime monitoring in CC based on dynamic integrity measurement. By introducing control and data flow measurements into the TEE and verifying the measurements by a trusted verifier, the remote attestation and runtime integrity protection are achieved on the CC platform. The implementation of our prototype on CSV/SEV servers with confidential VM/container architecture shows that the scheme achieves higher runtime security with about 16% performance overhead.

Key words confidential computing; TEE; runtime attestation; control flow integrity; data flow integrity

摘 要 机密计算(confidential computing, CC)技术基于硬件可信执行环境(TEE),通过隔离、完整性度量和远程证明等技术保护使用中数据的机密性和完整性,并免受特权敌手的攻击.然而,现有机密计算平台的完整性度量和远程证明机制主要针对启动时,而缺少运行时完整性保护,例如当用户工作负载潜在的内存漏洞被敌手利用时容易遭受控制流劫持等攻击,目前仅包含启动时完整性保护的机密计算平台无法有效防止或者检测此类运行时攻击.针对该问题,提出了一种基于动态完整性度量的机密计算运行时监控方案,通过向TEE中引入控制流和数据流度量,并扩展可信验证方功能以进行运行时远程证明,实现了机密计算平台内用户工作负载的运行时完整性保护.使用CSV/SEV服务器作为机密计算平台,在其机密虚拟机/容器基础上进行了系统原型实现与实验评估,评估结果表明,方案在增强运行时安全性的同时,引入了约16%的性能损耗.

收稿日期: 2024-06-05; **修回日期**: 2024-07-19
基金项目: 国家重点研发计划项目(2022YFB4501500, 2022YFB4501501); 小米揭榜挂帅青年学者科研专项(2024)
This work was supported by the National Key Research and development Program of China (2022YFB4501500, 2022YFB4501501) and Xiaomi Young Scholars' Key Research Program via Open Competition (2024).
通信作者: 秦宇(qinyu@iscas.ac.cn)

关键词 机密计算；可信执行环境；运行时证明；控制流完整性；数据流完整性

中图法分类号 TP309

机密计算(confidential computing, CC)平台基于可信执行环境(trusted execution environment, TEE)为用户的可信工作负载提供对使用中的数据的安全性保证. 然而, TEE 的度量值在其初始化后被确定, 并用于之后任意时刻的远程证明, 现有主流的机密计算平台 TEE 架构(CSV, SEV, TDX, CCA 等)仅提供 TEE 启动时度量和远程证明方案, 缺乏运行时证明能力, 无法将用户信任链扩展到机密虚拟机/容器中, 仅能依靠受信任的硬件所提供的隔离能力和严格规范的外部接口保证运行时安全性, 对控制流劫持等运行时攻击的防护能力不足. 此外, 现有硬件 TEE 的安全机制均依赖于硬件厂商在 CPU 中的信任根, 也容易导致用户对单一角色信赖过高和单点失效等问题.

本文基于目前机密计算平台中应用最为广泛的机密虚拟机/容器架构, 提出了一种基于动态完整性度量的机密计算运行时监控方案, 将完整性度量与远程证明能力扩展至 TEE 的全生命周期, 加强用户工作负载数据和程序的运行时安全性. 本文的贡献有 3 点:

1) 提出了一种针对机密虚拟机/容器中工作负载的有效动态完整性度量方案, 当用户工作负载在机密虚拟机/容器内运行时, 该方案可以有效捕获工作负载运行时控制流和数据流的动态完整性度量值, 弥补了启动时静态完整性度量的不足;

2) 设计了对机密计算平台机密虚拟机/容器工作负载动态完整性度量结果的运行时远程证明方案, 通过对比受保护程序的合法基线模型, 可以对度量结果进行验证, 并基于可信模块 TCM/TPM2, 将运行时证明的信任模型与原机密计算平台 TEE 信任模型解耦;

3) 在国产海光 CSV 等机密计算服务器平台上实现了原型系统, 并进行了安全分析和性能测试, 结果表明本文方案具有良好的运行时安全性, 且平均仅带来 16% 左右的额外性能开销, 是一种实用的 CC 运行时保护方案.

1 背　　景

1.1 机密计算和 TEE

随着云计算、大数据等新型应用场景的发展, 以及移动、物联网设备数量的不断扩大, 对使用中的数据进行保护已经成为一种趋势. 通常, 数据保护涉及数据的 3 种不同形态: 存储时、传输时以及使用中的数据. 传统的安全模型主要涉及到数据的前 2 种静止时的形态, 如在数据库中、在有线或无线网络传输时都会对数据进行加密, 大部分计算设备都会内置数据加解密功能, 并通过硬件芯片中的专用引擎来增强密码运算性能. 而对于“使用中”的数据, 需要对其在内存、处理器之中进行计算时的形态进行机密性和完整性保护, 目前仍然是系统安全前沿的研究方向.

机密计算技术就是对使用中的数据进行安全保护的实用方案之一. 一方面, 机密计算将基于硬件的加密保护扩展到使用中的数据, 通过 TEE 减少云计算平台等场景工作负载所依赖的可信计算基(trusted computing base, TCB), 使得暴露的攻击面最小化, 从而提升使用中数据乃至整个运行时系统的安全性. 另一方面, 机密计算使用基于硬件的安全机制, 将数据、特定功能或整个应用程序与操作系统、虚拟机监视器(virtual machine monitor(VMM), 或称(Hypervisor)以及其他不受信任的特权进程相互隔离, 并在启动时对使用中的数据进行度量证明, 从而在技术上保证了用户数据在计算服务上的隐私性.

早期阶段, TEE 技术主要聚焦于 SoC 总线级隔离(以 ARM Trustzone①为代表)和专用可信硬件模块(以 TCM[1] /TPM2[2] 为代表). Trustzone 首先提出基于硬件将安全世界与非安全世界隔离这一概念, 为安全敏感的代码提供安全的执行环境. 而 TCM/TPM2 等后续技术方案则主要基于隔离的物理芯片实现, 其核心思想在于将少数重要的安全功能隔离在一个与主机系统物理分离的类协处理器上. Intel 公司于 2015 年在其商用处理器上推出了软件防护扩展(software guard extensions, SGX)[3], 正式为 TEE 实例引入了“飞地(enclave)”这一称呼, 并引发了产业界和学术界的广泛关注. SGX 是用户级的硬件 TEE, 它以线程为粒度为应用程序受信任部分提供隔离环境, 并提供度量和远程证明等安全机制, 使得用户可以在不信任计算平台宿主机操作系统或虚拟机监视器等高特权级软件组件的前提下实现工作负载的安全防护. 然而, 尽管用户级 TEE 能提供更小的 TCB 和更强

① https: //developer. arm. com/documentation/100 690/0201

的安全假设,但由于其严苛的编程模型、受限的系统接口和层出不穷的侧信道安全威胁[4],机密计算平台上的大规模应用更趋向于采用具有更朴素编程模型的系统级 TEE,即机密虚拟机/容器,如已商用的 AMD 安全加密虚拟化(secure encrypted virtualization, SEV)①和海光的中国安全虚拟化(China secure virtualization, CSV)②技术. 此外, Intel 可信域扩展(trust domain extensions, TDX)[5] 和 ARM 机密计算架构(confidential computing architecture, CCA)③也已进入商业化流程;开源架构 RISC-V 亦在起草相关规范④.

然而,机密计算和 TEE 的安全能力也存在一定的局限性. 尽管机密计算考虑防御包括特权敌手在内的攻击者,但仍无法防御由于 TEE 内的程序漏洞如内存损坏(memory corruption)等导致的攻击[6],这是由 TEE 本身的信任模型和安全机制导致的. 完整性度量和远程证明技术在一定程度上可以保证 TCB 的可信,确保其中的程序来自正确的发布者,然而 TEE 内仍存在漏洞利用[7]的可能性. 此外,在实际使用中 TEE 用户和计算服务提供商通常会在 TEE 中部署包含大量代码的支撑库以及业务程序,并与不可信环境间保留复杂的通信接口,这些都显著地扩张了 TCB 的大小,增大了 TEE 的攻击面. 针对这类问题,研究者从不同方面提出了解决方案. 例如向 TEE 中引入地址空间随机化[8]、更多的指针和边界检查[9-10]、控制流完整性(control flow integrity, CFI)和控制流证明(control flow attestation, CFA)[11-12]、内存内省[13]、内部隔离[14-16]、漏洞挖掘[17-18]等. 此外还有侧信道方面的诸多研究[19-25].

另外,机密计算也不是保护数据使用的唯一技术. 其他如同态加密、多方安全计算、差分隐私等,也能起到保护使用中数据的效果,但相较而言机密计算通常具有计算性能和资源利用率上的优势. 这些技术与机密计算之间可以相互补充、协同使用以提供更高的安全性.

1.2 机密虚拟机/容器

机密虚拟机(confidential virtual machine, CVM)是机密计算平台上系统级 TEE 的表现形式,即使用了硬件内存加密和隔离的虚拟机,具有相应的硬件度量和远程证明安全能力,并受到严格的访问控制限制. 机密虚拟机并不信任其宿主机操作系统或虚拟机监视器,仅受其调度和管理生命周期.

以 AMD SEV/海光 CSV 为例,机密虚拟机在启动前会对平台证书链进行验证,以判断是否运行在可信的机密计算平台上,并导出自身的证书用于后续远程证明. 在启动时,虚拟机监视器所加载的固件(OVMF)将被硬件度量并受远程用户验证,而固件基于可信启动的方式加载并度量引导虚拟机镜像. 加载完成后,虚拟机所使用的内存将被硬件加密和完整性保护. 虚拟机所使用的内存也将在启动完成后被硬件度量,并作为后续远程证明过程中验证机密虚拟机身份和安全性的凭证.

由于机密虚拟机的部署较为受限,需要定制固件、引导等以确保可信启动被正确执行,一种更为实用的方式是机密虚拟机/容器方案,即将机密虚拟机与容器相结合. KATA Containers[26] 是其中的代表性方案,它将容器易于部署和集群化调度的特点引入了机密虚拟机,仅需基于特定的 CVM 固件、引导和内核镜像等,实现不同机密计算应用的实际部署. 在此类方案中,可信工作负载作为容器镜像被签名和发布,在需要运行时被调度到机密虚拟机中,并结合远程证明以确保机密虚拟机的安全性. 同时,机密虚拟机内部的容器代理还会对容器签名进行验证,或在远程证明通过后获取加密容器镜像的解密密钥.

1.3 动态完整性监控

动态完整性监控是系统安全领域的长期和重要研究之一,一般会与静态保护方法相结合,扩展其防护能力的生命周期. 其重点在于,对已启动并运行的对象,提取能反映其动态完整性状态的数据,并进行实时或异步监控. 对于动态完整性监控的研究起步较早,常见于 OS 内核安全研究,如 Linux 内核原生的完整性度量架构(integrity measurement architecture, IMA)、Copilot[27]、Vigilare[28] 等基于独立硬件的内核内存度量,基于语义完整性违规的运行时检测方案[29],LKIM[30] 提出的基于动态数据结构的上下文检测方案等. 也有针对完整性状态数据提取的研究,如 SBCFI[31] 和 KCoFI[32] 引入了 CFI,而文献 [33] 则提出了自动推导内核数据结构不变量的方法,KOP[34] 给出了一种提取覆盖率的评估方法,并提出对动态内

① https: //developer. amd. com/sev

② https: //openanolis. cn/sig/Hygon-Arch

③ https: //developer. arm. com/documentation/den0129/latest

④ https: //lists. riscv. org/g/sig-trusted-computing/message/8

核对象的定位方案，文献 [35] 提出了基于图的结构不变量签名方案，KENALI[36] 引入了数据流完整性（data flow integrity, DFI）以缓解针对非控制数据的攻击. 国内也有学者提出过各类动态完整性监控改进方案[37-39].

CFI[40] 与 DFI[41] 相结合是动态完整性监控中常用的防护手段. CFI 确保软件执行必须遵循其控制流图（control flow graph, CFG）中的合法路径，排除任何预期外的指令跳转地址；而 DFI 则确保运行时数据流不会违背其数据流图（data flow graph, DFG），关注于 CFI 无法解决的非控制数据攻击，如不改变控制流，仅通过覆写关键数据绕过访问控制. 通常 CFI 和 DFI 方案的安全模型不考虑受控的特权软件敌手，即仅在本地系统实现防护，信任主机 OS 且将完整性验证及进一步的控制策略交予其来完成. 为此，CFA 方案[42-44] 将远程证明（remote attestation）与 CFI 结合，进一步缩小了 TCB，不再在本地进行 CFI 审查，而是将提取的控制流信息发送至可信第三方进行验证. 验证方（verifier）能够向依赖方（relying party）提供验证的结果，使得受保护的系统具有了向外部证明其安全属性的能力. DIAT[45]，OAT[46]，ARI[47] 等还将 DFI 也引入了这一过程. 然而，引入远程证明也带来了额外的挑战，一是其关键过程需要额外保护，包括控制流的度量和证明时的通信等，二是这些额外的过程也会导致性能降低和资源占用，例如为防篡改需要实现证明报告（quote）的加密、报告会占用额外的存储空间等.

通过度量（measurement）获得的度量结果是唯一标识控制流或数据流信息作为证明报告提供给验证方的关键证据. 例如在机密计算硬件 TEE 中，可信硬件通过计算初始化时工作负载内存的散列值，结合平台身份、版本等信息生成该结果. 度量和证明也是可信计算（trusted computing）中的核心概念. 机密计算与可信计算是一脉相承的，其安全内核都在于信任根和信任链的扩展，强调对 TCB 的保护. 机密计算还基于硬件 TEE 来实现对不受信特权敌手的防护和可信工作负载的灵活部署.

2 研究现状与相关工作

2.1 现有 TEE 存在的不足

现有的 TEE 信任模型并未完全覆盖其生命周期. 一般而言，TEE 架构要求编程者在原有程序模型中建立信任模型，划分出可信部分和不可信部分，可信部分将运行在 TEE 中并受到硬件的隔离和加密保护. 该安全假设是非常严格的，甚至不信任外部操作系统，即能够防御来自特权软件敌手的攻击. 然而对于保证程序运行时安全性的目标仍有差距. 一方面，对程序作出分割并进行严格的边界检查，虽然确实能够控制外部程序对飞地的访问，缓解对可信工作负载的恶意利用，但仅仅对信任模型作出划分并不会减少程序潜在的缺陷，程序中固有的漏洞仍可能在 TEE 中受到利用，典型的如内存损坏导致的控制流劫持. 盲目地信任划分后的程序并将其放入飞地中甚至可能导致更低的实际安全性[6].

另一方面，在规模化部署下，应用总是倾向于将更多的程序模块置于 TEE 中以获得更高的易用性，例如通过整合运行时库、使用库操作系统（libOS）模拟部分系统调用，让用户级 TEE 能够兼容通用二进制程序. 但这一做法显著地扩大了 TCB，对保证程序的安全性不利. 系统级 TEE 由于其架构设计，该缺陷普遍更为明显.

最后，一旦控制流劫持完成，TEE 将不再保留任何防御能力，因为其对可信工作负载的所有安全检查都在启动时完成，而运行时安全性仅有硬件本身提供的隔离保证. 其所加载的受保护内存页由硬件度量所形成的度量值在启动后不再变化，后续的远程证明报告中将包含固定的这一结果. 当控制流被劫持后，若攻击者能够通过 ROP（return oriented programming）等攻击方式执行任意代码，伪装与其余参与方的正常通信，远程依赖方将无法感知到已经发生的攻击，为攻击者横向移动（lateral movement）提供了便利.

因此，现有的 TEE 并非完全可信，对其进行动态完整性监控、及时发现可能的入侵、捕获并报告 TEE 的异常行为，在一些安全敏感的场景中是必要的.

2.2 TEE 与动态完整性监控间的壁垒

现有的完整性监控方案难以直接应用于 TEE. 对于需要修改编程模型的用户级 TEE 而言，所有的方案都要进行源码级的重构，并重新评估其内外接口通信的安全性. 对于系统级 TEE，也存在与原有环境中在保护目标和安全能力上的区别.

对于保护目标而言，一方面，已有的动态完整性监控方案大多设计来保护 OS 内核的运行时完整性，而在 TEE 中，动态完整性监控的保护目标则主要是部署到该环境中的工作负载. 另外，系统级 TEE 中的可信 OS 内核往往也进行了定制化裁剪，保留最小化的功能以缩减 TCB，甚至使用形式化验证的内核[48]，

因此也难以直接进行集成. 另一方面, 已有的能够保护用户进程的动态完整性监控方案一般基于内核特定功能[39], 或仅针对嵌入式等特定形态系统上的应用[42,45-47], 不适用于 TEE 可信工作负载保护. 即使部分适用于复杂系统的方案[43]也存在与 TEE 安全模型不契合的问题, 对度量过程和结果的保护不足.

对于安全能力而言, 除 CFA 方案外, 传统的动态完整性度量方案仅完成本地防护, 不适用于机密计算硬件 TEE 的信任模型. 与机密计算平台现有的度量和证明想要达到的目标一致, 若引入动态完整性度量这一新的安全特性, 也需要其具有向远程依赖方证明自身安全属性的能力. 因此, TEE 中的动态完整性监控需要引入运行时证明过程. 与保护目标中所描述的问题类似, 传统的 CFA 方案需针对 TEE 安全模型进行重新设计. GuaranTEE[11]以云计算微服务为粒度, 通过提取 CFG 和插桩记录程序实际执行的控制流. SGXMonitor[12]在 GuaranTEE 的基础上增加了对飞地生命周期状态以及部分函数上下文的报告, 以覆盖更多可能的攻击类型[49-50]. Burmester 等人[51]则通过硬件调试接口获取 TEE 进程和 OS 控制流, 并基于二进制提取的 CFG 进行验证. 然而这些方案仍有不足, GuaranTEE 需预先在 TEE 下收集应用的合法控制流, 并基于类似黑/白名单的机制进行验证, 无法保证控制流基线模型的覆盖率; SGXMonitor 使用无约束符号执行和不敏感静态分析从飞地二进制文件中提取信息而非源码, 存在符号执行路径爆炸和高级抽象语义丢失的精度问题; Burmester 等人[51]基于 ARM Coresight 硬件调试接口, 难以迁移到其它机密计算硬件平台. 此外, 这些方案对其动态度量、远程证明关键过程的保护也存在不足, 信任根与原有 TEE 高度耦合, 信任模型仍可以进一步优化.

2.3 机密计算信任模型的改进空间

机密计算信任模型刻画出了机密计算各参与方相互间的信任关系. 以云计算场景为例, 目前较为通用的机密计算信任模型是, 在平台使用者(即远程依赖方)不信任计算服务提供者(即云服务商)的情况下, 由平台提供者(即硬件厂商)为平台使用者的数据提供机密性和完整性保证, 并为其提供相应的安全性验证能力. 机密计算平台的使用可以是多方参与的, 各个参与角色对其他参与方以及机密计算服务没有信任前提, 但需要对硬件厂商(例如 Intel, AMD, ARM 等为代表的上游制造商)达到无条件信任. 通过由机密计算 TCB 实现的安全机制, 各参与方可以验证机密计算服务的计算环境完整性和真实性, 确保只有授权的计算程序可以在 TEE 中使用己方的机密数据, 以此建立对机密计算服务的信任, 并最终将信任扩展到计算执行时数据和代码的机密性、完整性以及计算结果的正确性.

然而, 已有工作[52-53]指出, 在此信任模型下, 用户对 TEE 硬件厂商信赖过高, 所有度量和远程证明仅依赖其软硬件栈和服务提供, 且其执行过程对用户不透明, 难以判断其是否正确执行; 硬件厂商提供的中心化远程证明服务也容易发生单点失效等问题. 对此, 现有研究主要通过解耦机密计算与硬件提供商的信任强相关来缓解此类问题. OPERA[52]基于 SGX 提出了一种开放式远程证明平台, 可以作为 Intel 证明服务(Intel provisioning service, IPS)和 IAS(Intel attestation service)的替代. 尽管 OPERA 沿用了 SGX 原有的信任根和信任链, 但从证明服务上削减了原有的能力, 阻止其在进行远程证明时获知平台使用者的隐私信息, 也缓解了中心化服务容易发生单点失效的问题. MAGE[53]提出了一种不依赖于可信第三方在 2 个 TEE 之间建立信任的身份派生机制, 允许 TEE 通过该机制获知其他受信任 TEE 的身份, 且不需要硬件厂商给出对于各自的证明. 尽管 MAGE 仍然需要硬件保持诚实并提供相应的度量机制, 但已经消除了硬件信任根的限制, 降低了对硬件厂商及其服务的依赖. Apache Teaclave①通过对 TEE 进行第三方审计来保证 TEE 之间的互信问题, 受信任的审计者使用公钥为审计后的 TEE 颁发签名证书, TEE 需要在编译时配置其所信任的其他 TEE 的证书, 以达成相互之间的认证. 这一方案使用引入了新的无条件信任方的方案来削弱对原有硬件提供商的信任程度.

除原生的机密计算平台外, 一些第三方的机密计算平台也被提出, 以解耦底层硬件和 TEE 安全应用间的关联. 如 VSGX[54]提出了一种在 SEV 上虚拟化 SGX 的技术, 利用 SEV 提供的安全特性实现与 SGX 相近的安全保证, 并允许 SGX 程序在 AMD 平台上兼容运行. UniTEE[55]和 HyperEnclave[56]都提出了在不同硬件平台上统一的编程模型, 使得可信工作负载脱离底层硬件 TEE 架构, UniTEE 还通过预先为不同平台编译目标二进制程序以及提供的寄存器和栈帧翻译机制, 实现对飞地应用程序在多平台间进

① https: //teaclave. apache. org/docs/mutual-attestation

行动态迁移的支持. HyperEnclave、Hyper-V[①]等方案还对不依赖于特定底层安全架构的 TEE 构建方法进行了研究，基于虚拟化和 TPM 在传统平台上提供机密计算能力，避免绑定特定的硬件平台，显著降低了平台使用者对硬件厂商过高的信赖.

然而，这些方案的信任模型仍存在改进空间. 首先，仅改变 TEE 编程模型不影响底层硬件信任根及其提供的安全能力，并不会对原有信任模型做出根本性的改变. 这些平台仍需沿用可信工作负载实际运行平台上的远程证明等服务，以对外证明其安全性. 此外，引入额外软件组件以兼容不同硬件平台，还会导致攻击面的扩大. 其次，不沿用原有信任根，而依靠 TPM 的方案不能完全解决与云服务商之间的信任问题，难以保证其 TCB 不受云服务商控制，存在信任模型上的隐患. 最后，这些方案也没有引入新的安全机制，以缓解前述 TEE 运行时安全性问题，并对现有的信任模型进行限制.

3 研究目标和威胁模型

3.1 研究目标

综合上述问题，本文提出基于动态完整性度量的机密计算运行时监控方案，旨在弥补现有 TEE 所存在的安全性上的不足，同时改进现有机密计算的信任模型. 本文的研究目标具体有 3 个：

1）解决向系统级 TEE 中引入动态完整性度量的技术挑战，基于对 TEE 控制流和数据流的捕获，提取可信工作负载的动态完整性度量值，弥补静态完整性度量的不足，且能被基线模型有效验证.

2）设计 TEE 的运行时远程证明过程，实现机密计算运行时监控方案，提升可信工作负载的运行时安全性. CFA 和 DFA 已被证明是保护程序和系统运行时安全的有效手段. 为应对特权敌手，本文针对 TEE 和传统环境的差异，重新设计了面向 TEE 的 CFA 和 DFA 方案.

3）引入独立的 TCM/TPM2 作为信任根，将新引入的动态完整性度量、运行时证明安全机制与 TEE 原有的信任根解耦，确保在不影响原机密计算的信任模型和安全能力的前提下，打破上游制造商对于机密计算安全保证的垄断. 新机制还能对 TEE 原有安全机制的正确执行加以验证，独立地保证机密计算平台上可信工作负载的安全性.

3.2 威胁模型

本文方案基于现有的系统级 TEE 设计，因此沿用了系统级 TEE 的威胁模型. 此外，方案还引入了具有对 TEE 发起运行时攻击能力的敌手，以示意其设计如何抵御运行时攻击. 方案的威胁模型有 4 种：

1）敌手可以控制机密计算平台上，TEE 以外的特权组件，包括不受信操作系统、虚拟机监视器等，并具有完全的网络控制能力.

2）敌手可以利用 TEE 程序接口和潜在的漏洞，对可信工作负载发起运行时攻击，例如基于内存损坏漏洞的控制流劫持.

3）机密计算平台的 TEE 硬件是诚实的，即敌手不会是硬件制造厂商本身，也无法利用 TEE 硬件发起攻击. 然而正如前文所提，TEE 已有安全机制存在能力上的不足，远程依赖方无法判断 TEE 是否遭受运行时攻击.

4）引入了新的信任根，包括机密计算平台上的 TCM/TPM2 芯片和与其相关的可信验证方. 远程依赖方根据验证方返回的 TEE 运行时证明报告评估 TEE 的可信度.

本文方案不考虑系统级 TEE 所面临的侧信道威胁，也不考虑拒绝服务攻击以及物理攻击. 本文方案可以与这类问题的解决方案结合使用以缓解这些攻击.

4 架构设计

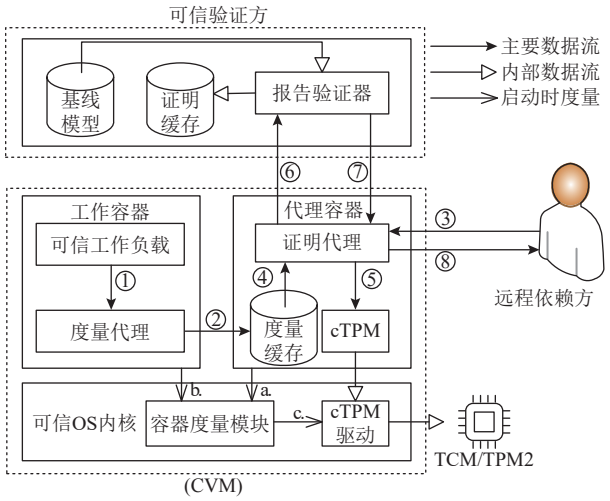
4.1 总体架构

如图 1 所示，本文方案主要包含 3 个实体，分别为机密虚拟机（CVM）、远程依赖方和可信验证方. 机密虚拟机即机密计算平台上的系统级 TEE. 其中运行的组件主要包括 5 种：

1）工作容器. 接受机密计算集群调度的工作负载容器，负责真正执行受保护的计算任务. 除了原有的可信工作负载之外，一个度量代理将会被集成到工作容器，使工作容器具备动态完整性度量能力. 度量代理负责跟踪可信工作负载的程序控制流和数据流，生成度量值并将其发送至代理容器.

2）代理容器. 代理容器被部署到运行工作负载容器的 CVM 中，使得该 CVM 具备运行时证明能力. 代理容器引入了 3 个子组件，包括接收度量值的度量缓存、负责各方通信的证明代理和为度量值提供签名的机密可信平台模块（confidential trusted platform

① https: //docs. microsoft. com/virtualization



①运行时度量 ②转发 ③证明请求 ④本地验证 ⑤签名 ⑥证明报告 ⑦证明结果报告 ⑧证明响应 a,b 启动时度量 c 签名

Fig. 1 Architecture of the runtime monitoring scheme in confidential computing

图 1 机密计算运行时监控方案架构

module, cTPM).

3) 可信 OS 内核. 该内核是系统级 TEE 的必要组成部分, 负责 CVM 内可信工作负载和其它 TEE 内的用户程序的资源分配和调度, 并提供基本的系统调用支持. 该内核集成了一个 cTPM 驱动模块, 通过映射和封装外部 TCM/TPM2 设备, 负责在 CVM 可信启动和动态完整性度量值签名时, 提供访问机密计算平台上 TCM/TPM2 的接口. 此外还有一个容器度量模块, 在机密容器加载时, 对其进行内存完整性度量.

4) 可信验证方. 可信验证方运行于不同平台的安全环境中, 包含报告验证器、基线模型和证明缓存 3 个子组件. 报告验证器接收证明代理发送的证明报告(quote), 验证并返回自身签名后的证明结果报告(report). 该证明结果即为远程依赖方评估 TEE 是否具有运行时完整性的凭据. 在此过程中, 基线模型为报告验证器提供对应工作负载的合法控制流、数据流信息, 以验证证明报告中的实际度量值是否正确. 所有的证明结果报告还会被存入证明缓存中. 在工作容器启动后, 除第 1 次证明外, 每一次证明结果都将包含上次证明结果的散列值.

5) 远程依赖方. 远程依赖方是机密计算平台的远程用户, 即云服务商的租户, 具有 CVM 的所有权, 并信任机密计算平台硬件厂商以及可信验证方. 远程依赖方会在工作容器启动后任意时刻向 TEE 中的证明代理发起运行时证明请求, 以获取证明响应. 证明响应中包含该次请求的证明结果、远程依赖方基于该结果及其签名来评估 TEE 是否具有运行时安全性.

本节后续部分将分别进一步介绍本文方案引入的子组件的架构、功能和安全设计, 以及动态完整性运行时证明流程.

4.2 度量代理

度量代理负责提取可信工作负载的控制流和数据流信息, 生成度量值, 并发送至代理容器的度量缓存队列中. 这一过程是持续的, 即当工作容器启动后, 一旦可信工作负载开始执行, 将会有持续不断的控制流和数据流度量值被生成并发送到度量缓存中. 度量代理被直接集成到可信工作负载的同一进程中.

4.2.1 控制流度量

以最简单的控制流度量过程为例, 当可信工作负载发生地址跳转时, 度量代理根据跳转前后的指令地址, 生成一条控制流度量值(control flow measurement, CFM), 之后重复该过程. 具体而言, 在程序的控制流图中, 程序通过基本块(basic block, BB)进行划分, 每个基本块具有唯一的入口点和出口点, 标志基本块结尾的指令为终结指令(terminator), 一般为条件/无条件跳转、间接跳转或函数返回(ret)指令. 除终结指令外, 函数调用(call)指令也会导致控制流转移. 若 2 个基本块之间存在边, 则程序存在可能的控制流路径. 因此, 度量代理会为可信工作负载的每个基本块分配一个一次性使用的 *ID*, 并在实际运行期间记录控制流转移源基本块的 *ID*、控制流跳转指令地址 $addr_{\text{from}}$ 及其目的地址 $addr_{\text{to}}$, 就可以得到一条控制流度量值. 该 *ID* 由度量代理维护, 在初始时为基本块起始地址的散列值 $ID=H(addr_{\text{BB}})$, 并在第 2 次执行时更新为与其目的地址的散列值 $H(ID|addr_{\text{to}})$. 这一变化的 *ID* 可以防止敌手对其进行篡改. 基本块起始地址是包含地址空间随机化影响的实际值, 而 $addr_{\text{from}}$ 和 $addr_{\text{to}}$ 则是不包含基地址的偏移值, 因此可以在保证每次可信工作负载启动时, 同一基本块都会生成不同 *ID*, 防止重放攻击的前提下, 基线模型还能正确验证该控制流度量值.

如图 2 所示, 左边是可信工作负载的部分控制流图, 其中实线部分是其实际执行的控制流路径. 右边展示了度量代理按顺序记录的 CFM. 当工作负载控制流从 BB_0 转移到 BB_1 时, 度量代理生成了度量值 CFM_0, 后续过程与之类似. 当出现循环时, 如当前控制流在 BB_7 到 BB_8 的路径上经过了 3 次, 而在每次返回已执行过的基本块时, 其 *ID* 需要在上次执行完成后更新, 以区分同一个基本块的多次执行, 如 BB_7 在第一次执行后更新为 $BB_{7'}$, 在第 2 次执行后更新为 $BB_{7''}$.

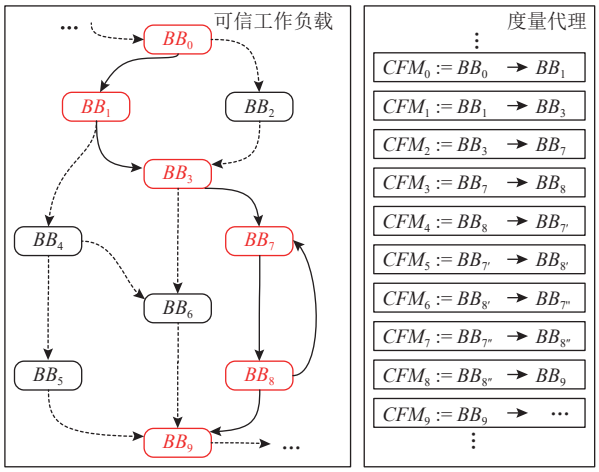


Fig. 2 Trusted workload control flow and measure agent

图 2 可信工作负载的控制流与度量代理

需要注意的是，该过程展示了冗余的控制流记录，实际系统中无需记录所有的控制流转移，以降低对系统性能造成的影响. 为实现动态完整性度量，需要记录的是关键的控制流转移，如间接跳转、函数调用、函数返回. 其中，间接跳转的目的地址是由程序执行期间动态生成的，地址值为某一寄存器中的值，而非直接跳转的固定地址，因此容易遭受攻击；函数返回的地址则是记录在程序栈中，亦容易遭受栈溢出攻击导致控制流劫持；而函数返回度量值的验证需要用到函数调用指令的度量值. 这些关键控制流在程序编译时由编译工具根据指令语义识别并插桩，并由度量代理在运行时捕获并记录. 此外，度量代理将在运行时获取自身代码段地址，并比对控制流目的地址来识别是否有跳转到自身非正常入口的非法控制流，以防止来自工作负载的运行时控制流攻击.

4.2.2 数据流度量

当可信工作负载发生内存访问时，度量代理通过监控相应访存指针的地址，生成一条数据流度量值(data flow measurement, DFM). 具体而言，在程序的数据流图中，通过分析程序敏感变量的定义-使用(define-use)关系，可以分辨影响该变量取值的数据来源，也可以分辨该变量的值的数据流向. 对于可信工作负载，其内存中的部分数据是敏感的，需要确保不能出现从该地址范围内流向 TEE 外部的数据流，也不能出现来自外部的数据流对这部分敏感数据造成影响的情况. 因此，度量代理需要监控这部分敏感数据的数据流，记录这些读取(load)或写入(store)指针的实际地址和其发生的指令位置，就可以得到一条数据流度量值.

如图 3 所示，左边是可信工作负载某一基本块中的指令，以 LLVM[①]中间语言(intermediate representation, IR)展示，其中有 2 条内存访问指令，分别为对局部变量 *secret.addr* 的静态单一赋值(static single assignment, SSA)定义以及对全局变量 *key.addr* 的使用. 这 2 个变量都是指向某一内存地址的指针，赋值 *secret.addr* 的变量 *secret* 就是影响 *secret.addr* 所指值的数据流来源，而使用 *key.addr* 所指值的变量 *key* 就是 *key.addr* 所存储值的数据流向. 右边展示了度量代理记录的数据流度量值. 当基本块 BB_0 执行完成后，度量代理生成度量值 DFM_a 和 DFM_b，分别按顺序对应这 2 次访存操作的实际地址. DFM_a 和 DFM_b 将合并为一条结果 DFM_0，对应该基本块本次执行的一条数据流度量值.



Fig. 3 Trusted workload data flow and measure agent

图 3 可信工作负载的数据流与度量代理

与控制流度量过程相似，实际系统中无需记录所有的数据流. 针对 TEE 中的可信工作负载，要实现数据流动态完整性度量，需要标记工作负载中的敏感数据，包括通信密钥、权限数据、用户隐私数据等，然后监控这些敏感数据上的数据流. 敏感数据的地址范围由度量代理在工作负载启动后进行识别. 对于静态变量中的敏感数据地址，度量代理通过程序数据段直接获取，而对于动态分配的堆内存，度量代理将实时跟踪可信工作负载的内存申请和释放，并基于预定义的策略标记申请用作敏感数据存储的内存地址. 此外，度量代理自身使用的所有数据都将标记为敏感数据，以防止来自工作负载的运行时数据流的攻击. 当且仅当访问敏感数据的指令中的指针与外部数据存在数据依赖时，该数据流将被度量代理捕获. 这一数据依赖基于编译工具提取的 DFG 进行识别. 度量代理将对与不同类型敏感数据重叠的数据流指针进行分类标记，以供可信验证方区分可能存在的不同威胁.

4.2.3 度量代理安全性

度量代理由于直接集成到可信工作负载之中，需要保证不引入额外的攻击点，也要对可能来自工

① The LLVM Compiler Infrastructure. https: //llvm. org

作负载的攻击进行防护. 本文方案基于以下设计来保证度量代理的安全性.

首先, 度量代理在受到 CVM 保护的前提下, 并未向 TEE 外部暴露接口, 攻击者无法直接对度量代理发起攻击. 度量代理在工作容器启动时, 与可信工作负载一起被容器度量模块度量, 而容器度量模块又在 CVM 启动时和可信 OS 内核一起受机密计算平台度量和证明. 该信任链保证了度量代理不会在启动前被篡改, 因此不会引入额外的攻击点.

其次, 度量代理在前述设计中, 对自身进行了控制流和数据流隔离, 确保其不会受到利用 TEE 程序接口间接发起的攻击, 以此来保证动态完整性度量关键过程和度量值的安全性.

最后, 在敌手进行攻击时, 非法的目的地址或指针将被记录到一条度量值中. 该度量值记录了当次攻击的异常控制流或数据流, 并将被包含到后续证明报告中. 即使攻击者对可信工作负载攻击成功, 由于基本块 *ID* 的单向链设计, 度量代理也将在此之前生成一条后续无法篡改的度量值. 可信验证方将在报告验证过程中发现该攻击, 远程依赖方也将依据该证明结果对受攻击的 TEE 进行处理.

4.3 证明代理

证明代理从度量缓存中批量读取度量值, 生成运行时证明报告, 并将 cTPM 签名后的报告发送至可信验证方以获取证明结果.

证明代理有 2 种驱动模式: 一是定期检查度量缓存, 当度量值达到一定数量时, 主动生成运行时证明报告并发送; 二是当远程依赖方发起的证明请求到达时, 将立即读取度量缓存中的所有度量值, 生成证明报告并发送. 生成证明报告后, 相应的度量值将从度量缓存中清空.

4.3.1 本地验证

在生成证明报告之前, 证明代理还会对从度量缓存中获取的度量值进行本地验证, 对度量值进行筛选、合并, 从而压缩数据以减轻与可信验证方之间的通信负担. 本地验证保证该合并过程不会导致含有攻击信息的度量值漏报.

以控制流度量值为例, 部分控制流信息验证无需参考基线模型, 因此无需发送至验证方而在本地即可校验, 例如具有直接地址的函数调用指令产生的控制流度量值. 在直接地址函数调用中, 目标跳转地址是固化在程序中的, 攻击者无法进行篡改. 度量代理生成目标跳转地址的目的是验证后续具有间接地址的函数返回指令. 因此, 只需要在证明代理中维护一个本地的影子栈(shadow stack, SS), 以及筛选和验证匹配的函数调用、返回度量值对, 即可将两者进行合并简化. 具体过程如算法 1 所示.

算法 1. 基于影子栈的本地控制流验证算法.

输入/输出: 度量值序列 m_i, m_{i+1}, m_{i+2}, …

① *ss*. *clear*();
② for m_i do
③ if m_i. *isDirectCall*() then
④ *ss*. *push*(m_i);
⑤ else if m_i. *isReturn*() then
⑥ for *ss*. *notEmpty*() do
⑦ m_{call} := *ss*. *pop*();
⑧ if m_i. *RetTo*(). *equals*(m_{call}. *nextInst*()) then
⑨ if m_{call}. *Callee*. *equals*(*mi*. *locateFunc*()) then
⑩ m_{call}. *combine*(m_i);
⑪ end if
⑫ end if
⑬ end for
⑭ end if
⑮ *i*++;
⑯ end for

除影子栈完成的匹配简化外, 若在 2 个度量值之间, 可以唯一确定 1 个或多个度量值, 这些度量值都将被简化. 如在图 2 中, 通过 CFM_0 和 CFM_2 可以确定 CFM_1 的值, 而通过 CFM_3 和 CFM_5 则可以确定 CFM_4 的值. 未简化的控制流度量值将不受影响, 包括原本可以匹配, 但被清空度量缓存所分割的 2 个函数调用、返回度量值. 被简化的函数度量值控制流信息将被清空, 仅保留其所在基本块的 *ID*, 以防止敌手重用该 *ID* 发起攻击.

4.3.2 报告生成

证明代理在本地验证和度量值合并后, 首先计算所有度量值的散列值, 包括容器度量模块在代理容器和工作容器启动时生成的度量值, 并使用机密可信平台模块 cTPM 对其进行签名, 使得可信验证方可以认证报告的来源. 所有度量值和该签名即为此次运行时远程证明报告的主体, 被发送至可信验证方.

4.3.3 证明代理安全性

本文方案基于以下设计来保证证明代理的安全性.

首先, 与工作容器相同, 代理容器也受到 TEE 和容器度量模块所建立的信任链保护, 会在启动时被度量, 并在运行时被证明. 在 CVM 内, 其仅接收由度

量代理传输的度量值记录而没有其余接口，无法直接成为敌手的攻击目标.

其次，证明代理与外部的信道均为可信信道. 远程依赖方、可信验证方在与 CVM 内的证明代理通信时，会通过机密计算平台已有的远程证明过程对 TEE 进行验证，以此识别证明代理是否被正确运行在目标机密计算平台上. 代理容器在部署前也会集成可信验证方和远程依赖方的证书，以进行相应的身份认证.

最后，证明代理运行于 CVM 中单独的代理容器中，证明容器本身与工作容器相隔离，仅从度量缓存间接获取度量代理发送的度量值，即使受到攻击，也不会影响可信工作负载的运行时安全.

4.4 机密可信平台模块（cTPM）

在机密计算场景下，由于主机操作系统和虚拟机监视器不再受到信任，传统的基于可信硬件模块（如 TCM/TPM2）的可信启动和远程证明过程已不符合安全模型，因此无法直接使用这些可信硬件来对 CVM 的启动和运行进行度量证明. 然而，仅使用该硬件模块对平台进行标识，生成签名密钥 *tpm_sk*，并对运行时证明报告进行签名依然是可行的，因为该签名密钥无法导出，因此不会被特权敌手获取，增加了敌手的攻击成本. 远程依赖方使用一个受 TEE 保护的秘密随机数 N 作为挑战，CVM 外部的硬件模块不能直接获取到该随机数，而仅能获取 N 与度量值的散列值，防止敌手在控制 TCM/TPM2 硬件后伪造签名.

代理容器中的 cTPM 运行于用户态，是提供给证明代理使用硬件模块的可信软件栈（trusted software stack, TSS）. 该 TSS 仅为证明代理提供其在度量值签名时需要用到的签名接口. CVM 的可信 OS 内核中集成了 cTPM 驱动内核模块，该内核模块封装了外部 OS 映射进 CVM 的设备且仅保留签名接口，并在运行时检查所有调用参数和返回值以避免敌手通过该接口进行攻击.

此外，cTPM 还可以在硬件模块不可用时，提供具有相当安全性的软件机密 TPM 实现. 已有方案[57-58]证明，在机密计算平台上实现软件 TPM，并提供相当的安全能力是可行的.

4.5 容器度量模块

现有的系统级 TEE 仅验证 CVM 启动时的完整性，而缺少 TEE 内部的视角. 已有的机密虚拟机/容器方案[30]仅通过容器代理在 CVM 内部验证容器镜像文件的签名或解密一个被加密的容器镜像，而没有内存中的完整性保护. 为了建立完整的信任链，本文方案向 CVM 中引入了容器度量模块.

该内核模块随可信 OS 内核一起被 TEE 硬件度量和证明，而又在机密容器启动时度量容器内工作负载进程的内存. 当容器代理的镜像加载完成，容器内的进程启动时，容器度量模块在可信 OS 内核中将该进程挂起并计算其启动参数和代码段的所有内存映射的散列值，该散列值即为容器工作负载启动时的内存度量值. 这一静态度量值是本文方案 CFI 和 DFI 动态完整性度量值的“起点”，因为 CFI 和 DFI 的度量是增量的，需要一个起始时的状态信息. 度量完成后，该度量值将被扩展至 cTPM，以完成机密容器的可信启动. 在后续执行运行时证明的时候，该度量值将被包含到证明代理调用 cTPM 生成的签名中，作为证明报告的部分内容.

4.6 可信验证方

可信验证方处于单独的可信环境中，可以是一个具有可信环境的独立服务器或其他受证明的 TEE. 在运行期间，当可信验证方与证明代理通过相互的身份认证，建立可信信道后，报告验证器对于接收到的运行时证明报告，还需要进行 2 个验证：

1）验证证明报告中的 TCM/TPM2 签名，包括对度量值 m 签名的验证以及对由证明身份密钥（attestation identity key, AIK）认证的签名密钥 *tpm_sk* 的验证. 可信验证方持有 TCM/TPM2 的背书密钥（endorsement key, EK）以及 AIK 的相应证书，可以认证该签名是否由对应可信硬件模块生成. 该身份认证过程和可信信道建立时的身份认证相互分离，具有解耦的信任根和信任模型，独立地保证 TEE 的安全性.

2）根据基线模型验证证明报告中度量值 m 的合法性. 这一步是对动态完整性的验证，以判断可信工作负载是否运行时安全.

验证完成后，可信验证方将根据该证明报告对应可信工作负载的前序证明结果报告、当次证明结果和度量代理签名，生成当次证明结果报告，并将该签名后的结果返回. 该结果也会被存入可信验证方的证明缓存中供后续证明过程使用. 证明结果报告中包含当前已被验证的控制流进度，对于远程依赖方而言，可以根据该证明结果报告确认在此进度之前的可信工作负载的运行时安全性. 除首次证明结果外，每个证明结果报告将嵌套上次证明结果的散列值 $H(report_{\text{pre}})$，形成证明结果链，以防止敌手进行重放攻击.

4.6.1 度量值验证

报告验证器对度量值的验证是运行时证明的关

键步骤. 合法的控制流度量值需满足 3 个条件:

1)对于单条度量值, 其所记录的 *ID* 需符合该基本块上次执行后的更新值. 报告验证器维护一个基本块 *ID* 列表, 在每个基本块执行后, 其对应 *ID* 按 4.2.1 节所述方式进行更新. 当度量代理再次生成该基本块的控制流度量值时, 新 *ID* 应符合由上次度量值计算的 $H(ID \mid addr_{to})$. 报告验证器可根据 $addr_{from}$ 确认度量值的源基本块和 *ID* 的对应关系.

2)对于单条度量值, 其所记录的 $addr_{to}$ 应为合法的基本块入口地址. 这些合法的基本块入口记录在基线模型中, 每条 $addr_{from}$ 对应的终结指令具有特定的入口地址列表.

3)对于具有顺序关系的多条度量值, 其顺序应符合合法控制流路径, 包括被 2 次运行时证明分割的度量值. 这些合法路径也记录在基线模型中, 即可信工作负载的 CFG. 具体而言, 当前一条度量值验证完成后, 报告验证器将标记度量值对应的 CFG 节点, 后一条度量值仅能出现在该节点可达的后续节点中.

合法的数据流度量值需满足 2 个条件:

1)对于单条度量值, 其所记录的指针访存地址不能是度量代理所使用的地址, 因为这将打破数据流隔离而影响度量关键过程的安全性. 若该指针是对其它位置敏感数据的访问, 则根据基线模型进行判断. 每条访存指令具有特定的合法访问范围, 这些合法地址范围也记录在基线模型中.

2)对于连续的度量值, 由于数据流度量值也记录了其源基本块 *ID*, 且与其它度量值具有顺序关系, 因此也需要基本块 *ID* 的匹配, 且需要符合控制流验证中多条度量值之间的合法控制流路径关系.

4.6.2 基线模型

基线模型在可信工作负载编译前后提取. 本文方案基于 LLVM 编译架构, 从工作负载在编译阶段的不同语言层级提取基线模型, 包括 IR 层级和二进制层级, 从而减少语义信息的丢失. LLVM 支持将不同语言的源码编译为 IR, 在该层级统一了不同编程语言表示, 并使用 DWARF 调试信息最大化保留了上层语义, 而后 LLVM 将 IR 编译为二进制文件, 在该层级可以提取到目标平台上精确的指针位置和地址偏移.

在 IR 层级, 本文方案提取可信工作负载的 CFG 和 DFG. 对于控制流, 基线模型通过 IR 的终结指令中携带的所有可能的跳转目标基本块, 标识 CFG 中每个节点终结指令的合法跳转入口信息. 对于数据流, 基线模型通过 DFG 分析所有访存指令(load 和 store)与可信工作负载输入输出数据的定义-使用及其它数据依赖关系, 标记存在数据依赖关系的指针, 并根据用户策略定义其合法的访存数据. 在二进制层级, 本文方案提取 IR 层级中已标识的所有跳转入口、敏感数据等的具体偏移地址, 形成最终的基线模型.

此外, 远程依赖方还可以基于更深入的程序分析结果来定制基线模型, 如进行上下文敏感、路径敏感的控制流提取等, 以实现具有不同安全需求的运行时证明策略.

当可信工作负载程序变更时, 基线模型也需要进行相应更新. TEE 用户需重新发布并部署工作容器, 而无需对 CVM 进行更改. 基线模型在可信工作负载编译时生成, 并动态更新至可信验证方. 更新的工作容器启动前将被容器度量模块再次度量, 度量缓存和证明缓存都将被清空. 工作容器重启后, 各方将重新建立可信信道, 并进入新的运行时证明流程.

4.7 运行时证明流程

图 1 中的编号和箭头展示了动态完整性度量和运行时证明的示例流程.

在工作容器启动后, 箭头①即为度量代理对可信工作负载的持续的控制流和数据流度量过程. 该过程不断生成的度量值, 将通过②发送至代理容器的度量缓存中. 在任意时刻, 远程依赖方通过③可以发起运行时证明请求, 证明代理在接收到该请求后, 通过④读取度量缓存中的所有度量值并进行本地验证和度量值合并, 使用⑤对度量值和随机数挑战进行签名, 并通过⑥发送证明报告至报告验证器. 报告验证器验证报告签名者的身份, 并使用基线模型和先前的结果验证所有度量值, 通过⑦生成签名后的证明结果返回给证明代理. 证明代理通过⑧再将这一结果作为运行时证明响应返回给远程依赖方, 最终完成了整个运行时证明过程.

图 4 还展示了在远程依赖方发起运行时证明请求后, 各参与方之间消息传递的序列图. 首先, 远程依赖方生成秘密随机数 *N*, 将其和远程证明目标 *target* 作为请求消息 *Req* 发送至证明代理. 证明代理接受该输入, 从度量缓存中获取该目标的度量值 *m*, 计算 *m* 和 *N* 的散列值后请求 cTPM 生成签名 *s*. 此步骤中证明代理还将确认 *target* 的真实性和 cTPM 的可用性, 若 *target* 不存在或 cTPM 无法正确生成 *s*, 将返回远程证明失败的结果. 而后, 证明代理将 *m*, *s*, *target*, 以及 cTPM 签名密钥证书 *Cert* 作为当前远程证明报告 *quote* 发送至可信验证方. 可信验证方从基线模型数据库中读入 *target* 对应的基线模型 *b*, 并验证 *m* 的正确性, 生成验证结果 *result*. 当 *b* 不存在或 *m*

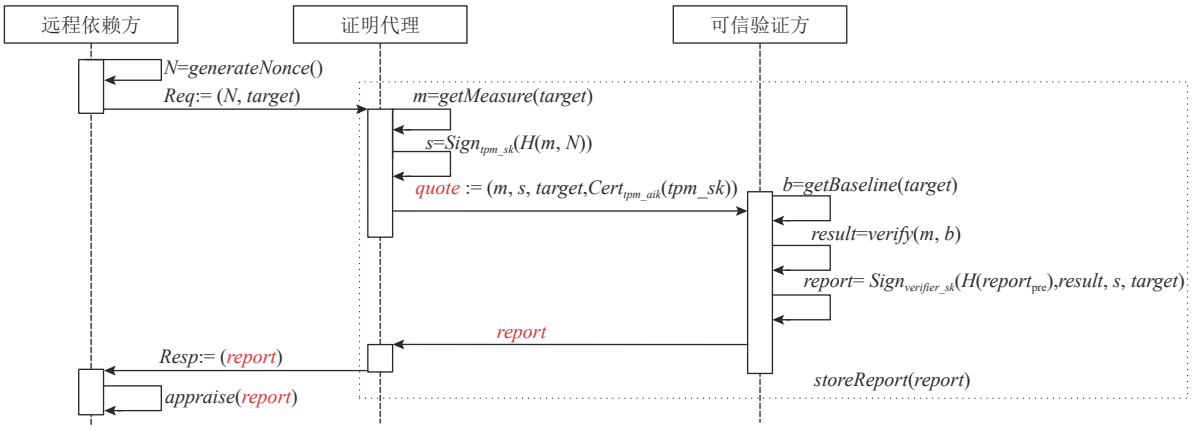


Fig. 4 Message sequence diagram of runtime attestation for dynamic integrity measurement

图 4 动态完整性度量运行时证明消息序列图

验证失败时，*result* 将记录失败值. 之后，可信验证方从证明缓存中读取前序验证结果报告 $report_{pre}$，将其散列值和 *result*, *s*, *target* 一起签名并作为当前证明结果报告 *report* 返回至证明代理，作为证明代理对 *Req* 的响应 *Resp* 返回至远程依赖方. 最后，远程依赖方将根据 *report* 和预定义的安全策略对 *target* 的可信度和安全性进行评估. 其中，虚线框中的部分也展示了证明代理主动发起的运行时证明流程，此时随机数 *N* 由证明代理生成.

运行时证明流程通过受加密保护的可信信道完成，该可信信道基于硬件 TEE 已有的远程证明服务实现，包括通信双方的相互认证和通信密钥交换. 此外，通过身份认证、消息加密和秘密随机数，也能防止中间人攻击、重放攻击等网络攻击.

5 实现和评估

5.1 架构实现

本文方案基于 KATA Containers 开源架构完成了原型系统. KATA 是虚拟机/容器的代表性方案，支持基于 Kubernetes(k8s)[①]部署和调度基于 CSV/SEV 的机密虚拟机/容器.

在原型系统中，工作容器和代理容器运行于 k8s 的同一个 Pod，该 Pod 即为一个 CVM. 度量代理在编译阶段通过 LLVM IR 插桩集成至可信工作负载，并打包在工作容器镜像中. 代理容器编译为单独的容器镜像. 其中，度量缓存为同一 Pod 内 2 个容器的共享内存卷. cTPM 基于微软 TPM 2.0 模拟器实现，其 AIK 和 EK 的证书由可信验证方直接颁发. 实验还测试了基于硬件 TCM 的可行性，型号为国民技术 Z32H330TC. 证明代理、报告验证器和远程依赖方之间基于 curl 和 Apache httpd 服务进行 HTTPS 通信，并验证相应的证书. 表 1 展示了本文完成时原型系统各组件的代码行数(不包括使用的源码库)，基于 VSCode Counter 进行统计.

Table 1 Line of Code in the Prototype

表 1 原型系统代码行数

组件	代码行数	实现语言
基线模型提取工具	3 819	C++/Python
代码插桩工具	737	C/C++
度量代理	347	C/C++
证明代理	378	C/C++
容器度量模块	1 138	C/C++
报告验证器	320	C/C++

5.2 安全分析

本文方案对该原型系统在面临来自特权敌手的威胁，包括内存漏洞利用等运行时攻击所能达到的安全性进行了分析.

在 CVM 启动前，敌手无法通过篡改 CVM 镜像发起攻击，因为该过程受 CSV 硬件度量和证明能力的保护. 此外，机密容器镜像的编译和发布均在可信的环境中进行，敌手也无法进行供应链攻击. 机密容器镜像发布后还将被加密保护，直到被调度到 CVM 时，再通过 CSV 远程证明在 CVM 中获取解密密钥解密并运行. CSV 远程证明过程不会被本文方案所影响，因而不会引入额外的攻击面.

在机密容器镜像被调度到 CVM 并启动后，敌手

① https: //kubernetes. io

可以通过 TEE 的对外接口发起运行时攻击. 然而, 方案的架构和安全性设计(如第 4 节所述)保证了敌手对于可信工作负载的运行时攻击, 将导致控制流或数据流的异常, 产生携带攻击信息的控制流或数据流度量值被可信验证方检测到, 影响运行时证明报告结果. 度量值仅受程序实际的控制流或数据流影响, 而与攻击载荷的具体形态无关. 远程依赖方将根据该结果和云服务商所提供的功能, 选择不同异常处理策略, 如重置 TEE 或重新发布可信工作负载. 即使敌手对于可信工作负载的控制流劫持成功, 也无法影响到另一容器度量缓存中已生成的度量. 敌手可以在网络中阻止证明代理和可信验证方之间的通信, 进而阻止该异常度量值被可信验证方检测, 但后续可信验证方生成的证明结果也将停留于已验证的控制流, 用户将不信任包括异常度量值在内的未被验证的控制流.

原型系统还向 TEE 引入了 2 个额外的接口, 即证明代理与远程依赖方或与可信验证方之间的可信信道. 对于硬件可信模块方案, 还引入了一个映射到 CVM 内部的设备接口. 这些额外接口都是经过严格检查、功能受限或受到认证和加密保护的, 且无法直接影响可信工作负载. 此外, 单向一次性基本块 *ID*、秘密随机数挑战、嵌套证明结果链的设计也防止了敌手进行重放攻击.

控制流和数据流度量以及验证的完备性和正确性, 也会影响本文方案的安全性. 在原型系统中, 度量代理通过 LLVM IR 插桩监控了所有易受到内存损坏漏洞利用的攻击点, 包括所有函数调用和返回过程、所有间接控制流跳转等, 还监控了与接口具有数据依赖性的所有内存访问指针, 有效保证了度量的完备性. 该原型系统基线模型所包含的 CFG 和 DFG 均在未丢失语义信息的情况下提取, 也能够有效保证其验证过程的正确性. 表 2 展示了本文方案与相关工作中其他 TEE 完整性保护方案部分特性的对比.

表 2 中, 原生方案指机密计算平台基于 TEE 的硬件度量和证明方案. 相比于其他方案, 本文方案使用了多语言层级结合的语义分析方法和过程间程序依赖性分析技术, 特别是引入了对 DFI 的动态完整性度量和证明, 在安全性方面具有显著提升. 本文方案还提出了从 TEE 硬件信任根到机密 VM 工作负载运行时的完整信任链建立方法, 该方法可以为运行中的程序提供基于密码学的可证明安全性. 此外, 本文方案还引入了 TCM/TPM2, 解耦了原生方案与运行时证明方案的信任模型, 使得运行时证明方案不依赖于单一的 TEE 信任根.

Table 2 Comparison of Schemes on TEE Integrity in Confidential Computing

表 2 机密计算 TEE 完整性方案比较

方案	信任根	目标 TEE	完整性	远程证明	基线模型
本文方案	CPU/TPM	CSV	CFI/DFI	运行时	CFG/DFG/二进制
原生方案	CPU/SoC		静态	启动时	二进制
CoCoTPM[57]	CPU/TPM	SEV	可信启动	运行时	二进制
SVSM-vTPM[58]	CPU/TPM	SEV	可信启动	运行时	二进制
SGXMonitor[12]	CPU	SGX	CFI	运行时	CFG/二进制
GuaranTEE[11]	CPU	SGX	CFI	运行时	控制流路径
Burmester 等人[51]	SoC	Trustzone	CFI	运行时	CFG/二进制

5.3 性能评估

1)实验环境. 对原型系统的性能测试实验主要基于海光 CSV 平台完成, 处理器为 32 核的 C86 7291, 内存 128 GB, 宿主机操作系统为 Ubuntu Server 20.04, Linux 内核版本 5.10.84. CVM 客户机为 CCv0 定制的 Linux 内核版本 5.11.0. LLVM 版本为 17.0.0. 此外, 实验还在 AMD SEV 平台上进行了相同测试以印证方案的实用性和可移植性, 其处理器为 64 核的 EPYC 7763, 内存 128 GB.

2)实验设置. 实验主要使用内存数据库程序 Redis(版本 6.0.5)及其自身的基准测试套件(18 项测试), 以及 CPU 基准测试套件 nbench(10 项测试). VM 设置 vCPU 核心数为 4, 内存 32 GB, 分别在 KATA 普通/机密虚拟机容器中进行重复测试, 探究本文方案对实际应用性能的额外开销, 结果如表 3 和表 4 所示.

表 3 中对测得的数据进行了统计, 并得出方案对 Redis 整体性能的影响. 其中, “普通 VM 总开销”一项指的是在普通虚拟机中, 插桩后的 Redis 程序相比于未插桩的程序引入的额外开销; “CVM 总开销”则是二者在机密虚拟机中的对比. 此外, “动态度量

Table 3 Performance Overhead of the Prototype (Redis)

表 3 原型系统性能开销(Redis)

性能开销	平均值/%	中位数/%	最优值/%	最差值/%	测试耗时/s
普通 VM 总开销	14.65	8.44		71.42	40.5/59.3
CVM 总开销	16.11	10.73		71.48	41.7/60.6
动态度量开销	13.76	9.72		68.86	
通信开销	2.35	1.73		7.50	
CVM 相比普通 VM 开销提升	3.09	2.91	1.64	4.96	

注: Redis 基准测试包含 PING_INLINE, PING_BULK, SET, GET, INCR, LPUSH, RPUSH, LPOP, RPOP, SADD, HSET, SPOP, LPUSH_LRANGE, LRANGE_100, LRANGE_300, LRANGE_500, LRANGE_600, MSET 共 18 项数据库操作, 每项使用 100 000 次请求, 每次 3 B 数据并行访问数据库.

Table 4 Performance Overhead of the Prototype (nbench)

表 4 原型系统性能开销（nbench） %

测试项	动态度量开销	通信开销
Numeric sort		
String sort	24.51	5.18
Bitfield		
FP emulation	2.67	
Fourier	51.12	3.43
Assignment	7.94	
Idea	0.65	
Huffman		
Neural net	19.31	4.60
LU decomposition		

开销”指生成度量值的开销，而“通信开销”指的是度量代理和度量缓存之间度量值转发的开销，此二者均对原程序的性能具有不同程度的影响. 最后实验还观察到了测试过程中 CSV 硬件平台对机密虚拟机内存加密造成的开销，即“CVM 相比普通 VM”中的结果. 可以看到，本文方案对 CVM 中运行的程序，平均额外开销在 16.11% 左右. 其中 13.76% 的平均额外开销由动态完整性度量导致，而 2.35% 的平均额外开销则来自容器间的通信. 在最优情况下（Redis 测试项 PING_BULK, INCR）开销低于 1%，而一些用例（GET, SPOP）甚至存在大于 5% 的负开销，推测是插桩对编译优化造成的干扰所致. 对于最差情况，动态完整性度量导致的最高开销出现在用例 MSET 中，为 68.86%，而通信导致的最高开销出现在 LRANGE 中，约为 7.50%. 最后，在此实验中，在 CSV 上机密虚拟机自身的平均额外开销为 3.09%. 对于高安全要求的应用而言，本文方案引入的开销是可接受的.

另外，实验还测试了度量值转发时，每次通信的度量值批次（batch）的大小对性能的影响. batch 的大小影响通信的频率以及度量进程被中断的时间. 在度量值总数一定时，batch 越大，通信频率越低，但被度量进程等待的通信时间越长. 图 5 展示了 batch 大小与通信开销之间的关系. 实验发现，通过控制 batch 的大小，可以将通信频率和通信时间维持在一个合理的水平，如在本实验中将 batch 设置为 4 MB 大小.

除性能开销外，使用本文方案编译的 Redis 二进制文件大小与原来相比增加了约 1 MB，由 12.9 MB 增加至 13.9 MB，这是由插桩的指令以及链接的度量代理、静态库等导致的. 在以中大型应用为主的机密计算平台上，该开销可以忽略不计. 实验观察到度量

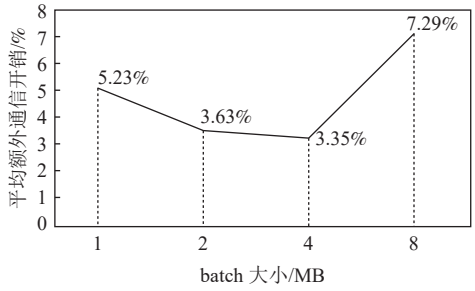


Fig. 5 batch size and communication overhead

图 5 batch 大小与通信开销

代理引入的额外内存开销约为 5.6 MB，主要受 batch 大小的影响.

实验还对本文 4.3.1 节提出的控制流合并压缩率和证明代理的通信效率进行了测试. 图 6 展示了具有代表性的测试数据，包括全部测试项（ALL）、在前述性能测试实验中存在最差通信情况的 3 个测试项（LRANGE_*），以及存在最差性能开销的测试项（MSET）. 可以看到，算法的总体压缩率达到了 32.5%，即节省了约三分之二的后续通信开销. 对于最差通信情况的测试项，算法压缩率可进一步降低至 29.1% 以下. 对于最差性能开销的测试项 MSET，算法的压缩率也有 52.9%. 未被本地验证合并的度量值主要是对系统调用、动态链接库的函数调用指令，以及在不同批次验证而被分开的调用/返回指令对的度量值. 这些指令不会匹配到 4.3.1 节所述的对应指令.

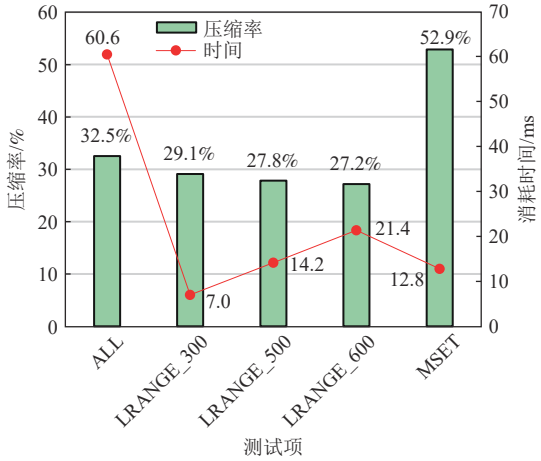


Fig. 6 Compression ratio and time cost of measurement combination

图 6 度量值合并的压缩率和消耗时间

此外，图 6 还展示了在各测试项中，证明代理在处理度量值时消耗的具体时间，该时间与度量值的数量成正比. 以全部测试项为例，使用单线程对全部度量值（约 1.58×10^9 条）进行读取约消耗 5.2 s，而处理仅消耗 60.6 ms，总和远低于测试执行时间 60.6 s. 证

明代理调用 cTPM 签名的速率约为 161 次/秒, 在使用硬件 TCM 的情况下仍可达到约 6.9 次/秒, 满足本文方案运行时远程证明的使用需求. 该结果表明, 在实际运行中, 证明代理中的度量值读取和处理速度大于度量值的生成和发送速度, 对工作负载而言可以达到线速, 不影响其性能. 最后, 实验还对容器度量模块进行了测试. 容器度量模块仅在启动时生效, 实际上不影响工作负载运行时的性能. 即使将其作用于正在运行的进程, 测得的平均性能损耗亦仅有 1.47%.

表 4 对 nbench 的测试结果进行了统计, 本文方案引入的平均额外性能开销约为 11.56%. 其中最差情况为用例 Fourier 上存在的 54.55% 额外开销(包括动态度量和通信开销); String sort 和 Neural net 也分别受到了 29.69% 和 23.91% 的性能影响; 其余用例受影响较小, 且未在 Numeric sort, Bitfield 和 Huffman 中观测到明显(>0.5%)的性能开销. 此外, 实验仅在存在较高开销的 3 个用例(Fourier, String sort 和 Neural net)上观察到了明显的通信开销.

结合上述测试结果可知, 本文方案对工作负载性能的影响与其负载特性(I/O 密集型或 CPU 密集型)没有明显关联, 而是取决于程序运行过程中关键控制流转移、敏感数据访问出现的频率; 另外, 较高的动态度量开销意味着度量值生成的频率更高, 同时可能导致较高的通信开销. 最后, 实验从 SEV 平台上测得的数据中也得到了相同的结论. 其中, 本文方案在 Redis 中引入的平均开销为 14.47%, 在 nbench 中引入的平均开销为 12.38%, 各用例结果与 CSV 平台基本一致.

基于上述结果可以得知, 本文方案在机密虚拟机/容器架构上具备实际使用价值, 在保证了可信工作负载运行时安全性的同时, 引入了合理的性能开销.

5.4 有效性验证

为了说明本文方案的有效性, 并演示原型系统如何进行动态完整性度量, 实验设计了一个存在缓冲区溢出的示例程序来测试异常的数据流如何被捕获并生成度量值. 目标程序代码(C 语言)如示例 1 所示.

在该过程中, 用户输入口令, 程序输出对应角色的密钥, 而仅有输入正确的口令才能获取 root 角色的密钥(行⑥). 然而, *get_secret_key* 函数中存在的缓冲区溢出漏洞, 允许攻击者使用超过限制长度的输入来覆盖栈中判断口令是否正确的 *auth* 变量. 即使敌手无法获知正确的口令, 也能够通过类似如示例 2 的方式(python 语言)实施非控制数据攻击.

该输入将会通过栈缓冲区溢出漏洞将 *auth* 变量覆盖为 1. 目标程序将会错误地将密钥 *root_key* 输出并被敌手窃取. 由于该攻击并未形成非法的控制流路径, 因此也难以被常用的控制流监控手段感知.

示例 1. 有效性验证演示代码.

target. c:

```
① const char * password = "root";
② char* keys[2] = {"user_key", "root_key"};
③ char*get_secret_key(char*input,size_t input_len){
④   size_t auth = 0; //被覆盖
⑤   char input_record[4]; //接收拷贝(行⑨)
⑥   if(strcmp(input, password) == 0){
⑦     auth = 1;
⑧   }
⑨   memcpy(input_record, input, input_len); //缓冲
      区溢出
⑩   return keys[auth];
⑪ }
⑫ int main(int argc, char *argv[]){
⑬   if(argc < 2){
⑭     return 1;
⑮   }
⑯   char* key = get_secret_key(argv[1],
      strlen(argv[1]));
⑰   printf("the obtained secret key is: %s\n", key);
⑱   return 0;
⑲ }
```

示例 2. 有效性验证攻击代码.

payload. py:

```
① #!/bin/env python3
② import subprocess
③ target = "target" # 目标程序
④ argument = "user" + '\1' # 攻击输入
⑤ subprocess. call([target, argument])
```

当目标程序经过本文方案原型系统的插桩后, 其数据流将被捕获. 原型系统复用了文献 [59–60] 工作的过程间程序依赖性分析工具, 实现了对函数 *get_secret_key* 参数和返回值的数据流分析和插桩.

图 7 展示了部分数据流分析和插桩时的输出. 实验将密钥 *root_key* 所在地址标记为敏感数据后, 当原型系统分析出数据流指针与程序输出具有数据依赖性时, 在 *get_secret_key* 函数执行返回指令前, 将插入该指针是否与敏感数据地址范围重叠的验证代码. 若发生重叠, 这一数据流将被记入度量值中. 其中, *get_secret_key* 的返回值 %5 即为目标程序代码中的

keys[*auth*](示例 1 行⑩)，其携带一个指向 *auth* 位置的 load 指针.

```
Find tainted parameter in get_secret_key:
    input
    input_len
    return: %5
Find tainted parameter num: 3
get_secret_key instrumenting.....
    Tainted: input --> get_secret_key:input
    Tainted: input_len --> get_secret_key:input
    Tainted: %4 --> %5
    Tainted: %5 --> %5
main instrumenting.....
    Tainted: argc --> get_secret_key:input
    Tainted: argv --> get_secret_key:input
```

Fig. 7 Process output of data flow analysis and instrumentation

图 7 数据流分析和插桩过程输出

在插桩完成后，当正确运行目标程序时，度量代理将生成 6 条控制流度量值，其中 2 条被本地验证合并(即 *main* 函数对 *get_secret_key* 的函数调用和相应的函数返回)，剩余 4 条分别是对 *strlen*、*strcmp*、*printf* 的调用和主函数的返回控制流. 当发生敏感访问时，还将生成 1 条数据流度量值，如图 8 所示.



Fig. 8 Measurements and their generating location

图 8 度量值与其生成位置

图 8 中，左侧为生成的度量值，右侧为目标程序二进制文件 *get_secret_key* 函数的一部分，为方便演示，度量值中基本块 *ID* 保留了初始值. 其中，箭头①连接了 *strcmp* 函数调用生成的度量值及其产生位置. 箭头②连接了异常数据流指针编号及其实际位置，其在数据流度量值中编号为 1，且与敏感数据范围 1(即 *root_key* 所在地址)产生重叠，在它之前还有一条指令被插桩并验证(lea rax)，但未在运行时检查出重叠，因此没有生成度量值. 箭头③指向该异常数据流度量值的产生位置. 可以看出，原型系统成功地捕获了这一非控制数据攻击. 报告验证器将在运行时证明的过程中进行与此类似的验证流程.

此外，该缓冲区溢出也可能被敌手实现为控制流劫持，如覆盖 *get_secret_key* 函数在栈中的返回地址实现 ROP 攻击. 然而，此时的 *get_secret_key* 返回指令度量值将不能再通过本地验证，运行时证明报告中将包含这一异常结果，最终该攻击也将被检测到.

基于上述有效性验证可以得知，本文方案能够捕获运行时的异常控制流和数据流，生成包含攻击信息的度量值，能提供给验证方检测出对应的运行时攻击，为实现运行时安全监控提供了有效保证.

6 结 论

针对现有 TEE 对运行时安全防护能力不足的问题，本文提出了一种基于动态完整性度量的机密计算运行时监控方案，通过程序插桩技术向 TEE 内用户工作负载中引入控制流和数据流动态完整性度量关键过程，获取工作负载的动态完整性度量值，由独立的 TCM/TPM2 信任根签名后，发送到可信验证方进行验证，最后将结果报告给远程用户，以实现对可信工作负载的运行时完整性证明，将信任链扩展到了 TEE 的全生命周期. 与其他为机密计算平台提供运行时证明能力的方案相比，本文方案具有更全面的运行时防护能力和更准确的验证基线模型. 此外，本文方案还在 CSV/SEV 平台上进行了分析和验证，实验结果表明，本文方案具有良好的运行时完整性度量与证明能力，且对于高安全需求的机密计算平台应用而言，该性能开销也是可接受的.

作者贡献声明： 李为完成论文设计和撰写；冯伟提供思路并完成实验；秦宇和冯登国提供指导并修改论文.

参 考 文 献

[1] State Cryptography Administration. GB/T 29829-2013 Information security techniques- Functionality and interface specification of cryptographic support platform for trusted computing[S]. Beijing: Standards Press of China, 2022 (in Chinese)
(国家密码管理局. GB/T 29829–2013 信息安全技术 可信计算密码支撑平台功能与接口规范[S]. 北京: 中国标准出版社, 2022)

[2] TCG. TPM 2.0 mobile reference architecture specification[S/OL]. Beaverton, OR: Trusted Computing Group, 2014[2024-06-03]. https://trustedcomputinggroup.org

[3] Costan V, Devadas S. Intel SGX explained[J/OL]. Cryptology ePrint Archive, ePrint: 2016/086, 2016[2024-06-03]. https://eprint.iacr.org/2016/086

[4] Wang Wenhao, Chen Guoxing, Pan Xiaorui, et al. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX[C]//Proc of the 2017 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 2421−2434

[5] Cheng P C, Ozga W, Valdez E, et al. Intel TDX demystified: A top-down approach[J]. ACM Computing Surveys, 2024, 56(9): 1−33

[6] Lee J, Jang J, Jang Y, et al. Hacking in darkness: Return-oriented programming against secure enclaves[C]//Proc of the 26th USENIX Security Symp. Berkeley, CA: USENIX Association, 2017: 523−539

[7] Biondo A, Conti M, Davi L, et al. The guard's dilemma: Efficient code-reuse attacks against Intel SGX[C]//Proc of the 27th USENIX Security Symp. Berkeley, CA: USENIX Association, 2018: 1213−1227

[8] Seo J, Lee B, Kim S, et al. SGX-Shield: Enabling address space layout randomization for SGX programs[C]//Proc of Network and Distributed System Security Symp. Reston, VA: The Internet Society, 2017. http://dx.doi.org/10.14722/ndss.2017.23037

[9] Kuvaiskii D, Oleksenko O, Arnautov S, et al. SGXBOUNDS: Memory safety for shielded execution[C]//Proc of the 12th European Conf on Computer Systems. New York: ACM, 2017: 205−221

[10] Mishra S, Polychronakis M. SGXPecial: Specializing SGX interfaces against code reuse attacks[C]//Proc of the 16th European Conf on Computer Systems. New York: ACM, 2021: 48−54

[11] Morbitzer M, Kopf B, Zieris P. GuaranTEE: Introducing control-flow attestation for trusted execution environments[C]//Proc of 2023 IEEE 16th Int Conf on Cloud Computing. Piscataway, NJ: IEEE, 2023: 547−553

[12] Toffalini F, Payer M, Zhou Jianying, et al. Designing a provenance analysis for SGX enclaves[C]//Proc of the 38th Annual Computer Security Applications Conf. New York: ACM, 2022: 102−116

[13] Lei Zhou, Ding Xuhua, Zhang Fengwei. Smile: Secure memory introspection for live enclave[C]//Proc of the 2022 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2022: 386−401

[14] Gu Jinyu, Zhu Bojun, Li Mingyu, et al. A hardware-software co-design for efficient intra-enclave isolation[C]//Proc of the 31st USENIX Security Symp. Berkeley, CA: USENIX Association, 2022: 3129−3145

[15] Li Wenhao, Xia Yubin, Lu Long, et al. TEEv: Virtualizing trusted execution environments on mobile platforms[C]//Proc of the 15th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2019: 2−16

[16] Zhao Wenjia, Lu Kangjie, Qi Yong, et al. Mptee: Bringing flexible and efficient memory protection to Intel SGX[C]//Proc of the 15th European Conf on Computer Systems. New York: ACM, 2020: 1−15

[17] Cloosters T, Rodler M, Davi L. TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclave[C]//Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 841−858

[18] Alder F, Daniel L, Oswald D, et al. Pandora: Principled symbolic validation of Intel SGX enclave runtimes[C]//Proc of 2024 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2024. https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00090

[19] Ghaniyoun M, Kristin B, Xiao Yuan, et al. TEESec: Pre-silicon vulnerability discovery for trusted execution environments[C]//Proc of the 50th Annual Int Symp on Computer Architecture. New York: ACM, 2023. https://doi.org/10.1145/3579371.3589070

[20] Li Mengyuan, Wilke L, Wichelmann J, et al. A systematic look at ciphertext side channels on AMD SEV-SNP[C]//Proc of 2022 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2022: 337−351

[21] Deng Sen, Li Mengyuan, Tang Yining, et al. CipherH: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations[C]//Proc of the 32nd USENIX Security Symp. Berkeley, CA: USENIX Association, 2023: 6843−6860

[22] Chen Sanchuan, Lin Zhiqiang, Zhang Yinqian. Controlled data races in enclaves: Attacks and detection[C]//Proc of the 32nd USENIX Security Symp. Berkeley, CA: USENIX Association, 2023: 4069−4086

[23] Checkoway S, Shacham H. Iago attacks: Why the system call API is a bad untrusted RPC interface[J]. ACM SIGARCH Computer Architecture News, 2013, 41(1): 253−264

[24] Khandaker M R, Cheng Yueqiang, Wang Zhi, et al. COIN attacks: On insecurity of enclave untrusted interfaces in SGX[C]//Proc of the 25h Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 971−985

[25] Bulck J V, Oswald D, Marin E, et al. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes[C]//Proc of the 2019 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2019: 1741−1758

[26] Randazzo A, Tinnirello I. KATA Containers: An emerging architecture for enabling MEC services in fast and secure way[C]//Proc of 2019 6th Int Conf on Internet of Things: Systems, Management and Security. Piscataway, NJ: IEEE, 2019: 209−214

[27] Petroni N L, Fraster T, Molina J, et al. Copilot - a coprocessor-based kernel runtime integrity monitor[C]//Proc of 13th USENIX Security Symp. Berkeley, CA: USENIX Association, 2004. https://www.usenix.org/conference/13th-usenix-security-symposium/copilot%E2%80%94-coprocessor-based-kernel-runtime-integrity

[28] Moon H, Lee H, Lee J, et al. Vigilare: Toward snoop-based kernel integrity monitor[C]//Proc of the 2012 ACM Conf on Computer and Communications Security. New York: ACM, 2012: 28–37

[29] Petroni N L, Fraser T, Walters A, et al. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data[C]//Proc of the 15th USENIX Security Symp. Berkeley, CA: USENIX Association, 2006: 289–304

[30] Loscocco P A, Wilson P W, Pendergrass J A, et al. Linux kernel integrity measurement using contextual inspection[C]//Proc of the 2007 ACM Workshop on Scalable Trusted Computing. New York: ACM, 2007: 21–29

[31] Petroni N L, Hicks M. Automated detection of persistent kernel control-flow attacks[C]//Proc of the 14th ACM Conf on Computer and Communications Security. New York: ACM, 2007: 103–115

[32] Dautenhahn N, Criswell J, Adve V. KCoFI: Complete control-flow integrity for commodity operating system kernels[C]//Proc of 2014 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2014: 292−307

[33] Baliga A, Ganapathy V, Iftode L. Automatic inference and enforcement of kernel data structure invariants[C]//Proc of the 2008

Annual Computer Security Applications Conf. Piscataway, NJ: IEEE, 2008: 77−86

[34] Carbone M, Cui Weidong, Lu Long, et al. Mapping kernel objects to enable systematic integrity checking[C]//Proc of the 16th ACM Conf on Computer and Communications Security. New York: ACM, 2009: 555–565

[35] Lin Zhiqiang, Rhee J, Zhang Xuxian, et al. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures[C]//Proc of the 18th Annual Network & Distributed System Security Symp. Reston, VA: The Internet Society, 2011. https://www.ndss-symposium.org/ndss2011/siggraph-brute-force-scanning-of-kernel-data-structure-instances-using-graph-based-signatures/

[36] Song Chengyu, Lee B, Lu Kangjie, et al. Enforcing kernel security invariants with data flow integrity[C]//Proc of the 23rd Annual Network & Distributed System Security Symp Proceedings. Reston, VA: The Internet Society, 2016. http://dx.doi.org/10.14722/ndss.2016.23218

[37] Liu Ziwen, Feng Dengguo. TPM-based dynamic integrity measurement architecture[J]. Journal of Electronics & Information Technology. 2010, 32(4): 875−879 (in Chinese)
(刘孜文, 冯登国. 基于可信计算的动态完整性度量架构[J]. 电子与信息学报, 2010, 32(4): 875−879)

[38] Jia Qiaowen, Ma Haoyu, Li Yan, et al. A novel integrity measurement architecture for embedded Linux systems[J]. Journal of Computer Research and Development, 2022, 59(10): 2362−2375 (in Chinese)
(贾巧雯, 马昊玉, 厉严, 等. 一种嵌入式 Linux 系统上的新型完整性度量架构[J]. 计算机研究与发展, 2022, 59(10): 2362−2375)

[39] Qin Yu, Liu Jingbin, Zhao Shijun, et al. RIPTE: Runtime integrity protection based on trusted execution for IoT device[J]. Security and Communication Networks, 2020, 2020(1): Article ID 8957641

[40] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity[C]// Proc of the 12th ACM Conf on Computer and Communications Security. New York: ACM, 2009: 340−353

[41] Castro M, Costa M, Harris T. Securing software by enforcing data flow integrity[C]//Proc of the 7th Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2006: 147−160

[42] Abera T, Asokan N, Davi L, et al. C-FLAT: Control-flow attestation for embedded systems software[C]//Proc of the 2016 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2016: 743−754

[43] Toffalini F, Losiouk E, Biondo A, et al. ScaRR: Scalable runtime remote attestation for complex Systems[C]//Proc of the 22nd Int Symp on Research in Attacks, Intrusions and Defenses. Berkeley, CA: USENIX Association, 2019: 121−134

[44] Yadav N, Ganapathy V. Whole-program control-flow path attestation[C]//Proc of the 2023 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2023: 2680−2694

[45] Abera T, Bahmani R, Brasser F, et al. DIAT: Data integrity attestation for resilient collaboration of autonomous systems[C]//Proc of Annual Network & Distributed System Security Symp Proc. Reston, VA: The Internet Society, 2019. https://dx.doi.org/10.14722/ndss.2019.23420

[46] Sun Zhichuang, Feng Bo, Lu Long, et al. OAT: Attesting operation integrity of embedded devices[C]//Proc of 2020 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2020: 1433−1449

[47] Wang Jinwen, Wang Yujie, Li Ao, et al. ARI: Attestation of real-time mission execution integrity[C]//Proc of 32nd USENIX Security Symp. Berkeley, CA: USENIX Association, 2023: 2761−2778

[48] Ji Dongxu, Zhang Qianying, Zhao Shijun, et al. Guan. MicroTEE: Designing TEE OS based on the microkernel architecture[C]//Proc of 2019 18th IEEE Int Conf on Trust, Security and Privacy in Computing and Communications/13th IEEE Int Conf on Big Data Science and Engineering. Piscataway, NJ: IEEE, 2019: 26−33

[49] Toffalini F, Graziano M, Conti M, et al. SnakeGX: A sneaky attack against SGX enclaves[C]//Proc of Int Conf on Applied Cryptography and Network Security. Berlin: Springer, 2021: 333−362

[50] Ding Ren, Qian Chenxiong, Song Chengyu, et al. Efficient protection of path-sensitive control security[C]//Proc of 26th USENIX Security Symp. Berkeley, CA: USENIX Association, 2017: 131−148

[51] Burmester F. Retrofitting remote control-flow attestation for ARM TrustZone[D/OL]. Lubeck: University zu Lubeck, 2022[2024-06-03]. https://www.its.uni-luebeck.de/en/teaching/theses

[52] Chen Guoxing, Zhang Yinqian, Lai T H. OPERA: Open remote attestation for Intel's secure enclaves[C]//Proc of the 2019 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2019: 2317–2331

[53] Chen Guoxing, Zhang Yinqian. MAGE: Mutual attestation for a group of enclaves without trusted third parties[C]//Proc of 31st USENIX Security Symp. Berkeley, CA: USENIX Association, 2022: 4095−4110

[54] Zhao Shixuan, Li Mengyuan, Zhang Yinqian, et al. VSGX: Virtualizing SGX enclaves on AMD SEV[C]//Proc of 2022 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2022: 321−336

[55] Gu Jinyu, Li Hao, Xia Yubin, et al. Unified enclave abstraction and secure enclave migration on heterogeneous security architectures[J]. Journal of Computer Science and Technology, 2022, 37(2): 468−486

[56] Jia Yuekai, Liu Shuang, Wang Wenhao, et al. HyperEnclave: An open and cross-platform trusted execution environment[C]//Proc of 2022 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2022: 437−454

[57] Pecholt J, Wessel S. CoCoTPM: Trusted platform modules for virtual machines in confidential computing environments[C]//Proc of the 38th Annual Computer Security Applications Conf. New York: ACM, 2022: 989–998

[58] Vikram N, Claudio C, Angelo R, et al. Remote attestation of confidential VMs using ephemeral vTPMs[C]//Proc of the 39th Annual Computer Security Applications Conf. New York: ACM, 2023: 732–743

[59] Liu Shen, Tan Gang, Trent J. PtrSplit: Supporting general pointers in automatic program partitioning[C]//Proc of the 24th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 2359−2371

[60] Liu Shen, Zeng Dongrui, Huang Yongzhe. Program-mandering: Quantitative privilege separation[C]//Proc of the 2019 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2019: 1023−1040



Li Wei, born in 1996. PhD candidate. His main research interests include trusted execution environment and confidential computing.

李　为, 1996 年生. 博士研究生. 主要研究方向为可信执行环境、机密计算.



Feng Wei, born in 1986. PhD, associate professor. His main research interests include system security, confidential computing, and trusted computing.

冯　伟, 1986 年生. 博士，副研究员. 主要研究方向为系统安全、机密计算、可信计算.



Qin Yu, born in 1979. PhD, senior engineer. His main research interests include confidential computing, trusted computing, and system security.

秦　宇, 1979 年生. 博士，正高级工程师. 主要研究方向为机密计算、可信计算、系统安全.



Feng Dengguo, born in 1965, PhD supervisor. Academician of CAS, Senior member of CCF. His main research interests include confidential computing, cryptography, and trusted computing.

冯登国, 1965 年生. 博士生导师. 中国科学院院士，CCF 高级会员. 主要研究方向为机密计算、密码学、可信计算.