# 融合虚拟化和操作系统的动态程序分析框架

潘家晔[1], 沙乐天[2]

[1] 南京邮电大学 现代邮政学院 南京 中国 210003
[2] 南京邮电大学 计算机学院 南京 中国 210042

**摘要** 各种高级恶意代码在网络空间中不断出现, 具有分析对抗能力强、恶意行为更隐蔽等新特点, 对各类信息系统的安全性产生严重威胁。为深度了解恶意代码及相关攻击活动, 需研究更实用和高效的分析方法, 以提高对威胁的分析能力和响应速度。针对二进制程序分析, 尽管已有较多的研究成果, 但随着软硬件技术的发展, 仍面临实用性和灵活性较低、性能和资源开销较高、难以适应新的应用场景等问题。因此在已有工作的基础上, 本文以动态细粒度程序分析为目标, 将操作系统和虚拟机监视器进行深度融合, 提出一种新的二进制程序动态分析方法。该方法充分利用硬件虚拟化新特性对目标程序的执行进行动态拦截, 能够更便捷地对用户模式应用程序进行自动化分析, 并采用新的动态分析相关内存管理方案, 以提高细粒度分析的效率和分析代码构建的灵活性; 同时综合程序执行和指令分析进行分离的策略, 进一步降低分析过程对目标程序运行时的性能影响。本文在 Windows 平台上设计了该方法的原型并实现相应的分析框架, 采用基准程序和实际应用程序进行大量实验, 验证了该方法的可行性和高效性, 并通过数据流分析案例进一步展示了框架在实际分析中具有较高的应用价值。

**关键词** 程序分析; 动态分析; 恶意代码; 系统内核; 硬件虚拟化
**中图法分类号** TP311 DOI 号 10.19363/J.cnki.cn10-1380/tn.2024.07.04

# Practical Dynamic Binary Analysis Framework via Integrating Hypervisor with the Operating System

PAN Jiaye[1], SHA Letian[2]

[1] School of Modern Posts, Nanjing University of Posts and Telecommunications, Nanjing 210003, China
[2] School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210042, China

**Abstract** All kinds of advanced malicious codes are constantly appearing in cyberspace. They have new characteristics such as strong ability of anti-analysis and more concealed malicious behaviors, which pose a serious threat to the security of various information systems. In order to deeply understand the malicious code and the related attack activities, it is necessary to develop more practical and efficient analysis methods to improve the analysis ability and response speed of malicious code. For binary program analysis, although there have been many research achievements, but with the development of hardware and software technology, it still faces many problems such as low practicability and flexibility, high performance and resource overhead, and difficulty in adapting to new application scenarios. Therefore, on the basis of existing research, this paper aims at the dynamic fine-grained program analysis, deeply integrates the operating system with the virtual machine monitor, and proposes a new dynamic analysis method for binary programs. This method makes full use of the new features of hardware virtualization to intercept the execution of the target program dynamically, which can be more convenient to analyze the target program in user mode automatically, and designs a new memory management scheme required for dynamic analysis to improve the efficiency of fine-grained analysis and the flexibility of analysis code construction; at the same time the method combines the strategy of decoupling the native program execution and the instruction analysis to further reduce the performance impact of analysis process on the target program at runtime. In this paper, a prototype of the method is designed on the Windows platform and the corresponding analysis framework is also implemented. The feasibility and efficiency of this method are verified by a large number of experiments with the benchmark programs and practical application programs, and the high application value of this framework in practical analysis is further demonstrated by the real data flow analysis cases.

**Key words** program analysis; dynamic analysis; malicious code; system kernel; hardware virtualization

**通讯作者:** 沙乐天, 博士, 副教授, Email: ltsha@njupt.edu.cn。
本课题得到国家自然科学基金(No. 62072253)和南京邮电大学科研基金(No. NY221036)资助。
收稿日期: 2022-08-17; 修改日期: 2023-01-05; 定稿日期: 2024-03-15

## 1 引言

随着恶意代码检测和对抗技术以及操作系统平台的发展, 攻击者不断创造新型恶意程序, 以多种多样的部署形态和植入方法威胁着目标信息系统, 例如: 勒索软件、挖矿程序以及 APT 活动中高级恶意代码等[1-3]。这些恶意程序普遍具有隐蔽性强、敏感活动频率低、对抗分析手段多等特点, 并采用系统内置程序和第三方合法程序组合完成恶意功能[4-5]。恶意代码的分析和检测面临新的挑战, 在分析程序行为的基础上, 多数场景需要对目标程序进行更深入和快速的分析, 例如: 提取通信域名生成算法、获取在内存释放和执行的功能模块, 以及针对一些合法程序的特殊隐藏功能进行挖掘和分析等, 从而为提取威胁情报信息, 深挖攻击活动线索, 提高威胁响应能力提供支撑。

无论是对恶意程序分析还是对其所采用的合法程序进行分析, 主要的分析对象是二进制程序。二进制程序分析技术在漏洞挖掘、软件可靠性测试和恶意代码分析等很多领域都有着广泛的应用[1,6-7], 静态分析和动态分析是常用的两类分析技术[8-9]。当前多数恶意程序将其恶意功能隐藏在大量正常运行的代码中并进行深度混淆, 而细粒度的动态分析可以获得目标程序实际执行的路径, 能够深度挖掘和发现潜在的恶意行为, 更能满足当前形势下实际分析的需要。很多年以来, 在二进制程序的分析实践中, 对各类程序进行函数级别的行为分析一直占据着重要位置, 包括基于内核驱动或外部虚拟机构建沙箱对目标程序的各类操作进行记录和分析[10-11], 通过对关键代码进行挂钩和调试来获取程序运行信息等[12]。然而仅基于程序行为分析难以跟踪在内存中完成的代码注入、数据变换加密和信息窃取等操作, 而传统沙箱技术在对抗检测方面仍然存在挑战, 难以完全满足新型恶意代码分析和威胁情报信息提取的需求。

指令级别的细粒度分析能够对整个程序或局部代码进行深入分析, 但由于部署环境或分析性能等原因, 细粒度的数据流分析或污点分析方法难以在实践中得到广泛和有效的运用。在过去很多年中, 研究人员围绕细粒度分析主题也一直在开展深入研究[13], 并提出和实现很多有价值的早期方案, 如: Dytan[14]、Panorama[15]、libdft[16], 展现了分析的价值。为进一步提高可用性, 研究人员又提出了新的分析框架和改进的分析方法, 例如: 利用硬件特性来提高分析透明性[17]; 基于分离或离线方法来提高分析性能[18-19]; 提出轻量化的分析框架来提高易用性[20]。一方面从新的角度提出和设计支撑程序执行和分析的基础软件和平台, 如: TinyInst[21]、DECAF[22]等; 另一方面依托基础平台, 提出并实现效率更高的分析方法, 以满足不同场景的需求等[23-24]。但在分析性能、可部署性以及检测对抗能力等方面仍然面临诸多挑战, 难以完全适用于所有分析场景。事实上, 在如今安全软件和操作系统访问控制机制更加成熟的环境下, 恶意程序更注重将其功能混杂在正常程序当中, 而多数深入分析是针对应用层的程序或者代码片段[25]。在此情况下, 全系统范围的数据流分析框架由于灵活性或者复杂度等问题其作用并不能得到最佳的发挥。随着形势的发展, 我们更需要轻量化、针对性的细粒度分析方法, 同时具有较强的分析隐蔽能力, 可以随时随地的对目标程序进行深度分析。

因此, 围绕二进制程序深度分析的需求, 在已有研究工作的基础上, 本文提出一种新的混合二进制程序动态分析方法 HyDBA(Hybrid Dynamic Binary Analysis), 旨在进一步提高细粒度分析的性能和易用性。该方法以虚拟化监视器和操作系统内核为基础平台[26], 并对其进一步融合, 强化对目标程序执行的拦截和控制; 以解耦污点分析的思想为核心[24], 构建在线分析框架, 提出并设计了新的分析阶段内存管理方案, 提高细粒度分析的效率和扩展性, 有效控制系统资源的开销。与其它方法相比, 该方法融合虚拟机和操作系统, 可以更便捷的进行部署, 并通过构建紧凑的分析代码, 进一步提升针对应用层程序的分析性能, 兼顾分析隐蔽性和易用性; 充分利用硬件虚拟化的多种特性, 设计基于在线解耦分析的代码构建和内存管理方法, 增强了对目标程序的拦截和分析能力, 并在更常用的 Windows 平台上进行研究和测试, 提高了分析框架的实际应用价值。

本文的主要贡献如下:

1)提出一种新的轻量级二进制程序动态分析框架, 充分融合虚拟化监视器和操作系统内核, 以在线细粒度解耦分析为核心, 能够更便捷地实现对用户模式下应用程序的自动化分析, 进一步提高了细粒度分析的性能和适用性。

2)利用虚拟化特性, 提出一种新的用于动态分析的内存管理方案, 提高了运行时信息记录和内存分析状态访问的效率, 降低了目标程序和分析代码的执行开销, 提高分析代码构建的灵活性。

3)在 Windows 平台上设计并实现框架的原型系统, 提高了分析框架的可应用性, 采用多种类型的

程序进行测试, 验证该框架具有较好的分析性能, 并通过实际场景验证框架进行动态数据流分析的有效性。

## 2 相关研究工作

近年来, 有较多关于二进制程序分析的研究工作, 并形成了一系列能够应用于实际环境下的分析系统和工具。随着技术发展, 针对二进制程序的分析方法也随着系统和软件开发技术的发展而不断得到完善。当前, 新的研究工作主要利用硬件技术或者系统优化设计以提高分析效率和适用性[27-28], 同时针对特定的应用场景, 如漏洞挖掘、恶意代码分析、物联网架构来修改或重新设计分析框架以更好的来解决实际问题。主要从以下两个方面来重点介绍和讨论与本文相关的研究工作。

一是侧重于基础分析框架构建的相关研究工作。以 Pin[29]、DynamoRIO[30]、Valgrind[31]为代表的工具是著名的针对单个应用程序的插桩分析基础平台, 并提供了丰富的接口, 在实际工作中得到广泛应用, 但其主要采用应用层的技术进行实现, 可扩展性和对抗能力较低。近年出现的 Frida[12]、QBDI[32]也是流行的二进制分析框架, 提供包括基于脚本语言的用户分析接口, 但是在细粒度分析效率方面存在不足, 更适合于对目标进行调试分析。TinyInst[21]也是轻量级插桩分析框架, 可以实现灵活的分析功能, 同样存在分析效率和对抗能力问题; Instrew[33]利用 LLVM 工具来实现二进制程序翻译和分析, 进一步提升了分析性能, 但依赖于程序得到正确的转换。还有些研究工作基于虚拟机技术构建面向全系统的分析框架, PinOS[34]和 PEMU[35]是针对 Pin 的扩展, 主要以实现兼容性为目的; DRAKVUF[10]、Panorama[15]可以在系统范围对目标程序进行分析, 获取程序执行的行为, 并且后者还能够支持细粒度的污点分析, 但是存在部署易用性等问题。Ether[36]、SPIDER[37]都是基于虚拟化技术实现的分析框架, 提高了分析的隐蔽性, 但在自动化分析程度方面存在不足。MALT[38-39]等工作利用硬件特性进一步提高了分析的隐蔽性, 但增加了语义获取和深入分析的复杂性。BAHK[20]通过拦截内存访问, 并在此基础上构建自动化的细粒度分析方法, 但在针对访存密集型的程序分析时, 其性能仍然面临挑战。OASIS[27]也是一个基于虚拟化的分析框架, 将原程序执行和分析过程交替进行, 但在对目标程序进行全面分析时仍面临性能方面的挑战。本文提出的 HyDBA 是一个全新的分析原型框架, 其在基于硬件虚拟化特性的基础上, 进一步深度融合系统内核, 实现对应用程序的自动化分析, 框架核心代码位于系统内核层面, 保证分析隐蔽能力的同时, 能有更好的可部署性和分析性能。

二是侧重于细粒度程序分析的相关研究工作。二进制程序分析需要进一步提高分析效率和自动化程度, 另一方面需要根据分析需求来设计和改进分析方法。ShadowReplica[18]和 TaintPipe[24]是比较典型的解耦分析方法, 前者在对程序进行静态分析的基础上对动态分析进行了优化, 后者利用管道思想, 通过多个线程并发和符号化技术来提高分析能力和效率。StraightTaint[40]为离线分析方法, 通过记录程序执行的控制流, 并离线构建分析, 降低了对目标执行过程的影响, 离线分析的构建过程相对复杂, 难以准确重现程序动态执行过程。ConDySTA[28]则将动态污点分析的结果用于静态分析, 提高对复杂代码的分析能力, 但主要针对移动平台的应用程序。TaintRabbit[41]基于 JIT 技术对动态插桩方法进行优化, 提高代码生成速度, 以进一步提高具体应用场景的分析效率, 仍借助于应用层的实现技术。此外, 各种基于记录和重放的分析方法[42-43], 也是类似于离线分析的方法, 通过记录动态运行信息来降低对原程序执行的影响, 可以重复进行重放分析, 但也缺乏灵活性, 以及应对程序执行环境变化情况。本文提出的 HyDBA 方法在实现定制化分析框架的基础上, 实现对目标应用程序的自动化细粒度分析, 它构建了基于解耦的在线分析框架, 能够进一步提高分析性能并更有利于控制系统资源的开销, 其目的是进一步提升基于在线插桩的细粒度分析能力, 能为特定应用场景提供技术支撑。

## 3 框架设计与描述

当前高级恶意代码和二进制程序的分析需求主要集中于 Windows 平台, 与 Linux 等系统相比, Windows 系统没有源代码可以参考和重新编译, 在其上方面构建分析框架需要克服一系列的困难, 如: 需分析和研究系统组件的工作原理; 选择合适的位置对内核代码进行修补等。本文直接以 Windows 平台为基础设计和实现分析框架原型并开展实验, 可以验证分析框架的可行性, 也可使其能在实际环境中进行应用。

### 3.1 总体架构

分析框架的总体架构如图 1 所示, 图中忽略了虚拟机管理器的细节部分, 而着重展示对目标程序进行分析的主要流程和涉及的主要组件。框架基于

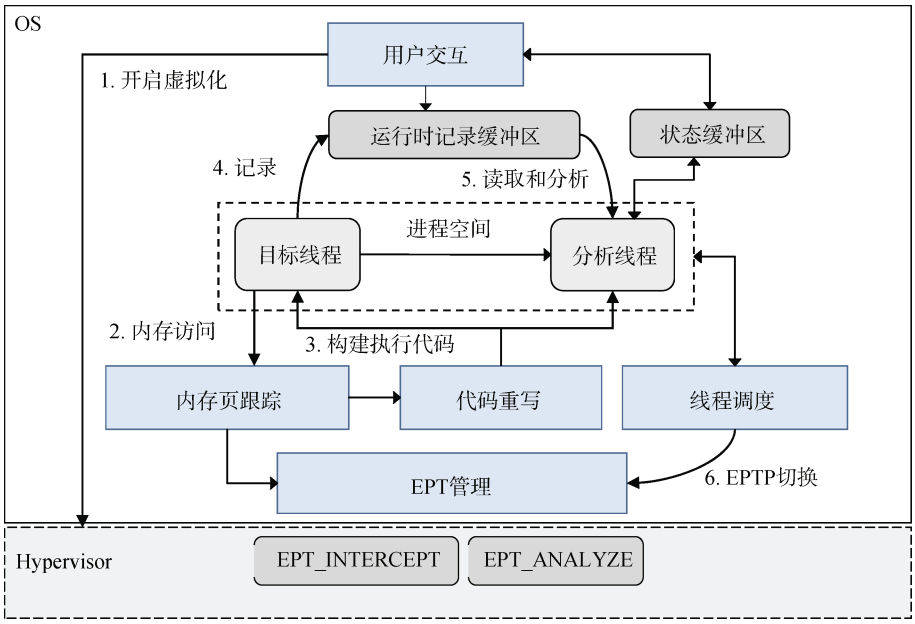


**图 1　总体架构**

**Figure 1　Architecture of HyDBA**

Intel VT 的硬件虚拟化技术构建轻量级虚拟机监视器, 并综合利用 EPT、#VE 和 EPTP Switching 等特性, 使得大多数因虚拟化机制而引起的特殊处理都可在操作系统内核中完成。EPT(Extended Page Table)为扩展页表机制, 是一种支持物理内存虚拟化的特性。在采用 EPT 进行内存访问时, 虚拟机中客户物理地址将通过 EPT 表转换映射到真实的宿主物理地址, 以替代传统基于软件的影子内存映射方案。通过硬件层面来降低内存虚拟化的复杂性并提升性能, 而 #VE 为虚拟化异常、EPTP Switching 为 EPT 表切换机制, 都是与之配套的新特性, 在较新的 Intel 处理器中这些特性均被支持。框架充分运用硬件虚拟化的特性, 在虚拟化监视器中设置两个EPT表来对虚拟机物理页面进行管理和控制, 分别用于系统其他线程正常执行和分析线程执行过程。目标程序执行时创建的每个线程对应一个分析线程, 用于执行分析代码, 完成对原程序相关代码的分析。同时与线程相关的还有用于存储运行时信息和分析结果的缓冲区。

如图 1 所示, 当对目标程序开始分析时, 首先将目标操作系统转化为在虚拟化环境中运行, 而后执行目标程序。位于内核中的内存跟踪模块对目标进程的内存分配和释放情况进行跟踪, 同时初始化两个 EPT 表中的物理页面映射结构, 一方面实现对目标程序执行的拦截和代码插桩[26], 另一方面为控制目标程序运行时信息记录, 也保证分析线程执行时能够正确的访问存储分析结果的内存区域。代码重写模块除了构建用于记录运行时信息的目标执行代码之外, 还需构建对原目标代码进行分析的代码, 供分析线程执行。为提高内存访问效率, 在记录运行时信息时采用多缓冲方法[24], 当缓冲区满或目标线程被换出时, 将切换用于记录的缓冲区。而分析线程将持续执行分析代码, 并读取记录缓冲区, 直到其中所有数据都已被处理。由于分析代码与程序执行代码是分离的, 在目标线程执行时, 通过在运行时缓冲区中设置特殊数据对分析线程进行控制, 可通知分析代码进一步检查分析状态和结果。此外通过对系统线程调度过程进行修改, 以加速分析代码的执行, 可缓解因解耦分析所带来的分析延迟和同步问题。

## 3.2　程序执行和分析

本文的分析方法延续将程序执行与分析代码进行解耦的思想[24], 其优点是能够降低对目标程序执行的性能影响。通过基于硬件虚拟化的执行拦截框架来实现对目标程序的动态插桩, 在新生成的程序执行代码中, 记录分析时所需要的运行时信息。此外, 在生成目标执行代码的同时构建相应的分析代码。

在目标程序执行时, 框架将对其所执行的代码进行跟踪, 在此基础上利用 EPT 机制提供的物理页面执行访问控制功能对目标程序的指令执行进行拦截, 并基于新的虚拟异常处理机制在客户机内核中对异常做进一步处理。当在 EPT 中将目标页面的访问权限修改为禁止执行后, 目标程序的任意指令执行均会产生异常[26]。框架采用称为“异常重定向”的方法来避免频繁出现的执行中断造成性能急剧下

降。在处理异常时以程序基本块为单位, 当该基本块中某指令执行触发拦截后, 以此地址开始进行反汇编和分析直到分支指令处, 并在新的内存区域重新构建相应的代码块, 而后将原始执行重定向到新分配的代码块中。进一步地完成指令插桩和分析代码的构建。

### 3.2.1 记录代码的构建

在使用动态污点分析技术对程序进行分析时, 需要记录程序运行时信息时, 包括寻址时的寄存器信息。与分析过程相比, 记录过程只涉及少量的通用寄存器, 并且不改变标志寄存器 EFLAGS 的状态, 极大减少对原程序执行的影响。框架以程序基本块为单位来构建记录代码, 并使用线程局部存储 TLS 来暂存占用的寄存器的值, 并在基本块结束前更新记录缓冲区的指针, 进一步减少上下文保存和重载的次数, 如图 2 所示。当针对某指令的信息记录代码构建完成后, 先不恢复之前保存的上下文信息, 而是继续对基本块进行解析, 直到下一个需要记录运行时信息的指令或者基本块结束位置。并将两者之间的其他指令暂存到临时缓冲区中, 在处理下一个需记录指令时, 按顺序插入生成的目标执行代码中。记录缓冲区指针的更新操作也采用类似方法完成, 进一步压缩目标代码执行时内存读写的数量。

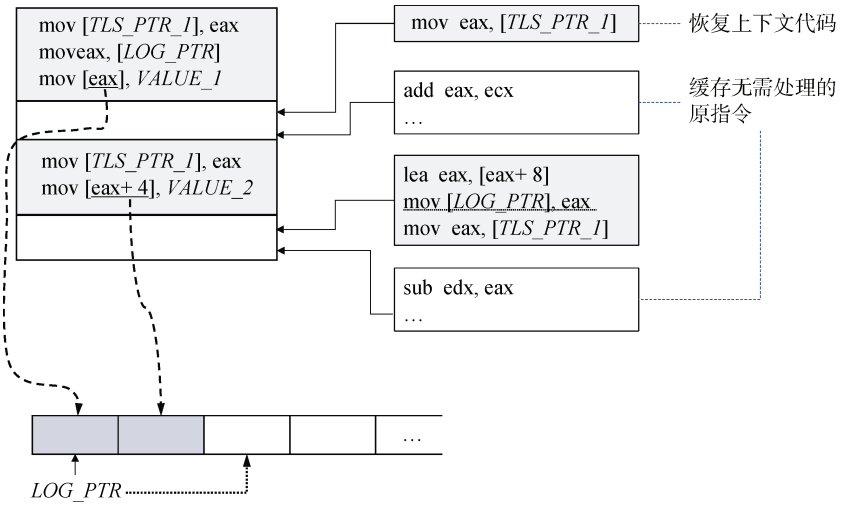


**图 2 执行代码构建过程**

**Figure 2 Construction process of the execution code**

### 3.2.2 线程调度处理

框架对系统进线程的创建和退出过程进行监控, 目标程序在执行过程时如果创建新的线程, 框架将同时为其创建相应的分析线程, 即每个目标线程对应一个分析线程。该分析线程属于内核线程, 将附加于目标进程空间运行, 以便于同时访问内核和用户空间的数据。分析线程主要完成对原目标线程执行代码的分析, 不同线程间互不干扰, 其运行可由原目标线程通过在记录缓冲区中插入特殊数据的方式来进行管理和控制。在对普遍存在的多线程程序分析中, 控制分析线程数量, 有利于平衡系统进线程资源以及处理器负载, 降低分析框架对目标平台配置的要求。采用单个分析线程也可以降低对分析同步要求高场景的影响, 框架同时对构建的分析代码进行优化, 以保证分析线程能够及时完成分析任务。

如上文总体架构部分所述, 框架引入两个 EPT 表来管理虚拟机物理内存页面, 分别用于目标程序分析过程和系统中程序的原来执行过程, 并确保分析线程所在的 CPU 当前用的 EPT 表为 ANALYSIS, 从而保证分析过程的正确性, 并对目标线程和系统其他程序运行不产生干扰。为实现该目标, 需对操作系统与线程调度相关的代码进行少量修改, 在拦截线程调度过程的基础上, 过滤与分析相关的线程并进行特殊处理。如图 3 所示, 线程调度过程中的处理主要包括两个方面, 一是线程因睡眠或时间片用完而引起的正常换入和换出。在此情况下, 如果分析线程被换入到目标 CPU 上, 则将目标 CPU 所关联的 EPT 表设为 ANALYSIS, 并刷新 TLB 缓存。而在分析线程被换出或其他线程被换入时, 则需保证目标 CPU 将使用 INTERCEPT 表进行宿主物理地址寻址。二是分析线程在执行核心分析代码时, 其所在 CPU 可能去处理中断请求, 并挂起原来分析线程的执行。对于这种情况, 需对内核中部分外部中断处理例程进行拦截和过滤, 并在 CPU 执行中断处理代码和离开时, 切换当前 CPU 所使用的 EPT 表, 即离开分析代码时采用 INTERCEPT, 反之则采用 ANALYSIS。

此外，分析线程在执行过程中，还会存在主动离开核心分析代码，去执行其他系统函数情况，例如：调用系统函数去处理同步过程、在执行过程中产生内存访问错误等。此时与上述情况一样，分析框架将对这些行为进行拦截，并加入额外的 EPT 表的检查和切换过程，以保证系统的稳定性和目标程序的正确执行和分析。

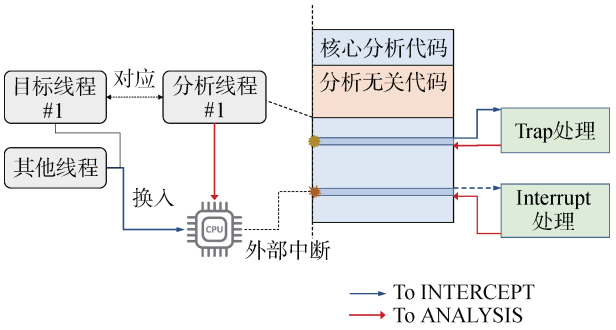


**图 3　线程执行和调度处理**

**Figure 3　Thread execution and scheduling process**

### 3.2.3　缓冲切换过程

解耦分析依赖于使用内存缓冲区来保存分析所需的程序运行时信息，框架同样采用多个缓冲区的方案[24]，而不同的是，为减少生成的目标执行代码规模，采用访问特殊内存地址以触发虚拟化异常的方法来实现记录缓冲区的切换。为使得分析线程能够尽快完成相关代码分析，紧跟原目标线程的执行，应创建较小的缓冲区。在分析时，为每个目标线程分配一定数量的缓冲区，并在相关的 EPT 结构中将每个缓冲区最后一个页面的访问权限设置为不可读写。对于原目标线程，其在写入缓冲区造成溢出时将产生 EPT 异常，并在内核异常处理过程中完成缓冲区的切换，避免了在目标代码执行过程中加入额外的边界检查。而对于分析线程，由于其有与线程独立的上下文，因此采用边界检查的方式来简化缓冲区的读写判断和切换。

需要注意的是，除记录缓冲区满时要进行切换外，当目标程序线程被挂起或换出所在 CPU 时，同样需要切换其使用的缓冲区。主要目的是提高分析线程的执行效率，并且防止目标线程挂起后，分析线程处于缓冲区饥饿状态，导致分析任务无法及时完成。如图 4 所示，缓冲区切换包含两种情况，在第(I)种情况下，缓冲区满后其操作指针将指向下一个空闲的缓冲区。而在第(II)种情况下，当前发生目标线程切换时，由于难以获取准确的缓冲区指针，此时先将当前缓冲区交由分析线程去处理，并将缓冲区指针邻近的页面设为不可访问。这样当目标线程下一次被调度运行后，将会产生访问异常，而后框架将更新缓冲区实际的写入长度，并重置缓冲区状态。

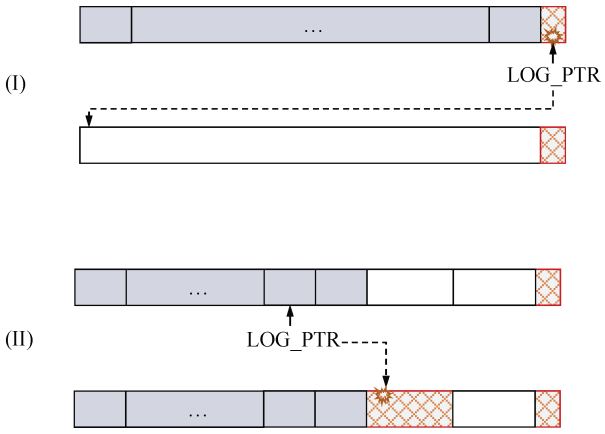


**图 4　缓冲区切换过程**

**Figure 4　Buffer switching process**

框架采用系统提供的内核同步函数来处理缓冲区的读写过程，优点是能够合理的控制线程的 CPU 使用率，并且采用系统提供的 DPC 和内核 APC 机制来处理缓冲区的切换，可提高分析框架运行的稳定性。

## 3.3　分析代码构建

如上文所述，在生成目标执行代码时，以基本块为单位进行，并同时根据目标代码记录的信息和分析需求来构建对应的分析代码。由于分析代码将由专用的分析线程执行，因此在执行时拥有独立的上下文环境，分析代码的构建将更加灵活。一是分析代码主要专注于核心的分析功能，而用户交互过程则通过特殊的内存访问异常来完成；二是利用虚拟化环境下的物理内存寻址映射可以简化分析状态的访问和传递过程。因此在进行动态污点分析时，可更加容易构建所需的分析代码。如图 5 所示，对图中的指令进行分析，在执行代码中记录寻址涉及的两个寄存器值，在分析代码中获取相应的地址值，而后直接传递污点分析状态，减少了上下文状态保存和指针初始化过程。

### 3.3.1　目标代码修改处理

对于每个生成的程序基本块，只在对应的分析代码开始处检查整个代码块所需的运行时记录数量，在运行时如果发现缓冲区中的剩余数量不足，则将转入缓冲区切换过程进行处理。框架采用与目标执行代码链接类似的方法来将不同的分析代码块链接在一起，这样通过分析代码仍然可以直观地了解原代码的分支结构。对于直接转移指令，直接从分析代码中获得目标跳转地址，对于间接转移指令，其目标跳转地址则从记录缓冲区中获取。由于程序原来

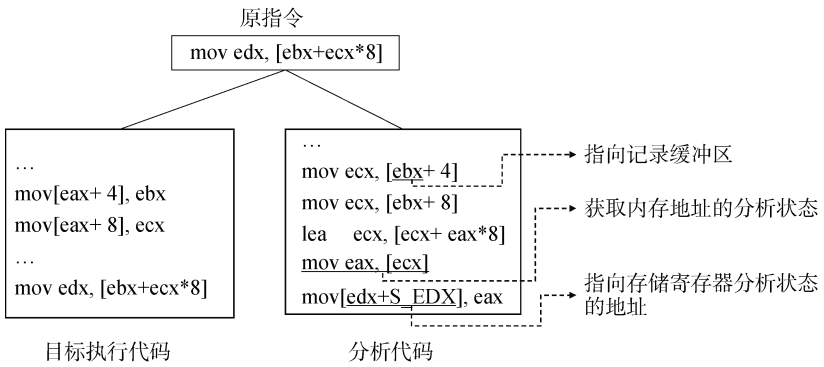


**图 5　动态分析代码示例**

**Figure 5　An illustration of dynamic analysis code**

的基本块在执行过程中可能会被修改, 导致对应的分析代码也出现变化。为处理这种情况, 目标执行代码在记录运行时信息时, 同时记录实时的分析代码所在的起始内存地址。

框架采用热补丁技术在分析代码开头加上空指令, 并在其中包含原执行代码的地址, 如图 6 所示。当目标代码发生修改后, 框架将重新构建分析代码, 此时环境中将会存在多个对应的分析代码块。对于条件指令, 由于在分析代码中跳转目标地址已经硬编码在执行的代码中, 通过对已存在的跳转目标代码的开头进行动态修改, 使其直接从缓冲区中获得运行时记录的目标地址去执行新的对应的分析代码。而对于在目标代码修改后发生的链接, 则将采用最新生成的分析代码来完成。

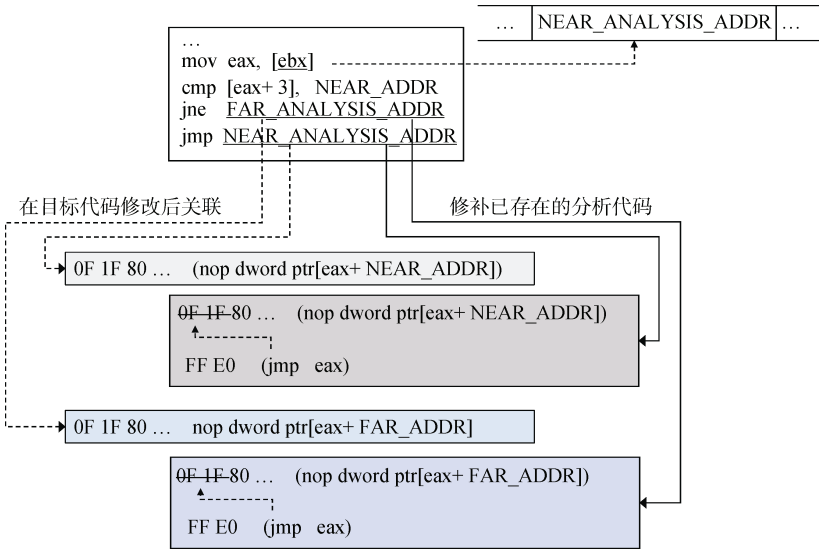


**图 6　目标代码修改后的分析代码调整**

**Figure 6　Analysis code adjustment after object code modification**

### 3.3.2　分析状态检查

为减少分析代码中不必要的检查操作, 框架同样采用访问特殊内存地址产生虚拟化异常的方式来获取和检查分析结果。由于分析与执行过程是分开进行的, 因此分析代码中对应的检查点会存在滞后的情况, 并且同一代码块会被多个线程或多种场景下被调用, 因而只有在特定的条件下才需要触发检查。如图 7 所示, 对于需要进行检查的指令位置, 在其生成的执行代码中多增加一项记录操作, 根据实际执行情况, 分别写入会或不会产生 EPT 异常的特殊内存地址; 而在对应的分析代码中同样增加对该地址的访问操作。选取的特殊地址对应的内存页面在不同 EPT 表中均已预先设置读写访问限制, 分析代码在执行时如果产生访问异常, 则在异常处理过程进一步进行分析状态检查。此外创建一个先进先出的检查队列用来传递状态检查时所需要的参数。事实上, 多数污点分析的 Source 和 Sink 点都设置在系统调用函数处, 可以直接通过拦截内核函数来完成, 而不需要重写用户态代码。

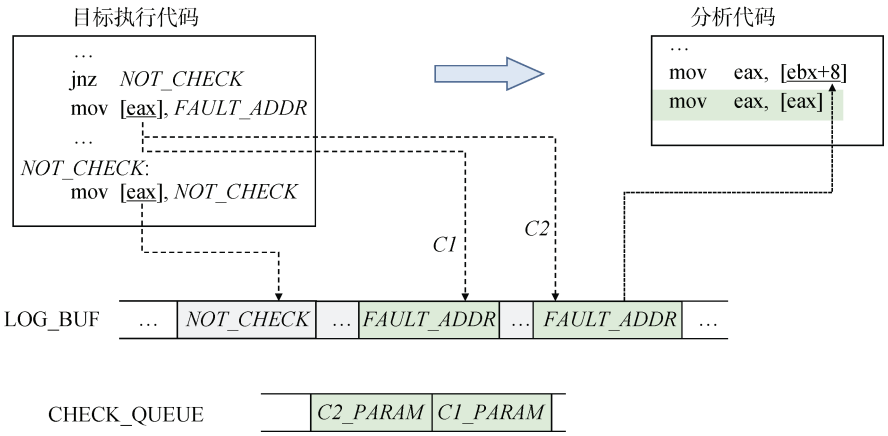


**图 7 分析状态检查**

**Figure 7 Analysis status check**

### 3.4 内存管理

在目标程序执行并完成进程初始化后, 分析框架会预分配一定大小的内存空间用于目标代码重写、分析状态的存储以及分析辅助等。其中, 用于目标代码重写的内存分别属于不同的 CPU 核, 可避免在重写时面临的访问同步问题。根据目标程序运行时内存实际使用情况来动态分配用于存储内存地址分析状态的内存空间, 即当操作系统为目标进程的虚拟地址实际分配物理页面时, 框架才为虚拟地址分配对应的分析状态存储页面。

#### 3.4.1 分析状态结构映射

为使分析代码能够快速访问分析状态和获取分析结果, 框架继续采用虚拟化特性在目标进程空间中构建一套新的内存映射方案, 以提高分析代码访存的便捷性, 同时避免对目标程序的运行造成影响。主要思想是通过调整页目录所在物理页的映射结构, 使得针对相同虚拟地址在不同的模式下访问不同的物理页面, 通过构建和使用不同的 EPT 表来实现。为此, 分析框架在运行后将会从非分页内存中预分配一定数量的虚拟地址并映射物理地址作为保留, 以用于特殊 EPT 结构的构建。如图 8 所示, 页目录地址 2de83000 在不同 EPTP 下将映射为不同的宿主机物理页面, 因此, 虚拟地址 3f70000 经过不同的 EPT 表进行转译后得到不同的物理地址, 分别存储目标程序产生的原始数据和分析代码使用的分析状态, 这样通过访问相同的虚拟地址即可根据场景需要得到不同的数据, 极大提高了分析代码构建的灵活性和执行效率。例如: 在 32 位 PAE 模式下, PDPT 所指向的页目录包括 4 个, 前两个主要用于用户模式地址的转译, 所以在分析时只需调整 2 个页目录客户物理页面到宿主机物理页面的映射。PAE 即物理地址扩展, 主要为 32 位操作系统提供 4GB 以上物理内存支持, 其采用的就是 64 位长度的页表项以及与 64 位架构类似的页表结构, 因此在 64 位平台下将可采用同样方式实现页目录的映射。此外, 虚拟地址转译相关的页表项空间同样也进行重新分配, 当创建虚拟地址对应的分析状态存储空间时, 同时检查页表结构并进行分配。针对新构建的页表结构, 在 32 位系统中可以直接进行预分配, 因为最大情况需消耗 4MB 空间, 并可以采用内核空间地址。但在 64 位平台下, 由于虚拟地址空间太大, 应按实际需求进行动态分配。

尽管额外分配的页目录、页表以及用于存储虚拟地址分析状态的内存空间在 EPT 结构中以物理页面形式存在, 但其仍需被映射至目标进程或者系统内核的虚拟地址, 因此无论目标代码还是分析代码均可以直接通过虚拟地址来访问分析状态, 可以为分析过程中的检查和交互提供便利。当内核或目标虚拟地址空间不足时, 可以采用傀儡进程的虚拟空间来对物理页面进行保留和管理; 而当系统物理地址空间不足时, 优先保证框架的正常运行。此时内存消耗大的应用程序可能会出现页面换出的情况, 框架则在拦截系统页面访问异常函数的基础上对其进行处理。但针对非开源系统需做进一步分析以增加对未知流程的覆盖面。

#### 3.4.2 动态回收与重加载

在目标程序执行的过程中, 目标程序可能会主动释放不再使用的内存空间, 而与之对应的分析状态存储空间也应随之同步进行, 否则可能会造成内存资源不足的情况。对于状态存储空间, 需根据平台的配置情况, 为其分配的空间大小设置限值。由于分析代码的执行会滞后于目标代码, 当目标程序释放特定内存地址后, 其关联的代码的分析可能还未完成, 因此需要延迟释放存储目标内存地址分析状态

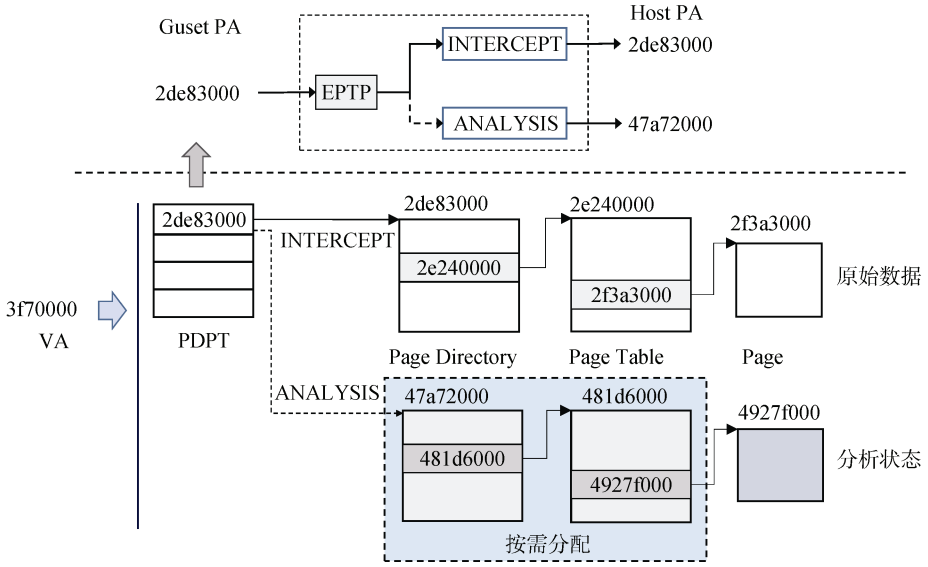


**图 8　内存地址映射示意图**

**Figure 8　Memory address mapping diagram**

的辅助内存空间, 框架采用一个延迟回收队列来处理该问题。如图 9 所示, 当目标程序释放虚拟内存地址时(如 A1、A2), 将其对应的状态存储页面插入到回收队列尾部, 如果该地址后又被重新分配(如 A1), 则重新标记其状态存储页为在用状态。在状态存储页面的额度分配完后, 将从回收队列的头部进行分配, 其中跳过标记为在用状态的空间。

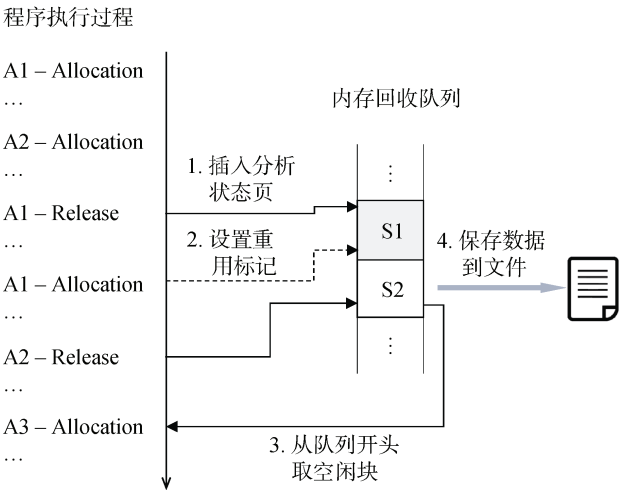


**图 9　状态内存回收和重用过程**

**Figure 9　State memory recycling and reuse process**

当队列中某空间(如 S2)被分配给新的虚拟地址后, 为防止分析代码还需再次访问, 可以将其内容暂时保存到磁盘文件中, 同时在分析 EPT 页表中将原虚拟地址对应的页面调整为不可访问, 这样在分析代码再次访问时则会出现异常, 而后框架将重新为其分配新的内存空间并加载原来的数据, 如图 9 所示。通过该策略使得在不影响分析的情况下, 内存空间得到更有效的利用。

## 4　实验与分析

本文基于 Windows 操作系统实现了所提出的分析框架的原型系统, 核心代码共享在 Github 平台[44]。实验和分析具体在 Windows 10 (1503) 32 位系统上进行, 主要为简化工程开发量并快速验证原型, 事实上在 64 位操作系统环境下同样可以完成框架实现, 并需要增加额外的处理, 一方面要增加对 64 位指令的支持和分析, 而另一方面要按需动态分配内存空间, 用于管理目标程序的内存页面和代码块的状态, 并对映射方法进行优化。在实验时所采用的硬件设备为普通用户终端, 具体配置为: Intel i5, 内存 4GB, 系统盘为 120GB 固态盘, 数据盘为 1TB(7200 转)。在实验过程中, 首先选择基准程序进行性能测试, 评估分析框架对目标程序的影响以及分析性能, 而后采用系统常用应用程序, 并结合具体使用场景来验证利用框架进行动态数据流分析的有效性。

### 4.1　性能测试

对于程序执行拦截和动态插桩的部分, 在之前相关研究工作中已经进行了较多的实验和讨论, 因此本文重点关注程序运行时信息记录、异常处理以及分析代码的执行速度等方面。

由于分析框架可以直接在目标平台上部署运行, 本文采用在 Windows 平台中有着广泛应用的 Intel Pin 工具作为实验比较参照物, 并选择 SPEC CPU 2017 中可以在实验平台上进行编译的程序作为目标

测试程序, 测试负载选择 train 或 ref。首先, 评估在拦截和动态插桩的基础上构建的记录运行时信息的代码的执行效率。实验时使用 Pin 提供的接口来实现上述功能, 需插桩和分析的指令包括应用程序常用的指令[26], 并且在记录缓冲区满时不进行切换而直接复用原来的缓冲区。而在采用 HyDBA 框架进行实验时, 补充测试另一种情况, 即当目标程序使用的记录缓冲区满时, 将其切换下一个空闲的缓冲区, 并且分析线程只进行缓冲区切换而不执行实际的分析代码, 其目的是评估多个缓冲区同步和切换对分析性能的影响, 如图 10 中 HyDBA_RS 所示。在实验过程中, 为每个线程设置 2 个大小为 1MB 记录缓冲区, 原则是选用较小缓冲区。

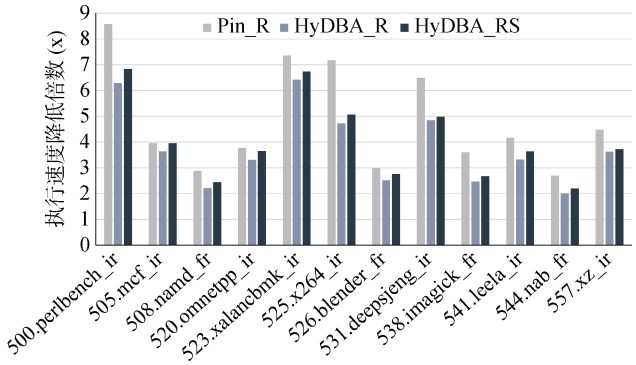


**图 10　基准程序在记录运行信息时的执行性能降低情况**

**Figure 10　Performance degradation of benchmark program execution with runtime information recording**

实验过程中分别记录各种场景下的程序执行时间, 而后以原程序执行时间为基准值计算速度降低情况, 实验结果如图 10 所示。众所周知, Pin 所提供的数据记录方案已经是非常优化的, 与其相比, HyDBA 对于各类基准程序均能够进一步降低程序执行时的性能开销, 因为在记录时采用更精简的指令, 在一定程度上抵消了插桩生成的目标代码在内存布局优化方面的不足影响。在当前实验中, 框架生成目标执行代码的同时构建分析代码, 尽管分析代码未被实际执行, 但从结果看, 分析代码的构建过程对目标程序的执行影响较小。另外从图 10 中可以看出, 记录缓冲区的同步和切换过程对目标程序的执行存在一定的影响但并不显著。与程序执行相关的更详细数据如表 1 所示, 从中可以看到, 在对以浮点指令执行为主的程序进行拦截时, 对程序进行插桩的指令数相对较少, 从而对其执行性能影响也较小。而当需要记录的运行时信息更多时, 引起的缓冲区数量切换也会增加, 但对性能影响不明显, 因为缓冲区规模小且连续分配, 处理器在写入时不需要频繁刷新缓存。

**表 1　目标程序执行和拦截情况统计**

**Table 1　Statistics of object program execution and interception**

| 程序名 | 原始执行时间(s) | 基本块(k) | #VE(k) | 指令数 | | 缓冲切换数(k) |
|---|---|---|---|---|---|---|
| | | | | 解析(k) | 插桩(%) | |
| 500.perlbench | 89.3 | 36.6 | 42.3 | 167.8 | 41.8 | 1961.5 |
| 505.mcf | 51.9 | 10.9 | 12.0 | 53.2 | 41.9 | 635.2 |
| 508.namd | 30.4 | 14.5 | 19.5 | 88.6 | 37.3 | 235.3 |
| 520.omnetpp | 64.2 | 36.5 | 56.3 | 179.1 | 45.3 | 555.8 |
| 523.xalancbmk | 50.1 | 32.6 | 49.8 | 173.2 | 51.5 | 959.7 |
| 525.x264 | 25.8 | 17.9 | 22.6 | 109.8 | 43.9 | 542.2 |
| 526.blender | 119.0 | 56.2 | 67.1 | 287.2 | 45.5 | 1040.0 |
| 531.deepsjeng | 77.2 | 12.3 | 47.5 | 62.2 | 42.8 | 1266.5 |
| 538.imagick | 31.5 | 20.4 | 21.5 | 97.3 | 40.9 | 306.3 |
| 541.leela | 91.4 | 14.6 | 37.5 | 73.4 | 43.5 | 1138.2 |
| 544.nab | 73.8 | 12.8 | 18.4 | 64.4 | 39.4 | 566.4 |
| 557.xz | 19.7 | 11.5 | 11.8 | 63.4 | 41.7 | 261.0 |

进一步采用上述基准程序来进行动态污点分析并评估性能开销情况, 并基于 Pin 以及 libdft 的核心部分构建分析工具[16], 作为分析实验进行对比的对象, 因为其与本文的研究和设计目标是类似的。其包括字节级别和比特级别的两种粒度的动态分析, 本文在 Windows 平台上分别构建和编译这两种情况的 Pin 工具并进行实验。比特级别的动态污点分析能够容纳更多的状态信息, 但会消耗更多的虚拟内存, 而 HyDBA 直接采用的是比特级别的分析方法。在实验过程中同样记录目标程序的执行时间, 而后计算不同分析场景下的执行速度降低情况, 并且使用的缓冲区的大小和数量与上文相同。实验结果如图 11 所示, 从中可以看到, 与基于 Pin 实现的分析工具相

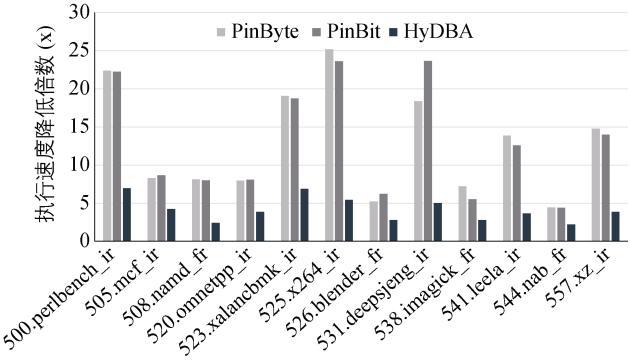


**图 11　不同分析场景下的目标程序执行性能开销**

**Figure 11　Execution performance overhead of the object program under different analysis scenarios**

比, 在采用本文方法进行分析时, 对目标测试程序产生的性能影响较低, 其原因是, 一方面与即时分析相比, 仅需记录运行时信息的解耦分析能够提高分析效率; 另一方面, 在低层次直接构建专用的分析代码有利于进一步降低目标代码的膨胀程度。得益于此, 结合图 12 可以看出, 与仅记录信息相比, HyDBA 在进行分析时所带来额外性能影响也较低, 尽管每个目标线程只是对应单个分析线程, 但它给目标线程执行所导致的迟滞影响也较低。

在分析影子内存的分配方面, 字节级别的动态污点分析一般只占用固定的内存空间, 而比特级别的分析所需要的空间因消耗大, 需跟随目标程序的动态分配而调整。如表 2 所示, 一些程序在执行过程中会分配较多的物理空间并占用大量虚拟地址, 而 HyDBA 基于物理页面分配分析状态存储空间, 可以减小对原程序虚拟地址空间的影响。有些程序在执行过程中会及时释放内存空间, 而其对应的状态存储空间则可以被回收复用, 在实验时 HyDBA 为分析状态存储预分配 512MB 的保留内存, 优先从中进行分配, 当超出限额时则尝试从回收的空间中去分配。如表 2 所示, 505.mcf 在执行过程中及时释放较多的内存, 而后回收的状态存储空间得以复用, 并且几乎没有导致分析线程出现访问冲突; 而 557.xz 则实际分配数量已超过保留数量, 尽管其部分内存得到复用, 依然存在较多冲突, 但分析框架能够正确处理该情况; 而 531.deepsjeng 在执行过程中一直分配内存而不会释放, 因此需要持续消耗大量内存空间; 其它程序由于执行时实际分配使用的内存较小而未超过保留限额。基于应用层的分析框架只能捕获虚拟地址的分配情况, 而 HyDBA 可以在内核层面根据目标程序的实际物理页面使用情况来分配状态存储空间, 可以适当降低内存资源的开销。此外, 如表 2 所示, 基准测试程序的执行代码页和指令数相对较小, 而 HyDBA 在构建执行代码、分析代码以及新的页表项等方面消耗的内存资源也比较有限。

在上文所述实验中, HyDBA 采用了 2 个大小为 1MB 的缓冲区来记录运行时信息, 事实上, 在解耦分析的过程中, 缓冲区的数量确实会对分析性能产生一定的影响[24]。但从上文实验结果来看, 与单纯信息记录相比, 分析过程对目标程序运行所施加的影响比较有限。为进一步验证该情况, 首先选择不同的缓冲区数量来进行对比实验。程序执行速度降低情况如图 12 所示, 其中, HyDBA_1 采用的缓冲区参数同上, HyDBA_2 使用了 4 个 1MB 大小缓冲区, 而 HyDBA_3 则使用了 6 个大小相同的缓冲区。从实验结果看, 缓冲区大小变化对分析性能的影响有限, 总体来说, 采用越多的大容量缓冲区越有利, 但内存分配和缓存使用也会产生一定的影响, 同时在实际应用时也要考虑到系统资源的开销。

**表 2 目标程序分析时资源占用情况统计**

**Table 2 Statistics of resource occupation during object program analysis**

| 程序名 | ECP (MB) | TAPP | RSP | CSP | APA (MB) | MEC (MB) | MAC (MB) |
|---|---|---|---|---|---|---|---|
| 500.perlbench | 2.4 | 33905 | 0 | 0 | 1.1 | 3.9 | 2.8 |
| 505.mcf | 1.4 | 170537 | 39466 | 1 | 2.7 | 1.3 | 0.9 |
| 508.namd | 1.6 | 40565 | 0 | 0 | 1.7 | 1.8 | 1.3 |
| 520.omnetpp | 2.4 | 42384 | 0 | 0 | 1.8 | 4.4 | 2.9 |
| 523.xalancbmk | 2.8 | 59208 | 0 | 0 | 1.9 | 4.4 | 2.8 |
| 525.x264 | 1.7 | 40057 | 0 | 0 | 1.8 | 2.4 | 1.7 |
| 526.blender | 6.1 | 32922 | 0 | 0 | 1.8 | 7.1 | 4.6 |
| 531.deepsjeng | 1.5 | 179229 | 0 | 0 | 2.8 | 1.5 | 1.1 |
| 538.imagick | 2.1 | 4689 | 0 | 0 | 1.5 | 2.3 | 1.6 |
| 541.leela | 1.5 | 4560 | 0 | 0 | 1.4 | 1.7 | 1.2 |
| 544.nab | 1.5 | 5062 | 0 | 0 | 1.4 | 1.5 | 1.1 |
| 557.xz | 1.4 | 225048 | 699 | 611 | 2.9 | 1.4 | 1.0 |

(注: ECP(Executable code pages)表示执行代码页数; MVS(Maximum occupied virtual memory space)表示最大占用虚拟空间; TAPP(Total allocated physical pages)表示总共分配的物理页数量; RSP(Reused state pages)表示复用的分析状态页数量; APA(Allocated pages in analysis mode)表示分析模式下分配的页表页面数; CSP(Conflicting state pages in analysis)表示分析时的冲突状态页数量; MEC(Memory usage of execution code)表示目标执行代码的内存占用; MAC(Memory usage of analysis code)表示分析代码的内存占用)

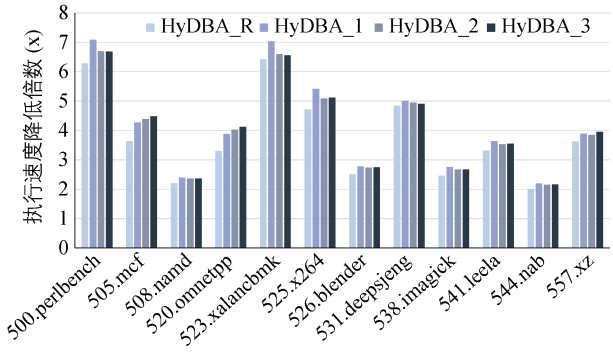


**图 12 采用不同 HyDBA 分析方案的程序执行降低情况**

**Figure 12 Program execution reduction with different HyDBA analysis schemes**

在实际执行过程中, 多数情况下缓冲区的切换操作是因缓冲区写满而引起, 而因线程切换所引起的只占不到 1%。而当采用 2 个缓冲区时, 目标线程在被调度时多数情况下未读写缓冲区, 意味着线程事实上处于等待状态, 在一定程度上影响执行效率。

由于分析代码执行效率较高, 实验中所采用的参数均适合实际场景下的分析, 而设置缓冲区数量大于 2 个则更能抵消同步造成的影响。

本文更进一步比较不同缓冲区大小的性能开销情况, 统一使用 4 个缓冲区, 并且设置 3 种不同的大小, 实验结果如图 13 所示, 不同情况对目标程序执行的影响均比较有限, 而当缓冲区大小降低时, 因缓冲区写满而引起的访问异常数量也成倍增加, 但因为能够直接在内核中较快的完成处理, 这些量级的异常数也未对性能产生显著影响。

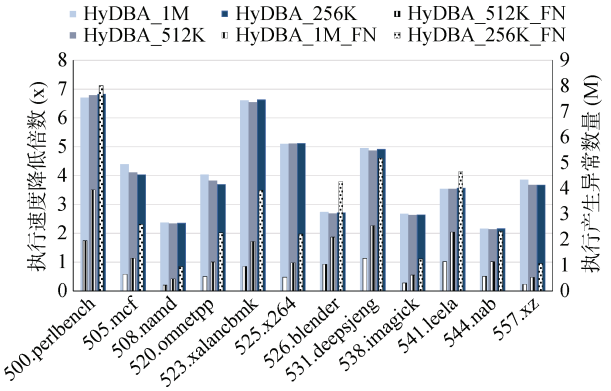


**图 13 不同缓冲区大小下分析性能的比较**

**Figure 13 Comparison of analysis performance under different buffer sizes**

## 4.2 功能测试

在进行性能测试的基础上, 进一步评估利用分析框架对实际程序进行分析的可行性, 以及验证进行动态数据流分析的效果。首先采用 Windows 平台下常用的工具程序进行实验, 涉及文件压缩程序 7z(18.05), 文件上传下载程序 curl(7.77)、aria2(1.35)、pscp(0.71)等, 具体如图 14 所示。在实验时, 框架对其文件读写和下载过程进行污点跟踪分析, 所操作文件的大小为 1MB。在内核中对文件操作函数 NtReadFile 和 NtWriteFile, 网络通信函数 NtDeviceIoControlFile 进行拦截, 并将其作为污点初始化和检测位置。例如: 在进行文件下载过程的跟踪分析时, 当接收到网络数据时, 将一定大小的内存区域设置为污染状态, 并在目标文件写入时进行检查, 主要检查第一个包含目标数据的缓冲区, 从而判断是否能够对下载的数据流进行跟踪。此外, 在分析时为每个线程设置 4 个 512KB 大小的记录缓冲区, 原则和目标与上节实验类似。实验结果如图 14 和表 3 所示。

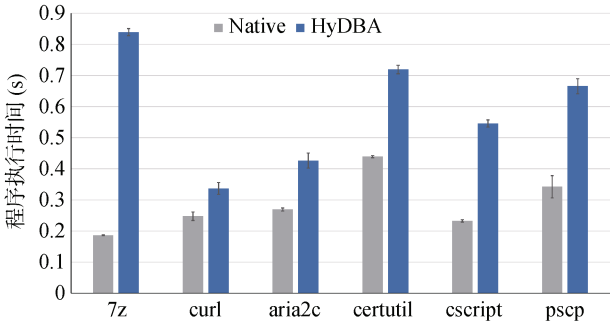


**图 14 实际环境下的程序分析执行时间**

**Figure 14 Program analysis execution time in real environment**

**表 3 程序分析过程中的详细执行信息统计**

**Table 3 Detailed execution information during program analysis**

| 程序名 | 线程 | 模块 | 基本块(k) | 缓冲切换(k) | MEC/MAC(MB) | Taint source (bytes/buffers) | Taint sink (buffer size/tainted bytes) |
|---|---|---|---|---|---|---|---|
| 7z | 3 | 15 | 23.5 | 7.4 | 2.8/1.8 | 4096/1 | 57436/57202 |
| curl | 2 | 22 | 24.4 | 1.4 | 2.8/1.9 | 4096/2 | 4096/3917 |
| aria2c | 2 | 41 | 48.6 | 0.6 | 6.0/4.0 | 4096/3 | 59/12 |
| certutil | 5 | 69 | 84.7 | 1.6 | 9.9/6.7 | 4096/1 | 1069358/622699 |
| cscript | 9 | 61 | 9.9 | 1.3 | 11.6/7.6 | 4096/2 | 845/845 |
| pscp | 2 | 34 | 34.1 | 1.8 | 4.1/2.7 | 4096/1 | 4176/4096 |

图 14 分别展示了测试程序的原始执行时间以及在分析模式下的执行时间, 从中可以看出, 在对实际程序进行分析时, 分析过程对原程序执行的性能具有更低的影响, 多数程序都在 1 s 内执行完成, 而分析过程也都控制在 1 s 内, 因为与基准程序测试不同, 很多实际分析目标都不是某类指令密集型。更详细的执行信息统计如表 3 所示, 实际程序在执行时创建的线程和加载的模块数量会更多, 因此代码生成所消耗的内存也有增加, 这些未对分析框架带来较多的负面影响。实用工具对内存资源的占用相对较少, 因此执行时产生很少缓冲区切换数量。对于污点分析过程, 从结果可以看到, 在检测点处均能够检测出一定数量被污染的内存地址, 这基本与常规分析保持一致。值得注意的是, 在实验时, 对于 7z 和 cscript 程序, 其污点源和检测点的设置由不同的线程来完成, 这也展现了分析框架对线程同步问题

的处理具有较好的能力。由于程序不同的执行过程会受到系统资源和网络影响, 因此对于部分程序, 每次执行实际检测到的污染地址数会略有上下浮动, 实验结果取平均值进行展示。

上述程序都是命令行程序, 在实验时能够便于控制程序执行的开始和结束, 以准确评估各项性能指标。为进一步验证分析框架的实用性, 采用带有图形化界面的系统内置程序 Notepad 来进行实验。具体实验场景是对 Notepad 读取文件并在界面显示的过程进行分析, 输入文件大小为 256 字节, 由于记事本程序采用内存映射方式来读取文件, 因此框架对 NtMapViewOfSection 函数进行拦截, 并在其中将读取的文件内容设置为污染状态, 而后在图形控件显示函数 GDI32!ExtTextOutW 中进行数据流分析结果检测。由于 Notepad 会在 MultiByteToWideChar 中涉及查表转换操作, 具体指令为 mov ax, word ptr [ebx+eax*2], 其中 eax 中包含文件内容, 之前的分析引擎不能处理该情况, 在实验时框架对其进行特殊处理。实验结果显示目标程序执行和分析过程都很快, 平均在 1s 左右, 分析框架能较为迅速地完成对程序的拦截和插桩。实验统计信息如表 4 所示, 与上文类似, 只展示了检测点处的第一个的缓冲区检测结果, 事实上对于 256 字节的显示会存在换行, 因此后面还有两个大小分别为 216 和 80 的缓冲区中可以检测到相应污染地址, 此外对内存中存储的转换后的宽字符的高 8 字节未能进行标识。

**表 4 Notepad 分析过程执行信息统计**

**Table 4 Execution information statistics of Notepad analysis process**

| 程序名 | 线程 | 模块 | 基本块(k) | 缓冲切换(k) | MEC/MAC (MB) | Taint source (bytes/buffers) | Taint sink (buffer size/tainted bytes) |
|---|---|---|---|---|---|---|---|
| Notepad | 8 | 44 | 97.4 | 7.4 | 11.5/7.6 | 256/1 | 216/108 |

最后利用框架对真实恶意代码 Trickbot 进行分析测试, 为避免其他影响, 实验分析目标为某样本中去除壳之后的核心代码。对其从本地读取文件并通过 HTTPS 协议发送到网络的场景进行数据流跟踪分析, 与上文类似对相关系统函数进行拦截, 同时在本地构建了用于通信的虚拟服务器。目标代码在执行时读取和发送的数据为 100 字节, 经过传递和加密后, 在大小为 275 的缓冲区中检测到 49 字节的数据被污染, 验证了对实际恶意代码进行自动化分析的有效性。目标代码在执行过程中创建了 10 多个线程, 也是由不同的线程完成文件读取和数据发送过程, 分析框架也较好的处理该情况。此外, 整个分析过程较快且未出现明显异常情况。

## 5 讨论

本文工作进一步减小在进行二进制插桩分析时所生成的执行代码的规模, 降低对目标程序执行的影响, 并提高分析效率。提供了一种基础性的分析框架, 也更加适合于有定制化分析需求, 并且需直接修改底层分析代码的情形。

所提出的分析框架进一步融合系统内核和硬件虚拟化技术, 可以更有效的利用内存资源, 减小对目标进程虚拟空间的占用。框架能够直接进行基于比特级别的污点分析, 克服传统细粒度分析时, 可能难以在目标进程空间连续和大量分配虚拟地址的问题。

本文在 Windows 平台上构建原型系统, 需面对非开源系统的挑战, 验证分析可行性并提高其在实际分析场景的可应用性, 但也存在原型框架实现的完备性问题, 如: 可能未对目标程序的所有操作进行拦截、未覆盖换页操作和页面文件访问的所有情况等, 需对操作系统机制进行深度研究后不断完善。

分析框架融合内核层面, 在易用性方面会有提高, 但在分析隐蔽性方面会有一定程度的不足。框架通过多种策略来应对解耦分析时的分析同步问题, 在实际分析时取得一定的效果, 但在处理指令级同步问题时可能会存在问题, 需要通过进一步的实验分析进行优化。此外, 与成熟的分析方法相比, 框架在代码链接和布局优化等方面仍存在不足, 在用户接口以及内存动态管理方面还需改进, 框架有待不断开发完善并扩展至 64 位平台。

## 6 总结

本文深度融合系统内核和硬件虚拟化技术, 提出一种新的二进制程序动态分析方法, 可对二进制程序进行细粒度数据流分析, 基于解耦分析并通过多种策略对其进行优化, 进一步提高了分析性能和效果, 并具有良好的可部署性。本文构建了实用的分析框架原型, 通过基准测试程序以及实际应用程序来对分析框架的性能和功能进行了验证, 后续将在易用性和扩展性等方面进行更深入的研究与实验。

## 参考文献

[1] Zhang J, Zhang C, Xuan J F, et al. Recent Progress in Program Analysis[J]. *Journal of Software*, 2019, 30(1): 80-109.
(张健, 张超, 玄跻峰, 等. 程序分析研究进展[J]. *软件学报*,

2019, 30(1): 80-109.)

[2] Zhang-Kennedy L, Assal H, Rocheleau J, et al. The Aftermath of a Crypto-Ransomware Attack at a Large Academic Institution[C]. *The 27th USENIX Conference on Security Symposium*, 2018: 1061-1078.

[3] Ahmad A, Webb J, Desouza K C, et al. Strategically-Motivated Advanced Persistent Threat: Definition, Process, Tactics and a Disinformation Model of Counterattack[J]. *Computers & Security*, 2019, 86: 402-418.

[4] Calleja A, Tapiador J, Caballero J. The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development[J]. *IEEE Transactions on Information Forensics and Security*, 2019, 14(12): 3175-3190.

[5] Li Z Y, Chen Q A, Xiong C L, et al. Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for PowerShell Scripts[C]. *The 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019: 1831-1847.

[6] Schumilo S, Aschermann C, Gawlik R, et al. KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels[C]. *The 26th USENIX Conference on Security Symposium*, 2017: 167-182.

[7] Cheng B L, Ming J, Fu J M, et al. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 395-411.

[8] Zhu E Z, Wen P, Ni K Q, et al. Implementation of an Effective Dynamic Concolic Execution Framework for Analyzing Binary Programs[J]. *Computers and Security*, 2019, 86(C): 1-27.

[9] He J X, Ivanov P, Tsankov P, et al. Debin: Predicting Debug Information in Stripped Binaries[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 1667-1680.

[10] Lengyel T K, Maresca S, Payne B D, et al. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System[C]. *The 30th Annual Computer Security Applications Conference*, 2014: 386-395.

[11] Bauman E, Ayoade G, Lin Z Q. A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions[J]. *ACM Computing Surveys*, 48(1): 10.

[12] Frida. https://frida.re/. Oct, 2021.

[13] D'Elia D C, Coppa E, Nicchi S, et al. SoK: Using Dynamic Binary Instrumentation for Security (and how you may Get Caught Red Handed)[C]. *The 2019 ACM Asia Conference on Computer and Communications Security*, 2019: 15-27.

[14] Clause J, Li W C, Orso A. Dytan: A Generic Dynamic Taint Analysis Framework[C]. *The 2007 international symposium on Software testing and analysis*, 2007: 196-206.

[15] Yin H, Song D, Egele M, et al. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis[C]. *The 14th ACM conference on Computer and communications security*, 2007: 116-127.

[16] Kemerlis V P, Portokalidis G, Jee K, et al. Libdft[J]. *ACM SIGPLAN Notices*, 2012, 47(7): 121-132.

[17] Basu K, Krishnamurthy P, Khorrami F, et al. A Theoretical Study of Hardware Performance Counters-Based Malware Detection[J]. *IEEE Transactions on Information Forensics and Security*, 2019, 15: 512-525.

[18] Jee K, Kemerlis V P, Keromytis A D, et al. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking[C]. *The 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, 2013: 235-246.

[19] Wang X J, Ma R, Dou B W, et al. OFFDTAN: A New Approach of Offline Dynamic Taint Analysis for Binaries[J]. *Security and Communication Networks*, 2018: 7693861.

[20] Pan J Y, Zhuang Y, Sun B L. BAHK: Flexible Automated Binary Analysis Method with the Assistance of Hardware and System Kernel[J]. *Security and Communication Networks*, 2020: 8702017.

[21] TinyInst. https://github.com/googleprojectzero/TinyInst. Oct. 2021.

[22] Henderson A, Yan L K, Hu X C, et al. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform[J]. *IEEE Transactions on Software Engineering*, 2017, 43(2): 164-184.

[23] Banerjee S, Devecsery D, Chen P M, et al. Iodine: Fast Dynamic Taint Tracking Using Rollback-Free Optimistic Hybrid Analysis[C]. *2019 IEEE Symposium on Security and Privacy*, 2019: 490-504.

[24] Ming J, Wu D H, Xiao G Y, et al. TaintPipe: Pipelined Symbolic Taint Analysis[C]. *The 24th USENIX Conference on Security Symposium*, 2015: 65-80.

[25] Black P, Gondal I, Layton R. A Survey of Similarities in Banking Malware Behaviours[J]. *Computers and Security*, 2018, 77(C): 756-772.

[26] Pan J Y, Yi Z, Zhao X J, et al. Lightweight and Efficient Hypervisor-Based Dynamic Binary Instrumentation and Analysis Method[J]. *IEEE Access*, 2020, 8: 164593-164610.

[27] Hong J Q, Ding X H. A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow across User-Kernel Spaces[C]. *2021 IEEE Symposium on Security and Privacy*, 2021: 1902-1918.

[28] Zhang X L, Wang X Y, Slavin R, et al. ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis[C]. *2021 IEEE Symposium on Security and Privacy*, 2021: 796-812.

[29] Luk C K, Cohn R, Muth R, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation[C]. *The 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005: 190-200.

[30] Bruening D, Zhao Q, Amarasinghe S. Transparent Dynamic Instrumentation[C]. *The 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012: 133-144.

[31] Nethercote N, Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation[C]. *The 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007: 89-100.

[32] QBDI. https://qbdi.quarkslab.com/. Sept. 2021.

[33] Engelke A, Schulz M. Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation[C]. *The 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020: 172-184.

[34] Bungale P P, Luk C K. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation[C]. *The 3rd international conference on Virtual execution environments*, 2007: 137-147.

[35] Zeng J Y, Fu Y C, Lin Z Q. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework[C]. *The 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015: 147-160.

[36] Dinaburg A, Royal P, Sharif M, et al. Ether: Malware Analysis via Hardware Virtualization Extensions[C]. *The 15th ACM conference on Computer and communications security*, 2008: 51-62.

[37] Deng Z, Zhang X Y, Xu D Y. SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization[C]. *The 29th Annual Computer Security Applications Conference*, 2013: 289-298.

[38] Zhang F W, Leach K, Stavrou A, et al. Using Hardware Features for Increased Debugging Transparency[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 55-69.

[39] Das S, Werner J, Antonakakis M, et al. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security[C]. *2019 IEEE Symposium on Security and Privacy*, 2019: 20-38.

[40] Ming J, Wu D H, Wang J, et al. StraightTaint: Decoupled Offline Symbolic Taint Analysis[C]. *The 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016: 308-319.

[41] Galea J, Kroening D. The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation[C]. *The 15th ACM Asia Conference on Computer and Communications Security*, 2020: 622-636.

[42] Dolan-Gavitt B, Hodosh J, Hulin P, et al. Repeatable Reverse Engineering with PANDA[C]. *The 5th Program Protection and Reverse Engineering Workshop*, 2015: 1-11.

[43] Cao M C, Hou X T, Wang T, et al. Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay[C]. *The 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019: 1883-1897.

[44] HyDBA. https://github.com/njptdev/HyDBA.

**潘家晔** 于 2020 年在南京航空航天大学计算机科学与技术专业获得博士学位。现任南京邮电大学现代邮政学院讲师。研究领域为信息安全。研究兴趣包括: 系统与软件安全、网络安全、恶意代码分析等。Email: panjy@njupt.edu.cn

**沙乐天** 于 2015 年在武汉大学信息安全专业获得博士学位。现任南京邮电大学计算机学院副教授。研究领域为信息安全。研究兴趣包括: 物联网安全、工控安全、漏洞分析与挖掘等。Email: ltsha@njupt.edu.cn