软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(9):3249−3270 [doi: 10.13328/j.cnki.jos.006300]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于渐进扩展的二进制程序数据流分析方法*

潘家晔[1], 庄 毅[2], 孙炳林[2]

[1](南京邮电大学 现代邮政学院, 江苏 南京 210003)

[2](南京航空航天大学 计算机科学与技术学院, 江苏 南京 210016)

通信作者: 庄毅, E-mail: zy16@nuaa.edu.cn

摘 要: 二进制程序分析技术广泛应用于软件的安全性评估, 恶意代码分析等领域. 动态分析技术能够准确体现程序真实的运行状态, 但面临目标程序运行负载过高、难以深入了解内部结构信息等挑战. 提出一种基于渐进扩展的二进制程序数据流分析方法. 方法旨在充分利用在线数据流分析的能力, 在局部细粒度分析的基础上逐渐扩展分析范围, 从而使分析能够覆盖整个目标程序. 通过设计的分治策略, 可降低对目标程序运行时的性能影响, 从而可使对延迟敏感的目标代码段能成功地执行. 并在此基础上, 进一步提出基于内存引用关系的函数参数相关性分析方法, 从函数调用层面获取数据流传递信息, 可辅助恢复参数的内部结构信息. 通过对大量真实案例进行研究和实验, 验证了所提出方法的可行性与有效性, 在降低对目标程序影响的同时未引入显著的额外分析开销, 能够用于实际环境下二进制程序的分析.

关键词: 二进制程序; 数据流分析; 污点跟踪; 恶意代码; 逆向分析

中图法分类号: TP311

中文引用格式: 潘家晔, 庄毅, 孙炳林. 基于渐进扩展的二进制程序数据流分析方法. 软件学报, 2022, 33(9): 3249–3270. http://www.jos.org.cn/1000-9825/6300.htm

英文引用格式: Pan JY, Zhuang Y, Sun BL. Data Flow Analysis Method Based on Progressive Dynamic for Binary Programs. Ruan Jian Xue Bao/Journal of Software, 2022, 33(9): 3249–3270 (in Chinese). http://www.jos.org.cn/1000-9825/6300.htm

Data Flow Analysis Method Based on Progressive Dynamic for Binary Programs

PAN Jia-Ye[1], ZHUANG Yi[2], SUN Bing-Lin[2]

[1](School of Modern Posts, Nanjing University of Posts and Telecommunications, Nanjing 210003, China)

[2](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

Abstract: Binary program analysis techniques are widely applied in software security testing, malware analysis and detection, etc. Dynamic analysis is an important analysis method that can accurately show the running status of programs. However, it is confronted with some challenges, such as too high load during target program running and difficulty in dissecting the data structure information in detail. This study proposes a new data flow analysis method based on progressive expansion for binary programs. By taking full advantage of the ability of online data flow analysis, it focuses on the fine-grained analysis for partial program and expands the analysis range progressively to cover the entire program. The method utilizes a divide-and-conquer strategy that can reduce the performance impact on the runtime of the target program and thereby enable the execution of the target code segment sensitive to delay. Meanwhile, this study also presents a correlation analysis method for function parameters based on the memory reference relationship. It can detect the data flow propagation at the function call level and aid in the recovery of the internal data structures of parameters. In the end, this study shows the results of the experiments on the programs in the real environment, which suggest the feasibility and effectiveness of the proposed method. This method does not introduce significant extra analysis overhead while reducing the performance impact on the target program, capable of being applied in binary program analyses in practice.

Key words: binary program; data flow analysis; taint tracking; malware analysis; reverse analysis

* 基金项目: 国家自然科学基金 (61572253); 江苏省研究生科研创新计划 (KYLX16_0384)
收稿时间: 2019-04-30; 修改时间: 2020-03-24, 2020-07-02; 采用时间: 2020-08-07; jos 在线出版时间: 2022-07-15

随着操作系统安全性和安全软件防御能力的不断提高, 针对终端用户的漏洞和恶意代码攻击得到一定程度上的遏制, 但是恶意程序仍然以新的形式影响终端用户, 如: 勒索软件以及挖矿代码等[1]. 对于安全研究者, 在未来的时间里, 恶意软件的检测与分析仍然是一个长期的任务与挑战. 在过去 20 年里, 已有较多的关于恶意软件和二进制程序分析方面的研究[2], 具体涉及静态分析和动态分析方法[3]. 其中, 静态分析具有较好的代码覆盖率, 能够发现代码中的潜在的执行路径, 但面临代码加密混淆等挑战. 动态分析对程序的执行过程进行跟踪和调试, 能够获取代码的真实执行逻辑和路径, 同样也面临潜在的反调试分析等对抗措施[4].

常见动态分析方法有基于沙箱的程序行为分析方法[5,6], 但基于沙箱的行为分析难以发现恶意代码内部的实现细节, 比如 C&C 地址生成算法, 配置信息的加密和解密算法, 数据的移动过程等. 基于指令粒度的动态数据流跟踪 (dynamic data flow tracking)[7,8]或者动态污点分析 (dynamic taint analysis)[9]技术, 能够在指令级别对每个内存地址或寄存器进行跟踪分析, 被广泛应用于漏洞挖掘、攻击检测以及隐私数据泄露检测等方面. 如今, 动态污点分析作为比较优秀的二进制程序分析方法, 其面临的主要挑战之一是对目标程序造成较严重的性能影响. 对于大型复杂的应用程序或者执行延迟敏感的程序, 可能会影响其正常运行, 从而导致分析失败. 例如: 目标代码量太大导致持续分析所需的内存不足, 恶意代码中存在对执行时间的检测, 或者程序某线程执行时间较长而引起其他线程的等待超时. 类似的场景出现在本文对 Firefox 的某功能分析中. 基于提升分析性能的目标驱动, 已有较多研究工作提出优化和改进方法, 包括预先对程序进行静态分析, 合并减少动态分析指令的数量[10,11]; 将目标程序的执行与动态污点分析代码进行分离[11,12]. 通过在线记录程序执行路径而后进行离线分析, 能够有效降低在线分析对目标程序运行的影响[13,14], 但在进行离线环境重建和分析时, 面临缺乏一些程序运行时信息, 包括实时的系统调用运行结果等. 总的来说, 在恶意程序及二进制代码的分析过程中, 了解数据的传递过程是一个重要的方面, 有助于发现可能存在的数据变换和传递过程[15]; 另外一方面, 正常的软件如果存在错误配置的情况也可能存在着数据泄露行为[16], 分析其中的数据流传递也具有重要的意义.

细粒度动态分析方法的具体实施主要涉及两个层面. 一是在目标系统外部构建分析框架, 通常基于 QEMU 等模拟器或 VMI 等虚拟化方法来对目标指令的执行进行转译和分析[17,18], 部分框架能够实现包含内核在内的全系统范围的分析, 并具有较好的分析隐蔽性, 但在程序运行时语义信息的获取, 部署便捷性等方面仍存在不足; 另外一种是在系统内对应用层的程序进行分析, 基于类似 JIT 的方法对目标程序进行解释翻译并生成新的执行和分析代码, 为此能够快速地实现分析工具, 具有较好的灵活性, 但也在分析检测对抗能力、分析性能等方面存在问题和挑战[7,11]. 当前由于用户模式下应用程序的分析需求仍然较大, 在系统中快速构建所需工具并对目标程序进行分析也是分析人员关注和研究的重点[19]. 为克服深度分析时过度的指令插桩对目标程序正常运行的影响, 本文从新的角度开展研究, 在已有研究工作的基础上, 提出了一种渐进扩展的二进制程序数据流分析方法, 能够提高对目标程序的在线分析能力, 同时不引入过多的性能开销而影响目标程序的正常运行, 该方法支持对分析状态进行保存和恢复, 可为延迟敏感或大型复杂程序的分析提供帮助. 其主要思想是将目标程序的数据流分析任务分解为多个阶段完成, 在每次的程序执行过程中, 只完成其中部分的数据流分析工作, 可以降低对目标程序运行时累积的性能影响, 保证目标程序不会因引入过多延迟而出现运行结果与正常情况不一致的情况. 另外, 对于分析时间较长的情况, 可以在中间阶段进行适时保存, 以防止出现异常情况, 导致已有分析结果的丢失. 同时我们设计了程序分析过程的状态信息保存和恢复方法, 以支持渐进扩展分析, 并能够支持对多线程程序的分析.

该方法主要面向在系统内对用户态应用程序的分析, 旨在降低持续的在线指令插桩分析对程序正常运行的影响, 充分利用在线分析的能力. 与系统范围的分析方法相比, 能够在目标系统内快速对程序进行在线重复分析, 具有更好的灵活性, 与其他应用层分析方法相比, 除采用分治策略外, 其进一步提升了在线数据流分析的能力, 同时不对目标程序进行转换和修改, 不需要进行深度的预分析处理, 且不会造成显著的性能影响. 另外, 该方法支持多线程环境下分析状态的保存和恢复, 可避开延迟敏感的程序代码对分析的影响. 通过对大量实际程序进行分析, 实验结果表明上述机制给目标程序带来的整体性能开销较低. 最后给出的案例分析表明, 本文所提出的方法能够在实际环境下正确和有效地对较多的目标程序进行渐进的深入分析. 本文的主要贡献如下.

1) 提出了一种渐进扩展的二进制程序数据流动态分析方法. 可提高局部细粒度的在线数据流分析能力, 利用

分治策略将分析逐渐扩展到整个目标程序. 该方法充分利用动态在线分析的优势, 可对程序进行深入分析, 同时降低对目标程序的性能影响.

2) 设计了程序分析过程的分析状态保存和恢复方法. 在进行数据流分析时, 记录程序部分运行时信息以用于后续的分析. 可支持多线程程序, 并克服程序在多次运行时控制流出现变化的影响.

3) 在数据流分析阶段, 提出了基于内存引用关系的函数参数相关性分析方法. 通过构建内存地址引用关系图并设计图搜索策略, 可在函数调用层面提取函数参数中潜在的数据流传递信息.

4) 在 Windows 平台中实现了本文所提出方法的原型系统, 通过真实的案例研究, 验证了方法的可行性, 该方法能够对目标程序进行深入分析, 同时具有较低的性能开销.

1 方法描述

1.1 分析场景和假设

假设目标程序或者程序的部分代码在每次独立运行过程中, 其功能保持不变, 我们称该分析目标具有稳定的执行流. 例如: 我们在分析目标程序的加密模块时, 其加密算法和输入可保持不变. 如果目标程序在多次执行过程中产生不同的结果或者目标程序实现复杂导致难以进行重复分析, 本文所提出的方法则不能适用于上述场景的分析. 事实上, 据实际观察, 很多程序在重复执行时其功能不会发生改变. 但一些实现复杂的程序在运行过程中会受到网络环境、时间因素、内存分配、线程调度以及用户交互操作等外部因素的影响.

因此, 在分析时我们首先尝试提取目标代码的稳定执行流, 并判断本文所提出的方法能否适用于分析目标. 同时应保持外部环境的相对稳定来降低目标程序执行的变化程度, 例如: 不改变系统内存资源的占用; 适时恢复被目标程序改变的配置; 不引入额外的图形界面操作等. 对于不影响目标程序分析但会引起控制流 (control flow) 变化的代码段, 可在分析时进行忽略, 如已知的内存分配和释放函数、线程同步函数、模块加载过程等.

1.2 渐进分析

渐进分析的核心思想是分多个阶段对目标程序进行分析, 在各阶段完成分析状态的恢复或者数据流分析任务, 并保证整个过程分析的连续性和一致性. 在重新执行和分析目标程序时, 对已经分析过的执行阶段, 以分析状态的恢复过程来替代数据流分析. 与程序执行状态不同, 分析状态是指在目标程序整个执行和分析过程中的某时刻产生的中间分析结果, 并用额外的存储空间来进行保存, 主要涉及目标进程所访问的内存地址和关联的线程上下文. 需要恢复的分析状态主要包括两部分: 一是程序进程共享的被标记或污染的内存地址. 当目标程序再次运行时, 其分配到的地址可能会发生变化, 在进行状态恢复时只需关注那些影响内存地址分析状态传递的指令及其执行情况, 并据此对涉及的内存分析状态进行恢复, 从而避免原来直接进行数据流分析时逐条指令插桩与分析; 另一部分是线程在状态保存处的上下文分析状态. 当目标线程重新执行到上次保存位置时, 恢复其上下文中相关寄存器的分析状态即可. 在完成恢复后, 将按需继续往前执行目标程序并同时推进后续的分析.

若将目标程序的分析过程划分为 n 个区间, 其中每区段执行的指令数量不一定相同. 实际中可以根据对目标程序执行的性能影响情况进行调整, 以减少对延迟敏感代码段的分析时间. 设 $P_i, 0 \leqslant i \leqslant n$ 表示各个区段的交界位置. P_0 表示等待分析的目标代码开始处, 而 P_n 则表示分析结束的位置. 对于每一次的渐进分析过程, 程序运行在 $P_0 \rightarrow P_{k-1}$ 区段时, 将恢复在位置 P_{k-1} 处保存的分析状态, 其中分析状态来自之前的分析过程. 当程序运行在 $P_{k-1} \rightarrow P_k$ 区段时, 进行数据流分析以及分析信息的记录, 其中, P_k 处为分析状态的保存点以及分析代码与目标程序的分离点, 且 $1 \leqslant k \leqslant n$.

如图 1 所示, 其中展示了对目标程序进行分析的若干阶段. 从阶段 1 中可以看到, 在分析开始后, 首先执行数据流分析并且同时记录所需的运行时信息, 在分析执行到达 P_1 时, 将退出分析过程. 此时可立即终止目标程序或者将程序转入原生执行直至退出, 与此同时保存当前的分析状态. 其中, 分离目的是随时可继续进行下一区间的分析, 如图 1 中阶段 3 所示, 程序在跳过的区间中为正常运行, 直到下个目标区间出现时再转入分析执行. 为了避免保存时的写文件操作对目标程序的影响, 可将写入过程延后到程序退出前进行.

如图 1 中阶段 2 所示, 首先恢复目标程序的分析状态到上一次保存的位置 P_1, 而后再继续对下一区间进行分析. 当进入下个阶段时, 对于程序的重新执行和分析, 有两个选项可以考虑. 一是重新执行目标程序, 并从开始起恢复分析状态. 如图 1 中阶段 2 所示, 需要恢复 $P_0 \to P_1$ 区间的分析状态. 二是从程序或系统快照处开始进行恢复, 如图 1 中所示, 在阶段 2 中完成恢复并保存程序快照后再继续进行分析, 然后在阶段 3 中, 首先恢复程序的执行状态到 P_1 处, 而后再进行 $P_1 \to P_2$ 区间的分析恢复过程. 其优点是可降低进程运行中的变化程度, 并有利于维持线程的一致性. 而缺点是稳定的快照技术多数依赖于虚拟机, 增加分析环境的复杂性, 并且可能会导致分析误差的累积. 另外需要注意的是, 在进行分析状态的恢复时, 只需进行增量部分的恢复即可. 例如: 在 $P_1 \to P_2$ 区间中如果没有数据流传递情况的出现, 那么只需确认线程的执行进度即可, 此过程接近于代码原始执行. 假设采用传统数据流分析方法直接对区间 $P_1 \to P_2$ 进行分析, 将可能会导致程序执行超时, 而此时采用渐进方法则可以规避该问题.

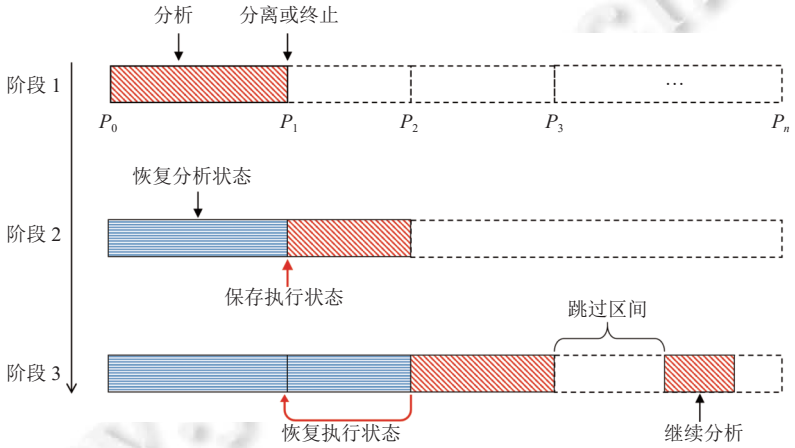

图 1 渐进分析过程示意图

令 $T_d(i)$ 表示采用传统方法对目标区间 $P_{i-1} \to P_i$ 进行数据流分析的时间开销, 因此对整个程序进行分析的总时间开销为 $T(0,n) = T_d(1) + T_d(2) + \cdots + T_d(n) = \sum_{i=1}^{n} T_d(i)$. 类似地令 $T_{pd}(i)$ 表示采用渐进方法对 $P_{i-1} \to P_i$ 区段进行分析的耗时, 用 $T_r(i)$ 表示对相同区间进行分析状态恢复的时间开销, $T_a(i)$ 则表示在分析前后额外的时间开销, 如: 数据处理等. 由于在分析时存在额外的记录操作, 因此有 $T_{pd}(i) > T_d(i)$. 同时, 本方法期望能够保证 $T_r(i) \ll T_d(i)$. 因此采用渐进方法分析整个目标程序的累计总时间开销为:

$$T'(0,n) = T'(0,n-1) + T_{pd}(n) + T_a(n) + \sum_{i=1}^{n-1} T_r(i) = T'(0,n-1) + T'(0,n-1) + T_{pd}(n) + T_{pd}(n-1) + T_a(n)$$

$$+ T_a(n-1) + \sum_{i=1}^{n-1} T_r(i) + \sum_{i=1}^{n-2} T_r(i) = \sum_{i=1}^{n} T_{pd}(i) + \sum_{i=1}^{n} T_a(i) + \sum_{i=1}^{n-1} (n-i) T_r(i).$$

与 $T(0,n)$ 相比, 如果分析过程的耗时接近, 即 $\sum_{i=1}^{n} T_{pd}(i) \approx \sum_{i=1}^{n} T_d(i)$, 那么需要额外付出的时间开销则为 $\sum_{i=1}^{n} T_a(i) + \sum_{i=1}^{n-1} (n-i) T_r(i)$. 若基于快照方法来重新执行目标程序, 额外所需的时间开销则降低为 $\sum_{i=1}^{n} T_a(i) + \sum_{i=1}^{n-1} T_r(i)$. 但对于程序运行期间, 所耗费时间为 $T'_{on}(0,n) = T_{pd}(n) + \sum_{i=1}^{n-1} T_r(i) \ll \sum_{i=1}^{n} T_d(i)$. 因此, 渐进分析可以有效降低数据流分析对目标程序运行的影响, 从而可以进一步提高在线分析的能力.

1.3 总体架构

方法的具体执行包括 3 个主要部分, 如图 2 所示, 第 1 部分是对目标程序或者部分功能组件进行评估执行, 通过跟踪程序运行时的控制流, 发现可能存在的性能瓶颈及其所处的位置特征, 并且统计实际执行时间和指令数量,

可以指导后续对目标程序的分析过程进行划分. 第 2 部分就是对目标程序进行渐进式的分析和执行, 具体包括状态恢复、在线分析和信息记录、控制流分析以及恢复信息处理等任务. 状态恢复则将目标程序的分析状态恢复到上次状态保存处, 在分析执行阶段以动态污点分析为核心, 可采用按需插桩以及分析方式, 同时记录用于恢复和分析的运行时信息. 记录信息处理在程序每次运行完成后离线进行, 当重新执行目标程序后, 可根据处理提取的信息进行分析状态的恢复. 执行过程分析模块用来在程序执行后观察每次程序执行的逻辑是否发生变化, 以确认是否会影响分析或者恢复, 是否成功完成上次恢复, 以及是否存在对延迟敏感的代码等. 如果出现上述情况, 则需要调整针对目标程序的分析策略. 第 3 部分是进行离线数据分析, 基于在线分析所获取的数据做进一步处理, 可根据需要同时结合静态方法进行深入分析.

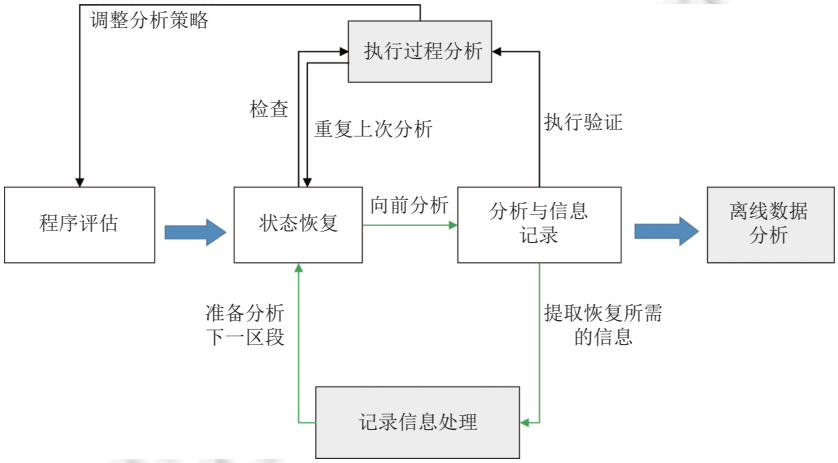

图 2 方法组成架构

1.4 执行过程分析

在实际环境中, 程序每次重复运行所生成的控制流可能会发生变化. 因此在出现这种情况时, 要能够判断控制流的变化是否会影响分析结果以及状态恢复过程. 在深度分析前可以通过轻量级地插桩分析来确定目标程序控制流变化规律. 而后, 可在程序往前推进分析过程中, 将当前的执行情况与上一次运行结果进行对比验证. 其中针对分析过程, 观察记录的指令序列以及受污染内存地址所关联的指令地址是否保持一致; 针对状态恢复过程, 观察待恢复分析状态的内存地址是否都能够准确进行获取, 以及线程是否能够执行到上次保存的位置处. 准确判断两次执行过程是否一致的方法是将执行的程序控制流进行比较, 我们称之为强验证方法, 但这会在分析过程中引入额外的信息记录性能开销, 比较折衷的方法是只针对记录的内存写操作指令序列进行比较, 称之为弱验证方法, 同样也能够反映目标程序的执行路径.

针对每区段的记录和验证过程可以重复进行, 以确定控制流的变化规律, 并尝试提取稳定的执行序列. 我们可以记录反映程序控制流的基本块 (basic block) 序列, 并通过文本相似性比较方法来获取不同执行所产生的差异区间. 依据假设及实际情况, 相同程序所产生的不同基本块序列的大部分是相同的, 因此可以通过不断寻找最长公共子序列的方法来进行两个序列的比较, 常用的是 Myers 方法[20], 该算法的时间复杂度为 $O(ND)$, 其中 N 为比较序列的长度, D 可认为是差异部分. 但是当目标序列很长且差异较大时, 该算法实际上将具有较长的执行时间, 难以在可接受时间内结束. 对于该情况, 尝试在分析时同时记录程序的 (call, next) 序列, 而后可以首先基于有层次的 call 序列对两个目标序列先进行一次对齐匹配, 而后再对剩余序列使用上述方法进行比较. 否则, 分别在缩小的局部范围进行比较. 这样算法的实际执行时间将得到极大的降低.

假设某线程的分析区段在执行时首先产生基本块序列 $(b_1 \ldots b_i b_{i+1} b_{i+2} \ldots b_n), 1 < i < n-2$, 并将其作为基准值, 而后在下次同样的执行中将产生另一序列 $\left(b_1 \ldots b_i b'_1 \ldots b'_k b_{i+2} \ldots b_n\right), k > 1$. 通过比较容易发现从基准序列的第 i 处

b_i 开始到 b_{i+2} 间的执行发生了变化, 我们将 (b_i, b_{i+2}) 称为变化的执行区间, 简称变化区间, 而其他部分则构成稳定的执行序列. 事实上, 除了第 i 处以外的其他位置的该区间也存在着潜在变化, 而前后相邻的变化区间也可以进行合并. 我们期望通过少量有限的执行和比较, 能够获取目标程序所有的执行变化区间, 从而能获取控制流的变化规律. 针对一些保持稳定的微小差异或者是明确的函数调用, 尤其是出现在系统库中的变化, 可以在记录时忽略引起差异的函数. 例如: 图 3 所示为同一程序的不同执行所产生的控制流比较片段, 可以跳过对 ntdll.dll!0x2ceba 所在函数的插桩. 另外, 在记录和分析时可忽略一些与分析无关的模块.

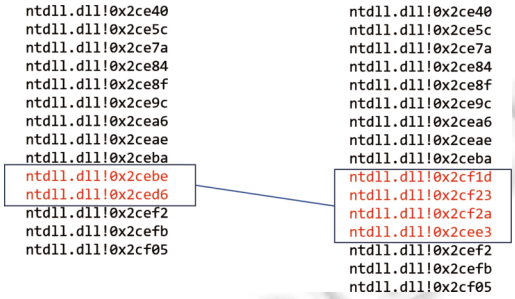

图 3 不同执行产生的基本块序列比较

1.5 多线程影响

目标程序多数情况下都是多线程程序, 其在执行过程中存在线程调度的影响, 不同线程间的代码执行顺序存在先后问题. 由于采用动态分析, 程序对线程间同步的正确实现能够保证每次运行结果的一致性. 但对于一些复杂程序, 尤其是有用户交互的大型应用程序, 会有几十个线程并发运行. 为了避免对无关线程的分析引入的额外开销, 针对这种情况, 在渐进分析的过程中, 可以逐步发现数据流相关联的线程, 选择性忽略不相关的线程.

多线程也会影响分析状态保存和恢复的过程, 在保存时需要保存每个线程的上下文信息, 主要是数据流传递过程中受影响的寄存器状态信息. 在恢复时同样要处理多个线程的问题, 增加问题的复杂度. 在恢复时, 由于是整个程序或者部分代码的另一次独立运行过程, 线程标识符 (thread idenfier, TID) 可能会发生变化. 此时要对保存的线程信息与运行时进行正确的关联绑定, 由于调度顺序的变化, 可能会出现新的线程, 要能够检测并且对其进行分析. 在恢复过程中, 对于原先出现过的线程, 首先执行状态恢复操作, 而对于新创建的线程, 则直接开始正常的数据流分析.

2 分析状态保存和恢复

2.1 状态保存点

状态保存点即为数据流分析过程与原目标代码执行进行分离的位置, 此时需保存当前的分析状态, 理论上可以在程序运行过程中任意的指令执行位置来保存状态. 保存点的确定主要取决于对程序进行渐进分析时所采用的分段策略. 较为通用的方法是根据程序执行的指令数量或者程序运行的时间, 其优点是不需要了解程序具体的执行逻辑. 前者是在分析过程中, 当程序执行一定数量的指令后进行状态保存, 通过指令计数来往前推进分析过程, 计数粒度包括指令粒度、函数调用或者基本控制块粒度. 而后一种策略是在程序执行一定的时间后对状态进行保存, 首先评估目标程序执行所需要的总时间, 并且检查性能瓶颈, 而后对其执行过程按时间进行划分. 当每个区间的保存点确定后, 后续的重复执行和分析将在此基础上, 依据稳定的执行流来进行分析状态的恢复.

在多线程情况下, 由于存在多线程调度, 可能有很多同步与等待的指令执行, 导致每次运行时的指令数量存在偏差. 为克服这种情况, 我们可以通过多次执行来确定稳定的执行序列, 合理选择保存点, 并可在各个线程的保存点附近选择特殊指令或者函数调用作为保存点标记, 有利于提高识别精度, 保持恢复过程的一致性. 比如: 将保存点设置在程序及其私有链接库的代码中, 而不是在容易被重复调用的库函数或者系统函数中.

在目标程序每次分析执行完成后, 可以根据实际运行反馈来重新调整渐进分析策略. 例如: 在分析过程中存在对延迟敏感的代码段, 此时需要对分析区间再进行分割, 直到延迟影响的消失. 可基于二分法来设计渐进步长调整

策略. 假设目标程序的位置区间 $[P_s,P_t]$ 存在延迟敏感代码, 一次性分析该区间将出现程序运行超时情况. 此时首先分析区间 $[P_s,(P_t-P_s)/2]$, 如果目标程序没有出现超时情况, 则可以当前步长继续往前分析直到结束, 即分析 $[(P_t-P_s)/2,P_t]$, 否则将首次分析步长减半, 即分析 $[P_s,(P_t-P_s)/4]$, 并同上依次类推等.

2.2 记录和保存内容

对目标程序进行数据流分析的过程中同时进行部分运行时信息的记录和保存, 以用于后续的分析状态恢复. 记录和保存数据的选择原则是避免引入显著的性能开销, 记录的内容包括分析时持续记录的信息, 以及分析退出时获取并保存的信息.

对于持续记录的信息, 当采用强验证方法时, 可以记录控制流信息以及污染新的内存地址的指令操作; 否则在程序分析过程中, 持续记录内存写操作信息. 因为对于用户模式下的程序代码, 只有发生内存写操作时, 才可能会引起分析状态传递到新的内存地址, 并在多数情况下, 写操作执行的数量相对较少. 考虑到信息完整度与日志大小的平衡, 具体记录内容包括指令地址、内存地址、操作大小、线程上下文、时间戳等. 内存操作信息按线程来区分进行输出, 每个线程采用独立的文件进行保存. 一个示例如图 4 所示, 其中, 污染状态表示涉及的内存地址在指令执行后的分析状态, 该状态信息取决于所采用的数据流分析引擎, 此处以动态污点分析为例. 时间戳可作为可选项, 用来辅助判断不同线程的指令执行顺序. 这样通过将这些信息记录下后, 在重新执行时就可以得知哪些指令的执行会引发实际的数据流传递. 另外, 在记录内存写操作时, 可以忽略一些清除污染状态的操作, 如: “mov dword ptr [esi+10h], 0”将常数值保存到特定内存地址中.

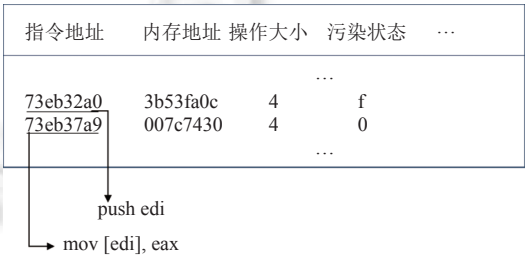

图 4 内存写操作记录示例

在分析过程结束时, 将记录的信息输出到本地文件中. 同时获取并保存当前状态下的各线程上下文信息, 包括线程实际启动地址、线程当前执行到的指令位置、寄存器分析状态信息等. 还需获取并保存当前进程的所有内存地址污染和标记信息, 只需保存被标记的内存地址. 例如: 采用字节级别的污点分析引擎时, 污点信息数组的每个字节可以表示 8 个内存地址状态, 只输出污点信息数组中比特为 1 所对应的内存地址信息. 污染内存地址信息属于进程层面, 保存时输出一次即可.

为避免大量的磁盘写入操作对目标程序影响, 我们对此进行优化处理, 即先将保存线程挂起, 等目标程序被分析的代码执行完成后, 再继续执行保存任务. 另外, 在线程数量过多时, 并发同时产生的记录数量会随之增加, 为降低开销还可采用优化方法, 比如: 一些地址可以通过静态方式来计算, 只记录上下文寄存器的值[13,14]. 本文方法当前重点关注动态获取信息与在线分析, 下一步将考虑在此基础上进行优化处理.

2.3 恢复元信息提取

在进行目标程序的下一阶段分析前, 需对保存的执行信息进行处理, 以提取用于分析状态恢复过程的插桩指令及执行统计信息, 本文称之为恢复元信息, 后续将基于该信息来进行分析状态的恢复过程. 我们在两次分析的间隔中以离线的形式来处理记录的日志, 以降低程序分析过程中的性能开销.

在分析状态恢复阶段, 对于每个被插桩的指令, 根据其执行计数来判断其执行进度, 以及根据元信息来确定在哪个执行阶段去获取相关的内存操作数地址并恢复其分析状态. 恢复元信息包括特定的指令地址、线程信息、指令在执行中出现待恢复分析状态的内存地址时的执行次数等. 其中, 指令地址采用相对偏移形式, 这是由于基址重定位机制会导致一些动态加载库每次加载的基地址位置不同. 如图 5 所示, 每个被插桩的指令拥有一个用于恢复

的执行次数统计列表, 以及涉及的变化区间列表. 括号外的数字表示在特定线程中当该指令执行第几次时, 恢复当前所访问的内存地址的分析状态. 括号内的数字为连续执行的恢复次数, 表示在此后的连续执行中所涉及的内存地址都要进行恢复. 例如: 7(49) 表示当 winhttp.dll!0x2733a 指令在执行第 7~55 次时, 获取运行时的内存地址并恢复分析状态. 在此之前, 我们需要确认目标程序在不同执行时的控制流变化情况, 并且提取稳定的执行序列和与状态恢复相关的指令. 因此, 上述指令的执行统计信息应是基于稳定执行序列的统计值. 进一步地, 如果目标程序在不同执行时的控制流变化较大, 并难以提取变化区间时, 可直接依据恢复信息中的指令及其计数信息来近似地判断是否可应用本文方法对目标程序进行分析.

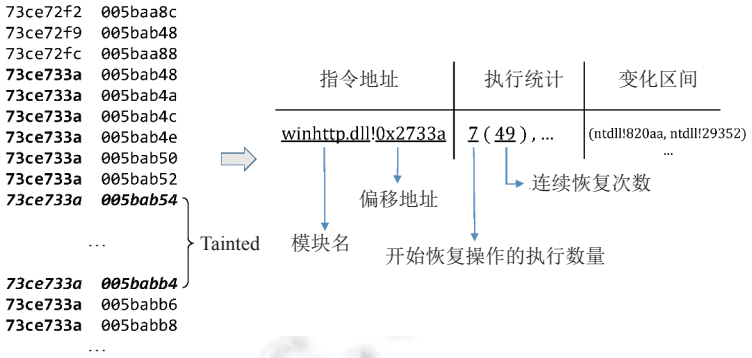

图 5 恢复元信息提取示意图

在对记录的信息进行处理时, 首先确定实际被污染或标记的内存地址所对应的指令, 选择适当的指令地址作为恢复时的插桩对象, 并统计其执行次数. 需要注意的是, 应当优先选择那些只出现稳定执行序列中的地址, 如上文所述. 在此原则下, 尽量选择最后出现的指令地址, 因为能够反映线程的最新执行状态, 并且避免内存地址释放后重分配的情况. 针对特定程序区间的分析和记录过程可进行多次, 以确认选择的指令具有稳定的执行计数.

同一指令地址可能会涉及不同的内存操作数地址, 因此需要进一步确定哪一次的执行涉及到污染或标记的内存操作数地址. 示例如图 5 所示, 左边是某线程的内存写操作序列片段, 右边是提取的用于恢复过程的插桩指令信息, 其中, 73ce733a 关联多个操作的内存地址, 但只有 005bab54~005babb4 是污染状态. 针对线程的内存分析状态恢复所需的元信息提取方法, 其核心部分如算法 1 所示, 利用哈希表通过一次遍历记录文件来提高数据的处理效率. 考虑到在多线程情况下, 同一个内存地址可能会在多个线程的执行上下文中出现, 因此可根据时间戳, 获取最近一次操作该内存地址的线程, 并只处理该线程涉及的地址, 忽略其他线程日志文件中的地址. 而对于相同指令地址关联多个内存操作数地址的情况, 则进行优化合并一如果相同指令地址处, 关联多个相同的内存操作数地址, 在状态恢复时只在该指令第 1 次执行时, 获取运行时的操作数地址并恢复其分析状态, 而后移除该指令处插入的用于状态恢复的插桩代码; 如果相同的指令地址关联不同的内存操作数地址, 但是其地址值是连续规律变化的, 可类似地对其进行合并处理.

对于线程在保存时所执行的指令地址的获取, 可在记录文件中获取其临近的记录信息, 作为线程上下文恢复标志指令并进行统计. 如上文所述, 内存操作指令序列与控制流类似, 也能够近似反映线程的执行路径. 在下一次执行时的恢复阶段, 也可以由此来判断当前线程是否已经执行到上次的保存点. 如果采用这种方式来表示线程的最新执行状态, 在信息处理时取每个线程记录文件中最后的若干指令作为线程上下文分析状态恢复完成的标志, 并进行分析统计. 在恢复时选择具有稳定执行计数的指令, 并对其进行插桩. 同时在信息记录时保存上述指令处目标线程的上下文分析状态.

算法 1. 恢复元信息提取.

输入: 线程对应的记录文件 *thread_log_file*;
输出: 恢复元信息列表 *res_list*.

```
1 procedure RestoreInforExtraction(thread_log_file)
2     创建指令地址字典 ins_dict, 内存地址字典 mem_dict.
3     for each line in thread_log_file do
4         提取指令地址 ins_addr.
5         根据变化区间计数, ins_dict[ins_addr]++.
6         提取 mem_addr 及具体操作, 与 ins_addr 和 ins_dict[ins_addr] 一起构建 mem_state.
7         mem_dict[mem_addr] = mem_state
8     for each mem_addr in mem_dict do
9         if mem_addr 不涉及恢复 then
10            continue
11        从 mem_dict[mem_addr] 中获取关联指令地址 ins_addr 及计数 pos.
12        将 pos 添加到 ins_dict[ins_addr].pos_list.
13    for each ins_addr in ins_dict do
14        pos_list ← ins_dict[ins_addr].pos_list
15        if pos_list 不为空 then
16            对 pos_list 进行排序, 合并连续地址.
17            将 ins_addr 转为相对偏移, 与 pos_list 一起保存到 res_list.
```

2.4 分析状态恢复

对已经分析过的执行代码则无需进行重复的数据流分析, 而仅需恢复其在上次状态保存处的分析状态即可. 当目标程序在分析状态恢复模式下执行到上次保存位置时, 再推进后续的数据流分析任务. 分析状态恢复过程主要涉及下面几个方面.

(1) 恢复内存分析状态. 在目标程序重新运行后, 由于受到地址随机化和内存动态分配的影响, 程序运行时访问的内存地址可能会与上次不同, 但如上文所述, 涉及内存操作的指令保持不变, 这些指令操作信息包含在提取的恢复元信息中. 在进行状态恢复时, 对恢复元信息中包含的指令进行插桩, 大多数插桩过程可在程序代码执行前完成, 该过程不会对程序产生较大性能影响. 当目标程序运行后, 在插入的分析代码中进行指令执行计数以及判断. 如果计数值匹配到元信息中的相应值后, 则获取此时涉及的内存操作数地址和操作大小, 并恢复其分析状态. 最后, 如果目标指令处关联的内存地址都完成状态恢复后, 则移除此处插入的执行恢复任务的代码. 另外, 在进行目标指令的执行数量统计时, 需要忽略其在变化区间里的执行. 因此我们针对每个线程的执行引入计数标记, 即当计数标记为打开状态时, 针对目标指令才进行计数. 如图 6 所示, ucrtbase.dll!0x537a9 为要进行执行计数的目标指令, 涉及到一个变化区间. 当目标线程的执行进入到该区间时, 计数标记将被清除, 在此期间的计数操作将会暂停. 因此, 如果恢复元信息中的指令地址涉及到变化区间, 还需额外的对变化区间的开始和结束指令进行插桩.

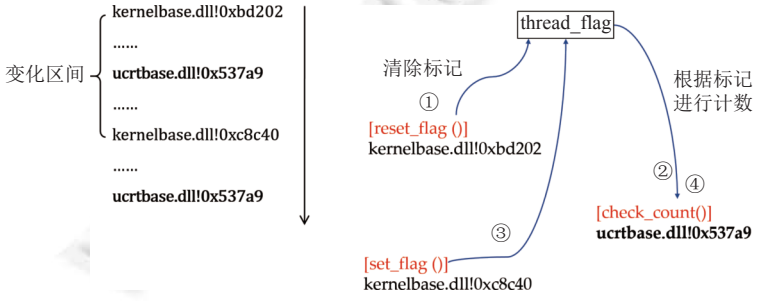

图 6 指令执行计数示意图

(2) 恢复线程上下文的分析状态. 该分析状态主要包括各寄存器的分析状态. 可以将线程最后记录的指令及其执行计数作为线程执行到上次保存点的标志. 如上节所述, 如果默认采用内存操作指令序列来表示线程执行轨迹, 则在分析时无需再插入额外的记录代码. 如果采用控制流信息来反映线程执行轨迹, 则在分析时还需要在每个基本块的前面插入代码来保存控制流信息, 这在增加精度的同时也会提高目标程序负载. 类似的还可以记录线程执行的函数调用序列. 与内存分析状态的恢复类似, 在进行恢复时对涉及的指令进行插桩, 并统计指令执行数量, 直到达到相应值. 而后恢复上下文涉及的寄存器的分析状态, 并对此线程后续执行的代码进行数据流分析.

(3) 线程识别与关联. 如果采用虚拟机快照方式来保存上次程序的执行状态, 则对于已创建的线程可以通过 TID 进行关联. 否则, 对于上个区段执行过程中新创建线程的情况, 线程 TID 和启动顺序可能会发生变化. 因此无法直接再通过 TID 进行恢复元信息与运行时信息的关联, 但主线程除外. 在此情况下, 可通过获取线程真实的起始地址来进行辅助识别, 例如: Windows 平台下可以通过系统调用 NtCreateThreadEx 以及库函数_beginthreadex 等函数来获取起始地址. 但是如果应用程序在线程管理上有自定义的实现, 则要首先了解或分析目标程序的线程管理机制, 以获取真实的启动地址, 例如: 线程池是常用的线程管理方法, 即多个线程执行相同的代码, 但实际执行的任务会变化, 此时需区分线程池代码和提交的任务代码. 对于无法通过线程起始地址进行区分的情况, 可通过线程启动后所记录的部分执行序列作为特征来进行识别关联.

3 数据流分析

本文所提出方法的重要特点是充分利用在线方式对目标程序进行数据流分析, 因此在常规动态污点分析的基础上可进一步扩展分析内容. 数据流分析过程基于动态污点分析方法, 由于采用了渐进式分析策略, 每次分析时对目标程序的性能影响得到有效降低. 动态污点分析可以复用已有的通用引擎, 在常用的在线污点分析方法中, 用一个比特来表示内存地址的污染状态, 因此在 32 位架构下, 进程虚拟空间大小为 4 GB, 污点标记映射内存最多需要占用 512 MB 的虚拟地址空间[7]. 污点传播过程涉及的指令主要包括逻辑运算指令 (如: add、adc、sub、mul 等)、数据传送指令 (如: mov、movsx、push、stos 等)、位运算指令 (如: and、or、xor 等) 以及其他常用指令. 在此基础上, 还包括下面的特别处理, (1) 针对目标程序的插桩及分析需区分线程进行. 如果执行指令的线程还未完成分析状态的恢复, 则先忽略对涉及指令的插桩, 即不在此指令之前插入分析函数, 可以避免重复分析; (2) 在进行污点分析的同时进行内存操作或其他信息的记录. 即在相关指令前还要插入额外的记录代码, 并按线程区分独立记录和保存数据; (3) 创建额外的辅助线程. 用于管理控制目标程序的执行, 评估线程执行时间以及输出记录信息等.

我们进一步提出基于内存引用关系的函数参数相关性分析方法. 在常规动态污点分析的基础上, 可对函数的局部参数结构及其关联的数据流传递信息进行分析. 一方面有利于在反编译等人工分析的基础上, 从函数层面了解潜在的污点传递的细节; 另一方面可以标记包含潜在污点数据引用的参数指针, 可有针对性地选择有价值的函数进行深入分析. 可以据此得知运行时参数的数据结构偏移情况, 以替代内存读写断点, 了解数据传递过程和依赖关系. 该方法的主要步骤如下.

(1) 首先利用渐进分析方法对程序进行分析, 并将分析推进到目标函数入口处. 当继续分析时, 在记录内存写操作的基础上, 同时记录内存读取操作和相关内存地址的污染状态. 针对其中涉及地址运算的指令, 如: mov、lea、add/adc、sub/sbb、div/idiv、mul/imul、shl/sal、shr/sar、xor 等, 同时获取实时寄存器的值并记录. 在目标函数分析完成后, 保存所记录的数据.

(2) 基于上述记录的信息, 构建内存地址引用关系图. 这是一个拓展的多重有向图, 定义为 $G=(V,E)$, 其中 V 为顶点集, 由内存地址及其存储的值组成, 顶点标签为具体的地址, 表示为 $LBL\_V(v)$. E 为边集, 包含两种类型的边, 一种称为关联边 E_{der}, 对于 $\langle u,v\rangle \in E_{\text{der}}$, 其连接的顶点 u 和 v 间存在指令运算关系, 即 u 通过运算和变换得到 v, 边的标签表示为 $LBL\_DER(u,v)=\{\text{timestamp},\text{transform}\}$. 其中, timestamp 表示指令执行时间戳, transform 为变换内容. 另一种称为引用边 E_{ref}, 其邻接点分别为内存地址及其指向的值, 即对于 $\langle s,t\rangle \in E_{\text{ref}}, [s]=t$, 边的标签表示为 $LBL\_REF(u,v)=\{\text{timestamp},\text{operation},\text{status}\}$, 其中, timestamp 表示操作进行的时间戳, operation 表示读写操作类型, 而 status 表示内存地址 u 的污染状态, 因此有 $E=E_{\text{der}}\cup E_{\text{ref}}$. 引用关系图的创建与有向图类似, 对于每一条记录

信息, 提取操作数值, 并作为顶点添加到图中. 确定有向边的方向和类型, 同时建立标签并添加到图中. 根据记录信息构建内存引用关系图时, 可准确锁定函数执行的指令范围, 从而去除其他指令信息的干扰.

(3) 基于图遍历来分析特定函数参数间的数据流传递情况. 主要思想是通过分析函数参数、函数返回值以及它们内部结构中引用的数据污染情况, 来确定该函数是否会存在对污点数据的处理过程, 并在此基础上获得涉及污染数据传递的内部结构信息. 目前暂未考虑函数内部引用全局变量的影响, 事实上, 全局变量也较容易识别.

首先根据反汇编和反编译等信息确定函数的参数和返回值位置, 获取对应的初始数值或者内存地址, 接着对引用关系图进行遍历, 并对函数参数直接或间接引用污点数据的情况进行标记. 具体的遍历标记算法如算法 2 所示, 基于深度优先遍历方法, 针对两种类型的边进行不同的处理, 对于 E_{ref} 中的写操作边, 选择最新状态的节点作为下一个访问的节点; 而对于读操作边, 则选择最早进行操作的节点下一个访问节点, 通过指令执行的时间顺序来获取最可能接近实际执行时的地址引用路径. 对于 E_{der} 中的边, 则优先选择时间戳较小的边的关联节点进行遍历. 我们将遍历路径中出现污染源读取操作的对应参数标记为{taint input}, 表示该参数会引入污染数据; 相反的, 将出现污染源写入操作的对应参数标记为{taint output}, 同时输出访问路径以及边标签信息. 通过算法可以得知, 在特定函数的参数中哪些参数会引入污染数据, 哪些参数会在函数执行完后关联上污染数据, 其中还包含了间接的污染地址引用关系. 对于相邻节点之间存在多条路径情况, 可以根据时间顺序确定实际有效的执行路径, 同时边的方向显示了内存地址引用关系的传递方向. 在算法中未将输入污染数据的参数和输出污染数据的参数进行更详细的关联, 如果需要关联则要在分析时同时记录更多的指令运行信息, 这将会提高程序运行负载, 导致分析次数的增加, 或者可采用离线方式来计算所有指令中的寄存器的值[13], 并结合人工方式做进一步分析.

考虑到分析时的负载问题, 如果分析当前整个函数导致程序无法正常运行, 则要对当前函数进行多次渐进分析, 方法同上文所述. 同时, 当前函数的参数结构和关系分析结果可以保存, 则在回溯分析上一个调用时, 可以忽略已分析过的子函数. 如果需要与子函数合并分析, 可根据子函数上次分析保存的状态, 恢复其内存分析状态即可, 无需再进行重复分析.

4 原型实现

我们在 Windows 平台上实现了本文所提出方法的原型, 采用二进制插桩方式来对程序进行分析. 其中, 二进制指令插桩工具采用的是 Intel Pin[21], 污点分析引擎采用的是 libdft 中的核心部分[7], 信息记录和代码分析部分采用 Visual Studio 2015 开发, 数据处理部分采用 Java 和 Python 开发, 目前只针对 32 位版本的程序进行分析.

针对内存访问操作的记录, 利用 libdft 提供的 ins_set_post 接口对内存读写操作的指令进行额外插桩, 并利用 INS_IsMemoryWrite、INS_IsStandardMemop 等函数进行过滤. 利用 Pin 提供的快速数据缓冲接口 (Fast Buffering API) 来记录数据, 能够在一定程度上降低性能开销. 该接口提供 INS_InsertFillBuffer 函数往缓冲区填充数据, 每个线程使用独立的缓冲区, 缓冲区大小可根据目标程序进行调整, 为避免频繁的文件写入, 缓冲区应设置较大. 为避免缓冲区满时发生文件写入操作, 从而影响程序运行, 我们利用 Windows 提供的 CreateFileMapping 函数实现进程间通信, 将数据复制到另一个进程中实现延迟写入. 另外, 为充分利用内存空间, 可在目标可执行文件头中增加/LARGEADDRESSAWAE 标记, 对 64 位平台上的 32 位程序开启大地址支持, 使得 32 位程序在用户模式下可以使用 4 GB 虚拟空间.

Pin 提供了 PIN_AddThreadStartFunction 接口用于对线程创建事件的拦截, 但无法获取线程的启动地址. 因此在线程创建事件的回调函数中, 我们通过 ntdll 模块中提供的未文档化函数 NtQueryInformationThread 来获取线程的启动地址, 同时对 C/C++运行时库中的线程创建函数_beginthreadex 进行插桩, 从而获取真实的线程代码执行地址. 对于其他封装线程管理函数的情况, 则需要预先进行单独分析. 对涉及数据流分析的部分系统调用进行插桩处理, 如 NtDeviceIoControlFile、NtWriteFile 等, 在处理例程中获取数据并进行标记. 针对采用异步 I/O 的情况, 还需对其完成例程进行插桩, 并通过文件句柄进行关联. 目前未对所有系统调用进行处理, 以及根据语义信息清除一些内存地址的污染标记.

算法 2. 函数参数污点传递标记.

输入: 内存引用关系图 *G*; 参数列表 *param_list; 返回值 ret_val*;
输出: 标记信息列表 *mark_list*.

```
1 procedure ParamsTaintMarking(G, param_list, ret_val)
2     for each p in param_list do
3       if p 是有效内存地址 then
4           在 G 中查找顶点 u, 其标签 LBL_V(u) = p
5           MARK_DFS_VISIT (G, u)
6       else if 引用 p 的内存地址或寄存器为污染状态 then
7           标记 p 为{input taint}, 并添加到 mark_list.
8       if ret_val 是有效内存地址 then
9       在 G 中查找顶点 u, 其标签 LBL_V(u) = ret_val
10       MARK_DFS_VISIT (G, u)
11 procedure MARK_DFS_VISIT (G, u)
12     if u 被遍历过 then
13         return
14     for each v in u 的邻接节点 do
15         if v 未被遍历过, 并且 ⟨u,v⟩ ∈ E_der then
16           if LBL_REF(u, v).operation 为 read then
17               取 LBL_REF(u, v).timestamp 最小的节点 v_min.
18           else if LBL_REF(u, v).operation 为 write then
19               取 LBL_REF(u, v).timestamp 最大的节点 v_max.
20     if v_min 和 v_max 都不为空 then
21         if LBL_REF(u, v_min).timestamp < LBL_REF(u, v_max).timestamp then
22           v_next ← v_min, 标记 v_next 为{input taint}, 并添加到 mark_list
23         else
24           v_next ← v_max, 标记 v_next 为{output taint}, 并添加到 mark_list
25     else if v_min 不为空 then
26         v_next ← v_min, 标记 v_next 为{input taint}, 并添加到 mark_list
27     else if v_max 不为空 then
28         v_next ← v_max, 标记 v_next 为{output taint}, 并添加到 mark_list
29     if LBL_REF(u, v_next).status 为污染状态 then
30         添加当前遍历的路径到 mark_list
31     else
32         MARK_DFS_VISIT (G, v_next)
33     for each v in u 的邻接节点 do
34         if ⟨u,v⟩ ∈ E_der then
35           取 timestamp 最小的节点 v_min
36           MARK_DFS_VISIT(G, v_min)
```

利用 Pin 提供的 IMG_AddInstrumentFunction 添加模块加载时的回调函数, 在回调函数中获取模块运行时的

基地址, 同时对恢复信息中涉及的指令进行插桩. 这里我们在目标指令处, 利用 RTN_CreateAt 创建一个虚拟例程, 而后用 RTN_InsHeadOnly 重新获取指令, 最后使用 INS_InsertCall 插入分析代码. 在模块加载前完成插桩过程, 而不是在代码执行时进行, 可适当提高效率. 使用 PIN_SpawnInternalThread 创建内部线程, 用于处理保存状态的输出过程, 线程创建后进入等待状态, 在分析完成后将保存的分析信息到文件当中. PIN_Detach 可用于分离分析代码与目标程序, 类似使用 PIN_Attach 继续对目标进行分析.

5 实验与结果分析

实验环境为安装有 Windows 系统的桌面电脑终端, 具体配置为 16 GB 内存, Intel Core i7-7700 @ 3.60 GHz, 系统盘为 120 GB SanDisk 固态盘, 数据盘为 1 TB/7200 转西部数据普通硬盘. 操作系统版本为 Windows 10 64 位, Pin 工具的版本为 3.4, 实验程序的开发和编译软件为 Visual Studio 2015, 分析的目标应用程序为 32 位版本. 在实验过程中, 首先采用真实的应用程序来分析在渐进分析方法下的性能开销情况, 而后结合多个实际案例分析来验证方法的有效性.

5.1 分析阶段的性能开销

理论上来说, 数据流分析阶段的性能开销与程序实际执行指令的数量成比例变化. 但是考虑到在进行常规数据流分析的同时, 还需要记录部分运行时信息以及控制分析进程, 这些操作可能会带来额外的程序性能影响. 在实验中选择多种程序来评估不同分析环境下的性能开销, 程序列表如图 7 所示 . 具体测试场景为: 分别使用 7-Zip (16.04) 和 WinRAR (5.30) 程序对一个 100 MB 大小的文本文件进行压缩, 压缩参数和格式为默认设置; FFmpeg (4.0.2) 则用来将 10 MB 左右的 WMV 转为 MP4 格式; 为测试代码插桩和分析过程切换的可行性, 进一步使用 Chrome (62.0.3202.62) 加载 10 MB 左右大小的文本页面, 其在执行时涉及多个进程; 使用 Office (2013) 内置的 PDF 转换功能将 20 MB 大小 Word 文档保存为 PDF 格式. 针对每种测试场景分别计算 4 种情况的程序性能开销: 一是程序在 Pin 环境下运行并且没有附加额外的分析代码; 二是对测试场景的整个阶段进行数据流分析; 另外两种都是对整个测试场景过程的前半部分进行数据流分析并且记录内存写操作信息, 而后程序将转为原生环境下运行, 其中一种是将分析阶段产生的记录信息进行输出, 另一种则不输出. 我们通过测试场景进行一半执行来衡量本文所提出方法的性能开销, 因为与单纯分析相比, 还有些附加操作的影响, 如: 数据记录、程序分离等. 测试场景的一半具体为: 对于 Chrome 是其接收网络传输数据的大小达到全部接收数量的一半. 对于其他程序, 则是在测试时输出文件的大小达到整个输出完成后大小的一半. 测试程序在原生环境下运行时, 上述半程运行情况下的时间消耗均为全部运行完成的一半左右. 因此能够直观的显示出额外的性能开销情况.

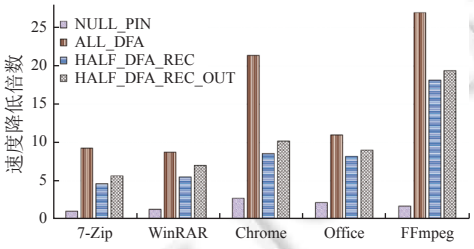

图 7 不同程序与不同分析情景下的性能比较

我们以程序正常执行测试场景所消耗的时间为基准值, 计算在分析执行环境下目标程序的运行速度降低比率. 结果如图 7 所示, 其中 NULL_PIN 表示仅使用 Pin 加载目标程序运行, ALL_DFA 表示在整个测试过程中进行数据流分析, 同样要采用 Pin 加载运行. HALF_DFA_REC 表示在数据流分析进行到约一半过程后, 分离分析过程和目标程序, 并且在分析的同时记录内存写操作信息. 而 HALF_DFA_REC_OUT 表示在缓冲区满时将保存输出记录信息, 其他情况与 HALF_DFA_REC 一样. 对于不同类型的程序, 均能够成功完成插桩, 信息记录和分析过程切换等过程, 并且从实验结果可以看出数据流分析对不同目标程序的运行有着不同的性能影响, 影响最大的是

Chrome 和 FFmpeg, 其页面加载的速度减缓了 21~25 倍多. 针对不同的程序, 在减半分析的情况下, 性能开销为全部过程分析的 0.47~0.81 倍左右, 会受到目标程序具体代码的影响. 而内存写操作信息的记录并未显著地减缓目标程序的运行, 其原因是数据流分析过程本身也需要对内存操作指令进行分析, 而记录功能仅是增加了部分额外的分析代码, 并在实现时采用了快速缓冲接口. 需要注意的是, 实验中进行记录的内存写指令数量要大于单纯数据流分析中插桩的数量, 此处采用的 libdft 引擎并未对所有的写指令进行分析, 如: 浮点指令. 另外可以看到记录信息的输出对目标程序存在一定的影响但并不显著, 主要是由于采用了进程间通信方式将文件写入转移到另外的进程进行. 从实验可知, 渐进分析可以显著降低对目标程序的性能影响, 并且额外的信息记录操作并未带来显著的性能影响.

实验采用的程序都是多线程程序, 在实际环境中属于比较常见的情况. 在实验过程中, 我们记录了上述应用分析过程中的资源平均占用情况. 如表 1 所示, 线程数量的增加会到导致内存使用量的线性增长, 32 位的虚拟地址无法满足所有记录信息的缓存要求, 从而容易在缓冲区满时引起日志信息输出. 同时结合图 7 来看, 程序负载并没有随着线程数量的增加, 而出现明显上升, 但是在线程数量增多的情况下, 不同线程间的同步问题可能会导致分析以及信息记录带来延迟的进一步累积增加. 表 1 中 Chrome 行中的加号是由于在对其进行分析时存在两个进程, 并要同时进行分析, 其中一个进程负责网络数据传输, 另一个负责页面渲染.

表 1　不同应用程序在减半分析时的资源占用情况

程序名	原始执行时间(s)	线程数量	内存占用 (MB)	记录块大小 (MB)	记录缓冲刷新次数
7-Zip	28	10	1 557	32	5 725
WinRAR	4	70	3 306	32	1 060
Chrome	7	49 + 20	2900 + 1560	32	1 133
Office	10	37	2 650	32	937
FFmpeg	6.5	24	2 233	32	5 494

下面进一步通过实验来分析采用不同的线程运行状态保存策略所带来的性能开销, 以及记录内存读取操作所带来的开销. 实验程序和测试场景与上面基本一致, 但只对整个测试场景过程进行数据流分析, 统计引入额外分析代码后的程序执行时间, 并且不输出记录信息. 以对程序单独进行数据流分析的耗时为基准值, 计算附加其他功能后的程序运行速度降低情况. 如图 8 所示, MW 表示在分析过程中对内存写入操作进行分析和记录; BBL 表示记录各个线程执行的程序基本块序列; CALL 表示记录各个线程执行的函数调用序列; CTX 表示在 MW 的基础上同时保存线程上下文分析状态; MR 表示在分析过程中对内存读取操作指令进行分析和记录. 采用 BBL 方式来记录线程执行状态是比较细粒度的, 但是也会导致较高的性能开销, 比较坏的情况下几乎是 CALL 方式的两倍. 通过记录函数执行序列或者通过内存写指令序列来标识线程执行状态比较合适, 拥有较好的性能开销. 同时, 从图中可以看到, 在进行数据流分析时记录内存读取信息, 并不会引起性能开销的明显增加, 给函数参数相关性分析方法的实施提供有利条件. 但是其性能开销取决于实际涉及的内存访问数量. 如: FFmpeg 涉及比较多的读操作.

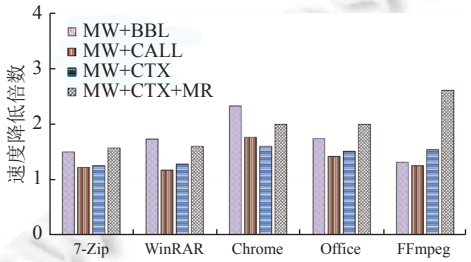

图 8　不同程序分析时附加性能开销对比

5.2　状态恢复性能开销

在分析状态恢复阶段有两个方面可能会影响目标程序的运行, 一是恢复元信息的加载和初始化过程. 如果元

信息中涉及的指令地址处于目标程序模块中, 则指令插桩过程在其启动前完成, 不对目标程序产生影响, 但如果指令地址处于程序运行后加载的动态链接库中, 则可能会对程序运行产生性能影响; 二是在内存操作指令地址处插入的完成恢复工作的代码在程序运行时产生的性能影响.

如图 9 所示, 插桩指令数量与时间消耗基本呈线性增长关系, 并且实际所占用的时间与数据流分析所消耗的时间相比可以忽略不计. 因为在实现时指令插桩过程在模块加载过程中进行, 通过模块哈希表来查找特定模块, 同一模块的地址使用双向链表进行链接, 遍历查找的时间跟节点数量成线性关系. 另外可以发现 Pin 在创建虚拟例程以及插桩过程中所产生的性能开销也可以忽略.

进一步地通过多个应用程序来观察恢复信息处理例程的性能开销情况. 选择不同类型的程序进行实验, 如图 10 所示. 为简化实验过程和突出重点, 继续选择部分上节实验场景 Office、FFmpeg、7-Zip 和 WinRAR 与上节测试场景一致, 在此基础上用网络程序 Curl(7.61) 从本地服务器中下载 gzip 格式压缩的 100 MB 文本, 另外 7-Zip_C 表示压缩, 而 7-Zip_X 表示针对同样大小的压缩文件进行解压, 测试时, 针对极端情况, 即对整个运行过程中的所有的内存写操作指令进行插桩分析, 这些指令数量大约占整个插桩指令数量的 25% 左右. 实际上, 涉及污染内存地址的指令地址数量会远小于此. 实验中以程序原始执行时间为基准值, 分别计算对整个阶段进行数据流分析以及执行恢复处理代码所带来的程序运行速度降低情况. 如图 10 所示, 在极端情况下, 单纯进行状态恢复的程序速度的降低程度, 要小于对整个过程进行数据流分析的情况. 降低程度主要受到实际执行的插桩代码数量影响, 另外, 模块加载时进行的插桩影响较小.

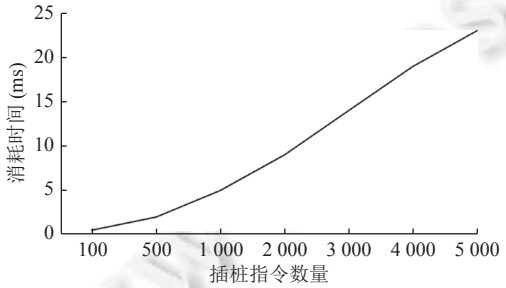

图 9 恢复时插桩指令数量与时间开销

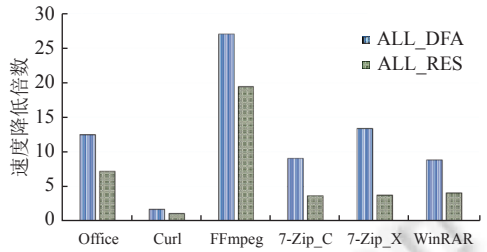

图 10 不同应用程序的状态恢复模拟性能开销

5.3 控制流变化的适应情况

我们进一步通过多种程序来评估各种控制流变化以及对本文方法的适用性影响. 为避免图形界面和无关线程的影响, 选择的测试程序列表如表 2 所示, 在实验中运行其基本功能, 并且记录主线程的基本块序列. 对于 WinRAR, 则记录其工作线程的执行序列. 依旧通过 Pin 来记录, 将执行的每个基本块开始指令输出为一行, 同时忽略 ntdll 模块中的 RtlAllocateHeap、LdrLoadDll 和 RtlEnterCriticalSection 等约 20 个函数的分析. 每个程序独立重复执行 10 次, 以第一次结果作为比较的基准值, 在分析比较时忽略进程的初始化和退出过程.

表 2 多种应用程序在不同执行中的控制流变化情况

程序名	序列平均长度	相同行数所占比例 (%)	平均差异部分数量	变化执行区间数量 (最小值/最大值)
Findstr	1 919 206	99.9	4	4/4
Certutil	725 494	99.9	33	2/33
Systeminfo	2 317 383	98.8	502	31/54
7-Zip	729 109	99.9	8	5/9
WinRAR	5 071 362	42.8	102 480	496/649
Curl	864 302	99.8	6	5/6
FFmpeg	8 881 022	99.9	66	17/40
Winver	2 279 329	99.1	158	90/194

如表 2 所示, 在多数情况下, 不同的基本块序列中相同行数占很高的比例. 对于差异部分数量较少的几个程序, 其引发差异的指令基本都位于不影响分析的系统库函数中, 如: msvcrt.dll!_woutput_l 等. 这种控制流的差异可以忽略, 适用于本文分析方法. 对于其他差异情况, 我们可以通过多次的执行来获取稳定的控制流信息. 如图 11 所示, 针对上述程序, 再经过少数几次执行后, 变化区间的数量很较快地收敛. 因此仍然可以应用本文的方法对类似程序进行分析.

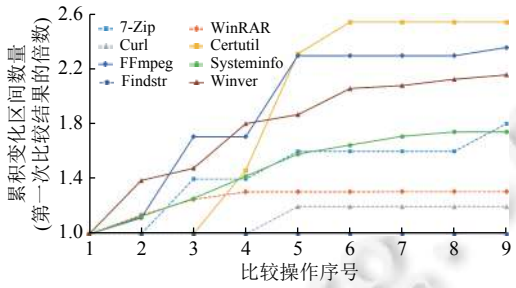

图 11 逐步比较后的变化区间数量累积变化情况

仍然有些特殊情况, 如: WinRAR 在执行时会有数十个工作线程, 线程间会有频繁的同步和调度, 因此在不同的执行下, 会引起大量的执行差异. 但是实际上引起差异的变化区间数量比较有限, 且集中在一个循环中. 这意味着较容易地总结执行变化规律. 而 Systeminfo 的不同执行也会产生较多的控制流差异, 因为系统环境的动态变化. 而类似的是 Winver 程序, 其主线程是 GUI 线程, 上节实验中出现的 Office 和 Chrome 程序也存在类似的情况. 对于此类程序, 我们可以根据需要去除图形相关模块的影响. 否则, 对于随时变化的图形化操作, 则难以应用本方法来进行分析.

5.4 真实程序的分析效果

我们通过多个真实的分析场景来说明所提出方法的分析效果和适用性. 与上述实验所用程序不同, 此处采用的程序涉及更明确的数据流分析场景, 包括多种类型, 具体的程序如表 3 所示, 其中, 使用 OpenSSL (1.0.2) 对大小为 1 KB 的文件进行 AES 加密; 使用系统自带的 Notepad 打开同样大小文件后, 另存为以 Unicode 方式编码的新文件. 上述两个场景都是在原文件读取时将其内容标记为污染状态, 而在新文件写入时进行污点标记状态检查. Curl 和 Powershell 分别用来从本地服务器中下载大小为 1 KB 的文件, 而用 Pscp (0.74) 来上传同样的文件. 而后我们分析名为 Trickbot 恶意程序的通信过程. Trickbot 在被安装后会在本地文件 client_id, 作为客户端的标识, 而后在与服务器通信时, 会读取 client_id 中的内容, 并作为 URL 中的内容发送到远程服务器. 与上面场景类似, 涉及网络发送接收数据以及文件读写的系统调用处, 将进行污染源设置和污点检查. 在采用渐进方法对上述场景分析时, 在检查点出现时保存分析状态. 最后测试 Firefox (56.0) 的网页保存功能, 在测试过程中我们发现, 如果程序分析过程带来的延迟较大, 可能会导致网页保存失败. 在保存网页时, 父进程同时负责网络通信和文件读写. 因此要分析其数据流传递情况, 只需对父进程进行分析即可. 这里我们重点分析其 Socket Thread. 当 Firefox 接收网页数据完成后, 开始进入数据流分析过程, 经过约 20 s 后保存分析状态. 在下一阶段则利用渐进方法继续往前分析, 而首先恢复到上次保存位置时, 时间消耗远低于上次分析过程, 大约需要 5 s. 上面场景的污点标记和检查也与上一段落中相同.

在进行分析时, 我们同时记录内存写操作序列, 并基于此来提取恢复元信息. 每种场景在进行恢复前重复进行 5 次分析, 以确定执行变化情况, 并在下一次分析过程中完成状态恢复. 在上述所有实验场景中, 分析状态都能够成功恢复, 并且在污点检查处能够检测到污点信息. 具体统计数据如表 3 所示, 在恢复时, 所需插桩的指令地址数量远远低于目标程序执行的数量. 表中涉及指令地址数为 5 次分析得到的恢复指令交集数量, 而括号中数量为多次分析中出现最多的数量. 从实验结果可以看出, 尽管部分程序在重复执行时产生的执行轨迹会有差异, 也产生较多的变化区间数, 但是实际影响分析状态的数量较少并且我们通过对变化区间的监测来消除其产生的影响.. 此

外, 恢复元信息的提取过程所耗费的时间也较少, 处于合理范围内.

在上述程序的分析中, 对于 Trickbot 我们同时保存了其主线程和网络线程的执行序列. 而 Notepad 和 Firefox 则都是具有图形化界面的, 不同的是 Notepad 的主线程即是窗口线程, 也涉及所要分析的功能. 当我们分析 Firefox 时, 只是在图形界面上增加少量点击操作, 而执行实际任务的线程不涉及图形窗口. 因此, 对 Notepad 的分析产生大量的界面相关的序列, 我们在后续处理时, 忽略了 dui70、gdi32 等涉及窗口渲染的模块, 使得其后续分析可行. 实验中采用的目标程序, 多数具有多线程和网络通信行为, 并且本文所述方法能够在一定程度上应对图形界面的影响.

表 3 多种程序分析状态恢复实验的相关数据统计

程序名	最少记录数量	变化区间总数	平均信息提取耗时(ms)	内存分析状态恢复		线程分析状态恢复	
				涉及指令地址总数	涉及变化区间数	涉及指令地址	涉及变化区间数
OpenSSL	5599	0	21	4 (4)	0	openssl!0x14c933	0
Curl	38 787	1	45	39 (56)	1	msvcrt!0x4acf7	1
Pscp	7540	0	38	2 (2)	0	mswsock!0x7692	0
Powershell	51 021	121	52	46 (49)	0	kernelbase!0xc9412	0
Notepad	5 893 822	670	3316	1 (1)	0	ntdll.dll!0x3950c	4
Firefox	99 355	232	154	5 (11)	9	xul!0x9cebc9	0
Trickbot	6 966 713	118	6035	7 (15)	11	bcryptprimitives!0x15480	10
	3 553 345	141	2525	5 (6)	18	crypt32!0x32ab8	0

5.5 数据流分析案例

SSL_Write 是 OpenSSL 库中提供的用于发送数据的接口, 在常规污点分析的基础上, 我们采用本文中提出的方法对其进行更详细的数据流分析, 分析其参数相关的污点传递信息. SSL_Write 原型为 SSL_write (SSL *s, const void *buf, int num), 参数 1 为 SSL 结构, 其中存放了 SSL 会话以及数据缓冲区等信息, 参数 2 是准备发送的数据指针, 参数 3 为发送长度. SSL_Write 实际上经过运行时绑定, 最终将调用 do_ssl3_write 函数, 但参数信息则保持不变.

进行数据流分析时, 在记录内存写操作的基础上同时记录内存读取信息, 并记录指针地址之间的运算关系, 记录 add、lea 指令的操作信息, 平均生成 3.6 万条记录. 而后构建内存地址引用关系图, 产生约 8 千个节点以及 1.5 万条边, 我们忽略了一些关系边, 其标签不同的仅是时间戳. 对于读操作, 保留了最早的操作记录. 对于写操作, 则保留最后的操作记录, 而后基于图进行路径的搜索与标记. 我们利用 neo4j 对部分路径进行可视化展示, 如图 12 所示, 参数 buf 仅有一条{input taint}路径, 反映了其作为输入数据的特点. 对参数 s 进行标记, 总共产生 85 条{output taint}路径, 以及 132 条{input taint}路径, 实际上多数路径拥有相同的前缀, 如图 11 所示, 路径 (0078f820-0078f878-0078fa88-0078fbb8-0078fbc4-00799d20-00000047) 与 (0078f820-0078f878-0078fa88-0078fbb8-0078fbc4-00799d20-cfd6f5fa). 对其进行合并后{output taint}和{input taint}路径各有 2 条, 并且输入和输出两类路径拥有共同的前缀节点. 如图 12 中所示, 地址 00799d20 处先后进行带污点标记的读和写操作. 实际上, 我们从 OpenSSL 的源码中可以得知, 在 do_ssl3_write 函数内部, 通过 s→s3→wrec→data 来引用等待发送的数据, 并经过加密后覆盖原来的明文数据, 从路径中可以体现这样的特点. 并且可以从路径中得知 s3 在 s 结构中的偏移值等信息. 因此, 基于内存地址引用关系图, 可以快速地了解参数间存在的污点传递关系, 包括指向污点数据的指针引用以及具体的数据结构偏移等信息.

6 讨 论

本文所提出的方法基于特定的前提, 即目标程序功能或者程序部分执行分支在多次独立运行过程中能够保持稳定. 在很多情况下, 分析场景能够满足该条件, 上文也通过许多真实案例来进行了验证. 但在面对一些代码庞大,

逻辑结构复杂的程序时, 仍然会存在挑战, 尤其是涉及用户界面以及随机因素的代码. 针对这种情况, 在进行实际分析时, 需要确认分析目标是否具有稳定的执行流, 从而判断是否能采用本文所提出的方法对其进行分析. 而对于执行随机性较强的程序, 以及多次执行却难以重现的程序行为, 则无法采用本文所提的方法进行分析, 但在渐进分析无法适用的情况下, 另外所提出的函数参数间的数据流分析方法也可在一些情况下发挥作用.

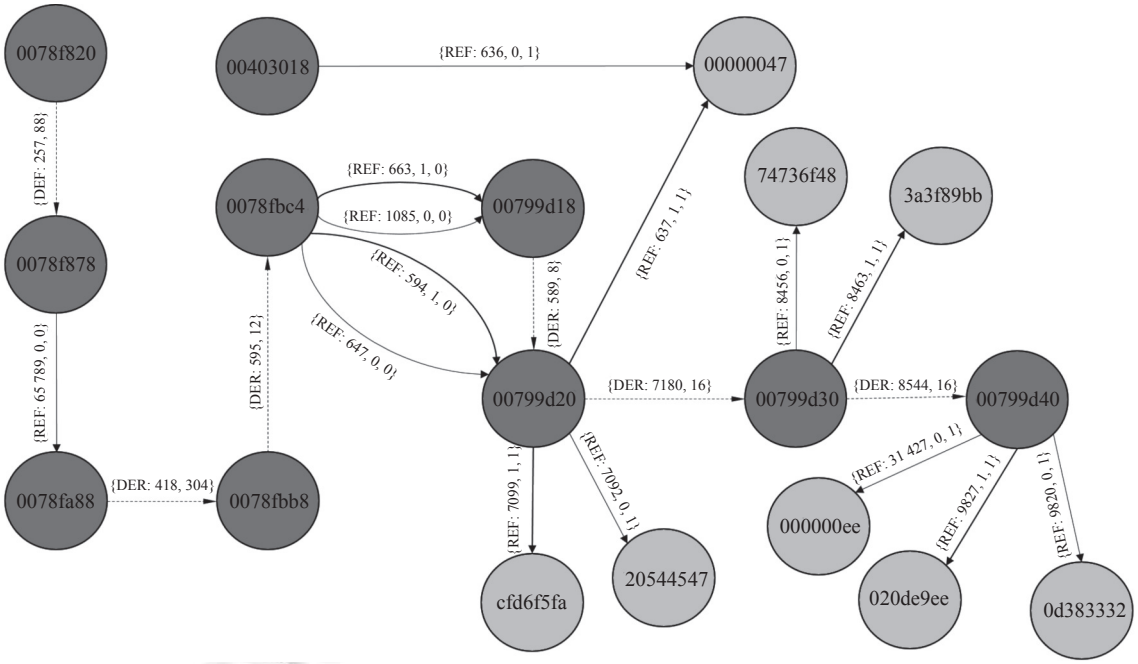

图 12 内存地址引用关系图局部展示

针对不同执行中出现的程序控制流变化情况. 由于系统库函数的影响比较大, 可根据实际情况灵活采取一些策略, 比如: 寻找引起控制流变化的库函数, 并忽略其插桩过程, 包括直接忽略特定的模块. 针对一些已知函数, 可以采用函数级别的分析策略[16], 并可以降低程序运行时的性能开销.

在进行原型实现时, 当前采用的是基于应用层的程序插桩方法来恢复程序的执行状态, 同样可以采用基于虚拟化的方法来完成. 在应对目标程序所采用的对抗与分析检测措施方面, 与传统分析方法类似, 本文所提的方法也面临同样的挑战[22]. 而对于分析引擎, 也可以采用其他的在线数据流分析引擎. 本文在 32 位环境下进行研究和实验, 进程能够使用的虚拟地址空间受限于 4 GB, 但在 64 位平台上, 将会有足够的虚拟地址空间来进行记录和分析. 在线程数量较多时, 所需的内存数量会较大, 需要采用更优化的方法进行信息记录和数据压缩. 另外对于运行时信息的记录和分析状态的恢复, 可以考虑利用基于硬件的方法来提高效率.

此外, 本文所提的方法当前主要面向用户模式下应用程序层面的动态分析, 不支持全系统范围的分析, 而且其在单次在线分析时的性能仍然不高, 在不同分析阶段衔接过程的自动化程度较低, 因此仍然存在一定的局限性, 后续需进一步提高在线分析的效率, 并探索与虚拟化技术相结合的渐进分析方案.

7 相关研究工作

已有许多基于动态数据流分析的研究工作, 广泛应用于恶意代码分析、漏洞检测、隐私数据泄漏检测等领域[9,23,24]. 典型研究方法包括基于系统范围动态分析技术的研究[25−28], 如: 基于 QEMU 等虚拟化技术来实现原型系统; 以及只针对二进制程序进行动态分析的研究, 如: 基于 DynamoRIO[29]、Pin[21]、Valgrind[30]等插桩工具来实现分析. 基于系统范围的动态分析方法涉及粗粒度和细粒度的分析, 粗粒度方法主要基于函数和内核对象级别来分

析程序行为. 典型的有, Lengyel 等人提出的动态恶意软件分析系统 DRAKVUF[5], Willems 等人设计并实现的恶意软件分析工具 CWSandbox[6]. 而细粒度的分析支持指令级别的数据流分析或污点分析, 能够更深入的了解数据在系统范围的传递过程, 如: Yin 等人提出全系统范围恶意代码检测和分析系统 Panorama[31], Henderson 等人提出基于 QEMU 虚拟机的全系统动态二进制分析框架 DECAF[32], Ji 等人提出精炼的攻击调查系统 RAIN[33]. 针对应用程序的动态分析研究重点聚焦于如何采用更好的设计策略来加速分析过程[7,9,16,24,34−36], 以及提高分析的准确程度[37−39]. 还有研究工作将动态分析和静态分析进行结合[30], 以及将程序分析过程与执行过程进行分离. 如: Ming 等人提出了并行化管道式的动态污点分析方法 TaintPipe[12], 以及基于离线符号执行的污点分析方法 Straight-Taint[13]. 与此类似, Jee 等人提出解耦加速的数据流分析方法 ShadowReplica[11]. Ganai 等人提出面向多线程程序的动态污点分析方法 DTAM[10], Zhu 等人提出 3 种状态的污点分析技术[40], Xiao 等人提出基于程序对象级别的数据流分析方法[41].

综上所述, 已有不少研究通过多种方法来提高动态数据流分析的精确度, 并且降低分析时的性能开销. 如: 将静态分析技术与动态分析相结合, 将在线与离线分析相结合, 将分析过程与程序执行过程进行并行完成等. 在具体实施方面, 基于系统范围的分析方法部署代价较高, 但分析范围更广, 可应用于分析 Rootkit 或者内核模块程序. 基于应用程序的分析方法在面临多线程大型应用程序时仍会存在性能挑战, 难以完全利用在线分析的优势, 同时依赖于静态分析的优化策略也会面临代码混淆, 代码动态生成等分析对抗措施的挑战. 本文从新的角度对二进制程序的数据流进行分析, 该方法基于在线的动态污点分析方法, 能够降低延迟敏感的代码执行对分析过程带来的影响, 最大程度地获取程序运行时的状态信息, 通过分阶段渐进分析目标程序来降低整个分析过程对程序运行的累积性能影响, 不依赖对目标程序进行静态分析来进行优化, 能够保持程序在真实环境下运行. 部分已有的分析引擎和动态分析的优化方法, 也可以应用于本文提出的分析方法中.

检查点和恢复方法在容错、调试等很多领域得到广泛研究和应用[42], 其中基于软件的技术通常在编译器或源代码层面进行实现, 基于硬件的技术则需特殊的硬件支持[43]. 由于是面向整个程序或系统环境的恢复, 因此在检查点保存的数据较多, 而本文提出的方法中涉及的信息记录主要是面向数据流或污点分析状态的恢复, 并不影响原程序和系统的正常运行. 实际上, 在恶意代码分析中更为常用的是虚拟机快照方法[44]. 本文所提出的分析方法同样可以基于上述方法来实现, 但在性能和部署灵活性上会受到影响.

还有较多研究工作基于记录和重放方法来进行程序分析和系统取证[33,45], 主要基于 QEMU 等软硬件虚拟化技术来实现[45,46], 如: PANDA[17], 其面向整个系统层面来记录中断、线程调度、网络、外部设备输入等各种事件并重现程序执行过程, 能够实现全系统范围的分析, 部分方法可支持指令级的程序分析, 其本质上属于解耦的离线分析, 因此同样对目标程序执行过程的影响较小, 并具有较高的运行时信息记录性能, 但在分析环境的构建和部署等方面不够灵活. 与此相比, 本文所提出的方法主要面向应用层的程序, 以在线分析为主, 并结合部分离线分析, 通过多阶段分析来克服对目标程序的运行时影响, 可快速地在目标系统环境中对程序进行重复分析, 但是在适用范围和检测对抗等方面仍存在局限性.

8 结 论

本文提出一种渐进扩展的二进制程序数据流分析方法, 旨在降低持续动态数据流分析对目标程序运行的影响, 从而使得对延迟敏感代码的分析能够成功完成. 同时充分采用在线分析的方法, 能够全面获取目标程序的运行时信息. 在此基础上, 进一步提出一种函数参数污点传递关系的分析方法, 从而可更加详细了解参数间的数据流传递和影响关系, 同时给逆向分析工程提供辅助.

本文通过对大量真实程序的案例分析研究, 验证了所提出方法的可行性, 并且通过多种复杂程序进行性能和功能的测试. 实验结果表明, 所提出的方法以及其实现策略, 能够显著降低分析对目标程序的影响, 并且能够用于多种分析场景. 下一步, 我们将完善在线分析与数据处理的自动化程度, 以及基于硬件的记录和恢复机制, 从而进一步提升分析效率和分析结果的准确性.

References:

[1] Alwarebvtes. Cybercrime tactics and techniques: 2017 state of malware. https://www.malwarebytes.com/pdf/white-papers/CTNT-Q4-17.pdf

[2] Egele M, Scholte T, Kirda E, Kruegel C. A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys, 2012, 44(2): 6. [doi: 10.1145/2089125.2089126]

[3] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[4] Ugarte-Pedrero X, Balzarotti D, Santos I, Bringas PG. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In: Proc. of the 2015 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2015. 659–673. [doi: 10.1109/SP.2015.46]

[5] Lengyel TK, Maresca S, Payne BD, Webster GD, Vogl S, Kiayias A. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Proc. of the 30th Annual Computer Security Applications Conf. New Orleans: ACM, 2014. 386–395. [doi: 10.1145/2664243.2664252]

[6] Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using CWSandbox. IEEE Security & Privacy, 2007, 5(2): 32–39. [doi: 10.1109/MSP.2007.45]

[7] Kemerlis VP, Portokalidis G, Jee K, Keromytis AD. Libdft: Practical dynamic data flow tracking for commodity systems. ACM SIGPLAN Notices, 2012, 47(7): 121–132. [doi: 10.1145/2365864.2151042]

[8] Jee K, Portokalidis G, Kemerlis VP, Ghosh S, August DI, Keromytis AD. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In: Proc. of the 19th Internet Society (ISOC) Symp. on Network and Distributed Systems Security. San Diego, 2012.

[9] Newsome J, Song DX. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the 12th Annual Network and Distributed System Security Symp. San Diego, 2005.

[10] Ganai M, Lee D, Gupta A. DTAM: Dynamic taint analysis of multi-threaded programs for relevancy. In: Proc. of the 20th ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering. Cary: ACM, 2012. 46. [doi: 10.1145/2393596.2393650]

[11] Jee K, Kemerlis VP, Keromytis AD, Portokalidis G. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security. Berlin: ACM, 2013. 235–246. [doi: 10.1145/2508859.2516704]

[12] Ming J, Wu DH, Xiao GY, Wang J, Liu P. TaintPipe: Pipelined symbolic taint analysis. In: Proc. of the 24th USENIX Conf. on Security Symp. Washington: USENIX Association, 2015. 65–80.

[13] Ming J, Wu DH, Wang J, Xiao GY, Liu P. StraightTaint: Decoupled offline symbolic taint analysis. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 308–319.

[14] Cui BJ, Wang FW, Guo T, Dong GW. A practical off-line taint analysis framework and its application in reverse engineering of file format. Computers & Security, 2015, 51: 1–15.

[15] Stamatogiannakis M, Groth P, Bos H. Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In: Proc. of the 5th Int'l Provenance and Annotation of Data and Processes. Cologne: Springer, 2014. 155–167. [doi: 10.1007/978-3-319-16462-5_12]

[16] Zhu D, Jung J, Song D, Kohno T, Wetherall D. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. ACM SIGOPS Operating Systems Review, 2011, 45(1): 142–154. [doi: 10.1145/1945023.1945039]

[17] Dolan-Gavitt B, Hodosh J, Hulin P, Leek T, Whelan R. Repeatable reverse engineering with PANDA. In: Proc. of the 5th Program Protection and Reverse Engineering Workshop (PPREW). Los Angeles: ACM, 2015. 4. [doi: 10.1145/2843859.2843867]

[18] Bauman E, Ayoade G, Lin ZQ. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. ACM Computing Surveys, 2015, 48(1): 10. [doi: 10.1145/2775111]

[19] D'Elia DC, Coppa E, Nicchi S, Palmaro F, Cavallaro L. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In: Proc. of the 2019 ACM Asia Conf. on Computer and Communications Security (Asia CCS2019). Auckland: ACM, 2019. 15–27. [doi: 10.1145/3321705.3329819]

[20] Myers EW. An *O*(*ND*) difference algorithm and its variations. Algorithmica, 1986, 1(1–4): 251–266. [doi: 10.1007/BF01840446]

[21] Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Notices, 2005, 40(6): 190–200. [doi: 10.1145/1064978.1065034]

[22] Polino M, Continella A, Mariani S, D'Alessio S, Fontana L, Gritti F, Zanero S. Measuring and defeating anti-instrumentation-equipped malware. In: Proc. of the 14th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). Bonn:

Springer, 2017. 73–96. [doi: 10.1007/978-3-319-60876-1_4]

[23] Espinoza AM, Knockel J, Comesaña-Alfaro P, Crandall JR. V-DIFT: Vector-based dynamic information flow tracking with application to locating cryptographic keys for reverse engineering. In: Proc. of the 11th Int'l Conf. on Availability, Reliability and Security. Salzburg: IEEE, 2016. 266–271. [doi: 10.1109/ARES.2016.97]

[24] Qin F, Wang C, Li ZM, Kim HS, Zhou YY, Wu YF. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In: Proc. of the 39th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Orlando: IEEE, 2006. 135–148. [doi: 10.1109/MICRO.2006.29]

[25] Wen Y, Zhao JJ, Chen H. Towards thwarting data leakage with memory page access interception. In: Proc. of the 12th IEEE Int'l Conf. on Dependable, Autonomic and Secure Computing. Dalian: IEEE, 2014. 26–31. [doi: 10.1109/DASC.2014.14]

[26] Zeng JY, Fu YC, Lin ZQ. PEMU: A pin highly compatible Out-of-VM dynamic binary instrumentation framework. In: Proc. of the 11th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. Istanbul: ACM, 2015. 147–160. [doi: 10.1145/2731186.2731201]

[27] Wang CW, Shieh S. SWIFT: Decoupled system-wide information flow tracking and its optimizations. Journal of Information Science and Engineering, 2015, 31(4): 1413–1429. [doi: 10.6688/JISE.2015.31.4.15]

[28] Lovat E, Fromm A, Mohr M, Pretschner A. SHRIFT system-wide HybRid information flow tracking. In: Proc. of the 30th IFIP TC 11 Int'l Conf. on ICT Systems Security and Privacy Protection. Hamburg: Springer, 2015. 371–385. [doi: 10.1007/978-3-319-18467-8_25]

[29] Sullivan GT, Bruening DL, Baron I, Garnett T, Amarasinghe S. Dynamic native optimization of interpreters. In: Proc. of the 2003 ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators. San Diego: ACM, 2003. 50–57. [doi: 10.1145/858570.858576]

[30] Nethercote N, Seward J. Valgrind: A program supervision framework. Electronic Notes in Theoretical Computer Science, 2003, 89(2): 44–46. [doi: 10.1016/S1571-0661(04)81042-9]

[31] Yin H, Song D, Egele M, Kruegel C, Kirda E. Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proc. of the 14th ACM Conf. on Computer and Communications Security. Alexandria: ACM, 2007. 116–127. [doi: 10.1145/1315245.1315261]

[32] Henderson A, Prakash A, Yan LK, Hu XC, Wang XJW, Zhou RD, Yin H. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 248–258. [doi: 10.1145/2610384.2610407]

[33] Ji Y, Lee S, Downing E, Wang WR, Fazzini M, Kim T, Orso A, Lee W. RAIN: Refinable attack investigation with on-demand inter-process information flow tracking. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 377–390. [doi: 10.1145/3133956.3134045]

[34] Ma JX, Li ZJ, Zhang T, Shen D, Zhang ZK. Taint analysis method based on offline indices of instruction trace. Ruan Jian Xue Bao/Journal of Software, 2017, 28(9): 2388−2401 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5179.htm [doi: 10.13328/j.cnki.jos.005179]

[35] Bosman E, Slowinska A, Bos H. Minemu: The world's fastest taint tracker. In: Proc. of the 14th Int'l Symp. on Recent Advances in Intrusion Detection. Menlo Park: Springer, 2011. 1–20. [doi: 10.1007/978-3-642-23644-0_1]

[36] Cheng W, Zhao Q, Yu B, Hiroshige S. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In: Proc. of the 11th IEEE Symp. on Computers and Communications. Cagliari: IEEE, 2006. 749–754. [doi: 10.1109/ISCC.2006.158]

[37] Clause J, Li WC, Orso A. Dytan: A generic dynamic taint analysis framework. In: Proc. of the 2007 Int'l Symp. on Software Testing and Analysis. London: ACM, 2007. 196–206. [doi: 10.1145/1273463.1273490]

[38] Kang MG, McCamant S, Poosankam P, Song D. Dta++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the Network and Distributed System Security Symp. San Diego, 2011.

[39] Wang XF, Ma HT, Jing LS. A dynamic marking method for implicit information flow in dynamic taint analysis. In: Proc. of the 8th Int'l Conf. on Security of Information and Networks. Sochi: ACM, 2015. 275–282. [doi: 10.1145/2799979.2799988]

[40] Zhu EZ, Liu F, Wang Z, Liang AL, Zhang YW, Li XJ, Li XJ. Dytaint: The implementation of a novel lightweight 3-state dynamic taint analysis framework for x86 binary programs. Computers & Security, 2015, 52: 51–69. [doi: 10.1016/j.cose.2015.03.008]

[41] Xiao GY, Wang J, Liu P, Ming J, Wu DH. Program-object level data flow analysis with applications to data leakage and contamination forensics. In: Proc. of the 6th ACM Conf. on Data and Application Security and Privacy (CODASPY2016). New Orleans: ACM, 2016. 277–284. [doi: 10.1145/2857705.2857747]

[42] Bronevetsky G, Fernandes R, Marques D, Pingali K, Stodghill P. Recent advances in checkpoint/recovery systems. In: Proc. of the 20th IEEE Int'l Parallel & Distributed Processing Symp. Rhodes: IEEE, 2006. 8. [doi: 10.1109/IPDPS.2006.1639575]

[43] Li T, Shafique M, Ambrose JA, Henkel J, Parameswaran S. Fine-grained checkpoint recovery for application-specific instruction-set

processors. IEEE Trans. on Computers, 2017, 66(4): 647–660. [doi: 10.1109/TC.2016.2606378]

[44] Cui L, Wo TY, Li B, Li JX, Shi B, Huai JP. PARS: A page-aware replication system for efficiently storing virtual machine snapshots. In: Proc. of the 11th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments (VEE2015). Istanbul: ACM, 2015. 215–228. [doi: 10.1145/2731186.2731190]

[45] Chow J, Garfinkel T, Chen PM. Decoupling dynamic program analysis from execution in virtual environments. In: Proc. of the 2008 USENIX Annual Technical Conf. (ATC). Boston: USENIX Association, 2008. 1–14.

[46] Ren SR, Tan L, Li CQ, Xiao Z, Song WJ. Leveraging hardware-assisted virtualization for deterministic replay on commodity multi-core processors. IEEE Trans. on Computers, 2018, 67(1): 45–58. [doi: 10.1109/TC.2017.2727492]

附中文参考文献:

[3] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展. 软件学报, 2019, 30(1): 80–109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[34] 马金鑫, 李舟军, 张涛, 沈东, 章张锴. 基于执行踪迹离线索引的污点分析方法研究. 软件学报, 2017, 28(9): 2388–2401. http://www.jos.org.cn/1000-9825/5179.htm [doi: 10.13328/j.cnki.jos.005179]

潘家晔(1985－), 男, 博士, 主要研究领域为系统与软件安全, 网络安全.

孙炳林(1994－) 男, 硕士, 主要研究领域为信息安全.

庄毅(1956－), 女, 教授, 博士生导师, CCF 专业会员, 主要研究领域为信息安全, 网络, 分布计算.