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

LMY

姓名	年级	联系方式（QQ）
于伯淳（队长）	大二	19801963591
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

2022-08

完成进度

本项目的主要目标如下：

1. 实现进程级的系统调用和资源占用（CPU、内存、网络流量）信息收集；
2. 对采集到的时序数据进行处理，从而检测到操作系统中可能发生的异常；
3. 适配不同的操作系统平台，使项目具有强泛化检测能力和高可移植性；

目前项目的完成进度如下：

目标	完成情况	说明
1	基本完成	能够对目标进程的资源占用和系统调用信息进行实时监测，并以 csv 格式记录 CPU 占用、内存占用、网络流量、系统调用的时序数据。
2	基本完成	提出了基于深度学习和压缩感知的两种异常检测算法，能够对得到的 csv 时序数据进行异常检测处理，在多种数据集下具有 80%以上准确率。
3	大致完成	项目基于 eBPF 的 python 框架 bcc，以及多个 python 算法库实现，得利于 python 的跨平台特性具有较好移植性；但 eBPF 在不同内核版本的变化会导致一些兼容问题。
总结	基本完成	相对于初赛阶段，完善了以下方面 1. 内存占用从 1G 左右优化至 200MB 以内； 2. 压缩感知算法具有更好的泛化检测能力； 3. 提供函数级异常定位解决方案 4. 增加 5 种异常内存操作告警功能 5. 项目应用于 4 个现实场景进行测试 6. 完善了项目的配置方式和使用说明；

概述

项目背景及意义

在信息化技术飞快发展的今天，计算机网络规模越来越大，无论是金融、电信、能源行业，还是工业制造、互联网、物联网等，都非常依赖网络。政府、各大企业、金融机构和科研院校所等企事业单位的业务大都建立在计算机网络之上。这为社会发展带来大量机遇的同时，也为操作系统运维带来了新的风险与挑战。

操作系统智能运维旨在提供精准的业务质量感知、支撑用户体验优化、全面提升运维服务质量。其中一个重要方面是对操作系统运行过程中发生的异常进行准确检测，为运维工作人员提供可靠的数据参考。

操作系统运行时产生的时序数据可以表征系统的运行状态，通过对这些数据进行挖掘分析，可以对系统异常运行状态进行诊断。系统运行状态监控的数据主要有 CPU 使用状态、内存使用状态、网络流量大小等，这类数据反映了操作系统内各类资源的使用情况。

Proj120 希望实现进程粒度的数据采集和异常检测。因此，收集数据和分析数据将成为本赛题的主旋律。

开发环境与项目依赖

当前项目开发环境为：

- 操作系统：Ubuntu 20.04
- 内核版本：5.4.0-122-generic
- 系统架构：x64

项目依赖主要为：

- python 3.8.10
- BCC release 0.24.0

其余 Python 算法库可参考项目仓库的 requirement.txt。项目的具体安装及使用方式可参考仓

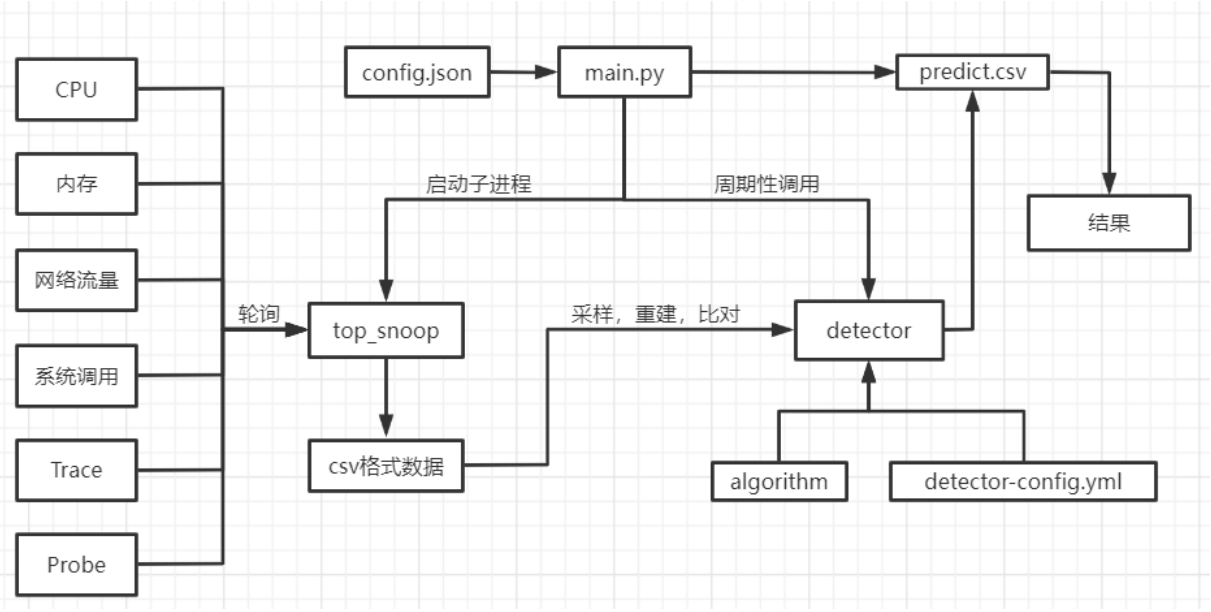
库的 README。

项目整体结构设计

项目主要包括数据收集模块和算法模块两大主要部分，通过 main.py 进行集成调度。

数据收集模块对目标进程进行实时监控，将该进程的 CPU 占用、内存占用、流量使用、系统调用等信息周期性地记录在目标文件之中。main 会根据配置文件获取收集信息的周期等配置，并启动一个子进程来调用数据收集模块。

算法模块被 main 周期性地调度以对已收集的 csv 文件中的数据进行处理，并根据结果将检测到的异常以图表的形式输出到指定的目录。整体的设计结构图如下：



数据采集模块

背景

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 程序能够加载到 trace points、内核及用户空间应用程序中的 probe points， 这种能力使我们对应用程序的运行时行为（runtime behavior）和系统本身 （system itself）提供了史无前例的可观测性。应用端和系统端的这种观测能力相结合， 能在排查系统性能问题时提供强大的能力和独特的信息。eBPF 原理图如图 1 所示。

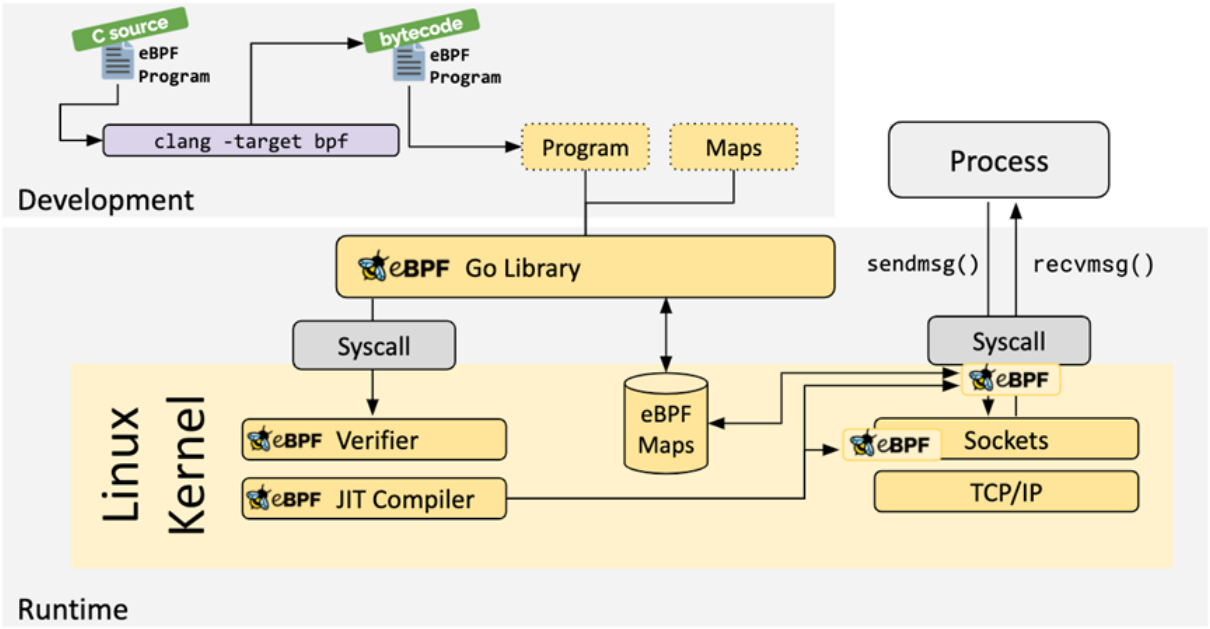


图 1 eBPF 原理图

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 装饰器），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 。在使用 BCC 编写 eBPF 程序时，eBPF 程序的 C 语言源代码会作为字符串嵌入到 Python 程序中，在运行时 BCC 再调用 Clang/LLVM 将 eBPF 程序动态编译并挂载到恰当的挂载点上。运行时编译的特性使得使用 BCC 框架编写 eBPF 程序变得非常方便简单，但同时也意味着更大的运行时资源消耗 。

功能说明

本模块使用 BCC 工具包，可实现在 Linux 操作系统内核、标准库函数、用户的自定义函数中的挂载程序，从而实现对指定进程 CPU 占用、内存占用等信息的采集。具体功能如下：

- 系统资源监控：CPU 占用信息采集，栈内存占用信息采集、网络带宽占用信息采集；
- 进程执行追踪：系统调用追踪，用户态函数追踪，用户态函数分析；
- 自定义装载模块，实现个性化信息收集需求；

模块设计

本模块由 1 个运行模块与 6 个插件模块组成，6 个插件模块可以根据需要嵌入运行模块中使用，具体设计如图 2 所示。

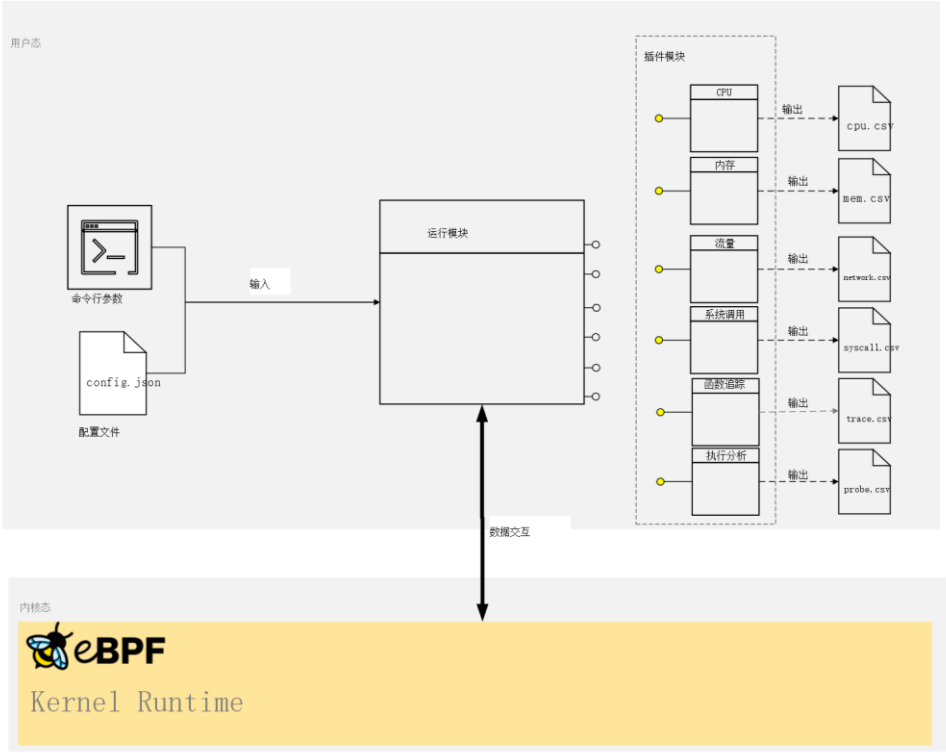


图 2 数据采集模块结构

下面分别展示运行模块与功能模块的设计，具体实现请看下节各模块的具体实现。

运行模块设计

运行模块负责读取并检查输入，装载功能模块，启动主循环并定期唤醒输出数据，如图 3 所示。

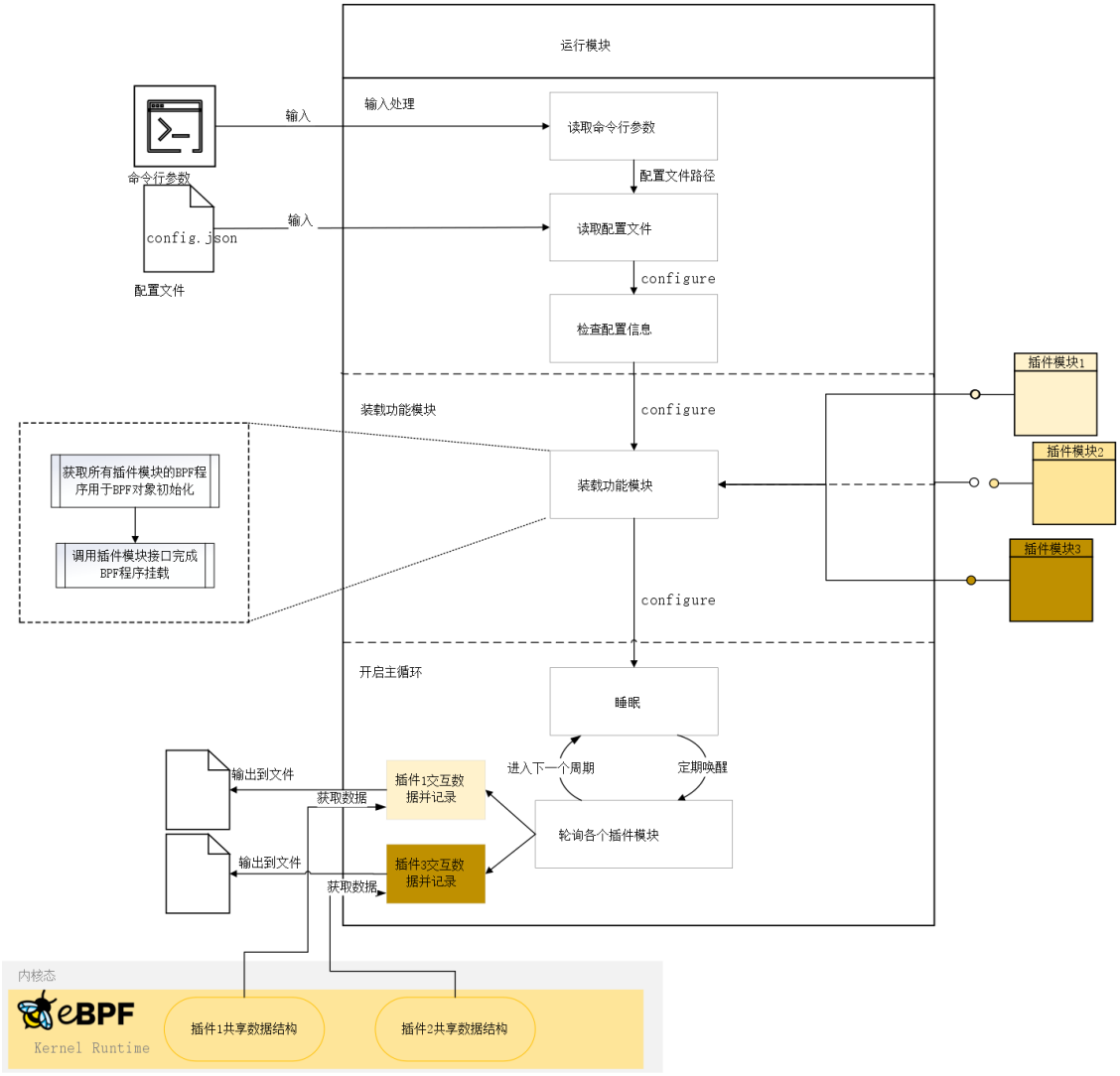


图 3 运行模块设计

输入处理

运行模块输入包含配置文件与命令行参数两部分；

1. 配置文件：

配置文件用于控制功能模块的加载，具体如图 4 所示：

```
{
    "snoop_cpu": "None",                                # 进程CPU占用监控模块，可选项为("bcc", "stat", "top", null)
    "cpu_output_file": "cpu.csv",                       # 进程CPU占用输出文件名
    "snoop_mem": "None",                                # 进程内存占用监控模块，可选项为("bcc", null)
    "mem_output_file": "mem.csv",                       # 进程内存占用输出文件名
    "probes":{                                          # 用户态函数执行情况统计，包括内存占用变化，CPU时间分布
        "event_name":["./mem_example/main:func1"],      # 检测的函数挂载点，bin:func
        "output_file":"tmp.csv",                        # 输出文件
        "spid":10000                                    # 对某个特定线程进行监控，spid为线程id，填入null表示不启用
    },
    "trace": {                                          # 用户态函数执行跟踪
        "tracee_name":["./test/search:search"],         # 检测的函数挂载点，bin:func
        "output_file":"trace_output",                   # 输出文件
        "enter_msg_format":["'Enter search, n=%d' % (arg1)"],   # 进入函数时输出的消息，支持的参数为arg1-arg6，类型为数字或字符串
        "return_msg_format":["Return search"]                   # 函数返回时输出的消息，支持的参数为retval
        "spid":10000                                    # 对某个特定线程进行监控，spid为线程id，填入null表示不启用
    },
    "snoop_network": "bcc",                             # 进程网络流量监控模块，可选项为("bcc", null)
    "network_output_file": "net.csv",                   # 进程流量监控输出文件名
    "snoop_syscall": "None",                            # 系统占用监控模块，可选项为("bcc", null)
    "syscall_output_file": "syscall.csv",               # 进程系统调用输出文件名
    "interval": 5,                                      # 监控周期，单位秒
    "trace_multiprocess":  true                         # 是否跟踪并监控进程产生的子进程（对系统调用监控输出会破坏原有顺序）
    "show_all_threads": true                            # 是否分别展示每个线程，设定为true输出文件中的PID属性改为SPID（线程号）
}
```

图 4 配置文件说明

2. 命令行参数：

命令行参数主要用于提供配置文件路径以及被监控进程的启动或寻找方式，具体如图 5 所示。

```
li@li-ThinkStation-P520:~/repository/bcc_detector/OSdetector/snoop$ python3 top_snoop.py -h
usage: top_snoop.py [-h] [-p PID] [-c COMMAND] [--configure_file CONFIGURE_FILE]

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)
  --configure_file CONFIGURE_FILE
                        File name of the configure.

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 5 命令行参数

装载功能模块

运行模块会根据配置文件确定需要装载哪些功能模块，然后将这些功能模块对应的 eBPF 程序与前端记录程序装载到主模块当中，最后分别用于 eBPF 虚拟机采集数据与 Python 前端记录数据。

启动主循环

完成前期准备工作后，模块正式进入主循环。在主循环中，模块将定期被唤醒，采用轮询的方式调用插件模块完成各自的数据获取与记录功能，并检查被监控进程的状态，如果被监控进程仍在正常运行，则继续下一个周期。

插件模块设计

插件模块具有统一的设计，通过固定的对外接口方便地实现自定义挂载。插件模块设计如图 6 所示。

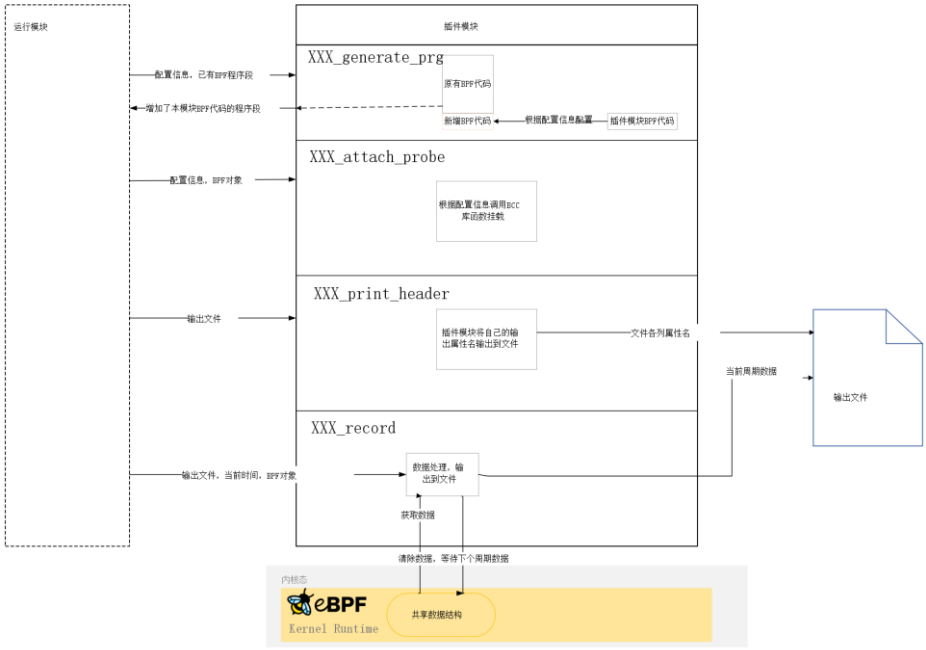


图 6 插件模块设计

1. xxx_generate_prg

根据模块需要对 eBPF 程序进行修改、替换、裁剪等，以适应本次监控需要。例如根据本次监控进程的 PID 修改过滤条件等。

2. xxx_attach_probe

将前一函数生成的 eBPF 程序挂载到相应的挂载点上。根据挂载点的不同，需要选择不同的挂载函数，由插件模块根据自己的挂载点调用 BCC 库函数实现。

3. xxx_print_header

向输出文件输出各列属性名，便于后续阅读分析。

4. xxx_record

在运行模块中被定期调用，完成每个模块自定义的记录功能。

通过统一的 4 个对外接口，由运行模块在不同阶段进行调用，从而将插件模块的功能集成到运行模块中。已有模块的实现方式请见下节。

模块实现

接下来将在上一节模块设计的基础上分别展示每个模块的具体实现方式，运行模块将由主程序部分和 eBPF 辅助函数部分组成，插件模块模块将由 eBPF 部分与前端部分组成。

运行模块

eBPF 辅助函数部分

运行模块中包含的 eBPF 辅助函数主要用于多进程程序的处理。针对监控进程是多进程程序的情况，将由被监控进程 fork 出的子进程也加入被监控进程当中，并进一步监控这些子进程是否进一步 fork 出新的子进程，以实现对整个进程树中所有进程的监控。

数据结构

BPF_HASH(snoop_proc, u32, u8);

特殊定义 1 个 hash 表用于记录某一进程是否是被监控进程，将被监控进程在 hash 表中的值设定为 1；

具体函数

1. static inline int lookup_tgid(u32 tgid)

功能说明

查看 tgid 是否在监控进程中。当配置文件中 trace_multiprocess 值为 True 时，会进一步判断当前进程的父进程是否在监控进程树中，如果是则将当前进程加入监控进程树。

输入参数

u32 tgid: 待查找进程号。

返回值

0：该进程不在被监控进程中；

1：该进程在被监控进程中；

具体实现

```
BPF_HASH(snoop_proc, u32, u8);
static inline int lookup_tgid(u32 tgid)
{
     if(snoop_proc.lookup(&tgid) != NULL)
     {
        return 1;
     }
     if(MULTI_PROCESS==true)
     {
        u8 TRUE = 1;
        struct task_struct * task = (struct task_struct
*)bpf_get_current_task();
        u32 ppid = task->real_parent->tgid;
        u32 task_tgid = task->tgid;
        if(snoop_proc.lookup(&ppid) != NULL)
        {
                snoop_proc.insert(&tgid, &TRUE);
                return 1;
        }
     }
     return 0;
}
```

2. static inline int clear_proc(struct pt_regs *ctx)

功能说明

挂载在 sched:sched_process_exit 这个挂载点上，在每个进程退出前检查其是否是被监控进程，如果是则将其从 snoop_proc 中移除，并调用已挂载的各个插件模块对应的函数处理，以清除该进程的数据。

输入参数

struct pt_regs *ctx 上下文结构体（固定参数）

返回值:

0

具体实现

```
int clear_proc(struct pt_regs *ctx)
{
    u32 tgid = bpf_get_current_pid_tgid() >> 32;
    if(lookup_tgid(tgid) == 0)
        return 0;

#ifdef _CPU_SNOOP
    clear_proc_time(ctx);
#endif
#ifdef _MEM_SNOOP
    clear_mem(ctx);
#endif
#ifdef _NETWORK_SNOOP
    clear_throughput(ctx);
#endif
    snoop_proc.delete(&tgid);

    return 0;
}
```

主程序部分

输入处理

输入处理将由 parse_args，read_configure，check_configure，print_configure 四个函数完成。

1. def parse_args()->dict: configure

功能说明

处理输入。首先使用 **argparse** 库对命令行参数进行处理读取，获取到配置文件路径后调用 **read_configure** 函数对配置文件进行读取，返回后调用 **check_configure** 函数对配置信息进行检查，如无异常则调用 **print_configure** 将本次监控的配置情况输出到标准输出。

输入参数

无

返回值

dict: configure 字典类型的配置变量，存储本次监控采集的所有相关配置信息。

2. def read_configure(str:file_name)->dict: configure

功能说明

读取 **file_name** 路径下的配置文件(.json)，使用 **json** 库处理并返回字典类型的 c**onfigure** 配置变量。

输入参数

str:file_name 配置文件所在路径

返回值

dict: configure 同上

3. def check_configure(dict: configure)->null

功能说明

检查 **configure** 变量中的各个字段的值是否有效，如果有效则顺利从函数返回，否则提示错误并终止程序。

输入参数

dict: configure 同上

返回值

无

4. def print_configure()->null

功能说明

在检查确定 c**onfigure** 变量中各字段值有效后输出各字段值。

输入参数

无

返回值

无

挂载模块

1. import 各个模块中的接口

根据 configure 中确定的插件模块，在对应的文件中将接口函数 import 到运行模块。

2. generate_prg(dict: configure)->str: prg, dict: output_fp

功能说明：

根据 **configure** 中确定的插件模块，调用这些插件模块对应的接口生成最终需要挂载的所有程序，并打开相应的输出文件存储文件指针，方便后续输出到文件。

输入参数：

dict: configure 同上

返回值：

str: prg: 最终需要挂载的 eBPF C 程序代码

dict: output_fp: 所有插件模块的输出文件指针集合

3. **attach_probes(dict: configure, obj: bpf_obj)->null**

功能说明：

根据 **configure** 中确定的插件模块，调用这些插件模块对应的接口将这些插件模块对应的 eBPF 程序挂载到挂载点上。最后，通过 BCC 的 helper 函数设置 snoop_proc 中进程树的根节点。

输入参数：

dict: configure 同上

obj:bpf_obj BCC 定义的 BPF 对象，通过调用该对象内的方法实现挂载

返回值：

无

启动主循环

1. **def main_loop(dict: configure, list: output_fp, obj: bpf_obj)->null**

功能说明：

启动循环，每隔一定时间唤醒，依次调用各个已加载模块的接口已将收集到的信息记录到文件中，并判断监控进程树的根节点进程是否正常运行，以决定是否继续进入下个周期。

输入参数

dict: configure 同上

list: output_fp 同上

obj: bpf_obj 同上

返回值

无

CPU 占用监控模块

在 CPU 占用监控模块上，我们发现初赛的方案，在 **finish_task_switch** 挂载点上插桩并记录被监控进程的执行时间，会有很大的额外性能开销，并且会造成记录的不准确，所以我们在决赛阶段又实现了另外 2 种不基于 BCC 的方案，基于读取 stat 文件与基于 top 工具，从而改善原有的问题。我们同样保留了基于 BCC 的插件模块以供选择。

BCC 实现

BCC 实现通过在 **finish_task_switch** 上挂载自定义函数，记录进程调度进出的时间，从而计算得到进程的运行与非运行时间。原理如图 7 所示。

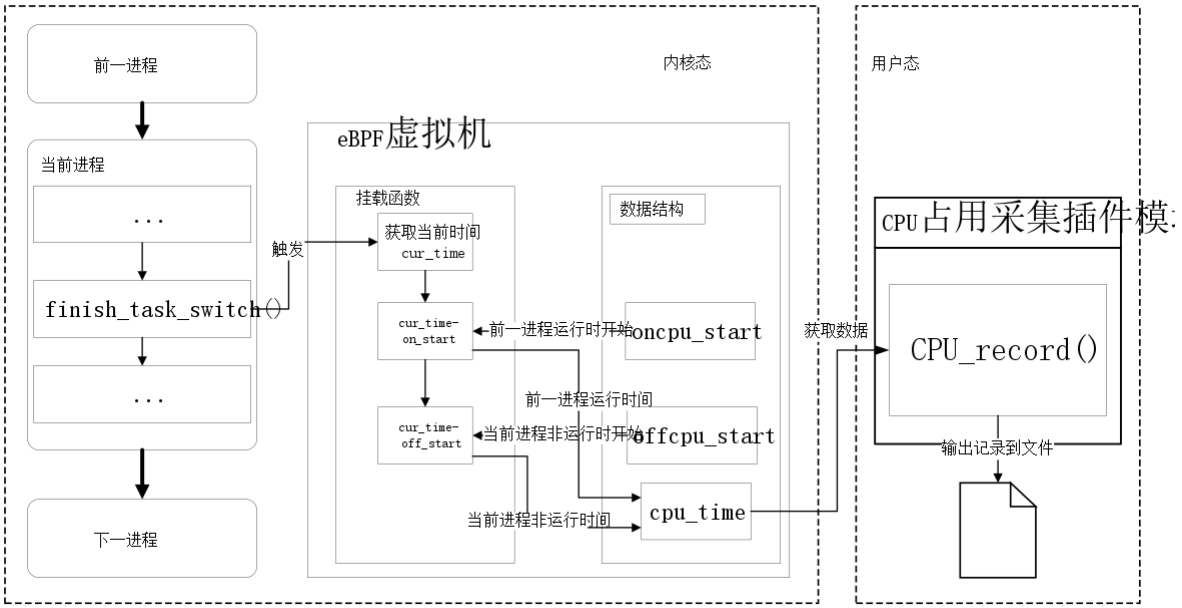


图 7 BCC 监控 CPU 占用设计

但经过我们的进一步分析，我们发现初赛的方案存在以下两处不足：

1. 进程调度在系统繁忙时可能会大量出现，经过我们的统计，**finish_task_switch** 每秒至少被触发 2000 次，这意味这我们的挂载点也将被触发至少 2000 次，这将造成很大的开销；
2. 一些长时间被阻塞的进程可能会在很多周期都很少甚至没有被调度进入 CPU 执行，从而造成我们无法得到该进程非运行时间的结束时间。这将造成某些周期数据的缺失。

所以，我们综合考虑了 eBPF 与其他的 CPU 占用率计算方案，决定不进一步开发基于 eBPF 的进程占用率采集方案，而是利用现有成熟方案获取进程的 CPU 占用率，转而将 eBPF 用于其他细粒度场景，例如下文的用户态函数追踪分析，以充分发挥 eBPF 的优势。

eBPF 部分

数据结构

```
typedef struct process_cpu_time{
    u64 oncpu_time;
    u64 offcpu_time;
}process_cpu_time;
```

进程 cpu 时间记录结构，分别记录运行时间与非运行时间

BPF_HASH(oncpu_start, u32, u64);

记录某个进程最近一次运行时间的开始，每次进程调度进入后更新

BPF_HASH(offcpu_start, u32, u64);

记录某个进程最近一次非运行时间的开始，每次进程调度离开后更新

BPF_HASH(cpu_time, u32, process_cpu_time);

记录进程的 CPU 时间，用于与前端进行数据交换；

具体函数

1. **static inline void store_oncpu_start(u32 tgid, u32 pid, u64 ts)**

功能说明

记录某次运行时间的开始，在进程被调度进 CPU 时调用。

输入参数

u32 tgid 进程组号，多线程程序中为父线程的 PID

u32 pid 线程 ID，在单线程程序中为进程号，在多线程程序中为线程号

u64 ts 当前时间戳，用于更新 **oncpu_start**

返回值

无

2. **static inline void store_offcpu_start(u32 tgid, u32 pid, u64 ts)**

功能说明

记录某次非运行时间的开始，在进程被调度出 CPU 时调用。

输入参数

u32 tgid 进程组号，多线程程序中为父线程的 PID

u32 pid 线程 ID，在单线程程序中为进程号，在多线程程序中为线程号

u64 ts 当前时间戳，用于更新 **offfcpu_start**

返回值

无

3. **static inline void update_oncpu_time(u32 tgid, u32 pid, u64 ts)**

功能说明

在进程被调度出 CPU 时调用，从 **oncpu_start** 中取出该进程本次运行时间的开始，与当前时间做差得到本次运行时长，更新本次进程的运行时间。

输入参数

u32 tgid 进程组号，多线程程序中为父线程的 PID

u32 pid 线程 ID，在单线程程序中为进程号，在多线程程序中为线程号

u64 ts 当前时间戳，用于更新 **oncpu_time**

返回值

无

4. **static inline void update_offcpu_time(u32 tgid, u32 pid, u64 ts)**

功能说明

在进程被调度进 CPU 时调用，从 **offcpu_start** 中取出该进程本次非运行时间的开始，与当前时间做差得到本次非运行时长。

输入参数

u32 tgid 进程组号，多线程程序中为父线程的 PID

u32 pid 线程 ID，在单线程程序中为进程号，在多线程程序中为线程号

u64 ts 当前时间戳，用于更新 **offcpu_time**

返回值

无

5. **int sched_switch(struct pt_regs *ctx, struct task_struct *prev)**

功能说明

挂载函数，挂载点为 **finish_task_switch**，该函数在进程每次被调度进 CPU 后马上被调用用于

清理上下文，此时可认为是新进程被调度进 CPU 与旧进程被调度出 CPU 的时间。

当进程进入该挂载点后，依次调用前述函数更新前一个进程的运行时间与当前进程的非运行时间。

输入参数

struct pt_regs *ctx 当前进程的上下文结构体，可获取进程的 pid，tgid 等信息，

struct task_struct *prev 前一个进程的上下文结构体，可获取进程的 pid，tgid 等信息，

返回值

0

具体实现

```
int sched_switch(struct pt_regs *ctx, struct task_struct *prev)
{
    u64 ts = bpf_ktime_get_ns();
    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 tgid = pid_tgid >> 32, pid = pid_tgid;

    u32 prev_pid = prev->pid;
    u32 prev_tgid = prev->tgid;

    if(snoop_proc.lookup(&prev_tgid) != NULL)
    {
        update_oncpu_time(prev_tgid, prev_pid, ts);
        store_offcpu_start(prev_tgid, prev_pid, ts);
    }

    if(lookup_tgid(tgid))
    {
        update_offcpu_time(tgid, pid, ts);
        store_oncpu_start(tgid, pid, ts);
    }

    return 0;
}
```

6. static inline int clear_proc_time(struct pt_regs *ctx)

功能说明

在进程退出时被调用，用于清楚该进程记录在本插件模块中的数据。

输入参数

struct pt_regs *ctx 上下文结构体，固定参数

返回值

0

前端部分

数据结构

bpf_obj['cpu_time'] 与 eBPF 部分共享数据结构，用于数据交互

具体函数

1. **def cpu_generate_prg(str: prg)->str: prg**

功能说明

将 CPU 监控采集部分的 eBPF 程序程序附加到 **prg** 上并返回

输入参数

str: prg: 已有的 eBPF 程序

返回值

str: prg: 加载上 CPU 监控采集部分的 eBPF 程序的 eBPF 程序

2. **def cpu_bcc_print_header(FILE: output_file)->Null**

功能说明

输出各列的属性名输出到文件中

输入参数

FILE: output_file 已打开的输出文件

返回值

无

3. **def cpu_attach_probe(obj: bpf_obj):->Null**

功能说明

调用 BCC 的 helper 函数将 eBPF 程序挂载到 CPU 监控采集模块的挂载点上

输入参数

obj: bpf_obj BCC 模块的 BPF 对象

返回值

无

4. **def cpu_record(FILE: output_file, int: period, unsigned int: time_stamp, obj: bpf_obj)->Null**

功能说明

通过 **bpf_obj['cpu_time']**数据结构获取 CPU 数据，并输出到文件中。

输入参数

FILE: output_file 已打开的输出文件

int: period 与上一次输出的时间间隔

unsigned int: time_stamp 当前时间

obj: bpf_obj 同上

返回值

无

读取 stat 文件实现

每次时间中断发生后，CPU 都会更新/proc/<pid>/stat 中的 utime，stime，cutime，cstime 等变量，以更新 CPU 的执行时间。通过定时访问该文件，记录相邻两次访问所得数据的差值即可得到在该周期内的进程运行时间。

前端部分

数据结构

dict: cpu_usage 记录运行时间，非运行时间与 CPU 占用率

具体函数

1. **def cpu_stat_record(FILE: output_file, unsigned int: cur_time, int: period, unsigned int: snoop_pid, dict: old_usage)->dict: usage**

功能说明

定期唤醒后读取监控进程的/proc/<pid>/stat 文件，获取进程运行时间，计算 CPU 占用率并输出。

输入参数

FILE: output_file 已打开的输出文件

unsigned int: cur_time 当前时间戳

int: period 距离上一次记录的时间

unsigned int: snoop_pid 监控进程号

dict: old_usage 上一次记录的数据，用于与本次数据做差得到本周期运行时间

返回值

dict: usage 本次记录的数据，用于下次传入做差

调用 top 实现

通过定期调用 top 工具获取监控进程的 CPU 占用率。

前端部分

数据结构

定期调用 top 命令即可，无需特殊的数据结构。

具体函数

1. **def cpu_top_record(FILE: output_file, unsigned int: cur_time, unsigned int: snoop_pid, bool show_all_threads=False)->Null**

功能说明

调用 top 命令获取监控进程的 CPU 占用率并输出到文件

输入参数

FILE: output_file 已打开的输出文件

unsigned int: cur_time 当前时间戳

unsigned int: snoop_pid 监控进程号

bool show_all_threads=False 是否分别展示各个线程的占用率

返回值

无

2. **def bfs_get_procs(unsigned int: snoop_pid)->list**

功能说明

获取以 **snoop_pid** 为根节点的进程树的所有进程号，实现对多进程程序的监控采集。

输入参数

unsigned int: snoop_pid 根节点进程号

返回值

list pid_list 包含进程树上所有进程的列表

具体实现

```
    def bfs_get_procs(snoop_pid):
        proc = psutil.Process(snoop_pid)
        proc_queue = queue.Queue()
        proc_queue.put(proc)
        pid_list = []
        while not proc_queue.empty():
            proc = proc_queue.get()
            pid_list.append(proc.pid)
            list(map(proc_queue.put, proc.children()))

        return pid_list
```

内存占用采集模块

内存占用采集模块具有两种实现方式，分别是基于内核挂载点与基于用户态挂载点。基于用户态挂载点可以获得更加精细的内存占用数据，细化到每一次内存申请，但对于不直接使用标准库函数申请与释放内存的程序无法采集；内核挂载点则可以完整采集所有内存占用情况，但因为内存分配器的存在，其无法精细的采集程序每一次内存申请释放。内核态内存占用采集模块与用户态内存占用采集分别如图 8，图 9 所示。

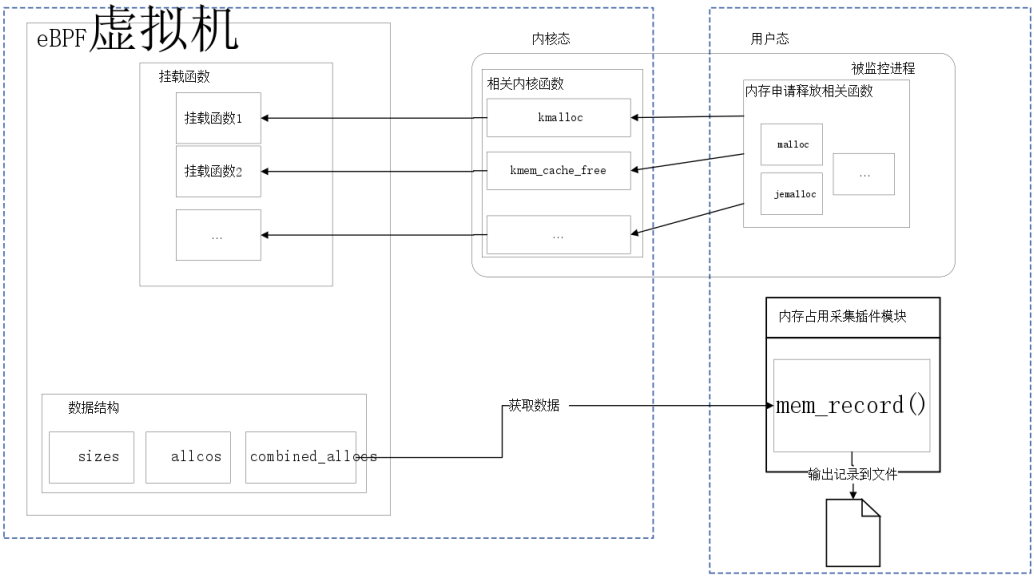


图 8 内核态内存占用采集

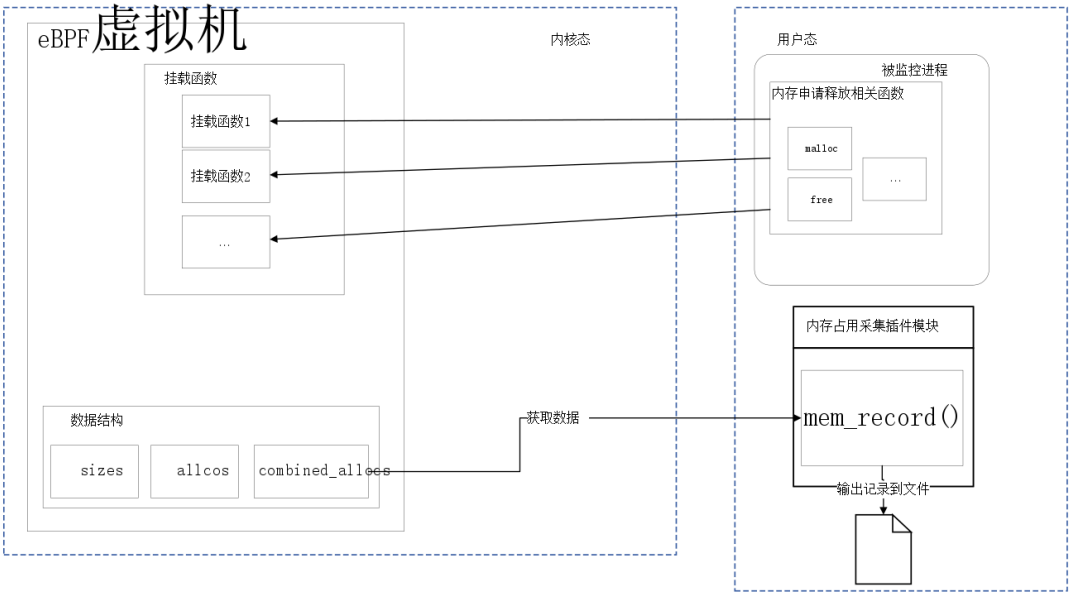


图 9 用户态内存占用采集

除挂载点不同外两种实现基本相同。

此外，为了更进一步发挥 eBPF 细粒度插桩的特性以及内存占用采集模块精细到每次内存申请释放的特性，我们在基于用户态插桩实现的内存占用采集模块中集成了对部分内存异常执行告警的功能。通过在各个插桩点中检查相关函数参数、内存地址实现告警功能。目前能够成功检测的内存异常执行如下表所示。

类型名	全称	中文	集成函数	判断方式
BFM	freeing the wrong memory block	释放错误的内存块	gen_free_enter	错误内存块地址将不在记录中
BRP	bad realloc address parameter	重分配地址参数错误	gen_free_enter	重分配将首先释放原有内存记录，此时传入的错误地址参数不在记录中
DFM	double freeing memory	重复释放内存	gen_free_enter	第二次释放时地址将不再记录中
FRP	freed realloc parameter	释放的指针传递给 realloc	gen_free_enter	重分配将首先释放原有内存记录，此时传入的已释放内存地址参数不在记

				录中
AZS	allocating zero size memory block	分配零大小内存块	gen_alloc_enter	参数为 0 引发告警

表 1 已实现异常内存操作警告

警告示例如图所示

```
[1660570066.97]Warning: Detect operation trying to alloc a zero size block , pid:1400351
[1660570067.97]Warning: Detect operation trying to alloc a zero size block , pid:1400351
free(): double free detected in tcache 2
[1660570068.98]Warning: Detect operation trying to free unknown memory, pid:1400351, address:4052c0
```

eBPF 部分

数据结构

```
struct alloc_info_t {
    u64 size;
    u64 timestamp_ns;
    int stack_id;
};
```

记录每次内存分配的信息

```
struct combined_alloc_info_t {
    u64 total_size;
    u64 number_of_allocs;
};
```

记录进程的内存占用信息

BPF_HASH(sizes, u64)

记录某个进程/线程某次分配的内存大小，方便在调用返回时获取内存申请大小

BPF_HASH(allocs, u64, struct alloc_info_t, 1000000);

记录每次内存分配信息

BPF_HASH(combined_allocs, u32, struct combined_alloc_info_t, 10240)

记录进程内存占用信息

具体函数

1. **static inline void update_statistics_add(u32 tgid, u64 sz)**

功能说明

增大进程的内存占用记录

输入参数

u32 tgid 进程号

u64 sz 内存变化大小

返回值

无

2. **static inline void update_statistics_del(u32 tgid, u64 sz)**

功能说明

减小进程的内存占用记录

输入参数

u32 tgid 进程号

u64 sz 内存变化大小

返回值

无

3. **static inline int gen_alloc_enter(struct pt_regs *ctx, size_t size)**

功能说明

内存分配调用时的挂载函数，主要记录本次内存分配的大小方便在函数调用返回时获取。因为内存优化的原因在调用函数返回时可能获取不到 **size**，所以需要提前记录 **size**。

输入参数

struct pt_regs *ctx 上下文结构体（固定参数）

size_t size 本次内存分配大小

返回值

0

4. **static inline int gen_alloc_exit(struct pt_regs *ctx, u64 address)**

功能说明

内存分配调用返回时的挂载函数，用于更新进程所占内存记录。

输入参数

struct pt_regs *ctx 上下文结构体（固定参数）

u64 address 内存分配函数返回的内存起始地址，根据该值可以判断内存申请是否成功

返回值

0

5. static inline int gen_free_enter(struct pt_regs *ctx, void *address)

功能说明

free 函数的挂载函数，用于更新进程所占内存记录。

输入参数

struct pt_regs *ctx 上下文结构体（固定参数）

void *address 释放的内存起始地址

返回值

0

以上即为内存占用采集模块的核心 eBPF 部分函数，各个挂载点将通过调用上述函数实现对进程内存占用记录的更新调整，从而获得进程的内存占用情况。

前端部分

数据结构

bpf_obj["combined_allocs"] 与 eBPF 共享数据结构用于数据交互

具体函数

1. def mem_generate_prg(str: prg, dict: configure)->str

功能说明

针对 **configure** 中 **show_all_threads** 是否为 **True** 判断使用 tgid 来表示进程还是 pid 来表示线程，替换完成后连接到已有 eBPF 程序上。

输入参数

str: prg 加载上 CPU 监控采集部分的 eBPF 程序的 eBPF 程序

dict: configure 记录本次监控记录配置信息的字典

返回值

str 附加上内存占用采集的 eBPF 程序字符串

2. def mem_attach_probe(obj:bpf_obj)->Null

功能说明

将内存采集程序挂载到对应的挂载点上。

输入参数

obj: bpf_obj 同上

返回值

无

3. def mem_print_header(File:output_file)->Null

功能说明

输出各列属性名到文件中

输入参数

FILE: output_file 已打开的输出文件

返回值

无

4. def mem_record(File: output_file, unsigned int: cur_time, obj: bpf_obj)->Null

功能说明

通过 **bpf_obj["combined_allocs"]**与 BPF 进行数据交互，并将数据输出到文件

输入参数

File: output_file 已打开的输出文件

unsigned int: cur_time 当前时间戳

obj: bpf_obj BPF 对象

返回值：

无

流量占用采集模块

流量占用采集模块原理如图 10 所示。通过在数据链路层的 2 个挂载点处挂载，实现对进程流量的采集。

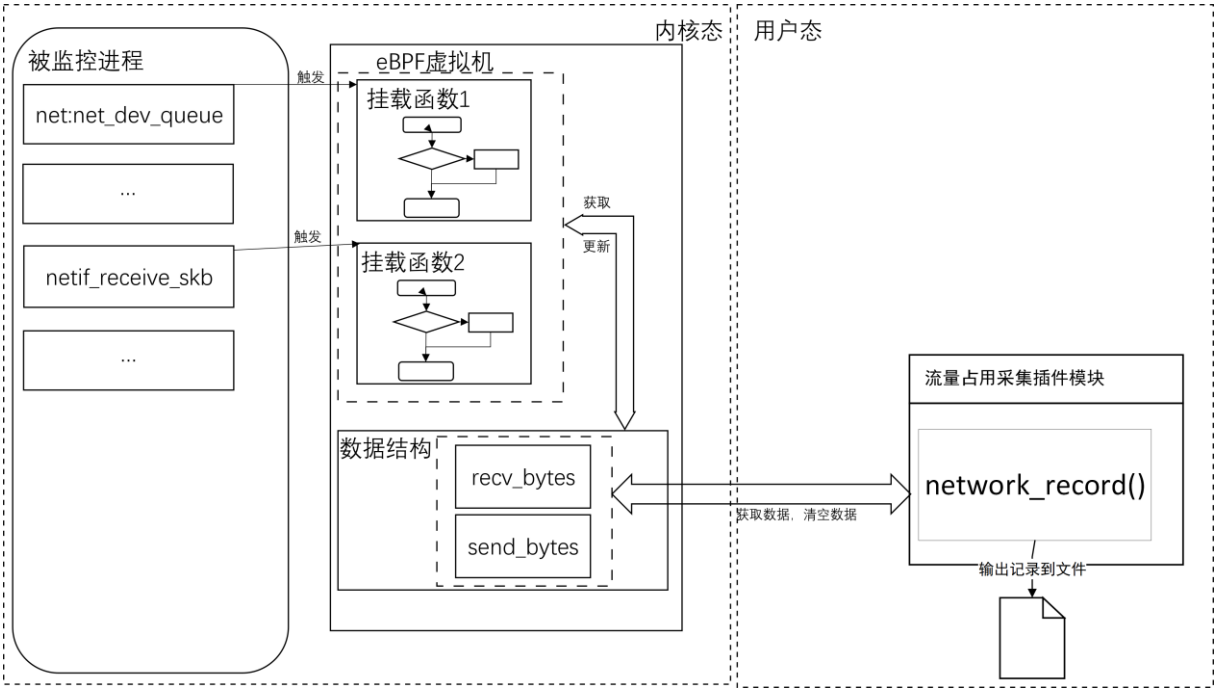


图 10 流量占用采集

eBPF 部分

数据结构

```
struct throughput_key_t {
    u32 pid;
    char name[TASK_COMM_LEN];
};
```

标识进程/线程的结构体

```
BPF_HASH(send_bytes, struct throughput_key_t);
```

记录发送流量

```
BPF_HASH(recv_bytes, struct throughput_key_t);、
```

记录接收流量

具体函数

1. TRACEPOINT_PROBE(net, net_dev_queue)

功能说明

挂载到 tracepoint net:net_dev_queue 上，用于收集数据链路层的发送流量。

输入参数

net，net_dev_queue 用于指定挂载的 tracepoint

返回值

0

2. TRACEPOINT_PROBE(net, netif_receive_skb)

功能说明

挂载到 tracepoint net:netif_receive_skb 上，用于收集数据链路层的接收流量

输入参数

net, netif_receive_skb 用于指定挂载的 tracepoint

返回值

0

3. static inline int clear_throughput(struct pt_regs *ctx)

功能说明

进程退出时调用，用于清除该进程存储在本插件模块中的数据。

输入参数

struct pt_regs *ctx 上下文结构体（固定参数）

返回值

0

前端部分

数据结构

bpf_obj['recv_bytes'] 接收流量统计

bpf_obj['send_bytes'] 发送流量统计

具体函数

1. def network_generate_prg(str: prg, bool: show_all_threads=False)->str:

功能说明

针对 **configure** 中 **show_all_threads** 是否为 True 判断使用 tgid 来表示进程还是 pid 来表示线程，替换完成后附加到已有 eBPF 程序上。

输入参数

str: prg 已有 eBPF 程序

bool: show_all_threads=False 是否按线程记录并展示数据

返回值

str 附加上流量采集插件模块的 eBPF 程序代码

2. def network_attach_probe(obj: bpf_obj)->Null:

功能说明

因为 eBPF 代码中直接使用 **TRACEPOINT_PROBE** 编写挂载程序并指定挂载点，所以在此处不需要额外再进行挂载，为保持接口统一保留该函数，但不做任何操作。

输入参数

obj: bpf_obj BPF 对象

返回值

无

3. def network_print_header(File: output_file)->Null

功能说明

向输出文件输出各列标题，方便后续阅读分析。

输入参数

File: output_file 已打开的输出文件

返回值

无

4. def network_record(File: output_file, unsigned int: cur_time, obj: bpf_obj)->Null

功能说明

定期唤醒，与 BPF 部分进行数据交互并输出到文件

输入参数

File: output_file 已打开的输出文件

unsigned int: cur_time 当前时间戳

obj: bpf_obj BPF 对象

返回值

无

系统调用追踪模块

系统调用追踪模块通过在 raw_syscalls:sys_enter 与 raw_syscalls:sys_exit 两个 tracepoint 上挂载，实现对进程系统调用的追踪，包括合适进入系统调用与从系统调用返回。sys_enter 与 sys_exit 是系统调用的总入口与总出口，每次系统调用与返回时都会经过这两个位置并携带系统调用的相关信息，使得我们能够追踪到所有系统调用。具体实现如图 11 所示。

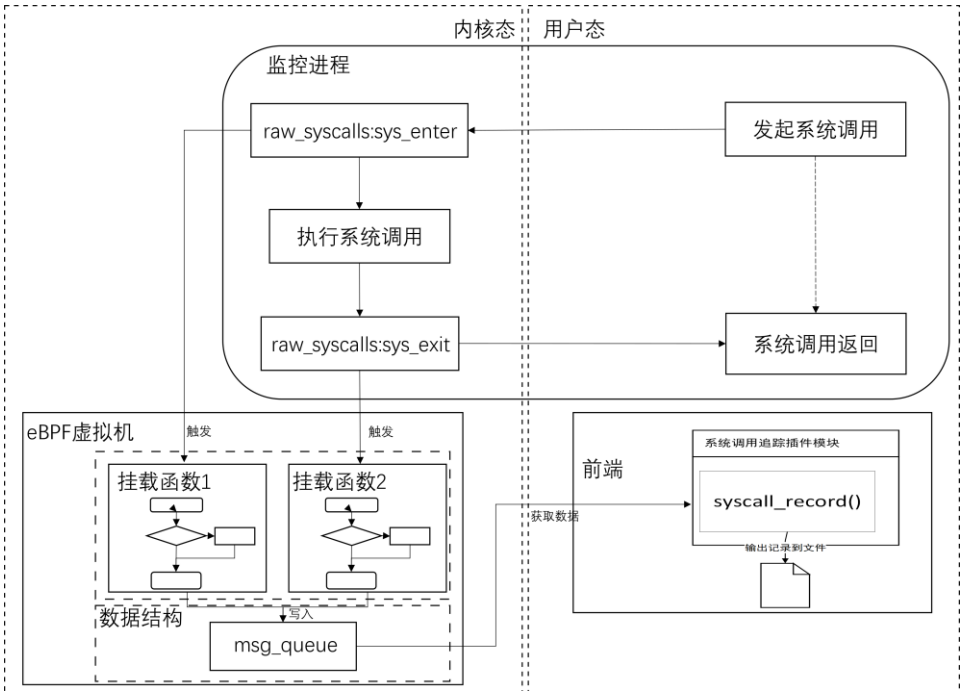


图 11 系统调用追踪

eBPF 部分

数据结构

```
struct data_t{
    enum output_type type;    // [Enter, Return]
    u32 pid;
    u64 ts;
    char comm[TASK_COMM_LEN];
    int syscall_id;
    unsigned long parm1;
    unsigned long parm2;
    unsigned long parm3;
    unsigned long parm4;
    unsigned long parm5;
    unsigned long ret;
    unsigned long fp;
    unsigned long rc;
    unsigned long sp;
    unsigned long ip;
};
```

每次系统调用的记录信息

BPF_QUEUE(message_queue, struct data_t, 10240);

消息队列，用于向外传送系统调用信息。为了满足周期性输出与保持系统调用的有序性，不直接使用 BCC 提供的输出方式，而是采用消息队列向外传送信息。

具体函数

1. TRACEPOINT_PROBE(raw_syscalls, sys_enter)

功能说明

挂载在系统调用总入口处，记录时间、调用号等信息并送入消息队列；

输入参数

输入参数用于指定挂载的 tracepoint

返回值

0

2. **TRACEPOINT_PROBE(raw_syscalls, sys_exit)**

功能说明

挂载在系统调用总出口处，记录时间、调用号等信息并送入消息队列；

输入参数

输入参数用于指定挂载的 tracepoint

返回值

0

前端部分

数据结构

bpf_obj['message_queue'] 与 BPF 部分进行数据交互的消息队列

具体函数

1. **def syscall_generate_prg(str: prg, show_all_threads=False)->str**

功能说明

针对 **configure** 中 **show_all_threads** 是否为 **True** 判断使用 tgid 来表示进程还是 pid 来表示线程，替换完成后连接到已有 eBPF 程序上。

输入参数

str: prg 已有 eBPF 程序

bool: show_all_threads=False 是否按线程记录并展示数据

返回值

str 附加上流量采集插件模块的 eBPF 程序代码

2. **def syscall_attach_probe()->Null**

功能说明

因为 eBPF 代码中直接使用 **TRACEPOINT_PROBE** 编写挂载程序并指定挂载点，所以在此处不需要额外再进行挂载，为保持接口统一保留该函数，但不做任何操作。

输入参数

obj: bpf_obj BPF 对象

返回值

无

3. **def syscall_print_header(File: output_file)->Null**

功能说明

向输出文件输出各列标题，方便后续阅读分析。

输入参数

File: output_file 已打开的输出文件

返回值

无

4. **def syscall_record(File: output_file, obj:bpf_obj)->Null:**

功能说明

定期唤醒，与 BPF 部分进行数据交互并输出到文件

输入参数

File: output_file 已打开的输出文件

obj: bpf_obj BPF 对象

返回值

无

用户态函数追踪模块

用户态函数允许用户在自己编写的程序中插桩（通过 uprobe 实现），并指定每次进入或从函数返回时的输出信息，包括进入函数时所携带的参数与函数返回时的返回值，结合时序信息，与其他采集模块得到的数据一起用于分析异常发生位置。

eBPF 部分

数据结构

```
struct TRACEE_NAME_enter_msg_data_t{
    u64 time;
    u32 tgid;
    ENTER_DATA_FIELD
```

};

进入函数后 BPF 向外传送的信息集合，其中 TRACEE_NAME 与 ENTER_DATA_FIELD 将在后续生成 BPF 程序时根据用户指定的挂载点名称与需要携带的参数填充替换。

```
struct TRACEE_NAME_return_msg_data_t{
    u64 time;
    u32 tgid;
    RETURN_DATA_FIELD
};
```

从函数返回后 BPF 向外传送的信息集合，其中 TRACEE_NAME 与 ENTER_DATA_FIELD 将在后续生成 BPF 程序时根据用户指定的挂载点名称与需要携带的参数填充替换。

BPF_QUEUE(TRACEE_NAME_enter_msg_queue, struct TRACEE_NAME_enter_msg_data_t, 10240);

进入函数向外传送信息的消息队列，根据挂载点不同将会有不同的队列名称。

BPF_QUEUE(TRACEE_NAME_return_msg_queue, struct TRACEE_NAME_return_msg_data_t, 10240);

从函数返回后向外传送信息的消息队列，根据挂载点不同将会有不同的队列名称。

具体函数

1. int TRACEE_NAME_enter(struct pt_regs* ctx)

功能说明

挂载在函数入口处，记录时间、所需参数等信息并送入消息队列；

输入参数

struct pt_regs* ctx 固定上下文结构体（固定参数），**ctx** 为上下文结构体指针，可用于获取 CPU 的寄存器内容，从而获得函数调用的参数；

返回值

0

2. int TRACEE_NAME_return(struct pt_regs *ctx)

功能说明

挂载在函数返回处，记录时间、返回值等信息并送入消息队列；

输入参数

struct pt_regs* ctx 固定上下文结构体（固定参数），**ctx** 为上下文结构体指针，可用于获取 CPU 的寄存器内容，从而获得函数调用的参数；

返回值

0

前端部分

数据结构

```
    c_type = {"u": "unsigned int", "d": "int",
            "lu": "unsigned long", "ld": "long",
            "llu": "unsigned long long", "lld": "long long",
            "hu": "unsigned short", "hd": "short",
            "x": "unsigned int", "lx": "unsigned long",
            "llx": "unsigned long long",
            "c": "char", "K": "unsigned long long",
            "U": "unsigned long long"}
```

用于根据用户给出的输出信息中的占位符确定变量类型，以决定直接通过寄存器获取变量还是需要进一步读取寄存器中存放的内存；

```
aliases_arg = {
        "arg1": "PT_REGS_PARM1(ctx)",
        "arg2": "PT_REGS_PARM2(ctx)",
        "arg3": "PT_REGS_PARM3(ctx)",
        "arg4": "PT_REGS_PARM4(ctx)",
        "arg5": "PT_REGS_PARM5(ctx)",
        "arg6": "PT_REGS_PARM6(ctx)",
        "retval": "PT_REGS_RC(ctx)",
}

aliases_indarg = {
    "arg1": "bpf_probe_read_user(&%s, sizeof(%s),
&(PT_REGS_PARM1(ctx)))",
    "arg2": "bpf_probe_read_user(&%s, sizeof(%s),
&(PT_REGS_PARM2(ctx)))",
    "arg3": "bpf_probe_read_user(&%s, sizeof(%s),
&(PT_REGS_PARM3(ctx)))",
    "arg4": "bpf_probe_read_user(&%s, sizeof(%s),
&(PT_REGS_PARM4(ctx)))",
    "arg5": "bpf_probe_read_user(&%s, sizeof(%s),
&(PT_REGS_PARM5(ctx)))",
    "arg6": "bpf_probe_read_user(&%s, sizeof(%s),
&(PT_REGS_PARM6(ctx)))",
    "retval": "bpf_probe_read_user(&%s, sizeof(%s),
&(PT_REGS_RC(ctx)))"
}
```

根据变量类型将变量名替换为相应的获取函数，具体见下 **tracer_generate_prg** 函数。

具体函数

1. **def tracer_generate_prg(str: prg, dict: configure)->str**

功能说明

根据用户指定的挂载点与记录消息生成对应的挂载程序，并添加到现有 BPF 程序末尾。

输入参数

str: prg 原有 BPF 程序代码

dict: configure 监控采集模块配置信息

返回值

str: 添加了本模块程序的 BPF 代码

函数细节

trace 模块因为需要按照用户自定义的输出格式进行输出，并且输出当中会包含函数的参数，所以需要进行更多的自定义程序定义。具体而言，函数需要从如下所示的输出格式中提取出其携带的变量：

```
    "enter_msg_format":["'Enter flushAppendOnlyFile, n=%d, str=%s' % (arg1, arg2)"],
```

函数需要从该字符串中提取出输出信息需要包含函数的第 1 个（arg1）与第 2 个形参（arg2），并根据前方占位符（%d，%s）确定变量类型，从而决定是直接读取寄存器得到参数值还是需要进一步读取用户态内存得到参数值。具体实现如下

```
    for format, type in zip((enter_msg_format, return_msg_format),
    ("enter", "return")):
        partion = format.split("\'")
        data_field = ""
        get_data = ""
        # 需要生成参数，根据用户输入生成 ebpf 部分代码
        if partion[0] == "":
            msg, param = partion[1], partion[2]
            data_type_list = []
            param_list = []
            for token in msg.split(" "):
                if "%" in token:
                    data_type_list.append(token.split("%")[1])
            for token in re.findall(r'[(](.*?)[)]',
    param.split("%")[1])[0].split(","):
                token = token.strip()
                param_list.append(token)

            for data_type, param in zip(data_type_list, param_list):
                if data_type == "s":
                    data_field += "char "+param+"[30];\n"
                    get_data += aliases_indarg[param]%("data."+param,
    "data."+param)+";\n"
                else:
                    data_field += c_type[data_type] + " " + param
    +";\n"
                  get_data +=
    "data."+param+"="+aliases_arg[param]+";\n"
```

2. def trace_attach_probe(obj: bpf_obj, dict: configure)->Null

功能说明

将本模块所有挂载程序挂载到对应的挂载点上。

输入参数

obj: bpf_obj BPF 对象

dict: configure 监控采集模块的配置信息

返回值

无

3. **def trace_print_header(File: output_file)->Null**

功能说明

向输出文件输出各列标题，方便后续阅读分析。

输入参数

File: output_file 已打开的输出文件

返回值

无

4. **def trace_record(File: output_file, obj: bpf_obj, dict: configure)**

功能说明

定期唤醒，与 BPF 部分进行数据交互并输出到文件

输入参数

File: output_file 已打开的输出文件

obj: bpf_obj BPF 对象

返回值

无

用户态函数执行分析模块

用户态函数执行分析模块可根据进入与返回指定用户态函数的状态变化，记录用户态函数执行期间中是否存在内存泄露、CPU 时间的分布情况、上下文切换的情况等，便于用户定位异常所在位置。

eBPF 部分

数据结构

```
struct probe_data_t {
    u64 total_size;
    u64 number_of_allocs;
    u64        utime;          //用户态消耗的 CPU 时间
    u64        stime;          //内核态消耗的 CPU 时间
    unsigned long      nvcsw; //自愿(voluntary)上下文切换计数
    unsigned long      nivcsw;//非自愿(involuntary)上下文切换计数
    u64 start_time;
    u64 end_time;
    u64 cost_time;
    char event_name[30];
    u32 tgid;
};
```

记录函数执行情况的结构体

BPF_QUEUE(probe_message_queue, struct probe_data_t, 1024);

与前端进行数据交互的消息队列；

BPF_HASH(EVENT_NAME_enter_status, u32, struct probe_data_t);

暂存进入 probe 时的状态（CPU 时间，内存占用）

具体函数

1. int EVENT_NAME_uprobe(struct pt_regs *ctx)

功能说明

挂载到指定用户态函数入口的函数，收集进入函数时的状态并暂存到 **EVENT_NAME** 中，当进入函数返回的挂载点时再取出。

输入参数

struct pt_regs *ctx 上下文结构体（固定参数）

返回值

0

2. **int EVENT_NAME_uretprobe(struct pt_regs *ctx)**

功能说明

挂载到指定用户态函数返回处的函数，收集函数返回时的状态，从 **EVENT_NAME** 中取出进入函数时的状态，做差得到函数执行期间的信息并通过消息队列向前端传送。

输入参数

struct pt_regs *ctx) 上下文结构体（固定参数）

返回值

0

前端部分

数据结构

bpf_obj["probe_message_queue"] 消息队列，用于与 BPF 进行数据交互

具体函数

1. **def probe_generate_prg(str: prg, dict:configure)->str:**

功能说明

根据用户指定的挂载函数生成相应的 BPF 程序并连接到已有 BPF 程序末尾。

输入参数

str: prg 已有 BPF 程序

dict:configure 监控采集模块配置信息

返回值

str 连接上本模块代码的 BPF 程序

2. **def attach_uprobe(obj: bpf_obj, dict: configure)->Null:**

功能说明

根据配置文件将所有用户态分析程序挂载到对应的挂载点上。

输入参数

obj: bpf_obj BPF 对象

dict: configure 监控采集模块配置信息

返回值

无

3. **def probe_print_header(File: output_file)->Null**

功能说明

向输出文件输出各列标题，方便后续阅读分析。

输入参数

File: output_file 已打开的输出文件

返回值

无

4. **def uprobe_record(File: output_file, obj: bpf_obj)->Null**

功能说明

定期唤醒从消息队列中获取数据并输出到文件

输入参数

File: output_file 已打开的输出文件

obj: bpf_obj BPF 对象

返回值

无

工程静态分析工具

在通过我们的检测工具检测到异常后，如果我们希望进一步使用监控采集工具帮助我们定位异常发生地，那我们首先要确定一些可能发生异常的函数来进行插桩分析。但是，现实运行的系统可能非常复杂，从一个拥有上百个文件的软件项目中定位出一个可能出问题的函数可能会非常困难。针对这个问题，受到相关异常检测论文的启发，我们开发了一个简单的静态分析工具，用于帮助快速找到脆弱函数用于进一步分析。

我们的静态分析工具基于 Clang LibTooling 开发实现。Clang 是 LLVM 项目的编译器前端，用于分析源代码生成语法分析树，以快速编译和较少的内存占用著称。并且，为了让 Clang 强大的语法分析功能得到更加充分的利用，clang 还开放了多种方式让我们能够利用其语法分析功能实现许多自定义的函数，而 LibTooling 则是其中的一种方式。LibTooling 是一个 C++接口，通过 LibTooling 能够编写独立运行的语法检查和代码重构工具，且可以作为一个命令单独使用，并完全控制 Clang AST（抽象语法树），从而实现代码语言转换、代码分析、规范、重构的功能。

具体来说，我们通过 LibTooling 提供的接口可以得到程序的语法分析树，根据给定函数名定位

到调用该函数的节点后，寻找其最近的函数定义类型的父节点，即可找到调用该函数的函数名称。具体表达式如下：

```
    Finder.addMatcher(
              functionDecl(
                hasDescendant(
                  callExpr(callee(

    functionDecl(hasName(func_name)).bind("func_name"))
                  )
                )
              ).bind(res.type_list[i]),
                Callback
            );
```

效果展示

简单异常检测

我们首先使用两个自行编写的简单程序对用户态函数追踪与分析模块进行测试。

CPU 占用异常冲高

异常说明

程序如下所示.

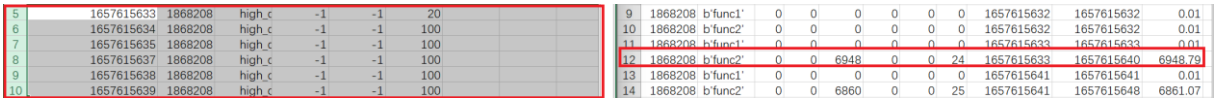
```
# include <stdio.h>
# include <unistd.h>
#include <stdlib.h>
int func1()
{
    int n = rand() % 20 - 3;  // -3-16
    return n;
}

void func2(int n)
{
    for(; n !=0; n--);
    return;
}


/**
 * @brief func1 产生 func2 的循环次数，但 func2 的循环边界默认次数>=0，进而导致
 * 传入的数字小于 0 时进入长循环直到数字溢出再减为 0
 * 表现出来的现象为不定期 CPU 冲高
 *
 * @return int
 */
int main(void)
{
    while(1)
    {
        int n = func1();
        func2(n);
        sleep(1);
    }
}
```

该程序用于模拟服务器行为。在一个定时唤醒的睡眠周期中先由 **func1** 产生一个数据，再交由 **func2** 对其进行处理。但是，由于 **func2** 缺少对输入异常值的检测，导致其可能陷入异常循环中，造成 CPU 占用异常冲高。

分析

使用用户态函数分析与追踪模块对 **func1** 与 **func2** 两个函数进行插桩分析。

比较 CPU 占用数据与用户态函数分析数据（如下图），

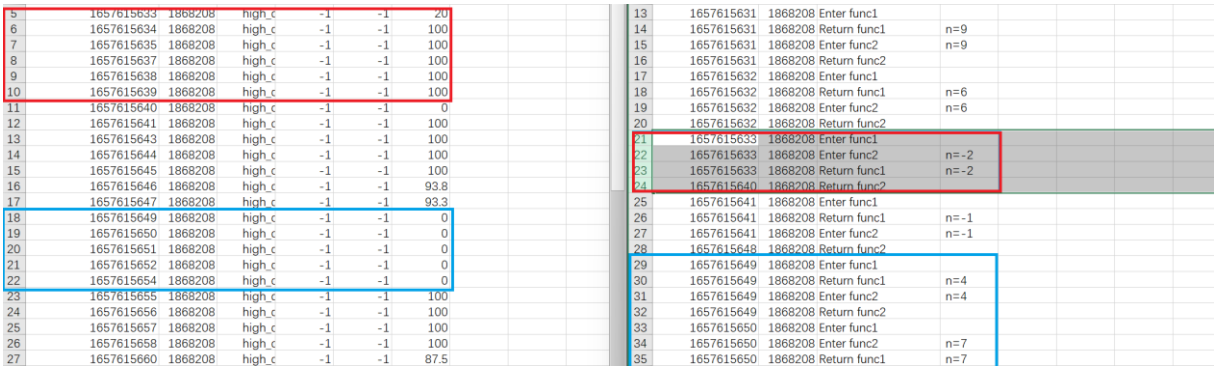


5	1657615633	1868208	high_c	-1	-1	20
6	1657615634	1868208	high_c	-1	-1	100
7	1657615635	1868208	high_c	-1	-1	100
8	1657615637	1868208	high_c	-1	-1	100
9	1657615638	1868208	high_c	-1	-1	100
10	1657615639	1868208	high_c	-1	-1	100

9	1868208	b'func1'	0	0	0	0	0	0	1657615632	1657615632	0.01
10	1868208	b'func2'	0	0	0	0	0	0	1657615632	1657615632	0.01
11	1868208	b'func1'	0	0	0	0	0	0	1657615633	1657615633	0.01
12	1868208	b'func2'	0	0	6948	0	0	24	1657615633	1657615640	6948.79
13	1868208	b'func1'	0	0	0	0	0	0	1657615641	1657615641	0.01
14	1868208	b'func2'	0	0	6860	0	0	25	1657615641	1657615648	6861.07

可以看到，在 CPU 占用较高期间，进程恰好位于 func2 函数中执行，从而推断异常位置在 func2 中。且在运行时间分布上，大部分时间位于用户态执行，且上下文切换为非自愿上下文切换，即进程时间片耗尽被调度出，我们可以推测是用户态函数逻辑导致 CPU 异常。

进一步分析用户态函数最终模块输出，如下图所示。



5	1657615633	1868208	high_c	-1	-1	20
6	1657615634	1868208	high_c	-1	-1	100
7	1657615635	1868208	high_c	-1	-1	100
8	1657615637	1868208	high_c	-1	-1	100
9	1657615638	1868208	high_c	-1	-1	100
10	1657615639	1868208	high_c	-1	-1	100
11	1657615640	1868208	high_c	-1	-1	0
12	1657615641	1868208	high_c	-1	-1	100
13	1657615643	1868208	high_c	-1	-1	100
14	1657615644	1868208	high_c	-1	-1	100
15	1657615645	1868208	high_c	-1	-1	100
16	1657615646	1868208	high_c	-1	-1	93.8
17	1657615647	1868208	high_c	-1	-1	93.3
18	1657615649	1868208	high_c	-1	-1	0
19	1657615650	1868208	high_c	-1	-1	0
20	1657615651	1868208	high_c	-1	-1	0
21	1657615652	1868208	high_c	-1	-1	0
22	1657615654	1868208	high_c	-1	-1	0
23	1657615655	1868208	high_c	-1	-1	100
24	1657615656	1868208	high_c	-1	-1	100
25	1657615657	1868208	high_c	-1	-1	100
26	1657615658	1868208	high_c	-1	-1	100
27	1657615660	1868208	high_c	-1	-1	87.5

13	1657615631	1868208	Enter func1
14	1657615631	1868208	Return func1 n=9
15	1657615631	1868208	Enter func2 n=9
16	1657615631	1868208	Return func2
17	1657615632	1868208	Enter func1
18	1657615632	1868208	Return func1 n=6
19	1657615632	1868208	Enter func2 n=6
20	1657615632	1868208	Return func2
21	1657615633	1868208	Enter func1
22	1657615633	1868208	Enter func2 n=-2
23	1657615633	1868208	Return func1 n=-2
24	1657615640	1868208	Return func2
25	1657615641	1868208	Enter func1
26	1657615641	1868208	Return func1 n=-1
27	1657615641	1868208	Enter func2 n=-1
28	1657615648	1868208	Return func2
29	1657615649	1868208	Enter func1
30	1657615649	1868208	Return func1 n=4
31	1657615649	1868208	Enter func2 n=4
32	1657615649	1868208	Return func2
33	1657615650	1868208	Enter func1
34	1657615650	1868208	Enter func2 n=7
35	1657615650	1868208	Return func1 n=7

可以看到，在 CPU 占用冲高的 func2 执行期间，func2 的输入参数变成了负数，进而导致 func2 执行异常。

通过上述方式，我们可以确定异常原因为 func2 因为输入参数异常导致执行一场，进一步查看函数实现，可以发现，当输入参数为负数时，循环会从负数开始，直到变量 n 溢出变成正数，再逐渐减小到 0，造成类似死循环的情况，从而导致异常发生。

CPU 利用率较低异常

异常说明

异常序如下所示。

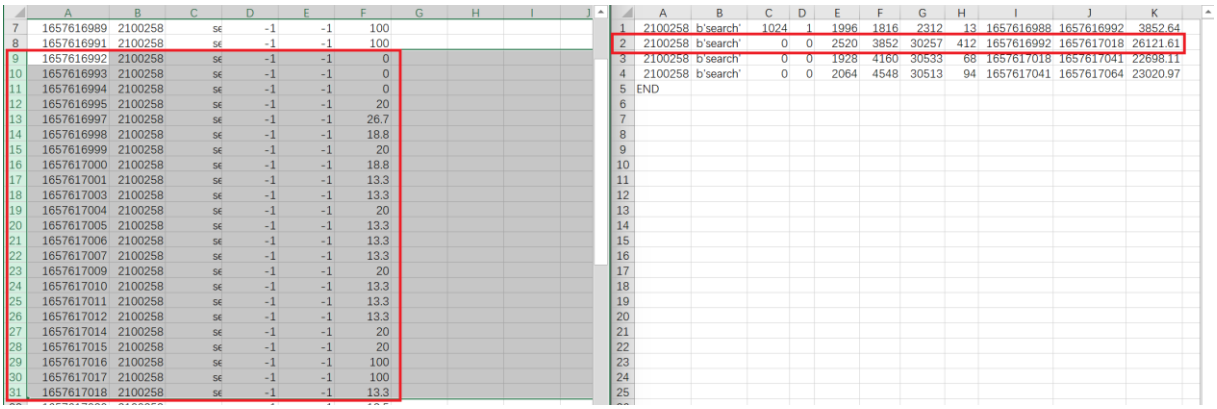
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define N 1000000

void search(int n)
{
    int cnt = 0;
    int* a = malloc(sizeof(int)*N);
    char filename[30] = "/home/li/data/search_file";
    char num[3];
    sprintf(num, "%d", n);
    strcat(filename, num);
    FILE* fp = fopen(filename, "r");
    fread(a, 1, sizeof(int)*N, fp);
    time_t t2 = time(NULL);
    for(int j = 0; j < N; j++)
    {
        if(a[j] == 10086)
            cnt++;
    }
    free(a);
    fclose(fp);
    printf("%d\n", cnt);
}

int main(void)
{
    sleep(10);
    for(int i = 4; i < 8; i++)
        search(i);

    return 0;
}
```

该异常程序模拟的是文本搜索任务，即在文本文件中搜索特定字符出现的次数。首先读取文件，然后遍历文件寻找匹配字符。但在实际使用中，发现程序偶尔会出现 CPU 占用较低的情况。

分析

使用用户态函数追踪与分析模块，结合 CPU 占用数据综合分析如下。

首先综合分析用户态函数分析与 CPU 占用情况，如下图所示。



	A	B	C	D	E	F
7	1657616989	2100258	s€	-1	-1	100
8	1657616991	2100258	s€	-1	-1	100
9	1657616992	2100258	s€	-1	-1	0
10	1657616993	2100258	s€	-1	-1	0
11	1657616994	2100258	s€	-1	-1	0
12	1657616995	2100258	s€	-1	-1	20
13	1657616997	2100258	s€	-1	-1	26.7
14	1657616998	2100258	s€	-1	-1	18.8
15	1657616999	2100258	s€	-1	-1	20
16	1657617000	2100258	s€	-1	-1	18.8
17	1657617001	2100258	s€	-1	-1	13.3
18	1657617003	2100258	s€	-1	-1	13.3
19	1657617004	2100258	s€	-1	-1	20
20	1657617005	2100258	s€	-1	-1	13.3
21	1657617006	2100258	s€	-1	-1	13.3
22	1657617007	2100258	s€	-1	-1	13.3
23	1657617009	2100258	s€	-1	-1	20
24	1657617010	2100258	s€	-1	-1	13.3
25	1657617011	2100258	s€	-1	-1	13.3
26	1657617012	2100258	s€	-1	-1	13.3
27	1657617014	2100258	s€	-1	-1	20
28	1657617015	2100258	s€	-1	-1	20
29	1657617016	2100258	s€	-1	-1	100
30	1657617017	2100258	s€	-1	-1	100
31	1657617018	2100258	s€	-1	-1	13.3

	A	B	C	D	E	F	G	H	I	J	K
1	2100258	b'search'	1024	1	1996	1816	2312	13	1657616988	1657616992	3852.64
2	2100258	b'search'	0	0	2520	3852	30257	412	1657616992	1657617018	26121.61
3	2100258	b'search'	0	0	1928	4160	30533	68	1657617018	1657617041	22698.11
4	2100258	b'search'	0	0	2064	4548	30513	94	1657617041	1657617064	23020.97
5	END										

可以看到，在程序 CPU 占用较低期间，内核态运行时间相比正常运行的状态显著增加，且资源上下文切换次数变多。据此我们可以推断程序可能在 IO 上被阻塞，导致 CPU 利用率不高。

现实软件异常检测

下面我们将用一个 redis 异常现象的定位过程来展示异常采集模块的功能。

运行环境说明

操作系统：Ubuntu 20.04

redis 版本：2.4.11

编译器：gcc 4.8.5

异常说明

本次异常选用 redis 的 GitHub 仓库的 issue641，该异常由 redis 的 AOF 持久化功能引起。打开 AOF 持久化功能后， Redis 处理完每个事件后会调用 write(2)将变化写入 kernel 的 buffer，如果此时 write(2)被阻塞，Redis 就不能处理下一个事件，直到 write(2)正常返回。而 Linux 规定执行 write(2)时，如果对同一个文件正在执行 fdatasync(2)将 kernel buffer 写入物理磁盘，或者有 system wide sync 在执行，write(2)会被阻塞，整个 Redis 被阻塞。如果系统 IO 繁忙，比如有别的应用在写盘，或者 Redis 自己在 AOF rewrite 或 RDB snapshot(虽然此时写入的是另一个临时文件，虽然各自都在连续写，但两个文件间的切换使得磁盘磁头的寻道时间加长)，就可能导致 fdatasync(2)迟迟未能完成从而阻塞 write(2)，阻塞住整个 Redis。

所以，在启动 Redis 之后，运行下面的脚本制造大量的磁盘写入，即可制造出异常导致 Redis

被阻塞。

```
import os
import multiprocessing
from time import sleep

def func(dir):
    for i in range(5):
        print("writing to file %s/%d!\n"  % (dir, i))
        os.system("dd if=/dev/zero of=%s/%d bs=1G count=20" %
(dir, i))

    for i in range(5):
        print("deleting file %d" % i)
        os.system("rm %s/%d" % (dir, i))
     sleep(1000)

if __name__=="__main__":
    loc = "/path/to/store/trashbin"
    procs = []
    for i in range(10):
        p = multiprocessing.Process(target=func,
args=(loc+str(i), ))
        procs.append(p)

    for p in procs:
        p.start()
    try:
        sleep(100000)
    except KeyboardInterrupt:
        for p in procs:
            p.terminate()
```

之后 redis 在响应请求时出现卡死的情况，如图 12 所示。

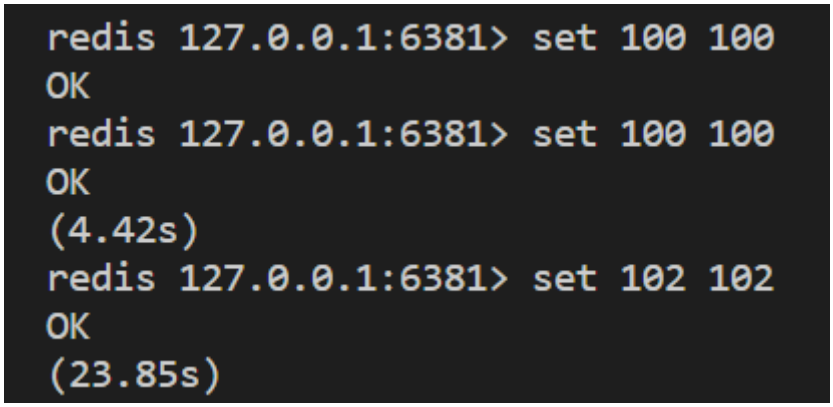


图 12 redis 卡顿

分析

下面将逐步展示如何利用监控采集模块定位到前述异常。

首先查看 CPU 占用的数据，如图 13 所示。

TIME	PID	COMM	ON CPU	OFF CPU	CPU%
1.66E+09	1328278	redis-server	0.23	1000.8	0.02
1.66E+09	1328278	redis-server	0.27	1005.06	0.03
1.66E+09	1328278	redis-server	0.46	1001.3	0.05
1.66E+09	1328278	redis-server	0.3	1002.83	0.03
1.66E+09	1328278	redis-server	0.39	1002.81	0.04
1.66E+09	1328278	redis-server	0.36	1003.05	0.04
1.66E+09	1328278	redis-server	0.36	1001.9	0.04
1.66E+09	1328278	redis-server	0.38	1002.92	0.04
1.66E+09	1328278	redis-server	0.47	1002.18	0.05
1.66E+09	1328278	redis-server	0.48	1001.94	0.05
1.66E+09	1328278	redis-server	0.35	1002.88	0.04
1.66E+09	1328278	redis-server	0.27	1002.79	0.03
1.66E+09	1328278	redis-server	0.44	1002.88	0.04
1.66E+09	1328278	redis-server	0.36	1002.98	0.04
1.66E+09	1328278	redis-server	0.35	1002.67	0.03
1.66E+09	1328278	redis-server	0.51	1002.43	0.05
1.66E+09	1328278	redis-server	0.39	1002.92	0.04
1.66E+09	1328278	redis-server	0.38	1002.64	0.04
1.66E+09	1328278	redis-server	0.14	1003.93	0.01
1.66E+09	1328278	redis-server	0.31	1002.36	0.03
1.66E+09	1328278	redis-server	0.36	1001.46	0.04
1.66E+09	1328278	redis-server	0.52	1002.22	0.05
1.66E+09	1328278	redis-server	0.47	1002.64	0.05
1.66E+09	1328278	redis-server	0.26	1003.02	0.03
1.66E+09	1328278	redis-server	0.38	1002.74	0.04
1.66E+09	1328278	redis-server	0.36	1001.93	0.04
1.66E+09	1328278	redis-server	0.39	1003	0.04
1.66E+09	1328278	redis-server	0.5	1002.8	0.05
1.66E+09	1328278	redis-server	0.4	1003.04	0.04
1.66E+09	1328278	redis-server	0.17	1003.13	0.02
1.66E+09	1328278	redis-server	0.41	1001.79	0.04
1.66E+09	1328278	redis-server	0.36	1002.76	0.04
1.66E+09	1328278	redis-server	0.48	1002.49	0.05
1.66E+09	1328278	redis-server	0.52	1002.67	0.05
1.66E+09	1328278	redis-server	0.41	1002.36	0.04

图 13 CPU 占用数据

可以看到，CPU 占用保持在较低水平，推测可能是进程被阻塞。

进一步在用户态函数进行插桩，使用静态分析工具寻找调用了脆弱函数的用户态函数，输出如图 14 所示。

```
================================================================================
Type:
IO

Location:
anet.c:7282
anetWrite

Function Name:
write

================================================================================

================================================================================
Type:
IO

Location:
aof.c:2781
flushAppendOnlyFile

Function Name:
write

================================================================================

================================================================================
Type:
IO

Location:
aof.c:29051
backgroundRewriteDoneHandler

Function Name:
write

================================================================================
```

图 14 静态分析工具输出

经过逐个排除，发现异常点位于 **flushAppendOnlyFile** 函数。插桩分析过程如下。

首先配置插桩点。配置文件如图 15 所示所示：

```
"probes":{
    "event_name":["/home/li/data/repo/redis-2.4.11/src/redis-server:flushAppendOnlyFile"],
    "output_file":"probe.csv",
    "spid": null
},
"trace": {
    "tracee_name":["/home/li/data/repo/redis-2.4.11/src/redis-server:flushAppendOnlyFile"],
    "output_file":"trace.csv",
    "enter_msg_format":["Enter flushAppendOnlyFile"],
    "return_msg_format":["Return flushAppendOnlyFile"],
    "spid": null
},
```

图 15 插桩配置

采集一段时间后结束采集，查看 probe 的输出结果，如图 16 所示。

PID	EVENT NA	SIZE	NUM	UTime	STime	NVCSW	NIVCSW	ENTER TIME	RETURN TIME	TOTAL TIM
1328278	b'flushApp	0	0	0	0	0	0	1660464095	1660464095	0.02
1328278	b'flushApp	0	0	0	0	0	0	1660464095	1660464095	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464096	1660464096	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464097	1660464097	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464097	1660464097	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464097	1660464097	0.01
1328278	b'flushApp	0	0	0	0	3	0	1660464097	1660464121	23850.2
1328278	b'flushApp	0	0	0	0	0	0	1660464121	1660464121	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464121	1660464121	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464121	1660464121	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464121	1660464121	0.01
1328278	b'flushApp	0	0	0	0	0	0	1660464121	1660464121	0.01

图 16 probe 数据

可以看到，记录到一次 **flushAppendOnlyFile** 的异常执行时间（23850.2ms）。进一步查看其运行时间分布（utime：0，stime：0）的与上下文切换情况（NVCSW：0，NIVCSW：0），发现进程在此期间并不繁忙。进一步推测可能是由于某些原因导致了进程阻塞。根据记录的进出该函数的时间（ENTER Time：1660464097，Return Time：1660464121）查看系统调用的执行情况，如图 17 所示。

1660464096	b'redis-se	1328278	0	232	0
1660464097	b'redis-se	1328278	1	232	0
1660464097	b'redis-se	1328278	0	232	0
1660464097	b'redis-se	1328278	1	232	0
1660464097	b'redis-se	1328278	0	232	0
1660464097	b'redis-se	1328278	1	232	0
1660464097	b'redis-se	1328278	0	232	0
1660464097	b'redis-se	1328278	1	232	0
1660464097	b'redis-se	1328278	0	0	0
1660464097	b'redis-se	1328278	1	0	0
1660464097	b'redis-se	1328278	0	233	0
1660464097	b'redis-se	1328278	1	233	0
1660464097	b'redis-se	1328278	0	1	0
1660464097	b'redis-se	1328278	1	1	0
1660464097	b'redis-se	1328278	0	75	0
1660464121	b'redis-se	1328278	1	75	0
1660464121	b'redis-se	1328278	0	232	0
1660464121	b'redis-se	1328278	1	232	0

图 17 系统调用执行

可以发现，进程在 75 号系统调用停止了很长时间。通过查看 Linux 系统调用表，我们可以找到 1 号系统调用对应 **write**，75 号系统调用对应 **fdatasync**。图 18 是 **flushAppendOnlyFile** 的**函数实现**，可以发现，在该函数中确实调用了 **write** 来执行文件写入，而 write 在执行时因为 **fdatasync** 的执行而被阻塞。

```
111     /* We want to perform a single write. This should be guaranteed atomic
112      * at least if the filesystem we are writing is a real physical one.
113      * While this will save us against the server being killed I don't think
114      * there is much to do about the whole server stopping for power problems
115      * or alike */
116     nwritten = write(server.appendfd,server.aofbuf,sdslen(server.aofbuf));
117     if (nwritten != (signed)sdslen(server.aofbuf)) {
118         /* Ooops, we are in troubles. The best thing to do for now is
119          * aborting instead of giving the illusion that everything is
120          * working as expected. */
121         if (nwritten == -1) {
122             redisLog(REDIS_WARNING,"Exiting on error writing to the append-only file: %s",strerror(errno));
123         } else {
124             redisLog(REDIS_WARNING,"Exiting on short write while writing to the append-only file: %s",strerror(errno));
125         }
126         exit(1);
127     }
```

图 18 flushAppendOnlyFile 函数实现

进一步使用 **dstat** 查看磁盘读写情况，如图 19 所示。

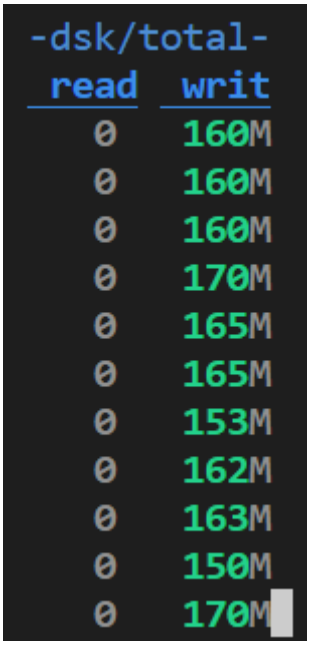


图 19 磁盘读写情况

可以发现此时磁盘确实在频繁写盘，导致 **fdatasync** 被阻塞，进而导致 redis 被阻塞，与预期的异常相符。

此外，我们还检验了模块在其他异常场景中的定位能力，结果如下表

序号	软件	异常类型	结果	异常报告链接
1	Apache	请求无响应	异常位于静态函数无法插桩分析	https://bz.apache.org/bugzilla/show_bug.cgi?id=65592
2	Redis	内存泄漏	成功分析定位	https://docs.google.com/document/d/1lG_xPiaWwpEHcz558Kort7KGV8-3QYctNTNwluaqtLw/pub
3	Redis	程序卡死	成功分析定位	https://docs.google.com/document/d/1OXMCzLfgQj91YDIQ71FxNCK53-I2voGfZenTc_1dhSA

				/pub
4	Redis	程序卡死	成功分析定位	https://github.com/redis/redis/issues/641

总结与展望

监控采集模块在初赛基本实现系统资源采集的基础上，在以下方面进一步进行了优化与开发：

1. 对多进程、多线程程序具有更好支持

   我们在初赛只能针对单进程程序进行监控采集的基础上，进一步对采集模块进行开发，使得现在的采集模块可以对多线程、多线程的程序进行数据采集，支持将被监控进程的子进程，子进程的子进程也加入监控采集的对象当中。

2. 优化资源开销情况

   相比初赛的多进程程序，在内存占用上取得了 5 倍的优化（如图 20 与图 21 所示），降低系统负载，提高系统的可用性。

```
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  1.0 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    976.8 total,     94.4 free,    346.9 used,    535.5 buff/cache
MiB Swap:   2400.0 total,   2350.0 free,     50.0 used.    461.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  25269 root      20   0  155840 132708  42384 S   0.3  13.3   0:03.64 python3
```

图 20 当前资源占用情况

```
top - 12:00:30 up 3 min,  1 user,  load average: 0.20, 0.15, 0.06
Tasks:   5 total,   0 running,   5 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    978.7 total,     81.2 free,    754.5 used,    143.1 buff/cache
MiB Swap:   2400.0 total,   2379.4 free,     20.6 used.     81.9 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1108 root      20   0  470140 101932   2832 S   0.0  10.2   0:00.01 python3
   1109 root      20   0  511772 177904  36328 S   0.0  17.8   0:01.10 python3
   1110 root      20   0  509000 171420  36244 S   0.0  17.1   0:01.04 python3
   1111 root      20   0  555820 221156  37428 S   0.0  22.1   0:02.23 python3
   1112 root      20   0  555888 219624  36188 S   0.0  21.9   0:02.23 python3
```

图 21 初赛阶段资源占用情况

3. 引入可选择性装载监控模块的框架

   针对内存占用较高以及程序拓展性较差的情况，我们对模块的框架进行了修改，由原来的固定模块、多进程程序修改为现在的可选择模块、单进程程序，从而在程序的灵活性、拓展性及未来进一步开发的潜力上取得较大进步，并解决了此前内存占用较高的问题。

4. 引入用户态函数追踪与分析模块

   为了更好的发挥 eBPF 的插桩能力，充分利用初赛已实现但没有得到充分得到利用的系统调用信息，并提供进一步定位异常所在的功能，我们开发了用户态函数追踪与分析模块，协助异常代码的定位。相比 gdb 等其他异常定位工具，我们的工具能够在程序不停、无需重新编译、无需修改代码的情况下进行分析，最大程度保留异常发生现场，加快异常定位速度。

未来我们还可以在以下方面进一步进行开发：

1. 针对 BCC 每次运行都会造成较大开销的问题，考虑未来向 CO-RE 的 BPF 程序开发方式迁移，从而将系统的性能开销进一步降低，并提高系统的可迁移能力；同时将程序开发语言由 Python 转向 C/C++也能够进一步提高程序性能，降低性能开销，特别是目前每个周期输出内容较多的模块同样会造成较大的性能开销；
2. 目前用户态函数插桩功能在插桩位置较多的情况下可能出现输出乱序的情况，未来可以进一步优化；
3. 进一步优化静态代码分析工具，通过更多更强的特征筛选出疑似异常所在函数，交由用户态追踪与分析模块进行分析。

异常检测算法模块

背景

操作系统的参数（CPU 占用、内存占用、网络流量等）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。异常的类型有点异常、上下文异常、集合异常等：

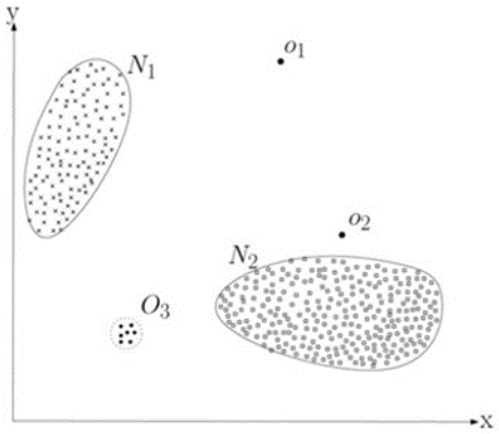


图 22 点异常

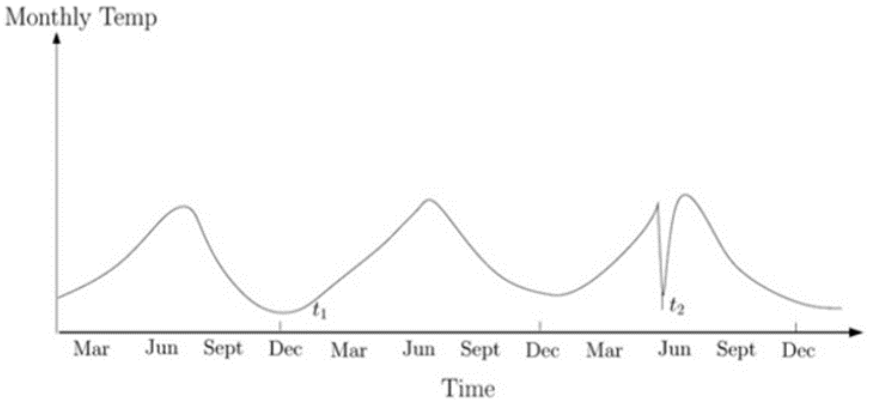


图 23 上下文异常

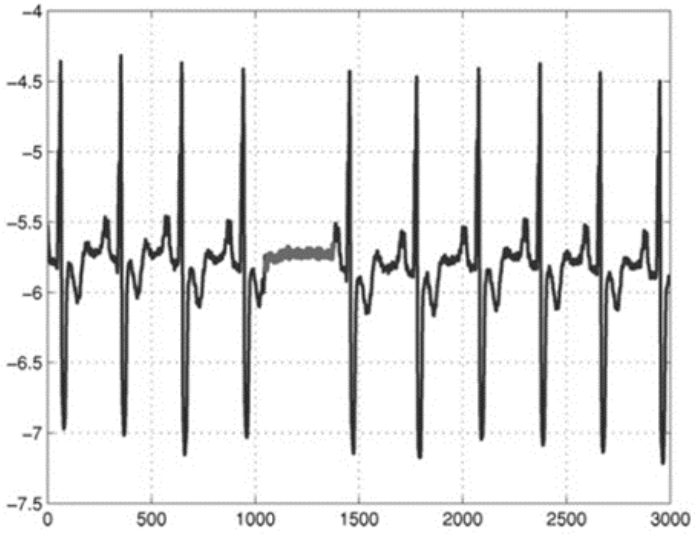


图 24 集合异常

异常检测的方法包括以下几种：

1. 直接检测，离群值检测
2. 间接检测，将上下文异常或集合异常转化为点异常，再进行检测
3. 时间跨度检测，学习一段时间的历史数据，预测下一阶段的数据，通过对比真实值和预测值的偏差来确定异常
4. 序列跨度检测，可以使用一个序列预测另一个序列，同样对比预测和实际来确定异常

时序异常检测方法有：

1. 基于统计的方法，3-sigma 等
2. 基于相似度量的方法，聚类等
3. 深度学习方法，RNN 和 Transformer

基于 LSTM 的异常检测方法

直接调用 KNN、LOF 等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。

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：

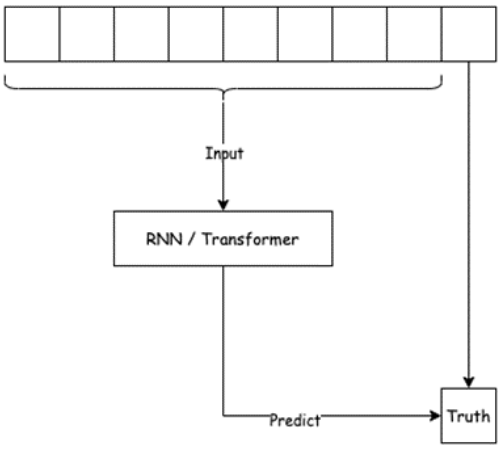


图 25 训练模型结构

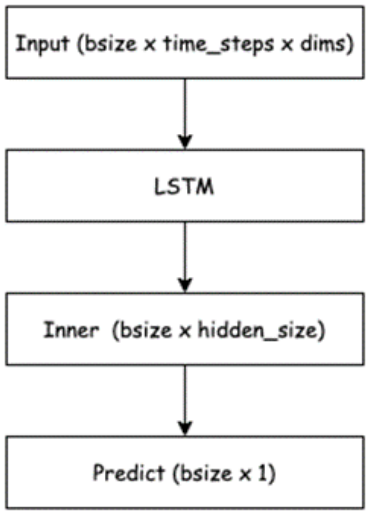


图 26 训练模型结构

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 27 OnnxRuntime 支持矩阵

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。

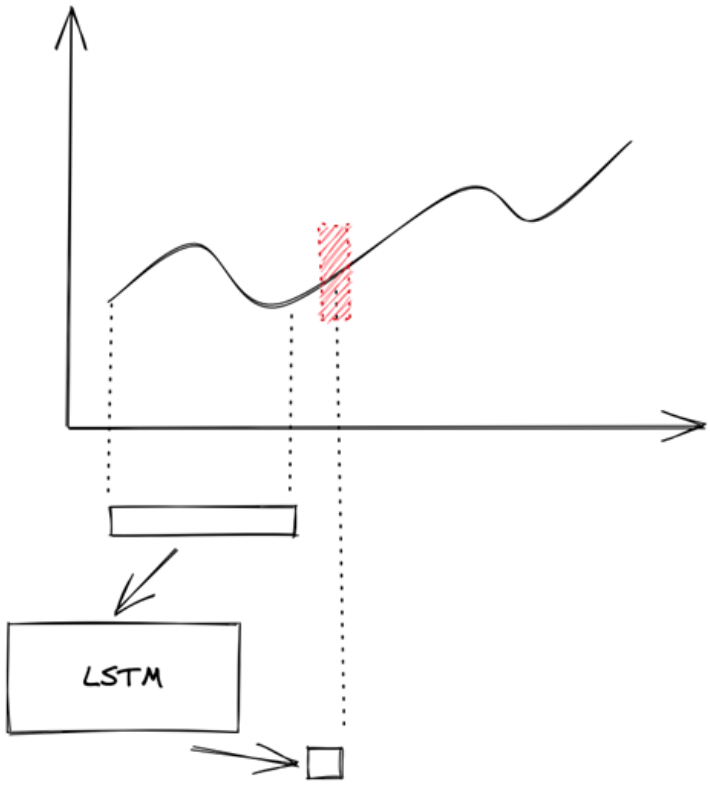


图 28 异常检测原理示意

初赛阶段我们实现了基于 LSTM 的模型结构，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。

效果演示

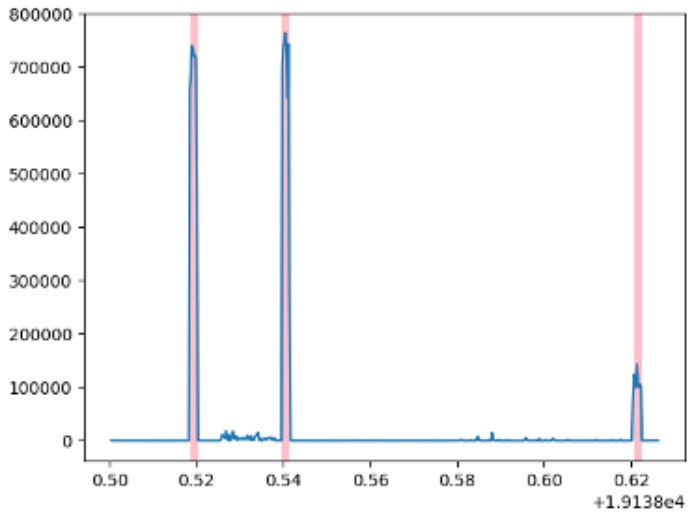
1. 手动注入异常检测

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
```

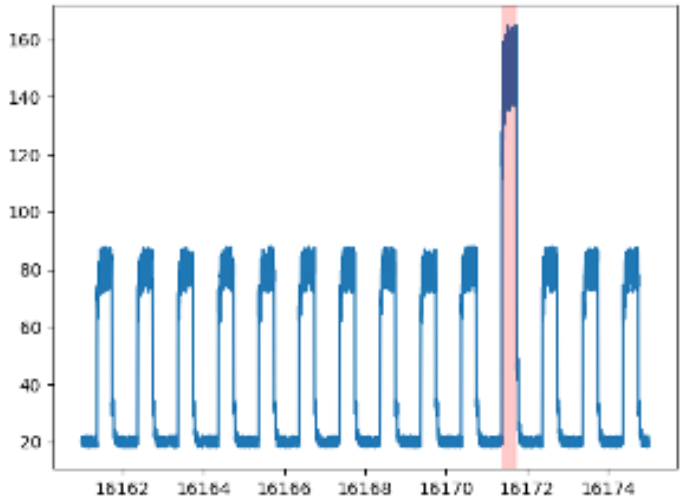
网络流量异常点偏差值



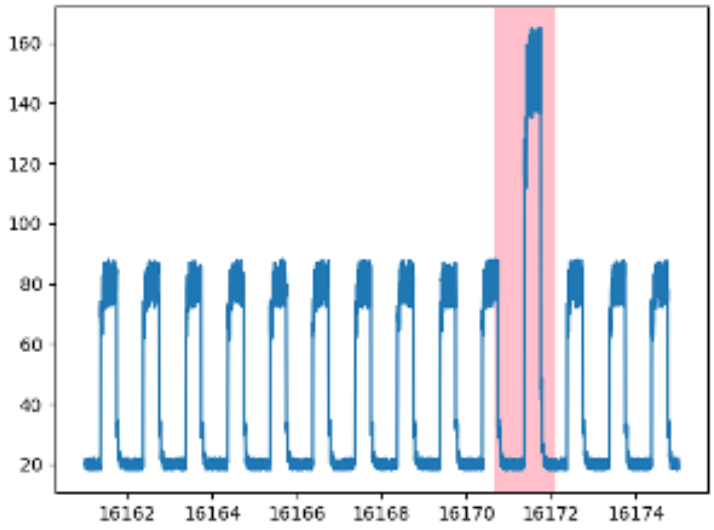
网络流量异常图表

2. NAB 数据集上的测试

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。



我们的结果



数据集标注

检测耗时

在上述人工注入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。

资源开销

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。

```
top - 12:16:14 up 18 min,  1 user,  load average: 0.17, 0.05, 0.01
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    978.7 total,    370.6 free,    248.1 used,    360.1 buff/cache
MiB Swap:   2400.0 total,   2380.8 free,     19.2 used.    588.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1171 root      20   0  482008 235648 113116 R  99.9  23.5   0:03.80 python3
```

算法部分资源占用示意

深度学习方法的不足

1. 训练耗时较长。对于不同的的系统，基于机器学习的方法通常需要针对每个系统收集足够长时间的数据进行训练，完成训练以后才能够进行有效的异常检测，无法做到一套参数适用于所有系统
2. 模型泛化能力差。大多数已有的模型都无法做到在可接受的参数量下对不同的工作负载都能达到较好的效果。
3. 无法应对系统负载的变更。在大型互联网公司，每日进行的软件更新有上千次，那么在系统中运行的软件对应资源消耗的“正常值”也是不断变化的。对于基于机器学习的方法，软件的变更通常意味着需要重新进行训练，这在软件高频变更的情况下是不现实的。

在初赛阶段，我们主要使用基于 LSTM 的异常检测方法，实践中还遇到了以下问题：

1. 为了达到合理的检测速度，LSTM 和线性层的参数量不能够选取得很大，导致模型泛化能力很差，无法应对与训练数据不同的波动。如果训练数据较为平滑，那么模型倾向于预测出平均的结果，对于波动较大的数据出现频繁的误报
2. LSTM 网络前没有添加 VAE 等结构，导致模型对周期性特征的识别能力差。如下图，如果输入 LSTM 的窗口不够大，那么在周期性负载的情况下，每次变动都会引起误报。

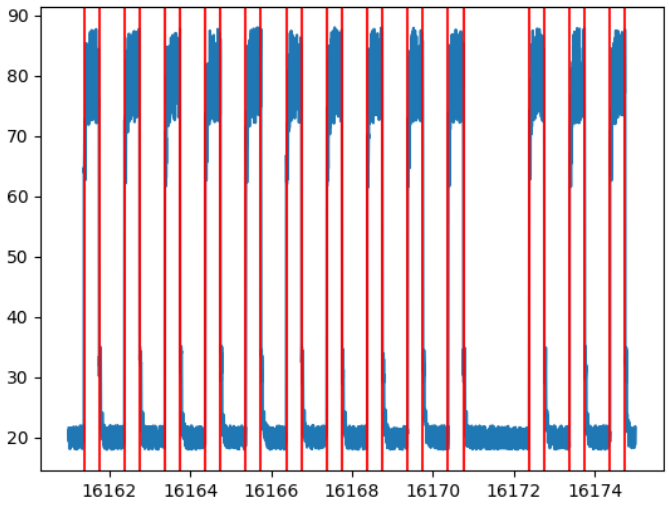


图 29 周期性波动效果图

基于压缩感知的异常检测方法

针对机器学习方法的上述不足，我们参考论文《Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems》，使用了基于压缩感知（Compress Sensing）的方法，在加入先验参数和抗干扰采样以后，这种做法具有以下的优点：

1. 无需进行训练。这使得只需经过少量数据点（<100）后就可以给出有效的检测。
2. 泛化能力好。只需要调整少量参数即可以满足大部分系统的要求。
3. 可解释性好。每一部分都可以根据实际情况进行更改。

算法的主要部分是压缩感知（Compress Sensing）。压缩感知是在信号处理方面常用的技术，将压缩感知运用在异常检测的主要原理是通常而言异常（如抖动，尖峰等）相对于正常的数值是一个高熵的信号，因此通过压缩感知重建的信号将不会包含此类信号，通过比较重建的数值和原本的数值，就可以检测出异常。

在实际应用中，有以下几个困难需要解决：

1. 压缩感知的重建是一个凸优化问题，在数据量巨大的情况下耗时较多。
2. 由于是从原始数据进行采样，因此采样可能包含异常信号，不利于检测算法的稳定性和健壮性。

为了解决数据量大时开销大的问题，我们使用了聚类的方法对输入数据进行降维，将形状相似的时序数据划分到相同的组，在采样时不重复采样。

同样的为了避免采样到异常的信号，使用基于聚类的方法对每个维度的输入数据进行评分，然后将评分转化为采样概率。这样离群值被采样的概率就显著下降了。

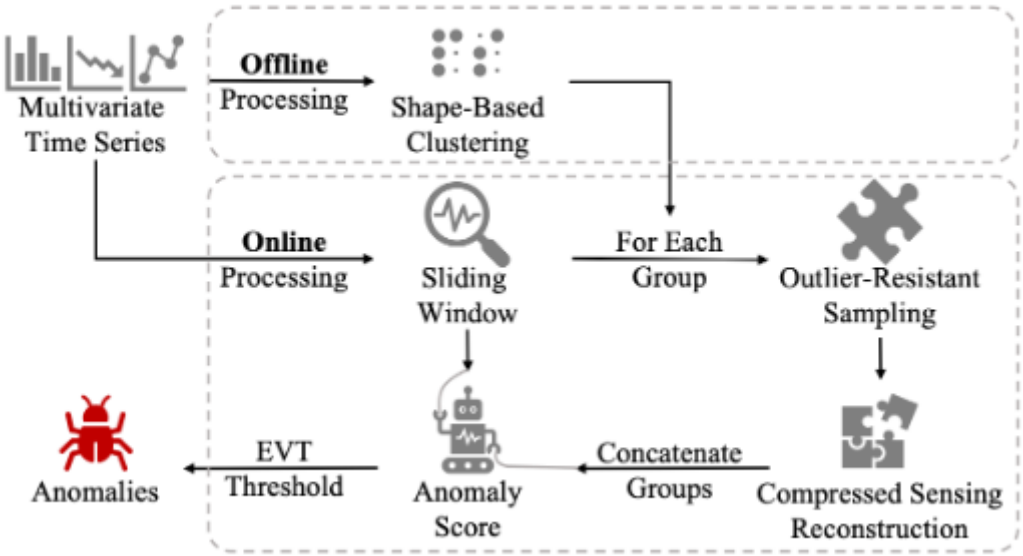


图 30 压缩感知异常检测方法流程示意图

算法详细介绍

输入的数据首先经过预处理：

1. 将输入的数据映射到[0,1)区间
2. 通过聚类算法对维度进行按周期聚类，也即每个周期（通常是 24h 等）都对维度进行一次聚类

接下来数据分窗口送入重建工作进程：

1. 首先通过 LESINN 对窗口内的数据进行离群值评分
2. 接下来根据评分进行带权的采样
3. 采得的数据调用 CVXPY 进行凸优化重建
4. 如果凸优化无解，增大采样的样本数量，重复 3
5. 得到重建的数据

得到重建的数据后，将重建的数据与原始数据进行比较，得到异常评分

1. 对每个时刻的重建数据与原始数据计算距离，可以是笛卡尔距离，也可以是其他距离
2. 通过距离得到异常评分，距离越远的评分越高

通过设定阈值，异常评分高于阈值的点认为是异常点

1. 阈值的选择采用 EVT 理论，选择数据的 rho + 3 sigma 作为阈值

在线检测

我们针对项目所需要的在线检测对原始算法进行了修改，使算法适合在线检测的特性：

1. 保存每次重建的数据，避免重复进行凸优化

2. 在线检测无法对比 ground truth 选出最佳阈值，而使用固定阈值会影响算法的泛化能力，因此在线检测时使用 EVT 进行阈值选择。

具体函数

1. utils.data_process.normalization(data: pandas.DataFrame) -> pandas.DataFrame

功能说明

对输入的数据进行正则化处理，映射到[0,1)区间

输入参数

data: pandas.DataFrame 输入数据

返回值

pandas.DataFrame 正则化后的数据

2. utils.metrics.**sliding_anomaly_predict(score: np.ndarray, window_size: int = 1440, stride: int = 10, ratio: float = 3):**

功能说明

滑动窗口的动态阈值异常预测

输入参数

score: np.ndarray 异常评分

window_size: int (default = 1440) 滑动窗口大小

stride: int (default = 10) 滑动窗口步长

ratio: float (default = 3) 标准差比例，例：threshold = mean + `ratio` * std

返回值

predict: np.ndarray 根据滑动窗口的动态阈值对异常做出 0、1 预测

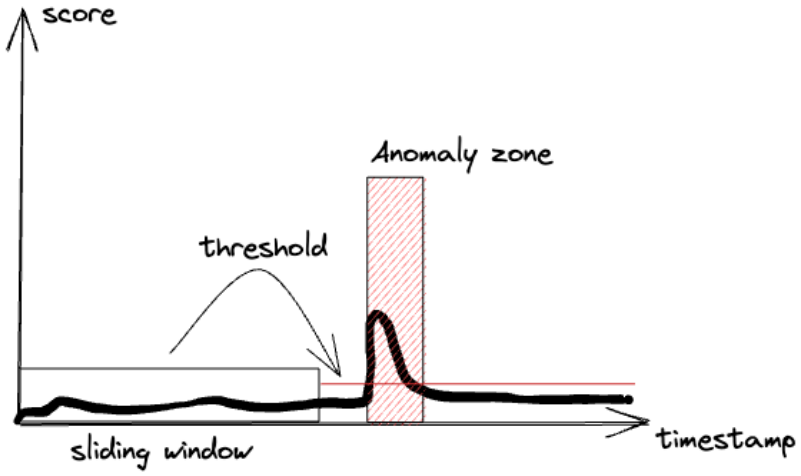


图 31 阈值选择和检测示意图

3. algorithm.cluster.**cluster(**

X: np.ndarray,

threshold: float = 0.01

) -> numpy.ndarray:

功能说明

BSD 聚类

输入参数

X: numpy.ndarray shape = (m, n)的数组

threshold: float (default: 0.01) 直接聚类的阈值

返回值

numpy.ndarray, 非均匀的三维数组，维度分别是[窗口，聚类数目，聚类中的维度]。其中的元素为 0 ~ (n-1)的整型数字，在同一个列表中即为一类

4. **algorithm.lesinn.online_lesinn**

incoming_data: np.array,

historical_data: np.array,

t: int = 50,

phi: int = 20,

random_state: int = None

) -> numpy.ndarray:

功能说明

在线离群值算法 lesinn

输入参数

incoming_data: np.array 需要计算离群值的向量

historical_data: np.array 参考的历史数据

t: int = 50　离群值参考的样本数量

phi: int = 20 每个样本的采样点数量

random_state: int 随机数种子

返回值

score: np.ndarray 与输入数据同维度的离群值评分

5. algorithm.sampleing.**localized_sample(**

x, m, score, scale=2, rho=None, sigma=1 / 12, random_state=None

):

功能说明

根据采样得分 score_func 函数对 x 进行随机局部化采样

输入参数

x:采样点的原数据矩阵, shape=(n,k), n 是数据个数, k 是 kpi 种数

m:采样数量

score: 每个采样点的采样得分, 得分越高, 越容易采样到该点的信息

rho: 采样单元中心点被采样概率, 若为 None, 取 1/(sqrt(2pi)*sigma)

scale: 将原采样点数扩充至原来的 scale 倍

sigma: 高斯随机采样标准差

random_state: 随机数种子

返回值

Tuple(numpy.ndarray, List): 采样矩阵 shape=(m,n)，每个采样单元的所在时间序列位置列表

6. algorithm.cvxpy.**reconstruct(n, d, index, value):**

功能说明

压缩感知采样重建算法

输入参数

n: 需重建数据的数据量

d: 需重建数据的维度

index: 采样点的时间维度坐标 属于[0, n-1]

value: 采样点的 KPI 值，shape=(m, d), m 为采样数据量

返回值

numpy.ndarray: 重建的 KPI 数据，shape=(n, d)

7. class WindowReconstructProcess()

窗口重建工作进程，用于支持离线多线程加速。离线评测时使用生产者-消费者模型进行并行化，每个重建工作进程负责接收某个窗口的数据并返回重建后的数据到队列中。

```
def window_sample_reconstruct(
        self,
        data: np.array,
        groups: list,
        score: np.array,
        random_state: int
    ):
```

功能说明

对某个窗口数据的重建

输入参数

data: 原始数据

groups: 分组

score: 这个窗口的每一个点的采样可信度

random_state: 随机种子

返回值

Tuple(numpy.ndarray, int): 重建数据, 重建尝试次数

算法准确率测试

测试所用数据集

1. JumpStarter

JumpStarter 是论文《Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems》提出的一种异常检测算法，我们使用这篇论文所提供的三个数据集进行对算法模块的效果检测，数据集的介绍如下图所示：

Dataset1 is collected from a large Internet company *A*.

Dataset2 and Dataset3 are collected from a top-tier global content platform *B* providing services for over 800 million daily active (over 1 billion cumulative) users across all of its content platforms.

Dataset	# Services	# Metrics	# Training Days	# Test Days	Anomaly Ratio
Dataset1	28	38	13	13	4.16
Dataset2	30	19	20	25	5.25
Dataset3	30	19	20	25	20.26

数据集参数

其中 Dataset2 和 Dataset3 含有在线的负载变更情况。

更多细节可以参考原论文。

2. AIOps 2018

AIOps(Artificial Intelligence for IT Operations)，国际智能运维挑战赛由清华大学联合中国计算机学会（CCF）发起，旨在在借助社区的力量，运用人工智能技术解决各类运维难题；AIOps 2018 的主题是 KPI 异常检测，赛方提供的数据集来自搜狗、腾讯游戏、eBay、百度、阿里巴巴等国内外一流互联网公司的真实运维环境，能够对算法在真实场景下的检测能力提供充分的测试。

因为我们的算法主要针对同间隔的监控指标设计，因此只针对数据集中同间隔的指标进行测试。数据集中共有 8 项指标是同间隔的，我们测试时分别使用全部 8 个维度和仅使用第一个维度进行了测试。使用的测试数据大小为 128559 个 timestamp，即大约 90 天的数据。

3. NAB

NAB(Numenta Anomaly Benchmark)是一个测试评估针对流数据的异常检测算法的开源工具，提供了多种数据集以供测试，分为真实数据和人工制造数据两种数据类型。我们选取了真实数据中 AmazonCloudWatch 服务收集到的 AWS server 的 CPU 占用率、网络流量、磁盘读写等数据，以及人工制造数据中的一部分进行测试。其中人工制造的数据会具有多种异常类型的典型特征。

评价方法

在测试中我们采用的评价指标是 F1，但是与 NAB 等数据集的评价指标类似，我们使用基于窗

口的准确率判定。使用这样的策略有一下几个方面的考虑：

1. 实际生产环境中，对于异常的检测并不需要按照数据点来进行检测，只需要在异常发生后的合理时间内发现异常就可以认为是成功检测出了异常
2. 对于不同的工作负载，如正常情况下的负载波动就比较大，由于我们的方法泛化能力较好，对于异常阈值的选择是基于统计的方法，此时对于异常波动的检测就会偏于迟钝，使用基于点的 F1 不能够正确反应算法的准确程度。
3. 使用基于点的 F1 对参数较为敏感。因为不同的参数对于某个特定的窗口内总共检出的异常点数量相差较大，因此 F1 变化较大，但对窗口内检出异常点数量做评价并无实际意义。

综合考虑后我们认为异常窗口第一个数据点后的 60 个数据点内如果算法检测出了异常，那么认为该窗口的异常被检测到了。

测试效果

测试所使用的参数绝大部分与提交项目中的配置一致，仅对于不同数据集的采样间隔调整窗口大小对应 1 小时的数据。

测试样例	Precision/%	Recall/%	F1/%
JumpStarter Dataset 1	100.00	67.37	80.50
JumpStarter Dataset 2	72.83	90.67	80.78
JumpStarter Dataset 3	92.46	100.00	96.08
AIOps 2018 - 1dim	94.58	59.30	72.90
AIOps 2018 - 8dim	71.62	73.38	72.49
NAB - RealAWSCloudWatch	100.00	69.07	81.70
NAB - ArtificialJumpUp	100.00	98.95	99.47

从上表可知算法模块对于多个数据集都有相对较高的准确率，可以认为我们的算法具有较强的泛化检测能力，能够应对多种真实场景的异常和人工生成的异常情形。

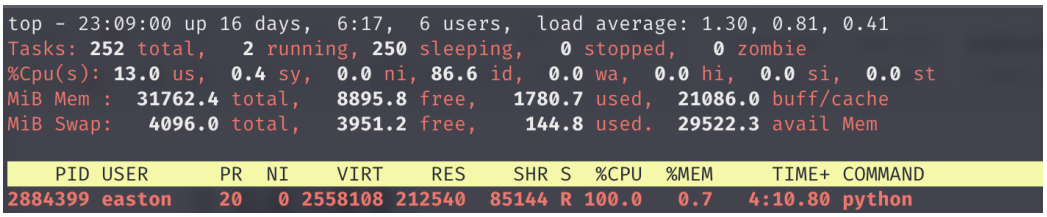
检测耗时

在离线测试中使用单核 CPU 对 JumpStarter Dataset 1 进行检测，平均速度为 23.36 sample/s，也即每秒可以处理 23 个左右的数据点。JumpStarter Dataset 1 共有 38 个数据维度，数据采样间隔为 1 分钟，因此可以认为检测性能完全可以满足在线检测的需要。

在离线测试 AIOps 2018 - 1dim 的数据共耗时 1 小时 49 分钟，平均速度为 19.65 sample/s，与 8dim 的测试速度相近。可以看到算法的检测速度在常见在线服务的监控指标数范围内没有明显变化。

资源开销

离线测试检测算法时资源开销如图



算法资源开销

可以看到相比于初赛使用的深度学习方法，压缩感知的方法无需存储大量参数，消耗内存很少，约为 200M 左右。

检测的及时性

在我们的项目中，算法使用的检测窗口步进为 10，对于采集间隔为 1 分钟的数据，最大的异常检测延迟即为 10 分钟。项目中使用了配置文件作为参数来源，如果允许更大的检测开销，可以减小窗口步长，或者使用更细粒度的数据收集，无需修改算法即可以支持更及时的检测。

与深度学习方法的对比

优点

1. 无需进行训练。只需经过少量数据点（<100）后就可以给出有效的检测。即使考虑到 EVT 在线阈值选取的有效性，基本上可以在部署的数个小时内进入有效的状态。
2. 泛化能力好，只需要调整少量参数即可以满足大部分系统的要求。这些参数与系统监控的参数和系统的特性有关，如监控数据的收集间隔，系统的行为周期等，这些参数一经部署就不会改变，在处理负载变更等与监控和异常检测无关的操作时无需进行修改。
3. 能应对在线的软件变更。在线软件负载的变更通常会引起深度学习方法的大量误报，JumpStarter 文章中所使用的数据集中含有此类软件负载变更的情况，根据我们的测试数据可以看到算法对于软件负载变更有良好的应对效果。
4. 可解释性好。每一部分都可以根据实际情况进行更改思考与改进。

缺点

1. 对于大规模数据扩展性不好。相比于深度学习有非常成熟的加速方法，基于压缩感知的方法依赖凸优化库，然而凸优化的并行性并不好，无法使用 GPU 等设备进行加速。在我们测

试数据的范围内（<50 个数据维度）算法在单核 CPU 上的速度完全可以满足在线监控的需要，但是如果继续增大数据量，凸优化将可能成为耗时瓶颈。

思考与改进

采样方式

出于简单起见，采样算法使用纯 Python 实现，速度并不快。但事实上该部分算法并行度很好，如果改用 numpy 实现，将会有不小的速度提升。

异常检测方式

得到异常评分后，我们的做法直接使用统计阈值的方法判定异常，没有使用时序信息，如果采用 SPOT 等时序单变量异常检测方法，可能有更好的效果。

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] “HOWTO:BCCto libbpf conversion BPF.”

[5] LeSiNN: Detecting Anomalies by Identifying Least Similar Nearest Neighbours

[6] NAB Dataset

[7] “WhyWe Switched from BCC to libbpf for Linux BPF Performance Analysis”

[8] Huang P, Guo C, Lorch J R, et al. Capturing and enhancing in situ system observability for failure detection[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 1-16.

[9] Lou C, Huang P, Smith S. Understanding, detecting and localizing partial failures in large system software[C]//17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). 2020: 559-574.

[10] Anomaly Detection Using Bidirectional LSTM