哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

Osdetector 开发设计文档

——proj120:实现智能的操作系统异常检测

项目成员

姓名	年级	联系方式（QQ）
于伯淳（队长）	大二	1978689494
满洋	大二	2579504636
李怡凯	大三	1351855206

指导教师：夏文、李诗逸

技术导师：高若姝、陈东辉

2022-06

完成进度

本项目的主要目标如下：

1. 实现进程级系统调用和占用资源（CPU、内存、网络）的采集；
2. 对采集到的数据进行处理，从中检测到操作系统可能发生的异常（CPU 冲高、内存泄漏、网络流量异常等）；
3. 适配不同实际场景和运行环境，提高系统的可移植性。

目前，我们的赛题完成度如下：

目标编号	完成情况	说明
1	基本完成（约 90%）	能够以 csv 格式，以时序记录目标进程的 CPU 占用、内存占用、网络流量、系统调用四种关键数据。
2	大致完成（约 80%）	以 lstm 神经网络为主，辅以 knn 等算法，在合理的阈值设定下能对明显异常达到 100%的准确率，并可以灵活调整阈值以调整对异常灵敏度。
3	初步完成（约 50%）	项目主体采用 Python 的 bcc 框架、pyod 和 pytorch 等工具，移植性的瓶颈主要是 ebpf 在不同内核版本的差异。
总计	大致完成（约 80%）	已经对 cpu 冲高、内存占用异常、网络流量异常实现了一套解决方案，后续将考虑算法的多样性以及项目在不同平台的移植性问题。 目前项目内存和 cpu 占用开销较大，约为 300MB；异常检测速度较快，每条数据的检测时间平均能达到 0.01s；准确率较为理想，对手动注入的异常能达到 100%的检出，对选取的开源数据集中的异常也都实现了检出。

目录

1 概述

1.1 项目背景及意义

随着云技术的飞速发展，云系统的复杂性和规模不断增加，云系统的稳定性受到了极大挑战。在日常的业务流程中，操作系统可能会出现各种异常现象，如CPU 冲高、内存泄漏、网络流量异常等。

为了使运维人员能够及时发现操作系统异常，定位到相关进程，并获取异常数据，构建智能化、自动化的操作系统异常检测平台已经成为重要的工作。

异常检测工作具有三个主要的评价标准：准确率、检测速度和资源占用。对于具体的异常检测项目而言，还需要考虑项目本身在不同操作系统平台下的可移植性，以及算法在不同场景下的通用性。因此，构建通用、高效、准确的异常检测平台有很大的挑战性。

本项目旨在从系统指标（CPU 占用率、内存分配、网络流量）的时间序列中分析并检测出可能存在的异常情况，并将检测结果通过日志、图表等形式呈现，为运维人员掌控系统异常情况、分析解决异常提供可靠参考。

通过本次项目，我们希望结合 eBPF 等前沿技术对异常检测这一重要领域进行有益的尝试与探索，并提出一些具有建设性的方案。

1.2 环境搭建和工具选择

本项目目前基于以下环境进行开发与测试：

- 操作系统：Ubuntu 20.04
- 内核版本：5.4.0
- CPU 架构：x64

本项目目前使用的工具：

- Python 3.8.10
- bcc release 0.24.0
- pyod 1.0.0
- pytorch 1.11.0
- onnx 1.11.0
- onnxruntime 1.11.1

2 项目模块设计

当前项目主要分为数据收集模块、算法模块。项目的总体结构如下图所示：

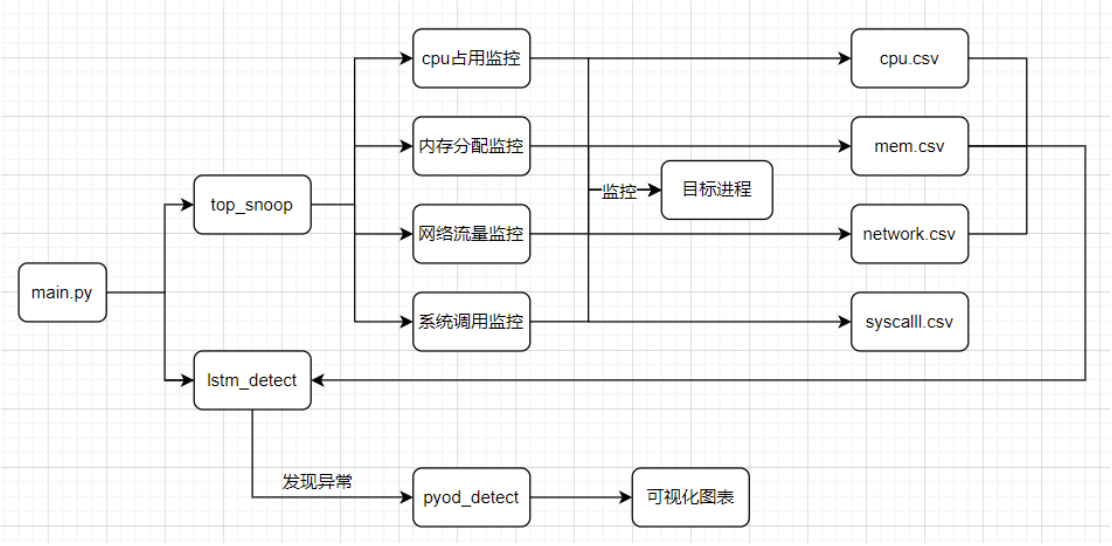

图 1 项目总体结构

用户通过 main.py 对指定进程进行监测，main.py 会根据输入的时间间隔参数对目标进程进行信息采集，同时定期调用 lstm_detect 监测程序对采集的数据进行处理，如果发现异常则会调用 pyod_detect 对异常以及附近的数据进行进一步

异常检测和可视化处理。

2.1 数据收集模块

此模块主要基于 eBPF 技术的 python 框架 bcc。eBPF 是一项革命性技术，起源于 Linux 内核，可以在特权上下文（如操作系统内核）中运行沙盒程序。它用于安全有效地扩展内核的功能，而无需更改内核源代码或加载内核。eBPF 原理图如图所示。

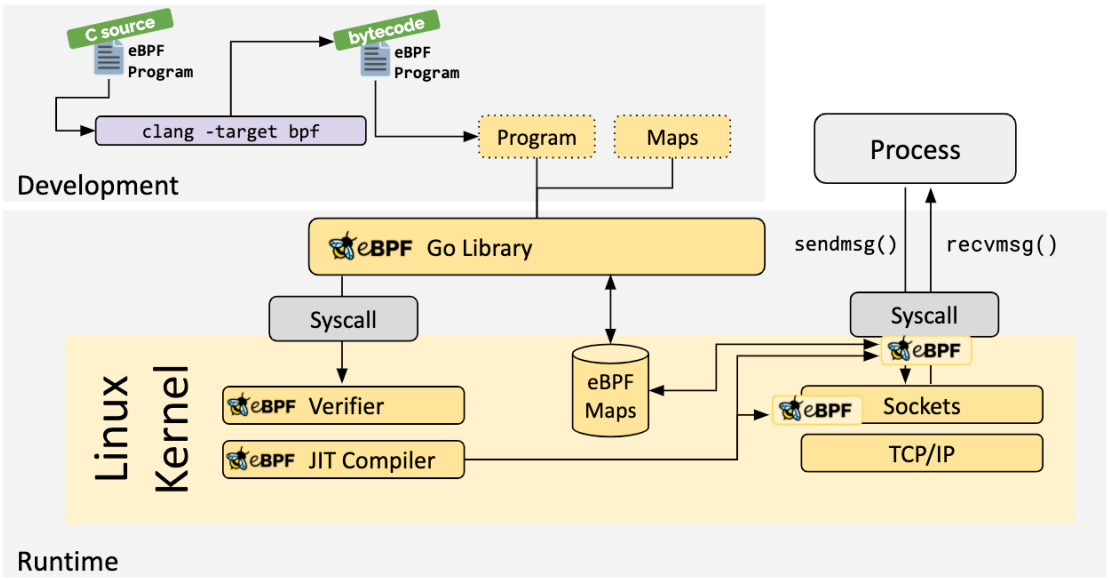

图 2 eBPF 原理图

bcc 是一个用于创建高效内核跟踪和操作程序的工具包，其使用 C 中的内核工具（包括围绕 LLVM 的 C 包装器），以及 python 和 lua 中的前端，使得 eBPF 程序更加容易编写，现已被应用于许多任务，包括性能分析与网络流量控制。

目前本模块由一个顶层模块和四个子模块组成，模块结构如图所示：

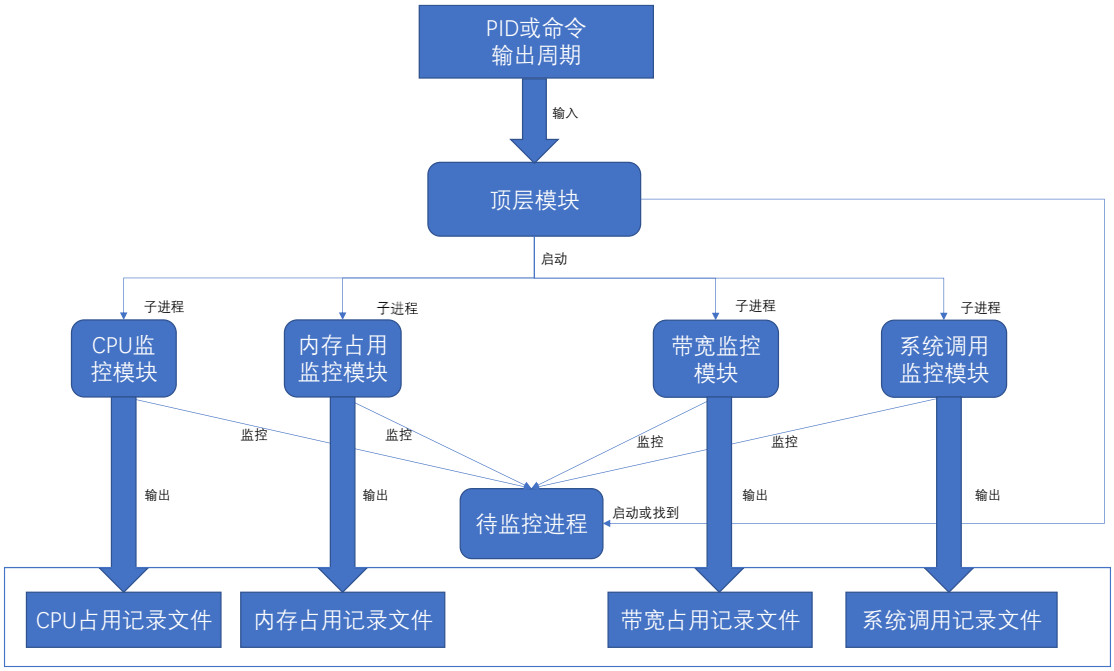

图 3 监控模块总体结构

其顶层模块负责处理命令参数，并启动子模块进程进行对目标进程的资源占用、系统调用的监控。四个子模块分别实现 CPU 占用、内存占用、带宽占用、系统调用的监控功能，并通过 csv 文件记录各资源指标的时间序列。

输入参数处理：模块支持参数如图所示。

```
usage: top_snoop.py [-h] [-p PID] [-c COMMAND] [-i INTERVAL]

Attach to process and snoop its resource usage

optional arguments:
  -h, --help            show this help message and exit
  -p PID, --pid PID     id of the process to trace (optional)
  -c COMMAND, --command COMMAND
                        execute and trace the specified command (optional)
  -i INTERVAL, --interval INTERVAL
                        The interval of snoop (unit:s)

EXAMPLES:
    ./top_snoop -c './snoop_program' # Run the program snoop_program and snoop its resource usage
    ./top_snoop -p 12345  # Snoop the process with pid 12345
    ./top_snoop -p 12345 -i 1  # Snoop the process with pid 12345 and output every 1 second
```

图 4 监控模块支持参数

顶层模块通过使用 Python 程序库 argparse 可以很容易从输入命令中获取各个参数字段。

启动子进程进行监控：在完成对输入参数的处理后，顶层模块会启动四个子进程，分别对应四个监控子模块，用于对指定进程进行监控记录。

2.2 算法模块

此模块负责对采集到的 csv 文件进行处理，并通过选取的异常检测算法进行异常的检出。

操作系统的参数实际上是具有连续性的时间序列参数，因此时序数据分析的很多方法都可以在异常检测中运用。

常见的异常类型有如下几种：

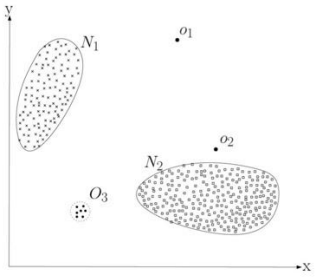

Fig. 1. A simple example of anomalies in a two-dimensional data set.

图 5 点异常

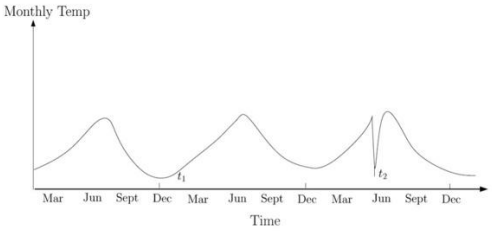

Fig. 3. Contextual anomaly t_2 in a temperature time-series. Note that the temperature at time t_1 is same as that at time t_2 but occurs in a different context and hence is not considered as an anomaly.

图 6 上下文异常

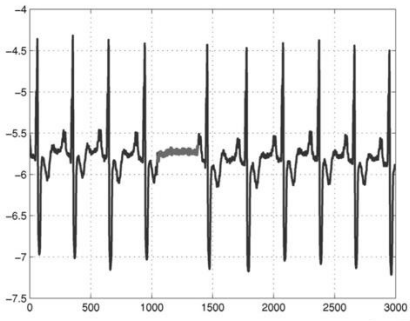

Fig. 4. Collective anomaly corresponding to an *Atrial Premature Contraction* in an human electrocardiogram output.

图 7 集合异常

算法模块以基于 lstm 的异常检测算法为主，并辅之以基于 PyOD 的异常检

测功能。

PyOD 是 python 的一个异常检测工具库，集成了 kNN/LOF/LOCI/ABOD 等多种异常检测算法，支持多种 python 版本和操作系统，一些常见的异常检测算法如图所示：

Method	Category	JIT Enabled	Multi-core
LOF (Breunig et al., 2000)	Proximity	No	Yes
kNN (Ramaswamy et al., 2000)	Proximity	No	Yes
AvgkNN (Angiulli and Pizzuti, 2002)	Proximity	No	Yes
CBLOF (He et al., 2003)	Proximity	Yes	No
OCSVM (Ma and Perkins, 2003)	Linear Model	No	No
LOCI (Papadimitriou et al., 2003)	Proximity	Yes	No
PCA (Shyu et al., 2003)	Linear Model	No	No
MCD (Hardin and Rocke, 2004)	Linear Model	No	No
Feature Bagging (Lazarevic and Kumar, 2005)	Ensembling	No	Yes
ABOD (Kriegel et al., 2008)	Proximity	Yes	No
Isolation Forest (Liu et al., 2008)	Ensembling	No	Yes
HBOS (Goldstein and Dengel, 2012)	Proximity	Yes	No
SOS (Janssens et al., 2012)	Proximity	Yes	No
AutoEncoder (Sakurada and Yairi, 2014)	Neural Net	Yes	No
AOM (Aggarwal and Sathe, 2015)	Ensembling	No	No
MOA (Aggarwal and Sathe, 2015)	Ensembling	No	No
SO-GAAL (Liu et al., 2018)	Neural Net	No	No
MO-GAAL (Liu et al., 2018)	Neural Net	No	No
XGBOD (Zhao and Hryniewicki, 2018b)	Ensembling	No	Yes
LSCP (Zhao et al., 2019)	Ensembling	No	No

图 8 PyOD 集成的算法

模块通过 pandas 读入 csv 文件，以 kNN、LOF、CBLOF 等为主要算法对数据进行处理，并最终通过 matplotlib 对处理结果进行可视化呈现。

直接调用模板算法对真实场景数据进行处理的准确率不够高，对特定场景进行参数优化又会导致迁移性较差。根据业界前沿研究动态，深度学习的方法在大量数据的工业场景下具有准确性高，可迁移性好的优点，因此我们还提出了一种基于深度学习的操作系统异常检测流程。

根据深度学习的常用流程，我们将流程分为两部分：训练和推理。其中训练部分可以离线完成，可以根据实际情况和实际数据对模型结构或训练参数进行调整，有利于提高模型的表现。而推理部分是设计在真实工业场景中运行，参数不会频繁改变，因此采用不同的模型部署方法，达到减少依赖，提高性能的目的。

2.3 项目工作流程简述

目前项目通过顶层 main.py 进行各模块的集成调度，监测模块位于 snoop 子模块，基于 PyOD 和 lstm 的算法模块分别位于 pyod_detect 和 lstm_detect 模块，各模块的独立性较高，可根据实际场景进行组合定制。

当前已采用的调度方式为 main.py 调用 snoop 的顶层模块 top_snoop 对目标进程进行数据监测，main.py 根据时间间隔参数定期调用 lstm_detect 模块对得到的 csv 文件进行异常检测，当检测到异常时，main.py 通过 pyod_detect 模块对检出异常的时间区间进行二次检测并将区间内数据可视化呈现。

main.py 能够将项目得到的 csv 文件和可视化图片文件进行定期备份，以便后续参考。

3 项目具体实现

3.1 数据收集模块

3.1.1 CPU 占用监控模块

CPU 占用监控模块负责对进程的 CPU 占用率进行监控与记录，通过在 finish_task_switch 这个挂载点挂载自定义函数，实现对进程的运行时间与非运行时间进行记录，如图所示。

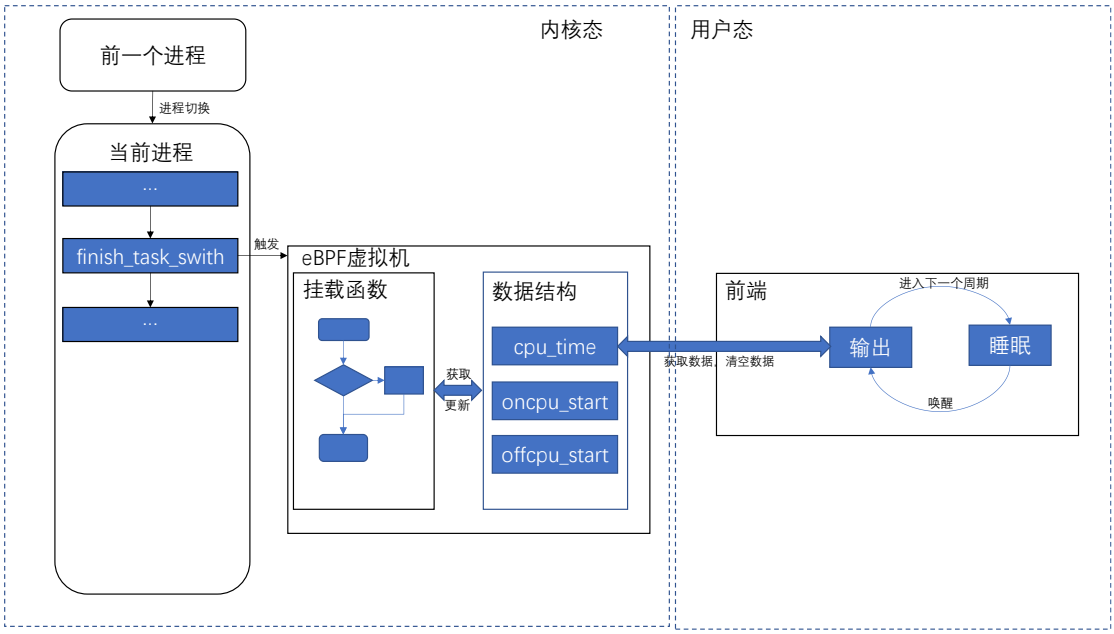

图 9 CPU 占用监测模块原理

具体设计如下：

（1） 数据结构

采用 3 个 BPF 哈希表记录有关信息：

oncpu_start：根据 pid 记录进程运行态的开始时间；

offcpu_start：根据 pid 记录进程非运行态的开始时间；

cpu_time：根据 pid 记录进程的运行总时间；

（2） 挂载程序

finish_task_switch 是 Linux 内核中进程切换后执行的一个函数，用于清理上下文。在此处挂载函数可以获得当前进程与前一个进程的信息。

挂载程序具体流程如下：

1） 获取当前进程 pid 与前一个进程 pid；
2） 获取当前时间戳；
3） 将当前时间作为前一个进程运行态的结束时间与非运行态的开始时间，更新 offcpu_start 与 cpu_time；
4） 将当前时间作为当前进程非运行态的结束时间与运行态的开始时间，更新 oncpu_start 与 cpu_time；
5）退出挂载程序，返回进程原来执行的位置。

（3） 前端处理

前端根据输出周期 interval 唤醒，并记录当前时间，与前一次唤醒时记录的时间做差计算总时间，转入输出函数。

在输出函数中，前端从 cpu_time 数据结构中取出键值对并删除该键值对，获取进程 pid 与运行态时间，根据$\frac{运行态时间}{总时间}$计算进程 CPU 占用率并输出到文件。然后重新进入睡眠等待下一次唤醒。

3.1.2 内存占用监控模块

内存占用模块负责对进程的内存占用情况进行监控记录，通过在用户态中的内存请求相关的函数处挂载自定义函数，记录内存的申请与释放情况，进而计算进程的内存占用，如图所示。

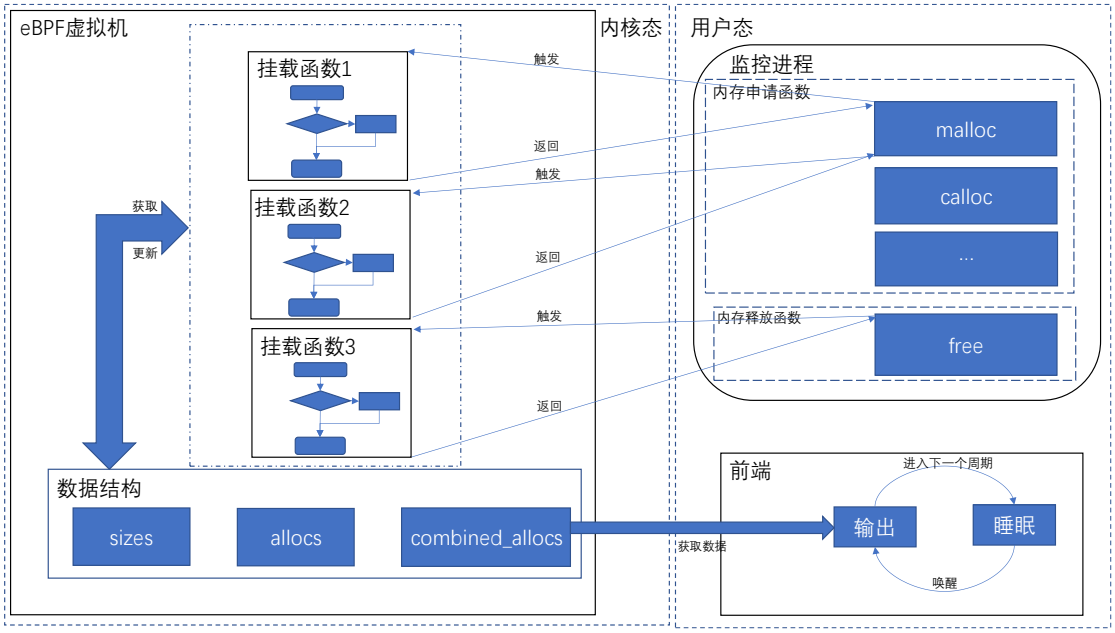

图 10 内存占用监测模块原理

具体设计如下：

（1） 数据结构

使用 3 个 BPF 哈希表记录相关信息：

sizes：根据 pid 号记录某次内存分配大小；
allocs：根据分配内存的起始地址记录某次内存分配信息；
combined_allocs：根据 pid 号记录进程占用的内存大小；

（2） 挂载函数

本模块挂载的函数如下表：

内存申请	malloc
	calloc
	realloc
	posix_memalign
	valloc
	memalign
	pvalloc
	aligned_alloc
内存释放	free

表 1 内存占用监测模块挂载函数列表

挂载在内存申请函数处的自定义函数分为两部分，分别是挂载在函数入口与函数返回后。

1） 挂载在函数入口的自定义函数

挂载在函数入口处的自定义函数能够获得本次内存申请的大小，而在函数返回处无法得到这个信息，所以需要在 sizes 哈希表中记录当前进程申请的内存大小。具体流程如下：

i. 利用 eBPF API bpf_get_current_pid_tgid()获取当前进程 PID；
ii. 判断当前进程是否是监控进程，不是则退出；
iii. 在 sizes 表中记录当前 PID 申请分配的内存大小；
iv. 结束，返回原来执行的位置；

2） 挂载在函数返回后的自定义函数

挂载在函数出口处的自定义函数能够通过上下文信息中 RC 寄存器中的信息获得内存分配函数的返回值，即内存分配的起始地址，并根据这个值是否为 0 判断本次内存分配是否成功。

该函数通过 pid 号在 sizes 哈希表中获得本次内存分配的大小，并将地址信息、大小信息记录在 allocs 哈希表中，并更新 combined_allocs 中记录的进程占用内存大小。具体流程如下：

i. 利用 eBPF API bpf_get_current_pid_tgid()获取当前进程 PID；
ii. 判断当前进程是否是监控进程，不是则退出；
iii. 判断 address 是否为 0，为 0 则说明内存分配失败，退出；
iv. 根据 PID 在 sizes 中查找申请分配的内存大小，更新 allocs 与 combined_allocs；
v. 结束，返回原来执行的位置；

挂载在内存释放函数处的自定义函数可以在函数入口处获得本次释放内存的起始地址，根据该地址查找 allocs 哈希表可以获得对应的内存分配信息。根据这些信息删除 allocs 表项，并更新 combined_allocs 中记录的进程内存占用大小。具体流程如下：

i. 根据参数 address 查找 allocs 表中对应的内存分配信息，如果不存在则直接返回；
ii. 从内存分配信息中获取内存分配大小；
iii. 删除 allocs 中的本次内存分配信息，更新 combined_allocs 中当前进程内存占用的大小；
iv. 结束，返回原来执行的位置；

（3） 前端处理

前端根据设定的 interval 唤醒，并将 combined_allocs 哈希表中的键值对取出，输出当前时间、内存占用大小与未释放的内存分配次数。然后重新进入睡眠等待下一次唤醒。

3.1.3 流量监控模块

流量监控模块负责对进程的流量进行监控记录，通过在数据链路层挂载自定义函数，记录进程流量，如图所示。

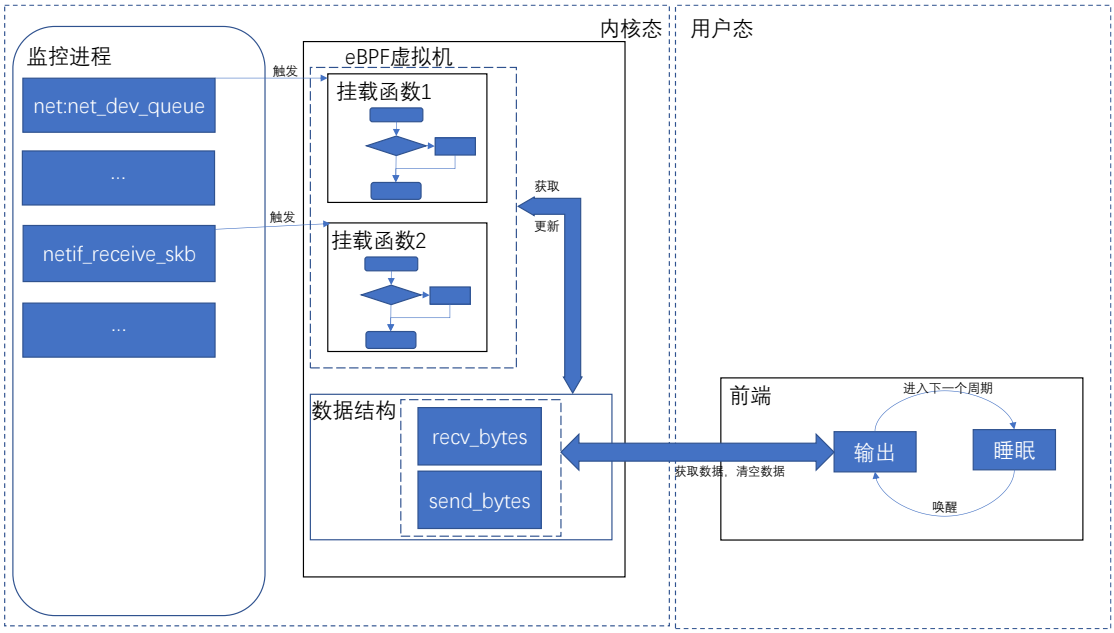

图 11 流量监测模块原理

具体设计如下：

（1） 数据结构

使用 2 个 BPF 哈希表记录收发数据量：

send_bytes：根据进程信息结构体记录发送流量；
recv_bytes：根据进程信息结构体记录接收流量；

进程信息结构体包含的字段如下：

进程 PID；
进程名；

（2）挂载函数

tracepoint 是 Linux 内核静态定义的一些调试点，它分布于内核的各个子系统中，可以用于挂载钩子函数实现追踪。相比直接将自定义函数挂载在内核函数上，tracepoint 具有更高的稳定性，因为其不会随着内核函数变动而发生改变。eBPF 同样支持对 tracepoint 的挂载。

本模块通过挂载自定义函数到 net:net_dev_queue 与 net: netif_receive_skb 两个 tracepoint 实现收发数据流量的监控记录。

自定义函数通过 eBPF API 获得当前进程 pid 与进程名，然后在 tracepoint 提供的 args 结构体中获取流量包大小，然后对 send_bytes 或 recv_bytes 进行更新。

具体流程如下：

1） 利用 eBPF API bpf_get_current_pid_tgid()获取当前进程 PID；
2） 判断当前进程是否是监控进程，不是则退出；
3） 创建进程信息结构体并填写字段；
4） 使用进程信息结构体查找 send_bytes 或 recv_bytes 中当前进程对应的流量值；
5） 使用 ebPF API increment()对流量进行增加，增加量为当前发送或收到的数据包的大小；
6） 结束，返回原来执行的位置；

（3） 前端处理

前端根据设定的 interval 唤醒，从 send_bytes 与 recv_bytes 两个表中获取该周期内的收发流量数据并清空这两个表，然后输出获取到的流量信息，重新进入睡眠等待下一次唤醒。

3.1.4 系统调用监控模块

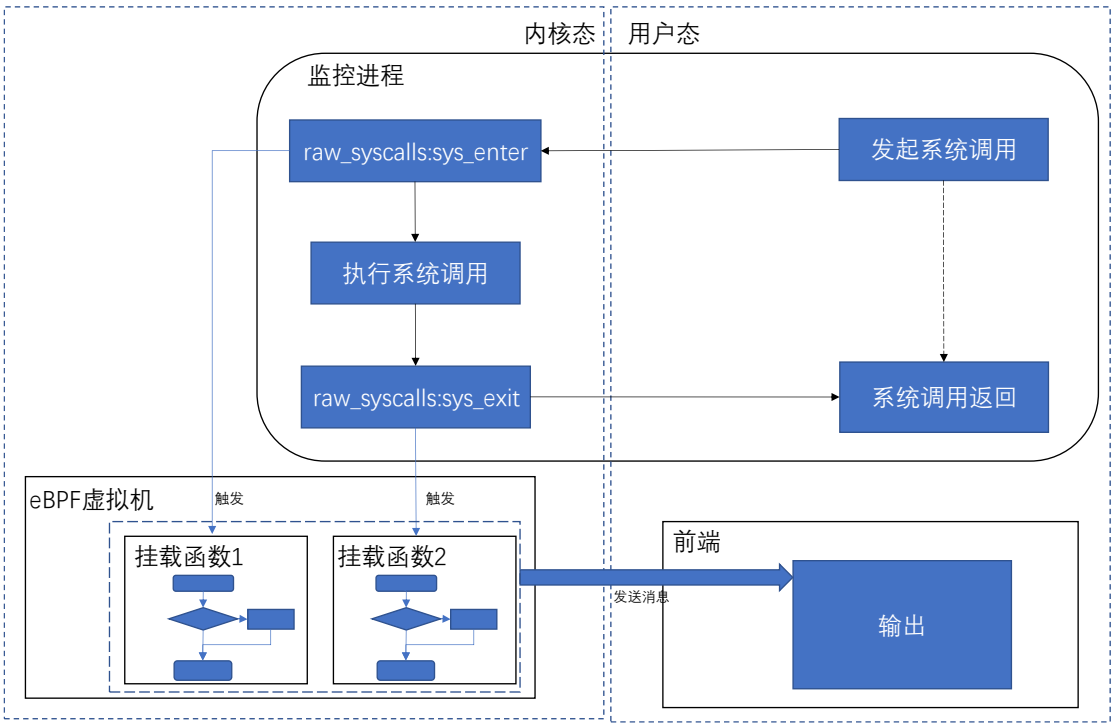

图 12 系统调用监测模块原理

系统调用监控模块负责监控并记录进程经历的系统调用，通过在每个系统调用的入口和返回处挂载自定义函数，实现系统调用追踪，如图所示。

具体设计如下：

（1） 挂载函数

本模块通过挂载自定义函数到 raw_syscalls:sys_enter 和 raw_syscalls:sys_exit 实现对系统调用的追踪，分别对应系统调用的进入与返回。

自定义函数通过 args 结构体可以获得当前系统调用的调用号，之后利用 eBPF 提供的输出 API bpf_trace_printk 可以向前端发送消息。具体流程如下：

1） 利用 eBPF API bpf_get_current_pid_tgid()获取当前进程 PID；
2） 判断当前进程是否是监控进程，不是则退出；
3） 利用 eBPF API bpf_ktime_get_ns()获取当前时间戳；
4） 从 args 结构体中获得系统调用号；
5） 使用 eBPF API bpf_trace_printk()向前端发送进入或退出系统调用的信息，并发送系统调用号；
6） 结束，返回原来执行的位置；

（2） 前端处理

前端持续等待 eBPF 后端发来的消息。收到消息后，输出当前时间戳、系统调用号等信息。

3.2 算法模块

3.2.1 基于 PyOD 的算法模块

目前本项目主要采用了 PyOD 中基于近邻的几种算法，包括 kNN、LOF、CBLOF 等，可以通过 pandas 库对 csv 文件进行读入和数据处理，利用 PyOD 中已集成的算法对处理后的数据进行训练，并对数据进行打分。

```
clf = KNN(method='mean')

x = npy.array(df['ticks']).reshape(-1, 1)
y = npy.array(df[' size(B)']).reshape(-1, 1)
train_data = npy.concatenate((x, y),axis=1)

clf.fit(train_data)

train_pred = clf.labels_
train_scores = clf.decision_scores_
```

图 13 PyOD 模块对内存数据进行检测

通过设定分数阈值，可以在原始数据中标记出可能存在异常的数据点，并

通过 matplotlib 将训练结果可视化呈现：

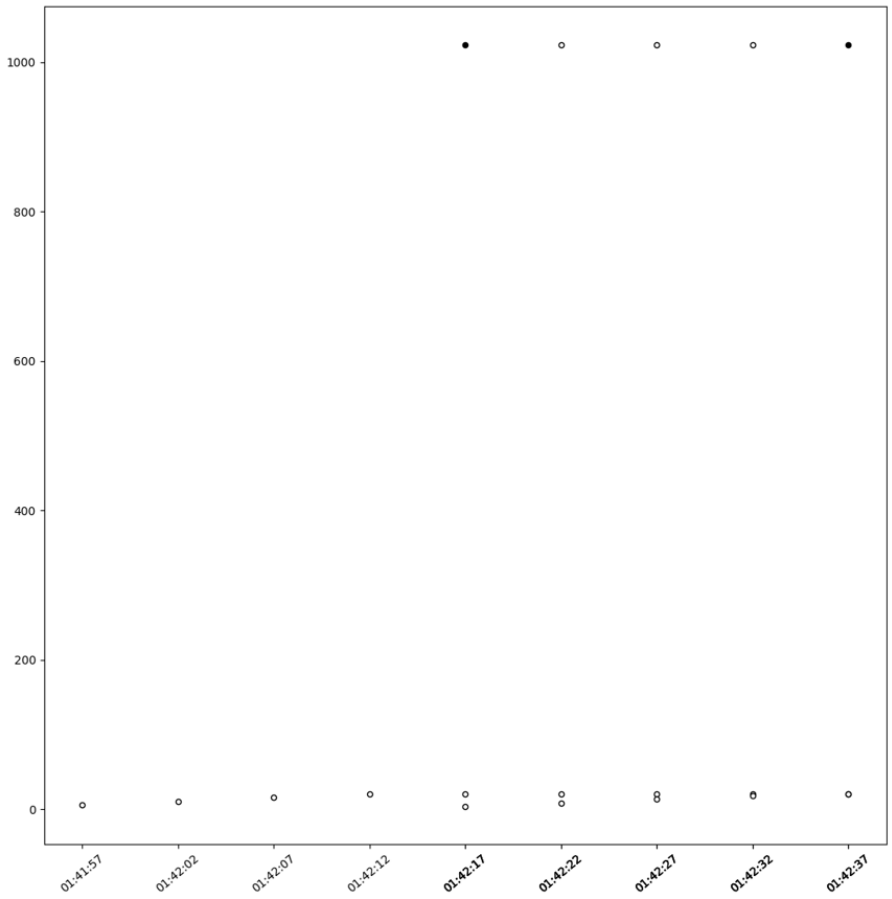

图 14 PyOD 对内存数据的检测结果

上图是以简单编写的测试程序进行内存数据检测的结果。x 轴为时间，y 轴为目标进程占用内存（单位：字节），通过近邻算法 kNN 可以检测到异常数据第一次和最末次出现的位置，并在图中以黑点标明，便于查明异常出现的时间节点。

3.2.2 基于 LSTM 的算法模块

训练阶段

训练阶段可以采用任意的数据，甚至是非操作系统中收集的数据，只要能够代表无异常的数据，体现合理数据的特征，就可以用于训练。训练模型也可以使用新兴的预训练-微调模式，也可以利用迁移学习等深度学习技术提高模型的通用性。

在我们的项目中，模型结构使用了简单的两层 LSTM+线性分类器，能够作为深度学习的异常检测方法的代表。

我们在训练阶段使用业界常用的 PyTorch 框架进行模型搭建和训练。

训练原理

工业场景中，常常没有可供训练的有标注数据，大多数情况下能够获得的是大量无标注的正常数据，所以无监督训练的实际应用场景更广泛。

在我们的系统中，使用无监督的训练模式，使用时序数据的一段窗口内的数据作为输入，训练模型预测之后一段时间的数据。模型结构：

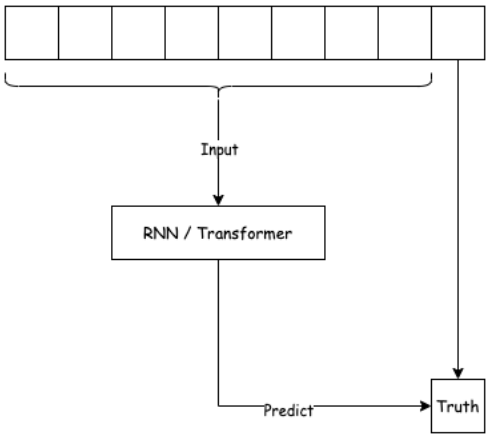

图 15 训练数据预处理

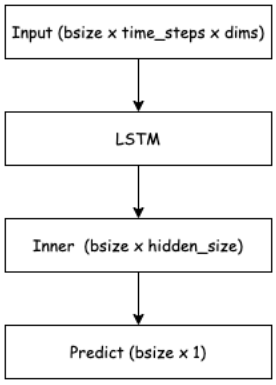

图 16 模型结构

上图中 Batch size 是可变的，在训练时使用较大的 Batch size 可以提高训练速度，在预测时通常使用的 Batch size 为 1。而 time_steps 是选取的窗口大小，这个参数需要选取合适。Dims 则是检测变量的维度，比如同时对 CPU、内存和网络进行监控，那么 dims 就设置为 3，如果有更多的数据那么也可以灵活地设置成更大的值。这种方法可以容易的扩展到多维度的数据，也可以扩展预测的范围以获得更灵敏的检测。

推理阶段

推理阶段需要完成的任务仅仅是对已有的参数进行矩阵运算，因此并不需要使用 PyTorch 这样重型的依赖。我们采用 OnnxRuntime 对模型的进行部署，OnnxRuntime 是一个轻量化的推理框架，可以支持多种平台，在 arm 架构或嵌入式设备的 NPU 中也可以运行。下图是 OnnxRuntime 的支持矩阵：

Optimize Inferencing	Optimize Training						
Platform	Windows	Linux	Mac	Android	iOS	Web Browser (Preview)	
API	Python	C++	C#	C	Java	JS	Obj-C	WinRT
Architecture	X64	X86	ARM64	ARM32	IBM Power		
Hardware Acceleration	Default CPU	CoreML	CUDA	DirectML	oneDNN		
	OpenVINO	TensorRT	NNAPI	ACL (Preview)	ArmNN (Preview)		
	MIGraphX (Preview)	TVM (Preview)	Rockchip NPU (Preview)	Vitis AI (Preview)			

图 17 OnnxRuntime 支持矩阵

异常的判定

同样从预测样本的数据中抽取一个窗口期的数据，根据模型的预测数据和一个可调的阈值，判定下一段时间内的数据是否异常。

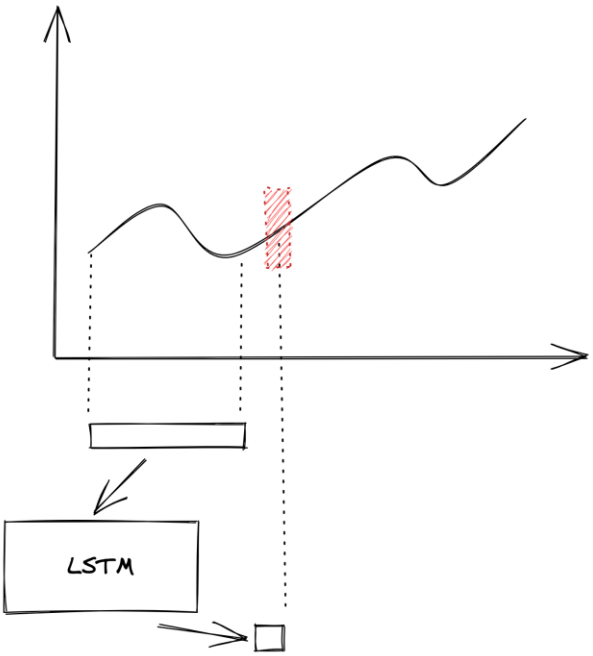

图 18 异常判定窗口示意

目前实现了基于 LSTM 的模型结构，然后根据 80 个样本的窗口预测下一个样本的情况，如果偏差超过阈值，那么认为这个样本是异常点。阈值的设定可以根据实际中事先界定的异常数据范围进行参数调整，不会影响偏差值越大，异常程度越大的基本规律。

3.3 方案的特点

我们设计的流程具有高度的灵活性，无论是模型的结构还是各种超参数都可以灵活地调整，可以针对不同系统所需的准确度和性能进行合适的定制。例如在嵌入式系统上运行的异常检测可能作为数据的初步筛选，并不需要很高的准确度，但是嵌入式系统的算力有限，因此可以使用简单的 LSTM 和低阈值进行筛选，将可能的异常信息上报后再由中心化的服务器使用 Transformer 类的大型模型对这些信息进行检测，最终确认异常的信息上报给管理人员。

4 项目测试

4.1 准确率检测

4.1.1 手动注入异常进行检测

以网络流量检测为例进行手动注入测试，对网络代理进程进行监测。监测过程从 12:00:36 开始，到 15:01:46 结束，每隔 20s 进行一次数据收集，共得到 544 条监测数据。

在 12:27-29，12:57-59，14:54-56 三个时间段，通过该代理进程进行大文件下载，将收集得到的数据进行异常检测，将阈值设定为 50000 时（根据程序输出的偏差值手动设定）输出结果如下图所示：

```
2022-05-26 12:26:58 abnomal Network traffic, 739393.9375
2022-05-26 12:27:18 abnomal Network traffic, 734510.75
2022-05-26 12:27:38 abnomal Network traffic, 723056.4375
2022-05-26 12:27:58 abnomal Network traffic, 719345.4375
2022-05-26 12:28:18 abnomal Network traffic, 715773.6875
2022-05-26 12:28:38 abnomal Network traffic, 458080.34375

2022-05-26 12:56:59 abnomal Network traffic, 702733.8125
2022-05-26 12:57:19 abnomal Network traffic, 739357.5625
2022-05-26 12:57:39 abnomal Network traffic, 747708.375
2022-05-26 12:57:59 abnomal Network traffic, 762925.375
2022-05-26 12:58:19 abnomal Network traffic, 734674.5
2022-05-26 12:58:39 abnomal Network traffic, 641838.125
2022-05-26 12:58:59 abnomal Network traffic, 742389.6875
2022-05-26 12:59:20 abnomal Network traffic, 274698.53125

2022-05-26 14:53:06 abnomal Network traffic, 74071.53125
2022-05-26 14:53:26 abnomal Network traffic, 123651.28125
2022-05-26 14:53:46 abnomal Network traffic, 109981.4765625
2022-05-26 14:54:06 abnomal Network traffic, 100126.390625
2022-05-26 14:54:26 abnomal Network traffic, 142947.34375
2022-05-26 14:54:46 abnomal Network traffic, 102278.734375
2022-05-26 14:55:06 abnomal Network traffic, 100626.0703125
2022-05-26 14:55:26 abnomal Network traffic, 105313.484375
2022-05-26 14:55:46 abnomal Network traffic, 100129.3984375
```

图 19 网络流量异常点偏差值

图 20 显示了数据的模式与检测出的异常时间点

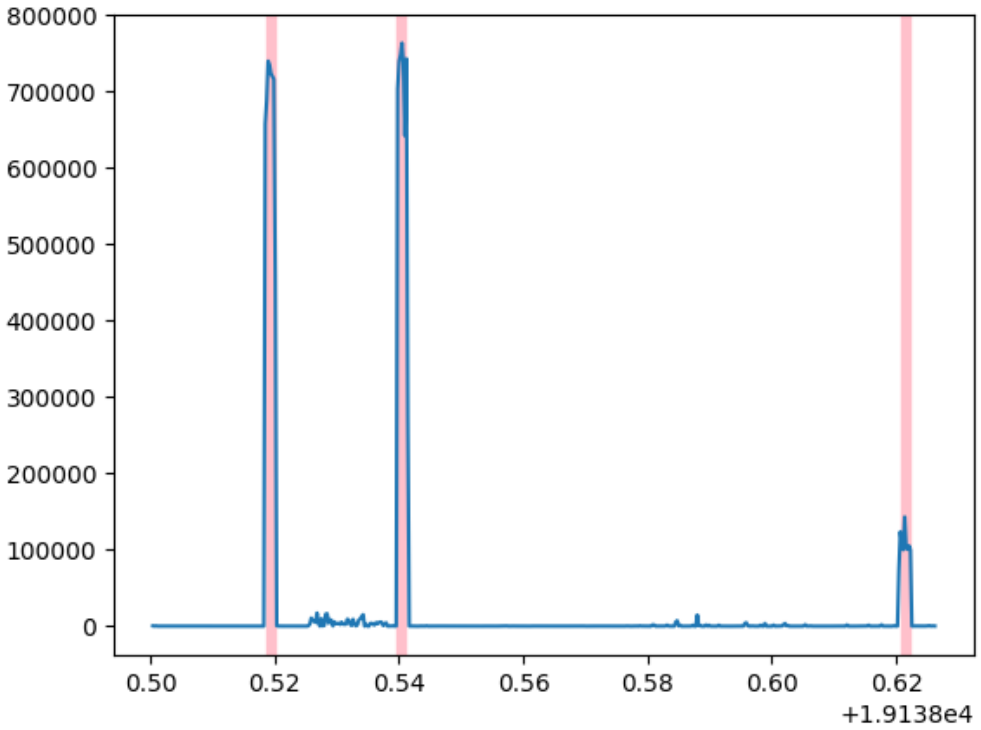

图 20 网络流量异常图表

图 21 是 LSTM 输出的 loss 与实际流量大小的对比，其中每行最右侧数值为该时刻网络流量数据的偏差值。

time	recv(kb)	send(kb)	loss_value
2022-05-26 14:51:26	148.41	41.99	72.4844
2022-05-26 14:51:46	114.8	34.48	412.9044
2022-05-26 14:52:06	455.22	43.62	224.2644
2022-05-26 14:52:26	266.58	57.62	248.1044
2022-05-26 14:52:46	290.42	40.98	147.1244
2022-05-26 14:53:06	189.44	44.33	74071.5312
2022-05-26 14:53:26	74113.84	28641.69	123651.2812
2022-05-26 14:53:46	123693.59	67785.35	109981.4765
2022-05-26 14:54:06	110023.79	47920.67	100126.3906
2022-05-26 14:54:26	100168.7	44080.52	142947.3437
2022-05-26 14:54:46	142989.66	32208.43	102278.7343
2022-05-26 14:55:06	102321.05	38612.42	100626.0703
2022-05-26 14:55:26	100668.38	35300.28	105313.4843
2022-05-26 14:55:46	105355.8	71381.86	100129.3984
2022-05-26 14:56:06	100171.71	48647.03	120.2844
2022-05-26 14:56:26	162.6	45.41	297.4344
2022-05-26 14:56:46	339.75	73.57	270.1344
2022-05-26 14:57:06	312.45	126.0	230.5144
2022-05-26 14:57:26	272.83	69.88	171.3744
2022-05-26 14:57:46	213.69	67.78	104.8144
2022-05-26 14:58:06	147.13	59.01	134.1744

图 21 正常数据与异常数据偏差值对比

从图 21 中可以看出项目针对网络流量波动具有很高的敏感程度，异常数据点的偏差值和正常数据相差可达 3 个数量级；在实际场景下，通过对 lstm 偏差阈值的调整可以实现对异常敏感程度的调整，阈值设定越小则敏感程度越高，因此具有很强的灵活性。在本次检测测试中，三次流量异常均被检出，检测的准确率可认为是 100%。

4.1.2 PyOD 准确率检测

PyOD 在本项目中的主要作用是对 lstm 检测到的异常点附近的数据进行异常检测，并将结果可视化展示。利用 LOF 算法对 lstm 检测到的 cpu 占用率异常数据进行区间长度为 40 的异常检测，得到的结果如下图所示：

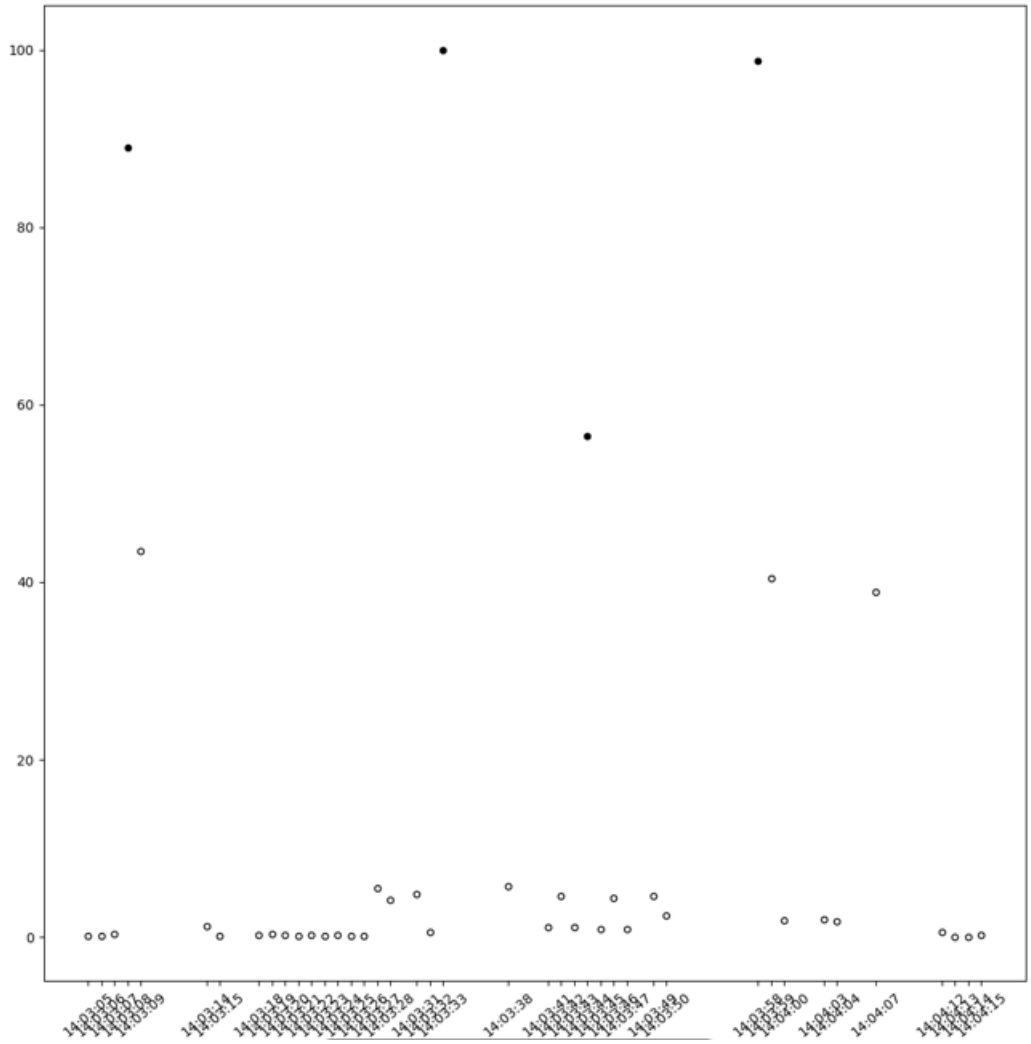

图 21 PyOD 检测结果

图中横坐标为时间，纵坐标为目标进程 cpu 占用率，黑色点即为检测到的异常点。从图中可知，在较小的区间范围内，LOF 算法可以将明显的异常数据点进行标记，对 lstm 算法起到了辅助作用。如果对 LOF 算法的分数阈值进行调整，可以将上图中 cpu 占用率在 40%左右的点也标记出来，可据此调整异常检测算法的灵敏度。

4.1.3 NAB 数据集上的准确率测试

数据集 1 是人工生成的异常测试数据，表现为周期性的 20-80 变化，其中参杂有噪声，然后在异常时间段有冲高现象。此数据集有良好的异常特征，项目异常检测部分可以有效检测异常时段。

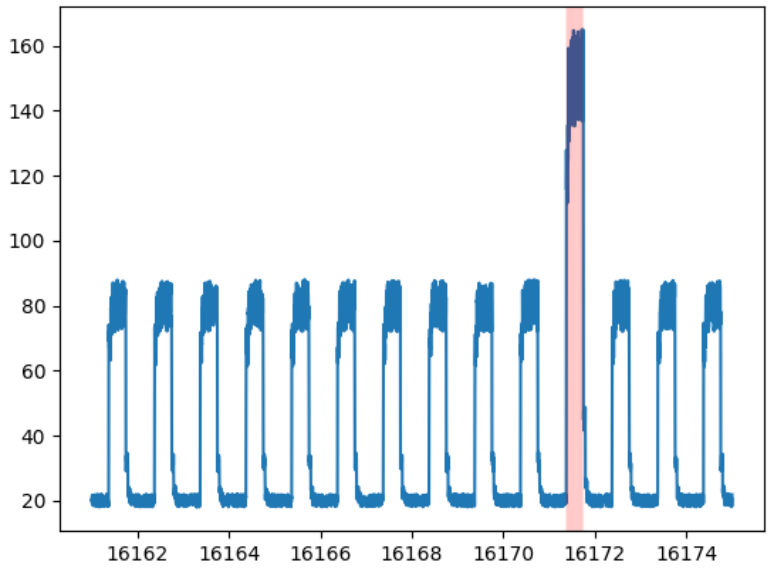

图 22 我们的结果 1

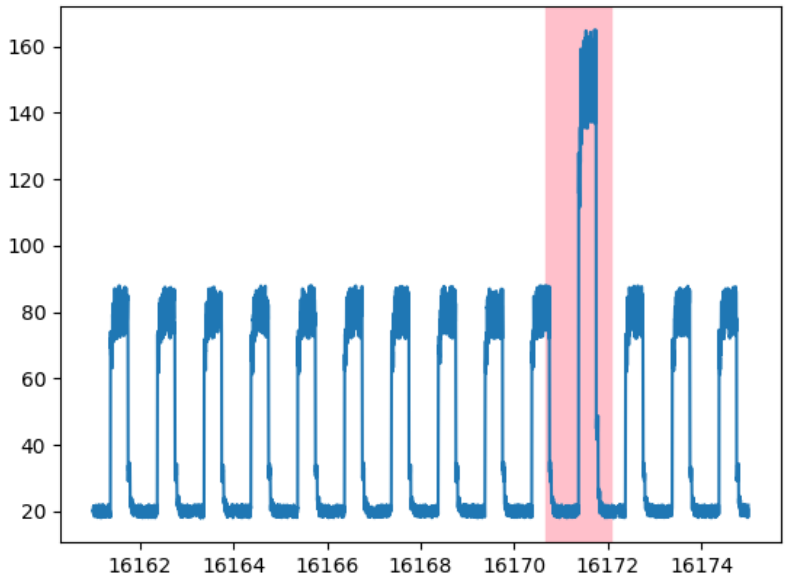

图 23 数据集标注 1

数据集 2 是 Amazon EC2 上的 CPU 占用监控数据，反映了实际生产环境中的数据情况。其中标注段有异常的高占用现象出现，本项目可以检测出 CPU 冲高的时间点。

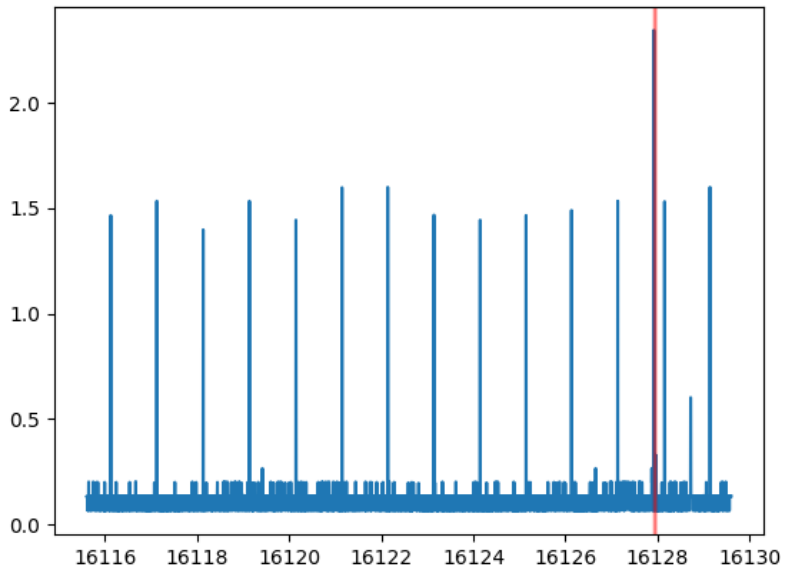

图 24 我们的结果 2

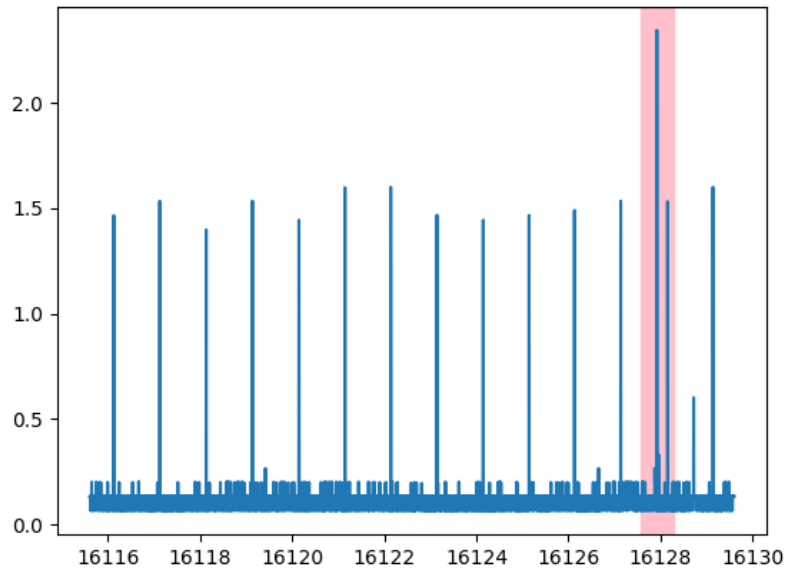

图 25 数据集标注 2

可以看到本项目基于的 LSTM 的方法在多种环境下都有良好的异常检出率。

4.2 项目实时性和资源占用检测

4.2.1 异常检测耗时

在 4.1.1 中进行的异常检测测试共计有 544 条记录数据，对其全部进行检测共耗时 4.224s，可以认为检测每条数据的时间约为 0.01s。

在实际场景中，可以自由设定时间间隔对监测模块收集的数据进行异常检测处理，并定期清理、备份收集的数据文件。例如，若设定每 30s 进行一次检测，待检测数据约两千条，则在异常发生后约 50s 可以检出。

按照上述检测速度，可以认为项目具有良好的实时异常检出能力。

4.2.2 异常检测资源占用

在 1 核 CPU，内存 1GB 的 x86 服务器上对目标进程进行监测，通过 top 命令

查看五个监测进程内存（top_snoop 和四个子模块）占用情况，可以发现项目占用内存较大。

可以看到，目前项目的内存占用较大，特别是对每个监控子模块而言，内存占用都超过 150MB。经过分析，我们认为对于每个监控子模块，其使用 BCC 框架时都会引入一个巨大的 LLVM/Clang 库用于在运行时编译程序，进而造成巨大的内存占用，这是目前 BCC 框架存在的一个问题，包括在 PingCAP 在内的商业工具以及 eBPF 社区也都发现了这个问题。后续我们将针对这个问题进行进一步的改进。

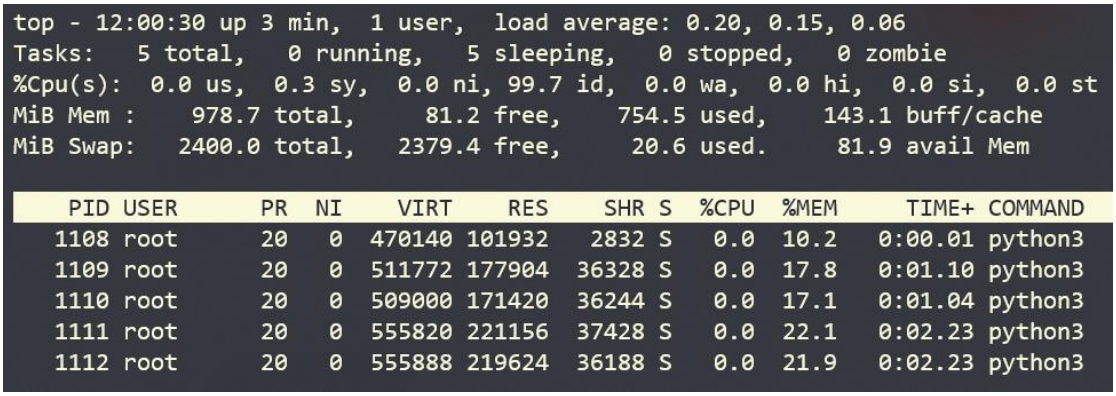

图 26 监测部分资源占用示意

对算法部分进行监测，可以发现其 cpu 占用率和内存占用率均较高，因此算法部分的调用可以根据实际情况按照一定周期调用。算法部分的运行速度较快，对系统的影响不会太大。

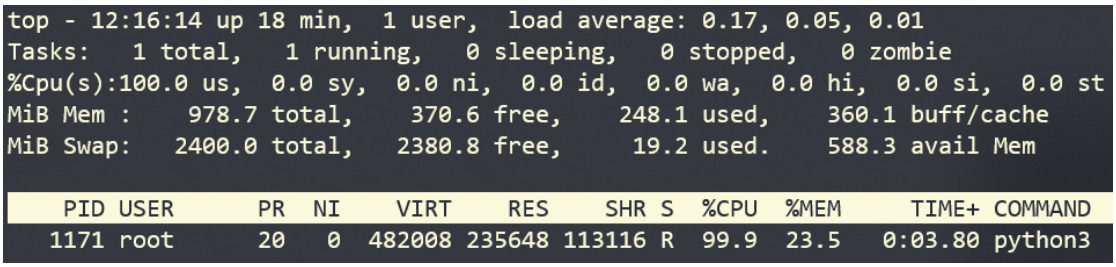

图 27 算法部分资源占用示意

5 总结与展望

目前项目已经实现了对目标进程 cpu 占用率、内存占用大小、网络收发流量大小的数据收集和异常检测方案，并拥有良好的异常检测准确率和灵活的异常敏感度设定。

与此同时，项目对内存和 cpu 资源的占用较大，算法对一些场景下的异常效果还不理想，这是后续需要优化的地方。

接下来可以在以下方面进一步完善该项目：

1. 进程的系统调用数据没有得到利用，可以考虑根据检出的异常点定位到相应时序的系统调用记录，进一步帮助进行异常分析工作；
2. 聚类算法的使用仍有较大优化空间，对异常数据点的检测准确率不够高，可能是所采取的算法和数据的分布不匹配，可以考虑采用一些更高阶的方法：如 z-sore，核密度估计，贝叶斯，卡方统计量，二阶差分，加权移动平均值以及突变点检测等；或者将采样数据进行平衡处理，对结果进行多特征判定或修改阈值等方式。
3. 项目占用内存资源较大，依赖较多，主要是由于 bcc 框架的特性造成的；eBPF 技术目前在不同版本内核之间还不能很好地兼容，会影响项目的可移植性。综上，后续可以考虑直接编写 eBPF 程序实现检测部分代码，或是采用 sysdig 等其他形式。

6 参考文献

[1] 《2021 国际 AIOps 挑战赛优秀奖伊莉丝·逐星团队方案介绍》

[2] 《Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems》

[3] 《eBPF 技术简介》

[4] 《pyod 1.0.1 documentation》

[5] NAB 数据集

[6] Automated Essay Scoring: A Siamese BidirectionalLSTM Neural Network Architecture

[7] "Why We Switched from BCC to libbpf for Linux BPF Performance Analysis,"

[8] "HOWTO:BCC to libbpf conversion BPF."