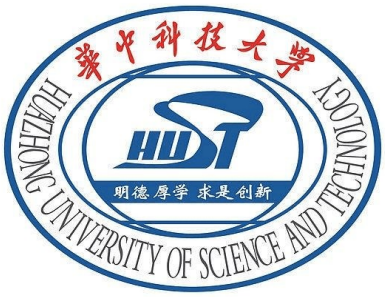
華中科技大學



2022 年度云计算与虚拟化论文阅读报告

A Study on the Security Implications of Information Leakages in Container Clouds

姓　　名：XXX

学　　号：U2019XXXXX

班　　级：网安 190X 班

任课教师：袁斌

论文编号：4

2022 年 12 月 28 日

2022年度云计算与虚拟化论文阅读报告

1 论文拟解决的问题

现阶段，容器技术提供了一个轻量级的操作系统级虚拟主机环境。它的出现深刻地改变了多层分布式应用程序的开发和部署模式。然而，由于Linux内核中系统资源隔离机制的实现不完整，在基于多租户容器的云服务上共享操作系统内核的多个容器仍然存在一些安全问题。作者介绍了他们发现的容器内可访问的信息泄漏通道。这样的通道将一系列系统范围的主机信息暴露给容器，而没有适当的资源分区。通过利用这些泄漏的主机信息，恶意对手（充当容器云中的租户）更容易发起可能影响云服务可靠性的攻击。

进一步说，尽管容器服务取得了成功，但在同一操作系统内核上运行多个容器（可能属于不同租户）时，始终存在安全和隐私问题。为了支持容器云上的多租户，作者已经观察到Linux内核正在努力实施跨容器隔离和取消用户级容器的特权。现有的支持容器的内核特性极大地缩小了暴露给容器租户的攻击面，可以抑制大多数现有的恶意攻击。然而，并不是所有的Linux内核子系统都能区分容器和主机之间的执行上下文，因此它们可能会向容器化应用程序公开系统范围的信息。一些子系统被认为是容器适应的低优先级。其余的人在转换代码库时面临实现困难，他们的维护人员不愿意接受剧烈的更改。为了堵塞这些漏洞，当前的容器运行时软件和容器云提供商通常利用访问控制策略来屏蔽这些容器无关子系统的用户内核接口。然而，这种手动和临时修复只能覆盖一小部分暴露的攻击表面。

在整个论文架构上，作者先介绍了他们发现的违规者泄露渠道及其泄露信息。然后演示了基于泄漏的全系统数据的高带宽可靠隐蔽信道的构建。并详细介绍了通过这些渠道利用泄漏信息的协同力量攻击。最后介绍了一种通用的两级防御机制，以及Linux内核中基于powerbased命名空间的具体设计和实现。我在下一节详细展开叙述。

2 论文提出的核心方法

2.1 容器云中的信息泄漏

Linux内核提供了大量支持，以强制执行容器抽象的资源隔离和控制。这样的内核机制是在多租户云上运行容器的启用技术。由于优先级和难度级别的原因，Linux内核的一些组件尚未转换为支持容器化。作者系统地探索内核的哪些部分没有被发现，根本原因是什么，以及潜在的对手如何利用它们。如图1左侧所示，作者设计了一个交叉验证工具，以自动发现这些基于内存的伪文件，这些伪文件将主机信息暴露给容器。关键思想是在两个执行上下文中递归地探索procfs和sysfs下的所有伪文件，一个在非特权容器中运行，另一个在主机上运行。作者根据文件路径对文件进行对齐和重新排序，然后在这两个上下文之间对同一文件的内容进行成对的差异分析。如果从特定伪文件访问的系统资源尚未在Linux内核中命名，则主机和容器将达到同一块内核数据。否则，如果名称空间正确，容器可以检索自己的私有和自定义内核数据。使用此交叉验证工具，作者可以快速识别可能向容器公开系统范围主机信息的伪文件（及其内部内核数据结构）。

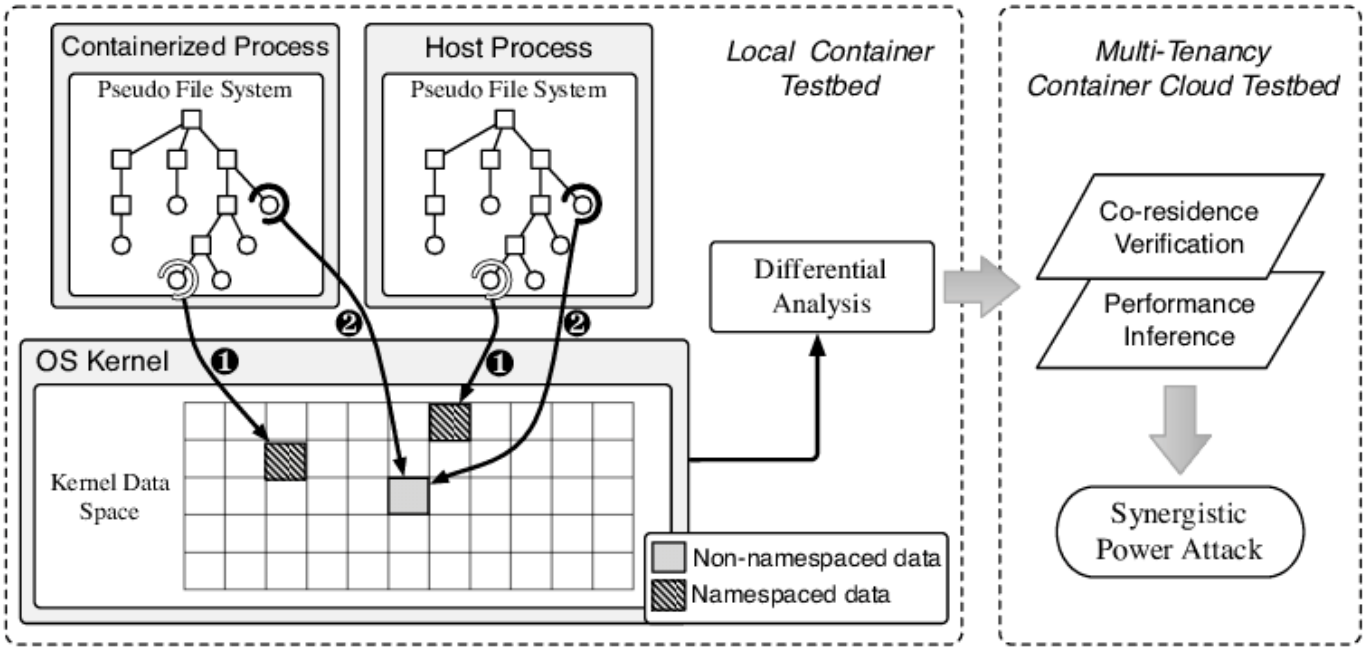


图 1: 信息泄漏检测和云检查框架

作者还列出了可能泄漏主机信息的所有伪文件。这些泄漏通道包含主机信息的不同方面。容器用户可以检索内核数据结构（例如，/proc/modules 显示加载的模块列表）、内核事件（例如，/proc/interrupts 显示每个 IRQ 的中断数）和硬件信息（例如/proc/cpuinfo 和/proc/meminfo 分别显示 CPU 和内存的规格）。此外，容器用户可以通过某些通道检索性能统计数据。作者通过检查内核代码（在 Linux 内核 4.7 版中）进一步调查这些信息泄漏的根本原因。通常，泄漏问题是由内核中命名空间的不完整实现引起的。更具体地说，作者总结了以下两个主要原因：（1）现有名称空间缺少上下文检查，（2）某些 Linux 子系统没有（完全）名称空间。作者对容器中的 net_prio.ifpriomap 和 RAPL 进行了两个案例研究，以揭示泄漏的来源。

作者进一步深入研究具体案例，看看它们是否可以被用来检测共同居住的容器。由于容器可以通过作者发现的泄漏通道读取主机信息，作者倾向于测量一些通道是否可以用于检查容器共住。作者定义了三个指标，即唯一性（U）、变异性（V）和操纵性（M），以定量评估每个渠道推断共同居住的能力。

度量 U 指示此通道是否提供能够唯一标识主机的特征数据。这是确定两个容器是否位于同一主机上的最重要和最准确的因素。作者发现了 17 个泄漏通道满足该指标。通常，作者可以将这些通道分为三组：

1. 包含唯一静态标识符的通道。例如，/proc/sys/kernel/random 下的 boot_id 是在启动时生成的随机字符串，对于每个正在运行的内核都是唯一的。如果两个容器可以读取相同的 boot_id，这表明它们在同一主机内核上运行。此组中通道的数据既静态又唯一。

2. 容器租户可以向其中动态植入唯一签名的通道。例如，从/proc/sched_debug，容器用户可以通过该接口检索主机的所有活动进程信息。租户可以在容器内启动具有唯一编制的任务名称的进程。从其他容器中，他们可以通过在自己的 sched_debug 中搜索此任务名称来验证 coresidence。类似的情况适用于 timer_list 和锁。

3. 包含唯一动态标识符的通道。例如，/proc/uptime 有两个数据字段：系统启动时间和启动后

的系统空闲时间（以秒为单位）。它们是累积值，对于每台主机都是唯一的。类似地，RAPL 系统界面中的 energy_uj 是以微焦耳为单位的累积能量计数器。从该组中的通道读取的数据会实时更改，但对于表示主机而言仍然是唯一的。作者根据这些渠道的增长率对其进行排名。增长速度越快，复制的机会越低。

度量 V 表明数据是否随时间变化。使用此功能，两个容器可以同时定期创建此伪文件的快照。然后，他们可以通过检查两个数据快照痕迹是否相互匹配来确定共同居住。例如，从同一时间开始，作者可以在一分钟内每秒从两个容器中记录/proc/meminfo 中的 MemFree。如果这两个 60 点数据跟踪彼此匹配，作者相信这两个容器在同一主机上运行。每个信道都包含不同容量的信息，用于推断共同居住，这可以通过联合香农熵自然测量。作者在下面的方程中定义了熵 H。每个通道 C 包含多个独立数据字段 X_i，n 和 n 表示独立数据字段的数量。每个 X_i 都有可能的值 $\{x_{i1}, \cdots, x_{im}\}$。

$$H[C(X_1, \cdots, X_n)] = \sum_{i=1}^{n}[-\sum_{j=1}^{m} p(x_{ij}) log p(x_{ij})]$$

2.2 构建隐蔽通道

进一步利用包含度量操纵（M）的通道在两个容器之间构建隐蔽通道。作者证明，这些信息泄漏通道中的动态标识符和性能数据都可能被滥用来传递信息。特别是，作者实现了两个通道，并测试了它们的吞吐量和错误率。

作者基于/proc/lock 构造了一个隐蔽通道。

具体来说，发送方可以锁定一个文件以表示 1，并释放锁定以表示 0。虽然锁和文件不在容器之间共享，但可以由/proc/locks 中的接收方检查锁的存在。通过检查特定锁的状态，可以传输信息。此外，可以同时设置多个锁以传输多个比特。为了建立可靠的高带宽隐蔽信道，攻击者需要考虑几个因素。作者详细介绍了在算法 1 中构造基于锁的隐蔽信道的过程。

通道/proc/locks 包含内部内核数据，因此它一直在变化。特别是在存在噪声的云环境中，内容可能会发生巨大波动。可靠传输数据块的第一步是握手：接收方需要找出发送方使用的所有锁。特别地，握手过程负责：（1）设置为数据传输的起点；（2）用于同时传送比特块的锚定锁；（3）同步发送器和接收器。

作者通过为每个数据锁创建特定模式来实现握手。对于一个简化的情况，发送方在短时间内不断获取和释放锁。然后，接收器检查每个数据锁锁定或解锁文件的触发次数。如果切换次数超过特定阈值，则接收器通过跟踪 inode 编号来记录该锁，对于特定文件上的相同锁，inode 编号不变。同时，作者使用一个额外的锁来表示传输信号，该锁用于通知接收器每一轮传输的开始。

在创建特定模式后，理论上发送者可以立即开始传输。然而，接收方的处理过程的时间是未知的和不确定的，特别是在多租户云环境中存在大量锁的情况下。如果发送方传输数据太快，可能会丢失一块比特。虽然发送方可以在发送下一个数据块之前等待一段时间，但这种方法将极大地影响传输速度。

作者还添加了一个 ACK 锁，用于同步发送方和接收方。在获得所有数据锁之后，接收器通过设置 ACK 锁来确认。发送方对 ACK 锁的检测类似于接收方对其他数据锁的检测方法。接收机在用 ACK 锁应答后进入就绪状态，并等待数据传输。

Algorithm 1. Clovert Channel Protocol Based on Locks

N: Number of transmissions.
M: Number of bits of data per transmission.
$L[M]$, L_{signal}, L_{ACK}: Locks on M data files, lock on the signal file, and lock on the ACK file.
T_{hand}: an amount of time for handshake.
$D_{Send}[N]$, $D_{Recv}[N]$: N transmissions to send and receive.

Sender Operations:
for amount of time T_{hand} **do**
  Set $L[M]$, L_{signal};
  Sleep for an amount of time;
  Release $L[M]$, L_{signal};
end for
for amount of time T_{hand} **do**
  Record entries in */proc/locks*;
  Sleep for an amount of time;
end for
for each recorded entry in */proc/locks* **do**
  if Number of toggles $>=$ THRESHOLD **then**
    $ACK_{recevier}$ = True;
  end if
end for

Receiver Operations:
for amount of time T_{hand} **do**
  Record entries in */proc/locks* and sleep;
end for
for each recorded entry in */proc/locks* **do**
  if Number of toggles $>=$ THRESHOLD **then**
    Remember lock's device number;
  end if
end for
if Number of recorded valid locks $== M + 1$ **then**
  for amount of time T_{hand} **do**
    Set L_{ACK};
    Sleep for an amount of time;
    Release L_{ACK};
  end for
end if

Sending Data:
for i = 0 to N - 1 **do**
  for j = 0 to M - 1 **do**
    if $D_{Send}[i][j] == 1$ **then**
      Set L[j];
    else
      Release L[j];
    end if
  end for
  Set L_{signal};
  Sleep for an amount of time;
  Release L_{signal};
  Wait for L_{ACK};
end for

Receiving Data:
while connection is active **do**
  Record entries in */proc/locks*;
  if L_{signal} is presented **then**
    for j = 0 to M - 1 **do**
      if L[j] is presented **then**
        $D_{Recv}[i][j] = 1$;
      else
        $D_{Recv}[i][j] = 0$;
      end if
    end for
    i++;
    Set L_{ACK};
    Sleep for an amount of time;
    Release L_{ACK};
  end if
end while

图 2: 算法 1

由于信息泄漏，容器用户可以检索主机服务器的系统范围性能统计数据。

例如，容器可以通过/proc/stat 获取每个内核的 CPU 利用率，通过/proc/meminfo 或/proc/vmstat 获取内存利用率。性能数据的值受容器运行状态的影响。尽管一个容器在云环境中只能容纳有限的计算资源，但容器中的恶意用户可以谨慎地选择容器中运行的工作负载。通过在性能通道中生成唯一的模式，容器可以构建隐蔽通道来传递信息。作者通过滥用/proc/meminfo 中的内存使用信息来构建一个隐蔽通道。

/proc/meminfo 报告了关于系统内存使用情况的大量有价值的信息，包括可用内存总量、系统未使用的物理 RAM 总量以及脏内存总量。这里作者利用了系统中未使用的内存量。在云环境中，这个值可能会有很大的变化，因为每个容器用户都会影响它。

这种显著的变化对于建立可靠的隐蔽数据传输是不期望的。但是，如果不运行内存密集型工作负载，使用率可能会在可接受的范围内波动。作者首先在服务器上记录一段时间内未使用的内存，以获得基线值。如果该值没有快速而显著的变化，则表明可以建立隐蔽通道。

然后，发送方容器可以分配不同数量的内存来表示位 1 和 0（例如，位 1 为 100 MB，位 0 为 50 MB），这将导致/proc/meminfo 中的 MemFree 字段立即删除。接收方可以通过监视空闲内存的变化来

简单地转换数据。MemFree 在三个级别中有所不同：基本情况、位 1 和位 0。发送方将在发送 1 或 0 之后释放所有分配的内存，以便接收方可以知道上一次传输的结束并为下一个比特做准备。

为了确保可靠的传输，握手是发送方和接收方之间的第一个必要过程。发送方可以选择发送固定模式，例如八个直比特 1，以启动传输。一旦接收方获得握手模式，它将进入数据接收模式。算法 2 中列出了详细的算法。为了减少其他容器的内存消耗所产生的噪声的影响，一旦 MemFree 落入预定义范围，接收方将标记数据传输。同时，增加分配内存的数量可以减少噪声的影响。然而，它会影响传输带宽，因为分配和释放内存会消耗时间。

为了减少来自环境的干扰，用户可以进一步设计高级别的可靠协议，例如使用校验和，以确保传输数据的完整性。

Algorithm 2. Covert Channel Protocol Based on /proc/meminfo

Mem_{free}: Amount of system memory currently unused by the system.
Mem_{high}, Mem_{low}: Predefined amount of memory representing a bit 1 and a bit 0, respectively.
$Limit_1$, $Limit_0$: Maximum value of Mem_{free} that will be recognized as 1 and 0, respectively.
$D_{start}[M]$, $D_{send}[N]$, $D_{recv}[N]$: Handshake sequence, sending data bits, receiving data bits.
$M_{recording}[]$: the recordings of free memory at the receiver.

Sender Operations:
for i = 0 to M-1 **do**
  if $D_{start}[i] == 1$ **then**
    Allocate Mem_{high};
  else
    Allocate Mem_{low};
  end if
  Set memory chunk, sleep, and free;
end for
for i = 0 to N-1 **do**
  if $D_{send}[i] == 1$ **then**
    Allocate Mem_{high};
  else
    Allocate Mem_{low};
  end if
  Set memory chunk, sleep, and free;
end for

Receiver Operations:
Recording current value of Mem_{free} to $M_{recording}$;
Obtain average Mem_{free};
Calculate $Limit_1$ and $Limit_0$;
for i = 0 to $M_{recording}$.length - 1 **do**
  if $M_{recording}[i]$ is a local minimum and $M_{recording}[i] <= Limit_1$ **then**
    $D_{recv}[i]$ = 1;
  else if $M_{recording}[i]$ is a local minimum and $M_{recording}[i] <= Limit_0$ **then**
    $D_{recv}[i]$ = 0;
  end if
end for

图 3: 算法 2

2.3 协同电力攻击

因为 procfs 和 sysfs 都以只读方式安装在容器内，所以恶意租户只能读取这些信息，但不允许修改。作者认为，通过利用泄漏通道，攻击者可以通过学习主机的运行时状态来做出更好的决策。作者介绍了可能影响数据中心可靠性的停电威胁范围内的潜在协同电源攻击。作者证明，对手可以利用作者发现的这些信息泄漏来放大攻击效果，降低攻击成本，并促进攻击协调。所有实验都在真实世界的容器云中进行。

发动成功的电力攻击的关键是产生一个可以超过电力设施供应能力的短时高功率尖峰。电力攻击的根本原因是功率超额订阅的广泛采用，这使得功率峰值有可能超过安全阈值。此外，机架级功率上限机制只能在分钟级时间粒度内做出反应，为短时高功率尖峰的发生留出空间。在最危急的情况下，过度充电可能会使分支断路器跳闸，导致停电，最终导致服务器停机。功率峰值的高度主要由攻击者控制的资源决定。现有的电源攻击通过定制功率密集型工作负载（称为电源病毒）来最大化功耗。

对于容器云中的协同电力攻击，对手可以通过 RAPL 通道监控整个系统的功耗，并实时了解功耗模式的波峰和波谷，而不是无差别地启动功率密集型工作负载。因此，它们可以利用后台功耗（由同一主机上的其他租户的良性工作负载产生），并在服务器处于高峰运行时间时叠加其电源攻击。

这与金融市场中的内幕交易现象相似——内幕信息较多的人总能在适当的时间进行交易。对手可以利用通过 RAPL 通道泄漏的“内部”功耗信息，将已经很高的功耗增加到更高的水平，从而提高其功率峰值。

与传统的力量攻击不同，协同力量攻击的另一个独特特征是其攻击编排。假设攻击者已经在控制多个容器实例。如果这些容器分散在数据中心的不同位置，那么它们在多个物理服务器上的电源添加不会对电源设施造成压力。现有的功率封顶机制可以毫无困难地容忍来自不同位置的多个小功率浪涌。发动实际电力攻击的唯一方法是将所有“弹药”集中到相邻位置，同时攻击单个电源。这里作者将深入讨论攻击容器实例的编排。

通过利用多个泄漏通道，攻击者可以将多个容器实例聚合到一个物理服务器中。特别是在 CC1 上的实验中，作者选择使用 timer_list 来验证多个容器的共同驻留。后续作者解释了详细的验证方法。作者重复创建容器实例并终止不在同一物理服务器上的实例。通过这样做，作者成功地在同一台服务器上部署了三个容器。作者在每个容器中运行 Prime 基准测试的四个副本，以充分利用分配的四个内核。结果如图 4 所示。正如作者所看到的，每个容器可以提供大约 40W 的功率。使用三个容器，攻击者可以很容易地将功耗提高到大约 230W，这比单个服务器的平均功耗高大约 100 W。

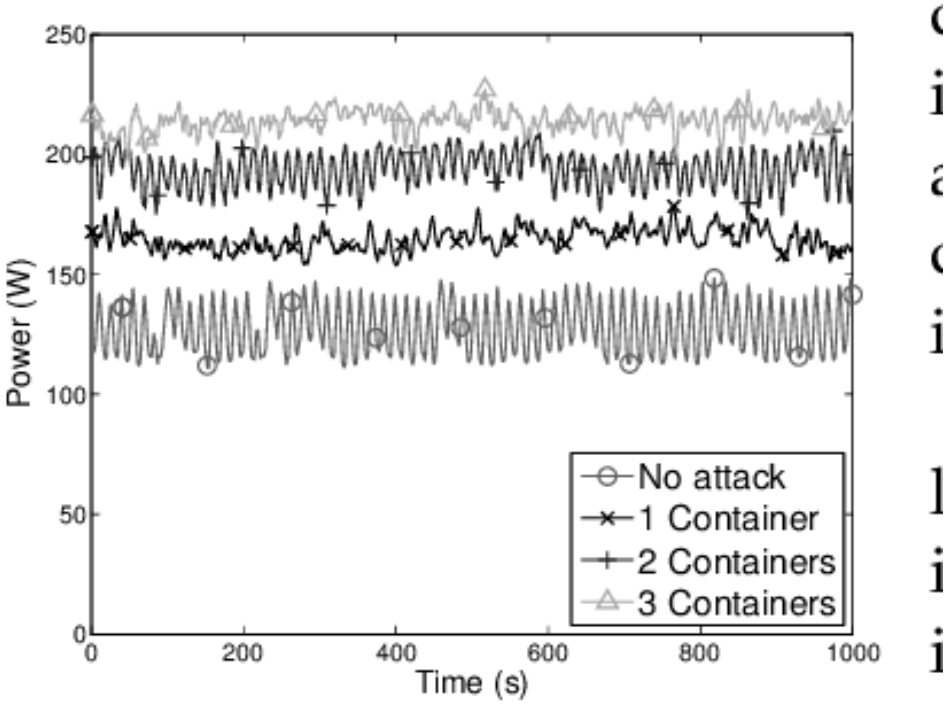


图 4: 受攻击服务器的功耗

2.4 防御方法

作者提到的首先是两阶段防御机制。

直观地说，该解决方案应该消除所有泄漏，以便不会通过这些渠道检索到泄漏的信息。作者将防御机制分为两个阶段来弥补漏洞：（1）屏蔽通道和（2）增强容器的资源隔离模型。

在第一阶段，系统管理员可以明确拒绝对容器内通道的读取访问，例如，通过 AppArmor 中的安全策略或装载伪文件“不可读”。这不需要对内核代码进行任何更改（合并到上游 Linux 内核可能需要一些时间），并且可以立即消除信息泄漏。此解决方案取决于容器内运行的合法应用程序是否使用这些通道。如果这些信息与容器化应用程序正交，那么屏蔽它不会对容器租户产生负面影响。作者已

经向 Docker 和表 1 中列出的所有云供应商报告了作者的结果，并收到了积极的响应。作者正在与容器云供应商合作，以解决此信息泄漏问题，并将对容器中托管的应用程序的影响降至最低。这种屏蔽方法可以快速修复基于内存的伪文件系统中的所有泄漏，但它可能会对容器化应用程序的功能增加限制，这与容器提供通用计算平台的概念相矛盾。

在第二阶段，防御方法涉及修复缺失的命名空间上下文检查，并虚拟化更多的系统资源（即新命名空间的实现），以增强容器的隔离模型。作者首先向 Linux 内核维护人员报告了与现有名称空间相关的信息泄露错误，他们很快发布了一个新的补丁来解决其中一个问题（[CCVE-20175967]）。对于没有命名空间隔离保护的其他通道，作者需要更改内核代码，以强制执行系统资源的细粒度分区。由于每个通道都需要单独固定，因此这种方法可能需要付出非平凡的努力。虚拟化特定的内核组件可能会影响多个内核子系统。

此外，有些系统资源不容易精确地分区到每个容器。然而，作者认为这是问题的根本解决方案。特别是，为了防御协同电源攻击，作者在 Linux 内核中设计并实现了一个基于电源的概念验证命名空间，以向每个容器显示分区电源使用情况。

第二种方法是基于电源的命名空间。通过每个容器的不变 RAPL 接口显示每个容器的电源使用情况。如果不泄漏系统范围内的功耗信息，攻击者就无法推断主机的电源状态，从而消除了将功率密集型工作负载叠加在良性功率峰值上的机会。

此外，有了每个容器的功耗统计数据，作者可以动态限制超过其预定义功耗阈值的容器的计算能力（或增加使用费）。容器云管理员可以基于这个基于电源的命名空间设计更细粒度的计费模型。

作者的设计有三个目标。（1）准确度：由于没有硬件支持每个容器的功率划分，作者基于软件的功率建模需要反映每个容器的准确功率使用情况。（2）透明度：容器内的应用程序应该不知道这个名称空间外的电源变化，电源子系统的接口应该保持不变。（3）效率：电源分区不应在容器内或容器外产生微不足道的性能开销。

作者在图 5 中说明了系统的工作流程。作者基于电源的命名空间由三个主要组件组成：数据收集、电源建模和动态校准。

作者在容器中保持相同的 Intel RAPL 接口，但更改了处理能量使用的读取操作的实现。一旦检测到能量使用的读取操作，修改后的 RAPL 驱动程序将检索每个容器的性能数据（数据收集），使用检索到的数据对能量使用进行建模（功率建模），并最终校准建模的能量使用（动态校准）。作者将在下面详细讨论每个组件。

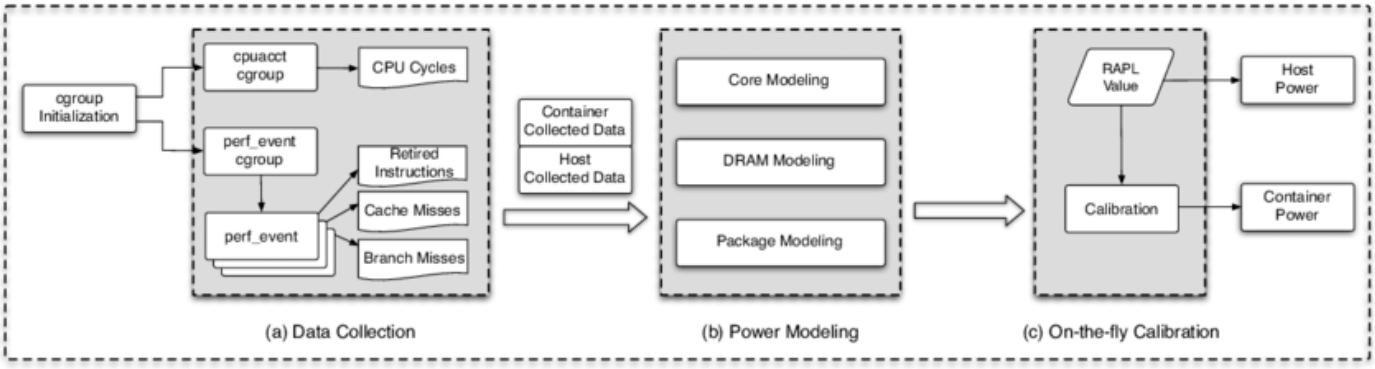


图 5: 基于电源的命名空间的工作流

重点说一下电源建模阶段。为了实现基于电源的名称空间，作者需要将电源消耗归因于每个容器。RAPL 不提供瞬时功耗，而是分别为封装、内核和 DRAM 提供累积能耗。功耗可以通过测量时间单位窗口内的能耗来计算。作者基于电源的命名空间还提供了与原始 RAPL 接口相同格式的每容器累计能量数据。

作者首先将核心的功耗归因于此。

对于 DRAM，作者使用缓存未命中的数量来描述能量。作者在文中列了核心实验中具有相同配置的相同基准的能耗。它清楚地表明缓存未命中的数量与 DRAM 能量近似线性。

基于此，作者使用缓存未命中的线性回归来建模 DRAM 能量。

对于封装的功耗，作者将核心、DRAM 和一个额外常数的值相加。具体模型如下面的等式所示，其中 M 表示建模能量；CM、BM、C 分别表示缓存未命中、分支未命中和 CPU 周期的数量；F 是通过多次线性回归得到的函数，以拟合斜率。I 是是失效指令的数量。$\alpha, \beta, \gamma, \lambda$ 是从中的实验数据得出的常数。

$$M_{core} = F(\frac{CM}{C}, \frac{BM}{C}) \cdot I + \alpha$$

$$M_{dram} = \beta \cdot CM + \gamma$$

$$M_{package} = M_{core} + M_{dram} + \lambda$$

这里作者讨论浮点指令对功率建模的影响。虽然单个浮点指令可能比整数操作消耗更多的能量，但浮点指令比率高的工作负载实际上可能会导致总体功耗降低，因为功能单元可能在流水线的不同阶段被迫闲置。有必要考虑微观架构，以构建更精细的模型。作者计划在今后的工作中追求这一方向。

3 论文贡献总结

作者系统地探索和识别可能意外暴露主机操作系统和共同驻留容器信息的容器内泄漏通道。这种信息泄漏包括主机系统状态信息（例如，功耗、性能数据、全局内核数据和异步内核事件）和单个进程执行信息（例如进程调度、cgroups 和进程运行状态）。在特定定时暴露的特征信息可以帮助唯一地识别物理机器。此外，恶意租户可以通过预先获取系统范围的知识来优化攻击策略并最大化攻击效果。作者在 Docker 和 LinuX Container（LXC）的本地测试台上发现了这些泄漏通道，并在五个公共商业多租户容器云服务上验证了它们（部分）的存在。

进一步，作者证明了，这些信息泄露渠道存在多种安全隐患。一般来说，尽管这些通道是只读安装的，但恶意容器租户仍然可以利用这些通道推断同一物理机器上其他容器的私有数据，检测和验证共同居住，并建立秘密通道来秘密传输信息。作者还提出了几种技术，让攻击者通过利用这些信息泄漏渠道推断共同居住。与基于缓存的隐蔽通道等传统方法相比，作者的方法对云环境中的噪声更具弹性。作者根据这些渠道的风险水平对其进行排名。作者发现，多个通道的系统范围价值可能会受到容器实例中活动的影响。通过专门操纵容器中运行的工作负载，攻击者可以实现可靠和高速的隐蔽通道，以打破部署在云中的隔离机制。例如，攻击者可以通过故意获取和释放锁而不引起网络活动，在容器之间秘密传输比特。同一物理机器上的所有容器都可以观察到泄漏的信息。为了揭示这些泄漏通道的安全风险，作者使用不同的技术构建了两个隐蔽通道，并在真实的多租户云环境中测试了它们的带宽。

作者进一步设计了一种高级攻击，称为协同电力攻击，以利用通过这些渠道泄漏的看似无害的信息。作者证明，这种信息暴露可以极大地放大攻击效果，降低攻击成本，并简化攻击编排。电力攻击已被证明是对现有数据中心的真正威胁。由于没有底层云基础设施运行状态的信息，现有的电力攻击只能盲目地启动电力密集型工作负载以产生电力尖峰，并希望高尖峰会导致分支断路器跳闸而导致停电。这种攻击可能代价高昂，效果不佳。然而，通过学习系统范围的状态信息，攻击者可以（1）选择发动攻击的最佳时机，即将功率密集型工作负载叠加在由良性工作负载触发的现有功率峰值上，以及（2）通过检测受控容器的近距离驻留来同步同一物理机器/机架上的多个电力攻击。作者在一个真实世界的容器云服务上进行了概念验证实验，并定量地证明了作者的攻击能够以较低的成本产生更高的功率峰值。

作者进一步深入分析了这些泄漏通道的根本原因，并发现这种暴露是由于 Linux 内核中容器实现的不完整覆盖造成的。作者提出了一种两阶段防御机制来解决容器云中的这个问题。特别是，为了防御协同电源攻击，作者在 Linux 内核中设计并实现了一个基于电源的命名空间，以在更细粒度（容器）级别划分功耗。作者从准确性、安全性和性能开销的角度来评估基于电源的命名空间。作者的实验结果表明，作者的系统可以以很小的性能开销抵消基于容器的电力攻击。

4 论文实验设计总结与分析

作者从三个方面评估本地计算机上基于电源的命名空间：准确性、安全性和性能。作者的测试台配备了 Intel i7-6700 3.40 GHz CPU，8 核，16 GB RAM，运行 Ubuntu Linux 16.04 内核版本 4.7.0。

4.1 准确性

作者使用 SPECCPU2006 基准来测量功率建模的准确性。作者将建模的电力使用与通过 RAPL 获得的地面实况进行了比较。功耗等于每秒的能耗。由于 Docker 容器的安全策略的限制，作者选择了一个子集 SPECCPU2006 基准，这些基准可以在容器内运行，并且与用于功率建模的基准没有重叠。定义如下：

$$\xi = \frac{|(E_{RAPL} - \delta_{diff}) - M_{container}|}{E_{RAPL} - \delta_{diff}}$$

其中 E_{RAPL} 是从主机上的 RAPL 读取的功耗，$M_{container}$ 是在容器内读取的相同工作负载的建模功耗。请注意，主机和容器都在空闲状态下消耗电力，但有细微差别。作者使用常数 δ_{diff} 作为修改器，反映主机和容器在空闲状态下的功耗差异。图 6 所示的结果表明，作者的功率建模是准确的，因为所有测试基准的误差值都低于 0.05。

4.2 安全性

作者还从安全角度评估作者的系统。

启用基于电源的命名空间后，容器应仅检索容器内消耗的电源，而不知道主机的电源状态。

作者在测试台上推出两个容器进行比较。作者在一个容器中运行 SPECCPU2006 基准测试，并让另一个容器空闲。作者记录容器和主机的每秒功耗。作者在图 7 中显示了 401.bzip2 的结果。所有其他基准都表现出类似的模式。

当两个容器都没有工作负载时，它们的功耗与主机的功耗相同，例如，从 0 到 10。一旦容器 1 在 10s 开始工作负载，作者可以发现容器 1 和主机的功耗同时激增。从 10 秒到 60 秒，容器 1 和主机具

有相似的功耗模式，而容器 2 仍然处于低功耗水平。容器 2 不知道整个系统上的功率波动，因为基于功率的命名空间强制了隔离。这表明作者的系统对于隔离和划分多个容器的功耗是有效的。如果没有与权力相关的信息，攻击者将无法发起协同权力攻击。

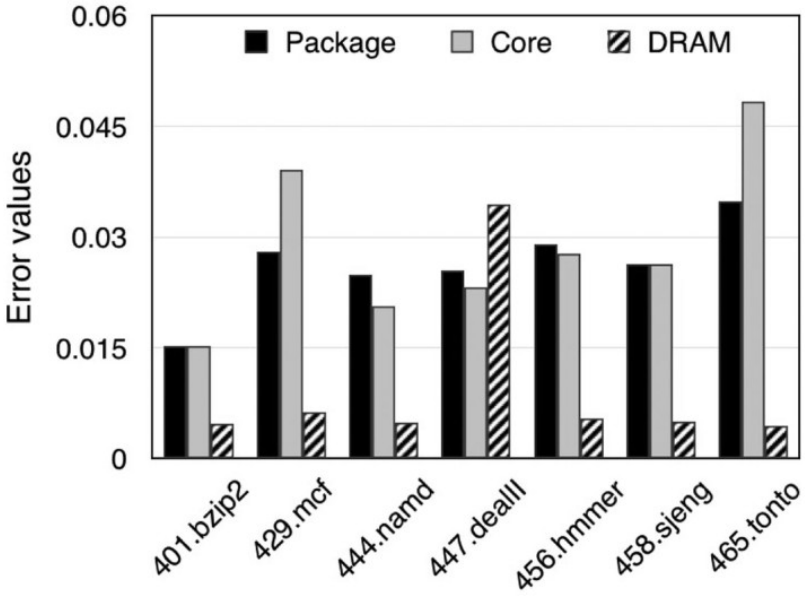


图 6: 能量建模方法的准确性

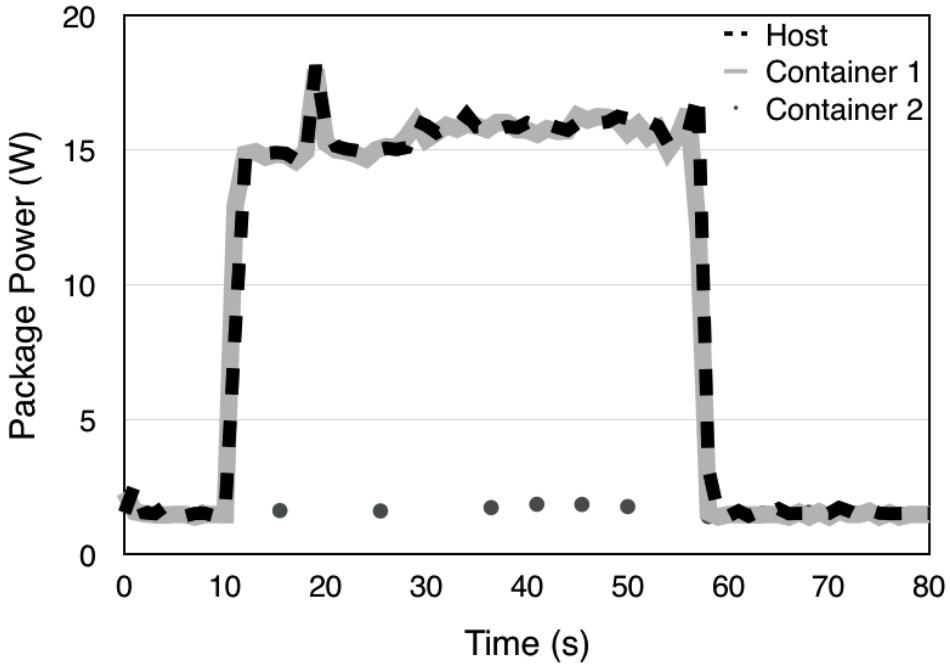


图 7: 透明度：恶意容器不知道主机的电源状况

4.3 性能

作者使用 UnixBench 比较启用系统前后的性能开销。图 8 列出了所有结果。

如结果所示，诸如 Dhrystone（测试整数和字符串操作）和 Whetstone（测试浮点运算性能）等 CPU 基准测试的开销可以忽略不计。其他基准测试（如 shell 脚本、管道吞吐量和系统调用）也很少触发开销。

在一个并行副本的情况下，基于管道的上下文切换确实会产生 61.53% 的开销，但对于 8 个并行副本，它会降低到 1.63%。作者预计组间上下文切换涉及启用/禁用性能事件监视器，而组内上下文切换不涉及任何此类开销。这可以解释为什么禁用基于电源的命名空间时，8 个并行副本可以保持类似

的性能级别。

此外，上下文切换只占整个系统性能开销的一小部分，因此对正常使用的影响很小。

在作者的系统中，对于每个新创建的容器，内核将在每个内核上创建多个 perf_events，以收集与性能相关的数据。该测量过程仅针对容器过程进行。因此，测量开销将随容器数量线性增加。然而，集装箱数量的增加对系统的影响很小。使用此机制，为了创建所有进程，内核将检查此进程是否为容器进程。这个检查过程是 Unix 基准测试中进程创建开销的主要原因。

如图 8 最后一行所示，UnixBench 的总体性能开销对于一个并行拷贝分别为 9.66%，对于 8 个并行拷贝则分别为 7.03%。作者的系统的性能在很大程度上取决于 perf_event cgroup 的实现，并且可以随着性能监控子系统的进步而提高。

	1 Parallel Copy			8 Parallel Copies		
Benchmarks	Original	Modified	Overhead	Original	Modified	Overhead
Dhrystone 2 using register variables	3,788.9	3,759.2	0.78%	19,132.9	19,149.2	0.08%
Double-Precision Whetstone	926.8	918.0	0.94%	6,630.7	6,620.6	0.15%
Execl Throughput	290.9	271.9	6.53%	7,975.2	7,298.1	8.49%
File Copy 1024 bufsize 2000 maxblocks	3,495.1	3,469.3	0.73%	3,104.9	2,659.7	14.33%
File Copy 256 bufsize 500 maxblocks	2,208.5	2,175.1	0.04%	1,982.9	1,622.2	18.19%
File Copy 4096 bufsize 8000 maxblocks	5,695.1	5,829.9	-2.34%	6,641.3	5,822.7	12.32%
Pipe Throughput	1,899.4	1,878.4	1.1%	9,507.2	9,491.1	0.16%
Pipe-based Context Switching	653.0	251.2	61.53%	5,266.7	5,180.7	1.63%
Process Creation	1416.5	1289.7	8.95%	6618.5	6063.8	8.38%
Shell Scripts (1 concurrent)	3,660.4	3,548.0	3.07%	16,909.7	16,404.2	2.98%
Shell Scripts (8 concurrent)	11,621.0	11,249.1	3.2%	15,721.1	15,589.2	0.83%
System Call Overhead	1,226.6	1,212.2	1.17%	5,689.4	5,648.1	0.72%
System Benchmarks Index Score	2,000.8	1,807.4	9.66%	7,239.8	6,813.5	7.03%

图 8: 性能测试结果

5 论文阅读心得

这篇论文的主要内容是研究了容器云中的信息泄露问题。

在阅读这篇论文之前，我对云计算与虚拟化这个领域并不是很了解。但是，通过阅读这篇论文，我对这个领域有了更深入的了解。作者在论文中提出了自己的研究问题，并通过大量的实验数据和分析，得出了结论。我觉得这篇论文非常有价值，因为它为我们提供了一种新的思路和方法来研究容器云中的信息泄露问题。

此外，我还发现作者在论文中引用了大量相关文献，这使我对这个领域的研究历史有了更全面的了解。我觉得这对于我们来说非常重要，因为它可以帮助我们更好地理解当前的研究背景和发展趋势。

尽管上过一些课程，但是我自身还是对云和容器比较陌生的，所以阅读这篇论文整体还是非常困难，有一些地方看不懂，除了对这个领域某一问题更深入的了解这一收获，最大的收获应该还是补充了相关的背景知识，比如命名空间，cgroup 和容器云。这些背景知识使得我有能力去看一些别的论文，我认为这是更关键的。

总的来说，我非常喜欢这篇论文，它为我提供了很多有价值的信息，也让我对云计算与虚拟化领域有了更深入的了解。我希望今后有机会能够继续阅读更多类似的论文，丰富自己的知识面。