DeepLog: Anomaly Detection and Diagnosis from System Logsthrough Deep Learning

## ABSTRACT

异常检测是构建安全可靠系统的关键步骤。系统日志的主要目的是在各个关键点记录系统状态和重要事件，以帮助调试系统故障并执行根本原因分析。这种日志数据几乎在所有计算机系统中都是普遍可用的。日志数据是理解系统状态和性能问题的重要且有价值的资源;因此，各种系统日志自然是在线监测和异常检测的极好信息来源。我们提出DeepLo​​g，一种利用长短期记忆（LSTM）的深度神经网络模型，将系统日志建模为自然语言序列。这允许DeepLo​​g自动从正常执行中学习日志模式，并在正常执行时检测日志模式与日志数据训练模型的偏差时检测异常。另外，我们

演示如何以在线方式逐步更新DeepLo​​g模型，以便它能够随着时间的推移适应新的日志模式。此外，DeepLo​​g从底层系统日志构建工作流，以便一旦检测到异常，用户就可以诊断检测到的异常并有效地执行根本原因分析。对大型日志数据的广泛实验评估表明，DeepLo​​g的性能优于其他现有的基于传统数据挖掘方法的基于日志的异常检测方法。

## 1 INTRODUCTION

异常检测是构建安全可靠计算机系统的重要任务。随着系统和应用程序变得越来越复杂，它们会遇到更多的漏洞和漏洞，攻击者可能利用这些漏洞和漏洞来发动攻击。此类攻击也越来越复杂。结果，异常检测变得更具挑战性，并且许多基于标准挖掘方法的传统异常检测方法不再有效。

系统日志记录各个关键点的系统状态和重要事件，以帮助调试性能问题和故障，并执行根本原因分析。这种日志数据几乎在所有计算机系统中都是普遍可用的，并且是理解系统状态的宝贵资源。此外，由于系统日志记录了主动运行过程中发生的值得注意的事件，因此它们是在线监控和异常检测的极好信息来源。

利用系统日志数据进行异常检测的现有方法可大致分为三类：基于PCA的日志消息计数器方法[39]，基于不变挖掘的方法捕获不同日志键之间的共现模式[21]，以及基于工作流的方法识别程序逻辑流程中执行异常的方法[42]。即使它们在某些情况下是成功的，但它们都不是有效的通用异常检测方法，能够以在线方式防范不同的攻击。

这项工作提出了DeepLo​​g，这是一种利用大量系统日志进行异常检测的数据驱动方法。 DeepLo​​g设计背后的关键直觉来自自然语言处理：我们将日志条目视为遵循某些模式和语法规则的序列的元素。实际上，系统日志是由遵循严格的逻辑和控制流程的程序产生的，并且非常类似于自然语言（尽管在词汇表中更加结构化和受限制）。为此，DeepLog是一个深度神经网络，使用长短期记忆（LSTM）[18]对这个日志条目序列进行建模。这允许DeepLog自动从正常执行中学习日志模式模型，并将与正常系统执行的偏差标记为异常。此外，由于它是一种学习驱动的方法，因此可以逐步更新DeepLog模型，以便它可以适应随时间出现的新日志模式。

**Challenges** 日志数据是非结构化的，它们的格式和语义可能因系统而异。即使在知道错误发生后，使用非结构化日志诊断问题已经具有挑战性[43];从海量日志数据中进行在线异常检测更具挑战性。一些现有方法使用基于规则的方法来解决该问题，这需要特定的领域知识[41]，例如，使用诸如“IP地址”之类的特征来解析日志。然而，这对于通用异常检测不起作用，其中几乎不可能先验地知道不同类型的日志中的有趣特征（以及防止不同类型的攻击）。

异常检测必须及时才能发挥作用，以便用户可以干预正在进行的攻击或系统性能问题[10]。决定将以流媒体方式进行。因此，需要对整个日志数据进行多次传递的offine方法不适用于我们的设置[22,39]。我们还希望能够检测未知类型的异常，而不是针对特定类型的异常。因此，使用正常和异常（对于特定类型的异常）日志数据条目来训练二进制分类器以进行异常检测的先前工作[44]在该上下文中是无用的。

另一个挑战来自并发。显然，日志中的日志消息的顺序提供了用于诊断和分析的重要信息（例如，识别程序的执行路径）。但是，在许多系统日志中，日志消息由多个不同的线程或同时运行的任务生成。这种并发性使得难以应用基于工作流的异常检测方法[42]，其使用单个任务的工作流模型作为生成模型来匹配日志消息序列。

最后，每条日志消息都包含丰富的信息，例如日志密钥和一个或多个度量标准值，以及它的时间戳。整合和利用这些不同信息的整体方法将更有效。大多数现有方法[22,32,39,41,42,44]仅分析日志消息的一个特定部分（例如，日志密钥），其限制了它们可以检测的异常类型。

**Our contribution.** 递归神经网络（RNN）是一种人工神经网络，它使用循环将最后状态的输出转发到当前输入，从而跟踪进行预测的历史记录。长短期记忆（LSTM）网络[13,18,27]是能够记住序列上的长期依赖性的RNN的实例。 LSTM已经在各种任务中取得了成功，例如机器翻译[35]，情感分析[8]和医学自我诊断[20]。

受到观察到系统日志中的条目是由结构化源代码的执行产生的一系列事件（因此可以被视为结构化语言）的启发，我们使用LSTM神经网络设计DeepLog框架以进行在线异常检测系统日志。 DeepLog不仅使用日志键而且还使用日志条目中的度量值进行异常检测，因此，它能够捕获不同类型的异常。 DeepLog仅依赖于由一系列“正常日志条目”组成的小型训练数据集。在训练阶段之后，DeepLog可以识别正常的日志序列，并且可以用于以流方式对传入的日志条目进行在线异常检测。

直观地，DeepLog隐含地捕获来自训练数据的日志条目中与正常系统执行路径相对应的潜在非线性和高维依赖性。为了帮助用户在识别出异常后诊断问题，DeepLog还会在其培训阶段根据日志条目构建工作流模型。 DeepLo​​g将并发任务或线程生成的日志条目分成不同的序列，以便为每个单独的任务构建工作流模型。

我们的评估显示，在以前的工作[22,39]探索的大型HDFS日志数据集上，只对非常小的一小部分（少于1％）的日志条目进行了训练，对应于正常的系统执行，DeepLo​​g可以实现几乎100％的检测精度其余99％的日志条目。大型OpenStack日志的结果传达了类似的趋势。此外，DeepLo​​g还提供了在检测阶段通过合并实时用户反馈来逐步更新其权重的功能。更具体地说，如果正常日志条目被错误地归类为异常，则DeepLo​​g提供用户反馈机制。然后，DeepLo​​g可以使用此类反馈随时间动态调整其权重，以使其自身适应新系统执行（因此，新日志）模式。

2 PRELIMINARIES2.1 Log parser

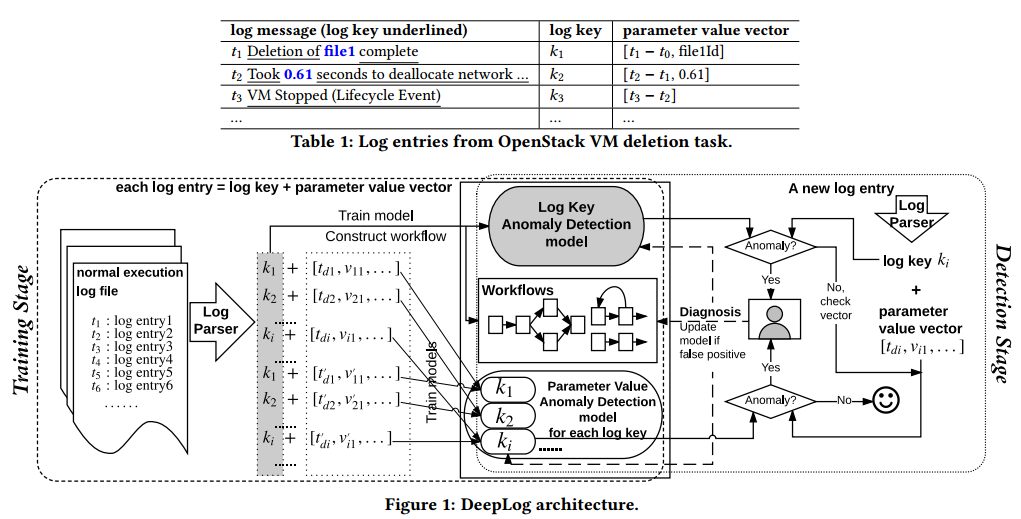
我们首先将非结构化的自由文本日志条目解析为结构化表示，以便我们可以学习此结构化数据的顺序模型。如前几个工作[9,22,39,42,45]所示，一种有效的方法是从每个日志条目中提取“日志密钥”（也称为“消息类型”）。日志条目e的日志键指的是源代码中print语句的字符串常量k，该代码在执行该代码期间打印e。例如，日志条目的日志键k =“构建实例的时间为10秒。”是构建实例的k = Took \*秒。这是来自print语句printf的字符串常量（“Took％f秒to build”例如。“，t）。请注意，参数在日志键中抽象为星号。这些度量标准值反映了基础系统状态和性能状态。某些参数的值可以用作特定执行序列的标识符，例如HDFS日志中的block\_id和OpenStack日志中的instance\_id。这些标识符可以将日志条目组合在一起，或者将并发进程生成的日志条目解开，以分离单线程顺序序列[22,39,42,45]。最先进的日志解析方法由Spell [9]表示，Spell [9]是一种无监督的流解析器，它基于LCS（最长公共子序列）的思想以在线方式解析传入的日志条目。

过去的日志分析工作[22,39,42,44]已经丢弃了日志条目中的时间戳和/或参数值，并且只使用了日志密钥来检测异常。 DeepLo​​g将每个日志条目e的参数值以及e与其前任之间经过的时间存储到向量中 - ！v e。除了日志密钥之外，DeepLo​​g还使用此向量。表1给出了一个示例，其中显示了OpenStack中多轮执行虚拟机（VM）删除任务的一系列日志条目的解析结果。

### 2.2 DeepLog architecture and overview

DeepLo​​g的体系结构如图1所示，包含三个主要组件：日志密钥异常检测模型，参数值异常检测模型和诊断检测到的异常的工作流模型。

**Training stage.** DeepLo​​g的训练数据是来自正常系统执行路径的日志条目。每个日志条目都被解析为日志密钥和参数值向量。 DeepLo​​g使用从训练日志文件解析的日志键序列来训练日志密钥异常检测模型，并构建用于诊断目的的系统执行工作流模型。对于每个不同的密钥k，DeepLo​​g还训练和维护用于检测系统性能异常的模型，由这些度量值反映，由k的参数值矢量序列训练。



**Detection stage.** 将新到达的日志条目解析为日志密钥和参数值向量。 DeepLo​​g首先使用日志密钥异常检测模型来检查传入的日志密钥是否正常。如果是，则DeepLo​​g使用该日志密钥的参数值异常检测模型进一步检查参数值向量。如果预测其日志键或其参数值向量异常，则新条目将被标记为异常。最后，如果标记为异常，DeepLo​​g的工作流模型为用户提供诊断异常的语义信息。执行模式可能随时间而变化，或者不包括在原始训练数据中。 DeepLo​​g还提供收集用户反馈的选项。如果用户将检测到的异常报告为误报，则DeepLo​​g可以将其用作标记记录，以逐步更新其模型以合并并适应新模式。

### 2.3 Threat model

DeepLo​​g了解由正常系统执行路径生成的一系列日志条目中嵌入的全面且复杂的关联和模式。从此以后，我们假设系统日志本身是安全的并受到保护，并且攻击者无法攻击日志本身的完整性。我们还假设攻击者无法修改系统源代码以更改其日志记录行为和模式。也就是说，从广义上讲，我们考虑了两种类型的攻击。

（1）导致系统执行不当行为的攻击，从而导致系统日志中出现异常模式。例如，拒绝服务（DoS）攻击可能导致执行缓慢，因此性能异常反映在与参数值矢量序列的日志时间戳差异中;导致重复服务器重启的攻击，例如盲目返回定向编程（BROP）攻击[5]显示为服务器重启日志密钥过多;以及任何可能导致任务中止的攻击，以便相应的日志序列提前结束和/或出现异常日志条目。

（2）由于系统监视服务的日志记录活动而可能在系统日志中留下痕迹的攻击。一个例子是入侵检测系统（IDS）记录的可疑活动。

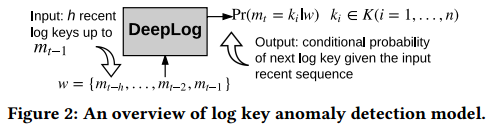
3 ANOMALY DETECTION3.1 Execution path anomaly

我们首先描述如何使用日志键序列检测执行路径异常。由于源代码中不同的打印语句（打印日志条目）的总数是常量，因此不同的日志键的总数也是不变的。设K = {k1; k2; :::; kn}是来自日志生成系统源代码的不同日志键的集合。

将日志条目解析为日志密钥后，日志密钥序列将反映导致日志打印语句的特定执行顺序的执行路径。令mi表示日志键序列中位置i处的键的值。显然，mi可以从K中取n个可能的密钥中的一个，并且强烈依赖于mi之前出现的最新密钥。

我们可以将日志键序列中的异常检测建模为多类分类问题，其中每个不同的日志键定义一个类。我们将DeepLo​​g训练为最近上下文中的多类分类器。输入是最近日志密钥的历史，输出是来自K的n个日志密钥的概率分布，表示序列中的下一个日志密钥是密钥ki 2 K的概率。

图2总结了分类设置。假设t是要出现的下一个日志键的序列ID。分类输入是h最新日志键的窗口w。也就是说，w = {mt-h; :::; mt-2; mt-1}，其中每个mi在K中，并且是来自日志条目ei的日志密钥。请注意，相同的日志键值可能会以w出现多次。训练阶段的输出是每个ki 2 K（i = 1;：::; n）的条件概率分布Pr [mt = ki | w]的模型。检测阶段使用该模型进行预测，并将预测输出与实际出现的观察日志密钥值进行比较。



**Training stage.** 训练阶段依赖于正常执行底层系统产生的一小部分日志条目。对于训练数据中长度为h的每个日志序列，DeepLo​​g更新其模型，以获得具有ki 2 K作为下一个日志密钥值的概率分布。例如，假设正常执行产生的小日志文件被解析为一系列日志密钥：{k22，k5，k11，k9，k11，k26}。给定窗口大小h = 3，用于训练DeepLo​​g的输入序列和输出标签对将是：{k22，k5，k11-k9}，{k5，k11，k9-k11}，{k11，k9，k11 - k26}。

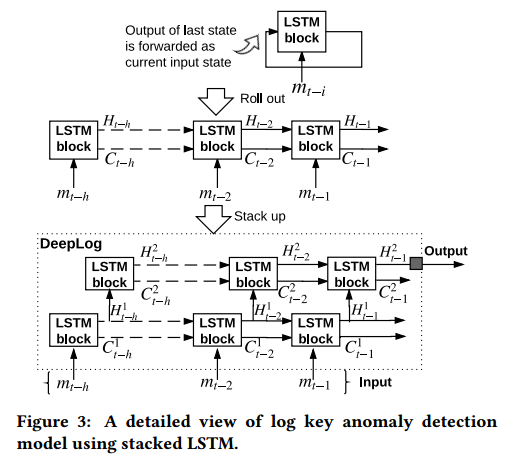
**Detection stage.** DeepLo​​g在在线流媒体设置中执行异常检测。为了测试传入的日志密钥mt（从传入的日志条目et解析）是否被认为是正常的或异常的，我们将w = {mt-h，::：; mt-1}作为其输入发送到DeepLo​​g。输出是概率分布Pr [mt | w] = {k1：p1，k2：p2; :::; kn：pn}描述从K开始的每个日志键的概率，作为给定历史记录的下一个日志键值。

实际上，多个日志键值可能显示为mt。例如，如果系统尝试连接到主机，则mt可以是“等待\*响应”或“连接到\*”;两者都是正常的系统行为。 DeepLo​​g必须能够在培训期间学习这些模式。我们的策略是根据它们的概率Pr [mt | w]对可能的对数密钥K进行排序，并将密钥值视为正常，如果它是最佳候选者之一。否则，日志密钥被标记为来自异常执行。

3.1.1传统的N-gram语言模型。将概率归因于从固定词汇表中提取的单词序列的问题是语言建模的经典问题，由自然语言处理（NLP）社区广泛研究[24]。在我们的例子中，每个日志密钥可以被视为从词汇表K中获取的单词。用于将概率分配给任意长序列的典型语言建模方法是N-gram模型。直觉是序列中的特定单词仅受其最近的前辈而不是整个历史的影响。在我们的设置中，这个近似值相当于设置P（mt = ki | m1; :::; mt-1）= Pr（mt = ki | mt-N; :::; mt-1）其中N表示长度最近的历史需要考虑。

对于训练，我们可以使用来自大型语料库的相对频率计数来计算该概率，以给出最大似然估计。给定一长串密钥{m1; m2; :::; mt}，我们可以使用{mt-N的相对频率计数来估计观察第i个密钥ki的概率; :::; mt-1; mt = ki}关于序列{mt-N; :::; mt-1}。换句话说，Pr（mt = ki | m1; :::; mt-1）= count（mt-N，：::，mt-1，mt = ki）/ count（mt-N，:::， MT-1）。请注意，我们将使用大小为N的滑动窗口在整个键序列上计算这些频率。

要在我们的设置中应用N-gram模型，我们只需使用N作为历史窗口大小，即，当使用N-gram模型时，我们在实验中设置h = N，其中h是历史滑动窗口大小，如图2.我们将其用作基线方法。



3.1.2 LSTM方法。近年来，使用递归神经网络的神经语言模型已被证明在各种NLP任务中非常有效[3,25]。与N-gram语言模型相比，基于LSTM的模型可以编码更复杂的模式并维持序列上的长程状态[34]。来自系统日志中的并发任务的复杂模式和交错日志条目可以使传统语言模型的效率降低。因此，DeepLo​​g使用LSTM神经网络[18]从日志密钥序列进行异常检测。

给定一系列日志密钥，训练LSTM网络以最大化将ki 2 K作为训练数据序列所反映的下一个日志密钥值的概率。换句话说，它学习概率分布Pr（mt = ki | mt-h，：::，mt-2，mt-1），其最大化训练日志密钥序列的概率。

图3说明了我们的设计。图的顶部显示了一个LSTM块，它反映了LSTM的重复性质。每个LSTM块都会记住其输入的状态，作为固定维度的向量。来自前一时间步骤的LSTM块的状态也与其（外部）数据输入（在该特定示例中为mt-i）一起被馈送到其下一个输入，以计算新状态和输出。这是历史信息传递到单个LSTM块并在其中维护的方式。

一系列LSTM块在一层中形成递归模型的展开版本，如图3的中心所示。每个单元保持隐藏向量Ht-i和单元状态向量Ct-i。两者都传递到下一个块以初始化其状态。在我们的例子中，我们从输入序列w（h log键的窗口）为每个日志键使用一个LSTM块。因此，单层由h个展开的LSTM块组成。

在单个LSTM块内，输入（例如mt -i）和先前输出（Ht -i -1）用于决定（1）先前单元状态Ct -i -1保留多少状态 Ct -i，（2）如何使用当前输入和前一个输出来影响状态，以及（3）如何构造输出Ht -i。 这是通过使用一组选通函数来完成的，以通过控制保持输入和先前输出的信息量来确定状态动态，并且信息流进入下一步骤。 每个门控功能由要学习的一组权重参数化。 LSTM块的表达能力由存储器单元的数量（即隐藏状态向量H的维度）确定。 由于空间限制，我们将读者引用到NLP引物（例如，[12]）以进行LSTM的形式表征。

训练步骤需要找到权重的正确分配，以便LSTM序列的最终输出产生训练数据集中输入所附带的所需标签（输出）。在训练过程中，每个输入/输出对通过梯度衰减通过损耗最小化逐步更新这些权重。在DeepLo​​g中，输入由h日志键的窗口w组成，输出是紧跟在w之后的日志键值。我们使用分类交叉熵损失进行训练。

训练完成后，我们可以使用一层h LSTM块预测输入的输出（w = {mt -h，...，mt -1}）。 w中的每个日志键都馈送到该层中的相应LSTM块。

如果我们堆叠多个层并使用前一层的隐藏状态作为下一层中每个相应LSTM块的输入，它将成为一个深LSTM神经网络，如图3底部所示。为简单起见，它省略了由标准编码 - 解码方案构造的输入层和输出层。输入层将来自K的n个可能的对数密钥编码为单热矢量。在图1中，为对数密钥ki∈K构造稀疏的n维向量→-u i，使得对于所有其他j \* i，→-u i [i] = 1并且→-u i [j] = 0。输出层使用标准多项逻辑函数将最终隐藏状态转换为概率分布函数，以表示每个ki∈K的Pr [mt = ki | w]。

图3底部的示例仅显示了两个隐藏层，但可以使用更多层。

### 3.2 Parameter value and performance anomaly

日志键序列对于检测执行路径异常非常有用。但是，一些异常不会显示为偏离正常执行路径，而是显示为不规则参数值。这些参数值向量（对于相同的对数密钥）形成参数值向量序列，并且来自不同对数密钥的这些序列形成对于性能监视和异常检测很重要的多维特征空间。

基线方法。一种简单的方法是将所有参数值向量序列存储到矩阵中，其中每列是来自日志键k的参数值序列（注意，根据其参数值向量的大小，可能有多个列用于k） 。该矩阵中的行i表示时间实例ti。

以表1中的日志条目为例。在该示例中存在3个不同的对数键值，并且它们的参数值向量的大小分别为2,2和1。因此，该矩阵中的行1表示时间实例t1，其值为[t1-t0，file1Id，null，null，null]。类似地，第2行和第3行分别是[null，null，t2-t1,0.61，null]和[null，null，null，null，t3-t2]。

我们还可以要求每一行表示一系列时间实例，以便每行对应于该时间范围内的多个日志消息，并且变得不那么稀疏。但是当存在许多日志键值和/或存在一些大的参数值向量时，矩阵仍将非常稀疏。此外，这种方法对异常检测过程引入了延迟，并且也难以确定每个范围的长度的良好值。

给定该矩阵，可以应用许多众所周知的数据驱动的异常检测方法，例如主成分分析（PCA）和自组织映射（SOM）。它们对于捕获不同特征维度之间的相关性很有用。然而，在日志数据的上下文中该方法的主要限制是通常在特定时间实例处出现多个日志键的可能性相同。例如，由于同时运行的任务，表1中的k1和k2的顺序是任意的。这种现象以及矩阵稀疏的事实使得这些技术在我们的环境中无效。最后，他们无法模拟参数值矢量序列中存在的自相关（在单个矢量序列中随时间变化的规则模式）。

**Our approach.** DeepLo​​g通过查看每个参数值向量序列（对于日志密钥）作为单独的时间序列来训练参数值异常检测模型。考虑表1中的示例。参数值矢量序列k2的时间序列为：{[t2-t1,0.61]，[t，-t ,, 1]}。因此，我们的问题从多变量时间序列数据减少到异常检测。可以再次应用基于LSTM的方法。我们使用类似的LSTM网络（如图3所示）对多变量时间序列数据建模，并进行以下调整。注意，为每个不同的日志键值的参数值向量序列构建单独的LSTM网络。

输入。每个时间步的输入只是该时间戳的参数值向量。我们将每个矢量中的值标准化为来自训练数据的相同参数位置的所有值的平均值和标准偏差。

输出。输出是实值矢量，作为下一个参数值矢量的预测，基于来自最近历史的参数值矢量序列。

培养目标功能。对于多变量时间序列数据，训练过程尝试调整其LSTM模型的权重，以便最小化预测和观察到的参数值向量之间的误差。 1us，均方损失用于最小化训练过程中的误差。

异常检测。通过均方误差（MSE）测量预测和观察到的参数值矢量之间的差异。我们不是以ad-hoc方式为异常检测目的设置魔术错误阈值，而是将训练数据划分为两个子集：模型训练集和验证集。对于验证集中的每个向量→-u，我们应用训练集生成的模型来计算预测之间的MSE（使用之前的矢量序列→验证集中的-u）和→-u。在每个时间步，预测矢量与验证组中的实际误差之间的误差被建模为高斯分布。

在部署时，如果预测和观测值向量之间的误差在上述高斯分布的高置信区间内，则输入日志条目的参数值向量被认为是正常的，否则被认为是异常的。

由于日志消息中的参数值通常记录重要的系统状态度量，因此该方法能够检测各种类型的性能异常。例如，性能异常可能反映为“减速”。回想一下，DeepLo​​g在每个参数值向量中存储连续日志条目之间经过的时间。上述LSTM模型通过将参数值向量建模为多变量时间序列，能够在该时间序列中检测一个或多个维度中的异常模式;经过时间值只是一个这样的维度。

### 3.3 Online update of anomaly detection models

显然，训练数据可能无法涵盖所有​​可能的正常执行模式。系统行为可能随时间而变化，另外还取决于工作负载和数据特征。因此，DeepLo​​g必须逐步更新其LSTM模型中的权重，以合并和适应新的日志模式。为此，DeepLo​​g为用户提供了一种提供反馈的机制。这允许DeepLo​​g使用误报来调整其权重。例如，假设h = 3并且最近的历史序列是{k1，k2，k3}，并且DeepLo​​g已经将下一个日志密钥预测为具有概率1的k1，而下一个日志密钥值是k2，这将是标记为异常。如果用户报告这是误报，则DeepLo​​g能够使用以下输入 - 输出对{k1，k2，k3→k2}来更新其模型的权重以了解此新模式。因此，下次给定历史序列{k1，k2，k3}时，DeepLo​​g可以输出具有更新概率的k1和k2。相同的更新过程适用于参数值异常检测模型。请注意，DeepLo​​g不需要从头开始重新训练。在初始训练过程之后，DeepLo​​g中的模型作为几个多维权重向量存在。更新过程提供新的训练数据，并调整权重以最小化模型输出与误报情况下的实际观察值之间的误差。

## 4 WORKFLOW CONSTRUCTION FROM MULTI-TASKS EXECUTION

每个日志键都是在源代码中执行日志打印语句，而像VM创建这样的任务将生成一系列日志条目。直观地，任务产生的日志条目的顺序表示用于完成该任务的每个功能的执行顺序。因此，我们可以将工作流模型构建为有限状态自动机（FSA），以捕获任何任务的执行路径。此工作流模型还可用于检测执行路径异常，但与DeepLo​​g的LSTM模型相比，由于无法捕获任务间依赖性和非确定性循环迭代，因此效率较低。但是，工作流模型对于使用户能够在检测到异常时诊断执行任务时出错的内容非常有用。

给定由任务的重复执行产生的日志序列，已经有几个工作探索工作流推理的问题[4,21,42]。 CloudSeer [42]表示使用工作流模型进行异常检测的现有技术。 CloudSeer有几个局限性。首先，它可以检测到的异常仅限于具有“ERROR”日志记录级别和日志条目未出现的日志条目。此外，其工作流模型构造需要一个日志文件，只能重复执行一个任务。关于来自日志文件的工作流构建的其他先前的工作[4,21]也受到这种限制。实际上，日志文件通常包含由多个任务生成的交错日志条目，并且可能在任务中同时运行线程。

### 4.1 Log entry separation from multiple tasks

一个简单的情况是多个程序同时写入同一个日志（例如，Ubuntu的系统日志）。通常，每个日志条目都包含创建它的程序的名称。另一个简单的情况是当进程或任务ID包含在日志条目中时。在这里，我们关注重复执行用户程序以在该程序中执行不同但逻辑相关的任务的情况。一个重要的观察是任务不会在时间上重叠。但是，相同的日志键可能出现在多个任务中，并且每个任务内的并发性是可能的（例如，一个任务中的多个线程）。

以OpenStack管理日志为例。对于每个VM实例，其生命周期包含VM创建，VM停止，VM删除等。这些任务不重叠，即VM停止只能在VM创建完成后启动。但是，相同的日志键可能出现在不同的任务中。例如，日志消息“VM Resumed（Lifecycle Event）”可能出现在VM创建，VM启动，VM恢复和VM unause中。在每个任务中可能同时运行线程，导致对应于一个任务的日志消息的排序的不确定性。例如，在VM创建期间，两个日志消息“Took \* seconds to build instance”和“VM Resumed（Lifecycle Event）”的顺序是不确定的。

我们的目标是在日志文件中分离不同任务的日志条目，然后根据其日志键序列为每个任务构建工作流模型。 1at说，我们问题的输入是从原始日志文件解析的整个日志键序列，输出是一组工作流模型，每个任务标识一个。

### 4.2 Using DeepLog’s anomaly detection model

4.2.1记录密钥分离。回想一下，在DeepLo​​g的日志键异常检测模型中，输入是一段长度为h的记录键，来自最近的历史记录，输出是所有可能的日志键值的概率分布。一个有趣的观察是它的输出实际上编码了底层的工作流执行路径。

直观地，给定一个日志键序列，我们的模型根据它在训练阶段观察到的执行模式预测接下来会发生什么。如果序列w在训练阶段中从不跟随特定键值k，则Pr [mt = k | w] = 0.相应地，如果序列w总是跟随k，则Pr [mt = k | w ] = 1.例如，假设在序列“25→54”上，输出预测为“{57：1.00}”，我们知道“25→54→57”来自一个任务。更复杂的情况是当序列w后面跟着来自一组不同键的日志键值时;这些键在w总和为1之后出现的概率。

为了处理这种情况，我们使用了一个受小型不变量挖掘[21]启发的想法。

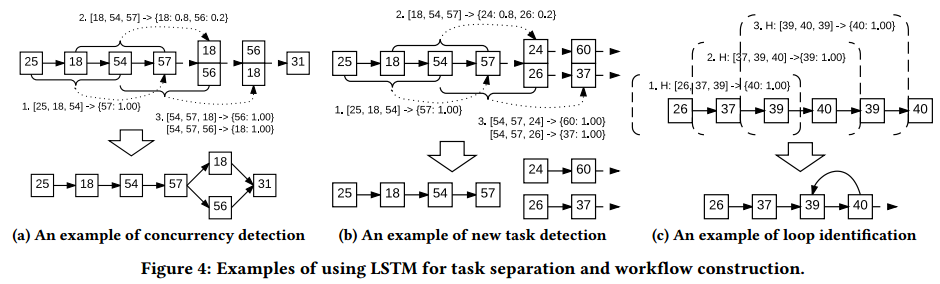
考虑对数序列“54→57”，并假设预测的概率分布是“{18：0.8,56：0.2}”，这意味着下一步可以是“18”或“56”。这种模糊性可能是由于使用不足的历史序列长度引起的。例如，如果两个任务共享相同的工作流程段“54→57”，则第一个任务具有“18→54→57→18”模式，该模式在80％的时间执行，第二个任务具有模式“31 →54→57→56“执行20％的时间。这将导致一个模型，根据序列“54→57”预测“{18：0.8,56：0.2}”。

我们可以通过训练具有不同历史序列长度的模型来解决该问题，例如，在这种情况下使用h = 3而不是h = 2。在工作流程构建期间，我们使用日志序列长度，这导致更确定的预测，例如，在上面的例子中，序列“18→54→57”将导致预测{18：1.00}，序列“31→54→57”将导致预测{56：1.00}。如果我们排除了一个小序列是来自不同任务的共享段（即，增加训练和预测的序列长度不会导致更确定的预测），那么现在的挑战是找出是否多密钥预测输出是由同一任务中的并发或不同任务的开始引起的。我们称之为分歧点。

我们观察到，如图4a所示，如果分歧点是由同一任务中的并发引起的，一个常见的模式是预测输出中具有最高概率的密钥将一个接一个地出现，并且确定性（由更高的由于某些并发线程的密钥已经出现，因此后续预测的密钥数量较少的概率将会增加。在并发线程的所有密钥都包含在历史序列中之后，预测最终将变得确定。

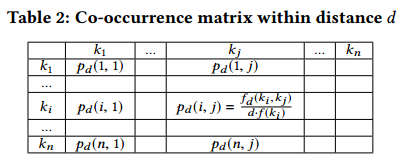
另一方面，如果发散点是由新任务的开始引起的，如图4b所示，预测的对数键候选（示例中的“24”和“26”）将不会一个接一个地出现。如果我们将每个这样的日志密钥合并到历史序列中，则下一个预测是新日志密钥的确定性预测（例如，“24→60”，“26→37”）。如果是这种情况，我们将停止增加当前任务的工作流模型（在此示例中停在键“57”），并开始为新任务构建工作流模型。注意，图4b中的两个“新任务”也可以是“if-else”分支，例如“57→if（24→60→...）else（26→37→...）”。为了处理这种情况，我们应用一个简单的启发式方法：如果“新任务”只有很少的日志键（例如3）并且总是出现在特定任务Tp之后，我们将其视为“if-else”分支的一部分。 Tp，否则作为新任务。

4.2.2构建工作流模型。一旦我们能够区分由同一任务和新任务中的并发（多线程）引起的分歧点，我们就可以轻松地构建工作流模型，如图4a和图4b所示。需要额外注意识别循环。检测循环实际上非常简单。循环始终在初始工作流模型中显示为展开链;参见图4c的示例。虽然此工作流链最初是“26→37→39→40→39→40”，但我们可以将重复的片段识别为循环执行（本例中为39→40）。



### 4.3 Using density-based clustering approach

4.3.1日志密钥分离。另一种方法是使用基于密度的聚类技术。直觉是同一任务中的日志键总是一起出现，但是来自不同任务的日志键可能并不总是一起出现，因为在多次执行不同任务期间任务的排序没有得到修复。这允许我们基于共现模式来聚类日志密钥，并且当共现率低时将密钥分离成不同的任务。



在日志键序列中，任意两个日志键之间的距离d被定义为它们之间的日志键数加1.例如，给定序列{k1，k2，k2}，d（k1，k2）= [1 ，2，d（k2，k2）= 1（注意，对（k1，k2）之间有两个距离值）。

我们构建如图2所示的共生矩阵，其中每个元素pd（i，j）表示在输入序列中出现在距离d内的两个对数键ki和kj的概率。具体地，令f（ki）是输入序列中的ki的频率，并且fd（ki，kj）是在输入序列中在距离d内一起出现的对（ki，kj）的频率。我们定义p（i，j）= fd（ki，kj），它表示kj对ki的重要性。

例如，当d = 1时，p1（i，j）= f（ki）= 1意味着对于ki的每次出现，其旁边必须有kj。注意，在该定义中，分母中的f（ki）被缩放d，因为在计算d内的共现频率时，密钥ki被计数d次。将f（ki）缩放d倍可确保任何i的In fd（i，j）= 1。注意，我们可以为d的不同距离值构建多个共现矩阵。

对于我们构建的每个距离值d的共生矩阵，我们的目标是输出一组任务TASK =（T1，T2，...）。聚类过程的工作原理如下。首先，对于d = 1，我们检查是否有任何p1（i，j）大于阈值τ（比如τ= 0.9），当它确实时，我们将ki，kj连接在一起形成T1 = [ki，kj]。接下来，我们递归检查T1是否可以从其头部或尾部延伸。例如，如果存在kx∈K使得p1（ki，kx）>τ，则我们进一步检查p2（kj，kx）>τ，即，kj和kx是否在距离2内具有大的共现概率。如果是，则T1 = [kx，ki，kj]，否则我们将T2 = [ki，kx]添加到TASK。

此过程继续，直到TASK中的任务T无法进一步扩展。在一般情况下，当要扩展的任务T具有多于2个对数键时，在检查是否可以包括kx作为新的头部或尾部时，我们需要检查kx是否与每个日志具有大于τ的共现概率密钥在T到距离d ,,其中d是较小的：i）T的长度，以及ii）我们为其建立共生矩阵的d的最大值。例如，要检查T = [k1，k2，k3]是否应该在其尾部连接k4，我们需要检查是否为min（p1（k3，k4），p2（k2，k4），p3（k1，k4）） >τ。

上述过程连接每个任务的顺序日志键。当任务T1 = [ki，kj]无法扩展到包含任何单键时，我们检查T1是否可以通过两个对数键扩展，即如果存在kx，kν∈K，那么p1（ki， kx）+ p1（ki，kv）>τ，或p1（kj，kx）+ p1（kj，kv）>τ。假设后一种情况为真，接下来要检查的是kx和kv是否是任务T1中的当前线程产生的日志密钥。如果它们是，则pd（kj，kx）总是随着较大的d值而增加，即p2（kj，kx）> p1（kj，kx），这是直观的，因为来自并发线程的密钥的出现顺序不确定。否则kx和kv不属于T1，因此我们将T2 = [kj，kx]和T3 = [kj，kv]添加到TASK中。

最后，对于TASK中的每个任务T，如果其序列作为子序列包含在另一个任务中，则我们消除T.

4.3.2构建工作流模型。一旦为每个任务分离并识别了一个日志键序列，任务的工作流模型构建遵循第4.2.2节中的相同讨论。

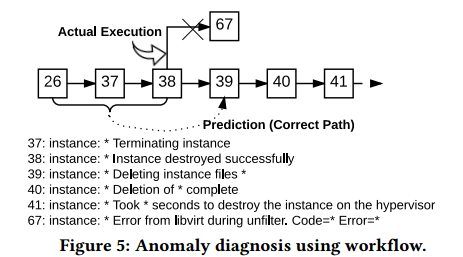
### 4.4 Using the workflow model

4.4.1设置DeepLo​​g模型的参数。在3.1节中，我们已经证明DeepLo​​g需要几个输入参数，特别是它需要历史序列窗口h的长度（用于训练和检测），以及预测输出概率分布中的前g个对数键的数量功能被认为是正常的。

为h和g设置适当的值取决于问题。一般来说，较大的h值将提高预测精度，因为在LSTM中使用了更多的历史信息，直到它达到历史上远处的键不会有助于预测出现的键的点。此时继续增加h并不会损害LSTM的预测准确性，因为LSTM能够知道只有长序列中的近期历史因此忽略了长尾。但是，大的h值确实会对性能产生影响。训练和预测都需要更多的计算（和层），这会降低DeepLo​​g的性能。另一方面，g的值调节了真阳性（异常检测率）和假阳性（误报率）之间的权衡。

工作流模型提供了为h和g设置适当值的指导。直观地说，h需要足够大以包含必要的依赖关系以进行良好的预测，因此我们可以将h设置为最短工作流的长度。可能的执行路径数表示g的良好值，因此，我们将g设置为所有任务的工作流中所有分歧点的最大分支数。

4.4.2使用工作流程诊断检测到的异常。每当DeepLo​​g检测到异常时，工作流模型都可用于帮助诊断此异常并了解其发生的原因和原因。图5显示了一个示例。使用历史序列[26,37,38]，来自DeepLo​​g的最高预测是日志密钥39（假设g = 1），但是出现的实际日志密钥是67，这是异常。借助此任务的工作流模型，用户可以轻松识别相应工作流中的当前执行点，并进一步发现此错误发生在“实例已成功销毁”之后和“删除实例文件\*”之前，这意味着在销毁VM后清理期间发生此错误。



### 4.5 Discussion

以前的作品[4,21,42]专注于从一个任务的多次执行构建工作流程。他们的方法的基本思想遵循3个步骤：1）挖掘每对日志密钥的时间依赖性; 2）从步骤1中确定的成对不变量构建基本工作流程; 3）使用输入日志键序列细化工作流模型。一个主要的限制是它们无法使用包含多个任务或一个任务中的并发线程的日志序列，这是我们的研究所解决的。我们的任务分离方法还为每项任务的工作流程构建提供了有用的见解。

## 5 EVALUATION

DeepLo​​g是使用Keras [6]和TensorFlow [2]作为后端实现的。在本节中，我们将展示每个组件的评估以及DeepLo​​g的整体性能，以显示其在查找大型系统日志数据异常方面的有效性。

### 5.1 Execution path anomaly detection

本节重点介绍如何评估DeepLo​​g中的日志密钥异常检测模型。我们首先将其在大型系统日志上的有效性与以前的方法进行比较，然后研究DeepLo​​g中不同参数的影响。

5.1.1以前的方法。以前关于通用日志异常检测的工作遵循类似的过程：它们首先从每个日志消息中提取日志密钥，然后对日志密钥序列执行异常检测。

主成分分析（PCA）方法[39]假设日志文件中有不同的“会话”，可以通过附加到每个日志条目的会话ID轻松识别。它首先按会话对日志键进行分组，然后计算每个会话中每个日志键值的出现次数。会话向量的大小为n，表示该会话中K中每个日志键的出现次数。形成矩阵，其中每列是日志键，并且每行是一个会话向量。 PCA通过测量变换坐标系的残余子空间上的投影长度来检测异常矢量（会话）。这种方法比其在线对应的在线PCA [38]更有效，特别是在减少误报方面，但这显然是一种异常方法，不能用于在线异常检测。实施是[17]开源的。

Invariant Mining（IM）[22]构造了与PCA方法相同的矩阵。 IM首先挖掘大多数向量可以满足的小不变量，然后将那些不满足这些不变量的向量视为异常执行会话。这种方法比使用工作流自动机的早期工作[11]更有效。实施是[17]开源的。

TFIDF是在[44]中开发的。虽然其目标是IT系统故障预测，但与[39]中所示的异常检测不同。尽管如此，我们仍然将此方法包括在我们的评估中，因为它还使用基于LSTM的方法。有几个关键的区别。 TFIDF按时间窗口对日志密钥进行分组（每个时间窗口由用户参数定义），然后使用TF-IDF（术语 - 频率，逆文档频率）向量对每个时间窗口（称为“纪元”）进行建模。它使用的拉普拉斯平滑程序需要了解时期总数（因此需要整个日志文件）。 TFIDF构建一个LSTM模型作为二进制类，它需要标记正常和异常数据进行训练。不仅难以获得异常日志条目，而且还可能检测不到训练数据中未包括的新类型的异常。相比之下，DeepLo​​g将其LSTM模型训练为多类分类器，并且只需要训练正常数据。

CloudSeer是专为多用户Open-Stack日志设计的方法[42]。它为每个与OpenStack VM相关的任务构建工作流模型，并使用该工作流进行异常检测。尽管它在OpenStack日志上达到了可接受的性能（精度为83.08％，并且报告的召回率为90.00％），但此方法不适用于其他类型的日志（例如，HDFS日志），其中日志键的模式是更不规则。例如，Cloud-Seer仅模拟在每个会话中“出现相同次数”的日志密钥。在HDFS日志中，29个日志密钥中只有3个满足此标准。此外，此方法不能将一个日志中不同任务的日志条目分成单独的序列。它依赖于多个标识符来实现这一点，这对于通用日志并不总是可行的。因此，它不与这里进行比较。

*5.1.2 Log data sets and set up*

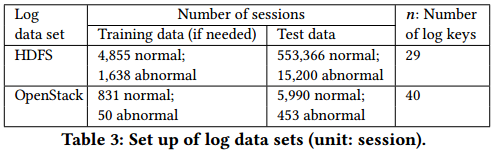
HDFS日志数据集。它是通过在200多个Amazon的EC2节点上运行基于Hadoop的map-reduce作业生成的，并由Hadoop域专家提供。在收集的11,197,954个日志条目中，约2.9％是异常的，包括诸如“写异常”之类的事件。这是基于PCA的基于[39]方法首先使用的主要数据集，随后被其他几项工作使用，包括在线PCA [38]和基于IM的[22]方法。该数据集的详细信息可以在[38,39]中找到。

OpenStack日志数据集。我们在CloudLab [30]上部署了OpenStack实验（版本Mitaka），其中包含一个控制节点，一个网络节点和八个计算节点。在收集的1,333,318个日志条目中，约7％是异常的。正在运行脚本以不断执行与VM相关的任务，包括VM创建/删除，停止/启动，暂停/取消暂停和暂停/恢复。 VM任务使用正则表达式模式进行调度（创建（停止启动）{0,3}（暂停取消暂停）{0,3}（挂起恢复）{0,3}删除）+。 VM生命周期以“VM create”开始，以“VM delete”结束，而“Stop-Start”，“Pause-Unpause”和“Suspend-Resume”等任务对可以在0到3次内随机出现生命周期。收集来自nova-api，nova-scheduler和nova-compute的INFO级别日志，并使用Elastic Stack [33]进行分析。在不同的执行点注入1种类型的异常：1）VM创建期间的中子超时; 2）破坏虚拟机时出现libvirt错误; 3）在销毁VM后清理期间发生libvirt错误。

建立。为了执行基于PCA和基于IM的方法，我们通过标识符字段将日志条目分组到不同的会话中，对于HDFS日志是block\_id，对于OpenStack日志是instance\_id。每个会话组分别是一个块或VM实例的生命周期。然后，我们将每个日志条目解析为一个日志密钥。 DeepLo​​g可以直接应用于日志键以训练其权重，随后用于检测异常，而其他方法则需要多一步。他们需要计算每个会话中每个不同日志键的出现次数，并构建一个矩阵，其中每列是一个不同的日志键（因此将有n列），每行代表一个会话向量，以及a的值矩阵中的单元Vij表示第i个会话中的日志密钥kj的计数。

DeepLo​​g需要一小部分正常日志条目来训练其模型。在HDFS日志的情况下，仅少于1％的正常会话（从前100,000个日志条目解析的4,855个会话与总共11,197,954相比）用于训练。请注意，DeepLo​​g可以查明哪个日志条目（及其对应的日志密钥）是异常的，但为了使用相同的度量来与竞争方法进行比较，我们使用“session”作为异常检测的粒度，即会话C是只要存在至少一个来自C的日志密钥被检测到异常，就认为是异常会话。

表3总结了两个数据集。请注意，PCA和IM是无需监督的offiine方法，不需要训练数据，而DeepLo​​g只需要正常系统执行产生的训练数据，而TFIDF需要训练正常和异常数据。



除了误报（FP）和漏报（FN）的数量之外，我们还使用标准度量，例如Precision，Recall和F-measure。精度= TP（TP代表真阳性）显示检测到的所有异常中真实异常的百分比; Recall = TP测量数据集中异常的百分比（假设我们知道地面实况）被检测到;和F-measure = 2•精度•召回是两者的调和平均值。

默认情况下，我们对DeepLo​​g使用以下参数值：g = 9，h = 10，L = 2和α= 64，并在我们的实验中研究它们的影响。回想g确定预测输出中的截止值被认为是正常的（即，具有接下来出现的top-g概率的g log键值被认为是正常的），并且h是用于训练和检测的窗口大小。 L和α分别表示DeepLo​​g中的层数和一个LSTM块中的内存单元数。对于所有其他方法，我们探索了它们的参数空间并报告了它们的最佳结果。当使用N-gram方法时，我们设置N = 1，除非另有说明，因为这显示了N-gram方法的最佳性能。

5.1.3比较。表4显示了HDFS数据上每种方法的误报和漏报的数量。 PCA实现了最少的误报，但代价是更多的假阴性。图6a显示了使用召回，精确度和F测量的更深入的比较。请注意，由于空间有限且相对性能非常差，因此图中省略了TFIDF。

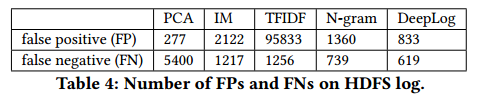
显然，DeepLo​​g实现了最佳的整体性能，F值为96％。当历史长度为1时，我们的基线解决方案N-gram也可以实现良好的性能。但是，随着历史窗口变长，其性能会急剧下降。相比之下，基于LSTM的方法更稳定，如第5.1.4节所示。

图6b研究了DeepLo​​g预测算法使用的top-g方法。设Dt是DeepLo​​g在t预测的top-g对数键值的集合，mt是在t处的数据中出现的实际对数键值。为了看到这种策略的影响，我们研究了不同g值的Pr [mt∈Dt]的CDF。在预测的超过11,000,000个日志密钥（标记为正常）中，88.9％的DeepLo​​g最高预测与mt完全匹配;而且96.1％的土地在DeepLo​​g的前2个预测范围内。当g = 5时，99.8％的正常mt在Dt范围内，同时异常检测率为99.994％（仅检测到一个异常会话）。

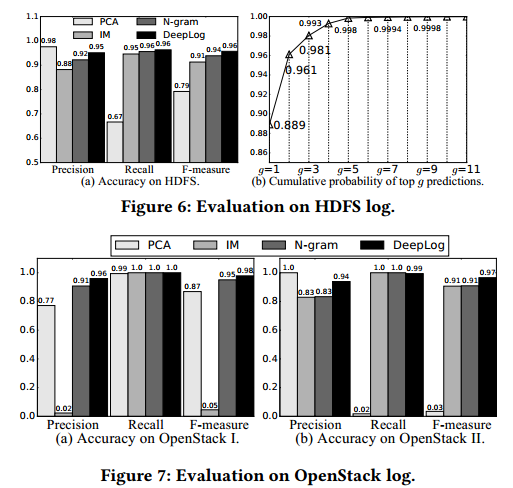
图7a显示了OpenStack数据集的性能。 PCA方法在该数据集上表现出合理的性能，但精度较低（仅为77％），而即使IM在这种情况下实现了完美的召回，它的精度也很差，只有2％（几乎所有VM实例都被检测为异常执行）。这是因为OpenStack日志是按照第5.1.2节中的描述随机生成的。请注意，在VM的生命周期（由一对Create和Delete定义）中出现的日志键（Stop Start）的次数是不确定的。这使IM很难找到异常检测的“稳定小不变量”。

为了测试这个假设，我们生成了第二个具有确定性模式的数据集，例如（Create Delete）+，导致总共5,544个正常VM执行和170个异常VM执行。我们将此数据集表示为OpenStack II，结果如图7b所示。 IM在这个具有更多规则模式的数据集上表现非常好。然而，在这种情况下，PCA方法的召回率仅下降到2％，因为数据中的正常模式过于规则，因此通过方差检测异常的PCA方法不起作用。

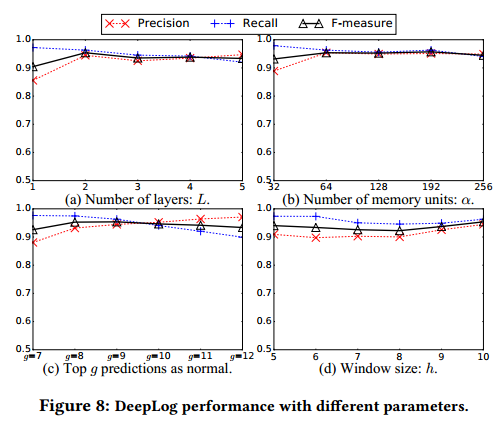
另一方面，DeepLo​​g在两个OpenStack日志上表现出优异的性能，F分别为98％和97％。最后，同样重要的是要注意PCA和IM是offine方法，并且它们不能用于每个日志条目执行异常检测。它们只能在会话级别检测到异常，但会话的概念甚至可能在许多系统日志中都不存在。



5.1.4 DeepLo​​g的分析。我们研究了DeepLo​​g中各种参数的性能影响，包括：g，h，L和α。结果显示在图8中。在每个实验中，我们改变一个参数的值，同时使用其他参数的默认值。



通常，DeepLo​​g的性能对于不同的值是相当稳定的，即，它对这些参数值中的任何一个或组合的调整不是非常敏感。这使得DeepLo​​g易于部署并在实践中使用。结果也很容易理解。例如，图8c显示较大的g值导致较高的精度但较低的召回率。因此，可以调整g以实现更高的真阳性率或更低的假阳性率。最后，DeepLo​​g的每个日志条目的预测成本在我们的标准工作站上仅为1毫秒左右，可以通过使用GPU等更好的硬件进一步改进。



### 5.2 Parameter value and performance anomaly

为了评估DeepLo​​g检测参数值和性能（包括日志条目之间经过的时间）异常的有效性，我们使用了OpenStack VM创建任务中的系统日志。此数据集包括两种类型的异常：性能异常（日志条目的延迟到达）和参数值异常（具有比其他更长的VM创建时间的日志条目）。

实验设置。和以前一样，我们在CloudLab上部署了一个OpenStack实验，并编写了一个脚本来模拟多个用户不断请求VM创建和删除。在OpenStack VM创建期间，一个重要的过程是将所需的映像从控制器节点复制到计算节点（将在其中创建VM）。为了模拟可能由DoS攻击引起的性能异常，我们将控制器的网络速度限制在两个不同点的计算节点，以查看DeepLo​​g是否可以检测到这些异常。

异常检测。如3.2节所述，我们将日志条目分为两组，一组用于模型训练，另一组（称为验证集）用于应用模型以生成MSE的高斯分布（均方误差）。在随后的在线检测阶段，对于每个输入参数值矢量→-u，DeepLo​​g检查→-u与其模型中的预测输出（矢量）之间的MSE是否在MSE的高斯分布的可接受置信区间内来自验证集。

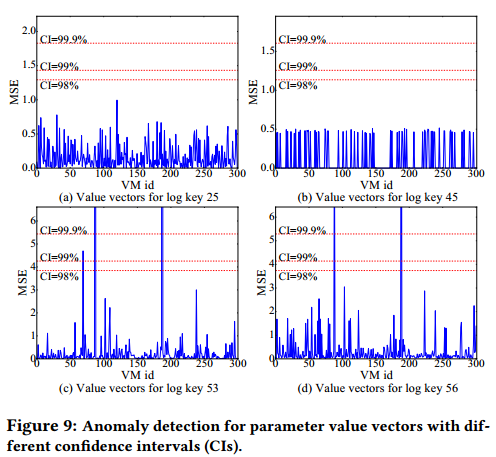
图9显示了不同日志键的参数值向量的检测结果，其中x -axis表示正在创建的VM的id（即，不同的VM创建实例），y轴表示参数值向量和之间的MSE。来自DeepLo​​g的预测输出向量。每个图中的水平线是相应的MSE高斯分布的置信区间阈值。图9a和9b表示两个对数键，其中它们的参数值向量在整个时间内是正常的。图9c和9d示出了在我们限制网络速度（即，注入的异常）的两个时间实例中，密钥53和56的参数值向量被成功检测为异常。

对于检测到的每个异常参数值向量，我们确定了与预测最不同的值，以识别异常列（特征）。我们发现，键53的两个异常参数值向量是由于异常大的经过时间值造成的。另一方面，键56是“花费\*秒来构建实例。”并且毫不奇怪，它的两个异常参数值向量是由非常大的值（秒）引起的。

### 5.3 Online update and training of DeepLog

第5.1节已经证明DeepLo​​g需要一个非常小的训练集（不到整个日志的1％），并且在训练阶段不需要用户反馈。但是有可能在检测阶段期间出现新的系统执行路径，这也是正常的，但是由于它没有被训练数据反映而被检测为异常。为解决此问题，本节将评估DeepLo​​g的在线更新和培训模块的有效性，如第3.3节所述。我们使用检测结果的差异来证明这一点，无论是有效性还是效率，有无增量更新。

5.3.1日志数据集。本节中使用的日志数据集是Blue Gene / L超级计算机系统日志1，其包含4,747,963个日志条目，其中348,460个条目被标记为异常。我们选择此数据集是因为一个重要特征：许多日志键仅在特定时间段内出现。这意味着训练数据集可能不包含所有可能的正常日志键，更不用说所有可能的正常执行模式。



5.3.2评估结果。我们进行了两次实验，一次使用前1％的正常日志条目作为训练数据，另一次使用前10％的日志条目进行训练。在这两种设置中，剩余的99％或90％条目用于异常检测。我们设定L = 1，α= 256，g = 6，h = 3。

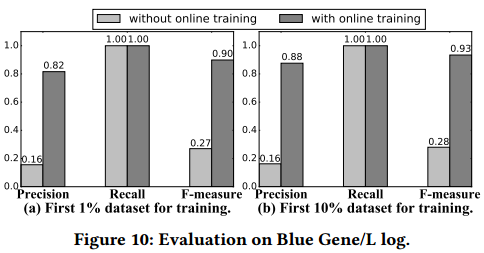
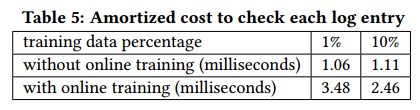


图10显示了两个实验的没有和在线培训的结果。在“没有在线培训”的情况下，我们运行DeepLo​​g来测试传入的日志条目，而无需任何增量更新。对于“在线培训”的情况，我们假设有最终用户报告检测到的异常是否为误报。如果是这样，DeepLo​​g使用该样本（现在是标记的记录）来更新其模型以学习这种新模式。图10显示，如果没有在线培训，只有1％的培训数据，这会导致许多误报（因此精度非常低）。尽管将其训练数据增加到10％会略微提高精度，但其性能仍然不尽如人意。另一方面，具有在线培训的DeepLo​​g显着提高了其精确度，从而显着提高了F-measure分数。在两种情况下，真正的阳性率为100％（完美召回），在线培训将1％训练数据的误报率从40.1％降低到1.7％，10％训练数据的误报率从38.2％降低到1.1％。

表5显示了检查每个日志条目的摊还成本。对于在线培训案例，我们报告了检测和在线更新所花费的时间（如果触发更新）。结果表明，在线更新和培训确实增加了每个日志条目的摊销成本，但只是略有增加。这是因为许多日志条目不会触发更新。注意，在线更新和在线检测可以并行执行;当模型使用当前权重继续执行检测时执行更新。



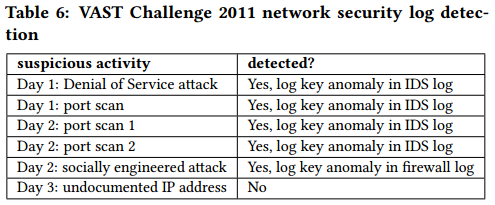
### 5.4 Security log case studies

具有从未在用于训练的正常日志中出现的日志键的异常（例如，“错误”或“异常”日志消息）易于检测。 DeepLo​​g可以有效地检测更多细微的案例。例如，在HDFS日志中，“删除块后没有更新Namenode”异常在会话中显示为缺少日志键;和“Redundant addStoredBlock”异常显示为额外的日志键。这意味着对于可能导致系统行为发生任何变化的任何攻击（通过日志反映），都可以检测到它。在下文中，我们将调查包含实际攻击的系统日志，以演示DeepLo​​g的有效性。

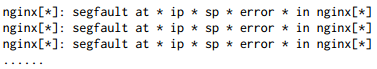
5.4.1网络安全日志。网络安全至关重要。防火墙和入侵检测系统（IDS）都会生成可用于在线异常检测的日志。

为了测试DeepLo​​g对网络安全日志的性能，我们使用了VAST Challenge 2011数据集，特别是Mini Challenge 2 - 计算机网络操作[1]。这一挑战是通过可视化技术手动查找可疑活动。它为异常活动提供了基本事实。对于基础事实中的所有异常，表6显示了DeepLo​​g的结果。唯一未检测到的可疑活动是首次出现未记录的计算机IP地址。

当DeepLo​​g报告在短时间内多次反复出现的异常消息时，发生了唯一的误报。这是由于突然变得突发的事件并在短时间内多次打印相同的日志消息。 VAST Challenge未将此视为可疑活动。



 5.4.2 BROP攻击检测。盲回报编程（BROP）攻击[5]利用了一个事实，即许多服务器应用程序在崩溃后重新启动以确保服务可靠性。这种攻击是强大而实用的，因为攻击者既不依赖于源代码也不依赖于二进制文件。导致服务器崩溃的堆栈缓冲区溢出漏洞足以执行此攻击。在BROP漏洞利用中，攻击者使用服务器崩溃作为信号来帮助完成执行shellcode的ROP攻击。但是，重复的服务器重新启动活动会在内核日志中留下许多非典型的日志消息，如下所示，DeepLo​​g很容易检测到这些消息。



### 5.5 Task separation and workflow construction

我们在第4节中实现了提议的方法，并在具有各种OpenStack VM相关任务的日志上进行了评估。 LSTM方法和基于密度的聚类方法都可以成功地分离所有任务。第一种方法需要LSTM;它是一种监督方法，需要提供训练数据。第二种方法使用对特定距离阈值内的对数密钥的共现进行聚类，这是一种无监督的方法。因此，它不需要训练，但它确实需要参数τ作为距离阈值。

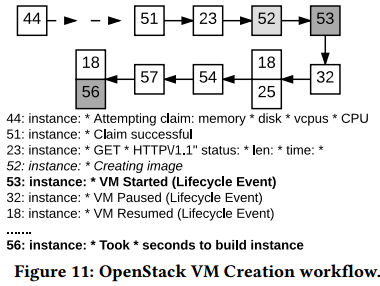
具体而言，对于基于密度的聚类方法，具有足够大的阈值τ∈[0.85,0.95]，所有任务都有明确的分离。注意，τ的值不能太大（例如，设置τ= 1），因为后台进程可能在随机位置产生日志条目，这将破坏来自相同任务的日志条目。

接下来，我们使用VM创建工作流程的一部分，以显示它如何在第5.2节中提供对性能异常的有用诊断。回想一下5.2节，参数值向量异常在日志键53的经过时间值和日志键56的参数位置（表示构建实例的秒数）上标识。如图11所示，一旦DeepLo​​g检测到异常，我们就知道构建该实例所花费的时间是异常的，但我们不知道为什么。然后，由于日志密钥53与其先前日志密钥之间的经过时间过长，通过调查由DeepLo​​g构建的工作流模型，其先前的密钥为52：“创建映像”，因此我们知道创建虚拟机的时间比平时长因为创建图像的时间太长了。根据此过程进行的进一步调查可能会发现它是由从控制节点到计算节点的网络速度较慢引起的。

## 6 RELATED WORK

主要用于记录显着事件以便于调试，系统事件日志提供丰富的信息，实际存在于每个计算机系统上，使其成为跟踪和调查系统状态的宝贵资源。但是，由于系统日志主要由各种自由格式文本组成，因此分析具有挑战性。

已经为不同的系统设计了许多日志挖掘工具。许多人使用基于规则的方法[7,15,28,29,31,32,40,41]，虽然这些方法准确，但仅限于特定的应用场景，还需要领域专业知识。例如，Beehive [41]通过无监督的数据特定聚类聚类识别日志中的潜在安全威胁，然后手动标记异常值。 Oprea [28]使用置信传播从DNS日志中检测早期企业感染。 PerfAugur [32]专门用于通过使用谓词组合等特殊功能挖掘服务日志来查找性能问题。 DeepLo​​g是一种不依赖于任何特定领域知识的通用方法。



使用系统日志进行异常检测的其他通用方法通常采用两步程序。首先，日志解析器[9,14,16,23,36,37]用于将日志条目解析为结构化表单，结构化表单通常仅包含“日志密钥”（或“消息类型”）。除了用于分隔和分组日志条目的标识符之外，将丢弃参数值和时间戳。然后，对日志密钥序列执行异常检测。一种典型的方法是通过计算唯一的日志键或使用更复杂的方法（如TF-IDF）为每个会话或时间窗口生成数字向量。然后，包含这些载体的矩阵适用于基于矩阵的无监督异常检测方法，例如主成分分析（PCA）[38,39]和不变采矿（IM）[22]。构建这样的矩阵通常是一个异常的过程，并且这些方法不能提供日志入口级别的异常检测（相反，它们只能在会话级别运行）。我们请读者[17]对这些方法进行概述和比较。

监督方法[17,44]使用正常和异常向量来训练检测未来异常的二元分类器。这种方法的缺点是可能无法检测到不在训练数据中的未知异常。此外，很难获得用于训练的异常数据。我们在评估中表明，只使用一小部分正常数据进行训练，DeepLo​​g可以实现更好的在线异常检测。此外，DeepLo​​g还使用时间戳和参数值进行异常检测，这在以前的工作中是缺失的。

工作流程构建主要使用从日志日志文件中提取的日志密钥进行研究[4,11,21,42]。已经表明，工作流程为异常检测提供了有限的优势[11,42]。相反，工作流程的主要用途是帮助系统诊断[4,21]。但是，所有过去的工作都假定模型的日志文件仅包含一个任务的重复执行。在本文中，我们提出了自动将不同任务与日志文件分开的方法，以便为不同任务构建工作流模型。

除了工作流程之外，使用系统日志执行异常诊断的其他系统还包括DISTALYZER [26]，它通过将有问题的日志与正常日志进行比较来诊断系统性能问题，LogCluster [19]聚集和组织历史日志以帮助未来的问题识别，以及Stitch [45]从系统日志中提取不同级别的标识符，并为用户构建Web界面，以便可视地监视每个会话的进度并找出性能问题。请注意，一旦检测到异常，它们用于诊断目的，并且不能用于异常检测本身。

## 7 CONCLUSION

本文介绍了DeepLo​​g，这是一种使用基于深度神经网络的方法进行在线日志异常检测和诊断的通用框架。 DeepLo​​g学习并编码整个日志消息，包括时间戳，日志密钥和参数值。它在每个日志条目级别执行异常检测，而不是在每个会话级别执行异常检测，因为许多以前的方法仅限于此。 DeepLo​​g可以从日志文件中分离出不同的任务，并使用深度学习（LSTM）和经典挖掘（密度聚类）方法为每个任务构建工作流模型。这可以实现有效的异常诊断。通过整合用户反馈，DeepLo​​g支持其LSTM模型的在线更新/培训，因此能够整合并适应新的执行模式。对大型系统日志的广泛评估清楚地证明了DeepLo​​g与以前的方法相比具有更高的效率。

未来的工作包括但不限于将其他类型的RNN（循环神经网络）整合到DeepLog中以测试其效率，并集成来自不同应用程序和系统的日志数据以执行更全面的系统诊断（例如，MySQL数据库的故障可能是 由单独的系统日志中反映的磁盘故障引起的）。

## 8 ACKNOWLEDGMENT

作者感谢匿名审稿人提供的宝贵意见。 作者感谢NSF拨款1314945和1514520的支持。国家自然科学基金委员会拨款61729202也部分支持飞飞李。我们要感谢TCloud项目的所有成员和Flux小组的有益讨论和反馈，特别是Cai（Richard）Li， 他对BROP攻击的宝贵意见。