分类号 TP309 密 级
U D C 编 号 10486

武漢大學

博 士 学 位 论 文

云计算环境下跨虚拟机侧信道的攻击、检测与防御

研 究 生 姓 名: 刘维杰

指导教师姓名、职称: 王丽娜 教授

学 科、 专 业 名 称: 信息安全

研 究 方 向: 虚拟化安全

二零一八年五月二十九日

Research on Cross-VM Side Channel Attack, Detection and Defense in Cloud Environment

Candidate: Liu Weijie

Supervisor: Prof. Wang Lina

Major: Information Security

Speciality: Virtualization Security

School of Cyber Science and Engineering

Wuhan University

MAY 29, 2018

论 文 原 创 性 声 明

本人郑重声明: 所呈交的学位论文, 是本人在导师指导下, 独立进行研究工作所取得的研究成果. 除文中已经标明引用的内容外, 本论文不包含任何其他个人或集体已发表或撰写的研究成果. 对本文的研究做出贡献的个人和集体, 均已在文中以明确方式标明. 本声明的法律结果由本人承担.

学位论文作者 (签名):

年　　月　　日

武汉大学学位论文使用授权协议书

本学位论文作者愿意遵守武汉大学关于保存、使用学位论文的管理办法及规定, 即：学校有权保存学位论文的印刷本和电子版, 并提供文献检索与阅览服务；学校可以采用影印、缩印、数字化或其它复制手段保存论文；在以教学与科研服务为目的前提下, 学校可以在校园网内公布部分及全部内容.

1、 在本论文提交当年, 同意在校园网内以及中国高等教育文献保障系统 (CALIS)、高校学位论文系统提供查询及前十六页浏览服务.

2、 在本论文提交 □ 当年/ □ 一年/ □ 两年 / □ 三年以后，同意在校园网内允许读者在线浏览并下载全文，学校可以为存在馆际合作关系的兄弟高校用户提供文献传递服务和交换服务.(保密论文解密后遵守此规定)

论文作者 (签名)：_____

学　　号：_____

学　　院：_____

日期：　　年　　月　　日

论文创新点

在云计算被广泛运用的今天，其上的侧信道安全威胁也逐渐凸显。国内乃至国际范围内对跨虚拟机的侧信道安全研究尚不充分。本文在跨虚拟机侧信道方面率先进行了从攻击到检测到防御的全方位的研究。

本文的主要创新点在于利用现有云平台的设计缺陷，针对 CPU、缓存、内存等硬件充分研究，优化了跨虚拟机的侧信道和隐蔽信道，并以此进行了对真实云环境的同驻攻击；充分结合硬件虚拟化技术和虚拟机自省技术，提出了多项跨虚拟机侧信道的检测和防御方案。本文包含跨虚拟机侧信道一系列问题的充分阐述，是对云计算、虚拟化安全领域进行了重要补充。具体创新点如下：

在跨虚拟机时间信道攻击方面，本文提出了基于最外层缓存 (Last Level Cache, LLC) 和基于共享内存总线的侧信道优化方案。此优化方案能够克服虚拟化环境中信道构建的困难，将信道载体所具备的特性充分利用，以达到实际环境下该侧信道可用于实施攻击的目标。

在虚拟机同驻方面，本文提出了一种基于后验概率的自动化虚拟机洪泛方案。以所构建的侧信道攻击为基础，实现了基于隐蔽信道的同驻检测方案。方案总体思路为先大量投放虚拟机，再对所投放的虚拟机进行同驻检测的分类，正确率高且耗时少，在实际应用中成功实施，并在行业应用中进行了验证。

在跨虚拟机的隐蔽信道检测方面，针对现有方案各自为战，检测不通用等问题，本文提出了一种通用的检测方法。采用事件关联的方式检测隐藏的信道，检测率高；所提出的框架扩展性好，具有广泛适用性；此方法并不采用信息流分析的方法，故不必担心状态空间爆炸问题；系统能够做到自动分析，且具备自反馈机制。

在云环境中侧信道威胁定位方面，利用硬件计数单元 PMU 和分支记录器 LBR 的特性，本文提出了一种定位方案，成功检测了多种侧信道威胁；结合虚拟机自省技术和内存二进制内容分析技术，利用获取的 LBR 信息，对侧信道威胁进行了精确的定位。该方案通过语义还原，能够做到指令级的威胁定位和攻击溯源，同时硬件软件相结合，使得在保证高效性的同时性能开销较小。

在跨虚拟机的信道防御方面，针对现有方案引入较大的性能开销等缺陷，本文提出了一种防护方案，能够动态按需地将攻击虚拟机所获得的时间信息进行模糊。本文所提方案针对适用于所有以 RDTSC 为计时方案的侧信道攻击，方法通用；使用 VMFUNC 作为方案的启动接口，动态灵活且引入的开销小，使该方案的广泛应用成为可能。

目　　录

摘　要

云计算的概念自提出以来，已经过去十多年。目前，国内外的商用云计算业务都已进入蓬勃发展的重要时期，云计算也越来越多地被视为存储数据和部署服务的 IT 基础设施。然而，由于云计算的商业服务模式，导致云平台天生就难以抵抗共享计算资源所带来的安全威胁。其中，跨虚拟机侧信道威胁首当其冲。由于云环境具有虚拟隔离、边界模糊、租户共享等特点，允许其上的同驻虚拟机共享物理主机的大部分资源，因此恶意的虚拟机可通过侧信道攻击破坏云平台中数据的机密性和资源的可用性，导致严重的安全问题，对大规模企业用户和普通云租户都会造成极大危害。本文以保护云平台虚拟机不受时间信道攻击为主线，阐述了相关研究背景和基础知识；重点从跨虚拟机侧信道攻击出发，研究商用云平台中虚拟机同驻方案，以指导之后的避免同驻的研究；利用虚拟化技术与事件关联算法对平台中存在的时间信道进行检测；利用虚拟机自省技术和 CPU 硬件特性对侧信道威胁进行定位；最后利用 CPU 硬件特性对跨虚拟机的时间信道进行防御。本文包含跨虚拟机侧信道一系列问题的充分阐述，是对云计算、虚拟化安全领域进行了重要补充。具体内容包括：

(1) 本文提出了适用于云环境的侧信道攻击构建和优化方案。基于微处理器架构的时间侧信道在多用户计算机系统中十分常见。不过在嘈杂的生产云环境中，虚拟机迁移、虚拟处理器的调度以及虚拟化管理器的活动对时间侧信道增加了许多噪声，对信道构建和精确同步带来挑战。

本文研究了两种典型的跨虚拟机时间信道：基于最外层缓存的和基于内存总线的侧信道，并给出其构建与优化方案。此方案能够克服这两种虚拟化环境中信道构建的困难，将信道载体所具备的特性充分利用，大大提升了信道传输正确率，在实际云环境中可以用于攻击。

(2) 本文提出了一种普适的虚拟机同驻方案。如果外部攻击者意图实施针对远程云平台的攻击，则它必须首先完成与目标虚拟机的同驻。

通过深入研究某云的商业策略和服务协议和一系列实验，本文对某云的内部部署结构进行了探测，提出了一种基于后验概率的自动化虚拟机洪泛策略。通过该策略能够得出洪泛虚拟机群的同驻情况与目标虚拟机分布到各台物理主机的概率，以便于减

少之后所需同驻攻击的实验成本和开销，能够对攻击者的下一步攻击作出指导。本同驻方案结合基于隐蔽信道的虚拟机同驻检测方法和自动化虚拟机洪泛策略，在国内知名商业云平台上进行了实验验证。作为一种典型的针对云平台的恶意行为，本方案通用性强，误检率不超过 0.5%，同时鲁棒性强，潜在威胁极大，亟需各大云服务提供商重视与防范。

(3) 本文从资源共享的角度，提出了一种云计算环境下的隐蔽信道通用检测方案。隐蔽信道是安全信息系统的重要威胁，且普遍存在于操作系统中。云计算环境中的隐蔽信道形式更加多样且难以被发现。因此本文构建了针对隐信道的普适性的隐蔽信道检测模型，使其既可检测系统级的隐蔽信道，也可以检测进程级的隐蔽信道。

从时钟、事件和隐蔽信道之间的两两关系来看，事件显然具备作为传输秘密信息的能力，事件关联分析将是隐蔽信道检测本质上的有效解决方案。本文充分考虑了虚拟化架构与普通计算环境的区别，通过事件记录机制探寻云环境下隐蔽信道的特征，结合共享资源矩阵法遍历和搜索可能存在的隐蔽信道。同时，所设计的检测方案充分考虑了虚拟化环境的特点，在信息收集阶段采用了穿透性较强的方式来收集必要的事件日志信息和安全策略信息；改进了原有的共享资源矩阵法，对事件日志和安全配置文件进行预处理，减少了构建共享资源矩阵工作量；最后根据实际应用场景，设计了仿真实验，以此验证了本文所提方案的优势。

(4) 本文提出一种侧信道威胁定位方案。为了克服云环境跨虚拟机侧信道威胁定位方法匮乏、现有威胁定位技术不精确等挑战，本文设计了基于硬件特性和虚拟机自省的定位框架来实时的检测和定位云中存在的跨虚拟侧信道攻击。

现有 CPU 中包含许多用于性能分析的硬件特性，本文将其利用，对大量侧信道威胁进行了学习。通过测量它们在性能方面 (如 Cache 命中，分支转移，运行时间) 的表现，多维度地分析了不同的侧信道，并定制了不同的策略来进行基于特征的检测。同时，利用分支记录器 LBR 的精确地址信息，通过引入虚拟机自省技术，本文能够将侧信道威胁的元数据获取并进行语义翻译。在此基础上，本文结合二进制内存分析技术，能够以最小的先验知识，还原出侧信道威胁的精确地址，从而进行攻击溯源。本方案将检测和定位有机结合，利用 Hypervisor 已有的信息传递机制，构建了信息分析和分发的快速通道，解决了云环境下虚拟机信息收集困难的挑战。其原型系统尽可能地借助已有架构，将虚拟机自省工具和内存分析工具集成在一起，能够快速透明地找到内存中的关键恶意代码。本方案率先解决了虚拟机中侧信道攻击定位困难的问题，对虚拟机审计提供了帮助。

(5) 本文提出了一种侧信道防护方法。随着新的虚拟机侧信道的出现，研究人员提出了一些针对性强的方法来缓解侧信道威胁。非随机地将敏感操作的运行时间填充成固定时间或固定时间的倍数也许是一种较好的方式。然而，现有的缓解方法引入了很多刻意的时间延迟，并且要求用户始终忍受这种性能损失。这些方法一方面会篡改时钟，影响许多依赖于细粒度时钟源的应用程序的运行。另一方面，长时间的修改系统时钟也会对系统性能造成极大影响。

本文利用硬件虚拟化扩展技术，提出一种轻量而有效的方法，以实现云平台的整体侧信道防护。根据客户虚拟机所提出的要求，允许虚拟机动态地请求模糊时钟源；通过 Intel VT-X 提供的 RDTSC 指令截获功能，来模拟虚拟的时钟源并提供给用户；通过新颖的方式重载了 Intel CPU 中的 VMFUNC 指令接口，允许虚拟机上的应用程序向 Hypervisor 发送按需的请求，以 “恰到好处” 地程度来模糊 TSC 的最低 n 位，以阻止其他恶意的同驻虚拟机对时间进行精确测量。本方案同样具有通用性与极强的可用性。本文以所实现的侧信道攻击为例，展示了本方案对多个隐蔽信道的防御效果，同时得出了在这些攻击场景中客户虚拟机的 TSC 值应该被模糊的程度。在性能分析方面，本文对三种不同工作负载的实验表明，本方案在加密应用中的额外开销很小；并且在云环境中最为普遍的 Web 服务器应用中，具有比现有其他防御方法更低的性能开销。

总而言之，本文从跨虚拟机侧信道的攻击、检测和防御等多个方面对这一云环境中的新型威胁作了系统性的研究，对攻防两端的方法均作出了创新性的贡献。通过结合处理器硬件特性与虚拟化技术，以通用性和透明性为出发点，提出了对跨虚拟机侧信道检测、定位和防御方案，为该方面问题的解决提供了新思路。本文是对现阶段云计算安全和虚拟化安全研究的进一步丰富和创新，对增强云平台的安全性有重要意义。

关键词: 云计算安全, 跨虚拟机侧信道, 虚拟机同驻, 事件关联, 虚拟机自省, 动态时间模糊

ABSTRACT

Since the concept of Cloud computing is proposed, it has been a decade. At present, the worldwide commercial cloud computing business has entered an important period of vigorous development. Meanwhile, cloud computing is gradually regarded as the IT infrastructure for storing data and deploying services. However, due to the characteristics of multi-tenant dynamic aggregation and boundary decentralization, cloud computing platform is inherently hard to resist the security threats brought by sharing computing resources between virtual machines (VM). Among them, cross-virtual machines side channel threats are the first to bear the brunt of a co-located attacker. Because of "virtual isolation but physical coexistence" characteristics in cloud which allows virtual machines to share most of the resources in the same physical host, malicious virtual machines will destroy data confidentiality and resource availability, causing serious security problems, which will do great harm to large-scale business users and ordinary cloud tenants. This thesis focuses on protecting the virtual machine from timing-based side channel attacks on the cloud platform, and elaborates the related research background and basic knowledge. Aiming on cross-virtual machine side-channel attacks, the thesis studies the virtual machine co-resident solution in commercial cloud platform to guide the later avoidance; makes use of hardware-assited virtualization technology and event correlation algorithm to detect the presence of the timing covert channel; takes advantage of virtual machine introspection and CPU hardware features to locate side-channel threats; Finally, leverages CPU hardware features to build a dynamic time blurring system to support side channel defense. The specific contents include:

(1) Construction and optimization schemes for side channel attacks in cloud have been proposed. Side channel based on microprocessor architecture is very common in multi-user computer systems. However, in a noisy production cloud environment, only one branch of side channels exists — the timing-based side channel. Virtual machine migration, vCPU scheduling and Hypervisor activities adds a lot of noise to this "only"

timing-based side channel and poses challenges to channel construction and precise synchronization.

This thesis studies two typical cross-VM timing channels: the Last Level Cache-based channel and the memory bus contention-based channel, and gives a series of construction and optimization schemes. These schemes can overcome the difficulties of channel construction in virtualization environments, making full use of the characteristics of the channel carrier, greatly enhancing the accuracy of channel transmission, and can be adopted for attacks in the real cloud environment.

(2) A comprehensive virtual machine co-residency scheme has been proposed. "For workers to be good, must first sharpen." If an attacker intends to implement an attack on a cloud computing platform, he/she must first make the attacker virtual machine co-located with the target virtual machine.

Through in-depth study of a commercial cloud business strategy and service agreements and a series of experiments, the thesis probes the internal deployment structure of it and proposes a automatic VM flooding scheme based on posterior probability is proposed. Through this scheme, the probability that the target virtual machine distributed to each physical host can be acquired, which reduces the experimental costs and overheads of the same attack and provides theoretical guidance for the next attack of the attacker. The co-residency method combined with covert detection method and automated virtual machine flooding strategy successfully conducted in the commercial cloud. As a typical malicious behavior against the cloud platform, the error rate of the false positive is no more than 0.5%, with strong robustness and low cost, so it is urgent for the major cloud service providers to pay attention to and prevent them.

(3) A universal covert channel detection method has been proposed from the idea of resource sharing. Covert channel is an important threat information security system, and it is common in operating systems. The ones in cloud computing environment are more diverse and difficult to be found. Therefore, this thesis builds a general detection model for covert channel detection, which can detect hidden side channels in system level and process level.

From the two-way relationships between the clock, the event, and the covert channel, it is clear that the event possesses the ability to transmit a secret information.

Therefore, event correlation analysis will be an intrinsically effective solution to covert channel detection. Therefore, the proposed scheme mainly aims at the potential threat of covert channel in cloud environment, fully consider the difference between virtualized architecture and ordinary computing environment, explores the characteristics of covert channel by event recording mechanism, and traverses for possible results using resource matrix method. An analysis based on event logs and security profiles is presented, fitting the features of today's cloud computing and data center environments. At the same time, the designed scheme uses a penetrating physical security configuration scanning method to collect the necessary event log and security policy in the information collection stage, which makes that the cloud provider does not need to worry about the state space explosion. It also improves the original shared resource matrix method, preprocesses the event log and security configuration file, and reduces the cost of building a shared resource matrix. Furthermore, according to the actual application scenario, the experiments are designed and compared with results of existing methods to verify the advantages of the proposed scheme.

(4) A side channel threat location scheme has been proposed. In order to overcome the shortage of current cross-VM side channel localization methods and the inaccuracy of existing threat location technologies, this thesis designs a framework based on hardware features and VM introspection to detect and locate the cross-virtual machine side channel attacks.

Modern CPUs contain many hardware features for performance analysis, which we use to learn about a number of side channel threats. By measuring their performance in terms of overheads (such as Cache hit, branch transfer, and run time), different side channels are analyzed multi-dimensionally and different strategies are customized for signature-based detection. At the mean time, by introducing the accurate address information of LBR, and by introducing virtual machine introspection technology, the scheme can obtain and semantically translate the meta-data of side channel threats. On this basis, combined with binary memory analysis technology, it restores the traceability with the smallest priori knowledge and the accurate address of the side channel threat. The scheme combines detection and localization, and uses the existing information transmission mechanism of Hypervisor to construct a fast channel for information

analysis and distribution, which solves the challenge of gathering information of virtual machines in a cloud environment. As much as possible, the prototype system integrates virtual machine introspection tools and memory analysis tools with the existing architecture to quickly and transparently find the key malicious code in memory. The solution takes the lead in solving the problem of positioning the side channel attack in virtual machines and helps the virtual machine audit.

(5) A side channel defense scheme has been proposed. With the advent of new cross-VM side channels, researchers have come up with some targeted methods to mitigate side channels. It may be a good idea to populate the sensitive operation's runtimes randomly, or at fixed, or multiple times. However, existing mitigation methods introduce a lot of deliberate time delays and require cloud tenants to always suffer this performance loss. One type of research includes ways to eliminate fine-grained clock sources in the Hypervisor layer. These methods, on the one hand, tamper with the clock and affect the operation of many time-sensitive applications. On the other hand, changing the system clock for a long time would have a huge impact on system performance.

This thesis proposes a side channel protection approach that allows virtual machines to request fuzzy clock sources dynamically. It uses hardware virtualization extension technologies to devise a lightweight and effective method to achieve the overall side channel protection. The RDTSC instruction interception provided by the Intel VT-X emulates a virtual clock source and provides it to the user. Moreover, it overloads the VMFUNC instruction interface in a novel way that allows applications on virtual machines to send on-demand requests to the Hypervisor, to "just enough" obscure the lowest n bits of the TSC, which prevent other malicious virtual machines from accurate time measurement. The approach also has versatility and extremely high availability. This thesis demonstrates the defense effect of this scheme on multiple covert channels. Meanwhile, it gives the extent that the TSC values of guest VMs should be obscured in these attack scenarios. In terms of performance analysis, experiments on three different workloads show that the proposed scheme has tiny overhead in cryptographic applications. In the most common Web-server application among cloud environment, this scheme has lower performance overhead than existing defensive methods.

In summary, this paper systematically studies new threats in cloud environment

from multiple aspects such as attacks, detections, and defenses, and makes innovative contributions to both offensive and defensive ends. Combining CPU hardware features and virtualization technology, taking commonality and transparency as the starting point, several cross-VM side channel detection, location and defense schemes are proposed, which provides new ideas for the solution of this problem. This article is a further enrichment and innovation of researches on cloud computing security and virtualization security, and a great significance for enhancing the security of the cloud platform.

Key words: cloud security, cross-VM side channel, VM co-residency, event correlation, VM introspection, dynamic time blurring

1 绪论

云计算，从概念提出, 到落地生根，短短的几年经历了巨大的变革。当下，云计算被视为存储数据、部署软件及服务的下一代基础设施，也代表着一种向共享的第三方基础设施按需购买核心计算和软件能力的商业模式[1]。跟据 2017 年云计算关键行业应用报告，2016 年，全球云计算市场稳定增长，典型云服务市场，以 IaaS、PaaS、SaaS 为代表，增速 25.4%，市场规模高达 654.8 亿美元，并且预计在 2020 年将达到 1435.3 亿美元，其中，以我国为代表的云计算新兴国家更是高速增长，中国市场全球占比已从 2013 年的 4.4% 上升至 2016 年的 7.3%[2]。目前，国内外的商用云计算业务都已进入蓬勃发展的重要时期，各大 IT 巨头都你追我赶，中小型服务提供商也百花齐放。其中的典型代表有国际上市场份额最大的亚马逊弹性云 Amazon EC2、微软的 Microsoft Azure 云、Rackspace Mosso，国内市场份额最大的阿里云、腾讯云、云杉、青云等云服务提供商。

而在现有的云计算服务中，安全和隐私的问题一直令人担忧。多租户动态聚合、边界泛化的特点使得云计算平台天生就难以抵抗虚拟机之间共享计算资源所带来的安全威胁。如在 2018 年初，由于 Intel 等芯片厂商被爆出存在由于硬件设计缺陷导致的侧信道 Meltdown 和 Spectre 漏洞，且无法通过微指令修复，使得 Windows 和 Linux 等操作系统被迫对加入内核页表的额外保护机制，进而导致其性能有 20% 至 50% 的下降，有研究者甚至因此称 2018 年为信息安全的新元年。而这一切均是由处理器 Prefetch 指令侧信道导致的[3,4]。

由客户与供应商之间的新型信任关系所带来的一部分威胁是不言而喻的。比如，用户必须相信云服务提供商能够保护用户数据的隐私性和计算的完整性。不仅如此，在虚拟机间透明共享物理资源的细微之处，还有一些来自其他用户的非明显威胁。并且，很多云服务供应商允许多租户的虚拟机复用同一物理硬件。因而可想而知，一台用户虚拟机可能与攻击者的虚拟机分配到同一物理服务器，这就产生了一种新的威胁 — 攻击者可能通过共享资源构建侧信道，渗透穿过虚拟机间的隔离，从而破坏客户机密性。这种存在在同一物理设备上的多台虚拟机之间的威胁，被称为跨虚拟机侧信道威胁。

目前，国内对跨虚拟机侧信道的研究相对较少。为了满足现有云平台在安全与隐私方面的需求，也为之后国内相关研究人员对跨虚拟机时间信道的研究打下基础。本文通过深入理解跨虚拟机时间信道的攻击，以及虚拟机同驻的方案，进而提出针对这些威胁的检测与防御方案。本文对增强云平台的安全性有重要意义。其研究是对云计

算安全和安全云技术的进一步丰富和创新，希望对我国云计算产业和信息安全产业产生推动作用。

1.1 研究背景和意义

虚拟化技术作为云计算的核心技术之一，其本身的安全性引起了广泛地关注[5]。一般情况下，虚拟机之间通过虚拟化软件互相隔离，攻击者很难通过正常渠道在虚拟机之间传递秘密信息[6]。随着越来越多地依靠计算机来处理和管理我们的个人数据，保护敏感信息免受恶意黑客的攻击是一个亟待解决的问题。在许多形式的信息泄漏中，最常见的便是隐蔽信道和侧信道威胁。攻击者可以使用隐蔽信道或侧信道绕过种种安全机制，进行信息泄露[7]或窃取秘密信息[8]。近十年来，侧信道攻击已经发展成为一种针对共享硬件的有力的攻击武器。侧信道攻击通过测量功耗、电磁辐射或者是时间信息，能够获取机密信息，尤其是获取密码系统中的密钥。最近的研究中，跨虚拟机的侧信道经常被提出，用于获取同驻虚拟机中的秘密信息。和传统的侧信道攻击相比，更加隐蔽的优势，因为攻击者往往不需要获得被攻击虚拟机的任何权限。

多租户动态聚合、边界泛化的特点使得云计算平台天生就难以抵抗虚拟机之间共享计算资源所带来的安全威胁。其中，虚拟机同驻威胁首当其冲。虚拟机同驻即将属于攻击者的虚拟机 (或僵尸虚拟机) 实例运行在目标虚拟机所在的物理主机上。自从 Ristenpart 等人[9] 在 Amazon EC2 中成功地利用侧通道攻击获取目标用户虚拟机中的私密信息 (如 Web 服务器的访问流量等)，在云平台或者说虚拟化平台上的侧信道攻击便高速发展。Zhang 等人能够通过在 L1 Cache 上的侧信道攻击获取同驻虚拟机的密钥，进而截获用户的账号[10,11]。由于虚拟化环境变得更加复杂，人工维护监测困难，自动发现时间信道的需求更加强烈。在另一方面，由于基于硬件特性的实现通常都因架构而异，并且 CPU 和缓存的版本更迭也会导致实现上的较大差异，所以在共享硬件上的侧信道防御手段都较难满足现阶段的安全需求。

由于云环境 “虚拟隔离，物理共存” 的特点，云平台允许同驻的虚拟机共享物理主机的大部分资源，因此同驻威胁难以避免。而如若攻击者虚拟机已经实现了与目标虚拟机的同驻，则其可以实施各种手段的攻击，主要包括资源干扰、拒绝服务、隐蔽/侧信道、虚拟机跳跃、虚拟机逃逸和迁移间隙等[12–20]。利用跨虚拟机侧信道，恶意的虚拟机可以破坏云平台中数据的机密性和资源的可用性，导致严重的安全问题，对大型云租户和普通云用户都会造成极大危害。

1.1.1 一切威胁的前提

传统针对单机系统的攻击或恶意行为大多是侵入式的，需要较强的条件和假设才能成功实施。在正常情况下, 由于管理程序提供的隔离, 恶意程序很难在虚拟机之间公

开数据。不幸的是, 攻击者可以使用各种隐蔽通道来绕过信息泄露的安全机制。而在云环境中，虚拟机之间通过访问共享的资源便会泄露信息，从而达到攻击的目的。云计算的动态特性以及边界泛化使得已有云安全机制无法满足需求，同样侧信道攻击的威胁也难以避免。

在同驻检测的防御方面，研究学者对此给出了目前较好的解决方案 [21,22]。然而这些方案采用被动防御的方式，没有主动规避同驻的发生。它们大多是当检测到同驻情况之后，再将可能的受害虚拟机迁移到安全的主机上。这也进一步说明同驻的危害性与不可避免的特性。

然而，所有针对云平台的外部攻击均来自与虚拟机租户 (不包括云平台的内部威胁) [23]，因此只有当攻击者与受害者同驻了之后才能有进一步的危害。其他形式的侧信道如声音、功耗与电磁辐射无法在云平台或计算中心中测量，导致时间侧信道是用于确定是否同驻的强有力工具。既然同驻是一切威胁的前提，那么时间信道便是一切威胁的源头。

1.1.2 更进一步的危害

当云环境中不同的虚拟机运行与同一物理主机之时，恶意租户不需要绕过虚拟化平台的隔离机制，便能通过探测共享资源的状态信息，建立泄露模型，窃取其他同驻虚拟机的隐私信息。这种攻击即跨虚拟机的侧信道攻击。侧信道攻击最初由 Kocher 等人 [24] 提出，经过二十年的发展，已经成为十分常见的攻击方式。它主要针对密码算法的硬软件实现，当密码算法执行时所产生的如声音、功耗、电磁波以及运行时间等各种敏感信息可以被攻击者用于攻击。这种信息泄露途径被称为侧信道。攻击者通过采集信息，再结合密码算法的具体实现，通过机器学习的方法，可以推断出密钥或部分密钥。

相对于针对传统计算环境的侧信道攻击研究来说，跨虚拟机的侧信道攻击研究起步较晚 [9]。在云计算环境中，声音、功耗和电磁波等信息处于远端，攻击者很难接近物理硬件来测量这些信息。因此，侧信道攻击的一种典型形式是**基于时间的**攻击。在这种攻击中，攻击者能够测算可观测到的时间范围内，由受害者产生的扰动。多租户同时分享的主机时，最容易面临**访问驱动的侧信道攻击**的风险。比如，一个用户 (攻击者) 可以通过测算另一个用户 (受害者) 在他们的共享资源上产生的影响来推测受害者信息。即攻击者通过观察其 Cache 行或 Cache 组在一定时间内是否被受害者清除 (访问该 Cache 行或 Cache 组对应的内存)，从而得知受害者是否访问过该处 Cache [25]。研究者已经证明这种攻击可以被用来窃取受害者的 Xen 虚拟机上的密钥 [10,17,26]，收集潜在的敏感应用数据，以及劫持受害者在 PaaS 云上的 web 服务器的账户 [11,27] 等等 [28–31]。

1.2 国内外相关工作

恶意的虚拟机通过时间信道可能破坏云平台中数据的机密性和资源的可用性，导致严重的安全问题，对大型云租户和普通云用户都会造成极大危害。因此，近年来，针对这些安全威胁，国内外研究者进行不少研究工作，主要集中在以下几个方面:

1.2.1 跨虚拟机侧信道攻击

上世纪末就有研究学者提出 Cache 是否命中信息可用于密码分析的思想。此后近二十年，基于 Cache 的侧信道攻击围绕传统计算环境展开，得到了长足的发展，而在跨虚拟机侧信道方面则未有建树。直至 2009 年，Ristenpart 等人 [9] 在 Amazon EC2 中实现了 L2 Cache 隐蔽信道，远小于可信计算机系统评估标准 (TCSEC) 建议的可接受阈值，从而开启了该方面研究的先河。研究学者后续将基于 Cache 的隐蔽信道 [32] 略微将带宽提高到了 1.77 bps，不过这种低带宽隐蔽信道能够给云环境带来的危害仍然十分有限。针对隐匿信道低带宽的问题，Wu 等人 [33] 重新设计纯基于时序的数据传输方案, 用内存总线作为高带宽隐蔽信道介质，实现了超过 100bps 的高带宽，并对各种虚拟化 x86 系统现实了隐蔽信道攻击，此时这种高带宽侧信道就已对云安全构成了严重威胁。当恶意虚拟机与目标虚拟机同驻时，恶意虚拟机便能实施跨虚拟机的侧信道攻击。

隐蔽信道是一种有效的回避云中多方之间的隔离机制的安全威胁。其中，基于 Cache 的隐蔽信道允许在不同虚拟机中的非特权用户程序之间传输率达到几千比特每秒。但是，由于缓存很小且是共享的，因此基于缓存的通信容易受到所有系统活动和中断的干扰。Maurice 等人提出，以前的工作依赖于两个矛盾的假设：其一、错误纠正码的直接适用性；其二、噪声能够有效地防止隐蔽信道，而这两种假设都是错误的 [34]。首先，由于噪声特性，纠错码不能直接应用。其次，即使系统活动干扰极大，也可以构建一个无差错和高吞吐量的隐蔽信道。其方案所构建的隐蔽信道比 Amazon EC2 上演示的先前隐蔽通道高出 3 个数量级。这种强大且无错的信道甚至允许在两台虚拟机之间建立 SSH 连接。

早期的基于缓存的侧信道能够泄漏细粒度的信息，例如，加密密钥在 L1 缓存会被泄露 [10,25]，不过攻击者虚拟机需要经常抢占受害者虚拟机，并观察其对 L1 的使用情况，使得这些攻击比较容易被用软件的方式缓解 [35]。Liu 等人 [18] 针对三级缓存 (LLC, Last Level Cache)，实现了高效的 PRIME + PROBE 侧信道攻击，他们设计了一种未知虚拟地址映射时探测缓存集的算法，而且使用时间访问模式来识别受害者的访存操作，其交叉 VM 隐蔽定时信道可实现带宽高达 1.2 Mbps。在侧信道攻击方面，Irazoqui 等人于 2014 年 [17] 将传统的侧信道攻击应用于 Xen、VMware 等虚拟化平台中，在实

验环境下恢复了受害者虚拟机中的部分密钥。之后在 2016 年，他们基于 Cache 一致性协议，首次提出了 Invalidate-Transfer 攻击，构建了跨 CPU 的时间信道，成功恢复了另一台虚拟机中的 AES 和 ElGamal 密钥。Yarom 等人 [26] 提出了 Flush-Reload 攻击，其利用云环境中被普遍使用的 Memory deduplication 特性，是数据重用产生信息泄露的一个典型。该攻击利用了 Hypervisor 中常用的共享内存机制，在加解密程序执行前将指定地址的一个或多个 Cache 块，能够推断程序执行时是否发生了 Cache 命中，进而可以推测出密钥或其他秘密信息。从 2015 年开始，Danial Gruss 及其团队针对跨虚拟机的时间信道的构建进行了深入的研究。他们改进了 Flush-Reload 攻击并提出了新式的 Flush-Flush 攻击，提高了攻击的隐蔽性。他们还首次将 Rowhammer 攻击运用在跨 CPU 的虚拟化环境中，实现了基于内存 row buffer 的隐蔽信道与侧信道攻击，进一步提高了跨虚拟机的时间信道容量 [36]。

1.2.2 云环境下虚拟机同驻威胁

如果攻击者意图实施针对云计算平台的攻击，则必须先实现其恶意虚拟机与目标虚拟机的同驻。早在 2009 年，Ristenpart 等人 [9] 首次提出了在 Amazon EC2 云计算平台上虚拟机同驻方案，但是其检测同驻的方法基于同驻的两台虚拟机之间网络包往返延时 (RTT，Round Trip Time) 会随着同驻与否而发生变化的原理，过于简单，未能考虑到云平台中复杂的网络环境所造成网络包延时测量不准确的问题。Bates 等人 [37] 提出利用“同驻水印”来进行同驻检测，利用检测虚拟机向外部周期性地发送网络包，同时通过外部代理向目标虚拟机发送网络包，通过测量网络包通信是否有足够的延时来判断虚拟机是否同驻。然而，此方法与 Ristenpart 等人的方法都依赖于云平台允许虚拟机自由地和外部进行网络通信，很容易被云平台因安全考虑将 ICMP 协议禁止而失效 [38]，且云平台一旦对不同的私有专用网进行隔离时，基于网络信息的同驻方案将会变得毫无用武之地。Zhang 等人 [21] 提出了一种虚拟机同驻探测方案，其让租户将一些暂不使用的处理器缓存行留为警报行，若探测到报警行被使用或其负载与之前不同，则表明其他虚拟机使用了该处理器缓存 (下文统称为 Cache) 区域，同驻检测器进程就会发出警报。同时，余思等人 [39] 将基于 Cache 负载作为特征，通过对比 Cache 负载的差异性度量来检测虚拟机是否同驻。

上述两种方案都根据 Cache 的负载差异进行同驻判断，但是云平台为了降低能耗，提高资源利用率，引入了各种基于硬件特性的资源优化机制和资源隔离机制 [40,41]，使得 Cache 的负载测量并不准确。并且，计算负载特征方法复杂，很难保证同驻检测的实时性。

1.2.3 跨虚拟机侧信道的检测

自隐蔽信道的概念在 1973 年被 Lampson 提出后 [42]，隐蔽信道和侧信道的检测就一直被研究学者所关注。后续的研究将隐蔽信道重新划分为存储隐蔽信道和时间隐蔽信道两种类型 [43]。针对描述的隐蔽信道产生原因，研究者提出了一系列检测和识别方法，包括无干扰分析方法 [44]、针对源代码的分析方法 [45]、基于信息流的分析方法 [46]、共享资源矩阵法 [47,48] 等。

而在云计算环境中，隐蔽信道分析是一个重要难题，其原因在于：其一，它以强制访问控制技术为基础 [49]，而系统越来越大的代码量分析起来极为复杂。其二，传统的隐蔽信道检测的方法并不适用于云计算环境，需要对顶层设计或者源代码进行分析，无法保障实时性；其三，基于统计学的检测方法不好实现，因为对隐蔽信道的利用面临许多干扰，如极短的时间间隔和重叠的噪声。Wu 等人提出 C2Detection [50]，它在 Hypervisor 层捕获信息流，并利用马尔科夫与贝叶斯模型的双阶段算法来检测隐通道。C2Detection 能检测的范围广，且可扩展 (检测算法是以 LKM 的方式加载到 VMM 内核)，但是该方法需要运行在 Dom0 的内核态，实施起来并不方便。Chen 等人 [51] 通过使用虚拟机回放的方式来检测侧信道，可以重现程序的执行，包括其精确时序。其方式虽然新颖，但是带来的开销很大。最近一种基于 LLVM 的 SGX 侧信道检测方案 HYPERRACE 被提出 [52]，它能够消除由于超线程导致的所有侧信道威胁。HYPERRACE 为每个 SGX enclave 线程创建一个影子线程，并要求底层的不可信操作系统在调用 enclave 代码时在同一物理内核上调度这两个线程，因此 HYPERRACE 也能检测到基于异常或中断的侧信道的发生。

虽然云环境下的隐蔽信道有多种 [50,53]，但是在此，本文主要考虑的对象是跨虚拟机的域间侧信道，以及由于虚拟计算环境设计架构缺陷导致的恶意隐蔽信道，来开展研究。此类隐蔽信道是云计算环境中需要特别重视的，需要用特殊的方法来检测。

1.2.4 跨虚拟机侧信道的威胁定位

隐蔽信道通过在两个进程之间建立一个非法通信信道和通过定时调制发送信息，违反了底层系统的安全策略。最近的研究表明，诸如云计算等流行计算环境对这些隐蔽的时间通道的脆弱性。Chen 等人提出 CC-Hunter [54]，能够成功地检测不同类型的隐蔽时间信道在不同的带宽下的消息模式，从而对隐蔽信道进行定位。Zhang 等人提出 CloudRadar [55]，通过结合两种基于行为的检测方法：(1) 它利用基于签名的检测来确定当受保护的虚拟机所执行的加密应用；(2) 它使用基于异常的监视定位和识别基于缓存的侧信道攻击中典型的异常缓存行为。相比其他侧信道检测与定位的工作，Cloudradar 在检测率上有显著提升，但是它依然只能用于检测毫秒级基于缓存的侧信道攻击。

与此同时，许多数字审计技术也被引用于云环境中的威胁定位中。其中，BEEP [56] 对应用程序进行静态和动态分析，以确定单元级执行执行点。文献 [57] 该方案结合 rekall [58]，提出了在 Linux 环境下自动化地生成审计分析结果的方案。Protracer [59] 结合了单元级污染源实现了溯源追踪的低开销。Xu 等人 [60] 利用系统事件之间的依赖性来减少日志条目的数量，同时仍支持高质量的审计分析。虽然这种方法允许对攻击进行细粒度的因果分析，但它们的大量运行时开销可能限制了它们在现实生产环境中的适用性，且这些方案均未能讨论移植到云环境的可能性。基于虚拟机监视器的 “out of the box” 检测方案能够协助构建一系列的安全工具，成为近年来研究的热点 [61]。然而基于虚拟机自省 (Virtual Machine Introspection, VMI) 的监控检测方法需要客户机系统的内部语义的支持。云环境下规模化多样化的客户机系统会使得语义鸿沟问题变得更加突出，同时也大大增加了检测开销 [62]。

1.2.5 跨虚拟机时间信道的防御

为了抵御云计算环境中的侧信道攻击，研究人员给他们在避免同驻的角度解决这些问题 [21,22]，实时检测侧通道利用 [55,63]。

除了绕过侧信道攻击方案之外，还存在大部分针对缓解侧信道攻击危害的防御机制，即通过干扰云环境中的资源共享机制，以消除攻击者在攻击过程中测时的事件中包含的信息。例如，资源分区 [64–66] 和访问随机化 [67–69] 已作为 CPU 缓存的时间侧信道攻击的硬件防护方案被提出。其他的方法在缓解侧信道攻击中修改了缓存的使用方式 [70,71]。还有一些方法通过更改 CPU 调度程序来减轻 CPU 核心共享，从而缓解的侧信道攻击 [72,73]。CATalyst 使用 Intel 特定的缓存优化 [41] 来抵抗基于 LLC 的侧信道攻击。其思路是将 LLC 分割成一个硬件软件混合管理的 Cache 分片分配给各个虚拟机，从而避免资源的共享。然而这只能防御基于 Cache 的侧信道，并不能防御基于内存或是 TLB 的侧信道。上述这些方式均采用新的硬件技术来实现，而这实现起来是一个十分复杂的过程，而且这些技术可能需要多年才能被被合并到生产，并最终使用在云中 [11]。

近年来，随着新的虚拟机侧信道的出现，研究人员提出了一些针对性强的方法。Wang 等人 [74] 分析了基于共享内存控制器的时间信道，并提出了关闭这个通道的方法。相比这些基于软件的解决方案，Cache 着色 [65,70] 具有更简单的设计，它并不受 Intel CPU 中用于物理地址虚拟地址映射的未知的哈希方案的影响，然而该方案对性能影响较大。Gruss 等人利用硬件特性 — 硬件事务存储，提出了一种能够防御现阶段所有基于 Cache 的侧信道攻击的方案 Cloak [75]。Cloak 可以防止攻击者对敏感代码和数据的 Cache 未命中事件进行观察，并且其性能开销较低。同时，Cloak 还可被应用在 Intel SGX enclaves 内运行的代码中，可以有效地阻止 SGX 侧信道的信息泄露，从

而解决了 SGX enclave 的一个主要弱点。

还有一类研究包括在 Hypervisor 层中消除细粒度时钟源来抵御侧信道攻击。Fuzzy time [76] 是第一个在虚拟化环境下的此类方案。与之相似，有些方案通过隐藏被观测时间 (如 Cache 命中或不命中) 的细微差别，提供一个模糊的计时器 [77] 或迫使所有执行的间隔是相同的 [78,79] 来抵御攻击者获取精确时间。然而，这些方法与许多依赖于细粒度时钟源的应用程序相悖。对于侧信道缓解来说，非随机地将敏感操作的运行时间填充到固定时间或固定时间的倍数也许是一种较好的方式。但是现有的缓解方法引入了很多刻意的时间延迟，并且要求用户始终忍受这种性能损失。文献 [80] 将云环境中每个驻留的虚拟机复制三份，并将 I/O 事件作为时钟源统一地提供给外部观察者，其额外的性能成本是相当大的。文献 [71] 提出的方法可以保护多个侧面的信道攻击，但是它改变了原始系统的优化机制 (例如页面重复数据删除)，这也将导致性能下降。

防范侧信道攻击的另一种方法是修改应用程序，以便更好地保护其中的敏感信息。这些解决方案包括：限制在敏感数据上的程序跳转分支的工具 [81]，特定应用的无侧信道实现 [24]，或者甚至执行多个程序分支，让程序的敏感信息看起来有多个值 [82]。然而，这些解决方案往往是针对特定应用程序或特定攻击的，也未考虑到云计算环境下跨虚拟机侧信道的特性，不能实现通用的、轻量级的、动态的侧信道防护。

终上所述，跨虚拟机的时间信道攻击已经成为云计算环境中威胁用户隐私的巨大挑战，现有的云安全防护机制很难发现和应对时间信道这种非侵入式的攻击。云计算平台或是数据中心都亟需一个良好的跨虚拟机的时间信道检测工具来深入地感知和检测系统和网络的安全状态，以便更好的应对随之而来的新型攻击。同时，需要一种能够全方位地防御大多数跨虚拟机侧信道的系统级防御措施。

1.3 研究思路与主要贡献

基于上述现有研究的缺点和弊端，本文针对性地对跨虚拟机侧信道的攻击、检测与防御进行了多项研究。

首先，在跨虚拟机信道构建的研究方面，本文改进了已有的两种主流的跨虚拟机时间信道攻击，将原有信道的正确率提高到了 95% 以上。利用其构建的隐蔽信道能够达到基本无误的传输率。

其次，在面向云平台的虚拟机同驻研究方面，本文的虚拟机同驻实现方案建立在所提出的跨虚拟机信道构建方案的基础上。首先以准确性、高效性和实用性为目标，提出了一种基于微处理器架构隐蔽信道的同驻检测方案，并以基于内存总线冲突的隐蔽信道和基于 Last Level Cache (LLC) 的隐蔽信道为例，实现两种同驻检测技术。接着，结合提出的虚拟机实例洪泛 (VM-flooding) 策略，并将其应用于同驻方案中。VM-flooding 即攻击者在同一时间在同一个云平台可用区，开启大量的虚拟机实例，其本

质上是滥用云平台的虚拟机分布策略，但其事实上并不违反云平台的商业策略和服务等级协议 [23]。提出的基于后验概率的自动化 VM flooding 方案能够在投放虚拟机之后对所投放的实例进行同驻检测的分类，然后进行约减。通过上述过程，能够得到攻击者实现与目标虚拟机同驻的精确概率；最后，结合实际，在某云的某个可用区中实施本同驻方案。

在侧信道识别和检测的研究方面，本文提出一种通用的云计算环境下的隐蔽信道检测框架和模型。之前的研究都是针对单一信道，根据信道的的负载不同、特点不同，分析的方法、模型都不相同，不具有通用性。而本文从资源共享的角度，提出了一种针对存储隐通道的普适性的隐蔽信道检测框架和模型，既可以检测系统级的隐通道，也可以检测进程级的隐蔽信道。提出基于事件关联的分析方法来实现隐蔽信道的行为检测：该方法根据攻击者的攻击意图来匹配隐蔽信道，能够显著降低误报率 (提升准确率)。这是因为隐通道一般都是精心构造的，所以按照本方案来进行检测更加贴合实际情况。框架具有良好的透明性与灵活性，不采用信息流分析的方法，故不必担心状态空间爆炸等问题；系统能够做到自动分析；具有广泛适用性；同时，本方案以插件形式部署于云平台上，方便装卸和管理，可扩展性也更强。

在侧信道威胁定位方面，本文设计了基于硬件特性和虚拟机自省的系统来实时的检测和定位云中存在的跨虚拟侧信道攻击。由于现有云环境中侧信道威胁定位方法不足，同时现有威胁定位技术不够精确，于是本文中提出一个新框架，通过动态跟踪共享的处理器硬件上的冲突模式来检测侧信道的存在，计算特定共享硬件上隐蔽信道的可能性。通过提供软件支持的虚拟机自省技术，还原底层硬件的语义信息，给出侧信道威胁的精确位置。受害者虚拟机在执行需要加密应用程序时，具有独特的硬件行为。同样，攻击者虚拟机在执行攻击程序时，也具有极强的异常硬件行为特征，而通过 CPU 的硬件特性能够记录这两种类型的事件。因此，本文通过使用基于特征的检测方法来识别这种事件的发生。之后，使用硬件提供的精确位置信息，利用虚拟机自省技术的语义还原机制和二进制分析工具，实现在虚拟机外部的侧信道攻击行为定位。

在时间信道的防御研究方面，本文提出了一个全局的动态时钟模糊方案，旨在利用最新的硬件虚拟化扩展技术，以提供一个轻量级的和动态的环境，使系统范围内的侧信道危害得到缓解。文献 [76] 中首先引入了 “虚拟时间” 一词，它通过将虚拟机从 Hypervisor 收集到的时间与实时时钟源之间的关系进行随机化，来提供一个模糊的供虚拟机观测的时间。在云计算环境中，虚拟时间可以是从外部观察者 (例如，跨虚拟机的攻击者) 的角度进行的测量。基于同样的观点，Bhanu 等 [77] 掩盖了 RDTSC 的最低有效位，并对性能下降的研究做出了很大的贡献。在类似的研究中，TimeWarp [83] 也提出了模糊硬件计时的软件方法，它增加了两种时间偏移量。第一个是虚拟的透明的 TSC 偏移量，第二个是直接加入到访问 RDTSC 指令上的实时延迟。本文从中吸取

教训，适当减少观察到的虚拟时间，同时不加入任何有目的的时间延迟。通过基于硬件虚拟化技术，有效拦截虚拟机获取的时间信息。此外，为了尽量减少性能开销，采用 VMFUNC 技术，提供灵活的控制时间细粒度。最后，基于 Linux 的虚拟机和 Xen 虚拟机管理程序，实现了本系统。通过提供显式的程序接口，动态插入 VMFUNC 指令，并将其提供给应用程序库 (如加密库) 使用。该实现确保恶意虚拟机无法测量访问共享硬件资源所需的时间，从而防止时间侧信道攻击。

为了避免跨虚拟机侧信道这种针对公共 IaaS 云中的共同威胁，本文分别构建了上述检测方案和防护方案，提供一个平台级的方案来从全局的角度解决跨虚拟机时间信道的问题，并引入最小的开销。在不失一般性，可信性和避免上述影响的前提下，本文遵循信息系统安全可扩展性和对可信计算基础 (TCB) 进行的修改最少的原则，均设计和实现了原型系统，以支持实验验证与评估。

1.4 论文组织结构

论文通过八章组织全文，从云环境下跨虚拟机时间信道模型出发，研究了跨虚拟机侧信道的攻击方案和攻击应用，研究了跨虚拟机时间信道的检测和定位方案，研究了跨虚拟机时间信道的防御方案，组织结构如图1.1。

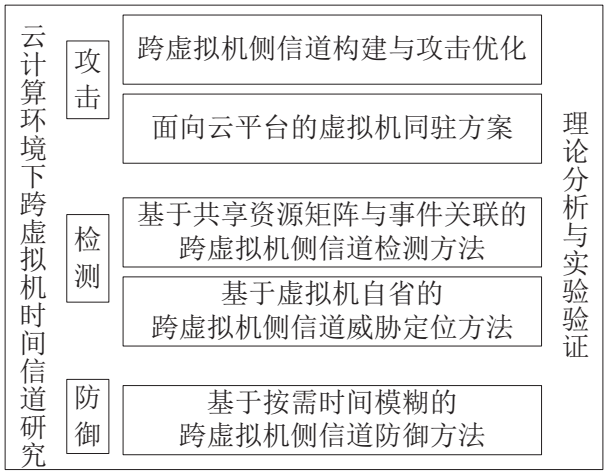


图 1.1 论文组织结构

第一章，绪论。主要包括跨虚拟机时间信道威胁的研究背景意义、目前国内外的针对该问题所提出的方法、本文所要研究的内容及所作出的贡献。

第二章，基础知识。首先阐述了云计算与虚拟化技术的相关概念和特点，详细介绍了 Xen 的虚拟化原理；然后对当前主流硬件虚拟化技术进行了深入探讨，为后续研究的介绍提供基础；接着介绍了跨虚拟机时间信道的定义、特点与模型。

第三章，跨虚拟机信道构建与攻击优化方案。对现有跨虚拟机的时间信道攻击进行深入研究，针对其威胁性质进行分类，并详细介绍了基于内存总线的和基于 LLC 的

两种攻击的威胁原理和相关研究。在此基础上，提出对这两种典型攻击的优化方案，并通过实验验证了优化方案的高效性。

第四章，面向云平台的虚拟机同驻方案。首先阐明了基于隐蔽信道的同驻检测机制，并定义了一种隐蔽信道信息模型；然后本文根据第三章的信道构建方法，提出了两种同驻检测方案；最后利用同驻检测方案，具体地设计并实现了一种同驻方案，且实验验证了其效果。

第五章，基于共享资源矩阵与事件关联的跨虚拟机时间信道检测方法。首先阐述了时间与事件的关联性，并根据该关联性提出了事件关联算法在时间信道检测中的可用性；其次，基于所提思路，设计了基于共享资源矩阵的时间信道检测方法，该方法利用时间与事件、事件与事件之间的关联性，能够从全局角度检测出隐蔽信道利用行为的关联，从而检测出这种威胁。最后实现了该方案，并通过实验验证了有效性，也证明了其开销在一定范围内是可接受的。

第六章，基于虚拟机自省的跨虚拟机时间信道威胁定位方法。针对现有方法仅能检测而无法定位侧信道威胁的缺陷，本章借助处理器硬件特性和虚拟机自省技术，提出一种侧信道实时检测与定位方法。首先介绍了所用到的虚拟机自省技术及其由来，然后介绍了所用的处理器硬件特性 PMU/PMC 和 LBR；之后，设计了将 LBR 信息用于定位，并利用 LibVMI 进行语义还原的方案；最后对实现的系统进行了评估，实验证明该定位方案能够达到指令级的精确程度，并且带来的额外开销十分有限。

第七章，基于按需时间模糊的跨虚拟机侧信道防护方法。该章提出了一种全局的跨虚拟机时间信道的缓解方法，即降低将所有虚拟机所能观测到的时间精确度 (或增大所能观测到的时钟源粒度)；并且该方案结合了硬件虚拟化技术和处理器硬件特性，使得这种降低时间精确度的行为能够按需触发，按量实施；接着，给出了该方案的具体设计思路和关键实现细节；最后用大量的实验证明了该方案的有效性和高效性。

第八章，结语。对本文所做工作进行了总结；对跨虚拟机侧信道安全方面的后续研究方向进行了展望。

2 云计算平台中的跨虚拟机时间信道

2.1 云计算与虚拟化技术

自进入大数据时代，传统的解决方案如超级计算、分布式计算、并行计算和网格计算已经能够满足部分商用计算的需求，有些也能够解决部分研究领域的难题。然而，弹性和可扩展性对于大数据计算服务是至关重要的。大数据的两个主要问题：大数据存储和大数据计算，在云平台和基于云的网络中心中都能得到很好的解决。同时，云计算很好地兼顾了高性能计算和绿色计算两者的平衡，提供了可以按需添加与卸除的一系列资源 (用于存储和计算)，具有很好的弹性和可扩展性。这些功能使云计算成为能够方便处理大数据应用的技术，其特性也使其成为 IoT、雾计算等新兴领域的基础计算环境。

而在云环境中，虚拟化技术是将计算平台中的各种资源、如计算、存储和网络资源抽象与分发的一种技术。它使得计算机资源成为了一种可被分割的资源，其利用效率得到进一步提高。其实现方式是在原有物理机基础之上实现虚拟计算机，即一台计算机可以同时运行多个操作系统，或以互相隔离似的方式、或同应用程序一般运行的宿主方式。这一特性使得虚拟化技术自被提出以来便显得尤为诱人，也以此被广泛应用。

虚拟化技术在逐渐增高对安全性的要求和在同一硬件平台上需要有效支持大量的虚拟机的计算机工业中变得越来越重要。它将计算机中的各种真实资源抽象出来，使得资源能够得到更加合理的分配与最大程度的利用。它被广泛应用在服务器整合、系统软件开发和调试、容错、系统安全、负载均衡等等。随着虚拟化技术的发展，以 XEN、KVM 和 VMWare 为代表的技术脱颖而出，以其在资源管理、软硬件资源利用率和系统管理等方面的出色表现，成为云环境中不可或缺的基础，也成为了云计算的重要技术支撑。

2.1.1 云平台基础架构

云计算系统的架构依托于整个系统和每个组件和子组件。云供应商通过云计算系统中的服务层提供服务，主要分为四个服务层：基础架构即服务 (IaaS)，平台即服务 (PaaS)，软件即服务 (SaaS) 和商业智能 (BI)。其他的一些服务层也归到主要服务层之中，如数据即服务 (DaaS) 也可算作 IaaS 或 PaaS 中。

基于基础设施位置的云计算系统可以以私有云、公有云或者混合云的方式实现。私有云模式是云计算系统的本地实现，在此模式中，硬件位于本地数据中心并使用云软件应用程序为本地用户提供服务。对于安全需求较低的 IT 消费者而言，该模式是最佳选择，因为该模式允许 IT 部门从传统模式迁移到云计算系统，并且仍然支持可扩展性。然而，该模式需要在维护、灾难控制与恢复、安全监管方面进行额外投资。公共云模式是云计算系统的常规模式，该模式由支持计费的云供应商和公共用户的订阅系统提供，与私有云模式不同，它不需要高投资，消费者可以按需支付云存储服务或云计算服务的费用，但是其安全隔离性则不如私有云。

混合云模式组合了私有云和公共云模式，这种模式可以通过互联网等网络连接将私有云连接到公共云。该模式兼备两种云的特点，具有以下优点: 第一，可协作性：两种云模式之间的频繁协作使得允许一个组织可以在保持自己的云安全和维护的同时与其他云合作；第二，可扩展性：混合云可以更加充分地发挥其按需服务的特性。例如在高峰时段资源有限的情况下，混合云用户可以在已有的私有云基础上，方便地从公有云中临时添加新资源。

2.1.2 虚拟化技术

虚拟化技术在大型机时代便被提出，最早可追溯至 IBM 于 60 年代提出的 VM370 系统 [84]，但是真正得以发展壮大则是借由个人计算机时代的普及和 x86 架构的流行。通过在计算机操作系统与底层硬件新增的虚拟机管理器 (Virtual Machine Monitor，VMM，又称 Hypervisor)，底层计算资源可以被 VMM 监管，分割并抽象呈现给操作系统。从操作系统到 VMM 的一些功能的转换有许多潜在的好处。首先，通过在 VMM 内实现功能之后，就可以通过某些手段用于其上的所有客户机操作系统中。其次，由于许多操作系统属于遗留系统、闭源系统或者两者兼有，其上的应用可能无法或需要很大代价才能被移植到现有的新环境中。虚拟化技术给这些问题的解决带来了新思路，老旧的系统可以通过制作镜像的方式，以全虚拟化的形式，重新运行在 VMM 上。最后，VMM 作为系统中唯一对系统资源有绝对完全控制权的区域，是最有可能产生和执行资源管理决策的地方。然而实现这一目标并非易事：VMM 需要能够完整的模拟出 CPU、内存和 I/O 等虚拟环境，且满足一定的效率，这样往往会带来些许性能损失。虽说如此，但是采用虚拟化技术仍是大势所趋。

采用虚拟机技术能够有效提高硬件利用率这一事实已被广泛应用的云计算浪潮证明。随着市场主流性能的 PC 和服务器性能越来越快，人们渐渐发现软件的利用率却并未同硬件一般依照指数增长，硬件资源闲置甚至是常态。网络服务运营商不得不追加服务器资源和网络资源投资，大规模数据中心的管理成本也越来越难以接受。虚拟化技术的引入可谓福音，其资源整合能力可以使得服务器可以承载更多的服务，通过

VMM 合理的资源调配，其硬件利用率可以得到大幅提升，以更好地满足要求。

其次，虚拟化技术降低了软件迁移成本和维护成本。自 x86 成为主流计算平台被业界大量使用以来，x86 平台累计了无法估量的软件资源，这也是 x86 不同于早前各类计算机系统得以长期占据统治地位的原因。然而长期的资源累计必然造成软件的兼容性问题，即使 x86 平台硬件兼容性俱佳，企业定制的或与平台绑定的程序依然或多或少的存在兼容性问题，特别是开发商已经不再维护但是对用户仍然有重要价值的软件，使得用户不得不被绑定到特定操作系统上而缺乏升级至更安全、特性更多的新操作系统的选择，对旧系统和旧软件的维护造成了一种难以避免的成本。然而虚拟化技术可以将计算资源切割，使得计算机可以创造兼容的软件环境用以维护遗留软件，甚至还可以抽象不同操作系统或者计算机架构的虚拟资源，降低了企业的软硬件环境迁移成本。

再次，虚拟化技术提供了更为安全可控的系统运行环境。在传统计算环境下，一台物理服务器可以承载多个服务，且多个服务的运行环境位于同一操作系统之上，一旦操作系统的关键安全节点被攻破，恶意代码便可长驱直入，一举破坏所有服务。而虚拟化技术的隔离特性可以更好的隔离攻击，甚至有些虚拟化防护方案可以让操作系统可控的运行在更为底层、恶意代码无法访问的 VMM 上；另一方面，VMM 却可以以最大权限控制虚拟资源，使得恶意攻击能够被有效记录和回放，能够更好的防护恶意入侵。同时被破坏的操作系统环境可以迅速恢复，大大减小了单服务的宕机率。

然而虚拟化技术虽好，但由于 x86 架构难以模拟特权指令，如何将硬件资源高效地实例化，研究人员想出了一些方案来弥补，可以分类为以下三种：

(1) 软件虚拟化。最常见的是二进制翻译技术，即将动态执行的程序中存在的特权操作进行动态转换，使得虚拟机中操作系统的特殊操作如特权级别切换、系统调用、中断处理等可被 VMM 捕获并做出对应的模拟，这是虚拟化技术早期的解决方案，软件无需修改但系统开销过大。

(2) 半虚拟化。将操作系统内核进行定制，替换特权指令为对应的版本，如将操作系统的系统调用修改为对 VMM 的调用，其好处是不再需要动态翻译运行代码，运行效率有所提高。但是由于需要修改内核，使得内核和 VMM 绑定；商业操作系统则由于闭源属性无法接受这种方案。

(3) 硬件辅助虚拟化。通过修改处理器架构，使其直接内置虚拟化支持，设置不同的运行模式，将 VMM 和 VM 运行环境在硬件上隔离。处理器通过 VM-enter 和 VM-exit 切换运行模式并模拟特权指令，使得虚拟机可以高效、无需修改的执行。这在 Intel 和 AMD 硬件虚拟化技术普及后成为绝对主流方案。

从虚拟化架构上，虚拟化管理器 (VMM) 又可以分为两类。一类 VMM 运行于更底层的虚拟机管理器上，如 Xen。它们通常还可以指定一台虚拟机供用户管理资源 (即

特权管理域 Domain-0)，好似物理主机的操作系统，但是本质仍然是虚拟机。另一类 VMM 则在操作系统内，虚拟机则由通过具有全部功能的操作系统管理，如 KVM。

2.1.2.1 Xen

Xen [85] 由剑桥大学实验室于 2003 年提出，它允许主流的操作系统 (Windows、Linux) 在主流处理器 (x86、ARM) 上以安全和资源管理的方式共享常规硬件，同时不会牺牲性能或功能。至今 Xen 已经经历了多个大版本的调整，全面适合软件虚拟化、版虚拟化和硬件辅助虚拟化等各种技术需求。对于 Linux，BSD 和 Windows XP 等操作系统，可以以极小代价移植到该平台。

虚拟机用户和开发人员通常会定量的测试和对比各种不同的 VMM。Xen 和 KVM 同为主流虚拟化平台，但整体来说 Xen 的扩展性更好，更贴近实际使用需求。虽然 Xen 项目在创建之初采取了半虚拟化方案，然而为了支持完全虚拟化，Intel 还是通过 VT-x 等扩展方式使得 Xen 利用 Intel 处理器的虚拟化技术实现了不修改 GuestOS 的虚拟化。在这个过程中，由于 Intel 在 x86 虚拟化支持上的工作进展，将计算密集型程序在完全虚拟化版本的 Xen 上的性能提高了 10%。同时，I/O 虚拟化的支持提高了利用并发处理虚拟磁盘的 DMA 操作性能，并通过使用事件驱动机制进行重构，大大增加了虚拟网卡性能。此外，为了在虚拟系统中实现更好的体验，还优化了虚拟视频显卡，并且通过共享的虚拟视频内存方法获得了更好的图形性能。

由于 Xen 持续走在推进基于虚拟化的安全应用程序开发的前沿，并允许第三方工具创建一些非常特殊的设置。Xen 的灵活性是如此强大，在可预见的将来它将继续成为趋势。故对于本文而言，选择在学术界更为活跃的 Xen 作为实验平台。

2.1.2.2 处理器硬件虚拟化技术

本节对本文涉及的处理器硬件特性进行介绍。

随着虚拟化的快速演进，Hypervisor 变成商品软件栈的新的系统软件基础，增加了更多的层次结构和特权层级。诸如 Intel、AMD 的处理器厂商通过加入一系列新的特权模式 (如 Intel 的 VMXroot 操作) 引进了相似的硬件扩展 (VT-x [40] 或 SVM [86])。这就导致了商品软件堆栈现在至少有四个特权环和多种地址空间，包括 hypervisor、hypervisor 应用程序、操作系统和用户级程序。同样的，ARM 平台上对于 TrustZone 和虚拟化的支持也引进了四个特权域：U 用户、内核、虚拟机管理器和监控模式。虚拟化的多租户性质进一步在计算机中引入了多个域 (domain，即虚拟机)。云操作系统的互操作性和嵌套虚拟化技术使得系统堆栈更为复杂 [87]。为了更好的支撑虚拟机环境，Intel 提供了 Intel VT-x(基于处理器的虚拟化技术)、Intel VT-d(基于 PCI 总线域设备实现的 I/O 虚拟化技术) 和 Intel VT-c(基于网络的虚拟化技术)。其中，Intel VT-x 在处理器层面实现了虚拟化技术，它的实现架构是虚拟机扩展 (Virtual-Machine

Extensions，VMX)。VMX 下引入了两种处理器模式，即 VMX root 和 VMX non-root，分别支持主机 (Host) 与客户机 (Guest-VM) 中程序的运行，使 guest 系统和软件不经过修改就能直接在处理器硬件上运行。由于 Intel 系列的处理器被广泛应用，且其特性在 x86 架构下的其他类型的处理器 (如 AMD) 中都可以找到对应项，故本文在此以 Intel CPU 为例，描述 Intel VMX 的具体情况。

VMX 架构存在许多功能，并且这些新功能与特性随着 CPU 的升级也在不断地增加。在此，本文仅介绍处理器硬件虚拟化技术 VMX 的基本知识，而后续可能会用到的 CPU 新特性则在后续章节中陆续介绍。下面介绍 VMX 重要指令与控制字段。

(1) VM-execution 控制字段

在 VMCS 区域的三个 VM-execution control 字段需要检查 VMX 的某些特殊的字段寄存器来确认，如 Pin-based VM-execution control 字段、primary processor-based VM-execution control 字段和 secondary processor-based VM-execution control 等字段。IA32_VMX_PROCBASED_CTLS 寄存器决定 primary processor-based VM-execution control 字段，secondary processor-based VM-execution control 字段只受到 IA32_VMX_PROCBASED_CTLS2 寄存器的影响。

(2) VMX 常用指令

VMX 架构提供 13 条 VMX 指令，负责管理四个职能范围：VMCS 区域管理、VMX 模式管理、Cache 刷新和调用服务例程。其中 VMCS 区域管理指令与调用服务例程指令与本文研究的内容息息相关。VMX 架构提供了两个调用服务例程指令：VMCALL 与 VMFUNC 指令。它们服务的对象不同，VMCALL 指令使用在 VMM 中，而 VMFUNC 指令仅能工作在虚拟机中。

利用 VMCALL 指令可以实现 SMM(system management mode，系统管理模式) 的 dual-monitor treatment(双重监控处理) 机制。在 CPU 的 non-root 模式下执行 VMCALL 指令时，Guest 将会退出到 VMM，但如果在 root 模式环境下执行 VMCALL 指令，当满足检查条件时，会在 VMM 产生被称为 “SMM VM exit” 的退出行为，从而切换到 SMM 模式的 SMM-transfer monitor 里执行。本文由于不涉及 SMM 模式，因此不在此做过多介绍。

VM-execution 控制类字段包含 26 个字段，在此不一一赘述。其中 TSC offset 字段、EPTP 字段、VM-functioncontrol 字段和 EPTP-listaddress 字段为本文将会涉及到的内容，故在此说明。

(1) TSC offset 字段当 “Use TSC offsetting” 为 1 时，在 TSC offset 字段中提供一个 64 位的偏移值。在 VMX non-root operation 中执行 RDTSC,RDTSCP 或者 RDMSR 指令读取 TSC 时，返回的值为 TSC 加上 TSC offset。前提条件是使用 RDTSC 指令时，“RDTSC exiting” 位为 0 值，使用 RDTSCP 指令时，“enable RDTSCP” 位为 1

值，使用 RDMSR 指令时，MSR read bitmap 相应位为 0 值。

(2) EPTP 字段 64 位的 EPTP 字段提供 EPT PML4T 表的物理地址，它的结构如图2.1所示。

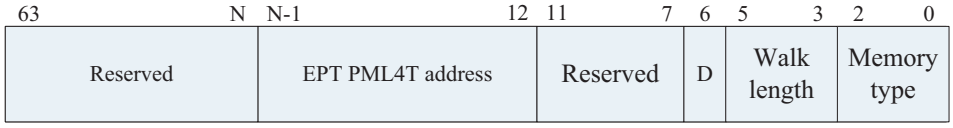


图 2.1 EPTP 字段

2.1.2.3 内存硬件虚拟化技术

在虚拟化平台中，每个 VM 有自己独立的物理内存，VM 本身并不知道有其他 VM 的存在，只认为自己独占从 0 到 MAXPHYADDR 这个范围的物理地址空间。这个原理与 OS 下进程独占地址空间一般无异。譬如在 windows 系统下，一个进程认为自己占有独立的 4G 内存空间，然而这个地址空间实际上是虚拟地址空间，用来隔绝每个进程对物理地址空间的访问。

VMM 在设计时就应将 VM 私有的物理地址空间隔离，并提供地址转换功能。VM 私有的物理地址空间被称为 GPA(Guest-Physical Address) 空间，而真实平台上的物理地址空间被称为 HPA(Host-Physical Address) 空间。而 Intel 提出的 EPT 扩展页表 (Extended-Page Table) 使用了硬件实现了上述功能，即能够通过类似硬件 MMU，将 Guest-physical address 转换为 host-physical address。EPT 的使用原理与保护模式的分页机制下的页表是一样的。VMX non-root operation 内的保护模式分页机制的页表结构被称为 “Guest paging-structure”，而 EPT 页表结构被称为 “EPT paging-structure”。

通常 OS 都会切换到保护模式或者 IA-32e(long-mode) 模式执行，并且开启分页机制。当关闭 EPT 机制时，Guest 内线性地址转换后的地址就是物理平台上的地址。VMM 需要非常细致，非常小心地处理每个 Guest 之间的内存隔离问题。避免 VM 之间及 VM 与 VMM 之间的相互干扰。当启用 EPT 机制后，Guest 内的线性地址经过两次地址转换：Guest-linear address 转换到 Guest-physical address, 以及 Guest-physical address 转换到 host-physical address。在这种情况下，VMM 可以通过 EPT 机制的 GPA 映射 HPA 来实现 Guest 内存的虚拟化。Guest OS 可以自由地实施它的内存管理策略，而 VMM 无须插手。

EPT 页表结构与 Guest 页表结构的实现原理类似。VMX 架构实现最高 4 级 EPT 页表结构 (2017 年底 Intel 推出了新的 5 级页表结构)，分别为：PML4T(EPT Page Map Level-4 Table)；PDPT(EPT Page Directory Pointer Table)；PDT(EPT Page Directory Table) 和 PT(EPT Page Table)。软件可以查询寄存器的 bit 6 来确定是否支持 4 级页表结构，为 1 时指示 EPT 支持 4 级页表结构。每个 EPT 页表大小为 4K，每个 EPT

页表项为 64 位宽。使用 1G 页面时，GPA 转换只需经过两级 EPT 页表的 walk 流程 (PML4T 与 PDPT)。使用 2M 页面时，GPA 转换需经过三级 EPT 页表的 walk 流程 (PML4T、PDPT 与 PDT)。而使用 4K 页面时，GPA 转换则需要经过四级 EPT 页表的 walk 流程，如图 2.2所示。

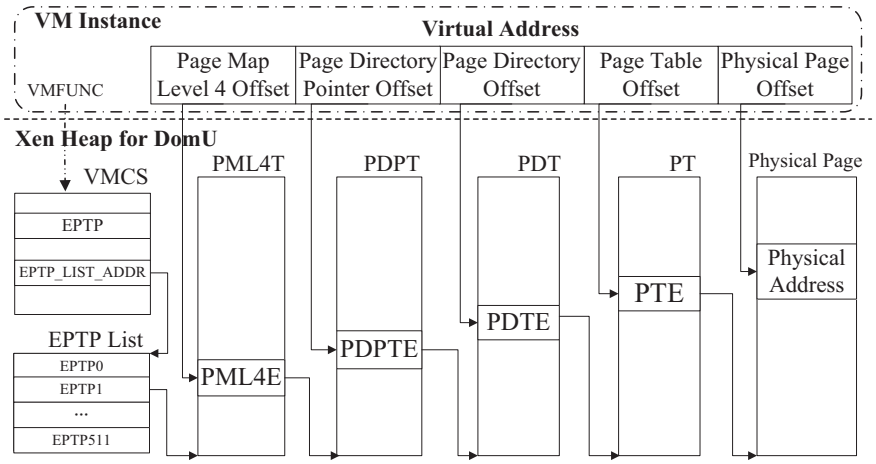


图 2.2 EPT walking 流程

Guest 线性地址转换到 Host 物理地址的过程中发生错误时，将会引发 #PF(page fault) 异常。例如，分页属于 not-present 时, 线性地址不能进行访问，或者页表项的保留位不为 0 等。在 EPT 扩展页表机制下，在 Guest 中 GPA(Guest-physical address) 转换到 HPA(host-physical address) 的过程中发生错误时，将会引发两类 EPT 故障，被称为 “EPT violation” 及 “EPT misconfiguration”。作为这两类 EPT 页故障的响应结果，处理器产生 VM-exit。

2.2 隐蔽信道与侧信道

隐蔽信道是一种允许通信双方以违背系统安全策略的形式而传递秘密信息的信道，现已经被广泛的应用于网络信息数据安全传输和某些跨物理隔离界限的攻击当中。对于隐蔽信道的分类方法有多种，常用的一种分类方案是根据共享资源属性分类，即将其划分为存储隐蔽信道和时间隐蔽信道。其中，存储隐蔽信道是指通信双方能够直接或间接访问到共享的目标值或变量，从而能够构建出某种存取型的通信渠道。而时间隐蔽信道则是发送者使用共享资源 (例如缓存、内存总线等) 造成时间上的不同来发送信息，同时接收者观测到并对信息进行解码。时间隐蔽信道又称为无记忆隐蔽信道，并不能长久地存储信息。这也就是说，发送的信息接收者必须及时接收，否则将会消失。

侧信道，又称 “旁路”，实际上侧信道和隐蔽信道的定义相似，在很多地方 (如从信道的角度考虑) 甚至可以混用。提及侧信道，我们一般指的是侧信道攻击，而侧信道

攻击的目标一般是密码算法。通常，密码算法 (或密码方案) 在实际应用中都会实现在具体的硬件平台上，形成密码模块、密码芯片、密码系统等，它们均实现在特定的数字电路中，而数字电路只存在 '0' 和 '1' 两种基本单元状态。如果我们能够有效地测量这两种基本状态之间的差别，就能够推断出当前所执行的操作 (或数据)，从而能够进行密码破解。这类通过其他渠道来猜测不同状态之间差别的攻击则被称为侧信道攻击，而一般这类渠道则包括时间、声音、功耗和电磁辐射等。侧信道攻击的本质是利用密码实现运行过程中产生与敏感信息有关的旁路信息来实施密钥恢复的，因此防御该攻击的核心手段就是减弱甚至消除这种旁路信息与密钥之间的直接依赖性。另外，侧信道与隐蔽信道的最直接差别在于侧信道一般不需要合谋，而隐蔽信道利用则一定是通信双方合谋进行的。

2.2.1 云环境下的信道载体

在云计算环境下，同一物理主机之上的租户使用着同一块 CPU、内存，甚至是存储系统。而只要是共享的硬件设备均可被用于信道载体。目前，除了常见的基于 Cache 的信道被广泛应用于攻击中，内存中的行冲突 [36]，内存总线冲突 [33]，内存共享页面信道 [26]，内存共享页面的授权机制 [88]，利用 TLB 快表构建的信道 [89] 和 page fault 产生的信道 [90] 等等都已经被研究者发现并可被用于攻击中。甚至在存储系统中，如 Linux 的 VFS [91]、HDD 硬盘的读写操作 [92] 等都可以被作为时间信道的载体。虚拟机之间的软件隔离机制有时也可以作为隐蔽通信的渠道 [53]。总之只要虚拟机之间存在公共的资源，那么没有想不到，也没有做不了的隐蔽信道。

2.2.2 云环境下的信道模型

传统物理主机之间存在物理隔离边界，只能进行网络通信。而在以虚拟化平台为核心的云计算环境中，同一物理主机的虚拟机之间是由虚拟化技术构建的逻辑边界 (虚拟边界)，它们在网络通信之外，还能够通过其他方式 (如共享内存、Hypervisor 设计缺陷) 交换信息；这些方式统称为虚拟机的隐蔽信道。恶意的虚拟机可以利用这些隐蔽信道，窃取其所在主机上其他客户虚拟机的敏感信息，造成敏感信息泄漏。

2.2.2.1 隐蔽信道威胁模型

在云计算环境下，假设有两个恶意攻击者想要传递信息，其恶意进程分别为 P_i 和 P_j，收发双方恶意进程分处同一物理主机上的不同虚拟机 (Domain-U) 中，机密信息经过共享的硬件或软件资源，从一个攻击者泄漏给另一个恶意用户 (如图2.3)。该恶意程序可能是通过劫持了某租户账号并部署或注入在虚拟机中，也可能是直接攻击者自己掌握的恶意虚拟机上运行。两个合谋的攻击者的主要目的是收集被劫持虚拟机的敏感信息，同时希望把这些信息传递到外部。为了区分两天合谋的虚拟机，一般称含有

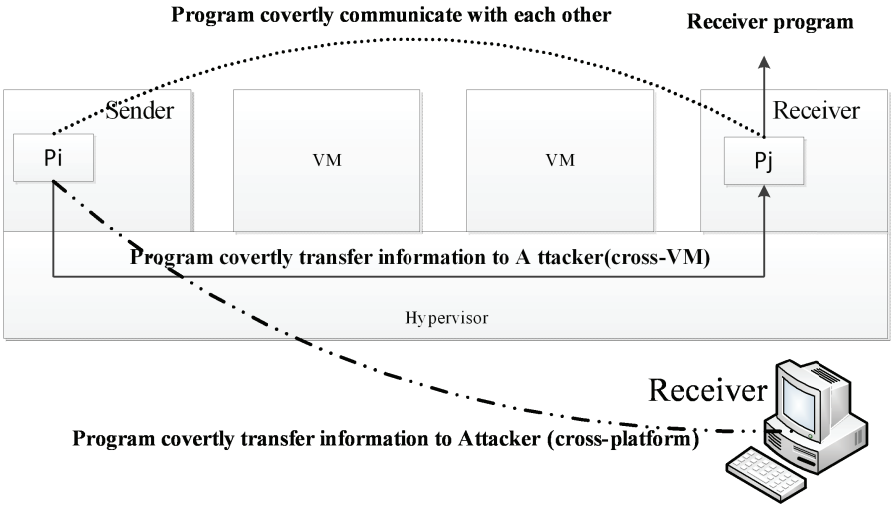


图 2.3 云环境中的虚拟机合谋构建隐蔽信道

调制信息的一方为发送方 (Sender)，称解调信息的一方为接收方 (Receiver)。

2.2.2.2 侧信道威胁模型

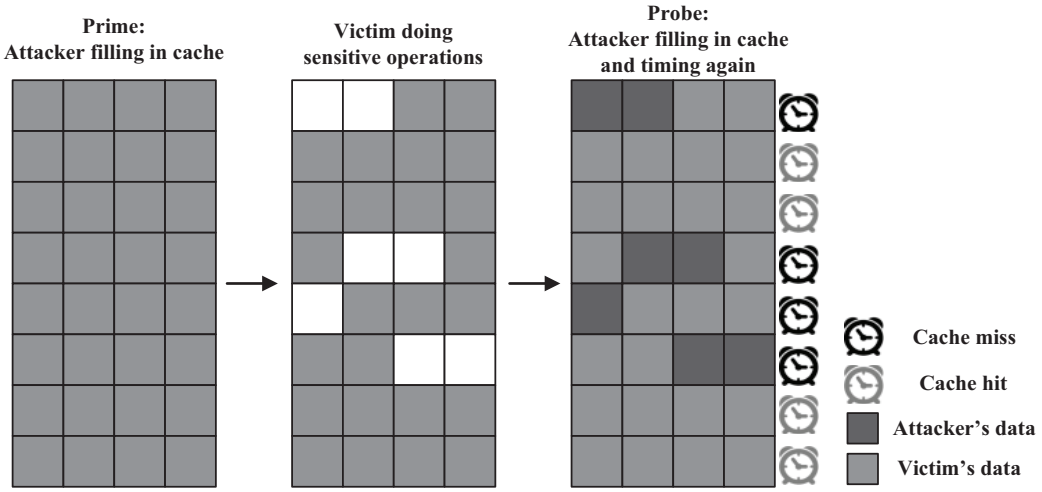


图 2.4 侧信道攻击示例

侧信道攻击模型相对与隐蔽信道模型更加严格。攻击者在此模型中只通过自己本身的访问时间来推断受害者对应的行为，并进一步得到一些关键信息，例如：在基于 Cache 的时间侧信道中，内存访问的部分地址信息，或历史内存访问的数据信息或指令信息。而这些信息之所以能够被攻击者获得都是因为侧信道与敏感操作或数据 (如密钥) 之间存在极大的关联性。

密码算法实现中的条件、分支语句会导致算法执行时与数据形成关联。换言之，相同算法在输入参数不同时，会进入不同的执行流，因此具有不同的访问时间。因此攻击者可以利用算法的执行特征推测数据。例如分组密码和公钥密码的执行通常都具有

和密钥相关的固定 Cache 访问模式，如果攻击者已知加解密算法的实现细节，且能够访问到特定的 Cache 块，结合一些密码分析技术，就能推断出一部分密钥或全部密钥。对于公钥密码算法，由于模幂运算被优化，所以当指数 e 为 '0' 或为 '1' 时的运算时间差别较大。如果攻击者能够通过这种时间差异推测出密码算法执行时的指令序列，就可以推测出指数 e 的取值，进而推测出密钥。

目前，访问驱动的时间信道应用最为广泛，它是由 Cache 争用产生的信息泄露，如图2.4显示了其典型例子：Prime + Probe 攻击。通常，Cache 失效表明发生了争用，攻击者虚拟机在较短时间连续访问两次一段固定的内存时，如果第二次发生了 Cache 未命中 (此时访问时间将会变长)，则可以断定受害者虚拟机在这段时间内访问了这块 Cache 对应的内存。当收集到足够多这样的测量结果后，攻击者就能在此基础上进行密码分析。

3 跨虚拟机侧信道的构建与攻击优化方案

基于微处理器架构的侧信道是多用户计算机系统中最常见的侧信道类型。然而在嘈杂的生产云环境中，基于微处理器架构侧信道 (隐蔽信道) 变得不那么稳定可靠。Hypervisor 的操作如虚拟机迁移、vCPU 调度以及租户的活动都会给信道引入各种噪声，对侧信道 (隐蔽信道) 的构建带来了很大挑战。本章提出两种典型的跨虚拟机侧信道：基于 Last Level Cache 的和基于内存总线的侧信道，并给出其构建与优化方案。方案能够克服这两种虚拟化环境中信道构建的困难，将信道载体所具备的特性充分利用，以达到实际环境下用于实施攻击的目标。

3.1 时间信道的建立

为了准确建模云环境下的时间信道，本章首先给出其数学抽象。以访问驱动的 Cache 时间信道为例说明该模型，然后再验证其普适性。

假设现有三个进程 Alice，Bob 和 Eve 均要访问高速缓存。每个进程都通过执行加载命令将内存地址载入高速缓存中, 因此，高速缓存中保存的总是最近的加载。在最简化场景下，假设每次进程加载, 都会加载大量数据，完全覆盖整个缓存。当 Alice 加载其数据时，如果没有其他人发出请求，Alice 的数据将保存在缓存中。当 Alice 再次加载同样的数据时，操作会更快，或者性能计数器指示 cache 命中，因此 Alice 能够得知，缓存中存放的仍然是她的数据。本章将共享资源，比如缓存，看作为一个具有数据存放能力的水桶，且每个进程都可以将水 (数据) 放入水桶。其中，水可以被不同用户的不同进程来标记。在每个进程加载操作，桶都被完全重新填装。因此，在任意时刻，桶中只有一个进程的水。在该模型中，每个进程可执行两种原语：填充 — 将自己的水装入桶中，探测 — 检查自己的水是否仍在桶中。通过这两种动作，即可建立稳定的时间信道。

3.1.1 基于 LLC 的隐蔽信道

与其他隐蔽信道介质相比，Cache 对于攻击者来说更具吸引力。因为其高操作速度可产生高带宽，并且不受软件系统限制，可以绕过许多高级隔离机制，所以近年来基于 Cache 的隐蔽信道备受重视 [93]。其中，最常用的一种交替通信技术是 Prime + Probe 方法 [94]。另一种类似的交替通信方法 Flush + Reload [26]，则是首先由接收者 Flush 一些共享内存，然后 Reload 以推断发送者的访问情况。

设通信双方是处于同一物理主机上的虚拟机 A 和 B，它们需要进行如下操作实现通信：

(1) B 首先用其自身数据来填充某商定好的 Cache 组 (Cache Set)；

(2) B 等待特定时间，此时 A 使用这块特定的 Cache Set；

(3) B 再次用相同数据来填充该 Cache Set，并测量这次操作所需时间。

3.1.1.1 访问时间的精确测量

在实现基于 LLC 的隐蔽信道之前，首先要解决时间的精确测量问题。处理器中的时间戳计数器 (TSC, Time Stamp Counter) 是目前计算机系统中最为细粒度的时钟源。而想要访问它，则需要使用 RDTSC 指令 [95]。从 Intel Pentium 系列处理器开始，80x86 系列的微处理器均引入了 TSC。它在每个 CPU 时钟信号 (CPU cycle) 到来时自增一。通过 TSC，软件就可以计算 CPU 的主频。如果想要准确的知道一段程序或一个函数的执行时间，则可以连续执行 2 次 RDTSC，相减得到这段指令执行过程中经历的 CPU cycle 数，再结合 CPU 主频，相除得到准确的运行时间。为了尽可能地减少测量函数本身所带来的误差，本章采用如下 C 语言内嵌汇编的方式来编写计时函数。

```
void access_counter(unsigned *hi, unsigned *lo){
        asm(``rdtsc\n''
            ``movl %%edx, %0\n''
            ``movl %%eax, %1\n''
              : ``=r'' (*hi), ``=r'' (*lo) :
              : ``%edx'', ``%eax'');
        return;
}
```

```
#define START_TSC(cl) __asm__ __volatile__ ( \
  ``cpuid \n\
    rdtsc \n\
    shlq $32, %%rdx \n\
    orq %%rdx, %%rax'' \
    : ``=a''(cl.r0) :\
    : ``%rbx'', ``%rcx'', ``%rdx'')
```

同时为了不影响 RDTSC 指令测量时由于处理器指令乱序执行带来的误差，使用 MFENCE、LFENCE、CPUID、RDTSCP 等能够强制指令按序执行的指令来确保被测量对象的一致性。为了进一步精确，可以使用宏进行预编译。

```
#define END_TSC(cl) __asm__ __volatile__ ( \
  ''rdtscp \n\
    shlq $32, %%rdx \n\
    orq %%rdx, %%rax \n\
    movq %%rax, %0 \n\
    cpuid'' \
    : ''=g''(cl.r1) :\
    : ''%rax'', ''%rbx'', ''%rcx'', ''%rdx''); \
    cl.ticks = cl.r1 - cl.r0
```

而由于现代的处理器缓存 Cache 是分级的。如何测量不同层级的缓存访问时间、正常内存的访问时间、甚至是内存行冲突或是内存总线冲突的准确时间，是构建侧信道之前需要了解的问题。本章将测量代码的 CPU 相关性设置为在其中一个核执行，并测量访问 LLC 中缓存的内存行的时间。为了确保我们测量 LLC 访问时间而不是 L1 或 L2 的访问时间，本章在测试时驱逐了 L1 和 L2 缓存的内存行。本章实验所用的 CPU 为 Intel i5-3470，其 L1 和 L2 均为 8 路组相联，而 LLC 是 12 路组相联。通过反复访问相同 LLC 集合的另外 10 条存储器内存行，由于这 10 条内存行在 LLC 中共享相同的 Cache 索引位，并且因为 L1 和 L2 的缓存集索引位都是 LLC 的 Cache 索引位的子集，所以我们知道这 10 条内存行必定与 L1 和 L2 上的 Cache 行冲突。L1 和 L2 的组相联路数为 8，因此访问这些缓存中的每一个缓存集中的 10 条内存行，必定能确保映射到这些缓存行是被逐出 L1 和 L2 缓存。

通过测试发现，访问 L1 需要约 2 个 CPU 时钟；访问 L2 约 30 左右个 CPU 时钟；访问 LLC 约需要 50 左右 CPU 时钟周期 (CPU cycles)；访问内存则需要 180 左右时钟周期；而如果发生内存行冲突，则访问内存需要约 250 以上的 CPU 时钟周期；内存总线冲突时，则需要花费约 3000 个时钟周期。

该测量结果同样可在本章3.1.2节使用。

3.1.1.2 HugePage

然而，除了要解决精确测量的问题，还需要解决两种地址映射不确定的问题。第一，用户层应用程序操纵的是虚拟地址，而 Cache 是物理标记的，数据在 Cache 中的位置由其物理内存地址决定。而物理地址和虚拟地址转换是由操作系统配合硬件 MMU 进行的，用户层很难获取这样的映射信息；第二，新式的 Intel 处理器使用了未在官方文档里声明的哈希函数来对 Cache 地址进行映射，即 LLC 的非公开 Hash 索引机制。在新式的 Intel 处理器中，为了增强有效性，LLC 被分为了多个分片 (Cache Slice)。每个 Slice 与物理内存地址之间的映射关系由上述 Hash 函数来确定，所以即使攻击者能

够定位到 Cache 行所在的 Cache 组，也无法知晓这些 Cache 行分分属哪个分片 [96]。文献 [96, 97] 提出了如何对该哈希函数进行逆向。但是这种逆向工程需要进行大量的实验，消耗大量不必要的时间。已有文献提出构建冲突集的方法来寻找能够映射到同一块 Cache Set 而虚拟地址不同的内存块。本章对这种方法进行了改进，提出了一种快速构建冲突内存块的方法，能够进一步地精确寻找冲突的 Cache Set。现描述如下。

在介绍本侧信道实现方法前，本章对所需的内核大页机制 (HugePage) 进行简要介绍。当进程使用内存时，CPU 会标记并分配该进程所使用的 RAM。为了提高效率，CPU 通常以 4K 的字节块为单位分配 RAM。由于进程使用虚拟地址进行映射，所以 CPU 和操作系统必须维护页面与进程的映射关系以及页面存储的具体位置。显然，当拥有的页面越多时，查找内存映射的时间就越多。当进程使用 1GB 的内存空间时，就存在 262144 条映射关系。如果每个页表项占用 8 字节，查找页面最大需要遍历 2MB 的内存空间。此外，在以数据库应用为例的内存访问密集型应用中，如果共享内存的某一部分被交换 (swap) 到硬盘上，那么再次访问该处就会花费非常多的额外时间。同时，当数据库管理系统或操作系统本身的占用内存较大时，维持内存页的管理或访问也需要更多的时间。

当前大多数 CPU 架构支持大页面的操作，这样带来的最为直观的好处就是节省页面的查找开销，提升查找密集型程序的性能。这样的大页面操作被称为 HugePage 或 SuperPage 机制。

以 Linux 内核为例，为了解决上述问题，Linux 2.6 内核之后实现了一种新的页表管理机制 HugeTlbfs。它不使用以往的 4K 页内存管理方式，而使用 2M 和 1G 的更大页面来对应用程序进行物理页面的映射和管理。其中，hugetlb 是 TLB 中指向 HugePage 的一个页表项 (通常大于 4K)。也就是说，HugePage 则是通过 hugetlb 的大页页表项 (hugetlb entries) 来实现。这些分配的 entry 就作为内存文件系统 hugetlbfs 提供给进程使用。这些大页除了页大小比常规页大以外，还不会被换出，这一功能更进一步地解决了上述问题。

HugePage 需要额外配置才能被使用。在 x86_64 体系上，可以通过查看虚拟文件系统 “/proc/cpuinfo” 来查看机器是否支持 HugePage。若其中具有 PSE 标志，则支持 2M 页面；若具有 PDPE1GB 标志，则支持 1G 页面。在使用前，首先需要更改用户配置文件 “/etc/security/limits.conf”，设置 memlock 的限制。然后需要修改文件 “/etc/sysctl.conf” 中的 vm.nr_hugepages 参数，从而配置可供使用的大页页数。

3.1.1.3 基于 LLC 的隐蔽信道实现算法

首先，使用 LinuxHugePages [98] 为发送方和接收方分配足够多的大页 (2MB page)。假设发送方和接收方以 Cache Set x (x 为 Cache Set ID) 作为其隐蔽信道的载体，如

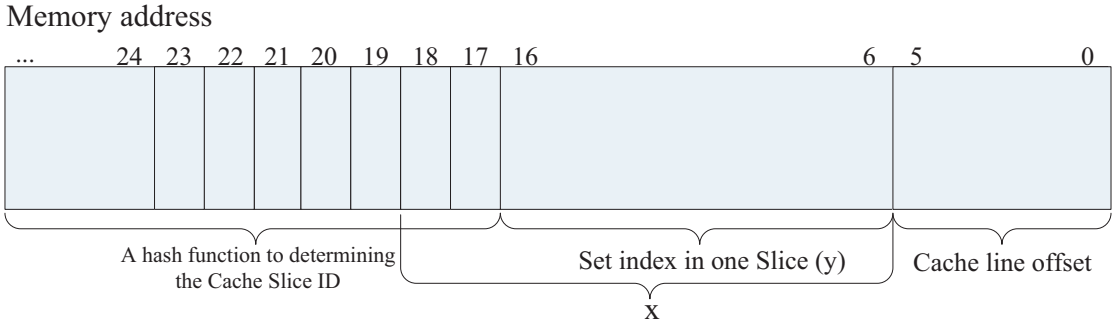


图 3.1 内存与新型 LLC 的映射索引

图3.1。此时通信双方只能定位到每一个 Cache 分片的索引值 y(第 16 位到第 6 位)，而 Slice ID 是未知的。

针对该问题，文献 [18] 提出了一种构建驱逐集的方法来寻找能够映射到同一块 Cache Set 而虚拟地址不同的内存块。本章对其稍作改进，提出构建驱逐集算法如3.1所示。

通过将具有同样元素的驱逐集合并，本章构建驱逐集的方案更加圆满，能够在每个 Slice 对应的驱逐集中找到超过 12 个 (组相联路数) 可供驱逐的内存行 (line)。这样一来，在隐蔽信道构建时，可以更加灵活地选用任意的 Slice ID 和 Set ID 作为 LLC 冲突集的 ID。

而在本方案中，由于实验所用 CPU 为 4 核，这时发送方在上述通信流程的第二步中可以同时访问：00 | y，01 | y，10 | y，11 | y 四块内存块，来清空目的 Cache Set。由于 LLC 分片的共享环机制 [18]，连续地访问四块 Slice 能保证该 Cache Set 一定被发送方的数据所占据。该现象同时证明了实验所用 CPU 的 Hash 索引函数中包含 “$17 \oplus 18$” [97]。同理，该方案也适用于 CPU 核数为 2 的整数次幂的情况。

使用上述方案目的主要是为了能够避免处理虚拟化环境下虚拟地址到物理地址之间的映射，从而构建跨虚拟机的 Cache 侧信道。(最外层 Cache，即 LLC 是物理地址索引的。) 如果仍想要还原虚拟地址和物理地址之间的关系，从而逆向出决定 LLC 分片的哈希函数，则需要使用 Linux /proc/pid/pagemap 接口[1]来获取所分配内存的物理地址。

本章提出的基于 LLC 的隐蔽信道算法, 如算法3.2所示。

3.1.2 基于内存总线的隐蔽信道

内存总线负责处理器与主存之间数据传输，是现代计算机系统中不可或缺的组成部分。处理器内的 CPU 核心，都共用该内存总线。所以，内存总线上的冲突将会导致系统范围可观测的内存访问延迟。因此，利用某些特定的程序行为所触发的内存总

[1]https://www.kernel.org/doc/Documentation/vm/pagemap.txt

表 3.1. 冲突集构造算法

输入: 候选内存行 lines.

输出: 驱逐集，每个分片一个驱逐集 eviction_set.

```
1 function probe(set, candidate) begin
2   读取 candidate;
3   foreach l in set do
4     读取 l;
5   endfor
6 endfunction
```

```
1  随机化 lines;
2  conflict_set = {};
3  foreach candidate ∈ lines do
4    if not probe(conflict_set, candidate) then 将 candidate 放入 conflict_set;
5    endif
6  endfor
7  foreach candidate in lines − conflict_set do
8    eviction_set_before = {};
9    if probe(conflict_set, candidate) then eviction_set = {};
10     foreach l in conflict_set do
11       if not probe(conflict_set−{l}, candidate) then 将 l 放入 eviction_set;
12       endif
13     endfor
14     if eviction_set ∩ eviction_set_before ≠ ∅ then
       eviction_set_before = eviction_set ∩ eviction_set_before
15     eviction_set_before = eviction_set
16     输出 eviction_set;
17     conflict_set = conflict_set−eviction_set;
18   endif
19 endfor
```

线冲突就可以构建出一种隐蔽信道 [33]。x86 架构中，处理器利用总线锁 (Memory Bus Lock) 实现内存原子指令。对于未对齐的内存区域，执行内存原子指令会触发内存总线锁。利用这种机制，可以构建一种高速、可靠、跨虚拟机的隐蔽信道，如图3.2。

表 3.2. 基于 LLC 的隐蔽信道算法

line_1：发送 1 需访问的行 line_0：发送 0 需访问的行 Dsend[N]: 发送方发送的 N 位数据	
发送方： 1 for $i = 1$ to N–1 do 2 if $DSend[i] = 1$ then 3 for 一段时间 T_{mark} do 4 访问 line_1; 5 endfor 6 循环等待一段时间 T_{pause} 7 else 8 for 一段时间 T_{mark} do 9 访问 line_0; 10 endfor 11 循环一段时间 T_{pause} 12 endif 13 endfor	接收方： 1 for 一段时间 $T_{monitor}$ do 2 探测 set_1; 3 探测 set_0; 4 endfor

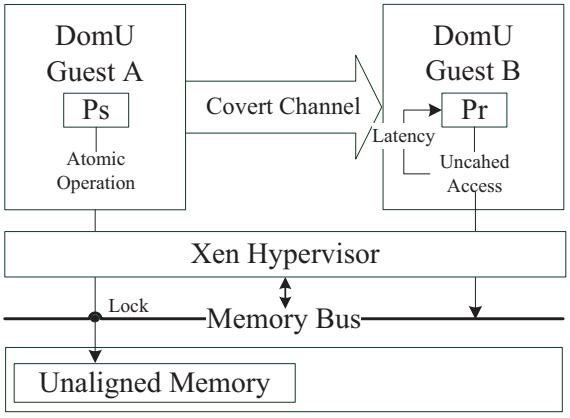


图 3.2 基于内存总线的隐蔽信道

3.1.2.1 内存总线冲突

内存总线冲突 (Memory bus contention) 是计算机总线系统中不经常发生的事件，大部分情况下是由于当多个内存设备尝试着将输出值同时输出到总线时而引发。通常，连接到内存总线的电路是预先设计好的，以尽可能性降低内存总线的发生概率，CPU 芯片也会在以配置好的速率工作，以免发生总线冲突。然而，如果总线在某些情况下，故意地被驱动得太快，则此时可能被干扰而导致无法正常工作。当不合法的值被写入

寄存器中时，内存总线冲突也可能出现，同样也会导致内存总线冲突。总线冲突如果未能及时正确地处理，可能会导致程序终止甚至系统崩溃。

长期以来，总线竞争信道一直被视为对虚拟机的潜在安全威胁，文献 [33] 首次发现了一个使用奇特操作数的原子指令 (算法中将其称为 M_{Exotic}) 的总线争用攻击。原子指令是特殊的 x86 内存操作指令，旨在促进多处理器同步，例如实现互斥和信号量的并行计算的基本构建块。原子指令 (即原子内存操作) 执行的内存操作保证完成不间断，因为其他处理器或设备对受影响的内存区域的访问被暂时阻止执行。这样一来，如果攻击者能够构造特定的原子操作，就能使得总线在一段时间被占用，从而实现攻击原语。早期的 x86 处理器 (Pentium Pro 之前) 通过使用总线锁定来实现原子存储器操作，该总线锁定专用硬件信号提供设备对存储总线的独占访问。

然而，现代的 x86 处理器 (Intel Nehalem 和 AMD K8 / K10 之前) 通过显着降低内存总线锁定的可能性来改善原子内存操作的实现。具体而言，通常情况下，在高速缓存的存储区域上执行原子操作，此时相应的高速缓存行会被锁定，而不锁定存储器总线。这样一来，内存总线冲突就无法造成了。

但是，原子内存操作仍然可以用于隐蔽通道，因为总线锁定的触发条件并没有被消除。当在具有特殊配置的内存区域上执行原子操作时 (跨越两个高速缓存行的未对齐的地址，即 M_{Exotic} 操作)，由于 Cache 行的锁定无法保证原子性，并因此 CPU 会发出总线锁定信号。在虚拟化环境中，因为没有共享内存总线可以锁定，所以原子性是通过一系列复杂得多的操作来实现的：所有处理器必须协调并完全刷新先前发出的内存事务。所以从某种意义上说，通过 “模拟” 旧平台 (Pentium Pro 之前) 的总线锁定行为，在新平台 (Intel Nehalem 和 AMD K8 / K10 之前，Pentium Pro 之后) 上处理特殊的原子内存操作时，仍然能造成总线冲突。尽管没有共享内存总线，内存访问延迟的影响仍然是可观察的。这一特性就能被攻击者所利用于隐蔽信道的构建中。

3.1.2.2 基于内存总线的隐蔽信道实现算法

设通信双方 A 和 B 位于同一物理主机，信道需要传送的消息为 $DSend$。首先 A 将 $DSend$ 分割成多个数据帧，每个数据帧使用差分曼彻斯特码来编码。而后，逐比特发送每个数据帧 $DMSend$ 的内容。在时钟周期 T 内，若需要发送的第 i 位 $DMSend[i]$ 为 1，则 A 持续做原子操作；若需要发送的比特为 0，则 A 阻塞。同时，B 在时钟周期 T 内不断访存，计算平均访存时间 λ，若 λ 大于阈值 $threshold$，则 B 收到信息 $DMRecv[i]$ 为 1，反之 $DMRecv[i]$ 为 0。

同样地本章对现有的基于内存总线的隐蔽信道进行了改进，当发送方使用特定的上文所述原子操作 M_{Exotic} 时，此时接收方不适用正常的访存操作，而使用 Streaming SIMD Extensions (SSE) 指令集的某些访存指令操作，避免访问 Cache，使得正常访问

内存总线时的访存时间不会由于 Cache 命中而影响，从而保证每次正常访存 (总线不冲突) 时的时间基本一致，进而保证了信道传输的准确性。

本章提出的基于内存冲突的隐蔽信道算法, 如算法3.3所示。

表 3.3. 基于内存总线冲突的隐蔽信道算法

M_{Exotic} ：异常操作数原子指令。 $DSend[N]$，$DRecv[N]$：发送和接收数据。	
发送方：	接收方：
1 for $i = 1$ to $N - 1$ do	1 for $i = 1$ to $N - 1$ do
2 if $DSend[i] = 1$ then	2 for $i = 0$ to $T - 1$ do
3 for $j = 0$ to $T - 1$ do	3 内存访问操作;
4 M_{Exotic} 操作;	4 endfor
5 endfor	5 if $Mean(AcessTime) > Threshold$
6 else	6 $DRecv[i] = 1$;
7 for $j = 0$ to $kT - 1$ do	7 else
8 普通内存写操作;	8 $DRecv[i] = 0$;
9 endfor	9 endif
10 endif	10 endfor
11 endfor	

为了提高信道可靠性，本章所实现的两种隐蔽信道采用了如下优化机制。

(1) 时钟同步：由于发送方和接收方分属不同的虚拟机，它们之间的调度差异将导致数据传输和检测过程不同步，进而对基于定时的数据调制造成很大的问题。本章通过自时钟编码克服时钟失步，并使用差分曼彻斯特编码传输数据位。

(2) 接收确认：由于内存的随机擦除错误有可能导致传输的数据非常不连续，明显降低其可用性。针对这个问题，发送方通过观察由接收方引起的 Cache miss 或内存访问延迟来判断接收方的存在。

此外，本章通过使用数据帧 (Frame) 来加强通信协议对时钟漂移和调度中断的弹性。传输的数据被分成固定长度，发送方使用伪随机数生成算法来产生每一个数据帧的载荷，并用起始和停止模式将每个数据段构造成数据帧。这具备两个优点。首先，当发送方检测到传输中断时，不需要重传全部数据，仅需重发相关数据帧。其二，在传输期间总是不可避免地会丢失部分数据，利用数据帧，接收方能更加容易地定位错误，进行纠错处理。

3.2 实验与分析

为验证有效性，本章以 Xen 为虚拟化平台，对这两种基于隐蔽信道的同驻检测技术进行了编码实现与一系列实验。新建的虚拟机实例 A、B，分别运行发送方和接收方的程序。其中实验平台的采用 Xen 4.4.2 作为 Hypervisor，CPU 采用 Intel i5-3470，虚拟机内核版本为 Linux 3.14.60，每台虚拟机分配 1024MB 内存。

发送的数据帧序列由伪随机产生器算法产生 [99]。由于在隐蔽信道的工作过程中，经常会发生传输序列的某一比特位或几位丢失、翻转和错误检测而导致的比特位增加。因此不能使用常规的序列按位比对来计算传输正确率。序列比对算法 Needleman-wunsch 算法 [100] 此处被用于隐蔽信道的正确率计算。

基于内存总线的隐蔽信道实现包含三个参数，$threshold$、T、S。其中 $threshold$ 为判断码元为 1 的延时阈值，T 为每个码元执行的原子操作次数，S 代表码元周期，即 S 个连续的 0 识别为一位 '0'，S 个连续的 1 识别为一位 '1'；$threshold$ 的设定对接收方能否识别信号，以及建立通信起着决定性作用，而 T 和 S 两个参数的配置则对隐匿信道的传输正确率具有显著影响，如图 3 所示。基于 LLC 的隐蔽信道也包含三个参数：$threshold$、S、T_{pause}。T_{pause} 即 Prime + Probe 方法中的发送方暂定延时 (Pause duration)。

根据图 3.3(a)，我们发现基于内存总线的隐蔽信道传输正确率基本上与 T 成正比，这是由于每个码元执行的原子操作次数越大，其产生的时间周期和平均延时更加均匀、稳定，从而提高了传输正确率。然而 T 的增大会降低传输速率，因此在以传输数据为目的的隐蔽信道中，T 的取值需权衡传输正确率与传输速率。对于基于 LLC 的隐蔽信道，如图 3.3(b)，传输正确率随 S、T_{pause} 同时变化。

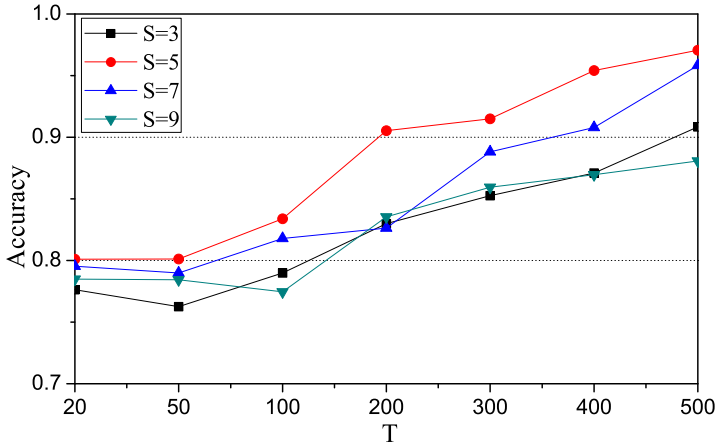
可以看到，码元周期并不是越高越好。对于基于内存总线的隐蔽信道，如图 3.3(a)，当 S 为 5 时，传输正确率最佳，因为当其过低时，任意码元的错误都很可能会导致接收数据的错误；当 S 的取值过于大时，某一个错误的位将导致整个码元无法识别。因此码元周期在一个适中数值时才能达到最佳的传输正确率。

3.3 基于码元识别的信道优化算法

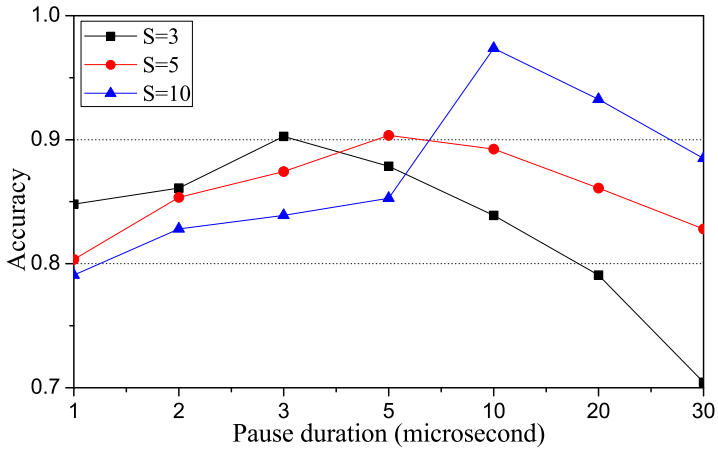
为了克服当 S 取值过大而导致的码元无法识别的缺陷，本章引入了码元识别技术。同时为了克服信道在嘈杂的公有云环境下传输正确率会受到影响的问题，以此来提升传输正确率。经过优化过的信道传输率在较大程度上得到了提升。

3.3.1 信号的初筛与平滑滤波

由于嘈杂的云环境带来的噪声，信道若未进行去噪便进行码元识别操作，则会引入相当大的传输错误。因此，信道去噪机制必不可少。



(a) 基于内存总线的隐蔽信道



(b) 基于 LLC 的隐蔽信道

图 3.3 不同的参数对传输正确率的影响

平滑滤波器 (Smoother) 是增加低频的信号过滤技术，在图像处理领域被广泛应用。它不经由傅立叶转换，直接处理影像中的像素，主要用于模糊化和去除噪声。本章利用滑动平均滤波来实现信号的初筛。平滑滤波器的输出是滤波器邻域的平均，过滤窗口越大平滑的效果越好，然而若窗口过大平滑效果会使边缘的信息失真越严重，使输出的信号过度模糊，因此需要合理选择窗口的大小。在本方案中，窗口大小 W_{size} 与码元 S 周期设为相同，这样既符合接收码率与滤波速度一致，还能够最大可能地不遗漏任何有效比特位。

滤波算法较为简单，表示如下：

$$\bar{X_n} = \frac{1}{N}\sum_{i=0}^{N-1} X_{n-i} \tag{3.1}$$

其中，X_{n-i} 为第 $n-1$ 次采样值。$\bar{X_n}$ 为第 n 次采样经过滤波后的结果，N 为平

均滑动滤波点数，即 S。

3.3.2 识别连续的信号与帧内容

经过去噪操作后，所得到的点已经较为准确。此时，接收到 '0' 和接收到 '1' 的区别已经较为明显。(在基于内存总线的信道中，发送 '0' 仅消耗约 100 个 CPU 时钟周期，发送 '1' 需消耗几千个 CPU 时钟周期。在基于 LLC 的信道中，发送 '0' 消耗约 30 个 CPU 时钟周期，而发送 '1' 需要消耗约 100 个时钟周期。) 因此，采取聚类的方式来区分接收到的信息即可。

在此本章采用最为简单有效的 K 均值聚类法来实现该功能，具体如下：

由于在此情况中，仅需要将数据分为两类，所以，仅需要求出聚类中心，之后再进行简单判断即可。由于不用考虑时间序列，需要聚类的数据仅为 1 维的数据。所以，每帧数据的聚类中心距 d 计算为：

$$d = \frac{1}{m} \sum \|X_i - X_j\| \tag{3.2}$$

其中 m 为一帧内采样个数，t 为最优分类界限，易得：

$$t = \frac{1}{m} \sum X_n \tag{3.3}$$

3.3.3 精确的同步

在通信过程中，为了减少重发次数，节省时间上的开销，必须确保发送方和接收方的时隙是对齐的。如果不对齐，两者将在重叠的时间争用该信道并引入位翻转错误。此外，这种不受限制的争用导致 '0' 和 '1' 的值的可分离分布较少，这又迫使发送者和接收者更剧烈地竞争 (即更长的持续时间或更高的共享资源使用)。未对齐的时间段会降低信道容量。另一方面，即使使用低幅度信号 (如采用较小的码元周期 S)，实现信号的对齐也能够实现更精确的竞争和更低的错误率。

通常，发送方和接收方 VM 均使用自身内部的系统时钟来计时，而本章的目标即为确定其内部时钟之间的时间偏移。通过实现时钟同步协议来完成发送接收双方的对齐操作。在本章的安全假设中，当建立隐蔽信道的目标时允许明确的同步。这是因为许多场景允许隐蔽通道的发送者和接收者明确地进行通信。例如，在本章中，发送者和接收者是友好的，并且秘密沟通来检测恶意的共同租户。因此，它们被允许首先通过 socket 明确地进行通信以同步它们的时钟。协议的第一阶段类似于网络时间协议，其中发送方和接收方通过显式通信交换时间戳，以使其内部时钟在 100 微秒内对齐。在第二阶段，对系统时钟的偏移进行细微更改。最后，发送方发送引导信号 (pilot signal)，以便接收方清楚地检测到通信的开始并进行明确的沟通。从实验中发现，发送方和接

收方虚拟机之间的挂钟定时器可能相差数秒钟。因此所提出的显式的时钟同步协议允许进程在几百微秒内同步其内部时钟。

本章提出的同步协议 3.4可能较慢，但更精细，能够实现在几百微秒之内的时钟差异内，对齐传输速率不超过 10 Kbps 下可靠的通信。我们通过在隐蔽通道上传送一个已知的比特序列来进行多轮处理。在每次通信之后，发送方将其时钟移动一微秒，稍微改变两个进程争用共享资源。随着时钟变得更加同步，两个进程的争用变得更加一致，通信变得越来越可靠，如算法 3.5所示。一旦达到最佳周期对齐，发送方的每次移位都会降低通道间的通信。此时接收方通过测量这个峰值便可能定位到最佳的移位大小。同步过程的耗时取决于用于测试的包大小，基本上每个虚拟机执行只需要执行一次即可完成同步。

一旦 VM 的时钟对齐，接收者只需要知道发送者何时正在进行通信。为此，发送者使用 (pilot signal) 来标记每个消息的开始。在验中，本章使用由 500 个 '10' 比特对然后是 100 个连续的 '0' 来组成引导信号。我们选择这个序列是因为：(1) 重复 10 的模式产生大量的振荡，接收方可以很容易地与噪声区分开来；(2) 100 个连续的 '0' 使得接收方能够检测到平坦无波动的信号，并在重复接收到 '10' 模式时就进行解码操作。

表 3.4. 同步协议

发送方：	接收方：
1 for n < 1 to 100 do	1 for n < 1 to 100 do
2 receive_request();	2 receiver_clock = get_time();
3 clock_time = get_time();	3 request_clock();
4 send_clock(clock_time);	4 sender_clock = receive_clock();
5 endfor	5 latency = get_time() −receiver_clock;
	6 sender_clock -= latency/2;
	7 total_offset += (sender_clock −receiver_clock)
	8 endfor
1 clock_offset = receive_offset();	
	1 offset = total_offset/100;
	2 send_offset(offset);

表 3.5. 通信双方对齐微调算法

发送方：	接收方：
1 for n < 1 to 20 do	1 for n < 1 to 20 do
2 send_covert_bits(10000);	2 receive_signal(data_array[n]);
/* 微调 1 微秒 */	3 end for
3 start_time = start_time + 1;	
4 end for	
1 clock_offset = receive_offset();	
	1 best_round = calculate_best_round(data_array);
	2 offset = find_offset(best_round);
	3 send_offset(offset);

3.3.4 其他优化策略

3.3.4.1 离线分析确定信道参数

例如，如果信道基于 Prime + Probe 技术，则发送器和接收器必须就信道参数 (诸如频率，即，每比特的时隙持续时间) 和每个比特内的通道参数达成一致，即访问时间和 Prime / Probe 的持续时间要基本保持相同水平。在实施攻击开始前，必须根据目标机器的特性，分析目标机器以确定这些参数; 例如，我们需要确定每个要执行的加载指令的数量和要访问的地址的范围。

3.3.4.2 阈值的重新校准

通信过程中，发送方和接收方必须就信道参数 (诸如频率，即每比特的时隙持续时间) 和每个参数达成一致。于是在实施攻击前，必须根据目标机器的特性，分析目标机器以确定这些参数。

在离线分析参数阶段，通过微调 *threshold* 的值来使得通信效率达到最大，并将其应用于真实攻击场景中。具体来说，攻击者通过求极值的手段来确定当阈值取何值时信道传输正确率最大，并且在之后的攻击中，使用该参数测量和判断。参数的设定对接收方能否识别信号，建立通信起着决定性作用，而两个参数的配置则对隐匿信道的传输正确率具有显著影响。由于每个码元执行的原子操作次数越大，其产生的时间周期和平均延时更加均匀、稳定，从而提高了传输正确率。然而由于 T 和 S 的增大，会降低传输速率。因此在以传输数据为目的的隐蔽信道中，阈值的取值需权衡传输正确率与传输速率。

3.4 本章小结

本章介绍了两种主流的侧信道，给出了它们的构建方案和针对他们的特定优化方案。然后结合信道的特点，给出了通用的跨虚拟机侧信道优化方案，使之能够用于现实攻击场景中。经过试验发现，经过优化后的两种隐蔽信道均可达到 95% 以上的传输正确率，均优于原方案 [18,33]。

4 面向云平台的虚拟机同驻方案

如果攻击者意图实现与目标虚拟机同驻，则必须拥有检测攻击者虚拟机是否与目标虚拟机同驻的能力。于是，在描述同驻实现方案前，本章提出其基础 — 基于隐蔽信道的同驻检测方案。随后利用该同驻检测方案，提出并设计了基于虚拟机洪泛的同驻方案。

现有研究中，基于网络信息的同驻检测方法实现简单，但准确率较低；基于资源干扰的同驻检测方法实现复杂。而随着硬件的不断发展以及 VMM 资源调度机制的优化，传统的虚拟机同驻检测方法逐渐不适用于新的硬件设备上 [101]。基于硬件的隐蔽信道 [53] 作为云平台里不可避免的存在，可被充分地应用于同驻检测中 [102]。因此，本章以准确性、高效性和实用性为目标，提出基于隐蔽信道的同驻检测方案，并以上一章提出的基于内存总线冲突的隐蔽信道和基于 Last Level Cache(LLC) 的隐蔽信道为例，实现了该同驻检测技术。在现有虚拟机洪泛策略的研究上做出了改进，使得攻击者同驻攻击成功率增加，并且同样不需要额外的社会工程信息，使攻击者可以有效地利用云平台平行放置的部署算法来实现一定概率的攻击者实例与受害者实例同驻。

4.1 云计算环境下同驻威胁模型

攻击者想要实现其控制的虚拟机能与被攻击虚拟机被云平台管理系统分配到同一物理主机，看似天方夜谭，其实有规律可循。文献 [9] 通过虚拟机洪泛的方式可以将此同驻攻击概率提升到 8.4%。即如果事先能够了解到被攻击虚拟机开启的精确时间，使用简单的 VM flooding，就能让攻击者所拥有的约总量 1/12 的虚拟机开启在处于被攻击者的物理主机上。本章在此基础上做出了改进，使得攻击者同驻攻击成功率增加，并且同样不需要额外的社会工程信息。在此将攻击者的威胁模型描述如下。

恶意的虚拟机可以利用侧信道或共享隐蔽信道的方式来窃取或者泄露敏感数据。文献 [50] 提到，在云环境中侧信道可分为三类：虚拟域内侧信道、虚拟域间侧信道 (跨虚拟机侧信道) 和跨物理平台侧信道。虚拟域内侧信道，即隐藏进程造成的侧信道，可以通过多种方式检测并消除 [78,79]。对于跨物理平台的侧信道，由于无法共享硬件资源，因此仅存在基于网络的信道。综上，本章仅考虑利用虚拟域间的侧信道 (跨虚拟机侧信道) 来实现云环境下的虚拟机同驻。

在公有云环境中，理论上攻击者可以通过合法渠道获得部分资源，即攻击者能够创建任意的虚拟机并对其进行管理。因此，本章基于该事实，假设：

(1) 云平台系统有良好的隔离性，云平台管理系统和其上的客户操作系统均是干净且安全的 (及时更新，补丁完备)，同时云管理员的操作均是是善意且合规的，不存在内部威胁 [23]。如此，本章中的攻击者仅通过侧信道的方式来实现与目标虚拟机同驻；

(2) 云平台的各个租户之间是互不信任的，但不具备任何特殊权限 (甚至可以不具备操作系统权限)。攻击者的目标为使用云平台上的正常租户，仅通过侧信道来获取其他用户的秘密信息。

(3) 假设攻击者有足够的资金来保证其能够开启足够多的虚拟机。同时，攻击者还能够通过社会工程的方式获取到目标对象开启虚拟机的准确时间。

上述假设均符合实际的行业应用场景，在现阶段的商用云平台中，大部分主流云平台如 Amazon EC2、MS Azure 等为了保护其上的租户虚拟机，均采取了某些强制的安全手段。

4.2 虚拟机同驻检测

现阶段国内对于虚拟机的同驻检测方面的研究甚少。其中，梁思等人提出基于隐蔽信道的同驻检测方案效果好，但难以实现 [101]。针对该问题，本章提出一种基于隐蔽信道检测和虚拟机洪泛策略的虚拟机同驻方案来实现真实情况下的虚拟机同驻。首先，利用第3章所实现的隐蔽信道作为检测工具进行同驻检测实验，然后通过分析云环境下隐蔽信道机理，提出具体的判定方案，并验证其效果。

4.2.1 同驻判定

4.2.1.1 同驻判定阈值计算

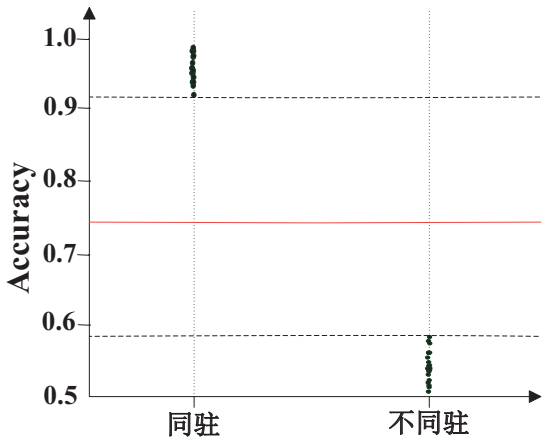


图 4.1 传输正确率的多次采样

为了确定传输正确率与虚拟机是否同驻之间的关系，在分布在两台物理主机上的共 10 台虚拟机之间进行隐蔽信道通信实验。以基于内存总线的隐蔽信道为例，其传输正确率与同驻状态，如图4.1所示。

其中，实际同驻的虚拟机共 20 对，非同驻的虚拟机共 25 对。实验中，同驻虚拟机传输正确率最小值为 90.7%，非同驻虚拟机传输正确率最大值为 57.9%。两类样本间有明显间隔，因此可以通过设定传输准确率阈值判定虚拟机是否同驻。

实际上，当隐蔽信道的参数 (如 S，T 等，详见第3章) 一定时，同驻虚拟机传输正确率与非同驻正确率均为客观上的定值。由于实验数据的局限性和某些可能出现的噪声的因素，传输正确率符合正态分布。而为了确定虚拟机是否同驻的判定阈值，需要先由样本估计出正态分布中 μ 与 σ 的值，然后经过计算得到最优的判定阈值。

记同驻虚拟机传输正确率 X_t 服从正态分布 $N(\mu_1, {\sigma_1}^2)$，非同驻虚拟机传输正确率 X_b 服从正态分布 $N(\mu_2, {\sigma_2}^2)$。由极大似然估计有

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.1}$$

$$\hat{\sigma^2} = \frac{1}{n}\sum_{i=1}^{n} {x_i}^2 - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)^2 \tag{4.2}$$

于是由公式4.1和4.2得

μ_1 =0.946，σ_1 =0.0233 ，μ_2 =0.540，σ_2 =0.233 。

同时，记 $\Phi(\bullet)$ 为标准正态分布的概率密度函数，则同驻时传输正确率分布函数为:

$p(x \mid \mu_1, \sigma_1) = \Phi\left(\frac{x-\mu_1}{\sigma_1}\right)$，

不同驻时正确率分布函数为:

$q(x \mid \mu_2, \sigma_2) = \Phi\left(\frac{x-\mu_2}{\sigma_2}\right)$。

本章认为漏判率与误判率占有同样的比重，因此引入 F_1 分数来衡量 x 的准确性，此时

$$F_1 = 2\frac{pq}{p+q} \tag{4.3}$$

最优判定阈值应满足：

$\hat{x_t} = \arg\min F_1$

为使 F_1 分数在 x 的取值区间内取得最大值，而由于 x_t 无解析解，极值无法计算。在此通过查表，以一定精度搜索 x_t，得到其极值解为 $x = 74.3\%$，如图 4.1中红线所示。

4.2.1.2 同驻判定效果评估

通过上述计算得到判定阈值后，重新在三台同样的物理主机上的共 15 台确定编号的虚拟机进行隐蔽信道通信实验。为了尽可能的模拟实际云平台环境，所有虚拟机在每隔 15 分钟进行一次随机迁移 (目标主机不确定)。

以基于内存总线的隐蔽信道实现为测试工具，同样在每 15 分钟内，对一共 105 对虚拟机进行同驻检测。每次检测持续 8 秒，持续测量 1 小时，实验结果如表 4.1。

表 4.1. 同驻检测效果评估

实际同驻对数	判断正确次数	误检率
218	217	4.6‰
实际不同驻对数	判断正确次数	漏检率
202	202	0

通过表4.1可以看出本章提出的基于共享硬件隐蔽信道的同驻检测方法具有极低的误检率和漏检率。这一方面是由于隐蔽信道的普适性，只要能够共享硬件 (CPU Cache 或内存总线)，在任意虚拟化平台、任意操作系统中都能通用；另一方面，在上一章中对两种隐蔽信道的实现作出了改进，充分考虑到云环境中的噪声和其他不良影响，对信道的通信进行了针对性的优化。随后，在本章所提出的同驻检测方法实验室环境中获得充分成功的前提下，将此阈值应用于商用云平台的同驻检测中，通过 VM flooding 碰撞的方式使两台云实例同驻。

4.2.2 容器实例的同驻检测

在以容器环境为主的云中，由于容器同样共享 CPU 等硬件，因此容器间也存在经由侧信道导致的数据泄露。不过，由于跨容器侧信道和跨虚拟机侧信道模型一致，原理其实相差无二。当在进行同驻验证时，同样可以借助已有的跨虚拟机侧信道来进行检测。

以 Linux Docker 为例，本章进行了容器实例的同驻检测。同样，本章在两台物理主机上创建了各 5 台 Ubuntu 容器实例。其中，实际同驻的 Docker 容器共 20 对，非同驻的 Docker 容器共 25 对。实验中，同驻容器传输正确率最小值为 94.2%，非同驻传输正确率最大值为 55.0%。两类样本间有明显间隔，因此同样可以通过上述设定传输准确率阈值 (74.6%) 判定虚拟机是否同驻。

测定阈值完，随后本章在三台用于测试的物理主机上的 15 台容器实例中进行了隐蔽信道通信实验，并以其传输正确率来判定容器是否处于同一物理主机。实验结果如表4.2。

实验证明，利用隐蔽信道检测容器是否同驻同样有效，其有效程度甚至高于检测虚拟机同驻的情况。

表 4.2. Docker 容器同驻检测效果评估

实际同驻对数	判断正确次数	误检率
30	30	0
实际不同驻对数	判断正确次数	漏检率
75	75	0

4.3 基于后验概率的虚拟机实例洪泛策略

由于虚拟机同驻造成的危害越来越大，研究人员在云平台的虚拟机投放策略上提出了抵御同驻危害的方法。Li 等人 [103] 面向可能发生的故障而导致的最坏情况，提出了一种基于 t-packings 组合结构的虚拟机部署策略；Moon 等人 [22] 提出了一种虚拟机部署算法，它利用由云提供商协助的虚拟机迁移服务 (VM Migration)，通过不断地重新部署 VM，使得任意两台 VM 无足够长的同驻时间，从而限制由侧信道引发的信息泄漏。上述方法虽然能够在一定程度上抵御同驻威胁，但是仍无法从本质上防止虚拟机实例洪泛造成的同驻。

本节提出了一种基于后验概率的自动化 VM flooding 方案。虚拟机实例洪泛即攻击者在同一时间在同一个云平台可用区，开启大量的虚拟机实例。这些实例可以是攻击者利用自身帐号投放，也可通过其所控制的虚拟机僵尸网络投放。当攻击者实例与受害者实例同时开启时，或其开启时间接近受害者虚拟机实例开启时间时，那么攻击者由很大几率能够与受害者同驻相当长的时间。并且，实例开启时间相近时，攻击者可以有效地利用云平台平行放置的部署算法来实现大概率的虚拟机同驻。而想要做到这一点并不困难。由于云计算的动态特性，服务器通常在实例上运行，不需要时就终止，稍后再运行。这样一来，攻击者可以通过不停监控服务器的状态 (如通过不间断的服务访问或网络探测)，直到发现实例消失时又继而重启时，推断出虚拟机的开启时间，然后进行实例洪泛。攻击者甚至可以利用该特点主动触发新的目标虚拟机实例。

通过所提出的虚拟机洪泛方案，可以计算出：(1) 被攻击虚拟机的分布在各台物理主机的概率；(2) 本次投放 VM flooding 虚拟机时，每一台攻击虚拟机与被攻击虚拟机同驻的概率。该方案的总体思路为先大量投放虚拟机，再对所投放的虚拟机进行同驻检测，将确定为同驻的虚拟机归类，最后对所投放的虚拟机进行约减。具体步骤如下。

首先，通过云平台提供的合法途径，同时申请一批虚拟机集群。该批虚拟机集群中，记一台虚拟机落在各台物理机的概率为 $\mathbf{p} = (p_1, p_2, ..., p_m)$，$m$ 为物理主机总数，且

$$\sum p_i = 1 \tag{4.4}$$

本章对 $\mathbf{p}$ 作出估计，从而确定 VM flooding 所能达到的最大同驻概率。

当外部请求发送到云平台时，记请求开启虚拟机实例数量为 n，经过同驻检测，得知所有的 n 个虚拟机实例在 m 台物理平台 (物理主机) 上的分布情况为 $\mathbf{x} = (x_1, x_2, ..., x_m)$，$x_i$ 表示第 i 台物理机上的实例数，则有

$$\sum x_i = n \tag{4.5}$$

在实验结果上有

$$P(\mathbf{x}) = \prod {p_i}^{x_i} \tag{4.6}$$

通过极大似然法来估计 $\mathbf{p}$，在实验结果上最大化对数似然函数

$$\ln(L(\mathbf{p})) = \sum x_i \ln p_i \tag{4.7}$$

$$\hat{\mathbf{p}} = \arg\max_{P} L(\mathbf{p}) \tag{4.8}$$

记

$$g(\mathbf{p}) = \sum p_i = 1 \tag{4.9}$$

显然公式4.7为凸函数，在4.9中引入拉格朗日乘子代入得

$$\ln(L(\mathbf{p})) = \ln(L(\mathbf{p}) + \lambda(1 - g(\mathbf{p}))) \tag{4.10}$$

当式4.10取得最大值时，自变量偏导数为 0。

$$\frac{\partial}{\partial p_i} \ln(L(\mathbf{p}) + \lambda(1 - g(\mathbf{p}))) = 0 \tag{4.11}$$

将式4.7、4.9代入4.11有

$$\frac{x_i}{p_i} - \lambda = 0 \tag{4.12}$$

由公式4.5、4.12可得

$$\hat{p_i} = \frac{x_i}{\sum x_i} = \frac{x_i}{n} \tag{4.13}$$

在得知其持有的 VM Flooding 虚拟机群的同驻情况后，攻击者可以将每台物理主机上的攻击实验用虚拟机数量缩减到每台物理主机上仅一台，这将大量减少之后所需

同驻攻击的成本和开销。而上述概率，更是可以告知攻击者如何进行下一步的攻击实验，并给予攻击者理论上的指导。例如，优先从同驻发生率较高的物理主机开始进行同驻检测，从而加快发现与目标同驻的速度。

4.4 虚拟机同驻方案行业应用

在国内主要的公有云市场中，阿里云占据最大份额、腾讯云、青云等位居其后。其中，α 云[1]作为行业标杆，具有很强的标准性和普适性[2]。因此本章选取 α 云作为目标，对其商业策略和服务等级协议进行研究，并在其上对前文提出的 VM flooding 攻击方案进行了一系列实验。

通过有效的 α 云帐号，用户可基于类 Xen 的内核创建一个或多个虚拟机镜像。运行中的镜像即称为实例，一旦创建了某实例，它将被分配到 α 云网络中的某一台物理机，且整个生命周期都将在这台物理机上。在按量付费的情况下，每个帐号可同时运行最多 50 个实例。本章选择在 α 云华南 1-可用区 A 进行投放实验，后续如无特殊说明，虚拟机实例都是在该可用区创建。

4.4.1 云平台网络布局分析

如前文所述，在 α 云云平台中，同一个地域的多个可用区之间是内网互通的，即同一个地域的所有虚拟机属于同一内网。为了探测云平台的基本网络布局，本文猜想 α 云内网 IP 地址空间可能在不同可用区之间进行了清晰地分区，以便于管理每个区域的单独网络连接，而且在同一可用区内的不同实例类型与内网 IP 地址也可能存在规律性。

首先，为了检验内网 IP 地址是否在不同可用区之间进行了分区，使用甲、乙、丙三个帐号，分别在华东 2 地域的可用区 A、可用区 B、可用区 C 各创建 20 台不同类型的虚拟机，其内网 IP 地址空间分布如图4.2所示。

对可用区与虚拟机 IP 映射关系进行分析，我们得出以下结论：在 α 云中，同一地域的虚拟机处于同一个 B 类网段；尽管有些虚拟机处于同一可用区，但是它们有可能并不处于同一个 C 类网段。

4.4.2 自动化同驻检测脚本设计

为了能够方便地对商用云平台上的远端虚拟机实例实施同驻检测，则隐蔽信道通信程序需要部署在每台实例上。为了方便起见，本章首先创建了一个通用的含有隐蔽信道发送和接收程序的实例镜像，并将它保存至 α 云提供的模板库中，以便之后批量

[1]根据相关规定，本章隐去实验所用云服务的具体名字，且下文中均以 α 云代替。

[2]α 云的运营模式与国际市场上被广泛运用的 Amazon EC2 十分相似，且国内其他商用云平台皆尽效仿。在其上进行实验与在其他商用云平台上进行实验的唯一不同是开启虚拟机的接口有细微差别。

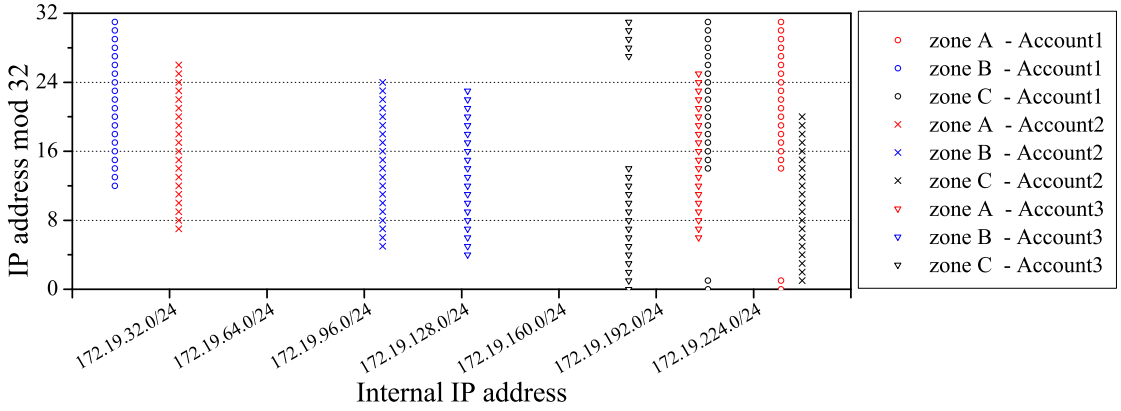


图 4.2 可用区-IP 规律

新建虚拟机实例。模板的用户操作系统为 64 bit 的 Ubuntu 14.04 LTS。

然而，当创建 n 台虚拟机时，需要对 C_n^2 对虚拟机进行同驻检测，若手动操作则会异常繁琐，因此，本章设计并实现了自动化同驻检测工具 ACD(Automatic Co-resident Detector)，该工具简化了同驻检测的操作流程，为本章的后续实验提供了十分有效的帮助。通过执行脚本程序，ACD 能够远程控制 α 云中的虚拟机实例，检测其两两同驻的情况，其工作流程如图4.3所示。

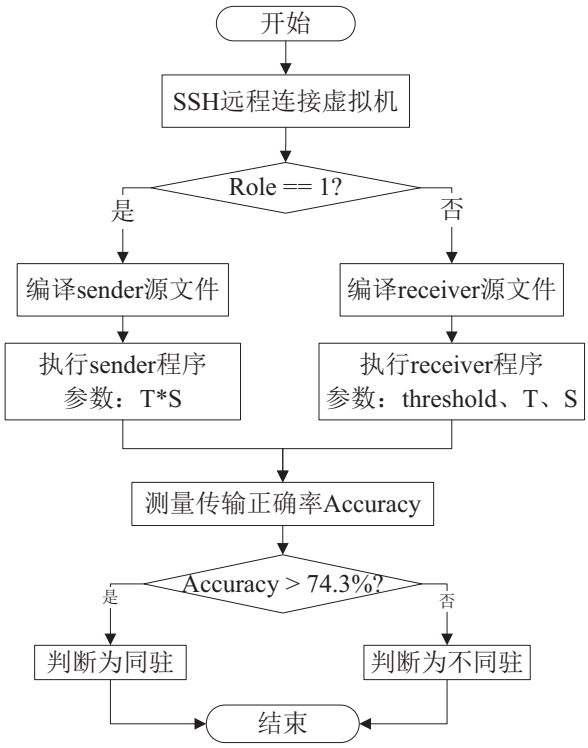


图 4.3 ACD 工作流程

4.4.2.1 基于 Expect 的自动化脚本实现

为了实现控制 n 个机器如上所述自动登录，传统自动化方案一般采用配置 n 对密钥 (用户私钥和公钥)，继而使用 SSH 自动登录协议进行远程控制。简单描述如下。

SSH-keygen 命令可以更改匹配 RSA 公钥私钥对的名字, 比如:

SSH-keygen -t rsa

然后将.pub 文件的内容追加到 sever 的 keys 文件中，在文件.bashrc 中加入脚本行，即可实现 SSH 和 SCP 自动登录。

而若借助 Expect 进行远程操控，则要方便许多。通过利用 Expect 来处理交互的命令，可以使同驻检测任务自动化完成。其思路与传统 SSH 方法基本相同，核心代码如表4.3：

4.4.2.2 基于 Ansible 的自动化脚本框架

Ansible 是一个配置管理与自动化运行工具，并且非常容易上手。Ansible 同样依赖服务器上运行 SSH 服务，通过 SSH 连接服务器并运行配置任务。它非常适合将 bash 脚本转换成 Ansible 任务，并且由于其基于 SSH，所以很容易检查运行结果。Ansible 适合重复以及复杂的配置脚本，并在会基于 Facts(Ansible 自定义的系统和环境信息) 运行。

Ansible 默认使用 Python 作为脚本语言，为了增强本方案的普适性，本章通过设置 inventory 变量 'ansible_python_interpreter'，统一修改脚本语言的版本，使得 Ansible 替换掉默认的 Python 解释器而使用符合本章要求的编译环境。

Ansible 支持在阿里云或 Amazon EC2 中采用组的形式管理虚拟机通过遍历组内的所有主机，可以在模板中访问 "groups" 字典，如果需要访问有关这些主机的必要参数信息 facts ，例如每个主机的 IP 地址及 DNS，并且需要确保它们已经被云端知晓且可被访问。Ansible 包含许多用于控制模块，我们将 Ansible 模块集成并在 AWS 上下文中使用 Ansible。我们以亚马逊 AWS 云为例，介绍 Ansible 在本章中的使用。

另外，AWS 相关模块的身份验证通过将访问和密钥指定为 ENV 变量来处理，并用 ansible-vault 加密，存储在文件 vars_file 中：

如果将凭证 credential 存储在 vars_file 中，则需要在每个 AWS 模块中引用它们。例如：

之后便可以批量创建实例：

由此，将使用 add_host 模块动态创建一个由这些新实例组成的 group，便于在后续任务中立即在主机上执行配置 IP 等操作。

由于本章并不需要除开启虚拟机实例之外的任何操作，因此之后的配置在此不赘述。

表 4.3. Expect 自动化脚本

```
#!/bin/bash
echo “请按提示依次输入发送方虚拟机的必要信息 (2 项)。”
read -p “请输入发送方虚拟机公网 IP 地址 (1/2)：” ip
read -p “请输入发送方虚拟机的 root 密码 (2/2)：” password
echo “请按提示依次设置发送方参数 (1 项)。”
read -p “请输入发送方参数 T*S(1/1)：” threshold
expect ≪ EOF
set timeout 30
spawn SSH -l root $ip
expect “(yes/no)”
send “yes”’
expect “password:”
send “$password”’
expect “#”
send “cd /root/whisper-amy-aliyun/algorithm4”’
set timeout 30
expect “#”
send “gcc sender64bits.c -o sender64bits”’
set timeout 30
expect “#”
send “./sender64bits $ts”’
sleep 0.5s
interact
EOF
echo “请保持此发送程序，并进入接收方终端等待检测结果。”
```

表 4.4. Ansible 密钥存储

```
ec2_access_key: “–REMOVED–”
ec2_secret_key: “–REMOVED–”
```

4.4.3 同驻实验设计及评估

4.4.3.1 单一帐号同驻实验

由于考虑到 CPU 核数和内存大的虚拟机，可能被分配到更具特殊性的物理机，首先用帐号甲同时创建了 n1.small 和 n2.medium 虚拟机实例各 50 台，采用前文提出同

表 4.5. Ansible 密钥配置文件引用

```
- ec2
  aws_access_key: “ec2_access_key”
  aws_secret_key: “ec2_secret_key”
  image: “...”
```

表 4.6. Ansible 定义组

```
- hosts: localhost
  connection: local
  gather_facts: False
  tasks:
    - name: Provision a set of instances
      ec2:
        key_name: my_key
        group: test
        instance_type: t2.micro
        image: “{{ ami_id }}”
        wait: true
        exact_count: 5
        count_tag:
          Name: Demo
        instance_tags:
          Name: Demo
      register: ec2
```

表 4.7. Ansible 配置 IP

```
- name: Add all instance public IPs to host group
  add_host: hostname={{ item.public_ip }} groups=ec2hosts
  with_items: “{{ ec2.instances }}”
```

驻检测方法，对其进行两两检测，检测结果显示任意两台虚拟机都不同驻。再考虑到独享型的虚拟机，可能被分配到特殊的物理机上，本章继续用帐号甲同时创建了 n2.small 和 n2.medium 虚拟机实例各多台，对其进行两两检测，检测结果显示任意两台虚拟机都不同驻。如表 4.8所示。

表 4.8. 一个帐号同时创建多台虚拟机

帐号	地域-可用区	实例类型	数量	同驻对数
甲	华南 1-A	n1.small	50	0
甲	华南 1-A	n2.small	50	0
甲	华南 1-A	n1.medium	50	0
甲	华南 1-A	n2.medium	50	0

表 4.9. 两个帐号同时创建多台虚拟机

帐号	地域-可用区	实例类型	数量	同驻对数
甲、乙	华南 1-A	sn1.medium	10	1

4.4.3.2 多帐号同驻实验

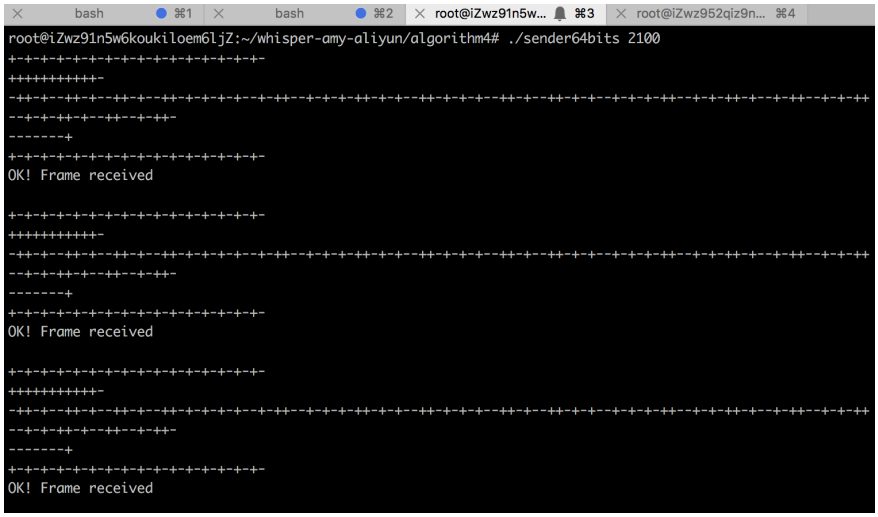
基于上述推测，改用两个帐号分别同时创建了 5 台 sn1.medium 虚拟机实例，对其进行同驻检测，检测到一对虚拟机同驻，如表 4.9所示。由此发现，只要采用不同帐号，就有实现虚拟机同驻的可能。

在同驻的该对虚拟机上，其基于内存总线隐蔽信道的传输效果如图 4.4所示。其中，发送信息被封装到一个 Frame 中，‘+’、‘-’ 分别代表所发送的 ‘1’ 和 ‘0’。

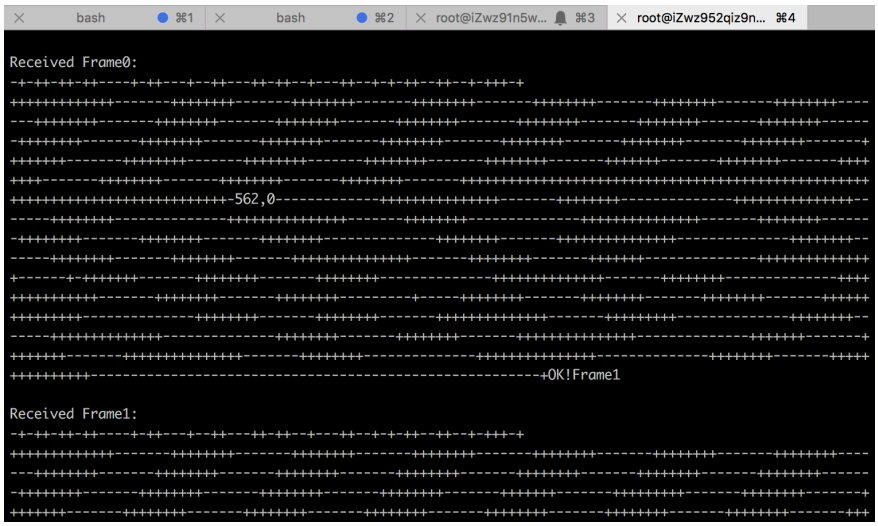
当使用三个帐号甲、乙、丙同时创建 sn1.medium 虚拟机实例各 5 台时，发现了两对同驻。于是为了进一步探究帐号数量与同驻之间的关系，同时具体测量同驻产生率，本章分别使用多个帐号同时创建虚拟机，进行了 50 次重复试验。当使用四个帐号甲、乙、丙、丁同时创建虚拟机实例时，共出现了 35 对同驻。其中，有 2 台物理机上同驻了 4 台虚拟机，3 台物理机上同驻了 3 台虚拟机，14 台物理机上同驻了 2 台虚拟机，另外 75 台物理机上只分配到一台虚拟机，总计 120 台虚拟机总计被部署到了 94 台物理机上。同驻结果如表4.10。

其中，两个不同帐号同时创建 medium 型实例各 5 台时，有 48% 的概率实现一对虚拟机同驻。三个不同帐号同时创建实例各 5 台时，有 90% 的概率出现至少一对虚拟机同驻，有 82% 的概率出现至少两对虚拟机同驻，有 52% 的概率出现三对虚拟机同驻。另外，通过查看所开启的实例的内外网网络状态，发现同驻的这虚拟机有很大比例分别处于两个不同的内网网段，所以判断 α 云有极大可能使用了类似 VxLan [104] 的网络隔离措施来构建安全组或私有云的网络边界。

为了探究虚拟机同驻概率与实例类型之间的关系，本章对 n1.tiny、sn1.medium、sn2.medium 三种类型的实例进行了实验。此次实验使用了 12 个帐号同时各创建 1 台虚拟机，重复实验 50 次，同驻结果如表4.11。由结果可知，sn1.medium 型实例的平均同驻对数稍高于其它两种类型。因此，在之后的大规模同驻实验中，均使用该型实



(a) 发送方



(b) 接收方

图 4.4 隐蔽信道在 α 云上的检测效果

例作为洪泛实验的样本主体。

4.4.3.3 大规模虚拟机洪泛实验

由上文可知：当帐号越多时，同驻比例越大；当实例类型为 sn1.medium 时，同驻比例最大。借助此规律，已经可以实现一定程度上的虚拟机同驻。不过为了进一步对本章所提的洪泛策略进行了改进与验证，我们花费了 30 天时间，在 α 云使用高峰时间段晚 8 时至晚 9 时，使用 12 个帐号同时开启共 360 台、480 台和 600 台 sn1.medium 型实例各 10 次。具体情况如图4.5所示。

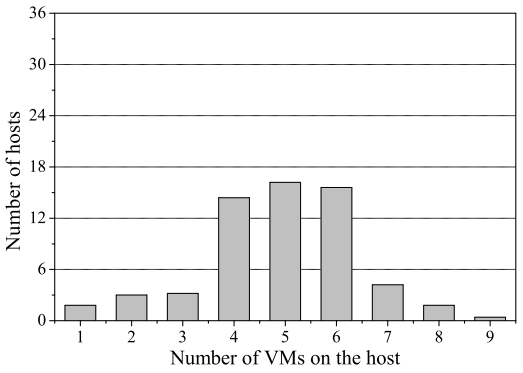
如图4.5(c)所示，110 台物理主机中，载有 5 个和 6 个同驻虚拟机的数量最多，分别有 32 和 31 台。所有的物理主机均有 3 个及以上同驻虚拟机，单台物理主机上最大

表 4.10. 多帐号同时创建多台虚拟机

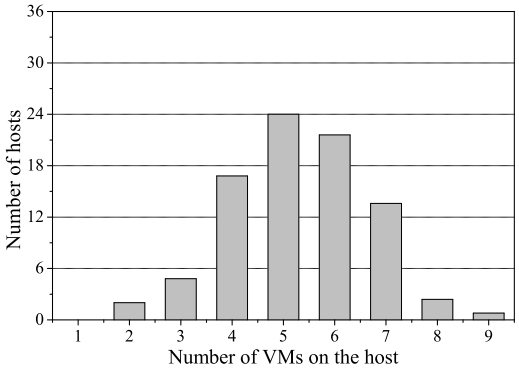
帐号	地域-可用区	实例类型	数量	同驻对数
甲、乙、丙	华南 1-A	sn1.medium	15	2
甲、乙、丙、丁	华南 1-A	sn2.medium	120	35
甲、乙、丙、丁、戊	华南 1-A	sn2.medium	120	53
甲、乙、丙、丁、戊、己	华南 1-A	sn2.medium	120	59

表 4.11. 多帐号同时创建不同类型虚拟机实例

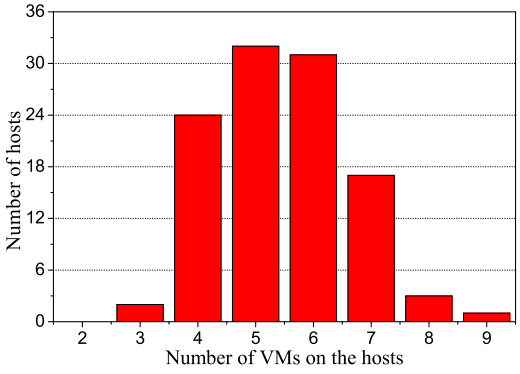
实例类型	数量	平均同驻对数
n1.tiny	12	2.46
sn1.medium	12	2.88
sn2.medium	12	2.62



(a) 360 台虚拟机实例投放分布



(b) 480 台虚拟机实例投放分布



(c) 600 台虚拟机实例投放分布

图 4.5 大规模虚拟机洪泛实验

的同驻虚拟机数为 9。则由4.3节结论可知：在同一时间再开启 10 - 15 台虚拟机，有约 15% - 21% 几率使得攻击虚拟机与这 9 台目标虚拟机同驻。另外，在所有的 12 个

帐号中，同一帐号的 50 台实例被投放到了 40 - 43 台物理主机上，其中载有两台虚拟机的物理主机均不少于 6 台。若此时目标虚拟机处于第 i 台物理主机上，且其上已有 N_i 台虚拟机同驻，则由4.3节的结论可知：在同一时间再开 $\ln(1-\delta)/\ln(1-\frac{N_i}{600})$ 台虚拟机，则有 δ 的几率使得攻击虚拟机与目标虚拟机同驻。

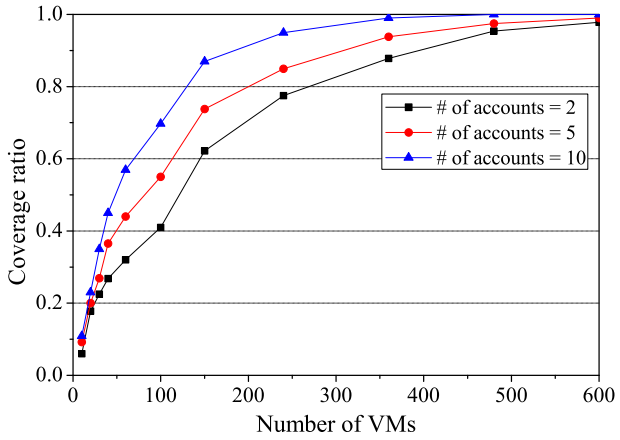


图 4.6 同驻覆盖比

图4.6显示了采用上述基于后验概率洪泛策略进行有针对性的虚拟机投放时，仅使用 2、5、10 个帐号迭代开启虚拟机，同驻的覆盖率。可以看到，当投放数量超过 100 时，同驻覆盖率有大幅提升；当使用 5 个不同帐号投放 72 次 (360 台虚拟机实例) 时，同驻覆盖率就已经接近 90%；当使用 10 个不同帐号投放 25 次时，覆盖率超过 95%；当投放实例数超过 500 时，无论使用帐号的数量多少，都可实现基本全部覆盖。

通过历时近多个月的同驻实验，本文发现和验证如下规律：(1) 现阶段 α 云对于同驻威胁作出了基本的防护，采取了一些简单的应对措施；(2) 不同帐号同时创建相同可用区相同实例类型的虚拟机时，产生同驻的可能性极大；(3) 采用的帐号越多，攻击成功率越高；(4) 实例类型对于同驻率的影响不大，而若想要尽可能地提高同驻概率，则建议使用 sn1.medium 型 (中型) 虚拟机实例；(5) 当启动大量的虚拟机来进行 VM Flooding 实验时，同一帐号的虚拟机无法被分配到同一物理机的规律被打破；(6) 在同一时间虚拟机被分布在百余台物理主机上，基本覆盖了该可用区子网的物理主机，充分说明了该洪泛策略的可行性。

4.5 同驻虚拟机精确定位

由上文所述，当洪泛虚拟机数量达到一定程度时，可实现云平台可用区物理主机的全部覆盖。然而尽管如此，攻击虚拟机如何定位目标虚拟机依旧困难。因此，本章提出一种基于内存硬件特性的同驻虚拟机精确定位方法。

4.5.1 基于虚拟化平台内存漏洞的 DoS 攻击

在云计算环境下，攻击虚拟机可借助前文所述内存总线冲突来实现云环境的拒绝服务 (Denial of Service, DoS) 攻击，其具体操作方式即不断触发内存总线冲突，造成大量内存访问时延，从而达到无法平台上其他虚拟机无法正常使用内存的目的，正如利用其构建隐蔽信道一般。

除了该内存总线占用漏洞利用，利用内存行锤击漏洞也可造成虚拟化平台的服务等级下降甚至停止。攻击虚拟机可通过不断锤击具有关键数据的内存行 (如 Hypervisor 所维护的 EPT/NPT 页表)，使得整个虚拟化平台暂停服务 [14,36]。

4.5.2 基于合谋的同驻虚拟机远程定位方法

基于上述 DDoS 攻击，结合合谋攻击的思路，本节提出一种能够快速定位目标虚拟机的方法。

假设攻击者已采用4.3节所述方案，成功将攻击虚拟机分布当各物理平台上，为了节省开销，此时攻击者可将各物理主机上的洪泛虚拟机减少至 1 台。为了叙述方便，本文将各台物理主机上运行的被攻击者持有的虚拟机称为 probe 虚拟机，并设其集群为 $P = \{probe|p_1, p_2, ..., p_m\}$；将攻击者所拥有的远程设备 (可访问云中某服务) 称为 remote 端。

当攻击者已通过社会工程学手段得知目标虚拟机位于某可用区且部署服务时，可使用 remote 端访问该服务。与此同时，在 probe 虚拟机中执行基于内存漏洞的 DoS 攻击，并观察 remote 端上目标服务的响应情况，若发现当 P 中某台虚拟机 p_i 在执行 DoS 攻击时，服务中断或时延极大，则可判定 p_i 已与目标虚拟机同驻。

4.6 本章小结

本章提出一种普适的虚拟机同驻方法。以 “先尝试，再验证” 的方式来完成攻击虚拟机与目标虚拟机的同驻，所用思路简单易行，且其在4.1节所假设的强安全条件下依旧有效。该方案能够使攻击者实现快速的恶意虚拟机与目标虚拟机的同驻，且尽可能地减少不必要的开销。所提出的同驻检测方法误检率不超过 0.5%，同时鲁棒性强；提出了一种虚拟机实例洪泛策略，结合该策略与同驻检测方法，获悉虚拟机洪泛过程中任意两台虚拟机的分布情况，进而能够得到攻击者实现与目标虚拟机同驻的精确概率；最后本文结合实际，在 α 云实施了本同驻方案，并取得成功，对同类方案 [9] 8.4% 的成功率有大幅提升，同时也优于方案 [105] 20% - 100% 的成功率。整个过程并不需要破坏 α 云本身的防护机制，且通过自动化脚本实施，开销小，灵活性高。更恶劣的是，当同驻覆盖率高达一定程度时，此攻击甚至可演变为大规模分布式拒绝服务攻击 (Distributed Denial of Service, DDoS)。

本章所做工作本质上属于对云平台的攻击，是一切基于同驻的攻击的必要前提。虽以 α 云作为实验目标，但其结果仍需国内其他各大商用云平台予以关注。之后将会针对此威胁，面向云平台中的隐蔽信道与侧信道，研究其检测与防御方案，研究尽可能通用的方法来解决同驻所带来的信息泄露问题。

5 基于共享资源矩阵与事件关联的侧信道检测方法

在云计算环境中，隐蔽信道分析是一个重要难题，其原因在于：其一，它以强制访问控制研究为基础，而系统越来越大的代码量分析起来极为复杂，而且缺乏有效的自动化检测工具[49]。其二，传统的隐蔽信道检测的方法并不适用于云计算环境，需要对顶层设计或者源代码进行分析，无法保障实时性；其三，基于统计学的检测方法不好实现，因为对隐蔽信道的利用面临许多干扰，如极短的时间间隔和重叠的噪声。Wu等人提出 C2Detection 检测方案[50]，它在 Hypervisor 层捕获信息流，并利用马尔科夫与贝叶斯模型的双阶段算法来检测隐信道。C2Detection 能检测的隐蔽信道范围较广，且其实现方案具有一定的可扩展性 (其检测算法是以 LKM 的方式加载到 VMM 内核的)，但是该方法需要运行在 Dom0 的内核态，实施起来并不方便。Chen 等人[51] 通过使用虚拟机回放的方式来检测侧信道，可以以精确的时序来重现程序的执行。然而这种方式尽管较为新颖，效果不错，但是带来的开销很大。

本章在现有研究基础上，提出了一种自动化且具有普适性的隐蔽信道 (侧信道) 检测方法。为了实现全局检测，利用 VMM 中的事件记录机制来收集必要的元数据。将共享资源矩阵与事件关联相结合，提出了一种从行为方面准确定位和分析恶意隐蔽信道的方法。

5.1 侧信道事件关联分析

在单处理器单进程的场景下，一般我们可以通过限制不可信进程对高精度时钟的访问来控制时间信道。然而在云计算环境下，内部攻击者成为外界的攻击者，数据信息泄漏变得更加难以追踪，同时基于恶意代码检测的缓解方案也更难实现，这就要求我们另辟蹊径来识别这些恶意信道。

5.1.1 时钟与事件

广义上来说，用于测量时间进度的任何方法均可以被称之为时钟。在跨虚拟机的时间信道模型中，为保证数据传输，需要一对时钟。一个用来测量事件，一个用来发出执行触发信号 (来标记执行事件的发生)。本章认为时间的流逝是以观察者可以彼此区分的事件序列为特征的。

计算机系统中存在四种可用的时钟源[80]：以 TSC 为代表的 CPU 内部时钟，实时时钟，I/O 子系统 (完成中断，DMA 数据到达速率等) 和内存子系统 (数据/指令提

取，互锁等)。其中，位于 CPU 内的指令周期时钟 TSC，位于主板的实时时钟如 RTC、PIT 和 HPET 等都是现代计算机系统所提供的较为精准的时钟源。

类似 I/O 子系统、内存子系统等能够提供任务调度的系统所能提供的一般是特定时间间隔发生的事件。这些事件虽然不能提供直接的时钟，但是这些时钟大多数都是可以被调制的，例如，内存子系统基于执行内存的提取时间来提供时钟。内存子系统中的事件同样可以被用于构建时钟，这主要取决于目标数据是否在高速缓存以及存储器总线上的通信量以及存储器控制器上。这些时钟中的一些不是独立的，这是因为事件来源经常通过接受输入与外部世界进行沟通。特别是云中的大规模并行环境创建了许多隐含的高分辨率时钟。例如，考虑使用 DMA 数据到达来构建时钟：某进程将磁盘读取请求发送到缓冲区中，然后轮询缓冲区的第一个字节。当字节中的数据发生改变时，该进程就知道 DMA 传输已经开始；然后，轮询另一个位置，当数据发生变化时，进程知道传输已到达缓冲区中的另一个点；循环往复，若每次发生这种访问的时间间隔基本相同，便能够以此构建一个粒度较粗的时钟源。

所以总的来说，事件流可以被认为是时钟，这些事件可能出现在任何地方和任何时间，比如在 DMA 信道之中，或 I/O 操作访问存储器期间。当这些事件产生时，就可以被用作时钟。

5.1.2 事件与隐蔽信道

隐藏信道与对等通信一样包括三个基本要素：发送者、接收者、通信信道。然而，隐蔽信道还具有一些额外的特殊属性。首先，发送者和接收者在通信时需要进行某些秘密的信道调制操作。其次，接收者可以感知到发送者信号的发布。第三，通信双方使用变量或多个变量来传输信息。

观察者通过设定变量来区分事件，如果事件由布尔变量组成，则只有两个本质上不同的事件。然而，如果观察者增加了代表变量已改变次数的计数器的值，则可能存在无限数量不同事件。足够的内部存储空间允许观察者区分任何数量的其他无法区分的事件。

从时钟，事件和隐蔽信道之间的两两关系来看，事件显然具备作为传输秘密信息时钟的能力。那么我们可以设想，事件关联分析将是隐蔽信道检测本质上的有效解决方案。故而，本方案利用事件关联分析来发现虚拟机间的隐蔽信道。

5.1.3 观察者的视角

正因为事件可以在信息系统中被感知和调制，我们可以通过使用一些特定事件来构建隐蔽信道。而要确定一个事件是否可以用作隐蔽信道利用中的信息载体，首先需要确定观察的视角。

本章将可以控制内部定时的发送器的 (如信号调制器) 视为内部，将接收机 (如信号解调器) 视为外部 [78]。观察视角的选择十分重要，不同视角下相应隐蔽信道的利用方法及其带宽和噪声也会有很大的不同 [79]。

在隐蔽信道检测方面，应该采取全局性的观点，最大限度地识别行为层面的未知信道。因此本章通过在 VM 事件中呈现的机器环境和操作，在 Hypervisor 层上部署的全局观测方案。

5.1.4 隐蔽信道的分类

在云环境中，文献 [50] 提到了三类隐蔽信道，分别是域内、域间和跨平台的隐蔽信道。对于虚拟域内的隐蔽信道，目前已有充分研究 [106,107] 对其进行检测。而对于跨平台的隐蔽信道，进程 P_i 和 P_j 只能通过网络连接通信，目前已有基于 Markov model 等方法 [108] 来检测。且这种隐蔽信道广泛存在于网络环境中，不属于云计算环境下的特殊信道，因此本章并不对此做深入研究。

本方案主要针对域内和域间的隐信道，由于虚拟计算环境设计架构缺陷导致的恶意隐蔽信道来开展研究。这样的隐蔽信道文献 [50] 将其归类为第三类。此类隐蔽信道才是云计算环境中需要特别重视的，需要用特殊的方法来检测的隐蔽信道。在此本章将其进一步分类：

(1) 基于物理层的攻击的，利用虚拟化体系架构缺陷，硬件层的隐信道；它们利用 CPU、Cache [8,109] 和 Memory 缺页或总线占用引起的抢占或中断信号来传递信息，如 Whisper [33] 等等。同时还有可能通过冷启动攻击及其他侧信道攻击建立隐蔽信道。

(2) 基于系统调用和 VMM 共享机制的攻击的，系统级、进程级的隐信道；为了完成虚拟机域间的通信与协作，虚拟化平台一般都共享资源的方法，最常见的即 Hypervisor 所提供的事件信道、VM 间共享内存 [110] 和 Foreign Map 等机制 [111]。如果发送方 domain 控制共享内存的填充时间，接收方 domain 可以观察获取数据的时间，则可以根据时间不确定性的特征构建时间信道。

(3) 基于云平台的虚拟机防护间隙 (Instant-on Gaps) 和 VM 迁移间隙等攻击，Guest-VM 间的隐蔽信道；云计算的按需和弹性服务，让云租户可以根据应用负载情况，自动化地申请、销毁、迁移虚拟机实例。这使得虚拟机创建和启动的过程不同于物理机器和操作系统进程，传统安全防护软件无法对这一关键步骤进行防护，造成虚拟机安全防护的间隙 (Instant-on Gaps)。攻击者若控制了云计算平台，则可以利用频繁发生的虚拟机创建防护间隙，创建隐蔽信道。攻击者还可能利用云平台对攻击虚拟机的迁移操作，挤占不同虚拟机的资源，创建隐蔽信道。

由于在云计算环境中隐蔽信道种类繁多、无处不在，根据上述分类，本章在 Lampson 最初提出的隐信道模型的基础上改进了隐蔽信道定义 (不考虑阈下信道和推理信

道)：

定义 1 隐蔽信道五元组 $\langle V,\ T,\ A,\ Ph,\ Pl\rangle$

$V = \{variable \mid$ Each variable represents one kind of resources in current cloud computing system.$\}$

$T = \{type \mid$ Each type represents one category of covert channels in cloud environment.$\}$

$A = \{attribute \mid$ Each attribute represents one special feature of a covert channel.$\}$

其中，变量 V 指共享资源 $variable$ 的集合；T 为隐蔽信道的分类的集合 (隐蔽信道的分类可由专家系统事先定义)，A 指隐信道能被识别的特殊属性 $attribute$ 的集合。T 与 A 可以专家系统或用户自定义，$attribute$ 表示隐蔽信道可能有的指标和属性。Ph 和 Pl 则表示利用该特定隐蔽信道通信的安全级不同 (或相同) 的双方。T 直接决定了用何日志和安全配置文件来检索识别隐信道，T 中的 $type$ 标识了该隐信道是时间隐蔽信道 (Timing channel) 或是存储隐蔽信道 (Storage channel)，是通过内存泄露的信道 (虚拟化系统结构缺陷，硬件层的隐蔽信道) 或是 VM 间的隐蔽信道 (系统级、进程级的隐信道) 亦或是平台间的隐蔽信道 (网络层的隐信道)。A 则影响了在安全配置文件中应该如何搜索到合适的条目。A 中的 $attribute$ 可以包含隐蔽信道的通信时间间隔的分布特征，$variable$ 变量的熵 (条件熵、熵率)，CPU、cache 的负载，VM Exit/VM Entry 的执行时间，任务响应时间等等。

当引入类型 T 和属性 A 时，我们可以通过使用预定义的类型和属性将隐藏频道划分成各种类别。这可以大大提高检测解决方案的效率和正确性。

5.2 基于共享资源矩阵的侧信道检测方法

根据上述思想，我们提出了一种新的隐蔽信道的检测方法，可归纳如下。

(1) 与传统的使用静态马尔科夫模型，或者贝叶斯分类器来分析访问间隔的时间序列的方法不同 [50]，本章采用了新的方法，从行为特征的角度出发，考虑隐蔽信道的基本特征来检测隐蔽信道的利用情况。

(2) 由于原始的 SRM 的方法对分析隐蔽信道是不可行的 [112]，我们改进了该方法，使它能够用于隐蔽信道的实时检测。此外我们还提出了基于 XSRM(扩展共享资源矩阵) 的事件关联分析方法。我们在分析现有共享资源矩阵数据结构的基础之上，提升了事件关联分析算法在实际应用中的适应性。第一点：矩阵数据结构转化为向量数据结构能够更加适合于关联挖掘算法；第二点：我们使用了分治的思想来优化算法，显著减小了系统开销。

(3) 本章方法利用虚拟机记录机制 [113] 和虚拟机自省机制 [114] 来收集虚拟机的行为信息。此时，虚拟机的所有基本事件都会被日志记录并作为之后的分析对象。另一

方面，本方法工作在 VMM 层，对所有的客户虚拟机透明。我们的方法可以使外部的观察者拥有一个全局的视角，这就可以更好地理解所有客户虚拟机的行为。并为观察大多数的跨虚拟机隐蔽信道带来了很好的效果。

5.2.1 整体框架

为了更好地解决上述问题，我们提出了云环境下隐蔽信道通用检测框架，如图5.1。首先，对信息进行收集，这里的信息包括：VMM 的事件日志、云平台的网络配置与安全策略等信息。然后，将收集到的文件按照定义好的事件的格式进行元数据抽取，进行清洗和标准化，之后形成结构化的事件元数据，使得这些隐蔽信道的特征能够凸显出来，以便处理。再对事件进行归并，从大量事件中将符合一定条件的事件归并为一条事件流。通过归并，我们可以提取出事件主体的行为，排除无关事件的干扰，提高系统的处理能力。为此，本章对所抽象出的事件进行关联性分析。针对每一次异常的事件，通过比对共享资源，考虑其数据、拓扑、资产等信息之间的关联，找出其中的规律 (特征)，建立起一套匹配规则。最后，根据这些事件序列来一一对应隐信道的特征，从而确定威胁点，达到检测的目的。

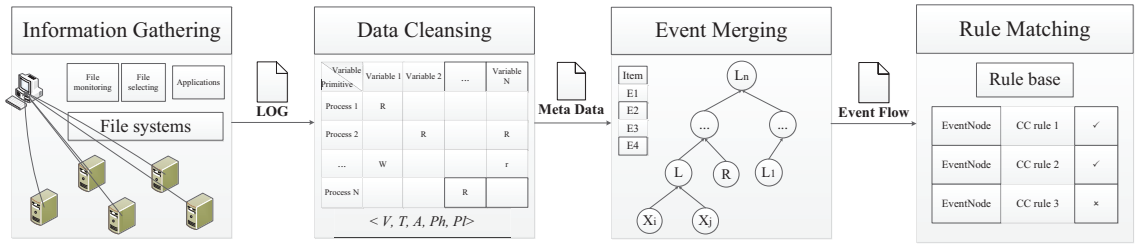


图 5.1 云环境下隐蔽信道通用检测框架

5.2.2 信息收集

本方案通过主动扫描的方式在云平台上收集各种信息，主要针对事件记录与日志及安全配置文件的收集。其中，需要的收集的目标包括：平台基本信息、任务响应时延、事件日志、网络配置、网络安全策略、网络连接及其流量等信息。

5.2.2.1 事件记录

为了更好地利用事件关联分析的方法来检测隐信道，对事件进行定义:

定义 2 事件 Event

$Event\ e = \langle eventId, time, vmId, processId, sharedVariableName, type, attributes, return \rangle$

其中，process 和 $sharedVariableName \in V$ 分别表示事件的主体和客体；字段 $type \in T$ 是用来描述该事件是何种类型的。$attributes$ 是复合类型的变量，对于硬件层事件，$attributes$ 包括 $\langle CPU\ load, memory\ access\ interval \rangle$；对于 VMM 共享内存

调用事件，attributes 可包括 $\langle HypercallId, grantTableNumber\rangle$；action 表示事件对变量的操作，取值可为读 “R”、写 “W”、其他 “Other” 等；“return” 表示事件的结果，在此分为执行成功和未执行成功两种状态。

虚拟机自省与事件记录机制在本方案中起到非常关键的作用。对于 Xen 或者 KVM 之类的虚拟化平台，有一些 VMM 的日志可以直接由守护进程获得，比如说 Xend 和 XenStore 的日志 (详见5.3.2节)。然而在某些系统中，确定性的和不确定的事件不会被记录。于是我们统一地使用 VMI 和 VM record 技术将这些信息记录下来，把它们按事件的形式存储并等待后续分析。

5.2.3 扩展的共享资源矩阵

经过上一步的信息收集和日志过滤，得到了精简过的数据，但是这些数据存在太过于零散，从中还并不能发现有用的信息。此时本方案借鉴了共享资源矩阵法 [47] 的思路，对这些数据开始进一步的处理时，同时建立共享资源矩阵，将可能存在威胁的信道提取出来。

将共享资源矩阵的定义进行扩展，定义适用于本方案的共享资源矩阵。

定义 3 共享资源矩阵 XSRM

假设 S 为当前的信息系统，定义扩展共享资源矩阵 $XSRM = (a_{ij})_{m\times n}$。XSRM 也是具有约束的概念格。在云环境中，很多数据集是包含数字属性和分类属性的混合数据集。因此 XSRM 是一种混合数据表 [115]。其形式化说明如下。

$XSRM = (E,\ A,\ V,\ f)$，是满足如下条件的四元组，

其中 A 与 E 已经在 **定义 1** 和 **定义 2** 中进行了定义。具体来说，A 是一个由 A^r 和 A^c 属性组成的集合 $A = A^r \cup A^c$。A^r 是一个数字属性的集合，它包括 CPU 负载，cache/内存的访问间隔时间等等。A^c 是一个种类属性的集合。包括共享变量名，SysCall/Hypercall 的参数等等。V 是属性值的并集，即 $V = \bigcup_{a\in A} V_a$ ，此处 V_a 是变量 a 的值域。$f : E \times A \to V$ 是相应的信息映射函数，即对于任意 $a \in A$ 和 $e \in E$，都有 $f(e, a) \in V_a$ 。

在构建共享资源矩阵时，将每一行对应的属性填充至表 5.1中，记行 $\{a_{i1}, a_{i2}, ..., a_{in}\}$ 为 $A(i,:)$ ，列 $\{a_{1j}, a_{2j}, ..., a_{mj}\}$ 为 $A(:, j)$ 。XSRM 中的每一行表示在当前信息系统中的一个事件。每一列表示在当前时间戳下某一个属性当前的状态。同时，在第一列中的 $sharedVariable$ 是一个概念格的扩展项，它表示共享变量，是一个特殊且相当重要的属性。

一般来说，共享资源矩阵只能被用作检测存储侧信道 (如文献 [116] 所提信道)。然而经过本方案改进之后，它同样可以被用来检测时间信道。那是因为本方案使用了事件作为列，而在原始的 SRM 方法中使用 TCB 原语作为列。XSRM 中的每个列都会在

表 5.1. 共享资源矩阵

XSRM	*sharedVariable* a	attr b	attr c...
Event e1			
Event e2			
Event e3			

之后基于关联分析的算法中被使用 (详见 5.2.4.1)。本章由以下两个理由做出该修改。

(1) 基于时间的侧信道和异步事件直接相关。换句话来说，发送进程的活动 (异步事件) 和接收进程的活动之间必然存在某种时间上的关联。可能他们产生的时刻很接近，或者他们使用了相同的对象 (共享变量)。当然这种关联不止存在于攻击者的信号调制的过程中，也存在于信号解调的过程中。

(2) 在一段程序执行过程中被记录的事件，提供了程序代码执行路径的细节。而且在多数虚拟化的系统中，客户虚拟机的使用者不能直接感知外部中断，但是他们可以感知到在中断处理流程中的某些事件的发生。我们的目标是捕获我们能感知的内部事件并和实际发生的外部事件做比较，然后努力弥补主机和虚拟机之间的感知差别，然后我们就能判断隐蔽信道的存在性。

5.2.4 关联匹配

在前面步骤中取得的可疑事件仍然混乱无序，为了更好的分析从事件中提取的行为，需要把它们归并到一个单独的事件序列中去。当归并之后产生一个由事件组成的序列，因此使用特定规则进行事件流的匹配来发现隐蔽信道的踪迹。

5.2.4.1 基于关联分析的事件归并

由于隐蔽信道同样属于信道的一种，所以它也具有一般信道的特征。于是在隐蔽信道形成时，攻击者的行为与结果事件的产生存在因果关系。正如 **??**中提及的，从事件和隐蔽信道之间的关系我们可以明白，在隐蔽信道的利用中，事件流可以用作相对时钟。而这一特点使得隐蔽信道在发生作用时，首先会传递大量关于信道本身的信息。举例来讲，利用隐蔽信道的攻击者必须执行同步和协商的步骤，才能进行信息交流的步骤，这个一定会留下蛛丝马迹。

此外，时间序列可以表达出隐蔽信道观察者 (或者接收者) 的时钟。由于这个原因，本方案努力去结合踪迹的特性，这能够帮助我们推断攻击者的行为。这些事件序列模式同样对于构建匹配规则时有很大的帮助 [117]。为了获得这些模式，同时基于 **定义 2** ，本章定义联合事件和事件流如下。

定义 4 联合事件 Associated Events

$\forall Event\ e1, e2$

If $e1.type == e2.type$ &&

$e1.attributes[i] == e2.attributes[i]$ &&

$|e1.time - e2.time| \leqslant TW$

then $e1$ and $e2$ are considered correlated, denoted as $e1 \Rightarrow e2$.

定义 5 事件流 Event Flow

$EventFlow\ eventflow = \langle eventFlowName, eventFlowId, eventList, TW \rangle$

其中，$eventList$ 是链接的事件节点的链表，表示出了事件流中对应事件之间的时序关系。$eventFlowType$ 指定了事件流中所有事件公共的类型。TW (Time Window) 指示了整个流中从第一个事件到最后一个事件的最大时间跨度。

至此，想要弄清楚事件之间的关系，关联性分析算法是最好的一种选择。然而，传统的数据挖掘算法在这个情况下无法使用，所以为了适合于云计算环境的这个应用场景，我们改进了这个算法。于是本章提出了利用针对性的 XSRM，基于向量数据格式的，改进频繁项目集的关联分析算法。存储事件的数据库实际上是一个概念格矩阵，它可以和共享资源矩阵进行并集操作。这样一个矩阵数据结构能够被很好的转换成向量数据结构，项链数据结构可以使用频繁项目集关联算法进行有效地处理。在本章中，采用了一个联合的方案来解决在现实生活中广泛存在的大数据挖掘的问题，具体如下。

(1) 共享资源矩阵减少了搜索项目最小值，并归并到最小值的开销，极大地提高了算法的有效性。

使用向量 “与” 算子搜寻了频繁项目集，然后在归并事件之后生成了事件流。每当算法执行交集操作 $Tidsets(R) = Tidsets(X_i) \cap Tidsets(X_j)$ 时，由于 XSRM 的特殊数据结构 (概念格矩阵)，并不需要遍历 $Tidsets(X_i)$ and $Tidsets(X_j)$ 中的所有对象。我们仅仅比较这些在同一个行中的对象 $Tidsets(X_i)$ 和 $Tidsets(X_j)$，如果他们相等就把他们放入结果交集中。

(2) 本方案利用了矩阵分割的思想，同时为了优化关联分析算法，添加了前置约束。

当程序运行时，XSRM 被持续的生成。为了使得在确定时间内 XSRM 连续生成的部分将被充分用作隐蔽信道的检测，本方案使用了对频繁项目集挖掘使用分治，剪枝和重组技术。实际操作如下：在一定时间范围内 (如 10 秒) 分析了小范围的 XSRM，而不是用长时间去分析一整个 XSRM。然而在这一段时间内，XRSM 生成的部分频繁项目集对全局性的分析是没有必要的。换句话来说，它们包含了一些干扰的数据。因此，我们借鉴古典先验算法，添加了剪枝的操作来去除无用的候选频繁项目集。在最小阈值支持的范围内挖掘一部分小的 XSRM 后，就能根据之前的 XRSM 得出的结果，去删选下一个 XSRM 中的频繁项目集，而需要删除的对象则是之前的块中没有起作用的部分。最终，我们通过归并所有频繁项集，从而获取全局结果。

表 5.2. 事件合并算法

输入: XSRM truncated by time window, s_{min}
输出: Frequent itemset L consisting of all event flows
1 scan event database to get initial frequent itemset L_1
2 $i=1,\ j=2,\ n=1$
3 $EventMerge\,(L_n)$:
4 for $X_i \in L_1$ do
5 for $X_j \in L_1$ && $j > i$ do
6 generate new candidate itemsets $R = X_i \cup X_j$
7 $Tidsets\,(R) = Tidset\,(X_i) \cap Tidset\,(X_j)$, 此处 $Tidset\,(X_i)$ 与 $Tidset\,(X_j)$ 为矩阵的列 $A\,(:, i)$ and $A\,(:, j)$ in truncated XSRM
8 if $\lvert Tidset\,(R)\rvert \geqslant s_{min}$ then $i = i+1,\ j = j+1$, else break;
9 $L_{n+1} = L_n \cap R$
10 if $n \leqslant k$, then $EventMerge\,(L_{n+1})$

首先在预定的时间窗口之内对相同的事件进行扫描，将项目集的编号记为 q 并对它进行记数操作。然后计算 n 个频繁项目集的交集，并为生成 $n+1$ 个频繁项目集做修剪操作。迭代上述操作步骤，直到只剩最后一个全局的频繁项集。当确定最小阈值 c_{min} 时 (此时 $c_{min} = k/q$), 我们需要将最小值 c_{min} 设置得合适来减少误报的比率。同时，考虑到的详细匹配规则，最小支持阈值 s_{min} 可以适当地设大一些，以便减少误检率。整个过程如算法 5.2。

5.2.4.2 自动检测规则构建

在归并可疑事件后，使用多种分析方法来手动进行开发检测。然而手动分析将会停止服务，这在云计算模型中是不能允许的。于是我们的解决方案必须包括自动检测。为了达到这个目的，给出了下列定义。

定义 6 子规则 Subrule

每个子规则按照范式格式给出，它们代表了某些属性的约束条件。自顶向下的子规则类 BNF 范式定义如下。

subrule = ruleHeader("ruleOptions");

ruleHeader = action type Ph Pl;

action = ("pass","alert","activatedBy","revert");

*ruleOption = (keywords":"StringValue";") **;

keywords = (general, attribute, processing);

*general = (metadata, VMid, Pid, reference *)*;

attribute = (CPULoad, memoryAccess, shareMemoryRequest, others);

processing = (countFunc, frequencyCalcFunc, expectationCalcFunc, entropyCalcFunc, others);

定义 7 匹配规则 Matching Rule

$rule_i = \{\ subrule_{i,1} \cap subrule_{i,2} \cap ... \cap subrule_{i,n}\ \}$

定义 8 匹配规则集 Rules Set

$Rules = \{\ rule_1, rule_2, ..., rule_n\ \}$

5.2.4.3 事件流匹配

事件流匹配程序包含了一系列步骤，这些步骤包括计算每个事件中的属性，对结果进行规则匹配 (如图5.2)。在获取了简明的事件流数据后，我们使用定义良好的规则去匹配属性关键字。一个隐蔽信道可能对应多个规则。然后当规则被匹配和触发的时候，规则匹配模型生成一个对应的警报 (详见5.3.3)。

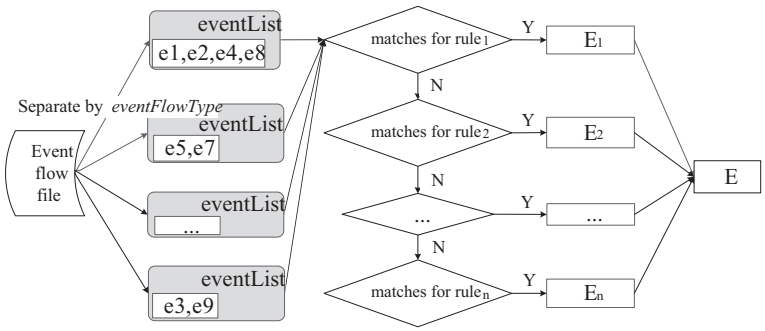


图 5.2 事件流匹配

当进入匹配阶段时，匹配断言由两个部分组成：条件和结论。检测的匹配算法如表5.3所示。

5.3 基于共享资源矩阵的侧信道检测系统实现

依据5.2节的设计，本章在 Xen 上实现了一个原型系统，使用了一个开源的云平台系统 OpenStack 来管理虚拟机。本方案的实现包含多个模块：日志记录模块、主机信息收集模块、文件过滤模块、事件关联模块和隐信道匹配检测模块。模块的组织方式如图所示。日志记录与监控模块处于 VMM 层，其他模块均处于 Dom0 的用户空间。

上述这些模块都被制作成插件，部署在定制的 OpenStack 上，通过这种方法，可以有效地缩短软件的运行时间，管理和部署也变得更加简单。用户可以依据自己的需求使用不同的驱动来管理插件的开关状态。这些依据可插入式 hook 实现的插件给予

表 5.3. 事件流匹配算法

输入: event flow file, Rules

输出: Threat Event Set E

scan the event flow file and separate all event flows into different sets by $eventFlowType$

calculate the total number n of all the event flow sets

$i = 1$

for $\forall rule_i \in Rules$, do

  for $subrule$ in a rule, do

    for $eventList$ in an event flow set, do

      for $\forall e \in eventList$, do

        $\forall attribute$ in e,

        if $LHS\,(attribute) == RHS$

        then add e into threat event set E_i

    $i = i + 1$

  if $i > n$, then $E = E_1 \cap E_2 \cap ... \cap E_i$, break;

了云品台开发者扩展系统的新方式。开发人员可以提前定义系统扩展点和新的后端逻辑，并且，这种扩展系统的方式是完全间接的，不需要修改源码和原始系统。除此之外，这些插件可以不依赖于云平台系统进行开发和发布，以下是一些关键模块：

5.3.1 日志记录与监控模块

为了兼容各种虚拟化需求，日志记录和监控模块的实现是兼顾了 Xen 的半虚拟化 (para-virtualization) 和硬件虚拟化 (HVM) 架构。默认情况下，商业云中的客户虚拟机 (guest VM) 无论是否为 HVM 架构，始终配备了 PV 驱动程序，所以本章不去区分 VM 是如何实现其事件记录模块的。该模块主要用于记录系统中的确定与不确定性事件。

日志记录模块会创建一个虚拟设备，该虚拟设备是一个 Linux 系统中的字符设备，具有通用的 Linux 系统调用接口如 read、write 和 ioctl 等。它实际上的作用是为 Xen 与用户空间数据、命令传递提供中介与缓冲服务。此外，它还要负责注册用户守护进程的虚拟中断处理回调函数，绑定相关的接口，读取 Xenstore 中 DomU 的信息 (包括事件信道号与共享内存环引用等)。日志记录模块还向用户空间导出标准化的 Xen 超级调用接口。这样一来，Dom0 就能够通过该接口灵活地开启和关闭日志记录功能。

如果 VM 没有使用 PV 驱动来实现事件的获取，则必须在 VM Exit / VM Entry

间隔期间拦截各种事件。包括 Hypercalls，I/O 请求，外部中断和各种异常的事件都可以从当前的虚拟机控制结构 (VMCS) 捕获。具体来说，Xen 提供了专门的截获和模拟的函数或功能，可以通过设置 VMCS 中的 bitmap 结构来与 VM 的行为进行交互。所以，我们利用和修改了这些函数来达到本章的目标。例如，为了拦截一个 Hypercall，我们只需截获专门的由 VMCALL 指令产生的 VM Exit，并重载用于 Xen 处理 DomU VMCALL 指令的专用函数。

对于如何拦截 Guest-VM 的中断与异常，在此简要说明。当 Guest OS 执行 INT 指令，则会访问 IDT 表，通过设置 IDT 中的 limit 值即可将超过调用号 limit 值的中断拦截。在处理中断和异常之前，Guest 首先会检查 limit 值，并判断向量值是否可以被读取。如果向量值越界，就会产生一个 #GP 异常，并产生 VM exit，从而被 Hypervisor 截获。因此，只需要设置一个相对较小的 limit 值 (如 32)，就可以在 VM Exit 后进行后续分析处理。

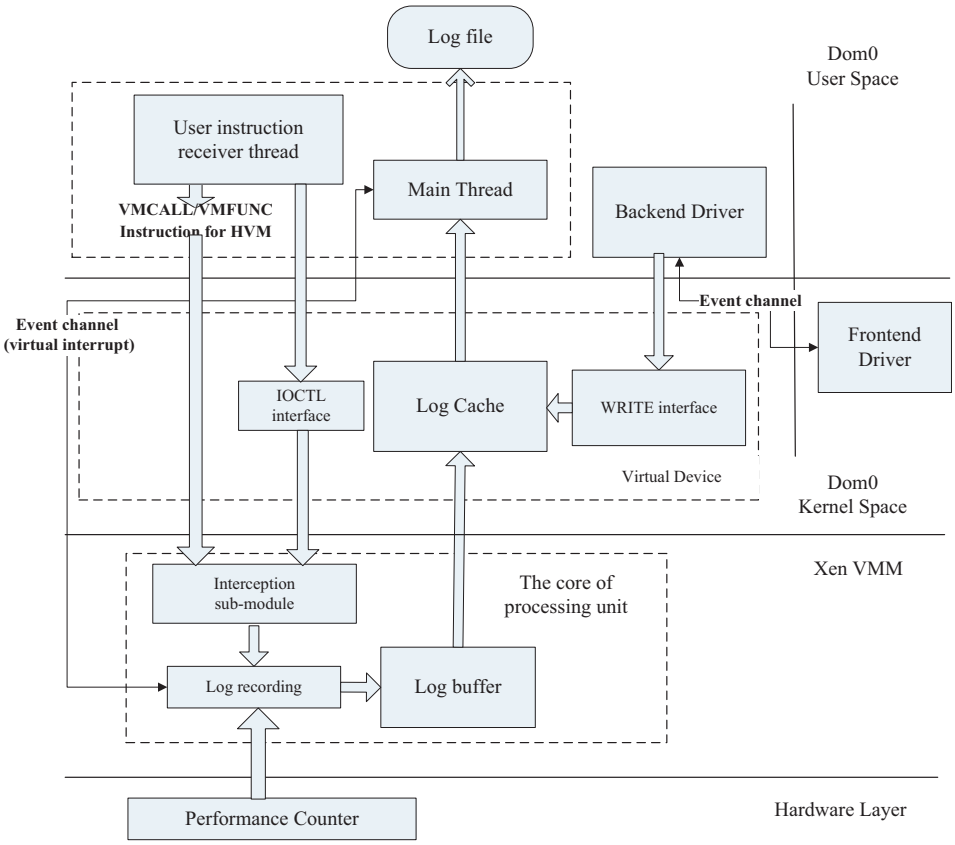


图 5.3 事件记录模块

5.3.2 主机信息收集模块

在信息收集模块中的所有操作都是依据于云平台系统的，而不是 guest VM。这使得此方案更具有全局性和透明性，也更适合于应用环境。

该阶段的主要目标是获取足够的日志文件和安全配置文件信息。为此，本章通过使用特定 API 如 VMWare 的 VI，XenAPI 或者 libVirt 来获取这些 virtual platform-specific data，然后将他们统一成 XML 格式的文件。与此同时，我们还通过爬虫脚本获取一些专用云平台 (如 OpenStack’ controller node, VMWare’ s vCenter 和 IBM pSeries’ HMC) 的日志文件和平台相关的安全配置。

需要监控的文件有:VMM 上各种守护进程的日志,Dom0 的事件日志 (参看5.3.1节)，物理主机的网络配置和网络安全策略。需要监控的条目有：对于主机而言，事件元数据包括用户对注册表的增删和修改，是否打开或关闭某服务，对于存储文件系统中文件的各种操作；对于网络来说，事件元数据属性包括网络端口的打开和阻断，对防火墙规则的修改，对网络处理进程的干预等各种操作。

5.3.3 隐蔽信道匹配检测模块

如前文所述，规则匹配模块的主要目标是处理属性 *attribute*，并将事件流与预定义的规则进行匹配。

5.3.3.1 规则与规则库的建立

初始的规则主要针对隐信道建立之初需要同步的特征，首先就可以用基于事件的引擎来筛选；其次还可以通过一些某种隐蔽信道专有的特征来进行筛选。之后的规则可有管理员和用户自行制定，也可以由专家系统或知识库生成。如果有预警反馈机制的话，也可以通过该机制改进规则并将改进后的规则添加进规则库，而本章并不研究该机制，不做进一步表述。

5.3.3.2 匹配引擎设计

对事件流进行匹配的核心是基于事件的脚本匹配引擎 (Policy script-based Engine)。仅仅根据规则匹配的话局限性太大，且不够细致，因此本章引入基于事件的脚本检测来设计匹配引擎。

引擎的核心部分是策略脚本解释器 (Policy script Interpreter) 和面向属性的分析器 (Attribute-specific Analyzers)。其中，策略脚本解释器负责解析规则中的正则表达式，面向属性的分析器则由 Attribute-specific Handler 调用，利用正则表达式匹配的方式将格式化的事件记录与匹配模型库中的规则序列进行匹配；若匹配成功，引擎会根据相应的策略，反查日志信息，然后定位到系统可能存在的侧信道漏洞和风险点，并发出警报。

5.4 实验与分析

本章模拟实现了所设计的系统，并通过实验证明了系统的可行性和检测侧信道威胁的能力。测试环境包括一个具有十个刀片的 Sugon TC4600 机架式服务器，每个刀

片服务器都有两个 2.50 GHz 的 Intel Xeon E5-2670 V2 CPU 和 16 * 8 GB 的 RAM 内存。在上述硬件上我们搭建了开源云计算实验平台 OpenStack，物理平台节点上部署了 Xen、VMWare 和 Hyper-V 等虚拟机管理平台。每台主机上部署 Windows XP 系列与 Ubuntu LTS 系列的虚拟机若干。

5.4.1 有效性验证

5.4.1.1 隐蔽信道利用样例及其检测

本章实现了三个实用的隐蔽信道利用漏洞作为样本，以检测本设计框架的有效性。

A. 如第3章所述，本文实现了基于 Cache 和内存总线的隐蔽信道，具有很高的带宽。其中，基于内存总线的隐蔽信道如果想要传输信息，则需要使用特殊的内存总线访问方法。该方法能够造成特殊的内存总线死锁，这将造成全局访问冲突，进而导致内存访问延时，以此用来构造隐蔽信道。本章对其进行了优化，并将该隐蔽信道作为检测样例。

B. Xen 半虚拟化环境的共享内存机制可以分为三种，其中之一即是页面拷贝，可通过超级调用实现。如果间谍软件已经掌握了某一客户虚拟机的所有权限与资源，它就能够通过超级调用接口的参数偏移量 (offset) 和数据长度 (length) 来传递秘密信息，然后进行正常的页面拷贝操作，这样就绕过了 Hypervisor 的检查，构建出一种存储隐蔽信道。

C. 在虚拟化环境中，物理 CPU 核心通过 vCPU 调度分配给每个用户的进程。然而，这同样也会被攻击者利用来进行隐蔽通信。物理机器中的调度器通常采用 Credit 算法，根据其安全级别为每个 vCPU 分配权重和 cap 值 (vCPU 调用时间的上限)。隐蔽信道可以根据调度策略来构建：当发送方想要发送信息时，可以故意加大其所用 CPU 的负载，当退出 CPU 执行时，接收端确认从发送端导出的 CPU 负载的增加。在接收端确认 CPU 负载增加后，它可以将该比特位转移到云平台外的一些间谍软件中，从而完成隐蔽信道的通信。

表 5.4. 特定属性在检测中的有效性验证

Attr. / Obj.	Cache hit	Memory access	CPU load	Shared mem Req	Params of gnttab
Covert channel A	✓	✓			
Covert channel B		✓		✓	✓
Covert channel C			✓		

5.4.1.2 检测子引擎

为了识别一些云环境下通用隐蔽信道，我们在提出框架的基础上实现了三个典型的检测子引擎。

子引擎 1：检测基于 cache 或内存总线竞争的时间信道。

基于缓存或内存总线的隐蔽信道利用程序通常与异常的内存访问事件相关。首先，当这种隐蔽信道处于活动状态时，缓存负载将大大高于正常值。同时，我们可以轻松地从日志记录和监控模块获取异常的内存访问事件，当然某些云平台 (如 vCenter) 也已经提供了一个阻止参数来计数内存总线竞争和系统阻塞时间，不过本章基于 Xen 实现了其定制的检测子引擎。因此，当这些特殊内存访问操作在特定时间内超过阈值时，即可将其识别为潜在的威胁。

示例规则: *alert MemoryCC allEventNode (memoryBlockNumber* $\geqslant$ *threshold) AND alert CacheCC allEventNode (CachePayloadExpectation* $\gg$ *noiseThreshold)*

子引擎 2：检测基于共享内存的存储信道。

基于共享内存的隐蔽信道通常通过修改授权操作的参数来传输数据。我们可以通过对授权访问和页面拷贝的操作进行测量，找到潜在的隐蔽渠道。观察请求存储器共享操作是否发生频繁，提取偏移量和数据长度，以及将共享页面与调用命令进行比较可以检查是否存在存储信道利用。因为子页面的长度必须小于或等于页面的长度和子页面初始地址之间的差异，因此作为授权操作的参数变量的长度不能太长。因此，每次传输的敏感数据量相当小，所以请求操作会非常频繁。本章利用该特定，设计了检测规则如下。

示例规则: *alert inter-VMCC from VMid to VMid (gnttabOP.lenth* $\geqslant$ *pagesize) AND alert MemoryCC allEventNode (times of shareMemoryRequest* $\geqslant$ *threshold value)*

子引擎 3：检测基于 vCPU 调度和 CPU 负载检测时间信道。

如上文所述可知，基于 CPU 负载的隐蔽信道必须构建在同一个物理 CPU 核上，即隐蔽信道的通信双方的共享资源变量至少包含同一个物理 CPU 核。同时，发送方和接收方通过重复执行程序循环并测量执行时间来传送比特流信息，且在发送信息时攻击者需要保持 CPU 的高占用率。但是从外部观察者角度来看，当执行程序循环时，发送端也几乎占满了整个 CPU 核心的负载。因此，可以根据监视 CPU 负载来执行检测。

通过设定 CPU 占用率阈值，如 75%，此时若占用率超过阈值，则可以记录为 1。相反，当占用率低于 75%，记录为 0。因此，可以不断监控 CPU 利用率来获得一系列连续字符串。如果这个字符串定期发生更改，而实际上没有运行任何程序，那么可以判定虚拟机遭到了攻击，可以找到相应的进程并及时发出警报。

示例规则: *alert inter-VMCC from VMid to VMid (CPUid = CPUid AND Operation = alternate) AND alert CPUCC allEventNode (cpuLoad ⩾ threshold)*

5.4.1.3 实验结果与分析

表 5.5. 总体检测结果

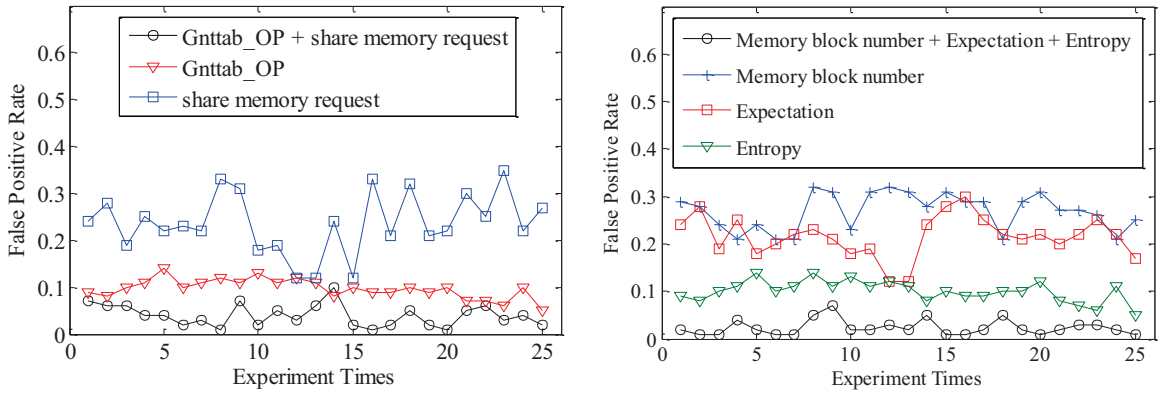
	TP	TN	FP	FN
1	587	30	0	13
2	590	29	1	10
3	590	29	1	10
4	592	30	0	8
5	597	28	2	3
Average	590.2	29	1	9.8
Rate	0.9837	0.9667	0.0333	0.0163

表 5.6. 不同检测方案的比较

Channel types / Methods	CPU load-based	Cache based	Shared memory-based	Others
Active traffic analysis [37]				✓
C2Detection [50]			✓	
Detection with TDR [51]	✓	✓		
BusMornitor [63]		✓		
Our framework	✓	✓	✓	✓

本章为子引擎设计了各种规则，并将时间窗口 (TW) 设置为 10 秒。恶意的发送方程序部署在 MintOS 上，同时在 Xen 管理的 Ubuntu 系统中部署了子检测引擎。为了帮助降低错误率，本章对隐蔽数据的每个信号编码，而不是整个比特流串。这定义了比特错误的效果位置（即信道噪声导致信号判断的错误改变）。为了判观察特定属性 (如内存访问间隔和 CPU Affinity) 的效果和实用性，经过长时间运行多个测试，结果如表5.4所示。在适当修正阈值后，成功检测到隐蔽通信 A、B 和 C 的警报和报警日志。

由表5.5所示，在五组实验中，重复 630 次样本 (600 个正常通信样品，30 个隐蔽通信样品)。平均检测错误率为 9.8，平均缺失率为 1.0。可以明显看到，误报率低于 5%，而漏报率也是微不足道的，几乎不到 2%。换句话说，我们的检测方案可以发现大多数可信隐蔽通信行为，因此我们主要关注虚警率。



(a) 子引擎 1 误检率　　(b) 子引擎 2 误检率

图 5.4 不同的参数对误检率的影响

图5.4(a)和图5.4(b)显示了检测每个子规则添加到子引擎后隐蔽信道利用程序 A 和 B 的误检率。上述 VM 事件记录维护有关硬件状态的许多有用数据，例如 CPU 负载，缓存命中和缺失以及内存访问频率。这是因为上层的所有操作都可以映射到底层的硬件事件。另外，我们可以从 Xenstore 的日志和 LibVirt API 获得的日志中获取所有共享内存请求和所有超级调用参数。

由图可知，当每次加入一种新的属性 *attribute* 时，误报率都为显著下降。因此，只要设置适当详细的规则，则可保证检测的精度。此外，它表明本方案（即通过引入事件关联分析来检测攻击行为）是有效的。

实验还表明，与其他单一检测方案相比，该方案的最大优点可以检测更多类型的隐蔽信道。表 5.6比较了该方案涵盖的所能检测的隐蔽信道类型。

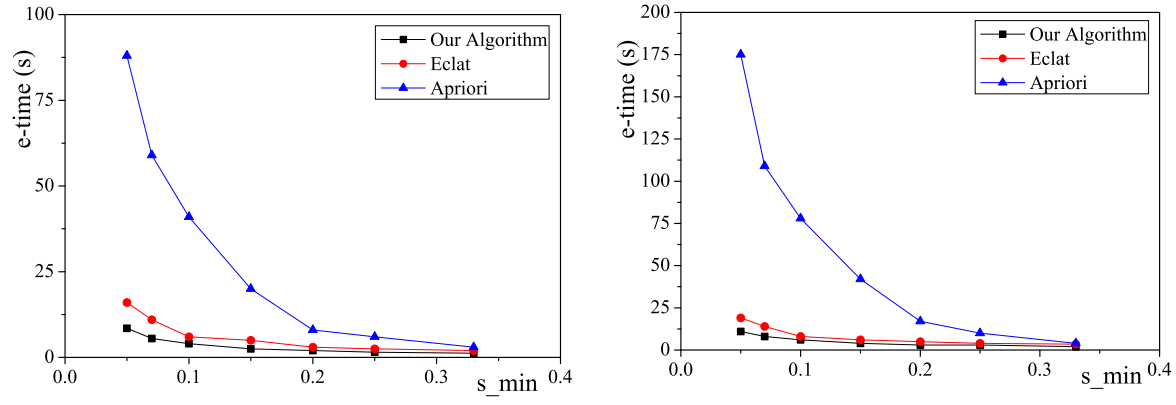
5.4.2 性能分析

为了进一步验证本章的原型系统的性能开销，测量了 XSRM 算法和检测模块的系统资源消耗情况。

5.4.2.1 算法性能分析

显而易见，XSRM 的大小在不同的时间窗口是各不相同的。图5.5(a)和图5.5(b)显示了不同算法在不同最小支持下的性能 (XSRM 时间分析的执行时间)。本算法明显优于传统的 Apriori 算法，处理速度比 Eclat 算法快两倍。

当 XSRM 中的行数 m 接近 10^5 数量级时，共享资源矩阵占用的空间将达到接近 MB 级别。当前的内存条件可能无法满足需求。因此，此算法不适用于挖掘非常大的数量级的项目。然而，当数据集不繁琐 (少于 MB 级) 时，为相对便宜的计算资源换取大量时间是非常有价值的。



(a) 针对 5 秒内产生的 XSRM　　(b) 针对 15 秒内产生的 XSRM

图 5.5 不同算法的执行时间比较

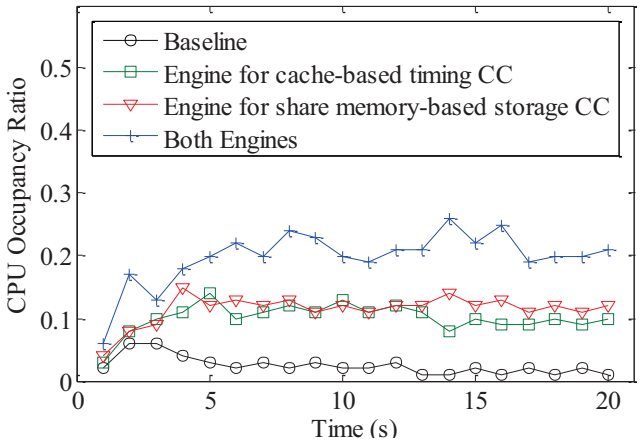


图 5.6 子引擎负载情况

5.4.2.2 原型系统性能分析

为了测试本章方案的性能，分别运行了两个子检测引擎。因为子引擎 3 本身依赖于 CPU 调度和 CPU 利用率检查，所以无需测试其关于 CPU 的性能。更重要的是，基于 CPU 负载的隐蔽信道运行时，CPU 使用率不够准确。图5.6显示了 Xen dom0 在不同情况下每次平均运行十次的 CPU 负载情况。由此可知本方案在一定范围内的性能开销都是可接受的。

5.5 本章小节

本章针对云环境 (虚拟化环境) 下的隐蔽信道的潜在威胁，充分考虑了虚拟化架构与普通计算环境的区别，通过事件记录机制探寻云环境下隐蔽信道的特征，结合共享资源矩阵法遍历和搜索可能存在的隐蔽信道。

本章所设计的检测方案充分考虑了虚拟化环境的特点，在信息收集阶段采用了穿

透性较强的基于物理平台安全配置扫描的方式来收集必要的事件日志信息和安全策略信息；所设计的匹配方案并不使用信息流分析方法探查隐信道，所以不必担心状态空间爆炸等问题；改进了原有的共享资源矩阵法，对事件日志和安全配置文件进行预处理，减少了构建共享资源矩阵工作量；采用事件关联分析的方法实现自动化的检测，能够灵活透明地对云环境安全状态进行评估。相较于同类方案 (如 [50,55])，本方案对所能检测的隐蔽信道范围大大增强。

6 基于虚拟机自省的跨虚拟机侧信道威胁定位方法

为了克服云环境跨虚拟机侧信道威胁定位方法匮乏、现有威胁定位技术不精确等挑战，本章设计了基于硬件特性和虚拟机自省的系统来实时的检测和定位云中存在的跨虚拟侧信道攻击。

本章提出一个新框架，通过动态跟踪共享的处理器硬件行为模式来检测侧信道的存在，计算特定共享硬件上侧信道的可能性。通过提供软件支持的虚拟机自省技术，给出侧信道威胁的精确位置。框架主要思想为：受害者 VM 在执行需要防止旁路攻击的加密应用程序时，具有独特的硬件行为。攻击者 VM 在窃取受害者 VM 信息时同样会发起异常的硬件行为，而通过 CPU 的硬件特性能够记录这两种类型的事件。本文通过使用基于统计特征的检测方法来识别和区分所记录的事件，之后使用硬件提供的精确位置信息，还原底层硬件的语义信息，实现在虚拟机外部的侧信道攻击行为定位。

在详细阐述本章所提威胁定位方法之前，首先简要介绍所需背景知识。

6.1 虚拟机自省

和传统计算环境不同，虚拟化环境引入了更加底层的 Hypervisor 来管理硬件。虚拟机中程序在执行时，由于内存读写的操作过于频繁，很多情况下并不会涉及 non-root 到 root 模式的转换，即不会产生 VM exit。这样一来底层 Hypervisor 很难有效地获取完整的进程语义。而虚拟机自省技术却可以通过 Hypervisor 来监控虚拟机内部行为。因此，本章将其作为云平台中的侧信道威胁的定位手段。

本小节介绍虚拟机自省技术 (Virtual Machine Introspection, VMI) 在国内外的发展现状，并根据技术依赖的不同将 VMI 分为几类介绍，列举了几种代表性的虚拟机自省技术，并对它们的优缺点进行了分析，并借此提出本章方案。

6.1.1 VMI 的出现

随着 VMM/Hypervisor 这类更为底层的操作系统的出现，学术界与工业界开始考虑如何在 VMM 中实现虚拟化安全增强。随着虚拟化系统的发展，其上的系统安全隐患也渐渐变得不容忽视，安全从业人员只能尽可能监控检测入侵行为，通过各种手段尝试堵塞漏洞和修复系统。而由于入侵检测系统存在一个架构难题 — 如果 IDS 部署在目标主机上，管理员可以比较精确地把握主机状态，但是系统极易被攻击；而如果 IDS 部署在网络中，系统虽不易受到攻击，但是管理员很难把握主机状态，检测技术

很容易被回避。基于这些问题，虚拟机自省技术由 Garfinkel [114] 首次提出。

VMI 技术充分利用了 VMM 的隔离性、自省性以及干预性。由于 VMM 具有隔离性，因此运行在目标虚拟机中的软件不能修改或者干预运行在 VMM 或者独立 VM 中的软件，即使入侵者完全控制了被监控主机，也不能够触及入侵检测系统。VMM 的自省性意味着 VMM 能够获取 VM 的一切状态：CPU 及寄存器状态、内存、I/O 设备状态等等。这就使得目标主机很难逃过 IDS 的监控，VMI IDS 能够充分发掘这项功能为其所用。例如，只需要对 VMM 进行小小的改动，一个 VMI IDS 就能够知道运行在 VM 中的代码是否正在试图修改一个特定的寄存器。因此，VMI 技术可以被应用与虚拟机内恶意行为 (如跨虚拟机侧信道操作) 的定位中。

图6.1展示了基于 VMI 的入侵检测系统 VMI-Based IDS 的基本工作原理，VMI 技术通过在需要监控的虚拟机的外部获取需要的硬件、内存等信息，然后在安全的虚拟机中监控目标虚拟机的执行状态。

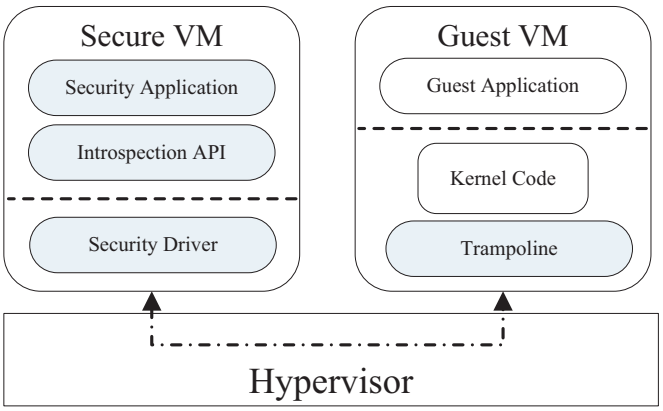


图 6.1 现有流行的 VMI-Based IDS 架构

如图6.1，右侧是运行着被监控模块的 VM。左侧是以 VMI 技术为基础的 IDS，包含以下组成部分：(1) Secure VM 的接口库 (security driver) 通过对 VMM 输出的硬软件状态进行解释，提供了一个对 VM 系统层级的视角；(2) Guest OS 的接口库 (trampoline) 对客户机的应用层和内核层行为进行解释，并将其传递给 Hypervisor；(3) Secure VM 中的安全应用协议模块执行特定的入侵检测引擎。VMM 提供了一个隔离 IDS 与被监控 VM 同时允许 IDS 检测 VM 状态的基础。同时，VMM 也允许 IDS 介入到主机操作系统与虚拟硬件之间的交互中去。

6.1.2 VMI 的应用发展

虚拟机自省在被定义之初，其目的就是在虚拟机之外对虚拟机内的软件运行状况进行检查，当时的各类 VMI 应用均需要虚拟机安装虚拟 I/O 设备并且提供内存存取接口，因而不被认为是透明的监控，也并没有做到隔离性。不过，由于 VMI 具有的隔

离性和安全性，其技术在各种安全领域都得到了发展与应用。在虚拟化环境中，VMM 是系统的主要资源管理者。有些服务通常在 OS 层级执行，如进程调度、内存管理；而如 I/O 调度或者特定种类的安全监控，会在 VMM 中被执行。然而，在 VMM 层执行如此复杂的任务对于 OS 和程序层级的知识要求很高。一个重要的问题是 VMM 中缺乏更高层次的知识，这种问题被称为语义鸿沟。研究人员在虚拟化环境工作中已经初步认识到了这个困境。

就面向一般需求而言，VMI 的实现不应该是以牺牲隔离性为前提实现的 [118]，早期 Garinkel 的实现由于虚拟化平台没有相关接口的缘故放弃了这一要求，允许虚拟机与监控软件直接进行对接的做法。然而坚持隔离性会导致从虚拟机获取的内存二进制数据存在语义鸿沟无法分析的难题。于是在此之后，相继有学者提出了更高级基于硬件资源使用状况的虚拟机自省技术。这些技术允许 VMM 更好地管理系统的资源，也解决了部分地址翻译问题。另外，一些基于 VMM 的服务使用关于在其上运行的操作系统的软件抽象的显式信息来弥补语义鸿沟。部分研究基于软件结构知识还原保证在不失隔离性的情况下还原底层数据，如利用已知的 Linux 内核中描述进程结构的 task_struct 来还原进程列表，并通过 init_task 符号来定位进程列表的表头以得到完整的操作系统进程信息。LibVMI [119] 便是其中的代表和一个较为成熟的解决方案。最初 LibVMI 利用 LibXC 读取内存以支持 Xen，之后也加入了 KVM 支持，不过其本质只是基于虚拟化平台相关接口的封装并加上了一定的语义还原功能。

6.1.3 VMI 的分类

如文献 [120] 所述，根据面对语义鸿沟问题时采取规避策略抑或是收集信息之后自行研究方法进行语义重构，VMI 技术可分为依赖 VM 自省技术和独立自省技术两类。在云环境下客户的透明性需求与硬件虚拟化主流趋势的共同作用下，独立自省技术显然更符合要求。而根据进行语义重构时所需要的语义知识的不同，独立型自省技术又可分为基于软件架构知识的独立自省技术和基于硬件架构知识的独立自省技术两类。

基于软件架构知识的自省技术能够在目标 VM 之外获取内存信息，透明性好；并且所获取内存信息的范围几乎没有限制。其缺点在于用于虚拟机自省的代码当 VM 内存信息发生变化时就极有可能失效；同时，攻击者也有可能利用这一点使得自省程序失效，此方法的鲁棒性也不尽理想。基于硬件架构知识的独立自省技术的透明性和上一种方法一样良好，但是由于硬件信息自身的特质，使得自省程序获得的信息量较小。但较基于软件架构知识的自省技术而言，硬件结构知识的迭代期长，所以自省程序可以长期有效，可移植性良好，攻击者也很难改变操作系统的硬件架构，故而鲁棒性较强，安全性良好。

由上述分析可知，现阶段仍不能开发出来一种完美的 VMI 技术来满足所有的设计要求。Antfarm 虽然能够很好的适应多种操作系统，但是其能够获取的信息极其有限。VMI-based IDS 作为最早的一批入侵检测系统，具有高可靠性，但很难进行二次开发 [120]。Lares 虽然能够解决了在隔离 VM 中安置安全监控程序不能够主动监控的问题，但是可移植性差。LibVMI 使用方便，获取高层信息较全，但其通过 gadget 获取偏移量信息的过程影响了其透明性。这些方案各自具有其优点与缺点，而如何加以利用、取长补短，是本章需要解决的问题。

采用硬件特性与特权域软件如 LibVMI 结合的虚拟机自省方案具备各种自省技术的优点，能够快速定位虚拟机的威胁。因此，本章基于此提出了基于虚拟机软件自省技术与硬件知识结合的侧信道威胁定位方法。

6.2 基于虚拟机自省的侧信道威胁定位方法与系统

本章以 Xen 虚拟化架构为例，假设 Hypervisor 安全可信，阐述客户虚拟机 DomU 用户层侧信道攻击检测与定位方案的整体架构。本章所关注的主要是对受害者虚拟机或是隐蔽信道中的通信双方的攻击点的定位和审计。因为受害者虚拟机被攻击时，通常是处于运行秘密软件或程序的情况下。此时由于秘密程序如 RSA 加解密中，受到攻击的点一般在于控制流中大量的直接分支跳转中 (如蒙哥马利模乘算法)。而这些攻击点均可被处理器的硬件特性 LBR 记录下来。

通过在 Hypervisor 中事先设定性能计数器的溢出阈值，使得 DomU 在执行到可疑操作后开启定位流程。然后将 LBR 中获取的分支跳转信息直接发送虚拟中断通知 Dom0 的报警模块，并将 LBR 信息、虚拟机 ID、时戳 TSC 连同 CR3 寄存器中存放进程页表基址记录到内存缓存区中。然后经由 VM-entry，恢复 DomU 的执行。Dom0 取得记录数据后，首先按虚拟机 ID 对数据进行分离重组，借助 VMI 技术获取 DomU 进程描述结构体 (task_struct) 中的虚拟内存空间 (Virtual Memory Areas, VMA) 信息，准确定位共享链接库的映射基址。然后将 LBR 记录的跳转指令地址转换成在对应链接库中的偏移，在内存分析模块中进行逐条比对，从而定位到具体的威胁进程。整个过程既不需要修改 DomU 的内核、编译器，也不需要 DomU 主动配合传递数据，对所监控的虚拟机完全透明。整体架构如图6.2所示。

对于跨虚拟机的攻击者来说，本章方案可以检测到攻击者，但并不能很好地定位攻击者程序。(这也是无意义的，因为攻击者已经可以控制虚拟机，则攻击者虚拟机的所有用户空间均是恶意的。) 本章的目标在于通过检测攻击，获取攻击的特征，定位攻击的目标，以辅助攻击溯源。在攻击发生后，可以通过虚拟机迁移的方式，将受害者虚拟机迁出所在的物理平台，从而避免攻击的二次发生。

6.2.1 阶段 1 - 硬件事件辅助定位

6.2.1.1 PMU 与 LBR

为了更好的监控调优程序性能，CPU 厂商在 CPU 中集成了一系列辅助调试的硬件支持。例如，性能监控单元 (Performance Monitoring Units, PMU) 能对处理器存储、浮点运算、指令流水、缓存命中率等与程序性能密集相关的性能事件进行计数。分支追踪存储 (Branch Trace Store, BTS) 支持将程序执行的指令分支信息记录到高速缓存或内存中，自定义的存储容量以及溢出中断的特性虽能保证不漏记，但中断处理会使得记录内容滞后于指令现场，频繁的访存操作也会带来可观的开销。最后分支记录器 (Last Branch Record, LBR) 则是一组能记录 CPU 上最近的 16/32 条跳转指令源地址和目的地址的循环寄存器栈。另外 LBR 具有基于分支指令类型过滤的机制。

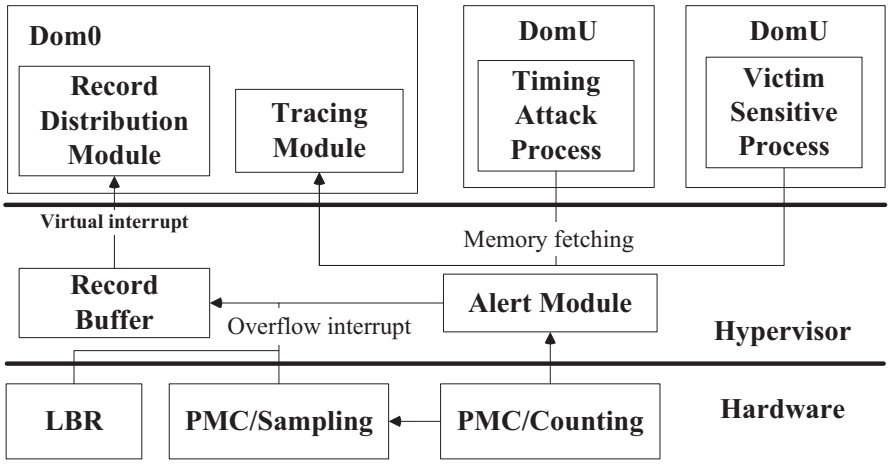


图 6.2 硬件辅助的侧信道威胁定位系统架构

其中，PMU 包含一个重要的计数器 PMC，也被称为 HPC(即 Hardware Performance Counter)。它的基本工作模式之一是基于中断的模式。在此工作模式下，当给定事件的发生超过预定义的阈值或经过了预定义的时间量时就会产生中断。因此，它使基于事件的采样和基于时间的硬件采样成为可能。性能计数器最初是为软件调试和系统性能调优而设计的。而 LBR 则是和 BTS 一样，也记录每个分支执行的源地址和目的地址。但是，LBR 以循环方式将这些地址存储在一组 MSR 中。当 LBR 缓冲区已满时，LBR 不会产生中断，其结果会导致最老的记录被即将到来的记录冲走 [121]。

最近，研究人员广泛利用上述硬件性能计数器来检测安全漏洞 [122,123]。计数器可以揭示程序的执行特性，进一步反映程序的安全状态。此外，性能计数器检测对程序的性能开销可以忽略不计。本章使用性能计数器 PMU 进行侧信道攻击检测，使用分支预测寄存器 LBR 进行威胁的进一步定位。

6.2.1.2 硬件事件特征检测触发报警

现有研究中，有不少方案基于侧信道事件的行为进行异常检测。然而，将基于异常的方法与性能计数器结合，应用于检测攻击却十分困难，这是因为使用性能计数器精确建模来区分正常应用程序和攻击程序往往面临挑战。Cache 侧信道攻击的行为往往与正常的内存密集型应用 (例如，内存流应用 [124]) 十分类似，因此很难仅使用硬件性能计数器 Performance counter 中的数据来进行区分。

另一方面，在虚拟化环境中，实现本章所提出的定位框架存在两个关键的技术挑战，它们分别是：(1) 透明性，即识别受保护虚拟机的敏感操作的执行，而不要求客户修改其应用；(2) 精确性，即精确定位不可信虚拟机的异常缓存或内存的使用模式。本章通过使用性能计数器 PMU/PMC，结合分支记录 LBR 来实现两者。为此，首先使用基于特征的检测技术来检测受保护虚拟机的敏感应用程序。由于这些特征明显，所以不会产生漏报。其次，将检测出的可疑程序的 LBR 信息报告给位于特权域的定位分析模块。因为 LBR 能够给出地址级别的精确信息，所以结合虚拟机自省技术和内存分析技术，即可精确定位到发出这些可疑操作的进程。若该进程不属于正常的内存访问，则可以认为它们属于利用共享的缓存和内存而进行的侧信道攻击。该攻击检测与定位方法由于能获得精准的事件发生地址，因此可以排除由于异常检测技术带来的误报。

另外，由于性能计数器 PMC 与分支记录器 LBR 现阶段都没有溢出中断机制，这样直接读取其数据会导致读取时的值与攻击实际发生时的值，由于时间上不同，已经有很大不准确性。这也被称为硬件计数的 skid 问题 [125]。为了解决该问题，在选择利用 PMC 的同时，引入 PMU 中断机制。

为了收集 PMC(即 HPC) 的值，本章采用了 Linux 中的 Perf 工具以及开源库 Libpfm [126]。Perf 事件支持两种收集 HPC 值的模式：HPC 值的计数模式和采样模式，当 HPC 达到预定阈值时触发溢出非屏蔽中断 (NMI)。在计数模式下监视总事件 (如 Cache miss) 的数量，并设 T_{NMI} 为采样模式下 NMI 溢出的阈值。

不同的硬件事件其特征不同。对于内存侧信道事件来说：检测隐蔽信道的第一步是识别硬件资源争用背后的事件。在存储器总线隐蔽通道的情况下，要监视的事件是内存总线锁定操作。比如在基于整数运算的侧信道 [127] 的情况下，要监视的事件是来自一个进程 (上下文) 的除法指令和等待来自另一个进程 (上下文) 的指令占用除法器的次数。再比如说在 controlled side channel 侧信道 [90] 中，想要检测的事件是多次的一个 page fault 和一个对页表项交替发生的次数。

对于 Cache 侧信道事件来说：需要进行 Cache 振荡模式检测。所谓振荡模式是指：为了传输 ‘1’ 或 ‘0’，木马或间谍程序在彼此之间交替地创建足够数量的 Cache 未

命中事件，这使得间谍程序可以根据平均访问时间来解密所发送的比特 (即通过观察 cache 是否命中)。与通过改变事件间间隔 (观察为突发和非突发) 执行定时调制的结构不同，基于缓存的隐蔽信道依赖事件的等待时间来执行调制。这导致了木马和间谍环境之间的冲突振荡模式。振荡可以被认为是事件序列中周期性的特性，这与事件序列中高频事件发生的特定时段的爆发不同。通过测量事件序列自相关性可以检测到这样的事件 [128]。报警阈值 T_{NMI} 可通过监测 CPU 事件中的 LLC/L1-loads、LLC/L1-stores、LLC/L1-load-misses 或 LLC/L1-store-misses 来设置。

对于内存总线冲突的隐蔽信道事件来说，通信双方均为攻击者，定位时可寻找虚拟机中定期频繁发生 lock 事件。因此，本章使用关联分析检测算法来检测是否存在循环突发模式的事件序列。该步骤由两部分组成：(1) 检查事件序列是否有显著的竞争序列 (振荡脉冲串)；(2) 确定时间序列模式是否表现出突发模式。报警阈值 T_{NMI} 可通过监测 CPU 事件中的 bus-cycles 或 BUS_LOCK_CLOCKS 来设置。

本章主要关注侧信道的威胁定位，关联分析检测算法详细介绍参见 5.2.4节。

6.2.2 阶段 2 - 定位信息收集

此时 Hypervisor 在监控虚拟机时，需要解决以下两个问题。(1) 如何区分不同虚拟机的事件。如图 6.3所示，在虚拟化环境中，vCPU 是为了解决虚拟机时分复用的重要数据结构，它被分配给每个虚拟机 Dom，作为操作虚拟机当前物理 CPU 的实例。但 CPU 调度机制使得不同时刻 vCPU 对应的 CPU 是时刻变化的。所以当前 CPU 中类似 LBR 等硬件信息的内容并不一定就是当前虚拟机产生的。(2) 当前 Hypervisor 没有提供 LBR 等特殊寄存器的模拟机制，现有基于硬件辅助的攻击检测方法不能直接运用在虚拟化环境。Hypervisor 如何使用真实的硬件信息并将其解释为可用于检测攻击的特征正是本章需要解决的问题。

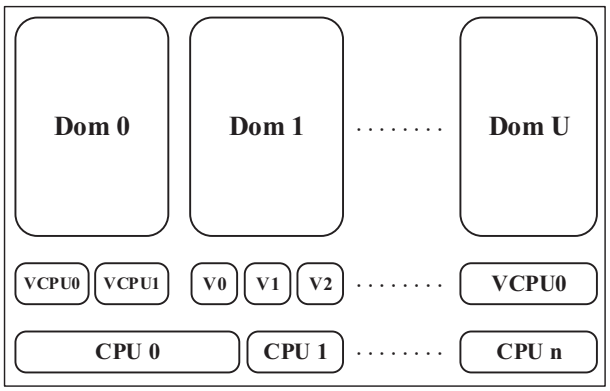


图 6.3 CPU 与 vCPU 映射关系

由于 LBR 中记录着精确的分支跳转地址，而这些地址信息是非常密集的，因此可以被用于记录当前进程的某些分支信息。在此引入 LBR 作为定位的核心记录模块。

LBR 由 IA32_DEBUGCTL 寄存器的最低位控制开启。然而，如果时刻不停的记录虚拟机的所有跳转，产生的信息体量无疑相当大且冗余度非常高，严重影响检测性能。由于攻击者在劫持控制流后要做出进一步恶意的操控和破坏，必将进行关键的系统函数调用。因此本章只记录所关心的跳转信息，保持较高检测有效性的同时降低开销。

Intel 提供 64 位的 MBR_LBR_SELECT 寄存器用于设置分支跳转类型过滤，其中某一比特位为 1 时，代表过滤相应类型的跳转指令。由于本章仅关心用户层的跳转，故保持表6.1所列标志位为 0，即通过 WRMSR 指令将 0x1 写入 MBR_LBR_SELECT 寄存器。

表 6.1. MBR_LBR_SELECT 过滤设置

标志位	名称	含义
1	CPL_NEQ_0	发生在用户层的跳转
63:10	-	保留位必须置 0

在读取 LBR 时，本章采用 C 语言内嵌汇编的方式来编写 LBR 设定函数。

```
void _intel_lbr_enable(void){
        unsigned long long select;
    asm volatile (  ''xor %%edx, %%edx;''
                    ''xor %%eax, %%eax;''
                    ''inc %%eax;''
                    ''mov $0x1d9, %%ecx;''
                    ''wrmsr;''::    );
    rdmsrl(MSR_LBR_SELECT,select);
    wrmsrl(MSR_LBR_SELECT, SELECT_CONFIG);
}
```

使用同样方式编写 LBR 读写函数。在读取 LBR 数值时需要首先获取物理主机的 CPU 所述系列与模型号，以确定读取条数。本章的设计与实验均在 Intel i7 型号的 CPU 上完成，而 Skylake 系列处理器 LBR 均为 32 条，因此需要读取 32 位的 LBR 信息。由于 CPU 在发生异常时直接跳转到中断处理程序，这将最多产生 1 条新的分支记录，通过配置 LBR 过滤了 ring 0 层的分支，选择性地只记录了用户层中产生的分支跳转，保证了冻结后的 LBR 记录不会发生任何覆盖。

另外在某些型号的处理器上，即使 IA32_DEBUGCTL MSR 中 LBR 标志位被清零但 TR 标志位保持置 1 时，CPU 将仍然继续记录新的分支信息更新 LBR 栈。这是因为这些处理器使用了 LBR 栈中的记录条目来生成 BTS 记录。所以在本方案在实现 PMI 处理中断冻结 LBR 时的同时，也要将 IA32_DEBUGCTL 第 6 位 TR 标志清零。

```
void read_lbr(void* info){
        tos = intel_pmu_lbr_tos();
        for (lbr_pair_i = 0; lbr_pair_i < NUM_MSR; ++lbr_pair_i) {
        unsigned long idx = (tos - lbr_pair_i) & (NUM_MSR - 1);
                msr_from_counter1 = MSR_P4_LASTBRANCH_0_FROM_LIP + idx;
        msr_to_counter1 = MSR_P4_LASTBRANCH_0_TO_LIP  + idx;
        asm volatile(    ``mov %4, %%ecx;''
                                ``mov %%eax, %0;''
                                ``mov %%edx, %1;''
                                ``mov %5, %%ecx;''
                                ``mov %%eax, %2;''
                                ``mov %%edx, %3;''
            : ``=g''(ax1f), ``=g''(dx1f), ``=g''(ax1t), ``=g''(dx1t)
            : ``g'' (msr_from_counter1), ``g'' (msr_to_counter1)
            : "%eax", "%ecx", "%edx"        );
                lbrentries[lbr_pair_i].from = ax1f;
        lbrentries[lbr_pair_i].to = ax1t;
        }
}
```

值得注意的是，在设置 IA32_DEBUGCTL 寄存器时，同时需要设置在该虚拟机所对应的 VMCS 中的 IA32_DEBUGCTL 字段，并且在设置时，为了保险起见，需要在每次发生 CR3 的 VM exit 时，在 VM exit 的 handler 中进行设置，具体用 C 语言表示如下。

```
cpu = current->arch.hvm_vmx.active_cpu;
on_selected_cpus(cpumask_of(cpu), read_lbr, current, 1);
/* enable the LBR */
_intel_lbr_enable();
__vmwrite(GUEST_IA32_DEBUGCTL, 1);
```

当对 LBR 栈所记录的内容进行读取访问时，必须暂停 LBR 的更新。在现代 CPU 处理器的典型代码场景下，每秒更新的分支记录可能高达百万规模，而 LBR 栈的有效容量最大只有 32 条。这样一来如果只是直接通过把 LBR 的开关配置寄存器 IA32_DEBUGCTL 的最低位置 0 来立即停止新纪录的产生是无法起到预期效果的，因为在该操作执行的些许延迟中，不断产生的分支跳转记录将在瞬息之间覆盖原有 LBR

栈中本文所需要的内容。唯一正确的使处理器立即停止 LBR 记录产生的方法是使用性能监控中断 PMI，该特性比先通过 IA32_DEBUGCTL 寄存器的第 11 位置位来开启。本方案使用 CPU 性能计数器配合，将 PMU/PMC sampling mode 设置为当分支计数指标超过阈值时发出 PMI 中断。

当 PMU/PMC 中的数据达到检测阈值后，即开启 PMU/PMC sampling mode，发出中断，陷入 Hypervisor，同时通知位于 Hypervisor 中的 LBR 信息收集模块记录当前的 LBR 值。与此同时，本章通过配置 control registers accesses 字段 (详见 [40])，能够使得 CR3 寄存器中的值发生变化时产生 VM exit 从而进入 Hypervisor 的处理。此时 VMCS 会记录 DomU 中包括 CR3 和 EIP 在内的寄存器现场。由此本章使用不同的 CR3 值来区分所保护的进程。具体来说，HPC 监视客户虚拟机内执行的所有进程时，有两种情况需要考虑：(1) 在信息收集间隔期间不发生上下文切换，(2) 在间隔期间发生上下文切换。

当 T_{NMI} 间隔期间没有上下文切换时：溢出 NMI 导致 VM 退出，处理器会切换到 root 模式；此时通过检查 VM exit reason 字段，判断该 VM exit 是否是由 NMI 溢出导致；若是，则进行下一步的检测操作。

当 T_{NMI} 间隔期间存在上下文切换：客户虚拟机中的多个进程会累加 HPC 的值，导致误报。为了避免该问题，本章使用查找表来存储客户虚拟机的每个用户级进程的在 T_{NMI} 间隔内的签名。在本章现阶段的实现中，由于上下文切换会导致 VM exit，并保存当前进程的 guest 虚拟机的状态，因此本章将 CR3 值作为签名存入属于当前虚拟机的 VMCS 中，当 VM-entry 时，通过利用 VMCS 能够保存 CR3 的特性，加载下一个进程的 guest 虚拟机的状态。在 Dom0 中，本章使用 VMI 读取所有虚拟机的 pgd 字段，本章使用这些值作为主键来更新进程的查找表条目。在 Linux 内核里，pgd 字段后 32 位的值等同于 CR3 内后 32 位的值，因此使用该方法能够区分虚拟机内不同的进程，并将这些进程各自对于 HPC 的产生的增量分开。具体见节6.2.2.2。

6.2.2.1 信息分发与传递

本方案通过将 Hypervisor 进行的轻量级扩展来实现信息分发与传递。(1) 利用现有的主机系统设施从在所有现代商品处理器中可用的硬件性能计数器收集微体系结构特征；(2) 利用现有的虚拟化框架非侵入性地交互以监视 VM 的侧信道攻击事件，尽可能地减少性能损失。

现有的侧信道检测研究大部分是针对某种特定的侧信道攻击或是特定的硬件资源所进行的研究。而这些检测技术均可完美地补充到本章所提出的检测框架中。同时，为了能够审计本章在阶段 1 中所捕捉到的事件，用户通过 Hypervisor 导出的特殊 API 来执行用户授权检查。这是为了防止敏感的系统活动信息被攻击者利用。本章不采用

单独的守护进程来触发和记录从 Hypervisor 中得到的信息和时间戳，而是通过添加轻量级代码，以避免扰乱系统状态，并尽可能准确地记录性能计数器等硬件的数值。为了进一步减少干扰效应，在特定的 CPU 核上运行分析和定位模块。

在此，本章引入 Xentrace 作为辅助手段，通过重载 Xentrace 来实现信息分发与传递。本章重用了其事件记录机制和消息传递机制，设计了一种能够迅速将 Guest-VM 的 VMI 检测需求连同 VMI 程序所需的参数一并传递至 Dom0 的机制。该机制具备通用性，且不增加过多的源代码，对系统的性能影响极小，最大程度上维护了系统的安全性和可移植性 [129]。

本章为记录 LBR 数据设计了专门的事件格式与事件解析器来存储解析其数据。除了 Domain-U 的标识 (Dom_ID) 与时间戳，Hypervisor 还需要提供 EPTP 索引，用以告知 VMI 程序 Guest-VM 所使用的 EPT 页表基址.。缓冲区中的每条事件记录格式如图 6.4所示.

TSC	Event Mask	Dom_ID	LBR_INFO

图 6.4 事件记录格式

本模块具体运行步骤如下：

首先在 Dom0 中启动 Xentrace：

(1) Hypervisor 检测是否有新的虚拟中断注入，若存在则根据上述事件格式解析传入的参数；

(2) 调用 Dom0 中的虚拟机自省程序；

(3) 将接下来需要传入 VMI 程序的参数传入到 Xentrace 的核心函数 TRACE_ND() 中，并通过该函数将数据写入 Hypervisor 提供的缓冲区中；

(4) 向 Dom0 注入虚拟中断 VIRQ_TBUF。

6.2.2.2 虚拟机内存信息获取

如图 6.5所示．在 VMI 开启的情况下，访问某字段时，可以通过首地址加偏移量直接访问内存。其中重要节点简要介绍如下：

System.map. Linux 系统内核编译后产生，用于存放当前内核的符号信息和虚拟地址；

task_struct. 存放 Linux 进程或线程的数据结构，其容纳了一个进程或线程的所有信息；

mm_struct. 内存管理数据结构；

pgd_t. 页目录表地址，也就是存入 CR3 地址对应的虚拟地址。

需要注意的是，本章检测的对象有很大可能是存在某个 (加密) 共享库的函数，而

在特权域 Dom0 中的解析器提取的 16/32 条 LBR 间接分支地址是线性地址，是包含了各个共享库起始地址偏移的信息，需要减去对应的基址才能用于精确定位。另外，当 DomU 中开启了地址空间随机化时 (ASLR)，同一个共享链接库在不同应用程序的装载基址是随机的。本章在此描述如何借助 VMI 技术获取指令所属进程的内存描述符，从内存映射信息中收集各个共享连接库的装载基址。

操作系统内核均有相应的数据结构来存储进程的内存映射信息，本章以 Linux 系统为例，采取 LibVMI [130] 提供的方法来获取数据元素 (init_task) 的初始地址、偏移量及数据类型等信息，线性扫描 task_struct 链表结构找到 CR3 对应的进程，直至重构完成数据结构中的每个数据元素。然后从结构体出发沿指针或引用的方向依次寻找 mm_struct。

同时，为了还原捕获 LBR 信息所在的进程 (Pid)，本章采用遍历查找表的方式来对所获取的 CR3 信息进行反查，如图 6.6。通过扫描 mm_struct 结构体，可以获得名为 pgd_t 的字段。该字段即为页表目录的地址，也就是需要传递到内存管理单元 MMU 用于地址翻译的最重要参数，它也是每次进程切换后 CR3 寄存器中存入的地址。本方案将所有 VM 的进程全部扫描后，建立进程 Pid 和其 pgd_t 的对应关系并保存在实时更新的查找表中。之后，当需要查询 CR3 所对应的进程时，即可通过查表得出进程号。

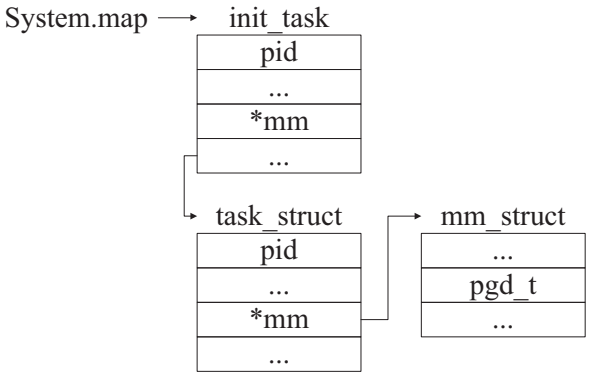


图 6.5 内核空间重要字段链接关系

当检测系统检测的某虚拟机遭受侧信道攻击时，将调用 VMI 程序，暂停该虚拟机，并在 Dom0 下遍历所有虚拟机的当前进程，定位到目标进程，之后可以对当前内存页的代码进行静态分析。由 3.1.2 节可知，对于基于 Cache 的侧信道，则需要借助所获取的 LBR 信息，分析其中地址所在的进程或共享库，系统记录其进程号和其所在的虚拟机，并发出警报。而对于基于内存总线的侧信道，定位系统若能匹配出如 atomic_set，atomic_fetch_add 等原子操作指令，则可进而认为该进程是恶意的。

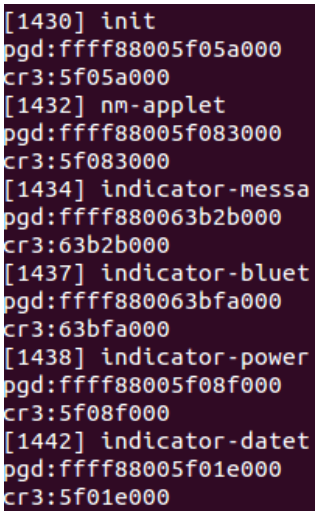


图 6.6 部分通过 LibVMI 获取的进程 PID 与 CR3

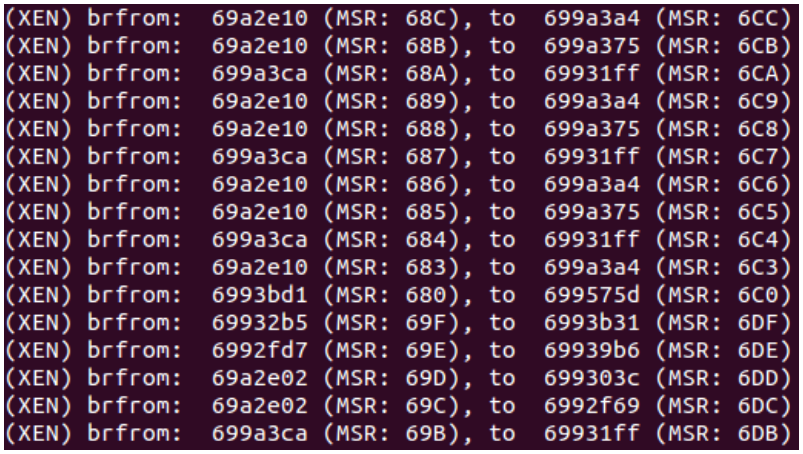


图 6.7 LBR 跳转信息

6.2.2.3 内存分析

Linux 平台上的实时内存分析传统上难以执行。内存分析需要精确了解内存中的结构布局信息，通常通过调试编译时生成的符号来获得。然而，Linux 内核是高度可配置的，意味着调试信息很少适用于除生成调试信息之外的系统。

Rekall 是一个先进的审计和事件响应框架。它作为一个内存分析取证框架，现在已经可以被集成到其他平台上 (如 LibVMI)。Rekall 为事件响应和审计提供了一个端到端的解决方案。借助 Rekall，可以利用生成的精确调试信息来更精确地定位重要的内核数据结构，而不是像其他工具一样依赖启发式或是基于特征来进行模糊的威胁定位。本章通过使用 Rekall，来提供最稳定和可靠的内存分析。

结合 Rekall 和 LibVMI，可以获取如何系统中的各类信息。图 6.8展示了所监控的实时系统调用。本阶段分析时，系统依次暂停所有虚拟域，使用 VMI 在 Dom0 中读取当前虚拟域的当前页框。扫描到恶意虚拟机时，定位到恶意进程，报告其虚拟机 ID，通

过 CR3 反查出其进程号。提取恶意虚拟机的 rekall 镜像 (命令：python rekall/rekal.py -f $HOST_IMAGE –ept $EPT_VALUE imagecopy -O guest_vm.raw)，通过 rekall 提供的交互接口 (命令：p task)，查找 LBR 地址信息 (如图 6.7)，定位到调用函数名。

```
[SYSCALL] vCPU:0 CR3:0x39ed5000,rtkit-daemon UID:107 linux!sys_write
[SYSCALL] vCPU:0 CR3:0x39ed5000,rtkit-daemon UID:107 linux!sys_clock_gettime
[SYSCALL] vCPU:0 CR3:0x39ed5000,rtkit-daemon UID:107 linux!sys_read
[SYSCALL] vCPU:0 CR3:0x39ed5000,rtkit-daemon UID:107 linux!sys_poll
[SYSCALL] vCPU:0 CR3:0x39ed5000,rtkit-daemon UID:107 linux!sys_poll
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_clock_gettime
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_inotify_add_watch
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_clock_gettime
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_poll
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_clock_gettime
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_inotify_add_watch
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_clock_gettime
[SYSCALL] vCPU:0 CR3:0x367ca000,gmain UID:0 linux!sys_poll
```

图 6.8 系统调用监控

6.3 实验与分析

本章使用两台虚拟机模拟攻击过程。客户机系统使用 Ubuntu 14.04 操作系统，发送端虚拟机使用 Windows 7 操作系统。检测平台通过修改 Xen 4.6.0 实现。首先，本章同样通过前文所述的攻击构造方案，构造了两种不同的侧信道攻击，并且采用不同的参数对不同的对象进行了攻击。

针对不同的负载，本章选择了 OpenSSL 中的多个加密算法，并实现其应用程序。同时，使用不同的参数 T 和 S 构建了基于 LLC 和基于内存总线的侧信道攻击。在进行测试时，为了使实验受各类误差的影响较小，每个加密应用程序与攻击程序同时在两台虚拟机内执行 10 次，并通过事先实现 Hypercall 接口的设定 T_{NMI}，对攻击检测进行阈值的设定。

6.3.1 有效性分析

在进行阶段 1 的检测前，首先需要确定性能计数器阈值。本方案以性能计数器中的 L3 Cache miss 为指标，当它超过阈值时则告警进入阶段 2 的定位。值得注意的是，Cache miss 次数、指令数和分支预测数等指标均可作为侧信道攻击的检测标准 [55]，本章重心在于阐述侧信道威胁定位，因此选用了区别较为明显的 L3 Cache miss 作为实验指标。

如图6.9所示，本文以三种常用且易受侧信道攻击的加密算法为例，当受害者虚拟机未被攻击时，通过离线学习，确定 AES、Elgamal 和 SHA512 的检测告警阈值分别为每毫秒 20000、24000 和 54000 次分支指令。当超过此阈值时，则认为此时分支指令过多且异常，即存在侧信道攻击的可能。检测定位系统在事先已开启，驻留在 Dom0，

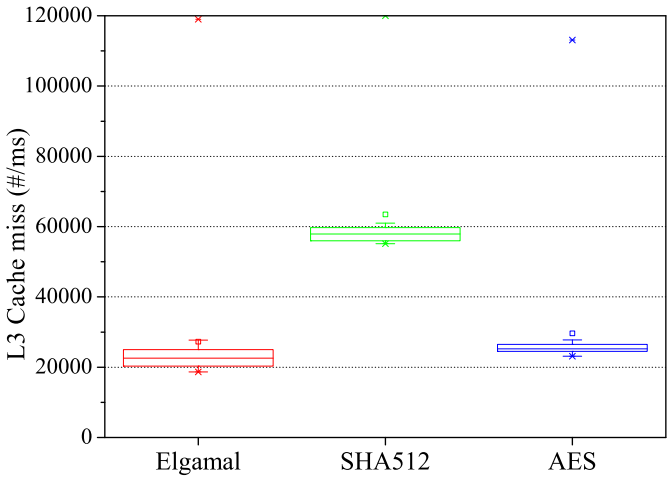


图 6.9 不同加密算法阈值

通信双方虚拟机开始通过基于内存总线的隐蔽信道通信。开启阶段 1 后，检测系统成功检测到每次基于 LLC 的侧信道攻击和基于内存总线冲突的侧信道攻击。

然后，开启阶段 2，系统定位到受害者虚拟机的进程，同时通过时间戳跟踪到了攻击者虚拟机的攻击进程，并发出了警告。系统同时一并将 LBR 信息写入到了日志，通过日志中提供的进程信息与详细跳转地址，开启了内存扫描模块，分别跟踪攻击者与受害者虚拟机的相应进程。告警信息前台展示如图 6.10。

图 6.10 告警日志记录

6.3.2 性能分析

检测阶段：由于检测算法运行在 Hypervisor，因此它对客户机系统性能的影响很小。检测算法 (详见第5章) 每隔一定时间被调用一次，每次所需要时间由阶段性的数据大小决定。注意到本章在此可以通过进一步的优化检测算法来提高性能 (详见节5.2.4)。

在定位阶段，按照本章的策略，在利用 rekall 扫描内存时，设平台运行 N 台虚拟机，则遍历当前进程的当前内存页框所需要的平均时间复杂度为 $O(N)$。内存分析是在每个虚拟机系统时间片结束时调用的，每次计算需要约 0.7 至 1.3 毫秒。同时，本方案可以利用文献 [129] 所介绍方案来避免暂停目标虚拟机，则触发自省时的开销可忽略不计。

6.4 本章小节

本章针对云计算环境下的侧信道攻击，提出基于虚拟机自省的侧信道攻击定位方法。该方法通过硬件特性记录攻击的特征，引入统计规律，得到检测结果并给出距威胁最近的分支的线性地址，最后利用虚拟机自省技术定位恶意进程。实验结果表明该方法能准确实现恶意进程的定位，并且具有较小的性能开销。现阶段同类方案大多是针对传统计算环境的已知威胁的定位与审计，本方案能够提供痕迹较少的跨虚拟机侧信道威胁的定位和审计，是对该领域的创新。

7 基于按需时间模糊的跨虚拟机侧信道防护方法

近年来，许多工作提出了针对基于同驻的时间信道攻击的防御方法。对时间源增加噪声是最早的时间攻击缓解方案之一 [76]。很多针对访问驱动的时间侧信道防御聚焦于减小资源共享的程度或粒度，如 [71, 78, 79]。然而这些防御方式必然会限制资源共享的灵活性，也通常会影响性能、效率和资源利用。针对这个问题，研究学者提出通过干预攻击者测算事件的时间来实现时间信道的防御。其中大部分都是针对一种名为 “时间戳计数器 TSC” 的时钟源来进行的。这是因为：即使在理论上攻击者有很多种方法多事件的时间进行测量，类似于测量足够细粒度的事件来提取密钥或是实现其他攻击，在 x86 平台上攻击者通常还是使用 TSC 作为其时钟源 [18, 26]。因此，在之前的研究中，降低 TSC 的保真度被认为是干预这类攻击的有效手段 (例如 [77, 83])。

然而降低 TSC(或是其他时钟源) 的保真度 (置信度) 不可避免地会给操作系统或是时间敏感的程序带来负面影响，并且置信度降低的程度越大，带来的负面影响也越大 [77]。因此，现有云平台需要一种能够按需自由改变时间保真度的技术，来尽可能地避免这种负面影响。也就是说，TSC 保真度的降低不一定随时间而变化，而是可以根据用户应用程序的操作进行调整。例如，由于 AES 操作比 RSA 操作快得多，因此在 AES 操作期间防御侧信道所需的 TSC 保真度的降低可远远小于 (或远大于) 在 RSA 操作时的需求。

本文提出了一种基于动态按需时间模糊的侧信道防护方案。不同的是，其他同类方法通常是将程序执行时间的测量值强行定为某个固定值 [78, 79]，或直接结合多台虚拟机上程序各自的时间，强制性地提供一个相对模糊的测量值 [103]，然而这些方法均开销过大，并不适合商用云平台使用。本文研究了如何动态地降低时间戳计数器的精度，同时以很少的开销提供按需保护，从而大大增强了方案灵活性。本文方法采用硬件新特性，大大节省了开销，使得该方案在云平台的大规模使用成为了可能。

7.1 需求分析

如前文所述，本文所做工作面向云环境中的最底层 — 基础设施即服务 (IaaS)，其大体模型是在相同的共享计算机硬件上多个客户执行多个虚拟机。类似这样提供云服务的云提供商的包括国外的 Amazon EC2 和 Rackspace，国内的包括阿里云、腾讯云、青云等。由于这种架构上共享计算机硬件的性质，侧信道攻击可能在不同客户的虚拟机中同时执行。

威胁模型. 具体来说，本文中考虑的攻击即基于时间信道的攻击，通过在受攻击者 VM 可以观察到的共享硬件上的某些事件之间的时间内，测量受害者虚拟机的操作或是扰动来进行攻击。我们假设云环境中的虚拟化软件是值得信赖的，且基础设施提供可靠的访问控制，以防止攻击者虚拟机直接或通过特权升级形式访问受害者虚拟机。

在这样的威胁模型下，本文考虑灵活有效的防御的同时需要满足以下需求：

“恰到好处地”时间模糊. 本文的主要创新点在于方案能够区分不同粒度的事件，从而防御各种不同程度侧信道攻击。例如，由于 AES 操作比 RSA 操作快得多，因此它们相应的成功攻击时间并不相同。比如，受害者虚拟机可能会要求在执行 AES 操作时将时间精度设为纳秒级，而在其 RSA 操作期间将时间精度设为微秒级。

按需的保护. 虽然潜在的受害者虚拟机中有敏感信息需要保护，而且可能仅仅只有受害者的虚拟机准确地知道涉及到这些数据的敏感操作是如何发生的，但是我们依旧假设有一个强大的攻击者能够获得受害者虚拟机的这些精确操作的信息 (如调用函数，何时发生等) 以及攻击者虚拟机可以观察的相应事件来尝试推断敏感信息。我们假设这样的敏感操作占受害者虚拟机的整个工作量的一小部分，所以受害者更愿意使用 “按需的” 保护来最小化性能开销。因此本文针对此情况，为了使得在受害者虚拟机能够动态的调整时钟，在需要开启本方案时才开启保护。例如，在保护服务器中的 HTTPS 响应时，本方案能够在进行密钥协商和数据包加密时被开启，在处理请求和发送包时被关闭。

虚拟时间的可用性. 除了在云端运行虚拟机所需的安全功能之外，租户可能还需要从云平台获得时间信息，以实现其通用计算的目的。例如，运行 Web 服务器的 VM 可能需要以微秒精度的时间戳用于信息记录的目的。因此，防范侧信道攻击的解决方案不应该在很大程度上影响时间信息的使用。

7.2 基于按需时间模糊的侧信道防护方法与系统

由上文所述，跨虚拟机时间信道的防御方法往往针对某种单一的攻击。因此，本章提出一种面向系统全局的，能够在一定程度上杜绝各种时间信道的防御方案。该方案借助于硬件虚拟化的特性，利用了新式且广泛存在于 Intel CPU 中的处理器虚拟机功能 — VMFUNC，来使得本方案在各种粒度下的需求中都能得到应用。

本系统提出一个简单的接口，用户应用程序可以通过该接口将平台上其他用户虚拟机观察到的 TSC 值进行模糊，并模糊到指定的级别 (即在向用户呈现正常的 TSC 读数之前，将 TSC 的低位数全部置为零)。此外，本章展示了通过新型的 VM-Function(VMFUNC) 技术，可以为用户虚拟机提供一个低开销的接口调用机制，以便动态地调整 TSC 的保真度。显然，这种保真度调整需要足够迅速，且不会对诸如一些正常的应用如加解密或是提供 Web 服务产生明显的影响。得益于 VMFUNC 技术的高性能，用户虚拟机

可以灵活地调整 TSC 保真度，而不会因此承担较大的性能开销。此外，本方案量化了 TSC 在防御简单而有力的隐藏信道攻击中必要的保真度下降的程度，得出了其确定的值，并可以用于防御基于 LLC 的和基于内存总线占用的时间信道攻击。

然而，在允许用户调整 TSC 的保真度的情况下，用户过度降低对其他用户的 TSC 保真度有可能对平台产生不利影响。实验表明虚拟机对 TSC 保真度降低并不敏感，然而显而易见这种结论并不在所有情况下都适用。一种直接应对过度降低 TSC 保真度的方法是对降低保真度的位数或持续时间施加限制，例如，限制 TSC 置零的位数的上限，或限制各用户要求降低 TSC 保真度的平均时长。在这种限制下，公有云中的用户可以被收取一定的额外费用，反之，也可以因其同驻用户在一定时间内降低 TSC 保真度而获得补偿。本章在此仅考虑该系统实现的技术细节，并给出一些量化的实验数据。而类似降低保真度的位数或增加持续时间等一些方案则属于云服务提供商的服务策略，本章在此方面不做过多讨论。

7.2.1 相关硬件虚拟化技术

由于本章提出的基于按需时间模糊的侧信道防护方案需要用到处理器对于 RDTSC 的模拟和截获以及 VMFUNC 指令的使用和重载，因此本节介绍其相关技术。

7.2.1.1 TSC 模拟

TSC 是继 Pentium 之后的 Intel 处理器拥有的单调增加的 64 位寄存器。过去，TSC 每过一个处理器周期都会增加，但是在最近的处理器中，即使处理器改变频率 (例如，降低处理器的功耗)，它也以恒定的速率增加。TSC 被 x86 程序员称之为时间的最快、最高精度的测量手段，因此经常被作为性能监控的基础 benchmark。并且由于它被保证是单调递增的，并且 64 位保证在 10 年内不会过时，所以它经常会被用作随机数或唯一的序列标识符。

在大多数较早的 SMP 和早期的多核机器上，TSC 在处理器之间并没有同步。因此，如果一个应用程序在一个处理器上读取 TSC，则由操作系统移动到另一个处理器，然后再次读取 TSC，可能会出现 “时间倒退” 的现象。当 TSC 敏感的应用程序从单处理器移植到 SMP 环境时，这种单调性的损失会导致许多难以理解的应用程序错误。因此，许多应用程序 (特别是在 Windows 环境中) 删除了对 TSC 的依赖，从而丧失了性能和精度。然而，在最近几代的多核机器上，TSC 可以在所有电源状态的所有处理器之间同步。因此，TSC 对于应用程序依旧是十分精确的，甚至更新的操作系统正在使用，并且在这些最近的机器上运行时依赖于 TSC 进行关键计时任务。

RDTSC(Read Time-Stamp Counter) 指令是读取 TSC 寄存器的关键指令，它能够将 TSC 的值加载到 EDX 和 EAX 寄存器中，是本方案所要截获的对象。它的操作码是 OF31。在 x86-64 模式下，RDTSC 会清空 RAX 和 RDX 的高 32 位，并将

TSC 中的值加载到 RAX 和 RDX 的低 32 位中。RDTSC 指令不是串行指令，它不一定等到它之前的指令全部执行完成之后才能读取计数器。同样的，它之后的指令可以在 RDTSC 指令进行读操作之前进行。如果软件要求在所有先前的指令之后才能执行 RDTSC，它可以使用 RDTSCP 或执行序列 LFENCE 来保证指令的顺序执行。所以在使用 RDTSC 时，需要进行一定的封装 (详见节3.1.1.1)。

在 Xen 4.0 及之后的版本中，可以为每个域指定一个名为 TSC_MODE 的新配置选项。由于操作系统以及其上正在运行的应用程序执行 RDTSC 指令的频率很高，即每个处理器的总和超过大约 10,000 RDTSC 指令，本章称之为 “高 TSC 频率” 操作系统/应用程序环境。在这种环境下，如果 TSC 的频率突然变化，操作系统或者应用程序使用 RDTSC 时则有可能会运行不正确，发生 “时间倒退” 现象。此时，我们称这种操作系统或应用程序为 “TSC 敏感” 的。为了确保程序执行的稳定性和可信性，除特殊情况以外，一般情况下都必须默认任何使用 RDTSC 的程序都是 “TSC 敏感” 的，这也是 Xen 4.0 及之后版本的默认设置。

Xen 在新版本中的 TSC_MODE 参数决定了 RDTSC 指令系列是否模拟执行。大致来说，原生指令执行意味着 RDTSC 执行会很快，但 TSC 敏感的应用程序可能在不可预测的情况下运行不正确；模拟 RDTSC 意味着会出现一些性能下降 (在大多数情况下是不可察觉的)，但由于有了软件模拟的保障，TSC 敏感的应用程序将始终正常运行。在 Xen 4.6.0 中，所有的 RDTSC 指令执行默认使用是原生的 TSC_MODE，虽然快速但可能不正确。在本方案中，为了尽量保证对租户透明，默认该设置处于自动配置的状态，并且 VMM 在适当时候，会使用硬件辅助的手段来模拟和截获虚拟机的 RDTSC 指令。

7.2.1.2 RDTSC 截获

通过分析 Xen 源代码发现，由于 TSC 读数更加精确和稳定，尽管 Xen Hypervisor 为其虚拟机提供了多个时钟源，其上的 Linux 虚拟机仍然默认使用 TSC 作为时钟源。

目前，AMD 虚拟化 (简称 AMD-V) 和英特尔虚拟化 (VT-X) 均提供 TSC 仿真。更具体地说，它们提供 RDTSC 和 RDTSCP 指令的条件捕获，这些指令以任何方式支持完全虚拟化。此外，它们允许软件控制在虚拟机执行期间读取的 TSC 的值，并且在虚拟机控制结构 (VMCS) 中规定额外的 offset field [40]。

根据现有 CPU 提供的上述特性，本章通过设置 VMCS 中的 RDTSC EXITING 位来截取并模拟 Guest-VM 的 RDTSC 指令。为了方便用户启动和终止 RDTSC 截取，本章修改了 Xen 4.6.0 的源代码，添加了超级调用 (Hypercall) 以供特权域开启和关闭

表 7.1. 封装的 Hypercall

Hypercall	描述
int do_enable_intercepting_rdtsc(void)	设置 RDTSC_EXITING 为 1，开启 RDTSC 模拟
int do_disable_intercepting_rdtsc(void)	设置 RDTSC_EXITING 为 0，关闭 RDTSC 模拟

RDTSC 的截获[1]。出于安全考虑，包括 VMCALL 指令的调用部分也已经被封装为系统调用，并添加到租户内核中。

开启和关闭的 Hypercall 细节如表 7.1。超级调用的作用主要在于启停 Hypervisor 对客户虚拟机的 RDTSC 模拟，进而进行系统的启停。

7.2.1.3 VMFUNC 指令调用

VM-functions(VMFUNC) 是由处理器提供的一个操作,其特殊之处在于它被 VMX 的 non-root 调用的同时不会产生 VM exit。VM functions 可以通过 VMCS 字段的设置来开启和配置。在 VMX 中 non-root 的操作使用 VMFUNC 指令来调用一个 VM function；EAX 寄存器的值选择被调用的特定 VM function。

举例来说，软件如果希望启用 EPTP switching(VM function 0)，它必须设置 “activate secondary controls” 这一 VM-execution 控制字段 (primary processor-based VM-execution 控制字段的第 31 位)，“enable VM functions” 这一 VM-execution 控制字段 (secondary processor-based VM-execution 控制字段的第 13 位) 和 “EPTP switching” 这一 VM-execution 控制字段 (VM-function 控制字段的第 0 位)，否则都将产生 #UD 异常。

此时，由 VM-functions control 字段的控制位来开启或关闭相应的 VM 功能，每一个控制位对应一个 VM 功能。例如，VM-functions control 字段的 bit 0 为 1 时，表明允许调用 0 号 VM 功能。不过当前的 Intel 处理器均只实现了 0 号 VM 功能 “EPTP switching”(EPTP 切换)，这个功能支持在 VMX non-root operation 模式内完成 EPTP（EPT 指针，EPT pointer）的切换。软件被限制从被 VMX 根模式下的软件预先设置好的 EPTP 值中做出选择。使用前，VMM 设置 VM-funcitons control 字段的 bit 0 为 1。示例代码如图 7.1。

EAX 寄存器提供 VM 功能号 0, 表明调用 “EPTP switching” 功能。EAX 寄存器的值记为执行 VMFUNC 指令由于这个 TV 值可能产生下面的情况。

1. 如果 $TV > 63$, 则产生 #UD 异常。
2. 如果 VM-fimctions control 字段的 bit/V 为 0，则产生 VM exit。

[1]Hypercall 是一种从 VM 到 Hypervisor 的软件 Trap。只要执行了 Hypercall，虚拟机就可以使用预先指定的方式 (例如，VMCALL) 请求特权操作。

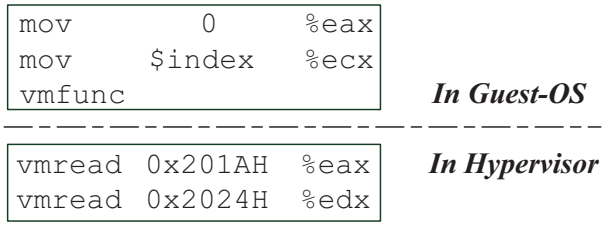


图 7.1 VMFUNC 代码示例

在 EPTP switching 功能中,ECX 寄存器提供的是 EPTP-list 表项的索引值。EPTP switching 功能允许在 VMX non-root operation 模式内切换 EPTP 值，从而可以使用不同 EPT 页表结构进行 Guest-physical address 的转换。在执行 VMFUNC 指令前，VMM 必须提前准备好 EPTP-list 列表。EPTP-list 列表的 64 位物理地址提供在 EPTP-list address 字段中，EPTP 表地址指向这个 4KB 的空间 (关于 EPTP-list 结构参见图 2.2,由于只有 4kB 的 EPTP 表只包含 512 个 8Byte 的值,所有如果 $ECX > 511$ ，VMFUNC 就会触发一个 VM exit)。如果被选择的值是一个有效的 EPTP 值 (不会使得 VM entry 失败)，它就会被储存在现有 VMCS 的 EPTP 字段中，会在稍后用到物理地址的时候用到。

一个 EPTP-switching VMFUNC 指令执行完毕后,处理器接着执行下一条指令。此时，EPTP 已经被切换，处理器将使用新的 EPT 页表结构转换 Guest-physical address。在这一时刻，原有层级 Guest paging structure 的 “顶层指针” 不变。也就是说，原有的 Guest paging structure 保持不变，Guest-linear address 仍旧通过原来的 Guest paging structure 进行转换。此时 VMM 需要特别注意的是：执行 EPTP switching 操作本身并不会产生 EPT violation 或者 EPT misconfiguration 故障，但是在下一条指令的 fetch 过程中，可能会由于使用新的 EPT 页表结构，而产生 EPT violation 或者 EPT misconfiguration，从而导致指令 fetch 失败。发生这种情况时，VMM 需要做出相应的处理。

7.2.2 基于按需时间模糊的侧信道防护系统总体设计

如7.1节所述，攻击者通常需要利用 TSC 去获取细粒度的时间信息，从而去推测受害者虚拟机的敏感时间信息。因此修改 TSC 即本方案的首要任务。之前有许多方案实现了这一目标 [77,83]。然而，现有的这种方法不能满足前文中对 ‘恰到好处’ 的和 ‘按需’ 保护的要求。针对此需求，基于按需时间模糊的侧信道防护方案应运而生。

本方案允许租户将在虚拟平台上的所有租户所观察到的 TSC 的保真度动态地调整到指定的级别，具体来说即模糊 TSC 的后 n 位。n 可以由受害者虚拟机要执行的敏感操作的性质以及攻击者虚拟机可用的已知漏洞来确定，以便仅删除恰到好处的 n 位来禁止这些攻击。在执行其操作之后，受害者虚拟机可以收回其请求以还原 TSC(即

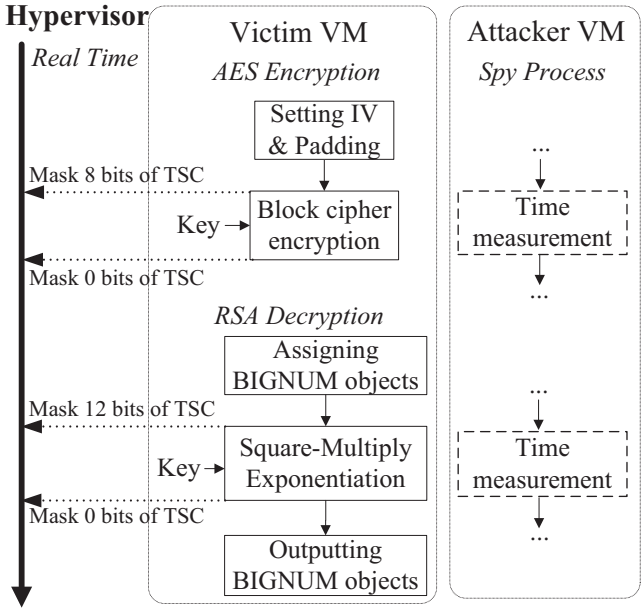


图 7.2 基于按需时间模糊的侧信道防护方案

通过请求 $n = 0$)。

图7.2展示了本方案的总体设计。其中，一个潜在的受害者虚拟机通过发送按需的和 '恰到好处' 的请求来请求 Hypervisor 模糊所有虚拟机的 TSC 值。在本章给出的例子中，受害者虚拟机在进行 AES 和 RSA 操作时，攻击者虚拟机正在进行某种时间侧信道攻击 (即不停地测量访问某个共享资源操作的时间)。为了以最小的开销来防止这样的攻击，受害者虚拟机仅仅在需要保护的代码执行前后发送降低 TSC 保真度的请求。例如，受害者虚拟机可以在专门进行加解密的函数执行时发送这样的请求。另外，TSC 的模糊程度可以被动态地设置，以便于实现 '恰到好处' 的保护。在本例中，对于 AES 进行设置 $n = 8$ 的操作，对于 RSA 进行设置 $n = 12$ 的操作。

尽管在本方案中 TSC 的模拟可能相对简单 (即通过在 VMCS 中设置 `RDTSC_EXITING` 位来对虚拟机的 TSC 访问进行截获，详见7.2.1.1节)，但有两个设计细节仍然需要注意。第一个是当有多个线程在多个虚拟机中同时发送请求或者在同一段较短时间内发送请求时 (如发生争用情况时)，如何确定 n 值。在这种情况下，n 值必须为当前所有请求值中的最大值。因为只有这样才能尽可能地保证攻击者获取不到任何有效的信息。同时，当争用中的多个请求中的某一个请求结束而其他的并未结束时，不至于发生 $n = 0$ 的情况。这样以来，本方案即采用了一种保守的加强型侧信道防御方案，因为当某些时候受害者虚拟机不需要 n 很大时，本方案依然有可能将最大的 n 值作为需要模糊的 TSC 的位数来执行侧信道防御[2]。此时每个虚拟机内核都应该记录来自其线程的所有请求，并且 Hypervisor 应记录来自虚拟机的所有请求。

[2]当目前请求 n 的线程 (虚拟机) 完成敏感操作时，依旧要将 n 调整为之后的最大值 (即之前第二大的值)。

第二个需要解决的问题是如何实现按需保护，且不引起实质性的开销。具体来说，需要允许虚拟机动态请求并以较少的成本更改 n 的值。云平台上的租户可以通过 VMCALL 或是超级调用进行这些请求，但这样做会导致客户机 VM 遇到 VM-exit，因此，当虚拟机经常更改 n 时，会导致过高的开销。

在这里，本章提出利用虚拟机功能 (VMFUNC)，这是 Intel 微架构指令集的一个功能，本章用来减少开销。VMFUNC 允许虚拟机在没有 VM exit 的情况下使用 Hypervisor 的功能。本章进行了一个简单的实验来比较 VMCALL/Hypercall 接口和 VMFUNC 调用接口之间的开销，并发现它们在 i7-4790 CPU 上分别花费 1622 和 160 个 CPU 周期。因此，如果可以使用 VMFUNC 接口，则可以节省大量成本。

虽然 VMFUNC 被设计为具有多达 64 个不同功能的通用目的，但目前的处理器仅实现了其中一个，即 EPTP switching，它能使 VM 灵活地切换其扩展页表 (EPT)。Guest-VM 通过将相应的索引放入 `ecx` 中来指定 EPT 指针，然后从用户模式或内核模式执行 VMFUNC 指令。这时执行会陷入 Hypervisor，而不会引发任何 VM-exit，之后该功能将 EPT 指针切换到 `ecx` 指定的指针，然后返回到 VM，且不引发任何 VM Entry。

这也就是说，当 `ecx` 寄存器包含这些特定值之一时 (在本原型系统中为 0 到 k)，可以使用它来设置 n 的值，并用于设置 TSC 的模糊程度。这种设计在实践中是不可行的，因为它会中断 EPT 切换，从而影响 EPTP switching 的正常使用。然而，只要 Intel 能够扩展，支持本章所提出的利用 VMFUNC 来降低 TSC 保真度的机制，那么本章的原型系统就可以用于评估与通过 VMFUNC 接口动态调整 TSC 保真度相关的开销。

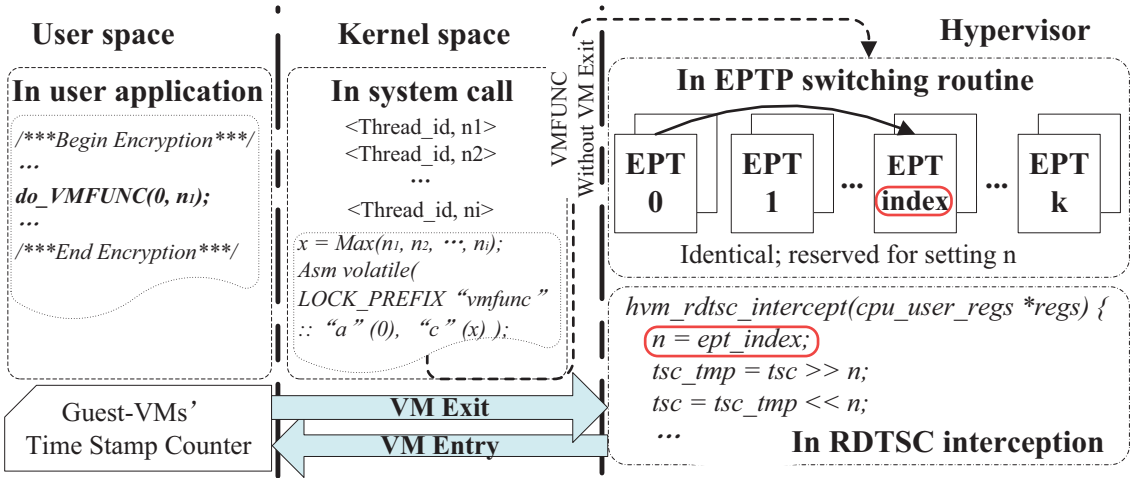


图 7.3 原型系统设计

图7.3显示了本系统原型的设计。其中 `do_VMFUNC` 是专门为客户虚拟机所定制的能够记录当前所有 VMFUNC 请求的系统调用。首先，创建多个相同的 EPT 页表，记

为 EPT 0 ẼPT k。用户空间进程此时可以用过定制好的系统调用来请求动态地模糊 TSC 的后 n 位。所定制的系统调用 *do_VMFUNC* 由客户虚拟机内核处理，它负责跟踪来自不同线程的不同请求，并使用 `ecx` $\in [0\ k]$ 原语来执行 VMFUNC 指令。在 Hypervisor 中，切换 EPT 表对客户虚拟机执行没有影响 (由于用于切换 EPT 表与原表相同)。当任何 `rdtsc` 指令执行时，Hypervisor 通过从 EPT 基地址计算出当前 EPT 指针与原 EPT 指针之间的偏移来重建 n 的设置 (即重构最近的 VMFUNC 调用中的 `ecx` 的值)，并模糊后 n 位 TSC 的值。这样一来，就创建了一个从虚拟机到 Hypervisor 的有效通信渠道，以便按需要求修改 n。

7.3 实验与分析

在本节中，本文对所提方案进行了评估。首先引入了在第3章中提出的两个代表性的隐蔽通道攻击，并展示了如何相应地设置不同的 n 来阻止它们。然后，本文通过展示当虚拟机上运行各种负载时的效果，以说明将 n 设置为各种值并以不同粒度修改 n 时本方案所带来的影响。

7.3.1 抵御时间信道攻击

在本小节中，评估了本原型在防御真实侧信道攻击方面的有效性。具体来说，本文应该如何动态配置 n 才能掩盖攻击者所能得到的精确的时间，并提供 “恰到好处” 的保护，让攻击失效。

本文选择了前文实现的两个代表性的隐蔽信道攻击，即基于 LLC 的攻击和基于内存总线争用的攻击。专门选择这两个攻击是因为它们在实施时通常需要以不同的粒度对时间进行测量。另外，如果可以证明本文提出的防御方案能够防止这些发送者与接收者协同合作，即利用隐蔽信道来传递信息，那么这同样可以保证当类似的侧信道攻击 (发送者在其中不知情地与接收者配合) 发生时，本方案可以对此进行一定程度的缓解。

7.3.1.1 跨虚拟机时间信道攻击

在此本节简单回顾一下前文所提到的本文实现的两种隐蔽信道。

在 LLC 攻击中，接收者进程：(1) 用其自己的代码或数据填充一个或多个缓存集；(2) 等待发送者进程以利用相同的缓存集；(3) 测量再次加载该代码或数据的时间。这遵循典型的 PRIME + PROBE 技术，其中发送者进程通过利用 (或不利用) 相同的 cache 来发送一些信息，这导致在接收者的最后一个 PROBE 操作中所花费的时间不同。本文遵循前文实现的基于 LLC 的侧信道攻击，使用由 i7-4790 CPU 上的所有四个 Cache Slice 的高速缓存行组成的驱逐集，对缓存集进行探测。

内存总线竞争攻击的工作方式类似，但与基于 LLC 的攻击相比，它的时间测试通

常更粗糙一些。发送者选择性地执行触发总线锁定行为的异常原子存储操作，这将导致接收器更长的访问时间，并因此有效地创建隐蔽通道。将接收者配置为使用最新的 Streaming SIMD 扩展 (SSE) 指令来绕过 Cache，直接访问内存总线，从而减少噪声 (因为不这么做的话很可能会掩盖总线锁定效果)。

为了尽可能多的揭示降低 TSC 保真度对隐蔽信道传输率的影响，在这两种攻击中，除了处理虚拟机调度和提供传输同步的必要编码机制，本文去除了以前的工作中 (详见第3章) 实现信道纠错或其他优化措施。

7.3.1.2 降低 TSC 保真度对攻击效果的影响

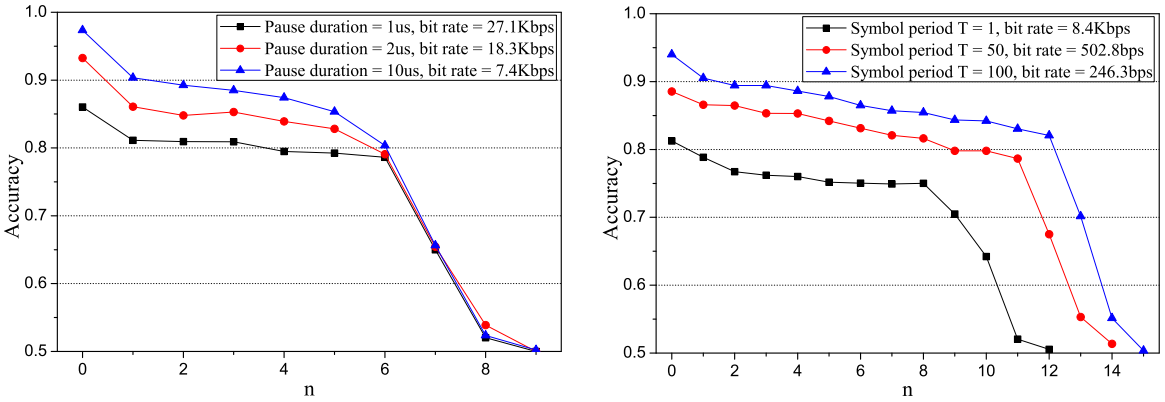
本文首先在未修改的 Xen 系统上采用一些实际参数执行基于 LLC 和基于内存总线争用的侧信道攻击，并观察接收过程中接收到的信息的准确性。然后开启保护并修改 n 的值，观察对所观测精度的相应影响。本文通过在两种不同类型的隐蔽通道攻击之间进行粗略的比较，以证明本方案在处理不同类型的威胁方面的灵活性。

为了获得一些实际的攻击设置，本文为两种攻击配置了各种发送比特率。由于它们使用了不同的编码方案 (LLC 攻击使用 RZ 编码，而内存总线竞争攻击使用曼彻斯特编码)，所以发送比特率按下文所述间接配置。对于 LLC 攻击，配置三个不同的暂停持续时间 (两个连续比特之间的等待时间) 为 1 微秒，2 微秒和 10 微秒，这导致发送比特率在 27 Kbps 和 7 Kbps 之间。对于存储器总线争用攻击，设置标记周期 T(发送每比特信息时被重复的连续异常原子操作的数量) 为 1，50 和 100，从而使发送比特率介于 8 Kbps 和 246 bps 之间。

在各种发送比特率下，本文测量了信号接收的准确率。假设一个强大攻击者能够通过执行实验来找出最佳阈值 (详见3.3.4.2节)，即攻击者的接收进程能够推断发送方是否执行了在隐蔽通道上发送 1 或 0 信号的操作 (将数据加载到 LLC 攻击的缓存集中或执行用于存储器总线争用攻击的异常原子存储操作)，从而调整时间阈值；同时，进一步假设当从 TSC 读数 (n) 中删除不同数量的比特位时，攻击者可以执行阈值的重新校准。在如此强假设情况下，本文对所提方案进行了实验。图 7.4显示了实验结果。

图7.4清楚地表明，当 n 增加时隐蔽通道攻击的准确性降低。有趣的是，当 n 大于一定数量时准确度会下降更多，而且准确度会快速接近 50%(即最低值，与随机猜测相同的准确度)。

通过比较两种类型的攻击效果可注意到，本方案在降低准确性方面可以使用较小的 n 为 LLC 攻击获得相同的保护。这是因为：LLC 攻击中需要更精细的时间测试，因为 LLC 的操作速度比物理内存快得多。这同样也印证了本文的观点 — 不同的受害者程序和不同的攻击需要 TSC 保真度的不同程度的降级，并且需要动态地设置 n 的值。



(a) 基于 LLC 的侧信道传输率　　(b) 基于内存总线的侧信道传输率

图 7.4 降低 TSC 保真度的效果

7.3.2 性能评估

既然已经展示了本方案在防范隐蔽通道时序攻击方面的有效性，现在关注开销评估。更具体地说，当潜在的受害者虚拟机动态地向 Hypervisor 发送更改 n 值的请求时，希望看到性能开销方面的影响。因此，本文希望能够揭示本方案的推荐用法，以便在攻击保护和性能开销之间取得平衡。在性能评估中，本章考虑三种不同的虚拟机工作负载，它们在对抗侧信道攻击需要保护的指令的百分比上有所不同。

实验平台为编译安装了修改过的 Xen 4.6.0 的 Dell XPS 8700-R39N8 台式机，其 CPU 为 Intel Core i7-4790 3.6GHz * 8。用于实验的 Web 服务的服务端为在 Xen 上开启的虚拟机，其内核为 Linux 3.14.60，分配 1 个 vCPUHE 2048M 内存；而客户端则运行在具有相同硬件配置和 Ubuntu 16.04 LTS 的另一台物理机的主机操作系统上。

性能开销评估中的一个微妙之处在于测量 TSC 保真度降级时的精确的时间开销。本方案的保护机制使得任何虚拟机无法进行精确的时间测量，但本文 (为了进行性能开销评估) 仍然希望能够获得最细粒度的 TSC 读数。为此，在 Xen 中引入一个自定义的超级调用，在调用它时总是返回精确的 TSC 读数。在本节其余部分的所有实验中，都使用此自定义的超级调用来衡量虚拟机产生的时间开销。同时，本文修改了 PARSEC [131] 中有关计时的部分实现 [132]，使得在有关 PARSEC 作为标准性能基准时，其计时系统不会受到本文所实施的方案而改变。

WL-1 – Encryption. 在这部分性能开销评估中，本章着眼于受害者虚拟机上的合法工作负载，而非时序攻击。目的是找出受害者 VM 在请求时序攻击保护时产生的性能开销有多少。在这里考虑的第一个工作负载 (WL-1) 包括两个加密工作负载，一个是让受害虚拟机执行 AES 加密，另一个是让受害者虚拟机执行 RSA 加密，二者都实施易受跨虚拟机时序攻击的操作。在这两种情况下，受害者虚拟机通过将 VMFUNC

指令插入到其加密库 (`libcrypto.so` in OpenSSL 1.0.2g) 来修改 n 以尝试保护其密钥免受同驻时间侧信道攻击的危害。

当测试 OpenSSL 源代码以插入 VMFUNC 指令时，本文专注于保护关键的密钥部分 (也就是在加解密中与密钥相关的部分)。图 7.5显示了本文的插桩方案。对于 AES，在加解密的第一轮 (开启 TSC 模糊) 之前和每个块加密 (关闭 TSC 模糊) 的最后一轮之后插入 VMFUNC 指令，即实际上为每个 1 KB 文本加密时，再额外执行的 32 对 VMFUNC 指令。对于 RSA，在函数 `bn_mod_mul_montgomery()` 中插入一对 VMFUNC 指令，该函数执行蒙哥马利模乘法并且通常是时间侧信道攻击的目标。这两个工作负载上的插桩之间有一个明显的区别，那就是本方案在保护 (一对 VMFUNC 指令之间的代码) 涵盖了 AES 运行时间的 90%，但只有 RSA 运行时间的 30%-40%。

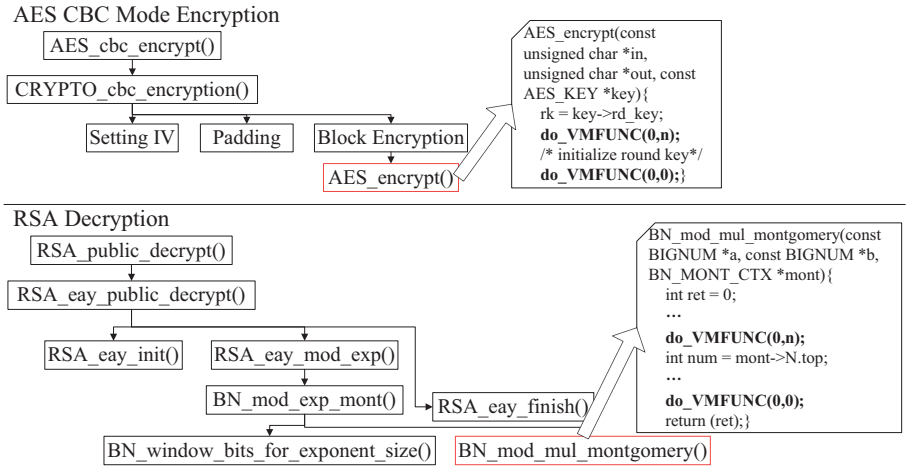


图 7.5 AES 和 RSA 在密码库中的插桩

图 7.6展示了在未修改的 Xen 上运行的虚拟机上未测量的 OpenSSL 使用基准线的标准化运行时间开销。在这里，本章尝试比较提供时序攻击保护的两种不同策略：S1 伴随着在虚拟机的生命周期 (移除了 8、16 和 24 位) 里 TSC 保真度不断恶化，在使用寿命范围内受害者虚拟机和虚拟机管理程序之间不需要通信；S2 具有按需保护，其中 VMFUNC 指令用信号通知管理程序来调整 TSC 保真度。直观地说，S2 支持按需的和 “恰到好处” 的保护，但是由于 VMFUNC 指令，特别是在密码操作中被频繁调用时，会产生额外的开销。

而令人惊讶的是，测试结果表明，S1 和 S2 之间的差异不仅很小，而且有时 S2 实际上胜过 S1 ，例如在 16 或 24 位被屏蔽时的 RSA。这可能是由于与 AES(约 90%) 相比，RSA 中密钥相关指令的部分较小 (30% 至 40%)。当密钥相关指令构成一个小百分比时，由 S1 中与密钥无关的指令产生的不必要保护的开销可能超过由 VMFUNC 指令产生的，这导致 S2 产生较低总体开销。这个结果表明，方案在提供按需和 “恰到好处” 的安全性方面的额外收益不一定会带来更高的开销。换句话说，我们可以在某些

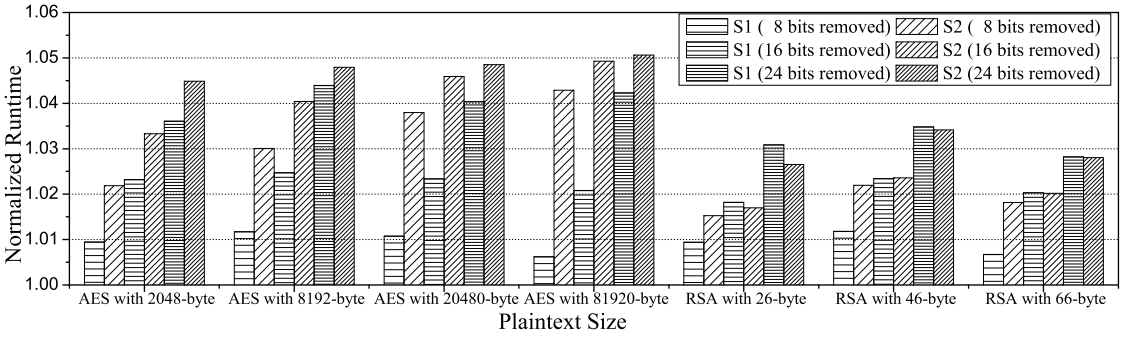


图 7.6 AES/RSA 运行时负载

VM 工作负载下同时获得更好的安全性和更低的开销。

WL-2 – 运行 PHP 的 HTTPS 服务器. 在之前对于负载 WL-1 的实验中发现，S2 (通过频繁的从受害者虚拟机到虚拟机管理程序的 VMFUNC 指令，实现提出的按需和 “恰到好处” 的保护) 可能会在特定的场景下的效果优于 S1 (永远保护整个虚拟机的生命周期，而不需要 Hypervisor 进行通信)，在这些场景中密钥相关指令在虚拟机所有执行指令中占比较小。本小节中主要关注密钥相关指令占比更小的实际场景，即在虚拟机上运行一个使用 PHP 解析 HTTPS 的 Web 服务器。在这个案例中，其工作负载主要包括加密 (密钥相关指令的一部分，如 WL-1 所示)、web 服务 (密钥无关) 和网络通信 (密钥无关)。

实验运行在一个隔离的 1 Gbps 局域网上。受害者虚拟机使用 HTTPS 和 PHP 运行 Apache Web 服务器，并且在另一个主机 (不在虚拟机中) 运行的客户端执行 Web 服务器负载测试软件 Siege(详见 `https://www.joedog.org/siege-home/`)。Siege 能够向 Web 服务器发送持续的 HTTPS 请求（两个连续的 HTTPS 请求之间无延迟），持续 10 分钟。单个并发请求被用来测量响应时间，而最大数量的并发请求 (可以测试出最大吞吐量) 被用作测量最大网络 I/O 性能的输入。

Apache Web 服务器配置为使用 RSA_WITH_AES_256_CBC_SHA 进行密钥交换和协商，使用 AES256 CBC 进行数据加密，这两者都与密钥相关并易受时间信道威胁，需要保护。为了展开对比，配置了四种不同的 Apache Web 服务器。服务器 A 是没有插入任何 VMFUNC 指令的未修改版本，而服务器 B、C、D 被用来以从细到粗的粒度插入 VMFUNC 指令，以保护与密钥相关的部分；如图7.7。

服务器 B 具有最细粒度的 VMFUNC 指令插桩，插入了如 WL-1 中描述的加密算法的 VMFUNC 指令对。服务器 C 和服务器 D 插入的 VMFUNC 指令较少，覆盖了大部分包括 Web 服务器的其他与密钥无关的操作。表7.2说明了在 HTTPS 响应的准备和发送时执行的 VMFUNC 指令的数量。如预期，例如在服务器 B 中，更精细粒度的指令插入将会导致更多的 VMFUNC 指令被调用。

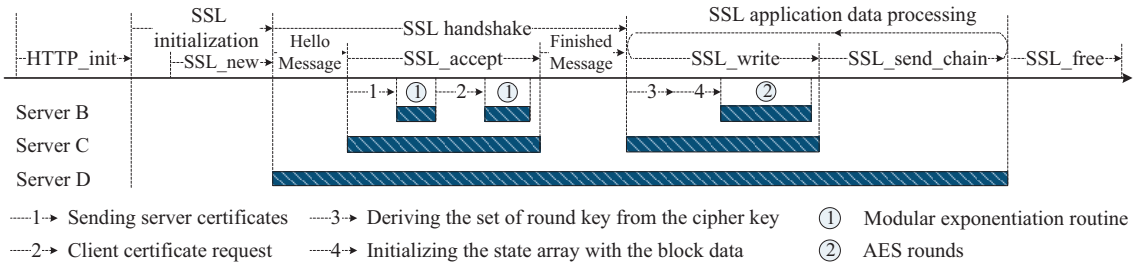
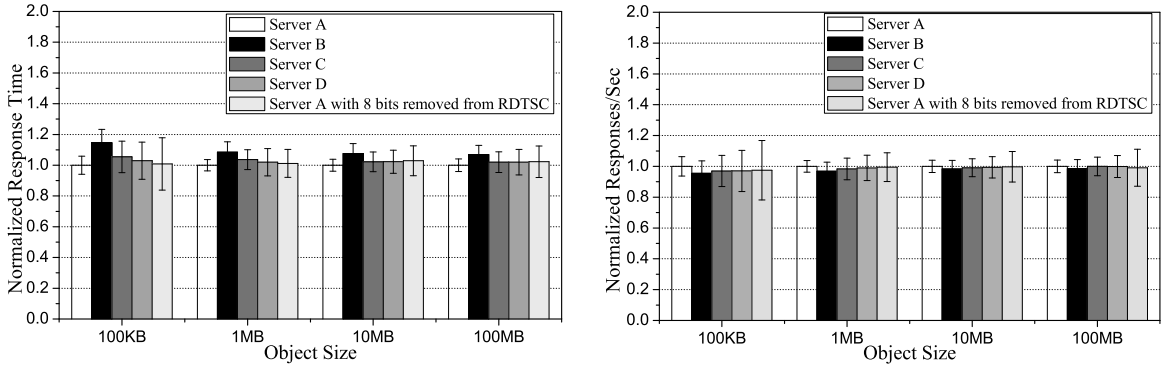


图 7.7 对 HTTPS webserver 不同程度的插桩

表 7.2. 单个 HTTPS 响应中 VMFUNC 指令执行数量

	Server			
File Size	A	B	C	D
100KB	0	6576	142	2
1MB	0	65596	1402	2
10MB	0	655520	13986	2
100MB	0	6554006	139824	2

图7.8展示了对于五种不同配置的 WL-2 的归一化响应时间和归一化吞吐量：无保护的服务器 A(归一化响应时间和吞吐量的基准值)，服务器 B、C、D 和长时间保持 TSC 模糊的服务器 A。此处本章只显示移除后 8 位 TSC 读数时的结果。本章对于移除 12 位时执行了同样的实验，结果与之前非常相似，只因 TSC 模糊粒度增大导致实验结果的标准差较大，故在此不表。



(a) 响应时间　　(b) 吞吐率

图 7.8 WL-2 性能开销

本文首先比较五种不同配置的 Web 服务器的性能。没有任何对时间进行保护的服务器 A 显然给出了最佳的吞吐量结果。值得注意的是，ServerC(S2 的一个实例) 一如既往地优于具有恒定的保护的服务器 A(S1)，而对于较大的文件，优势更为显著。其原因与 WL-1 中的情况相似：本方案对于密钥相关指令上的不必要保护的优化空间大

于调用的额外 VMFUNC 指令所需的开销。服务器 B 的吞吐量最小，主要是因为其 VMFUNC 指令粒度最小。

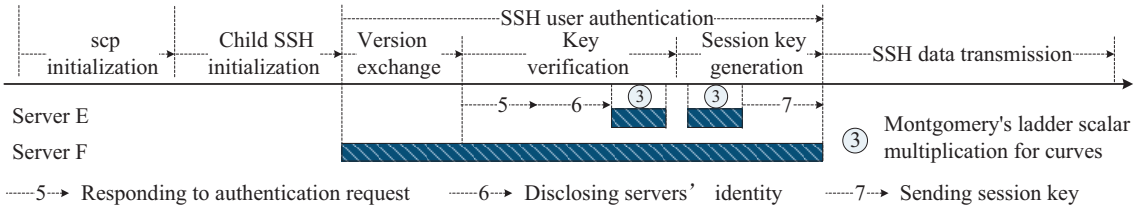


图 7.9 对 SCP server 不同程度的插桩

WL-3 – SCP 服务器. 实验 WL-1 和 WL-2 展示了在特定配置中，“按需” 和 “恰到好处” 的保护 (S2) 优于持续保护 (S1) 的情况。现本节研究第三种工作负载 — SCP 服务器，其密钥相关操作占安全文件复制和发送操作等整个工作负载的一小部分 (WL-3)。在 WL-3 中，只有的最开始的密钥交换与协商部分涉及认证密钥等敏感操作，对此要提供侧信道防护 (且 OPENSSL 中的 ECDSA 计算易受基于时间的侧信道攻击 [16,133])，而其余部分与密钥无关，因此不需要防护。

服务器 E 和 F 是经过 VMFUNC 插桩后重新编译的 SCP 服务器，如图7.9所示，服务器 E 具有更细粒度的指令和更多的 VMFUNC 指令。在服务器 E 中执行的 12 个 VMFUNC 指令用于保护 SCP 传输，服务器 F 中的两个 VMFUNC 指令用于保护 SCP 的安全传输。

服务器的传输速率如图7.10所示 (仅显示 TSC 模糊的低 8 位的结果，因为其结果屏蔽低 12 位的数据非常相似)。再次观察到，S2 (服务器 E 和 F) 的性能优于 S1 (具有整个 VM 生命周期中的定时保护的服务器 A)，而且在 S2 中的表现优于 S1 时，这两者的差异更显著。这三种不同的工作负载所展现的一致性结果清楚地表明：在许多实际的工作负载中，从虚拟机到 VMM 的频繁通信 (其他同类方案) 的好处可能会超

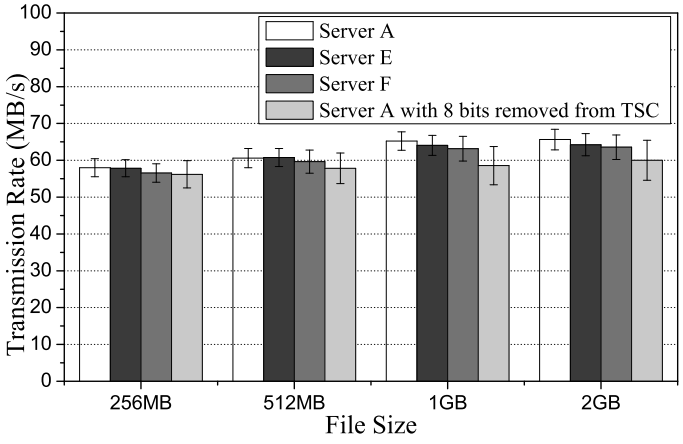
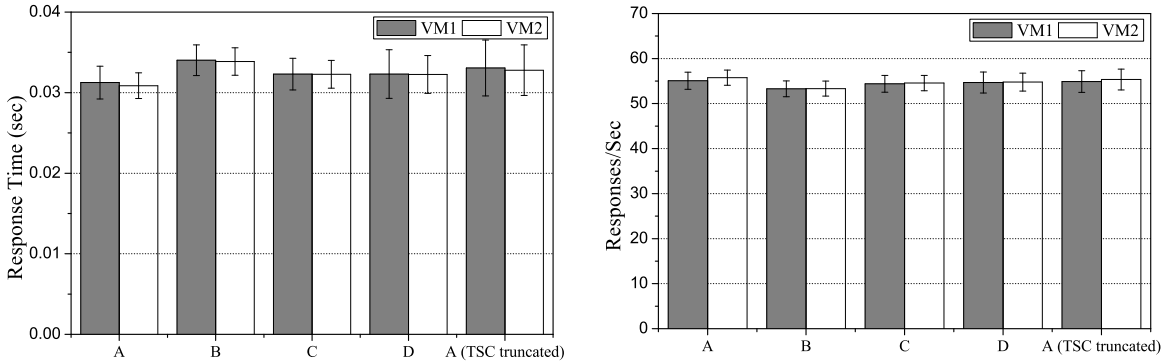


图 7.10 WL-3 网络性能开销

过纯粹 VMFUNC 指令带来的开销。因此，本文提出的“按需”和“恰到好处”的模糊 TSC 读数的机制不仅提供了更好的安全性，而且在许多情况下也会降低开销。

无须保护的其他同驻虚拟机的开销. 本节测量单个虚拟机在使用 TSC 模糊时对其他同驻的潜在受害者虚拟机的影响。这些受害者虚拟机不受任何时间侧信道攻击，但由于需要与更高安全性的虚拟机同驻 (如实施本方案的受害者虚拟机)，因此可能会产生一些额外性能开销。



(a) 响应时间　　(b) 吞吐率

图 7.11 与无须保护的虚拟机同驻时性能开销

图 7.11显示当 VM1 请求 $n = 8$ TSC 的低位被屏蔽，而 VM2 不会发出这样的请求时，各个虚拟机的响应时间和吞吐量。与在 WL-2 中一样，这两个虚拟机正在运行 Apache 服务器，并连接到 Siege 客户端。这些结果表明，VM2 的性能开销很小。服务器 C 及其同驻的其它普通虚拟机产生的开销最小，这同样与 WL-2 中获得的结果一致。

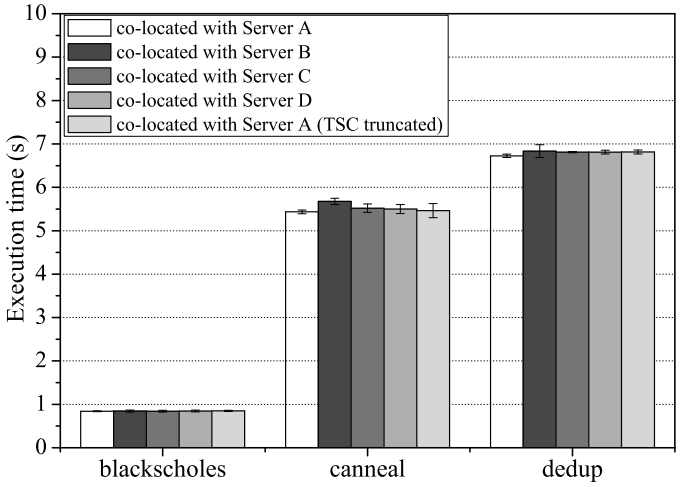


图 7.12 同驻情况下 PARSEC 应用的性能开销

本节的最后一个实验为测量本方案对 PARSEC benchmark 的影响,使用运行 PAR-

SEC 基准测试的虚拟机，并与另一台运行服务器 A，B，C 和 D 的虚拟机同驻，测试同驻虚拟机对 PARSEC 程序的影响。图 7.12表明，一般情况下，性能影响是很小的。不过，canneal 似乎更易受 TSC 模糊的影响。本文认为这是由于canneal 大多是内存密集型程序，大量的 VMFUNC 指令 (在服务器 B 的情况下) 可能导致另一个虚拟机上的更多缓存未命中。

7.4 本章小结

本章提出一种面向虚拟化系统全局的侧信道防护方法，利用硬件虚拟化扩展，允许虚拟机动态地请求物理平台上所有虚拟机的时间戳计数器 (TSC) 按需地模糊 (达到其要求的级别)，以防御基于 TSC 的时间侧信道攻击。同时通过新颖的方式利用 VMFUNC 接口，该技术允许虚拟机应用程序向管理程序发送按需请求，以 “恰到好处” 地程度来模糊 TSC 的最低 n 位，以阻止另一个同驻虚拟机的精确时间测量。实验展示了本方案对两个隐蔽信道的防御效能，从而得出在这些攻击场景中应该 TSC 置零位数量。对三种不同工作负载的实验表明，本方案具有比现有防御方法 (如 Düppel [35]) 更低的性能开销。

但是，该设计确实存在一些限制。首先，该设计依赖硬件支持，特别是通过 VMFUNC 指令来调用其功能。第二，任何模糊时间来源的防御方法 (本方案或其他方案 [77,83,134]) 都可能影响到某些时间敏感型操作。第三，本方案同样需要修改应用程序或用户库。虽然这些修改非常直接，包括在包含有敏感信息的操作前后插入 VMFUNC 指令，但这仍然需要修改者具备一定的相关知识和经验，即本方法依赖于应用程序开发人员来定位易受定时侧信道攻击的敏感代码部分。第四，攻击者虚拟机可能会要求大量的模糊 n 位 TSC 值的请求，这样会导致一定程度的拒绝服务 (DOS) 攻击。不过，如7.1节所述，本文认为可以通过制定相应的服务策略来遏制这些行为。最后，允许应用程序改变 n 本身可能会导致侧信道泄漏，因为攻击者通常可以根据其他情况来推断这些事件，甚至触发它们 (例如，通过向受害者虚拟机提交请求)。不过本文同样认为这种风险是极其小的，而且可以通过管理来避免。

虽然在本文聚焦于模糊时间戳计数器的方法来进行按需的侧信道防御，并且这种技术被应用到其它实时时钟平台上 (如移动设备) 也具有同样出色的效果。然而，应用程序仍然可以自己实现一个计时器线程，来对系统中的操作进行时间测量 [112]。例如，Shwarz 等人 [135] 和 Chen 等人 [136] 使用这种技术在 SGX enclave 内建立一个软件的时钟，而因此无需访问系统中固有的实时时钟，从而加强或防御侧信道攻击。按需模糊 “时钟” 如果能被应用于防御这类技术，则是未来工作中一个有趣的领域。不过，最近一种新的利用 TSX 硬件特性的 Prime + Abort 的侧信道攻击被提出 [137]，它不需要任何时间观测信息，因此能够完全抵御针对时间精度的防御方式。

另外,本文并不是第一个使用 VMFUNC 来实现虚拟机内的安全增强的。SeCage [138] 改进了 Intel 处理器中的 VMFUNC 和嵌套分页，以便以较低开销透明地为不同安全级别的用户提供不同的内存视图，从而防止因 rootkit 造成的私钥泄漏和内存扫描。而在本文中将 VMFUNC 作为虚拟机与 Hypervisor 之间的低开销接口，使它们之间的频繁通信更有效率，从而用于动态地调控时钟源。

8 总结与展望

8.1 总结

本文从跨虚拟机侧信道的攻击、检测和防御等多个方面对这一云环境中的新型威胁作了系统性的研究，其主要成果集中在以下几个方面。

跨虚拟机侧信道攻击与虚拟机同驻方面.

Maurice 等人在 2017 年 [34] 提出两点论点：首先，由于噪声特性，纠错码不能直接应用与隐蔽信道构建中；其次，即使系统活动干扰极大，也可以构建一个无差错和高吞吐量的隐蔽信道。该观点与本文不谋而合。因此本方案除了引入的纠错码，考虑到码元识别对于提高信道传输率至关重要，创新性地使用了基于码元识别的信道优化算法。本文首先优化了当前流行的两种跨虚拟机的侧信道攻击，并给出了一些通用的优化手段。然后通过实验验证了这些手段能够极大程度地提高现有侧信道的传输正确率。所提出并实现的这两种典型的攻击，均作为了本文之后所提的检测方案和防御方案的实验对象。

与此同时，针对云平台的同驻问题，本文提出了一种高效的适用于商用云平台的虚拟机同驻攻击方案。该方案能够使攻击者实现快速的恶意虚拟机与目标虚拟机的同驻，且尽可能地减少不必要的开销, 具有很大的现实意义。本文工作体现在以下几点：(1) 检测方法的高效性。本文采用基于隐蔽信道的虚拟机同驻检测机理, 提出并实现了一种具体的基于内存总线冲突的检测方案, 揭示了隐蔽信道传输正确率和同驻概率之间的关系并在阿里云云平台对其进行了实验验证，证明本文所提同驻检测方法高效易行、误检率极低，用时较少。(2) 虚拟机洪泛策略。本文以国内著名云平台服务提供商阿里云为研究案例，提出一种基于后验概率的自动化虚拟机洪泛方法，本文所提虚拟机同驻方案作为一种典型的针对云平台的恶意行为，亟需各大云服务提供商重视与防范, 且对其后的面向云平台同驻的攻击研究打下基础，对增强现有商用云平台的安全性有重要意义。

跨虚拟机侧信道信道检测和定位方面.

传统的隐蔽信道检测方案始终依赖于人为干预。为了避免繁琐的手动分析和状态空间爆炸，本文提出了自动化框架中不使用信息流分析的检测方法。这些程序可以在云平台上实时运行，几乎没有开销，可以透明地评估云环境。本文使用事件关联分析来检测云计算环境中的隐蔽信道。改进了原有的共享资源矩阵算法，并使用 XSRM 来

预处理事件日志和安全配置文件。它可以减少构建庞大的共享资源矩阵的负载，区分发生异常事件的目标设备，并提供重要的事件元数据进行分析，也可以防止管理员被大量虚假警报所淹没。此外，本解决方案部署在云平台上作为插件，这使得云更加可扩展，更易于管理。

同时，为了克服云环境跨虚拟机侧信道威胁定位方法匮乏、现有威胁定位技术不精确等挑战，本文设计了基于硬件特性和虚拟机自省的威胁定位方案来实时的定位云中存在的跨虚拟侧信道攻击，并辅助审计。通过结合诸多 CPU 硬件特性、虚拟化关键技术和安全分析技术，本文实现了该系统。

提出的云环境下跨虚拟机用户层侧信道攻击检测与定位方法易于实施，方便部署，能根据资源情况和检查粒度按需调度，并且不需要客户虚拟机的参与配合，具有良好的透明性。通过 PMU/PMC 收集到的攻击特征对典型侧信道进行检测，通过 LBR 信息并还原得到具体发生攻击时的场景。通过改写 Xentrace 实现对虚拟机状态信息的传递和区分，借助虚拟化环境下硬件信息辅助，减小了 VMI 分析内存规模的数量级，通过分析 VMA 结构体对 LBR 的语义进行充分理解，使得整个定位流程做到实时高效。在 Xen 上实现了原型系统，实验结果表明方案能有效的检测和定位前文所述两种侧信道攻击，同时引入的平均性能为 $O(N)$ 数量级。

跨虚拟机侧信道防御方面.

随着新的虚拟机侧信道的出现，研究人员提出了一些针对性强的方法。例如，强行将敏感操作的运行时间固定或返回固定时间的倍数是一种较好的方式。然而，这种方案引入了很多刻意的时间延迟，并且要求用户始终忍受这种性能损失。为了解决该问题，本文提出了一种侧信道防护方法，根据客户虚拟机所提出的要求，允许虚拟机动态地模糊时钟源。我们利用硬件虚拟化扩展技术，给出一种轻量而有效的方法，实现了云平台的整体侧信道防护。

具体来说，本文通过 Intel VT-X 提供的 RDTSC 指令截获功能，模拟了虚拟的时钟源并提供给用户；通过新颖的方式重载了 Intel CPU 中的 VMFUNC 指令接口，允许虚拟机上的应用程序向 Hypervisor 发送按需的请求，以阻止其他恶意的同驻虚拟机对时间进行精确测量。本方案同样具有通用行与极强的可用性。在可用性分析方面，我们通过以所实现的侧信道攻击为例，展示了本方案对多个隐蔽信道的防御效能。在性能分析方面，对三种不同工作负载的实验表明，本文方案在各种应用中最多造成不超过 5% 的额外开销。

8.2 未来工作

跨虚拟机侧信道作为一种无法避免的云环境安全威胁，它在可预见的未来都将会安全界被视作亟待解决的问题。同样，本文现有的工作亦不能完全地涵盖各方面，也

未能完美地解决所提问题。在此阐述与本文研究相关的未来方向，以求抛砖引玉，旨在促进跨虚拟机侧信道威胁的安全解决方案愈发成熟。

跨虚拟机侧信道攻击.

安全永远是动态的。正所谓 “道高一尺、魔高一丈”，现有流行的攻击终究会被某些更强的检测方法和防御方法阻拦。更为隐蔽、更具威胁的攻击手段一定会层出不穷。跨虚拟机侧信道攻击方面的研究在今后可以往一下方面发展：(1) 新的攻击技术，如利用新的硬件特性如 Intel Process Tracer 来构造隐蔽信道；(2) 新的攻击场景，如从传统 x86 平台转换到现流行的 ARM 平台上；(3) 新的攻击目标，如利用侧信道去逆向内核 ASLR 机制，或去深层次的获取用户的隐私并用于 APT 攻击，或去针对现有人工智能和机器学习算法，逆向其算法或窃取其中的秘密信息。

今后的工作中，将结合针对内存的 Rowhammer 攻击，将其与现有缓存侧信道结合，实现一种不需要攻击者主动锤击的跨虚拟机 Rowhammer 攻击。现阶段 Rowhammer 攻击已经能够允许未经授权的软件擅自修改 DRAM 单元中的特定位，从而实现强大的特权升级攻击。而现有的复杂的 Rowhammer 防御对策大多旨在缓解 Rowhammer 错误或其开发，却忽视了对这些防御措施的完整性没有足够的了解。Gruss 等人提出新型的 Rowhammer 攻击原语 [139]，打破了以前对触发 Rowhammer bug 的要求的假设，能够实现单边 DRAM 锤击。同时他们提出了操作码翻转技术，通过在用户空间二进制文件中以可预测和有针对性的方式翻转位来绕过最近的隔离机制。隐藏后的 Rowhammer enclave 可用于云中的拒绝服务攻击以及个人计算机上的特权升级。然而，这种方案所用到的 Memory chasing 方案不够灵活，所需时间较长。本文之后可在此方面做出改进，并在此基础上，提出基于内存 Rowhammer 漏洞特征的 PUF 方案。

虚拟机同驻方面.

本文所提方案在现阶段揭示了虚拟机分布情况与 IP 之间的关系：即虚拟机内网 IP 随着虚拟机的分配自增。在未来的工作中，本文将从其他规律出发，从根本上探究云平台所使用的虚拟机分布策略。当虚拟机发生防御性的迁移时，还可跟踪其迁入目的，利用云平台负载均衡漏洞，一并将攻击者虚拟机迁移至同目的主机，以实现 “牛皮糖” 似的同驻方案。

同时，在今后的工作中，可利用上述提到的 Rowhammer 与 Cache 侧信道结合的新型跨虚拟机内存锤击攻击方案，通过诱发受害者自行访问内存实行锤击，绕过现有方案的隔离措施 [140]，用于虚拟机同驻方案中，以实现更准确的定位。

跨虚拟机侧信道信道检测和定位方面.

本文提出的隐蔽信道检测框架在已知时间隐蔽信道的检测上有良好效果，基于事件日志和安全配置文件的分析也贴合了现今云计算环境和数据中心环境的特点。在以

后的工作中，我们将对现有方案进行改进，研究适用于云计算环境的未知时间隐蔽信道的识别和检测方法，并讨论其安全威胁解决或缓解方案。也可以加入检测预警和反馈机制，帮助进一步改善检测规则的制定。

现阶段，本文设计的基于硬件特性和虚拟机自省的威胁定位方案借助 LBR 记录精确的跳转信息。在今后的工作中，本文会寻找更为有用的硬件特性，来更好地为已有框架服务，提供更为精确的定位和上下文信息。本文在实现过程中通过修改 Xen 源代码解决了 LBR 的多核虚拟问题，在之后的研究中，将会探索类似硬件特性的虚拟化方法，并将之用于现有检测和定位框架中。

跨虚拟机侧信道防御方面.

本文所提基于时间动态模糊的防御方案针对时间这一环节来解决侧信道防御的问题，方法较为通用。因此，在未来的工作中，可引入高次曲线模型与差分隐私模型来模拟所返回的 TSC 读数，提高对时间的模糊程度，并在适当情况下引入物理上的访问延时，在不对系统性能造成过多影响的情况下，可防止攻击者对于时间的粗粒度猜测，使攻击者自身的时钟源同样失效。

如第1章所介绍，最近一种被称为 Controlled side channel 的侧信道攻击能够在不可信的内核中设置页面的权限，使任何对于该页面的访问都触发缺页异常，从而能推测出使用页面的进程的状态与数据 [141]。同样的，这种侧信道也可以用来针对 SSL / TLS 实施 [142]，称之为控制流推理攻击。Controlled side channel 的可怕之处在于它能够绕过许多已有的防御与保护机制 (如 Intel SGX)。不过值得庆幸的是，这些基于不可信内核的攻击，都必须依靠内存虚拟化技术来管理页面，因此在未来工作中，可利用硬件虚拟化 EPT 来对该攻击进行检测与防御。

参考文献

[1] 林闯, 苏文博, 孟坤, 刘渠, and 刘卫东. 云计算安全: 架构, 机制与模型评价. 计算机学报, 36(9):1765–1784, 2013.

[2] 中国信通院. 云计算关键行业应用报告. `http://http://www.chinacloud.cn/upload/17092108184894.pdf`, 2017.

[3] Moritz Lipp, Michael Schwarz, Daniel Gruss, and et al. Meltdown. Report, 2017.

[4] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[5] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation computer systems*, 28(3):583–592, 2012.

[6] Hai Jin, Guofu Xiang, Deqing Zou, Song Wu, Feng Zhao, Min Li, and Weide Zheng. A vmm-based intrusion prevention system in cloud computing environment. *The Journal of Supercomputing*, 66(3):1133–1151, 2013.

[7] Steven M Bellovin. Virtual machines, virtual security? *Communications of the ACM*, 49(10):104, 2006.

[8] Keisuke Okamura and Yoshihiro Oyama. Load-based covert channels between xen virtual machines. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 173–180. ACM, 2010.

[9] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *16th ACM Conf. Comp. and Comm. Sec.*, 2009.

[10] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM Conf. Comp. and Comm. Sec.*, 2012.

[11] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *21st ACM Conf. Comp. and Comm. Sec.*, 2014.

[12] Anthony Desnos, Éric Filiol, and Ivan Lefou. Detecting (and creating!) a hvm

rootkit (aka bluepill-like). *Journal in computer virology*, 7(1):23–49, 2011.

[13] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symp.*, pages 1–18, 2016.

[14] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symp.*, pages 19–35, 2016.

[15] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Dos attacks on your memory in cloud. In *12th ACM Asia Conf. Comp. and Comm. Sec.*, pages 253–265. ACM, 2017.

[16] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-VM attacks on Xen and VMware. 2014.

[17] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *17th Intern. Symp. Research in Attacks, Intrusions, and Defenses*, 2014.

[18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symp. Security and Privacy*, 2015.

[19] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *11th ACM Asia Conf. Comp. and Comm. Sec.*, pages 353–364. ACM, 2016.

[20] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[21] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *32th IEEE Symp. Security and Privacy*, 2011.

[22] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *22nd ACM Conf. Comp. and Comm. Sec.*, 2015.

[23] 王国峰, 刘川意, 潘鹤中, and 方滨兴. 云计算模式内部威胁综述. 计算机学报, 40(2):296–316, 2017.

[24] Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In *RSA Conf., Cryptographers' Track*. Springer, 2008.

[25] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *32nd IEEE Symp. Security and Privacy*, 2011.

[26] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symp.*, 2014.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symp.*, 2015.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symp. Security and Privacy*, 2013.

[29] Naomi Benger, Joop Van De Pol, Nigel P Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *16th Intern. Workshop on Cryptographic Hardware and Embedded Systems*, 2014.

[30] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R Gross. CAIN: Silently breaking ASLR in the cloud. In *9th USENIX Workshop on Offensive Technologies*, 2015.

[31] Ben Gras, Kaveh Razavi, E Bosman, H Bos, and C Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *ISOC Network and Distributed System Security Symp.*, 2017.

[32] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh R Joshi, Matti A Hiltunen, and Richard D Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *3rd ACM Cloud Computing Security Workshop*, pages 29–40, 2011.

[33] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Trans. Networking*, 23(2), 2015.

[34] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. *NDSS, San Diego, CA, US*, 2017.

[35] Yinqian Zhang and Michael K Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *20th ACM Conf. Comp. and Comm. Sec.*, 2013.

[36] Peter Pessl, Daniel Gruss, Clementine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. pages 565–581, 2016.

[37] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. Detecting co-residency with active traffic analysis techniques. In *4th ACM Cloud Computing Security Workshop*, 2012.

[38] Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium*, pages 929–944, 2015.

[39] 余思, 桂小林, 张学军, 林建财, and 王君飞. 云环境中基于 cache 共享的虚拟机同驻检测方法. *计算机研究与发展*, 50(12):2651–2660, 2013.

[40] Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[41] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *22nd Intern. Symp. High Performance Comp. Arch.*, 2016.

[42] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[43] Steven B Lipner. A comment on the confinement problem. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 192–196. ACM, 1975.

[44] J Thomas Haigh, Richard A Kemmerer, John McHugh, and William D Young. An experience using two covert channel analysis techniques on a real system design. *Software Engineering, IEEE Transactions on*, (2):157–168, 1987.

[45] Chii-Ren Tsai, Virgil D Gligor, and C Sekar Chandersekaran. On the identification of covert storage channels in secure systems. *Software Engineering, IEEE Transactions on*, 16(6):569–580, 1990.

[46] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[47] Richard A Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems (TOCS)*, 1(3):256–277, 1983.

[48] John McHugh. Covert channel analysis: a chapter of the handbook for the computer security certification of trusted systems. *Department of Computer Science, Portland State University, USA, http://chacs. nrl. navy. mil/publications/hand-*

book, 1995.

[49] SiHan Qing and ChangXiang Shen. Design of secure operating systems with high security levels. *Science in China Series F: Information Sciences*, 50(3):399–418, 2007.

[50] Jingzheng Wu, Liping Ding, Yanjun Wu, Nasro Min-Allah, Samee U Khan, and Yongji Wang. C2detector: a covert channel detection framework in cloud computing. *Security and Communication Networks*, 7(3):544–557, 2014.

[51] Ang Chen, W Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting covert timing channels with time-deterministic replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 541–554, 2014.

[52] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races*, pages 1–17. IEEE, 2018.

[53] Lina Wang, Weijie Liu, Neeraj Kumar, Debiao He, Cheng Tan, and Debin Gao. A novel covert channel detection method in cloud based on xsrm and improved event association algorithm. *Security and Communication Networks*, 9(16):3543–3557, 2016.

[54] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 216–228. IEEE, 2014.

[55] Ruby B. Lee Tianwei Zhang, Yinqian Zhang. Cloudradar: A real-time side-channel attack detection system in clouds. In *19th Intern. Symp. Research in Attacks, Intrusions, and Defenses*, 2014.

[56] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[57] Arkadiusz Socała and Michael Cohen. Automatic profile generation for live linux memory analysis. *Digital Investigation*, 16:S11–S24, 2016.

[58] Michael Cohen. Rekall memory forensics framework. *DFIR Prague*, 2014.

[59] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[60] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng

Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 504–516. ACM, 2016.

[61] Rui Wu, Ping Chen, Peng Liu, and Bing Mao. System call redirection: A practical approach to meeting real-world virtual machine introspection needs. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 574–585. IEEE, 2014.

[62] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 416–427. IEEE, 2014.

[63] Brendan Saltaformaggio, Dongyan Xu, and Xiangyu Zhang. Busmonitor: A hypervisor-based solution for memory bus covert channels. In *6th European Workshop on Systems Security*, 2013.

[64] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *1st ACM Cloud Computing Security Workshop*, 2009.

[65] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *41st IEEE/IFIP International Conf. Dependable Systems and Networks*, 2011.

[66] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Architecture and Code Optimization*, 8(4), 2012.

[67] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th Intern. Symp. Comp. Arch.*, 2007.

[68] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM Intern. Symp. Microarchitecture*, 2008.

[69] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3), 2008.

[70] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX*

Security Symp., 2012.

[71] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *23rd ACM Conf. Comp. and Comm. Sec.*, 2016.

[72] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *18th European Symp. Research in Computer Security*, 2013.

[73] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symp.*, 2014.

[74] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *20th Intern. Symp. High Performance Comp. Arch.*, 2014.

[75] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, 2017.

[76] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4), 1992.

[77] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *3rd ACM Cloud Computing Security Workshop*, 2011.

[78] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. Predictive black-box mitigation of timing channels. In *17th ACM Conf. Comp. and Comm. Sec.*, 2010.

[79] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Predictive mitigation of timing channels in interactive systems. In *18th ACM Conf. Comp. and Comm. Sec.*, 2011.

[80] Peng Li, Debin Gao, and Michael K Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In *43rd IEEE/IFIP International Conf. Dependable Systems and Networks*, 2013.

[81] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *ISOC Network and Distributed System Security Symp.*, 2015.

[82] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symp.*, 2015.

[83] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Intern. Symp. Comp. Arch.*, 2012.

[84] Robert J Creasy. The origin of the vm/370 time-sharing system. *Ibm Journal of Research and Development*, 25(5):483–490, 1981.

[85] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *symposium on operating systems principles*, 37(5):164–177, 2003.

[86] Secure virtual machine architecture reference manual. `https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf`.

[87] Peng Li and Lee Toderick. Cloud in cloud: approaches and implementations. In *Proceedings of the 2010 ACM conference on Information technology education*, pages 105–110, 2010.

[88] JingZheng Wu, Liping Ding, Yongji Wang, and Wei Han. Identification and evaluation of sharing memory covert timing channel in xen virtual machines. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 283–291. IEEE, 2011.

[89] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. Reverse engineering hardware page table caches using side-channel attacks on the mmu.

[90] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *36th IEEE Symp. Security and Privacy*, pages 640–656, 2015.

[91] Qiuyu Xiao, Michael K Reiter, and Yinqian Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1582–1594. ACM, 2015.

[92] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. Acoustic data exfiltration from speakerless air-gapped computers via covert hard-drive noise diskfiltration. In *European Symposium on Research in Computer Security*, pages 98–115. Springer, 2017.

[93] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *36th IEEE Symp. Security and Privacy*, 2015.

[94] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conf., Cryptographers' Track*, 2006.

[95] Weijie Liu, Debin Gao, and Michael K Reiter. On-demand time blurring to support side-channel defense. In *European Symposium on Research in Computer Security*, pages 210–228. Springer, 2017.

[96] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Workshop on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[97] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 629–636. IEEE, 2015.

[98] Linux Hugepages. `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`.

[99] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[100] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3), 1970.

[101] 梁鑫, 桂小林, 戴慧珺, and 张晨. 云环境中跨虚拟机的 cache 侧信道攻击技术研究. *计算机学报*, 40(2):317–336, 2017.

[102] 沈晴霓 and 李卿. 云计算环境中的虚拟机同驻安全问题综述. *集成技术*, (5):5–17, 2015.

[103] Peng Li, Debin Gao, and Michael K Reiter. StopWatch: A cloud architecture for timing channel mitigation. *ACM Trans. Information and System Security*, 17(2), 2014.

[104] Cisco VxLan white paper. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-738503.html`.

[105] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24thUSENIX Security Symp.*, pages 913–928, 2015.

[106] Wang Lina, Gao Han-jun, Liu Wei, and Peng Yang. Detecting and managing hidden process via hypervisor. *Journal of Computer Research and Development*,

48(8):1534–1541, 2011.

[107] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *32nd Intern. Conference on Distributed Computing Systems*, pages 285–294. IEEE, 2012.

[108] Jiangtao Zhai, Guangjie Liu, and Yuewei Dai. Detection of tcp covert channel based on markov model. *Telecommunication Systems*, 54(3):333–343, 2013.

[109] Sachin Kadloor, Negar Kiyavash, and Parv Venkitasubramaniam. Mitigating timing based information leakage in shared schedulers. In *INFOCOM, 2012 Proceedings IEEE*, pages 1044–1052. IEEE, 2012.

[110] Hermine Hovhannisyan, Kejie Lu, Rongwei Yang, Wen Qi, Jianping Wang, and Mi Wen. A novel deduplication-based covert channel in cloud storage service. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015.

[111] Hanjun Gao, Lina Wang, Wei Liu, Yang Peng, and Hao Zhang. Preventing secret data leakage from foreign mappings in virtual machines. In *Security and Privacy in Communication Networks*, pages 436–445. Springer, 2011.

[112] John C Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3-4), 1992.

[113] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[114] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *ISOC Network and Distributed System Security Symp.*, volume 3, pages 191–206, 2003.

[115] Jiye Liang, Xingwang Zhao, Deyu Li, Fuyuan Cao, and Chuangyin Dang. Determining the number of clusters using information entropy for mixed data. *Pattern Recognition*, 45(6):2251–2265, 2012.

[116] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. Os-level side channels without procfs: Exploring cross-app information leakage on ios. In *Proceedings of the Symposium on Network and Distributed System Security*, 2018.

[117] Wenke Lee, Salvatore J Stolfo, et al. Data mining approaches for intrusion detection. In *7th USENIX Security Symp.*, 1998.

[118] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 477–487. ACM, 2009.

[119] Bryan D Payne. Simplifying virtual machine introspection using libvmi. *Sandia report*, pages 43–44, 2012.

[120] 李保珲, 徐克付, 张鹏, 郭莉, 胡玥, and 方滨兴. 虚拟机自省技术研究与应用进展. 软件学报, 27(6):1384–1401, 2016.

[121] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-assisted fine-grained code-reuse attack detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 66–85. Springer, 2015.

[122] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[123] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, pages 109–129. Springer, 2014.

[124] Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/.

[125] Cody Pierce, Matthew Spisak, and Kenneth Fitch. Capturing 0day exploits with perfectly placed hardware traps, 2014.

[126] PERFMON2. https://perfmon2.sourceforge.net/.

[127] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 473–482. IEEE, 2006.

[128] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[129] 刘维杰, 王丽娜, 谈诚, and 徐来. 基于 VMFUNC 的虚拟机自省触发机制. 计算机研究与发展, 54(10):2310–2320, 2017.

[130] libVMI API. `http://libvmi.com/api`.

[131] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Intern. Conf. Parallel Architectures and Compilation Techniques*, 2008.

[132] PARSEC Hooks Instrumentation API. `http://parsec.cs.princeton.edu/doc/`

man/man7/hooks.7.html.

[133] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *23rd ACM Conf. Comp. and Comm. Sec.*, 2016.

[134] Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in Deterland. In *Conf. Timely Results in Operating Systems*, 2015.

[135] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. arXiv:1702.08719, 2017.

[136] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *12th ACM Asia Conf. Comp. and Comm. Sec.*, 2017.

[137] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx. In *26th USENIX Security Symposium (USENIX Security 17),(Vancouver, BC)*, pages 51–67, 2017.

[138] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *22nd ACM Conf. Comp. and Comm. Sec.*, 2015.

[139] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. *arXiv preprint arXiv:1710.00551*, 2017.

[140] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad Reza Sadeghi. Can't touch this: Practical and generic software-only defenses against rowhammer attacks. 2016.

[141] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[142] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 859–874. ACM, 2017.

攻博期间发表的科研成果目录

发表论文：

[1] On-Demand Time Blurring to Support Side-Channel Defense. In Proc. of ESORICS 2017, Part 2, LNCS, 1-19. DOI：10.1007/978-3-319-66399_12. (第一作者, CCF B & EI Indexed)

[2] 基于 VMFUNC 的虚拟机自省触发机制 [J]. 计算机研究与发展, 2017, 54(10): 2310-2320. (第一作者, EI Indexed)

[3] A novel covert channel detection method in cloud based on XSRM and improved event association algorithm[J]. Security & Communication Networks, 9(16):3543-3557, 2016. (第二作者 & 导师一作, CCF C & SCI Indexed)

[4] The lifting factorization of 2D 4-channel nonseparable wavelet transforms[J]. Information Sciences, 2018. DOI: 10.1016/j.ins.2018.05.012. (通信作者, CCF B & SCI Indexed)

[5] Factoring two-dimensional two-channel nonseparable stripe filter banks into lifting steps[J]. IET Image Processing, 2018. DOI: 10.1049/iet-ipr.2017.0935. (通信作者, CCF C & SCI Indexed)

[6] CAPT: Context-Aware Provenance Tracing for Attack Investigation[J]. China Communications, 2018, 15(2): 153-169. (SCI Indexed)

[7] 面向云平台的硬件辅助 ROP 检测方法 [J]. 清华大学学报（自然科学版）, 2018, 58(3): 237-242. (EI Indexed)

[8] Construction of Sampling Two-Channel Nonseparable Wavelet Filter Bank and Its Fusion Application for Multispectral Image Pansharpening. In Proc. of Pacific-Rim Conference on Multimedia 2017, LNCS 10736, 859-868. DOI: 10.1007/978-3-319-77383-4_84. (CCF C & EI Indexed)

[9] 八通道 MSVD 构造及其在多聚焦图像融合中的应用 [J]. 电子学报, 2016, 44(7): 1694-1701. DOI: 10.3969/j.issn.0372-2112.2016.07.025. (EI Indexed)

[10] 一种面向不可信第三方匿名服务的位置隐私保护方法 [OL]. 中国科技论文在线, 2014.

[11] Construction method of three-channel nonseparable symmetric wavelets with arbitrary dilation matrices and its applications in multispectral image fusion[J]. IET

Image Processing, pages 679-685, 2013. (CCF C & SCI Indexed)

[12] 云环境下 APT 攻击的防御方法综述 [J]. 计算机科学, 2016, 43(3):1-7.

[13] 基于虚拟机回放的恶意行为检测技术 [J]. 武汉大学学报 (理学版), 2016, 62(5):437-443.

[14] 虚拟化架构下管理域安全增强方法 [J]. 武汉大学学报 (理学版), 2016, 62(3):218-224.

[15] 基于网络编码的远程数据验证方法 [J]. 华中科技大学学报 (自然科学版), 2014(11):17-22. DOI: 10.13245/j.hust.141104. (EI Indexed)

[16] 基于采样二通道不可分小波的多光谱图像融合 [J]. 电子学报, 2013, 41(4):710-716. (EI indexed)

[17] CACDP: 适用于云存储动态策略的密文访问控制方法 [J]. 计算机研究与发展, 2014, 51(7):1424-1435. (EI indexed)

[18] IDVE: 面向跨数据中心的虚拟域一致性迁移机制 [J]. 四川大学学报 (工程科学版), 2015, 47(1):20-26. DOI: 10.15961/j.jsuese.2015.01.003. (EI Indexed)

学术报告：

[1] On-Demand Time Blurring to Support Side-Channel Defense. ESORICS 2017, Oslo, Norway.

发明专利与软著：

[1] 一种基于 CPU 硬件特性的虚拟机自省快速触发机制

[2] 一种基于 Xen 的硬件辅助代码复用攻击检测方法

[3] 面向 IaaS 云平台的同驻检测软件

[4] 基于 Xen 的硬件辅助代码复用攻击检测系统 V1.0

承担项目：

[1] 国家 863 计划项目 - 云计算环境中恶意行为安全监控与检测技术研究

[2] NSFC 基金项目 - 基于可信虚拟域的云安全边界防护模型与方法研究

[3] 国家发改委重大专项 - 云计算业务专业化安全服务评估

[4] 华为公司预研项目 - 虚拟云边界防御

[5] XXX 预研项目 - 基于身份认证与任务角色访问控制机制研究

[6] NSFC 基金项目 - 基于可信虚拟域的敏感数据防泄漏模型与方法研究

获得奖励：

[1] 湖北省科技进步一等奖 - 云计算环境下敏感数据防泄漏方法与系统 (排名第 11), 2012

致　　谢

时光荏苒，岁月如梭。一转眼在武汉大学的十年时光已成过往。在求学过程中，我收获过论文被接收的喜悦，也遇到过各种各样的困难和挫折。我从一开始的选题 - 基于虚拟机自省的安全增强，到 2015 年后重新开始现在的新方向 - 跨虚拟机侧信道，经历了许多，学习了许多，成长了许多。在此，向帮助我的人表示衷心的感谢。

首先我要感谢的是我的导师，王丽娜教授。在自保研以来攻读博士学位的七年里，我受到了老师的悉心指导和巨大帮助。王老师为人和善、治学严谨、务真求实态度给我留下深刻的印象，对我的锻炼为我以后的工作和生活打下了坚实的基础。恩师时刻想着学生的成长让我敬佩不已，这一切都是我终生受用的瑰宝，将永远铭记在心，并受益终生。

第二需要感谢的是我在新加坡管理大学 (SMU) 的指导老师，Prof. Debin Gao，和在美国北卡罗纳大学 (UNC) 的指导老师，Prof. Mike Reiter。他们是将我带入跨虚拟机侧信道前沿方向的领路人。Prof. Debin 执行力强、充满热情、思路广阔。Prof. Mike 高屋建瓴、待人温和、治学严谨。他们不仅在我的专业方向上给予技术指导，同时教会了我正确对待科研的态度。

同时，需要感谢的是项目组的余荣威老师和赵磊老师，师兄般亲切的余老师经常带领我们做项目，跟余老师学到了很多处事原则和人生哲理。亦师亦友的赵磊老师在学术道路上给了我非常好的机会和经验，跟赵磊老师学到了许多需要具备的科研素质与技巧。

其次，要感谢博士师兄弟们。他们是：师兄高汉军、彭瑞卿、张浩、任正伟、王旻杰、谈诚、徐一波；并肩作战的徐来、翟黎明；师弟唐奔宵、汪润、任魏翔、王文琦、嘉炬、张桐等。和你们一起奋斗的青春终生难忘。

感谢朝夕相处的虚拟化安全小组的成员，他们是：邓入弋、杜钢、尹正光、章鑫、付楠、吴頔、鞠瑞、周伟康、强辰谊、王斐、廖利坚等。我取得的成绩很大一部分来自于你们的帮助，正是与你们的讨论和学习，才促进了我科研工作的进步。尤其感谢尹正光在阿里云时给本文虚拟机同驻方案提供快速便捷的检测；章鑫在本文实现 Whisper 侧信道时给予的帮助。感谢与我一起在 SMU 共同奋斗的：刘西蒙、王伟、常冰、章张锴、丁文秀、赵思齐、袁平海、林艳、姜百合，和你们在一起的时光十分宝贵，你们对我在学术上的帮助极大，让我难以忘怀。在此，深切感谢你们的帮助和鼓励，祝愿你们学业、事业有成！

感谢我的父母给予我物质和精神上的巨大支持，感谢你们对我的养育之恩，为我创造的宽松优越的生活和学习环境，使我能够专心进行学习科研工作。感谢所有关心我、支持我和帮助我的朋友们，是你们友情伴我渡过了博士期间终生难忘的日子，我将永远铭记在心！感谢空天信息安全与可信计算教育部重点实验室的各位老师，他们在我的从本科到博士期间给予了无私的帮助和指点。感谢武汉大学提供如此便利的学习和工作条件。

最后，十分感谢王丹磊师弟，给我关于数学证明方面的建议。十分感谢周伟康师弟，帮忙校稿，提出很多宝贵的意见。