DOI:10.7544/issn1000-1239.20220525
59(10):2362-2375,2022

一种嵌入式 Linux 系统上的新型完整性度量架构

贾巧雯[1,2] 马昊玉[3] 厉 严[3] 王哲宇[3] 石文昌[4]
[1](中国科学院软件研究所 北京 100190)
[2](中国科学院大学 北京 100049)
[3](西安电子科技大学网络与信息安全学院 西安 710126)
[4](中国人民大学信息学院 北京 100872)
(jiaqw@ios.ac.cn)

A Novel Integrity Measurement Architecture for Embedded Linux Systems

Jia Qiaowen[1,2], Ma Haoyu[3], Li Yan[3], Wang Zheyu[3], and Shi Wenchang[4]
[1](*Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)
[3](*School of Cyber Engineering*, *Xidian University*, *Xi'an* 710126)
[4](*School of Information*, *Renmin University of China*, *Beijing* 100872)

Abstract Integrity measurement architecture (IMA) is an important component of trusted computing. However, existing IMA schemes possess a number of practical limitations when applied in embedded systems. In this paper, we propose dynamic integrity measurement architecture at kernel-level (DIMAK), an effective and efficient runtime integrity measurement architecture for embedded Linux systems. DIMAK supports just-in-time integrity measurement for code texts and static data in both kernel and user space, as well as dynamic linking information maintained by position independent executables (PIE). Exploiting the process, memory and page management mechanism of Linux kernel, DIMAK is capable of measuring the to-be-measured contents at physical-page-level, hence avoids potential time-of-check to time-of-use (TOCTTOU) vulnerability that has been discovered in existing techniques. On top of that, by proposing a predictive integrity baseline generation technique for the relocation and dynamic linking information of ELF files, the proposed architecture achieves better completeness than the state-of-the-art schemes in case of responding to threats like hooking-based control flow hijacking and dynamically loaded malware. Also, with a novel trusted software hot-fix protocol, the proposed architecture becomes the first IMA scheme capable of correctly distinguishing on-the-fly software patching behaviors from malicious code loading. Given different types of contents to be measured, DIMAK generates the corresponding integrity baselines at a variety of timings, e.g., during off-line phase, system booting, process loading or dynamic code loading, thus ensures correctness of the architecture's integrity measurement for all possible scenarios. Experiments on real commercial embedded devices have also shown that performance overhead caused by DIMAK is sufficiently acceptable for embedded devices.

收稿日期:2022-06-11;**修回日期**:2022-08-10
基金项目:国家自然科学基金项目(61972215,61972073,62172238);国家重点研发计划项目(2018YFA0704703)
This work was supported by the National Natural Science Foundation of China (61972215, 61972073, 62172238) and the National Key Research and Development Program of China(2018YFA0704703).
通信作者:马昊玉(hyma@xidian.edu.cn)

Key words trusted computing; integrity measurement architecture; trusted platform module; embedded system; Linux

摘 要 完整性度量框架是可信计算平台的重要组成部分之一.但过往研究工作所提出的完整性度量框架设计在实际应用于嵌入式设备场景时,往往体现出不同程度的局限性.提出了内核级动态完整性度量架构(dynamic integrity measurement architecture at kernel-level, DIMAK),一种针对嵌入式 Linux 操作系统的实用化完整性度量架构,为基于 Linux 的嵌入式设备提供有效且高性能的运行时完整性验证能力.该架构支持对映射至系统内核空间及用户进程的可执行文本、静态数据以及动态链接信息等关键内容实施即时(just-in-time)完整性校验.利用 Linux 内核的进程、内存和页面管理机制,DIMAK 实现了对被度量内容所驻留物理页面的运行时校验,避免了基于文件的静态度量方法可能存在的检查与使用时差(time-of-check to time-of-use, TOCTTOU)漏洞.通过首次引入对位置无关代码的重定位/动态链接信息的完整性基线预测方法,DIMAK 在面对包括基于 hooking 的控制流劫持、恶意代码运行时载入等攻击威胁时具有较之现存同类技术更强的完备性.另外,通过引入对软件热补丁功能的可信验证支持,DIMAK 在系统完整性度量问题中将该应用场景与恶意攻击行为正确地加以区分.根据各种被度量实体的不同类型,DIMAK 在离线阶段、系统启动时、进程加载时和代码动态加载时等时机分别生成其对应的完整性基线,确保其完整性验证行为的正确性.真机测试显示,所述的 DIMAK 架构产生的性能开销完全可以满足嵌入式设备场景下的实际应用要求.

关键词 可信计算;完整性度量架构;可信平台模块;嵌入式系统;Linux

中图法分类号 TP309.1

长期以来,基于漏洞利用和恶意代码的攻击一直是计算机软件与操作系统安全所面临的主要威胁之一.在各类计算机系统当中,嵌入式操作系统(如车载系统、部分互联网关键节点和一些能源/制造业关键设备等),以及在该类平台上运行的应用软件,是漏洞利用和恶意代码攻击所关注的高价值目标.因此,建立针对嵌入式操作系统的可信计算平台,对于推进面向未来的互联网安全生态有着重要的意义.

1) 完整性度量架构及其研究现状

在与漏洞利用及其后续可能发起的恶意代码植入攻击的对抗中,完整性度量架构(integrity measurement architecture, IMA)是为计算平台可信性提供保护的重要技术之一.2004 年,文献[1]采用了 IMA 的概念,并提出了一种以可信计算组织(trusted computing group, TCG)规范① 为基础的 IMA 方案.在此方案基础上,后续研究通过引入权限分级的访问控制策略或采用经过哈希运算和加密处理的压缩度量值等方式[2-3],进一步降低了实施 IMA 对系统负载的压力,形成了多种较为完备的静态完整性度量方法.然而,基于文件的完整性度量机制仅能够在文件加载至内存时验证其当前完整性状态,这使得该类 IMA 系统无法对针对内存的运行时完整性损害予以响应,包括:

① 针对驻留内存的代码文本的篡改行为.如基于 hooking 的控制流篡改、自修改代码(self-modifying code)等.

② 不依赖静态文件、直接向内存注入恶意代码的攻击行为.例如,利用缓冲区溢出漏洞、以数据形态注入恶意代码、再借助代码重用攻击(code-reuse attack)修改页面权限使所注入代码可被执行的恶意代码动态载入攻击.

因此,动态完整性度量架构作为一种改进策略被提出[4-10].现有该类方案往往利用系统内核机制(如 Linux 系统的内核全镜像机制[6])或特定硬件(如辅助处理器[7])的支持,对驻留在内存中的系统关键完整性状态进行运行时验证.除了系统内核的代码文本外,一些方案还支持对系统中各进程所维护的关键上下文状态和环境参数的完整性予以监控[8-10],或通过与用户态进程的交互实现对其运行时完整性的验证[6,8,10].目前,IMA 技术已经被扩展应用至基于 hypervisor 的虚拟化环境[11-13]、软件虚拟机[14-15]以及软件控制流完整性验证(control flow

① TCG Specification Architecture Overview Revision 1.4, https://trustedcomputinggroup.org/wp-content/uploads/TCG_1_4_Architecture_Overview.pdf

integrity, CFI)[16]等多个有关系统完整性验证的安全应用场景中.

近两年来,一些针对嵌入式设备及系统的动态完整性度量架构方案被陆续提出[17-21].Wehbe 等人[17]利用独立硬件芯片对嵌入式系统各进程内的可执行文本实施基于页面的运行时度量和验证.Qin 等人[18]和 Ling 等人[19]各自提出了一种基于可信执行环境(trusted execution environment, TEE)的物联网设备运行时完整性保护系统.其中,文献[18]通过代码插桩拦截用户进程发起的函数调用及其返回等控制流行为,并在提供代码完整性验证的同时支持 CFI 功能;文献[19]则在 TrustZone 机制的支持下周期性地对用户代码实施基于页面的完整性度量.Duan 等人[20]提出了一种基于 TEE 虚拟化架构的系统完整性度量方案,通过在 TEE-visor 层部署度量组件,实现同时对多个虚拟 TEE 环境以及正常执行环境内的系统代码和系统调用的可信完整性校验.Cheng 等人[21]提出了一种针对航空片上系统(system-on-a-chip, SoC)的可信控制架构,通过专用的监控硬件架构为系统提供动态完整性度量保护.

然而,现有动态/静态 IMA 方案在应用于计算资源有限、同时又对运行性能较为敏感的嵌入式设备时,往往存在 3 方面的局限性:

① 针对位置无关代码所依赖的重定位和动态链接信息,现有 IMA 方案缺乏有效的校验策略.以 Linux 为例,其 ELF 文件的重定位由过程链接表(procedure linkage table, PLT)和全局偏移表(global offset table, GOT)配合完成.其中,GOT 由动态链接器在文件加载时实时填充,全过程均在用户态完成,这使得针对 GOT 的完整性度量问题成为 IMA 技术的一个两难困境:若严格服从权限隔离原则并将 IMA 部署于内核空间或可信平台模块(trusted platform module, TPM)中,则将因缺乏适当的态势感知手段而难以及时地捕捉到用户进程中文件重定位和动态链接行为的发生时机,因而无法及时算得 GOT 载入完成后的完整性基线、难以实现对其进行动态完整性验证的目标;反之,若在用户进程空间内部署辅助接口、以插桩动态链接器等方式直接在用户空间内拦截动态链接行为并为 GOT 生成完整性基线,又将不可避免地将负责实施该功能的 IMA 组件直接暴露在攻击者视野中,使得 IMA 架构的整体可信任性产生瑕疵.

② 现有 IMA 方案无法与包括代码热修复(hot-fix)在内的一些特殊的运行时代码加载/改写场景实现有效的兼容.具体来说,由于主流代码热修复方法在实现原理上与基于 hooking 的恶意代码动态载入基本一致,仅凭通用的常规技术无法将两者加以区分.这意味着应用 IMA 功能的系统可能无法同时启用热修复机制,这将有可能严重地限制系统对 0-day 漏洞的应急响应能力.

③ 对于嵌入式系统而言,现有 IMA 造成的性能开销往往超出了工程实践可以接受的范畴.例如,文献[18]引入了细粒度动态插桩和 CFI 等保护机制,使得单个应用程序由此承受了十几甚至上百倍的运行时性能开销.文献[17]和文献[19]所采用的基于页面的完整性度量机制则占用了大量运行内存空间以维护针对具体内存页面的完整性基线.以 glibc 为例,该基本系统库的 2.3.2 版本在加载后的代码段大小为 1 208 KB,对应 302 个页面,在考虑使用 SHA256 等安全哈希算法的情况下,仅该系统库一例,即需要占用 9 KB 内存以维护其代码完整性基线,这还未将运行时如何实现对海量基线的索引问题纳入考虑范畴.由于嵌入式系统受到物理约束的限制,所能够利用的硬件资源,包括 CPU 主频、CPU 核心数量、运行内存大小等往往极为有限,并需要在满足最小软硬件需求的情况下保证实时性.故在性能开销方面,多数嵌入式系统很可能并不具备现有的大多数 IMA 方案所需的扩展空间.

2) 创新点与贡献

针对现有 IMA 方案存在的完备性不足、无法兼容软件热修复、性能开销较大等问题,本文提出了一种应用于嵌入式 Linux 系统的高性能内核级动态完整性度量架构(dymamic integrity measurement architecture at kernel-level, DIMAK).利用 Linux 内核自身定义的进程空间和页面管理机制,DIMAK 将自身组件完全部署在系统内核空间,并通过拦截与代码加载/改写行为、进程创建行为以及系统 shell 命令执行有关的关键系统调用等方式,实现以虚拟地址映射分段为基本度量单元、面向运行内存而非文件存储的运行时完整性验证.根据进程参数、代码文本、重定位和动态链接数据等各种被度量实体所具有的不同初始化时机,DIMAK 确立了一套动态/静态基线相结合、基于最小信任根的逐级信任扩展策略.在上述底层设计基础上,DIMAK 方案的一个核心创新点在于提出了一种内核级的“进取型动态链接预测技术”,使其能够在不暴露于用户空间且无需动态链接器主动配合的情况下正确地计算出用户态位置无关代码所对应重定位和动态链接信息

的完整性基线.这一机制允许 DIMAK 在与用户空间保持完全权限隔离的情况下,将针对用户进程的完整性保护扩展到与可执行行为完整性相关的所有要素上,从而实现了现有方案无法达成的高度完备性.此外,DIMAK 的另一个关键创新点是将软件热修复行为纳入系统内核的标准内存管理,从而在完整性验证过程中有效地将该场景与恶意代码篡改行为加以区分.通过提供专用系统调用,DIMAK 以其内核态组件接管了软件热修复过程中所涉及的补丁代码加载、被修复程序改写等工作,确保所涉及的完整性基线能够被及时地生成和更新,实现了将软件热修复纳入系统运行时信任依赖的范畴中这一安全目标.

简而言之,本文所述 DIMAK 方案的主要创新性贡献可总结为:

① 在技术层面上.DIMAK 通过其进取型动态链接预测技术,解决了过往 IMA 方案因缺乏针对重定位和动态链接信息的完整性基线获取手段而存在的完备性缺陷,即无法完全部署在内核态.

② 在工程层面上.DIMAK 在不损害系统安全性的前提下,实现了 IMA 方案对代码热修复场景的支持,在实用安全性能方面实现了零的突破.

模拟环境和真机测试显示,DIMAK 主要工作于系统日常运行过程中占比极低的代码加载阶段,故其运行时总体性能开销和内存占用均显著低于现有同类方案,并能良好地适应嵌入式设备可能具有的硬件配置情况.

1 威胁模型与安全目标

现有各类 IMA 方案在适配嵌入式系统时所存在的一些实用性方面的不足,究其原因,在于当前针对 IMA 等可信计算概念的研究往往倾向于设置越来越高的安全目标以使得所设计的方案能够"兼容并包"地对抗多种类型的威胁对手.为此,相关方案不断引入开销较大的度量机制,所采用的度量策略则向着高度实时、细粒度化的方向发展[9-10,18].然而,不同于通用系统,嵌入式系统的应用场景往往比较单一,其体量也相对较小,故嵌入式系统对 IMA 方案的需求也有所不同.

1) 应用场景单一使得嵌入式系统所受的攻击面较小,且无需考虑通用系统可能面临的诸多不确定性,如安装/卸载任意应用带来的不可预测的系统完整性基线变更等.

2) 体量较小,意味着嵌入式系统要求 IMA 方案必须在计算资源极为有限的情况下仍具备可用性;否则,无论其预期安全目标如何强大,还是难免因无法部署而成为无本之木.

特别需要指出的是,现有 IMA 方案中额外引入的细粒度执行状态度量等机制(如 CFI 等)主要以不确定性系统环境中的任意执行行为和以代码重用为代表的高级漏洞利用编码技术为防护对象.然而,代码重用本质上是一种漏洞利用手段而非后果,其对目标系统的潜在影响也并未脱离传统漏洞利用方法的威胁模型范畴.例如,当通过漏洞利用实施权限提升攻击时,无论是否以代码重用为载荷编码手段,攻击者最终仍需要修改系统内核参数(如进程权限)并启动一个高权限 shell.因此,即使不具有细粒度动态执行监控能力,IMA 仍可通过对内核关键数据的完整性验证来检测并响应上述篡改.这就使得现有 IMA 方案在应用于嵌入式系统时显得过犹不及:一方面,它们所采用的安全机制在实际有效性上很可能并无显著提升;另一方面,它们却肯定会因为这些机制而给目标系统造成不可承受的性能负担.因此,本文所提出的 DIMAK 所考虑的威胁模型主要针对可能对系统完整性造成破坏的攻击类型,包括(但不限于):

1) 恶意代码/进程的动态载入和内存驻留;

2) 权限提升攻击;

3) 基于 hooking 等技术的函数调用劫持.

考虑到嵌入式 Linux 系统在实际应用中的安全需求,针对上述威胁模型,DIMAK 主要期望达成的安全目标包括:

1) 为嵌入式 Linux 系统内核及用户空间内的所有可执行文本(包括 shell 脚本)提供运行时完整性校验;

2) 为用户态位置无关代码加载后的动态链接和重定位信息提供运行时完整性校验;

3) 为用户进程可能应用的代码热修复功能提供完整性校验支持;

4) 除实时校验外,支持对系统内全部待度量内容实施人工或计划性的全量扫描;

5) 对系统运行中发生的进程/子进程创建等敏感行为同样予以实时合法性验证.

上述安全目标的逻辑在于对目标系统同时应用两套校验策略,既支持对系统日常运行中的潜在安全威胁予以实时响应,又能够在计划性全量扫描中实施"查漏补缺".为了在性能开销方面更有可能适

应嵌入式系统和设备的实际计算资源状况,上述安全目标的实现不应依赖于用户应用程序的侵入式改写,例如文献[18]中所采用的动态插桩,或依赖于对执行行为的即时(just-in-time)跟踪等性能开销较大的操作.

考虑到嵌入式系统所具有的专用性和以应用为中心等特点,DIMAK 假设目标系统运行一组相对固定的内核 KO 模块及用户应用程序,并因此具有稳定的代码完整性基线.在安全性方面,DIMAK 的主要目标在于保护嵌入式系统运行过程中的完整性状态,故本文假设部署 DIMAK 的操作系统已经具备某种可信启动技术,使得 DIMAK 总是能够在系统处于同一确定的可信状态时予以接管.另外,DIMAK 还假设目标系统在用户和内核空间均已应用了地址空间布局随机化(address space layout randomization, ASLR)、数据执行保护(data execution protection, DEP)等现代操作系统的基本安全机制.由于 ASLR 机制的存在,DIMAK 进一步假设目标系统在加载动态链接库时采用立即绑定模式;否则,用户进程在运行时将具有与程序控制流相关的非静态结构化数据(如 GOT 将变为可读写),使得决定进程完整性的部分要素无法被有效地度量.

2 架构的设计与实现

2.1 总体架构

整体上,DIMAK 由部署在 Linux 内核空间内的主校验模块、部署在 TEE 环境内的根模块以及由 TPM 维护的完整性基线组构成.通过应用这一部署策略,DIMAK 建立了自身与系统用户空间之间的完全权限隔离,同时也在根模块与可能从内核空间发起的潜在威胁(如驱动程序等)之间建立了可信的入侵检测机制,使得框架整体的信任逻辑具有良好的完备性.

如图 1 所示,当 DIMAK 启动时,其主校验模块在第一时间对内核空间内的一组系统调用函数实施代码插桩(code instrumentation),由此建立一个位于内核空间的状态感知界面.此设计使得 DIMAK 能够实时探测发生在系统内核及用户空间内的完整性状态变更事件,包括进程创建/终止、代码加载/卸载、代码改写以及 shell 命令执行等.通过复用 Linux 自身的进程和内存管理机制,DIMAK 主校验模块能够直接获取用户态进程内下列内容所对应的物理页面并计算其完整性度量:

1) 用户进程空间内的所有代码段;

2) 附属于任一代码段的结构化数据段(主要为相关代码的重定位、动态链接信息等).

而对于内核空间中的 Linux 内核镜像和 KO 模块等内容,DIMAK 主校验模块可以直接使用虚拟地址予以访问和度量.利用状态感知界面,DIMAK 还能够侦测来自远程管理员的人工指示,并根据具体命令对系统实施单次或周期性的全量扫描,即一次性对系统内的所有代码和结构化数据区段予以检验.此外,自 DIMAK 启动时起,其根模块即开始对主校验模块以及状态感知界面所涉及的内核指令实施周期性扫描,确保 DIMAK 自身始终处在可信的运行状态下.最后,在与实时完整性验证或全量扫描时所测得的完整性度量实施比对时,DIMAK 的主校验模块与根模块均通过 TPM 安全地访问其完整性基线组.

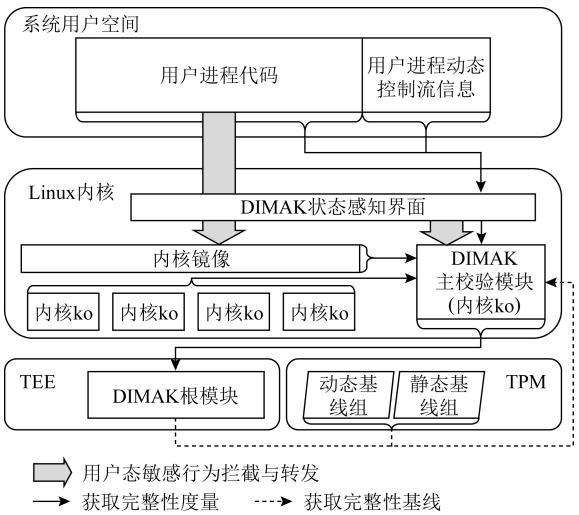

Fig. 1 Overall architecture of DIMAK

图 1 DIMAK 的总体架构

不难看出,状态感知界面是 DIMAK 方案是否能够实现其安全目标的一个关键点.但除此之外,针对不同类型的被度量实体所采用的具体完整性基线生成策略,以及面对多种特殊的系统完整性变更事件时所采用的创新性度量方法和系统设计约定,同样是 DIMAK 达成有效性和完备性的重要保障.因此,2.2 节将详细阐述 DIMAK 状态感知界面的具体设计;针对各类被度量实体的完整性基线维护策略,特别是针对用户态位置无关代码所对应的重定位和动态链接信息等关键结构化数据的可信基线生成方法,将在 2.3 节中介绍;2.4 节将详细解释 DIMAK 对于软件热修复这一特殊应用场景的完整性度量支持及其具体协议设计.

2.2 状态感知界面的实现

由于在架构部署策略上建立了不同地址空间上的权限隔离,DIMAK 需要能够在处于内核态的情况下实时地感知用户空间内发生的系统完整性关键事件,这一任务由其在启动时所建立的内核态状态感知界面来完成.从过往研究以及工业界实践中可以初步总结出该状态感知界面所需监测的用户态程序行为主要包括:

1) 进程的创建和终止行为.这是由于对用户进程的恶意攻击有可能会为后续恶意载荷的执行创建新的子进程.

2) 代码的加载和修改行为.典型代表是恶意载荷的注入以及对用户态代码的控制流篡改,它们是 IMA 所需考虑的主要威胁.

3) 请求执行 shell 脚本的行为.这是由于对用户空间进程的恶意攻击还有可能通过执行特定的 shell 命令来达成某些攻击目的,如权限提升等.

此外,作为一个特殊情形,由用户空间进程发起的内核 KO 模块加载行为同样也需要纳入状态感知界面的监测范围.

上述关键事件的执行过程均依赖于 Linux 内核所提供的系统调用支持,故如 2.1 节所述,DIMAK 通过对这些系统调用实施代码插桩的方式实现在所需监测的关键事件发生时拦截并获取控制的目的.另外,利用所设置的插桩点,DIMAK 还能够直接访问 Linux 内核用于文件、进程和内存管理的一系列关键变量,从而实现对全系统范围内被度量实体的定位和访问.图 2 示意了 DIMAK 为构建状态感知界面所采用的内核函数拦截面,以及藉此能够访问的关键内核数据结构.

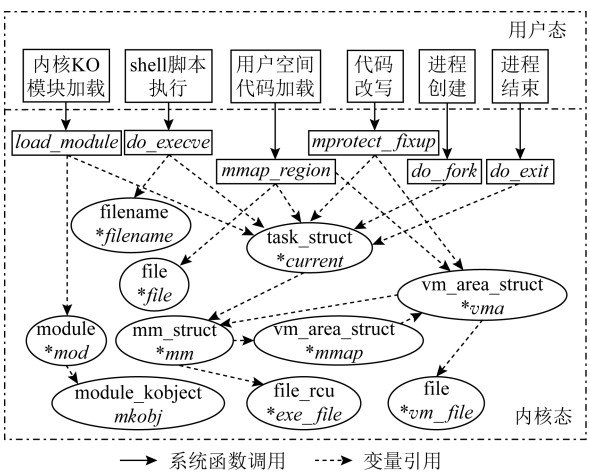

Fig. 2 Design of DIMAK's situation awareness interface

图 2 DIMAK 状态感知界面的构造

图 2 中需要说明 4 方面:

1) 利用系统调用实例 *do_fork*()与 *do_exit*(),可拦截用户进程的创建/终止事件;

2) 利用实施内存映射所需的关键系统调用实例 *mmap_region*()以及加载 KO 模块时所使用的内核函数 *load_module*(),可监控到系统范围内所有 ELF 文件的加载事件;

3) 利用系统调用实例 *mprotect_fixup*()函数,并在其目标权限参数为可执行或只读时触发响应,用户空间内的任意代码改写和控制流劫持行为均可被拦截;

4) 利用文件执行操作的内核入口函数 *do_execve*(),shell 脚本的执行请求事件可以被感知,且通过该函数的参数还可以获取到被执行脚本的文件路径以实现对被度量实体的定位.

通过上述任一被插桩函数,DIMAK 均可访问系统内核所维护的 task_struct 类型(Linux 进程管理数据结构)变量 *current* 以获知引发相应关键事件的用户进程信息.较为特殊的情况是,在执行外部 shell 命令或 shell 脚本时,程序往往会为该次执行单独创建子进程,故在感知此类行为时,DIMAK 除获取当前进程信息外,还需回溯父进程信息以确定行为来源.最后,上述状态感知界面的设计还能够为 DIMAK 所期望实现的面向运行内存的完整性验证功能提供支持.具体地,前述 Linux 进程管理数据结构 task_struct 内部维护有指向进程内存映射的对象指针 *mm*(类型为 *mm_struct*),*mm* 进一步指向一个描述该进程空间内所有虚拟内存段的 vm_area_struct 类型双链表 *vma*,表内的每个对象均与进程空间内的一个虚拟内存段相对应,且维护该内存段的虚拟地址范围和对应文件指针.利用 *vma* 链表所提供的参数,DIMAK 可以在内核态遍历位于各用户进程空间内的诸如 ELF 文件代码段、静态变量段和只读结构化数据段等被度量实体的对应虚拟内存描述符,进而结合其所给出的虚拟地址和对 Linux 页表的解析寻获被度量实体所驻留的物理页面,实现对实际运行内存状态而非抽象文件内容的完整性验证.

最后,利用上述状态感知界面,DIMAK 进一步实现了对人工/计划性全量扫描功能的控制.通过在 Linux 的标准 shell 命令行范畴外额外约定一组自定义的 shell 命令行,DIMAK 可在其位于 *do_execve*()函数内的拦截点检测到 shell 命令行执行行为时判断被捕获参数是否为自定义命令行,并在匹配成功

时触发全量扫描事件.与前述的 *vma* 链表类似,Linux 的进程管理机制还将用户空间内的活动进程在总体上组织为以 task_struct 结构体为基本元素的双链表.因此,在触发全量扫描时,DIMAK 可由指向当前用户进程的 *current* 指针出发,对 task_struct 双链表进行遍历以访问所有用户进程,并使用各进程的 *vma* 链表以遍历扫描其地址空间内的所有被度量实体.同时,由于 DIMAK 的主校验模块本身部署于系统内核空间,故可以直接通过虚拟地址遍历扫描内核镜像本身以及内核 KO 模块列表.

2.3 完整性基线的分组与生成

针对系统中不同类型的被度量实体,DIMAK 使用了静态/动态相结合的双完整性基线组模式,以确保在面对所有可能发生的完整性状态变更时总是能够做出正确的判断.

2.3.1 静态基线组

在众多类型的被度量实体中,DIMAK 应用纯静态基线予以保护包括 ELF 文件内的静态变量区段、用户进程的元数据信息及其请求执行的 shell 脚本.值得注意的是,尽管用户进程在每次启动时所获得的标识符是动态生成的,但 Linux 的 task_struct 结构体通过内嵌的 mm_struct 结构体关联进程所对应的主可执行文件(参考图 2 中通过 mm_struct 索引的 *exe_file* 变量),故 DIMAK 可利用上述关联生成针对进程信息的静态基线.

2.3.2 动态基线组

由于用户所加载的 ELF 文件采用位置无关代码的编码形态,使得该类文件在加载后需由动态链接器进行符号解析、修改其重定位表及 GOT 以维持程序控制流的正确性.因此,承载这些重定位和动态链接信息的结构化数据段只在用户进程的单个生命周期内具有稳定的完整性基线.该基线在 ELF 文件加载时动态生成,并随其卸载而失效.此外,对于全局描述符表(global descriptor table, GDT)、系统调用表(system call table)等操作系统内核所依赖的关键动态数据,DIMAK 同样只能在系统启动后为其维护动态基线.

2.3.3 静态/动态双基线组

较为特殊的是,由于内核 KO 模块所对应的 ELF 文件并不使用位置无关代码,故在内核 ASLR 启用的情况下,KO 模块的每次加载时所加载的具体代码文本都会有所改变.因此,当 KO 模块加载时,DIMAK 需要验证其可执行文件的静态完整性;但在实施计划性的全量扫描时,DIMAK 又需要验证 KO 模块所驻留运行内存的动态完整性.此外,由于 DIMAK 本身需要通过对 Linux 内核镜像实施 inline hooking 以建立状态感知界面,使得其对内核镜像的完整性验证存在同样的问题.因此,对于上述 2 种情形,DIMAK 在 KO 模块首次加载/系统启动时首先应用离线生成的静态基线实施完整性验证,随后生成并在系统运行的剩余周期内转而应用动态完整性基线.

另一个可能需要应用静态/动态双基线的被度量实体类型是位于用户空间内的 ELF 文件代码文本.这是因为 DIMAK 需要考虑兼容代码热修复机制,且现有代码热修复技术在实施过程中均需要对已加载的可执行文件进行 inline hooking 来实现对被修复函数的重定向(至其补丁版本所在的位置).因此,尽管 DIMAK 在所有用户态 ELF 文件的首次加载时均使用离线生成静态基线予以验证,但每当一次热修复操作完成后,DIMAK 就必须对受到影响的 ELF 文件代码段进行完整性基线的更新,并转而在其之后的生命周期内应用动态基线;而对于未受到热修复影响的 ELF 文件,DIMAK 则始终在对其代码段的完整性验证中应用静态基线.

2.3.4 动态基线的生成机制

表 1 列出了当前 DIMAK 所考虑的所有被度量实体类型及其各自对应的完整性基线组.其中,归属于静态/动态双基线组的 3 类被度量实体均具有可从内核空间内准确感知的基线转换时机.然而,用户态 ELF 文件所附带的动态链接和重定位信息是在其加载过程中经动态链接器处理后写入的,期间系统内核仅在内存分配和页面权限改写时有所参与.这就使得主校验模块完全位于内核空间的 DIMAK 很难捕捉到动态链接器完成上述结构化数据写入的

Table 1 Grouping Strategy of DIMAK's Integrity Baselines

表 1 DIMAK 的完整性基线分组策略

被度量实体类型	基线类型	基线生成时机
shell 脚本	静态	离线阶段
ELF 静态数据段		
用户进程信息		
ELF 动态链接/重定位信息	动态	位置无关代码加载时
敏感内核数据		系统启动时
内核镜像	静态/动态	离线/DIMAK 状态感知界面创建后
内核 KO 模块		离线/模块加载时
ELF 代码段		离线/热补丁修复后

确切时机,因而无法仅凭被动的状态感知建立可信的动态基线.针对这一问题,DIMAK 创新性地采用了一种进取型动态链接预测技术,在用户态 ELF 文件尚未完全加载时主动"模拟"动态链接器的行为,从而预测正在载入的 ELF 文件最终的重定位和动态链接状态.

图 3 示意了 DIMAK 所采用的动态链接预测技术的工作原理.当 Linux 系统加载一个用户态 ELF 文件时,其动态链接器会在符号解析时首先判断该文件是否依赖于尚未加载的其他 ELF 文件.例如,Linux 在为应用程序创建进程时总是首先加载后缀为.o 的主可执行文件,然后根据符号解析获取到的依赖关系递归地加载该程序所需要的其他 ELF 文件(如共享类库等).利用动态链接过程的这一特点,DIMAK 借助状态感知界面对 *mmap_region*() 的拦截点(参见 2.2 节),在任意用户态 ELF 文件加载时主动介入、抢先解析其文件依赖关系,并将被加载文件所依赖但尚未加载的其他 ELF 文件添加至一个动态列表.在此之后,每当 DIMAK 感知到新的用户态 ELF 文件加载时,即检查该文件的存储路径及所属进程等信息是否存在于当前处于活动状态的某个依赖关系列表内,若是则从相应列表中删除该项.当一次表项删除导致 DIMAK 所维护的某一依赖关系列表为空时,即意味着该列表对应的 ELF 文件所需的依赖文件已经全部载入内存、即将进入动态链接阶段.因此,DIMAK 得以在该时机介入并依照动态链接器的工作规则抢先为相应的 ELF 文件计算

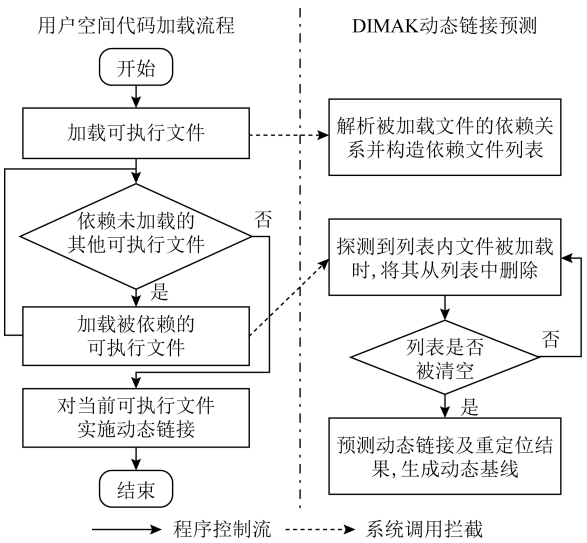

Fig. 3 An illustration of the dynamic linking prediction technique of DIMAK

图 3 DIMAK 的动态链接预测技术示意

动态链接和重定位信息.到这一步,DIMAK 已经能够在用户态 ELF 文件的加载和动态链接过程尚未完成时为其只读结构化数据段生成正确的动态基线.

上述进取型动态链接预测技术的存在,使得 DIMAK 即使处在系统内核空间中也能够及时(确切地说,是抢先)且正确地维护与用户态位置无关代码相关联的所有关键控制流完整性信息,无需修改动态链接器或对其动态插桩,也无需将自身任何组件探入用户进程空间.截止目前,该安全目标尚未在现有 IMA 方案中真正实现过,是 DIMAK 在系统完整性度量的完备性方面的一次创新性突破.

2.4 对代码热修复功能的支持

尽管代码热修复是一种工业界常见的软件工程需求,但对于 IMA 技术而言,实现对该功能的兼容所面临的最大困难在于代码热修复在技术本质上与恶意代码注入行为的界限十分模糊.事实上,若不对热修复所注入的代码文本进行格式上的事先约定,则无法仅凭两者对内核函数的调用特征将其加以区分.借鉴了 Android 平台所采用的 APK 签名验证机制,DIMAK 采用了一种类似的代码热补丁格式与签名规范约定,结合其状态感知界面所利用的 Linux 进程空间管理机制,首次实现了对代码热修复功能的可信验证支持.

图 4 示意了 DIMAK 所采用的可信热补丁格式约定.该格式以一个固定长度的头部为起始,依次容纳了一个补丁位图以及一个或多个补丁段(即允许对用户进程内的多个可执行文件同时进行热修复).其中,补丁头部包含了魔数字符、补丁区所容纳的补丁段总数、补丁位图大小等关键信息以帮助 DIMAK 对补丁包进行识别和解析.此外,补丁头部内还写入了补丁包的签名信息以及全体补丁段的总和校验和.紧接在头部之后的补丁位图给出了后续补丁段以及段内各补丁函数的索引,并给出了每个补丁函数的修复对象所在的可执行文件名,使得 DIMAK 可以判断该次热修复对系统当前基线组的具体影响状况.

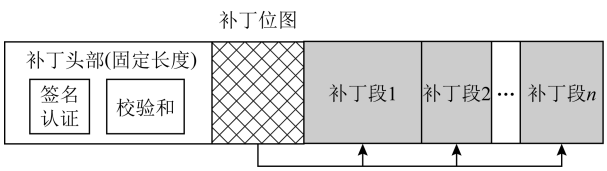

Fig. 4 The format of trusted hot patch defined by DIMAK

图 4 DIMAK 的可信热补丁格式约定

除图 4 所示的格式约定外,DIMAK 进一步规

定:在可信的热修复过程中,补丁包必须先将符合格式约定的热补丁载入内存并设为可执行权限,并等待该操作成功后方能对进程内已经加载的其他代码段进行修改.按照这一设计,DIMAK 对软件热修复功能的支持可由 4 个流程描述:

1) 当 DIMAK 在运行时检测到某一用户进程正在加载一个具有可执行权限、但未被该进程的 *vma* 链表收录的虚拟内存段时,即对该段首地址进行魔数识别以判断该次加载是否可能为一次可信的热补丁行为.

2) 若魔数识别成功,则 DIMAK 通过补丁位图确定被加载虚拟内存段内部的各补丁段界限,并对全体补丁段的总和校验和进行完整性验证.

3) 若验证通过,DIMAK 将前一步骤所使用的总和校验和作为当前补丁包的完整性基线纳入静态基线组,并为该补丁包创建 2 个新的 vm_area_struct 结构体.其中,第 1 个融合了补丁包内的所有补丁代码段并被赋予可执行权限;而第 2 个则被映射到补丁头部和位图区段,并被赋予只读权限.

4) DIMAK 根据补丁位图信息定位到进程空间内即将被修改的所有代码段,允许对这些代码段各实施一次改写,并在该次改写完成后更新相应代码段的完整性基线.

1)~4)设计使得 DIMAK 能够保证正常的代码热修复过程所涉及的所有用户态代码(包括补丁本身以及被修复的运行代码段)在该过程的任意时刻总是具有可信的完整性基线,从而保证系统整体在热修复发生时全程处于可度量状态.值得一提的是,该设计的 DIMAK 除要求用户态应用程序依照其格式约定编码补丁包之外,对代码热修复功能的支持无需额外的用户/内核代码组件的协助.

3 有效性与性能评估

3.1 信任扩展逻辑

在整体信任扩展逻辑的构建方面,DIMAK 所采用的分层权限隔离的架构设计(如 2.1 节所述),使得除 DIMAK 根模块外,系统各地址空间内的程序模块——包括 DIMAK 自身的主校验模块及用户空间状态感知界面——均由权限更高的 DIMAK 模块予以校验.

首先,对于处在用户地址空间内的各应用程序而言,DIMAK 的所有组件均位于系统内核空间或具有更高权限的安全执行环境内,且不依赖于任何用户态辅助组件.因此,当用户空间内的某一程序进程中出现恶意执行行为时,其所拥有的权限不足以对 DIMAK 本身实施篡改或欺骗.因此,只要部署于内核中的各组件能够被信任,则经过 DIMAK 验证的用户空间可被认为得到了有效的信任扩展.

其次,由于 DIMAK 通过部署在 TEE 内的根模块为其主校验模块提供完整性验证,故只要该根模块能够被信任,则可以认为受到其周期性验证的 DIMAK 主校验模块也得到了有效的信任扩展,其执行行为因而可以被信任.

最后,DIMAK 的主校验模块与系统内核镜像、KO 模块等组件同处于内核空间内,故当系统遭到源自驱动程序等同样位于内核空间的第三方组件的恶意行为时,攻击方确实具有破坏 DIMAK 各内核态组件完整性的可能性.然而,由于 DIMAK 的主校验模块得到了根模块的背书,故该模块较之内核空间中的其他代码组件具有更高的信任等级.此外,由于 DIMAK 在主校验模块启动后由该模块负责验证系统内核镜像及其加载的其他内核 KO 模块的动态完整性,故在主校验模块能够被信任的情况下,即可以认为:经该模块验证的其他内核空间组件均得到了有效的信任扩展.

综上可以认为,当假设部署于 TEE 内的根模块为可信时,DIMAK 能够基于此信任根、凭借操作系统的地址空间层次实现逐级的信任扩展,使得其对系统内所有组件的完整性验证均可得到更高权限的可信模块之背书.而无论是位于用户空间或是内核空间的潜在攻击者,均无法完全规避、欺骗或破坏包括根模块在内的所有 DIMAK 组件,故无法绕过该架构的信任逻辑.

3.2 正确性

在 3.1 节所述的信任逻辑基础上,DIMAK 所采用的状态感知界面这一设计(如 2.2 节所述),使其能够从基线生成时机和实时完整性验证的触发机制 2 个方面保证自身行为的正确性.

由于采用了静态/动态双基线组的设计,DIMAK 得以依赖其动态基线组对系统中部分正当的运行时代码/数据改写需求提供有效的可信计算支持(如 2.3 节、2.4 节所述).这一机制是决定该方案正确性的主要因素之一.而状态感知界面使得 DIMAK 能够在适当的时机介入此类运行时的改写行为,由此保证与之相关的动态基线生成过程始终满足两项原则:

1) 任意一项动态完整性基线的生成均得到了

某项与之相关的静态基线的完整性背书.例如,在内核 KO 模块的加载中,DIMAK 在加载前后分别介入、首先引导一次基于静态基线的完整性验证以确认待加载模块的对应文件是否具有可信的静态度量,随后才会在 KO 加载完成后生成并切换至动态基线.由此,DIMAK 使得内核 KO 模块所对应 ELF 文件的静态基线成为其运行时动态基线的完整性背书.在其他动态基线场景中,

① 用户态 ELF 文件内的动态链接/重定位信息的生成发生在其代码文本的加载过程中,故代码文本的静态基线天然地成为该类信息动态基线的完整性背书;

② 类似地,用户代码的热修复行为也得到了补丁包以及被修复代码段二者所对应静态基线的双重完整性背书.

2) 状态感知界面的存在,保证了任一动态完整性基线的生成均发生在对应被度量实体所发生的完整性状态变更实际生效之前.例如,在为用户态 ELF 文件的动态链接/重定位信息生成动态基线时,DIMAK 所选择的拦截点为 *mmap_region*()这一代码加载行为的必经系统调用,而在该系统调用执行时,所加载代码的重定位和动态链接过程甚至尚未开始.类似地,在 KO 模块加载过程和用户代码热修复过程中,相关动态基线同样在对应代码文本被标记为可执行之前即生成完毕.

此外,在面对各种可能造成系统完整性状态变更的事件时,DIMAK 所选择的各内核拦截点同样保证了对应的完整性校验发生在事件生效之前.以 shell 脚本执行行为为例,如图 2 所示,当用户空间内进程发起一次 shell 脚本执行时,DIMAK 的状态感知界面通过 *do_execve*()拦截该请求,并通过输入参数获取到脚本名称及路径,进而引导其主校验模块对脚本完整性实施验证.而被请求的脚本在这一时刻甚至尚未被解析,故其执行行为在完整性校验发生时显然尚未生效.

上述特点,决定了 DIMAK 在其基线维护及实施完整性验证的过程中不会遗留下检查与使用时差(TOCTTOU)漏洞:DIMAK 在任意一次完整性验证中总是能够获取到被度量实体所对应的正确基线值;同时,在 DIMAK 对任一被度量实体的完整性验证完成之后到该对象被变更为可执行、只读等生效权限状态之间,攻击者难以获得对该对象实施不受控改写的时间窗口.综上可以认为,DIMAK 架构的设计具有完整的正确性逻辑.

3.3 完备性

在 IMA 技术的有效性评估方面,存在的一个主要问题是:不同的 IMA 方案往往因工作原理上的显著差异,使得其在有效性方面难以形成可比性.然而本文认为:仍然可以从 IMA 方案的完备性设计入手,对现有的不同 IMA 方案进行有效性层面上的比较.针对 IMA 技术的完备性指标主要包括:

1) 验证对象的范围,即 IMA 方案能够将软件系统环境内的哪些资源和对象纳入其“保护伞”之下;

2) 基线类型,即 IMA 方案采用何种完整性基线以支持其对系统的验证行为;

3) 干预时机,即 IMA 方案能够在软件系统启动及运行过程中的哪些时机介入并实施其保护行为.

4) 对正常软件功能的支持/限制,应被视为 IMA 方案完备性的一个指标.显然,如果软件系统的部分功能不能在一个 IMA 方案启动的状态下被正常地实现,则该 IMA 方案应该被认为在完备性上存在缺陷.具体来说,是否能够支持对软件的热修复功能,是本文主要考虑的一个属于此类型的完备性指标.

5) 完整性度量过程是否需要对被度量实体实施侵入式改写(如动态代码插桩等),应被视为 IMA 方案完备性的另一个指标.这是因为基于侵入式改写的完整性验证措施自身在本质上属于对被改写软件完整性的侵犯.因此,一些软件很可能出于性能、正确性或隐私保护等因素的考虑而无法在真实应用场景中允许来自外部的该类行为.

上述 1)~3)为现有关于 IMA 技术的研究工作中广泛采用的完备性指标.在此基础上,本文则进一步认为 IMA 技术的完备性还应额外体现在指标 4)和 5)两个方面.表 2 从完备性指标的 5 个方面出发,对本文所提出 DIMAK 方案与现有学术研究及工业实践所提出或应用的 IMA 方案进行了完备性层面的对比.可以看到,相比早期一些仅关注内核完整性的 IMA 方案[5-7],DIMAK 具有更为完备的验证范围,且在干预时机方面有效地覆盖了系统运行过程中的主要关键节点.而与近年来新出现的一些以用户态进程完整性为主要目标的 IMA 方案[8,18-19]相比,DIMAK 在功能实现上并不需要通过对用户进程实施主动插桩或其他改写,且在校验时机方面兼具最优的实时性.最后,除 DIMAK 外,现有 IMA 方案均未考虑到热修复等实用软件场景与恶意攻击的区分问题.综上,可以认为较之现有同类技术,DIMAK 具有较好的完备性.

Table 2 Completeness Feature Comparison of DIMAK and the Existing IMA Schemes

表 2 DIMAK 与现有 IMA 方案的完备性指标对比

IMA 方案	验证范围	干预时机	基线类型	是否支持软件热修复	是否依赖侵入式改写
Linux IMA	内核镜像、内核模块、用户态代码、配置文件等	系统启动、文件首次创建、文件更新	静态	否	否
文献[3]	ELF 文件	文件首次执行时	静态	否	是
文献[5]	内核镜像、内核模块	内核任务的全过程	动态		否
文献[6]	内核镜像、内核模块	系统启动、KO 模块加载、敏感内核事件、响应主动请求等	静态		否
文献[7]	内核镜像、内核模块	周期性介入	静态/动态		否
文献[8]	用户态进程映像	进程创建/终止、运行时代码改写、响应主动请求等	动态	否	是
文献[10]	用户态代码、线程信息、运行栈、环境变量、特殊寄存器数值等	进程创建、缺页中断及其他中断请求、环境切换等	静态/动态	否	否
文献[17]	内核镜像、内核模块、用户态代码	进程启动时	静态	否	否
文献[18]	用户程序代码、运行时控制流信息	可执行文件加载、用户程序关键控制流节点(如函数调用、返回等)	静态/动态	否	是
文献[19]	Trustzone 架构下的非安全世界进程代码	周期性介入	静态	否	否
文献[20]	内核代码、内核关键数据	系统启动、内核模块加载、中断请求发生时	静态/动态	否	否
DIMAK	内核镜像、内核模块、敏感内核数据、用户态代码及动态链接/重定位信息、shell 脚本、静态变量等	系统启动、KO 模块加载、进程创建/终止、可执行文件加载、运行时代码改写、脚本执行、响应主动请求等	静态/动态	是	否

此外,我们测试了 DIMAK 面对不同类型的漏洞利用攻击时的实际防御效果,包括:

1) 利用用户进程漏洞发起的返回导向编程,攻击目标包括篡改进程代码段、GOT 以及执行自定义的 shell 脚本;

2) 利用驱动程序漏洞发起的内核注入,攻击目标包括篡改内核参数、破坏 DIMAK 主校验模块和状态感知界面等.

测试显示,在面对上述攻击时,DIMAK 均能有效地发现系统完整性状态遭到破坏的事实,并对遭到破坏的被度量实体予以准确定位.该测试进一步证明了 DIMAK 方案在面对潜在恶意攻击时具有良好的完备性.

3.4 性能开销

需要指出的是,基于 3.3 节所述的完备性差异的原因,不同的 IMA 方案很难在性能开销方面形成具有可比性的对照评估.因此,本文主要对受 DIMAK 保护的 Linux 系统与同版本的标准 Linux 系统进行性能层面上的对比分析.

3.4.1 执行时间开销

从设计原理出发可以初步判断,DIMAK 可能引起执行时间方面的性能开销的操作主要包括:

1) 系统启动过程中,校验内核镜像及载入的 KO 模块,并为二者更新动态基线;

2) 进程创建时,或在进程执行过程中发生共享库动态加载时,对被加载的可执行文件进行校验,并为其动态链接/重定位信息生成动态基线.

其他可能引起执行时间开销的操作还包括 DIMAK 在进程执行过程中发生动态代码改写(包括恶意篡改行为和正常的软件热修复)时的校验和基线更新/维护行为.然而,考虑到此类事件并非嵌入式系统运行时出现的常态行为,故本文不将其视作 DIMAK 的常规性能开销.此外,对于占软件系统运行周期主要部分的进程执行阶段,DIMAK 并不产生可测量范围内的常规开销.

本文对执行时间开销的测试采用了一台搭载 8 核 1.35 GHz 主频的 ARM v7 处理器、具有 6 GB 运行内存、搭载内核版本 5.1.0 的 Linux 系统的嵌入式接入设备作为真机测试对象.本文进行的第一项实验分别测试了上述测试设备分别在 DIMAK 功能启用和禁用状态下的系统启动时间、用户态进程创建时间和单个共享类库的动态载入时间.其中,选自开源嵌入式测试集 MiBench 的 9 个实例 dijkstra, patricia, blowfish, SHA, adpcm, CRC32, FFT,

basicmath 和 qsort 被用于测试用户态进程创建时间,而标准系统库 glibc 则被用于测量单个共享类库的载入时长.

如表 3 所示,可以看到 DIMAK 所引入的执行时间开销主要分布于用户态进程的创建过程以及运行时单个共享库的动态载入过程.这主要可以归因于该架构通过对 ELF 文件动态链接/重定位信息的抢先预测以更新和维护其只读结构化数据段动态基线的功能设计.尽管如此,经过 DIMAK 验证的进程创建和单个共享库动态载入过程的总体时间开销仍然处于毫秒级,且 DIMAK 在未涉及到动态链接行为的系统启动过程中所引起的开销仅为 7.532%.考虑到多数嵌入式系统中所运行的用户态进程具有较高的稳定性,即进程不会频繁地被创建/终止,可以认为:通过将主要性能瓶颈从运行时转嫁到加载时,DIMAK 有效地降低了其保护功能为系统带来的性能负担.

在此基础上,本文所进行的第 2 项实验进一步测试了各 MiBench 实例在 DIMAK 功能启用/禁用状态下的实际执行性能,以此评估 DIMAK 可能为嵌入式系统中运行的用户应用程序所带来的执行性能开销.其中,对于每个被测 MiBench 实例,实验中采用其自带的大型测试数据集作为运行输入,并测量其在 DIMAK 启用/禁用情况下的各 10 次完整执行所消耗的平均执行时间.如表 4 所示,实测中各 MiBench 实例在 DIMAK 启用/禁用状态下的平均执行时间相差均在 1%以下.这一结果进一步支持了前述结论,即:由于 DIMAK 的主要开销发生在应用程序加载过程中,其对嵌入式系统带来的实际性能负担几乎可以忽略.

Table 3 Runtime Overhead of DIMAK in Cases of System Booting, Process Creating and Dynamic Library Loading

表 3 DIMAK 在系统启动、进程创建和动态库加载事件中的执行性能开销

系统运行关键节点	标准 Linux 系统/ms	DIMAK/ms	平均开销
系统启动	12.705 4±0.525 3	13.662 4±0.362 8	≈7.532%
进程创建	0.376 6±0.000 0	50.904 2±0.070 3	≈134.17 倍
共享库动态载入	0.011 3±0.001 1	6.321 6±0.174 2	≈558.43 倍

Table 4 Runtime Overhead of DIMAK During the Execution of User Processes

表 4 DIMAK 在用户态进程运行一般过程中的执行性能开销

被测 MiBench 实例	平均执行时间/ms		平均开销
	DIMAK 启用	DIMAK 禁用	
dijkstra	691.215 5	685.139 6	≈0.89%
patricia	3 752.071 1	3 768.450 4	≈−0.43%
blowfish	6 182.783 1	6 140.973 6	≈0.68%
SHA	348.070 1	346.575 5	≈0.43%
adpcm	8 424.296 9	8 452.797 7	≈−0.34%
CRC32	16 343.924 3	16 341.619 3	≈0.01%
FFT	6 508.579 0	6 476.305 7	≈0.50%
basicmath	15 178.1000	15 173.375 8	≈0.03%
qsort	2 493.077 8	2 490.505 7	≈0.10%

3.4.2 运行内存开销

由于嵌入式系统往往倾向于最大化利用计算资源,故其为附加安全功能预留的可扩展空间通常极为有限,使得运行内存开销成为一个 IMA 架构是否适合嵌入式系统的重要指标.而从 DIMAK 的设计原理触发,可以判断其运行内存开销主要包括 3 个方面:

1) 对静态/动态基线组的维护,是 DIMAK 的一项常规内存开销;

2) 在对任意被度量实体实施完整性验证时,所需要的缓冲区、环境变量等关键数据是 DIMAK 的一项临时内存开销;

3) 在对用户态 ELF 文件实施动态链接/重定位信息的抢先预测时,DIMAK 所需维护的诸如依赖

文件列表等关键数据是其另一项临时内存开销.

为了解上述运行内存开销可能给嵌入式系统带来的实际性能负担,本文采用与 3.4.1 节相同的嵌入式接入设备为测试对象,对 DIMAK 进行了两项真机测试,分别模拟了其总体性能开销最大的系统全量扫描过程(即对系统环境内的所有被度量实体进行遍历完整性验证)和可以视为性能开销基本单元的单个共享库动态载入过程.结果显示:

1) 常规状态下,DIMAK 的自身代码占用内存共计为 3 120 KB;

2) 在全量扫描中,DIMAK 对总计 3 481 MB 的运行数据实施了完整性检验,其自身的运行内存占用仅为 2 MB;

3) 在共享库动态载入时,DIMAK 的平均内存开销仅为 4.1 KB.

上述数据充分显示了 DIMAK 在内存开销方面的设计优势,我们有充分的理由相信:即使被应用于硬件性能更为有限的其他嵌入式设备,DIMAK 仍然能够以极低的内存占用实现对系统的运行时完整性验证保护.

4 结 论

本文提出了一种针对嵌入式 Linux 操作系统的实用化完整性度量架构 DIMAK.DIMAK 在设计上采用了依托系统内核空间与 TEE 环境的逐级信任扩展机制,并根据位于内核及用户空间内的可执行文本、静态数据以及动态链接信息等各种被度量实体的不同类型,采用了灵活的静态/动态双基线组的完整性基线维护策略.通过对 Linux 内核代码内的关键函数实施 inline hooking 的方式,DIMAK 所建立的状态感知界面能够在不探入用户空间的情况下感知并介入由用户进程发起的、可能影响系统完整性状态的程序行为,从而保证了该架构完整性验证行为的正确性.此外,借助专门的私有协议设计,DIMAK 还能够在无需用户/内核代码协助的情况下实现对用户态软件热修复功能的支持.真机测试显示,本文所述的完整性度量架构产生的性能开销完全可以满足嵌入式设备场景下的实际应用要求,同时在安全特性方面优于现有同类技术.

作者贡献声明:贾巧雯和马昊玉负责方案设计与论文撰写;厉严和王哲宇负责实验与数据整理;石文昌负责方案和实验指导.

参 考 文 献

[1] Sailer R, Zhang Xiaolan, Jaeger T, et al. Design and implementation of a TCG-based integrity measurement architecture [C] //Proc of the 13th USENIX Security Symp. Berkeley, CA: USENIX Association, 2004: 223-238

[2] Jaeger T, Sailer R, Shankar U. PRIMA: Policy-reduced integrity measurement architecture [C] //Proc of the 11th ACM Symp on Access Control Models and Technologies. New York: ACM, 2006: 19-28

[3] Apvrille A, Gordon D, Hallyn S E, et al. DigSig: Runtime authentication of binaries at kernel level [C] //Proc of the 18th USENIX Large Installation System Administration Conf. Berkeley, CA: USENIX Association, 2004, 4: 59-66

[4] Weng Xiaokang. Research and implementation on integrity protection techniques for terminal computer [D]. Zhengzhou: Information Engineering University, 2014 (in Chinese)
(翁晓康. 终端完整性保护关键技术研究与实现[D]. 郑州: 解放军信息工程大学, 2014)

[5] Li Yu, Zhao Yong, Lin Li, et al. Method of trusted measurement for operating system kernel [J]. Journal of Chinese Computer Systems, 2013, 34(5): 997-1002 (in Chinese)
(李瑜, 赵勇, 林莉,等. 一种操作系统内核完整性度量方法[J]. 小型微型计算机系统, 2013, 34(5): 997-1002)

[6] 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

[7] Petroni N L, Fraser T, Molina J, et al. Copilot—A coprocessor-based kernel runtime integrity monitor [C] //Proc of the 13th USENIX Security Symp. Berkeley, CA: USENIX Association, 2004: 179-194

[8] Liu Ziwen, Feng Dengguo. TPM-based dynamic integrity measurement architecture [J]. Journal of Electronics and 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)

[9] Shi E, Perrig A, Van Doorn L. Bind: A fine-grained attestation service for secure distributed systems [C] //Proc of the 2005 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2005: 154-168

[10] Li Xiao, Shi Wenchang, Liang Zhaohui, et al. Operating system mechanisms for TPM-based lifetime measurement of process integrity [C] //Proc of the 2009 IEEE 6th Int Conf on Mobile Adhoc and Sensor Systems. Piscataway, NJ: IEEE, 2009: 783-789

[11] Azab A M, Ning Peng, Sezer E C, et al. HIMA: A hypervisor-based integrity measurement agent [C] //Proc of the 2009 Annual Computer Security Applications Conf. New York: ACM, 2009: 461-470

[12] Azab A M, Ning Peng, Wang Zhi, et al. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity [C] //Proc of the 17th ACM Conf on Computer and Communications Security. New York: ACM, 2010: 38-49

[13] Stelte B, Koch R, Ullmann M. Towards integrity measurement in virtualized environments—A hypervisor based sensory integrity measurement architecture (SIMA) [C] //Proc of the 2010 IEEE Int Conf on Technologies for Homeland Security. Piscataway, NJ: IEEE, 2010: 106-112

[14] Ba Haihe, Zhou Huaizhe, Ren Jiangchun, et al. Runtime measurement architecture for bytecode integrity in JVM-based cloud [C] //Proc of the 2017 IEEE 36th Symp on Reliable Distributed Systems. Piscataway, NJ: IEEE, 2017: 262-263

[15] Luo Wu, Shen Qingni, Xia Yutang, et al. Container-IMA: A privacy-preserving integrity measurement architecture for containers [C] //Proc of the 22nd Int Symp on Research in Attacks, Intrusions and Defenses. New York: ACM, 2019: 487-500

[16] Davi L, Sadeghi A R, Winandy M. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks [C] //Proc of the 2009 ACM Workshop on Scalable Trusted Computing. New York: ACM, 2009: 49-54

[17] Wehbe T, Mooney V, Keezer D. Hardware-based run-time code integrity in embedded devices [J]. Cryptography, 2018, 2(3): 20

[18] 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: 1-14. DOI: https://doi.org/10.1155/2020/8957641

[19] Ling Zhen, Yan Huaiyu, Shao Xinhui, et al. Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT nodes [J]. Journal of Systems Architecture, 2021, 119: 102240

[20] Duan Jialiang, Cai Guoming, Xu Kaiyong, et al. Integrity measurement based on TEE virtualization architecture [C] // Proc of the 5th Int Conf on Mechanical, Control and Computer Engineering (ICMCCE). Piscataway, NJ: IEEE, 2020: 370-376

[21] Cheng Dongxu, Zhang Chi, Liu Jianwei, et al. An attack-immune trusted architecture for supervisory aircraft hardware [J]. Chinese Journal of Aeronautics, 2021, 34(11): 169-181

Jia Qiaowen, born in 1992. PhD candidate. Her main research interests include concurrent system and program verification, as well as computer system security.

贾巧雯,1992 年生.博士研究生.主要研究方向为并发系统和程序验证、计算机系统安全.

Ma Haoyu, born in 1986. PhD, assistant professor. His main research interests include software security, operating system security, Web security and trusted computing.

马昊玉,1986 年生.博士,讲师.主要研究方向为软件安全、操作系统安全、Web 安全及可信计算.

Li Yan, born in 1996. Master. His main research interests include operating system security and trusted computing.

厉　严,1996 年生.硕士.主要研究方向为操作系统安全及可信计算.

Wang Zheyu, born in 1995. Master. His main research interests include operating system security and trusted computing.

王哲宇,1995 年生.硕士.主要研究方向为操作系统安全及可信计算.

Shi Wenchang, born in 1964. PhD, professor, PhD supervisor. His main research interest is system security in cyberspace.

石文昌,1964 年生.博士,教授,博士生导师.主要研究方向为网络空间系统安全.