融合 Intel PT 与动态插桩的二进制程序跟踪分析框架

潘家晔，沙乐天，鲁京，肖甫
（南京邮电大学计算机学院，江苏 南京 210023）

摘　要：针对动态二进制插桩技术在自动化分析方面得到广泛应用，但面临性能开销、易用性等问题，提出了一种软硬件结合的动态二进制程序跟踪框架，能够对程序运行过程进行跟踪和分析。该框架将 Intel PT 硬件跟踪机制与动态二进制程序插桩技术进行融合，提高了利用动态插桩框架对二进制程序进行跟踪和分析的能力，并在此基础上提出了一种基于在线跟踪的二进制程序分析方法，进一步提高了分析的灵活性和性能。框架和方法的原型在 Windows 平台上实现，分别通过基准程序和真实应用程序进行实验，结果验证了所提方法的有效性和实用价值。

关键词：程序分析；动态跟踪；二进制插桩；硬件机制
中图分类号：TP319.0
文献标志码：A
DOI：10.11959/j.issn.1000–436x.2025228

Binary program tracing and analysis framework by integrating Intel PT and dynamic instrumentation

PAN Jiaye, SHA Letian, LU Jing, XIAO Fu
School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China

Abstract: To address the problem that while dynamic binary instrumentation frameworks are widely used for automated analysis, they face challenges such as performance overhead and usability issues, a hybrid hardware-software framework was proposed for dynamic binary program tracing, enabling comprehensive monitoring and analysis of program execution. The Intel processor trace (PT) and the dynamic binary instrumentation framework were integrated, thereby enhancing its capability for tracing and analysis of binary programs. Building upon this foundation, an online tracing-based binary analysis method was introduced further to improve both analytical flexibility and performance. A prototype of the proposed framework had been implemented on the Windows platform and evaluated using both benchmark suites and real-world applications. The experimental results confirm the effectiveness and practical value of the proposed analysis method.

Keywords: program analysis, dynamic tracing, binary instrumentation, hardware mechanism

0　引言

随着互联网与通信网络的深度融合，网络安全形势也面临新的挑战。一方面，各类高级恶意代码层出不穷，其传播途径呈现多样化，常利用网络协议漏洞、供应链等渠道，在通信链路中隐蔽渗透，干扰信息系统的正常运行，安全研究人员需要不断开展深入分析和溯源。另一方面，软件代码的复杂程度不断提高，围绕软件漏洞开展的攻防研究也在

收稿日期：2025–10–16；**修回日期：**2025–12–15
通信作者：沙乐天，ltsha@njupt.edu.cn
基金项目：国家自然科学基金资助项目（No.62302193）；江苏省 2024 前沿技术研发计划基金资助项目（No.BF2024071）
Foundation Items: The National Natural Science Foundation of China (No.62302193), The 2024 Frontier Technology Research and Development Program of Jiangsu (No.BF2024071)

持续，亟待构建高效、可扩展的自动化分析方法。而这些分析和研究多数情况下面对二进制程序，其中静态分析和动态分析是常用的分析技术[1-2]。动态分析技术可以结合自动化分析方法，减少无关代码干扰，提高分析效率，因此得到广泛使用[3]。例如，在自动化恶意代码分析时，经常利用沙箱来执行恶意代码，提取行为特征；利用模糊测试方法进行漏洞挖掘时对程序执行路径进行跟踪，以提高测试代码覆盖率。

二进制程序跟踪技术通常可以利用硬件机制或者软件代码实现，典型的硬件机制，如 Intel 分支跟踪存储（BTS, branch trace store）、性能监控计数器（PMC, performance monitoring counter）、处理器跟踪（PT, processor trace）等，已成为主流处理器内置的功能，其优点是速度快、隐蔽性高，而缺点是缺乏足够的程序语义信息。很多研究工作基于底层硬件机制实现程序跟踪和分析，如 MALT[4]、LoopHPC[5]。其中，Intel PT 得到广泛应用，可以高效记录程序执行的控制流信息。较多的漏洞挖掘研究工作（如 kAFL[6]）利用 PT 机制提高覆盖率和收集性能。基于软件实现的跟踪工具通常基于动态二进制插桩（DBI，dynamic binary instrumentation）技术，在目标程序运行时，实现指令、代码块等层面的插桩和分析，能够针对目标程序进行更深入的跟踪分析。常用的动态二进制插桩工具（如 Intel Pin[7]、DynamoRIO[8]）可以方便地直接在目标系统内进行部署应用。基于虚拟机技术实现的跟踪分析框架（如 DECAF[9]、PANDA[10]），其分析范围能够覆盖整个系统，并支持多种平台架构，但同时也增加了部署的复杂性和虚拟化开销。此外，一些研究工作（如 PalanTir[11]）引入硬件机制进行程序混合跟踪和分析，以实现更高效的网络攻击溯源和动态污点分析[12]，其在利用硬件增强跟踪的基础上进一步结合静态程序分析、图分析等方法实现综合分析系统[13]，但在分别进行二进制插桩和 PT 时仍采用传统方案。基于动态二进制插桩技术实现程序跟踪能够记录更多的语义信息，并支持后续更深入的离线分析，降低对目标程序执行的影响[14]。同时，在线插桩也会给目标程序带来较大的性能开销，影响使用体验，甚至会导致程序执行因超时而偏离预期结果[15]。纯硬件的 PT 机制无法感知程序在运行时动态分配和执行的代码，而仅基于控制流信息和原程序模块难以还原程序实际的执行情况。

为此，本文提出一种软硬件结合的动态二进制程序跟踪框架，将硬件跟踪机制与动态插桩技术进行融合，提高利用动态插桩框架对程序进行跟踪和分析的能力。通过引入硬件机制辅助对控制流等信息进行跟踪，提高程序跟踪的性能，并简化动态指令插桩的复杂度，同时基于动态插桩框架获取程序语义信息，扩展了硬件跟踪技术的应用场景。本文在 Windows 平台上基于 DynamoRIO 和 Intel PT 实现了跟踪框架的原型 DRIPT，并在此基础上提出一种基于在线跟踪的二进制程序分析方法，将运行时信息提取、跟踪数据解析以及目标代码分析进行结合，进一步提高了针对目标程序进行动态插桩分析的性能和灵活性。本文方法能够优化基于动态插桩工具的程序跟踪和分析能力，也为更复杂和深入的分析提供基础支撑。与 HardTaint[12]、PANDA[10]等工作相比，本文面向用户应用程序层面的动态跟踪，聚焦于融合硬件跟踪机制对传统二进制插桩框架进行拓展，并实现轻量级的程序动态跟踪分析框架，可便捷地在目标系统内进行部署和应用。

本文主要贡献如下。

1) 本文提出一种软硬件结合的动态二进制程序跟踪框架，将硬件跟踪机制与动态插桩技术进行融合，提高了对二进制程序进行跟踪分析的能力。

2) 在此基础上，本文提出一种基于在线跟踪的二进制程序分析方法，进一步提高了利用动态插桩工具对程序进行分析的性能。

3) 本文在 Windows 平台上基于 DynamoRIO 和 Intel PT 实现了原型框架，通过真实场景的程序进行测试，验证了跟踪和分析方法的有效性。

1　方法描述

本文提出的框架和方法直接在目标系统平台上实施，并对目标程序开展跟踪分析。在应用时需要目标分析环境的处理器支持 PT 功能，并于目标系统内部署 DBI 工具，在此基础上融合两者实现对目标程序的跟踪和分析。框架总体工作流程如图 1 所示，主要包括构建硬件跟踪管理程序、增强 DBI 框架、启动跟踪程序以及执行分析代码等环节。

首先，在目标系统中通过构建和部署驱动程序来对硬件跟踪功能进行管理，引入线程和模块

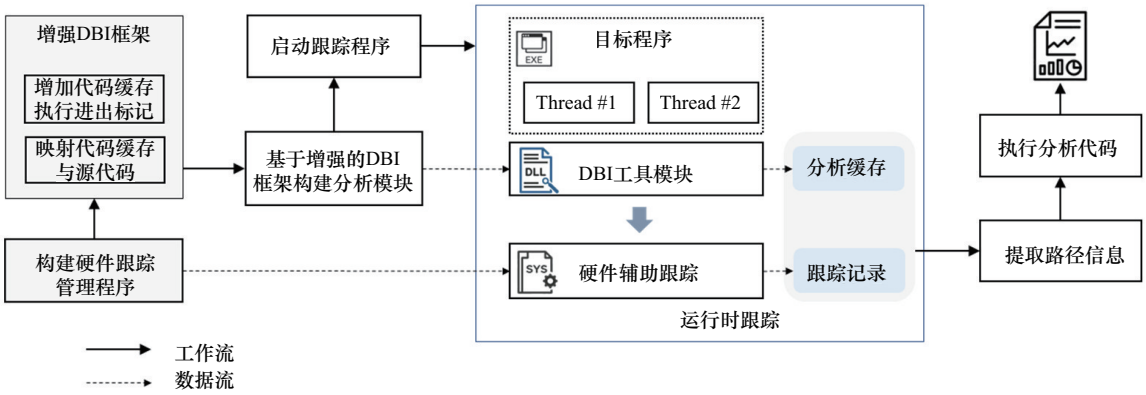


图1　框架总体工作流程

过滤机制，并优化线程级别的运行时跟踪流程，实现与应用层DBI框架的协同工作。其次，对传统DBI框架进行修改和功能增强，提高软硬件协同跟踪下的程序语义的获取能力。当程序在插桩后运行时，其执行的代码已非源代码，需在DBI框架中构建实际执行代码与源代码的映射关系。在此基础上，用户可以基于增强的DBI框架开发所需要进行跟踪和分析的模块，而后加载目标程序完成后续的跟踪分析功能。在程序运行过程中，DBI工具和PT分别记录所需的运行时数据。在跟踪结束后，基于所记录的数据提取和输出程序实际执行的路径信息，并可进一步执行基于动态插桩预先构建的分析代码，从而完成程序的跟踪和分析过程。

1.1　线程级别跟踪

为提高跟踪性能，在利用动态二进制插桩框架对程序进行跟踪分析时按线程分别处理，而操作系统内置硬件管理模块以及现有研究工作暂未直接支持线程级别的跟踪，并且无法与现有DBI框架进行融合。为此本文对基于PT的硬件跟踪功能进行优化，实现线程级别的高效跟踪，通过线程调度跟踪和直接内存映射提高对程序运行数据的存取速度。与PT硬件跟踪机制相关的指令属于特权指令，需要通过内核驱动模块实现跟踪功能的管理，并实现与DBI框架的协同工作。在执行跟踪时，驱动模块为目标程序每个线程创建用于保存跟踪记录的缓冲区，并创建对应的处理线程对缓冲区和数据进行管理。线程级别跟踪如图2所示，驱动模块采用PT

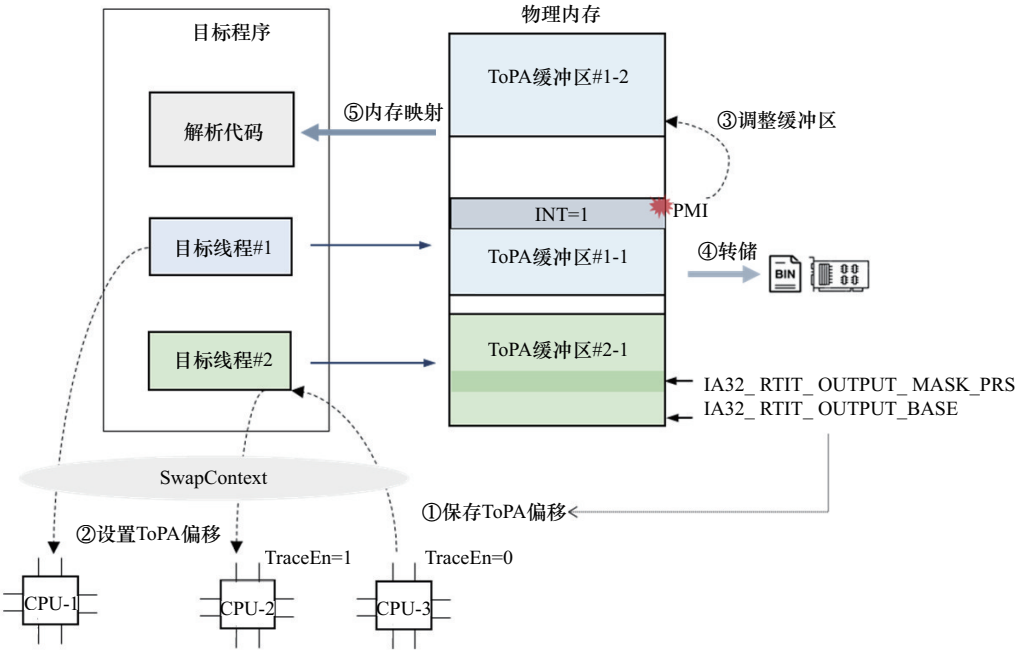


图2　线程级别跟踪

机制支持的物理地址表（ToPA, table of physical addresses）模式来存储记录数据，即将记录缓冲区映射为多个连续的物理内存区域，每个线程有独立的存储区。而后为当前线程所关联的处理器开启 PT 功能，并基于 DBI 框架所加载的目标进程对整个跟踪过程进行过滤。当目标线程退出时，则停止其关联处理器的硬件跟踪功能。此外，在程序跟踪过程中针对下面 2 个环节进行优化。

1) 线程跟踪和调度。在程序执行过程中，操作系统本身的线程调度方式将导致目标线程在多个处理器之间切换运行，但硬件跟踪功能与处理器相关。因此，本文利用内核驱动模块直接监控系统线程切换函数，并即时更新当前处理器 PT 功能所对应的线程及记录缓冲区，保证记录过程准确性的同时提高缓冲区切换的效率。在此过程中需维持线程与处理器的对应关系，以及保持线程上下文的一致性。如图 2 中①和②所示，当目标线程执行从处理器离开时，停止当前处理器的跟踪功能，并保存该线程缓冲区中的记录指针，主要涉及与 PT 输出相关的 2 个寄存器。当目标线程被调度至新处理器上运行时，则开启该处理器的跟踪功能，并恢复之前所保存的记录缓冲区指针，从而继续记录该线程相关的控制流信息。

2) 缓冲区更新和访问。在线程当前使用的内存输出缓冲区写满时需进行数据转储，而 PT 机制提供了记录缓冲区写满的监测方法，即通过在 ToPA 的表项中设置中断标记，当跟踪数据写入带有该标记的内存位置后将触发性能监控中断。为此，驱动模块注册该中断处理程序，并在其中利用系统支持的延迟过程和异步过程调用完成当前线程所对应的记录缓冲区的调整。如图 2 中③所示，即将线程对应的记录缓冲区设置为下一个空闲区域。在实际中，如果物理内存足够大，则可以将记录数据全部保存在内存中，以提高访问效率；否则将写满的缓冲区放入转储队列中，由其他工作线程保存到磁盘，如图 2 中④所示。另外，在中断处理程序中，处理器可能处于任意线程上下文中，驱动模块需提前将目标线程控制块地址保存至 ToPA 中的预留表项中，从而能够获取与目标线程相关的输出缓冲区信息。最后，当目标程序跟踪完成后，驱动模块将缓冲区对应的物理内存或文件映射至用户空间，解析代码则可以直接访问记录的跟踪数据，如图 2 中⑤所示。

1.2　DBI 框架增强

传统动态二进制插桩框架工作在用户态，为了与管理 PT 功能的内核模块进行协同，需要对其进行适当修改增强。DBI 插桩框架的基本工作过程如图 3 所示，DBI 框架加载目标程序运行后对其进行动态插桩，即将目标应用程序原始指令的执行转移到新创建的代码缓存中，并在代码缓存中对程序原始指令进行修改，包括替换原有指令或增加额外功能等。当 DBI 框架接管程序运行后，程序源代码实际都在代码缓存空间中执行，此时 PT 所记录的内

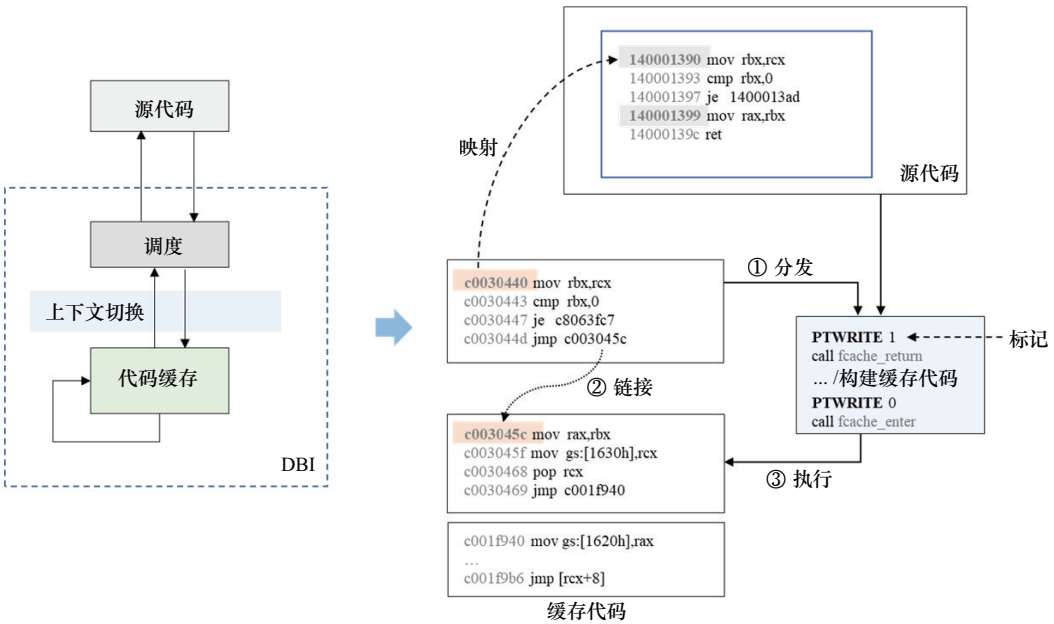


图 3　DBI 插桩框架的基本工作过程

存地址信息实际为代码缓存地址，而不是程序代码原来的地址。因此，本文对传统DBI框架的功能进行增强，以更好地支持混合跟踪和分析方案，从而能够访问记录缓冲区并丰富跟踪记录的语义信息。

在对二进制程序进行动态插桩的过程中，框架调度模块每次将目标程序的一个动态基本块复制到代码缓存中。在即将跳转到下一个目标代码块时，框架首先进行指令拷贝、代码链接等分发过程。如果该代码块在缓存中已经存在，则直接跳转到已缓存的代码去执行，并将下一个目标代码块缓存地址链接到当前代码块的缓存；否则执行将进入分发调度模块，为目标代码生成对应的代码缓存块，而后程序将跳转至该缓存代码执行，同时完成缓存代码块链接，以提高后续执行的效率。例如，在图3中140001390开始的基本块对应缓存地址c0030440，第一次执行路径如图3中①所示，即进入分发调度模块，生成140001399对应的目标缓存块c003045c，而后程序将跳转至该缓存代码执行。此外，调度模块还将缓存c0030440处的代码直接链接到新生成的缓存代码c003045c，如图3中②所示。

尽管程序实际执行的是插桩框架新生成的缓存代码，但缓存代码包含的控制流信息与源代码基本是一致的，因此PT所记录的缓存代码执行控制流信息同样能够反映目标程序的执行情况。一方面，针对条件跳转，PT通过一个比特来记录跳转状态，对应的记录类型为“TNT”，如图3所示，c0030447处指令的执行结果为不跳转，其同样反映在原指令地址140001397处的代码结构中；另一方面，针对间接跳转，PT会直接记录程序执行的目标地址信息，对应的记录类型为“TIP”，在动态插桩环境下实际上为代码缓存地址。例如，图3中以140001399为起始地址的代码块包括RET指令，在其对应的缓存代码c003045c处RET指令则已被重写为一个查表过程，并最终通过间接跳转指令完成执行转移。PT硬件同样能够记录间接跳转时读取的代码缓存地址信息。因此，在构建程序源代码与缓存代码映射关系的基础上，本文通过PT记录中包含的控制流信息和内存地址就能够还原程序源代码的执行路径，主要从以下3个方面对DBI框架进行增强。

1) 代码地址映射。首先，在DBI框架生成的代码缓存与源代码间建立映射关系，即通过代码缓存地址快速定位源代码块，反之亦然。例如，图3中源代码地址140001390与缓存地址c0030440存在对应关系。动态插桩框架在多数情况下基于动态基本块来逐步生成对应的缓存代码，用户可以利用插桩框架提供的开发接口注册对应的回调函数，并在其中对基本块中的指令进行解析和插桩。为此，当框架拦截到程序基本块的执行时，对基本块中的指令进行解析，并为每个基本块额外创建一个解析信息摘要，其中包括源代码地址、跳转目标地址、预先构建的分析代码以及其他标志信息，同时建立源代码地址与解析信息摘要所在地址的映射关系，以及代码缓存地址与解析信息摘要的映射关系。如图4所示，源代码地址7ffed48dda25对应的缓存块地址为c809c79c，解析信息摘要地址为c007e7e0。因此，在建立映射关系后，可以分别通过源代码和缓存代码地址快速定位代码块对应的解析信息摘要，在分析时可以获取更丰富的源代码结构和语义信息。

2) 缓存代码标记。当目标程序的执行进入DBI框架分发模块时会产生大量控制流信息，而它们与源代码的执行并没有直接关系，需对这部分代码的执行过程进行标记，在后续解析跟踪数据时可以跳过那些与之相关的执行记录。因此，本文利用PT机制所支持的PTWRITE指令在跟踪记录中插入相关标记，该指令为非特权指令，可以读取源操作数中的数据并将其发送到PT硬件，生成类型为“PTW”的记录。本文分别在目标程序的执行进入缓存代码和离开缓存代码处添加该指令进行标记，如图3所示。由于分发模块只是在DBI框架接管目标代码执行时被调用，当代码块之间进行链接后大多数时间将在代码缓存中进行跳转，少量PTWRITE指令执行不会对PT的性能开销产生较大影响，但其可以在跟踪数据中对源代码功能无关的记录进行过滤，提高后续记录解析的效率。

3) 热代码块处理。对于一些高频率执行的基本块代码，DBI框架会将其重新组合放在相同的代码缓存空间中，称为热代码块或TRACE。在出现TRACE的情况下，程序部分源代码块会生成2个以上对应缓存块。在程序执行完成后，仅根据PT记录难以得知实际执行的缓存代码块，因为对于条件跳转指令，PT仅记录跳转状态而非目标地址。例如，图4中原程序地址为7ffed48dda25和

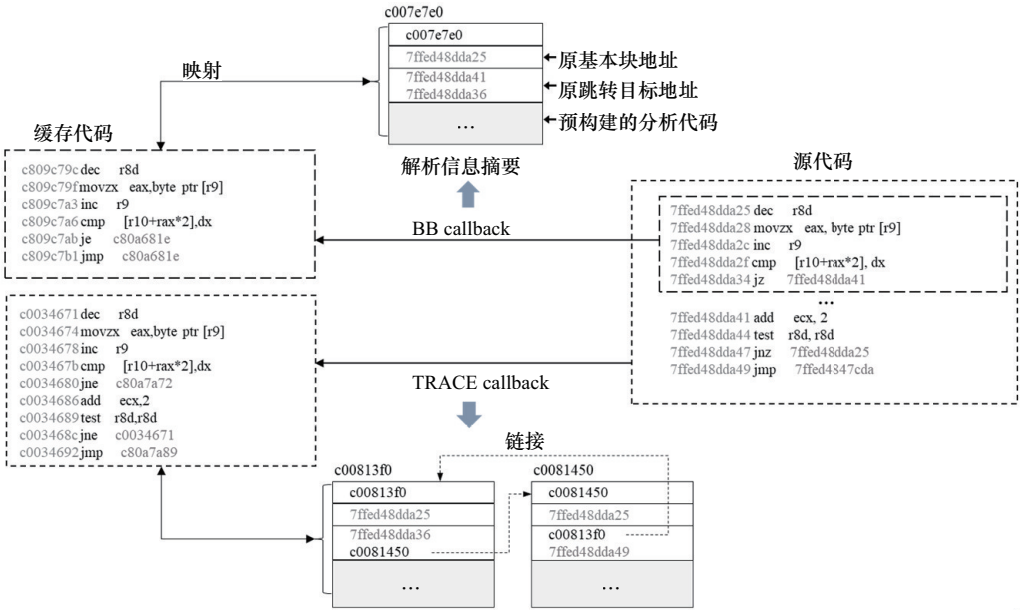


图4　基本块生成和处理过程

7ffed48dda41 的 2 个代码块在缓存地址 c0034671 处是连续存储的，并且基本块 7ffed48dda25 对应 2 个地址分别为 c809c79c 和 c0034671 的缓存块。另外，在 TRACE 相关的缓存中，部分源代码块对应的分支指令可能会因缓存代码块重新组合而发生变化，导致缓存代码执行时产生的控制流信息与源代码不一致。例如，图 4 中 7ffed48dda25 处代码块的最后采用 jz 指令跳转到下一个代码块执行，但在缓存中对应的却是 jne 指令，在跳转条件不满足时进入下一个代码缓存处执行。这些情况都会给基于 PT 记录获取程序实际执行路径带来挑战。一个解决方案是通过 DBI 框架提供的控制参数关闭 TRACE 功能，可能会对一些程序的运行增加额外的性能开销，并且有些框架也未提供该功能参数。另一个解决方案则是对 TRACE 相关代码块进行特别处理。本文可以基于 DBI 框架提供的 TRACE 生成事件回调接口添加处理函数，并在其中提取 TRACE 缓存中包含的基本块信息，为每个基本块重新生成对应的解析信息摘要，并将这些解析信息摘要在内存中链接在一起，如图 4 所示。而对于缓存中代码块重新组合并改变原来跳转关系的情况，则在 TRACE 对应的缓存代码开头增加 PTWRITE 指令，以在进行 PT 时插入第一个基本块的信息摘要地址（如图 4 中的 c00813f0），这样在后续解析 PT 记录时，可以直接获取实际执行的缓存代码所对应的解析信息。此外，该方法也能够解决 DBI 框架在运行时因内存管理而复用之前代码缓存地址所产生的问题，即复用会导致在 PT 记录中一个代码缓存地址可能会对应不同的源代码地址。在建立缓存代码和源代码地址映射时，可以通过是否有冲突来检测该情况，而后在新生成的缓存代码开头同样增加上述指令。

1.3　数据处理

本文不仅可以根据 PT 数据高效获取程序执行路径，还可以利用传统 DBI 框架提供的丰富接口来对目标程序进行额外的插桩，并构建与源代码执行解耦的分析代码。当针对程序的执行过程跟踪完成后，框架将对 PT 和 DBI 记录的数据进行解析，在目标进程未退出的情况下，直接访问内存数据并进行代码分析。框架将存储 PT 记录的物理内存直接映射到用户空间地址，并结合 DBI 中记录的代码映射关系按线程来获取程序执行路径，在此基础上，进一步执行利用 DBI 工具接口预先构建的解析信息和分析代码，完成其他分析任务。

程序执行路径提取算法如算法 1 所示，在执行时逐条解析 PT 记录，通过“PTW”类型的数据包获得程序执行时进入和离开代码缓存的标记，从而跳过 DBI 框架调度模块的代码执行记录。而后算法将 PT 数据转化为程序实际执行过的源代码块序列。对于程序执行时产生的间接跳转目标，可以根据“TIP”“FUP”等类型的数据包直接获得缓存代码地址，并继续通过查找映射表获取目标代码块信息；而对于条件跳转目标，则直接根据“TNT”类

型记录并查找哈希表来获取目标代码块信息。

算法1 程序执行路径提取算法

输入 跟踪记录缓冲区trace_buffer

输出 带有解析数据的代码块集合parse_blocks

```
1) 初始化
   skip ← true
   addr_info ← ∅
   tnt_array ← []
   parse_blocks ← []
   start_addr, end_addr, idx ← 0
2) for each packet in trace_buffer do //遍历每个记录
3)   if packet.type == 'PTW' then
4)     if packet.value == 0 then//执行进入代码缓存标记
5)       skip ← false
6)     else if packet.value == 1 then//执行离开代码缓存标记
7)       skip ← true
8)     else
9)       提取packet.value中的地址信息至addr_info
10)    end if
11)  else if packet.type ∈ {'TIP','TIP.PGE','TIP.PGD','FUP'} then
12)    提取packet.value中的地址信息至addr_info
13)    if skip then
14)      continue
15)    end if
16)    根据addr_info确定需分析的地址区间[start_addr, end_addr]
17)    p ← get_parse_block(start_addr) //获取给定地址对应的解析信息
18)    while p ≠ ∅ and p.code_addr != end_addr do
19)      parse_blocks.append(p)
20)      if p.branch_type为条件跳转 then
21)        if tnt_array[idx++] == 1 then
22)          t ← get_parse_block(p.target_addr)
23)        else
24)          t ← get_parse_block(p.next_addr)
25)        end if
26)      else if p.branch_type为直接跳转 then
27)        t ← get_parse_block(p.target_addr)
28)      end if
29)      p ← t
30)    end while
31)    tnt_array.clear()
32)    idx ← 0
33)  else if packet.type == 'TNT' and not skip then
34)    提取packet.value中的条件分支执行结果至tnt_array
35)  end if
36) end for
37) return parse_blocks
```

2 实验和分析

本文基于DynamoRIO(11.2.0)的源代码进行修改和编译，实现所提出的跟踪和分析框架的原型，主要在emit.c和emit_utils_shared.c这2个源文件中添加约100行的增强代码，而后基于DynamoRIO提供的接口额外开发了分析模块，主要用于对需分析的目标程序进行插桩并记录运行时信息，包括约1 000行代码。PT数据的解析代码则基于libxdc项目进行开发，包括约1 500行代码。整个原型框架的开发环境为Visual Studio 2019，实验环境的配置为普通终端，Intel i5-12400 2.50 GHz处理器，12逻辑核心，16 GB内存，安装Windows 10(20H2)64位操作系统。实验数据为5次实验结果的平均值。

2.1 性能评估

首先评估DRIPT进行在线跟踪时对目标程序的执行影响情况，测试程序采用CPU SPEC 2017基准测试程序。分别针对不同的场景进行测试和对比，其中PT_NA表示直接在开启PT的环境下运行原始程序，DR_NA表示在DynamoRIO环境下运行原始程序但不进行额外的插桩和分析，而DR_BB则利用DynamoRIO内置的插件模块来记录程序控

制流信息，所有测试均将数据存储于内存而不输出到文件中。基准程序跟踪性能开销对比如图 5 所示，其中性能开销表示在不同测试场景中程序的执行速度，其为程序原始执行时间的倍数。

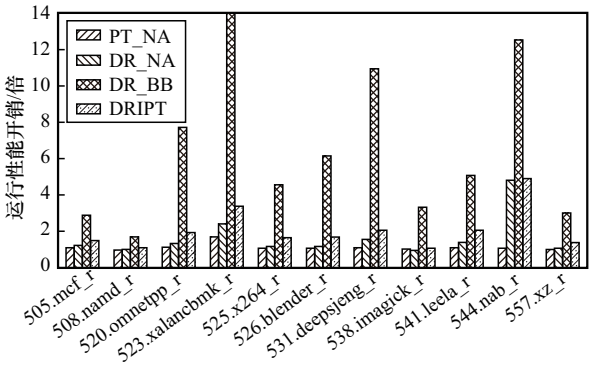


图 5　基准程序跟踪性能开销对比

从图 5 可以看出，DR_BB 普遍会带来较高的性能开销，而与之比较，引入 PT 机制进行辅助跟踪可以显著提高性能。单纯在源程序执行基础上开启 PT 功能，所带来的执行性能开销基本可以忽略。对多数程序而言，在单纯 DR_NA 环境下运行时的性能开销普遍较小，也有些会带来较高的额外性能开销，例如 544.nab_r。而 DRIPT 所带来的性能开销则在 DR_NA 的基础上有一定的提升，主要原因在于当程序在动态插桩框架下运行时，实际执行的基本块数量会变多，其中包括插桩工具自身模块代码以及上下文切换代码等，还有额外记录的“PTW”事件也会产生一定程度的性能影响。但总体性能开销平均为 2 倍，与直接利用动态插桩框架在线记录控制流相比，DRIPT 能显著降低跟踪时的程序执行性能开销。

另外，在实验时未将记录数据输出到文件中，而实际上单纯 PT 场景下所记录的数据大小范围为 1.5~132 GB，平均为 37 GB，而 DRIPT 记录的数据大小范围为 5~250 GB，平均为 103 GB。主要原因如下：在 DRIPT 运行环境下，首先程序执行的代码块数量会增加，尤其是与插桩调度模块相关的代码，其中包括大量间接跳转指令，对此 PT 硬件会直接记录跳转目标地址而非跳转状态，从而导致跟踪数据量的大幅增加；其次，PT 硬件在跟踪时会添加大量用于对齐填充的数据至输出缓冲区，随着记录数量的增加，这些额外数据也会快速增长。在 PT 产生较大数据量的情况下，如果将部分数据转存至文件则可能会对目标程序运行性能造成较大影响。但主流服务器所支持的物理内存容量基本能满足上述存储要求，并且 64 位系统支持 128 TB 用户虚拟地址空间，框架也能够较容易地完成内存地址的分配和映射。此外，基准测试程序作为内存访问密集型程序，多数情况下运行时间较长并且包括大量循环结构，而在实际应用场景下，所分析目标程序的代码范围和执行时间都可以进行控制，因此产生的跟踪数据量也会小于上述情况。

2.2　案例研究

1) 应用程序

利用实际环境下的程序和分析场景来评估 DRIPT 的分析效果。首先针对应用程序进行实验，主要涉及 CPU 和内存访问密集型的压缩工具，实验测试场景包括统计程序执行的指令数量、跟踪函数调用和返回过程以及评估内存访问跟踪性能。在实验中，本文选择目标系统目录下的 mshtml 文件作为压缩对象，文件大小约为 20 MB，采用程序提供的默认命令行参数分别将其压缩为 7z、xz、rar、cab、pea 格式。本文增加了与经典跟踪分析框架 PANDA 的最近版本（v1.8.72）进行比较，PANDA 基于 QEMU 框架实现，采用记录和重放方法实现对程序的跟踪分析，它同样直接对二进制程序进行跟踪而未引入额外的静态分析，与本文框架的设计目标和应用场景类似。本文在相同的测试主机上利用 WSL2（Windows subsystem for Linux table of physical addresses）子系统构建了运行环境，并在实验时将其记录的数据存储在内存盘中。为评估跟踪分析对目标程序执行的影响，本文在不同测试场景中对比了实际程序执行时间，结果如图 6 所示。DR_IC、DR_CALL 和 DR_MEM 分别表示利用 DynamoRIO 内置的 inscount、instrcalls 和 memtrace_binary 插件模块在线实现上述场景的测试，而其他程序则是先进行在线运行的信息记录，而后基于记录数据完成相应功能。PANDA_R 表示在其运行环境下针对目标程序执行过程进行记录，而且对于相同程序的不同分析场景，PANDA 的记录过程是相同的。DRIPT_IC 表示在 DRIPT 框架下对程序进行跟踪，同时利用动态插桩预构建用于指令统计的分析代码；DRIPT_CALL 利用动态插桩解析函数调用相关指令，并关联函数所对应的模块信息；DRIPT_MEM 则利用 PTWRITE 指令记录程序执行时所有的

内存访问地址。如图6所示，与直接在线跟踪分析相比，DRIPT在记录阶段对目标程序的执行性能影响较低，在轻量级跟踪需求下，对于部分程序的执行影响接近直接使用PT进行跟踪的场景。PANDA在进行记录时所花费时间普遍较长，因为QEMU会带来指令翻译执行的额外开销，而DRIPT在融合PT基础上可以充分利用主机的处理器和内存资源。此外，除DRIPT_CALL外，其他场景所记录的数据均在内存中进行存储而未输出至文件。而对于DRIPT_CALL，其在跟踪时将数据完全写入本地固态盘中，在保存数据量较低时对于执行性能的影响也相对较小。

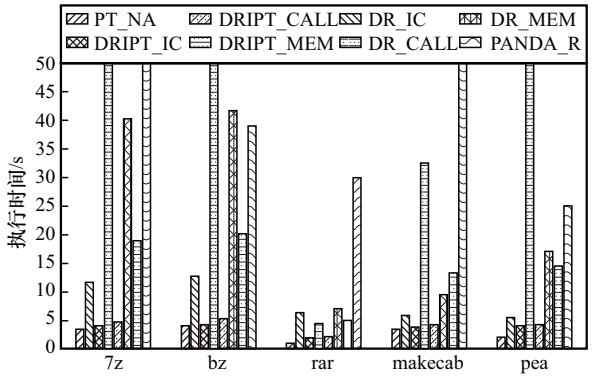


图6　不同测试场景中实际程序执行时间对比

在跟踪记录的数据基础上，进一步实现指令统计和函数调用和返回跟踪功能，表1中具体展示了DRIPT与PT_NA及PANDA在解码和分析时的统计数据，主要涉及跟踪产生的数据量和解析时间的对比。与基准测试程序比较，实际程序在执行时涉及的模块和线程数量较多，但程序执行时间相对较短。因此测试过程所输出的控制流数据总大小也较低，普通主机的配置就能够支持在内存中完全存储跟踪数据。从表1中可以看出，在轻量级跟踪场景下，不同程序执行所产生的记录数据大小不超过8 GB。但在内存跟踪测试时产生的数据量较大，平均为90 GB，与2.1节中的实验开销接近，该内存要求在实际应用场景下也可以得到满足。对于DRIPT而言，为完成上述2种功能进行记录时产生的数据量接近，但总共耗费的时间会有差别。

在解析数据时，对于原生PT记录利用官方提供的ptxed工具进行解析，以提取程序执行的基本块和指令数量。如表1所示，尽管PT_NA实验所产生的数据量很低，但是多数情况下，其解析时间远高于DRIPT。主要原因在于，ptxed在解析记录并统计执行指令数量时，依赖对原来的二进制模块进行反汇编分析，PT硬件不会记录这些数据，并且会丢失在动态分配的内存中所执行的代码信息。而DRIPT则在程序运行时利用动态插桩进行指令解析，并将结果保存于内存中，既能利用动态插桩工具的丰富接口，又能加快后续的代码分析速度。此外，由于程序执行的控制流信息是按线程进行记录的，因此在提取控制流时可以采用多个解析线程并发处理，实际耗时主要受到最大记录容量解析的影响。例如，对于rar程序，其在执行时产生了近20个线程，实际每个线程记录的信息量较低，因此ptxed的解析时间会略低于DRIPT。

PANDA框架利用重放记录数据的方式实现对执行指令数量的统计，在实验中利用PANDA内置的instr_count插件完成。从表1中可以看出，与DRIPT相比，PANDA记录数据量总体较小，因其只记录非确定性事件并采用压缩存储，但在重放数据并进行解析时所花费的时间也更长。对于函数调用和返回指令的跟踪，PANDA提供了callstack_in-

表1　　**DRIPT与其他方法的实验数据对比**

程序	模块	线程	PT_NA					DRIPT				PANDA	
			解析基本块/×10⁶个	解析指令数/×10⁶个	解析时间/s	总记录大小/GB	最大线程记录/GB	解析时间/s		总记录大小/GB	最大线程记录/GB	解析时间/s	总记录大小/GB
								指令统计	函数跟踪				
7z	21	7	2 597	16 930	206	1.52	1.18	57	122	5.70	3.27	323	0.65
bz	28	4	2 600	17 692	223	1.74	1.42	74	132	6.45	3.60	101	0.84
rar	41	17	993	6 402	13	0.30	0.03	21	23	2.37	0.81	227	0.86
makecab	18	3	1 830	8 682	314	0.54	0.54	34	58	2.65	2.56	73	0.64
pea	48	5	790	6 223	131	0.38	0.38	32	53	4.14	4.07	83	0.67

str 插件，但它面向系统范围，所以执行的时间也非常长，远大于基础重放的时间，未在表 1 中列出。在 DRIPT 中，本文参考其插件代码实现了函数调用和返回行为的跟踪，并为源程序的每个线程维护函数调用堆栈等信息。与 PANDA 相比，该功能可以较快完成，但与 DRIPT 的指令统计相比，其耗费的时间也会更长。总体来看，与直接利用 PT 解码工具解析以及 PANDA 重放分析进行比较，DRIPT 在不同的测试场景下都能够取得较好的效果。

此外，在对程序进行插桩时，动态插桩工具支持的 TRACE 功能可能会调整部分源程序代码块的结构，从而影响 PT 实际记录的控制流信息与原程序结构的对应关系，前文也对此进行了描述并给出解决方案。由于 DynamoRIO 提供了关闭 TRACE 功能的命令行参数，因此本文分别测试该功能开启和关闭 2 种模式并进行对比分析，实验涉及上述跟踪并实现指令统计的场景。统计和比较的结果如表 2 所示，其中 E 表示默认情况，即启用 TRACE 功能；D 表示通过命令行参数关闭该功能。

如表 2 所示，当 TRACE 功能关闭时，DRIPT 产生的跟踪数据量会明显减少。一方面，对于部分 TRACE 代码缓存块，DRIPT 增加使用 PTWRITE 指令来直接记录其地址，会产生较多的“PTW”事件记录。另一方面，在 TRACE 功能开启的情况下，框架在添加代码缓存块地址到哈希表时产生冲突的情况也会增加，对此框架会利用上述指令直接记录缓存块地址；当 TRACE 功能关闭后，代码缓存块在构建映射关系时出现地址冲突的情况也会减少。从表 2 中还可以看出，由 PTWRITE 指令所引入的“PTW”类型数据包在 2 种测试情况下所占的比例明显不同。尽管大量“PTW”记录会在一定程度上增加 PT 的性能开销，但在实验中对各程序的影响并不显著，2 种运行模式带来的程序执行时间开销接近。此外，由于总体记录数据量的增加，在默认情况下，对跟踪记录进行解析所耗费时间会略高于 TRACE 功能关闭的场景，但也会有例外情况，例如 makecab 程序，主要原因是在关闭场景下，该程序执行时产生较多的“TNT”跟踪类型，即程序会执行更多的条件跳转指令，针对该记录类型，解码程序需要进行大量的反汇编分析，从而导致整个时间的增加。

2) 恶意样本

渗透测试工具 Cobalt Strike 常被用于各种网络攻击活动，其核心组件是一个被称为 Beacon 的后门载荷。在多阶段加载模式下，攻击者通常首先通过钓鱼邮件等方式投递初始载荷，其先执行一段 shellcode，从 C2 服务器下载完整的 Beacon 载荷并在内存中执行，随后回连 C2 服务器完成“上线”，以实现远程控制、数据窃取等恶意操作。在实验中，本文在本地搭建了 C2 服务端及通信环境，利用 DRIPT 针对其上线过程进行分析。

首先在 DRIPT 环境下加载初始载荷执行，与正常执行类似，经过约 2 s 后上线连接到控制端，然后中止程序并基于产生的跟踪数据进行分析。其中初始执行的程序大小约为 18 KB，在执行期间产生约 10 个线程共约 800 MB 记录数据。在分析时，针对记录数据进行解析并提取程序执行的所有代码块，针对每个代码块地址判断是否为 WININET 模块中的函数，例如 InternetOpen、HttpSendRequest 等表示程序利用系统提供的网络相关 API 函数发起通信请求。在分析过程中，程序会出现多次网络通信相关函数调用，同时框架输出一些最近执行的代码块地址，通过这些地址可以回溯程序的执行流

表 2　　DRIPT 在不同执行参数下的实验数据

程序	解析时间/s		总记录大小/GB		Packet 数量/10^6 个		PTW		TNT		代码缓存块数量							
											总数		冲突		TRACE		重组	
	E	D	E	D	E	D	E	D	E	D	E	D	E	D	E	D	E	D
7z	57	48	5.70	1.74	1 573	777	27.8%	0.01%	43.7%	73.1%	17 671	18 005	21.9%	1.4%	5.0%	0	2.2%	0
bz	74	42	6.45	2.17	1 750	925	27.5%	0.01%	40.9%	70.3%	23 553	24 436	18.3%	1.1%	4.2%	0	1.6%	0
rar	21	8	2.37	0.96	704	376	15.4%	0.03%	48.2%	66.4%	38 967	39 997	18.6%	0	4.3%	0	1.8%	0
makecab	34	48	2.65	0.84	835	417	19.0%	0.08%	53.1%	82.4%	8 404	8 589	28.8%	0	6.7%	0	3.3%	0
pea	32	22	4.14	1.97	1 219	673	10.7%	0.04%	42.1%	52.3%	94 359	109 641	29.0%	0	6.0%	0	3.1%	0

程，定位具体的代码和模块，并获取在运行时生成的解析摘要。

样本通信过程分析示例如图7所示，框架在执行时捕获到Beacon模块利用HttpSendRequestA函数将请求发送到服务器，此时可以得到最近执行的代码块地址及相应的解析信息。在图7中0x331ddc5处的代码块最后将会调用位于7ffd`256c5250处的发送函数，本文可以从其对应的解析摘要中提取关联模块名等信息。而对于之前执行的其他代码块，也可以通过调试器加载目标进程转储做进一步分析确认。此外，由于Beacon模块直接在内存中通过反射式加载执行，因此这些代码块地址并没有关联具体模块信息，通过该现象也能够迅速发现恶意代码的隐蔽运行方式。利用DRIPT框架能够实现对样本程序的高效跟踪和分析，既能利用动态插桩工具提取语义信息，又能够利用硬件机制加速跟踪过程，还能够避免传统基于断点调试方式所导致的网络超时等异常情况。

3　相关研究工作

二进制程序跟踪和分析技术在恶意代码分析以及漏洞挖掘等领域有较多的应用和研究，包括基于动态插桩框架实现数据流分析、代码去混淆、相似性检测等[15]，以及利用Intel PT等硬件和系统机制在模糊测试、控制流防护等场景中实现高效的程序执行跟踪[6,16-17]。

1) 基于动态插桩工具。杨盼等[18]提出了基于函数摘要优化的动态污点分析框架FSTaint，基于Intel Pin开发插件来记录程序执行信息。张沈芊芊等[19]利用Pin对虚拟化保护程序进行插桩和跟踪，而后进行离线分析以提取虚拟化指令。Straight-Taint通过记录执行信息来重建污点分析代码，降低运行时开销，同样利用Pin实现了原型框架[14]。Galea等[20]提出了Taint Rabbit以在动态污点分析中生成快速路径，并实现了新的DynamoRIO插件。TypeSeezeR将静态分析与动态插桩技术进行结合，恢复二进制程序中的函数签名[21]。已有工作大多直接利用传统插桩工具或其插件实现更高效深入的程序分析方法，而本文对传统动态插桩工具进行增强，基于其源代码进行修改，使其能够与硬件跟踪技术进行融合，提高了利用动态插桩框架对程序控制流进行跟踪的性能，并且在此基础上实现新的解耦分析框架。

2) 基于硬件和系统机制。毕野川等[22]提出基于系统调用接口的驱动漏洞挖掘方法，利用Intel PT获取代码覆盖率。Kilic等[23]提出Memgaze，利用Intel PT实现快速高效的内存跟踪分析。Zeng等[11]在攻击调查中，基于PT记录进行污点分析来恢复指令级的因果关系，优化了基于审计日志的攻击场景重建和溯源能力。Zhang等[12]提出了一种混合的动态污点跟踪系统HardTaint，结合了静态分析、PT硬件跟踪和图处理技术，以降低运行时开销。PANDA跟踪框架基于系统记录和重放技术来进行程序分析[10]，主要基于QEMU实现，支持多种

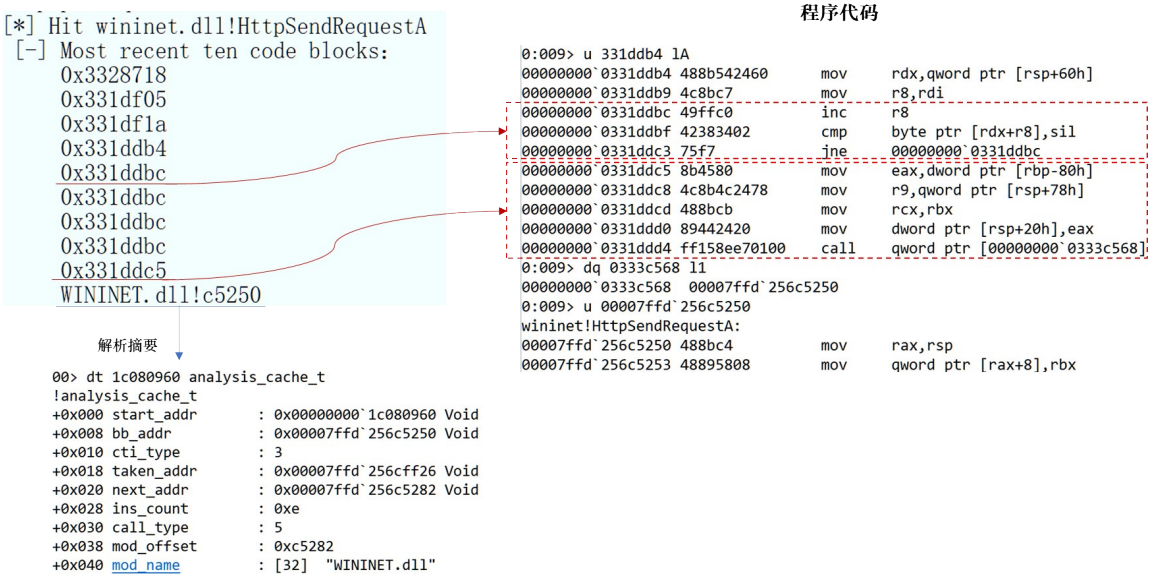


图7　样本通信过程分析示例

CPU 架构并提供插件实现不同的分析功能。很多已有工作采用 Intel PT 等硬件和系统机制实现对二进制程序的动态跟踪，并结合静态分析、溯源图等分析技术实现了更复杂的分析系统。而本文聚焦于用户应用程序的直接动态跟踪和分析，利用硬件机制增强了传统动态插桩工具的能力，提供了针对二进制程序进行跟踪分析的轻量级基础框架，可直接在目标系统环境中部署运行。

4　讨论

当前在增强动态插桩工具的功能时依赖于源代码层面的修改，而部分插桩工具（如 Intel Pin）未开放源代码，如果直接修补二进制模块则存在较大挑战，依赖大量的逆向分析工作。框架的内核模块由于涉及线程跟踪功能，需要适配更多的操作系统版本并进行测试，并且当前只适用于 Windows 系统，在其他系统上的实现还需要进行探索和验证。

如果目标程序长时间运行，PT 硬件可能会产生大量的跟踪数据，尤其在目标程序运行产生较多“TIP”“PTW”等类型事件的情况下，可能会超出主机物理内存容量。此时，可以在管理 PT 的驱动模块中增加对插桩工具的跟踪，通过设置指令地址范围进行过滤，从而只记录与分析目标代码相关的执行过程。事实上，除了在本地保存数据外，PT 机制还支持通过高速硬件通道直接将数据传输到外部专用设备。

在分析阶段，本文利用指令统计、函数调用跟踪等案例来验证原型框架的分析功能，实际上也可以基于动态插桩工具开发更复杂的分析功能，例如进行动态污点分析、与系统溯源工作结合等。此外在如何利用硬件跟踪提高解耦分析的能力方面还需要开展进一步的研究。

5　结束语

本文针对二进制程序动态跟踪和分析问题，提出了一种新的软硬件结合的跟踪和分析方法，在传统动态插桩工具的基础上，引入硬件跟踪机制，进一步降低目标程序的执行性能开销，并提高离线分析的灵活性。本文在 DynamoRIO 和 Intel PT 基础上实现了所提方法的原型框架，通过不同实验场景验证了方法的有效性，后续将在结合其他动态插桩工具、实现其他分析模块等方面做进一步扩展研究。

参考文献：

[1] 周忠君, 董荣朝, 蒋金虎, 等 . 二进制代码安全分析综述[J]. 计算机系统应用, 2023, 32(1): 1-11.
ZHOU Z J, DONG R C, JIANG J H, et al. Survey on binary code security techniques[J]. Computer Systems and Applications, 2023, 32(1): 1-11.

[2] 郭曦, 王盼 . 面向代码重用检测的程序语义分析模型[J]. 通信学报, 2024, 45(12): 179-196.
GUO X, WANG P. Program semantic analysis model for code reuse detection[J]. Journal on Communications, 2024, 45(12): 179-196.

[3] OR-MEIR O, NISSIM N, ELOVICI Y, et al. Dynamic malware analysis in the modern era: a state of the art survey[J]. ACM Computing Surveys, 2019, 52(5): 1-48.

[4] ZHANG F W, LEACH K, STAVROU A, et al. Using hardware features for increased debugging transparency[C]//Proceedings of the 2015 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2015: 55-69.

[5] CHENG B, LEAL E A, ZHANG H, et al. On the feasibility of malware unpacking via hardware-assisted loop profiling[C]//Proceedings of the 32nd USENIX Conference on Security Symposium. New York: ACM Press, 2023: 7481-7498.

[6] SCHUMILO S, ASCHERMANN C, GAWLIK R, et al. kAFL: hardware-assisted feedback fuzzing for OS kernels[C]//Proceedings of 26th USENIX Security Symposium (USENIX Security 17). Berkeley: USENIX Association, 2017: 167-182.

[7] LUK C K, COHN R, MUTH R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. ACM SIGPLAN Notices, 2005, 40(6): 190-200.

[8] SULLIVAN G T, BRUENING D L, BARON I, et al. Dynamic native optimization of interpreters[C]//Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators. New York: ACM Press, 2003: 50-57.

[9] HENDERSON A, PRAKASH A, YAN L K, et al. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. New York: ACM Press, 2014: 248-258.

[10] DOLAN-GAVITT B, HODOSH J, HULIN P, et al. Repeatable reverse engineering with PANDA[C]//Proceedings of the 5th Program Protection and Reverse Engineering Workshop. New York: ACM Press, 2015: 1-11.

[11] ZENG J, ZHANG C Q, LIANG Z K. PalanTír: optimizing attack provenance with hardware-enhanced system observability[C]//Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2022: 3135-3149.

[12] ZHANG Y Y, LIU T Y, WANG Y Y, et al. HardTaint: production-Run dynamic taint analysis via selective hardware tracing[J]. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA2): 1615-1640.

[13] DAVID Y, SUN X, SOFAER R J, et al. UPGRADVISOR: early adopting dependency updates using hybrid program analysis and hardware tracing[C]//Proceedings of 16th USENIX Symposium on Operating

Systems Design and Implementation (OSDI 22). Berkeley: USENIX Association, 2022: 751-767.

[14] MING J, WU D H, WANG J, et al. StraightTaint: decoupled offline symbolic taint analysis[C]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. New York: ACM Press, 2016: 308-319.

[15] 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]//Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. New York: ACM Press, 2019: 15-27.

[16] 李贺, 张超, 杨鑫, 等. 操作系统内核模糊测试技术综述[J]. 小型微型计算机系统, 2019, 40(9): 1994-1999.

[17] GE X, NIU B, CUI W. Reverse debugging of kernel failures in deployed systems[C]//Proceedings of 2020 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2020: 281-292.

[18] 杨盼, 康绯, 舒辉, 等. 基于函数摘要的二进制程序污点分析优化方法[J]. 网络与信息安全学报, 2023, 9(2): 115-131.
YANG P, KANG F, SHU H, et al. Binary program taint analysis optimization method based on function summary[J]. Chinese Journal of Network and Information Security, 2023, 9(2): 115-131.

[19] 张沈芊芊, 董卫宇, 林键. 虚拟化混淆程序的指令提取方法[J]. 信息工程大学学报, 2025, 26(1): 83-89.
ZHANG S, DONG W Y, LIN J. An instruction extraction method for virtualization obfuscated program[J]. Journal of Information Engineering University, 2025, 26(1): 83-89.

[20] GALEA J, KROENING D. The taint rabbit: optimizing generic taint analysis with dynamic fast path generation[C]//Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. New York: ACM Press, 2020: 622-636.

[21] LIN Z Y, LI J K, LI B W, et al. TypeSqueezer: when static recovery of function signatures for binary executables meets dynamic analysis[C]//Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2023: 2725-2739.

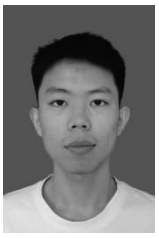
[22] 毕野川, 彭建山, 林志强. 基于系统调用接口的Windows驱动程序漏洞挖掘方法[J]. 计算机应用与软件, 2025, 42(1): 359-366.
BI Y C, PENG J S, LIN Z Q. Windows driver vulnerability mining method based on system call interface[J]. Computer Applications and Software, 2025, 42(1): 359-366.

[23] KILIC O O, TALLENT N R, SURIYAKUMAR Y, et al. MemGaze: rapid and effective load-level memory trace analysis[C]//Proceedings of the 2022 IEEE International Conference on Cluster Computing (CLUSTER). Piscataway: IEEE Press, 2022: 484-495.

[作者简介]

潘家晔（1985– ），男，江苏扬州人，博士，南京邮电大学副教授，主要研究方向为系统安全、软件安全、网络安全。

沙乐天（1985– ），男，江苏徐州人，博士，南京邮电大学教授，主要研究方向为漏洞挖掘、恶意软件分析、软件安全。



鲁京（2002– ），男，江苏扬州人，南京邮电大学博士生，主要研究方向为系统安全、软件安全。

肖甫（1980– ），男，湖南邵阳人，博士，南京邮电大学教授、博士生导师，主要研究方向为无线传感器网络、物联网安全、信息安全。