中国研究生操作系统开源创新大赛

项目功能说明书

《基于 eBPF 的容器异常行为检测系统》

学　　校：	西安电子科技大学
参 赛 队 伍：	Mobisys容器云团队
队 伍 成 员：	李华东、田震雄、马熙瑞
指 导 教 师：	杜军朝
完 成 日 期：	2024年8月

目 录

第 1 章 绪 论

1.1 目的与意义

本系统主要应用于容器化部署环境，旨在监控和检测容器运行中的异常行为。特别是在云计算、大数据处理和微服务架构等需要高效、灵活的资源管理和快速部署的场景中，该系统能够提供强大的运行时安全保障。

1.1.1 目的

1. 提高应用容器化部署安全性：应用容器化部署虽然具有可扩展、轻量级等优势，但也因为其共享操作系统内核的运行原理，存在隔离性差的缺点。设计该系统的目的是为了弥补容器隔离性差的不足，通过实时监控容器内部工作进程的行为，及时发现和阻止潜在的安全威胁。

2. 事件响应和故障排除：通过实时检测和记录容器内部进程的异常行为，可以帮助运维人员提前发现安全问题、迅速定位问题根源，缩短故障排除时间，提高系统的稳定性和可靠性。

3. 合规性和审计：在一些严格要求合规性的行业中，实时监控和记录容器行为能够满足审计需求，确保系统操作符合规。例如：

a) 金融行业受制于大量的监管法规，如《中华人民共和国商业银行法》、《金融机构信息系统等级保护规定》等。这些法规要求金融机构对所有交易和数据处理过程进行详细的记录和监控。容器异常行为检测系统可以实时监控金融交易应用中的所有活动，确保每一笔交易和数据访问都被记录和审计。这有助于金融机构及时发现和防范欺诈行为，确保数据完整性和客户信息的安全。

b) 政府军事机构需要遵守如《中华人民共和国国家安全法》、《中华人民共和国保密法》等法规，这些法规对信息系统的安全性和数据保护提出了高标准。容器异常行为检测系统可以监控和记录涉及国家安全和机密信息的所有容器活动。这样一来，可以确保任何未经授权的访问或异常行为都能被及时发现和处理，从而保护国家安全信息的完整性。

c) 电信行业需要遵守如《中华人民共和国电信条例》、《中华人民共和国数据安全法》等法规，这些法规要求对用户数据进行严格的保护和管理。该系统可以监控

和记录所有处理用户数据的容器活动，确保数据处理过程符合相关法规的要求。这不仅帮助电信公司防范数据泄露风险，还能在审计过程中提供详细的操作记录。

1.1.2 意义

1. 防范重大安全事件：例如在 2023 年 11 月，滴滴公司因升级配置错误导致宕机 36 小时，损失千万级订单和超过 4 亿成交额。这一事件强调了容器的隔离性差、系统稳定性和安全性的重要性。业界迫切的需要一个容器异常检测系统，防止类似事件的发生。

2. 应对新型安全漏洞：例如 CVE-2024-21626[1]，该漏洞允许容器通过符号链接访问宿主机文件系统目录。我们的系统可以及时检测到此类异常访问行为，避免对宿主机造成潜在的安全威胁。

3. 技术创新：eBPF[2]技术的应用，使得系统能够在不影响性能的前提下，进行高效的内核级监控。这不仅是对传统安全监控方法的一次创新，更是在容器化部署中提升安全防护能力的一次重要实践。

基于 eBPF 的容器异常行为检测系统使用操作系统内核级 eBPF 技术，提供了一种高效、安全的容器监控解决方案。该系统在保障云计算和大数据处理等场景环境安全性方面具有重要意义，同时为企业运维和合规审计提供了有力支持。

1.2 项目背景

在容器监控和异常检测领域，Falco[3]和 cAdvisor[4]是两款常见的软件工具。以下是对这两款软件的优缺点分析：

工具/系统	优点	缺点
Falco	- 实时监控和告警：Falco能够实时监控系统行为并在检测到异常时立即发出告警。 - 社区支持和文档：作为一个开源项目，Falco拥有广泛的社区支持和详尽的文档资源，便于用户查找和解决问题。 - Kubernetes集成：Falco可以与Kubernetes无缝集成，提供容器化环境下的安全监控。	- 基于规则，不够灵活：Falco主要依赖预定义的规则进行监控和检测，这种方式虽然高效但灵活性不足，无法应对复杂多变的攻击模式和新型威胁。
cAdvisor	- 资源使用监控：cAdvisor专注于监控容器的资源使用情况（如CPU、内存、网络等），提供详细的资源使用统计数据。 - 轻量级：cAdvisor设计轻量，易于部署和使用，适合在各种规模的集群中使用。 - 数据可视化：cAdvisor提供内置的可视化界面，方便用户实时查看容器资源使用情况。	- 无法获取容器内进程级别的信息：cAdvisor只能监控容器的整体资源使用情况，无法深入获取容器内的进程级别信息，限制了其在细粒度监控和异常检测中的应用。 - 缺乏安全监控功能：cAdvisor主要用于资源监控，缺乏专门的安全监控和异常检测功能，无法及时发现和响应安全威胁。

表 1 - Falco, cAdvisor 优缺点分析表

我们的参赛作品基于 eBPF 的容器异常行为检测系统在设计上具有以下新颖之处：

1. 内核级监控：借助 eBPF 技术，我们的系统能够在内核级别进行高效监控和数据收集，确保对容器内进程级别的行为进行详细分析，弥补了 cAdvisor 无法获取容器内进程信息的缺陷。

2. 动态灵活的检测机制：相比于 Falco 依赖静态规则的检测方法，我们的系统采用了动态灵活的检测机制，能够根据实时监控数据进行智能分析，及时发现和响应新型威胁，提升了检测的准确性和灵活性。

3. 综合监控能力：我们的系统不仅能够基于 cAdvisor 监控容器的资源使用情况，还能深入检测容器内部的异常行为，提供全面的安全保障。这种综合监控能力涵盖了 cAdvisor 的资源监控功能，也涵盖了 Falco 的安全监控功能，并在此基础上进行了改进和创新。

4. 低性能开销：通过优化 eBPF 程序，我们的系统在实现高效监控的同时，尽量减少对系统性能的影响，确保在高负载环境下依然能够稳定运行。

综合以上分析和描述，我们的基于 eBPF 的容器异常行为检测系统在灵活性、监控深度和性能优化等方面进行了创新设计，弥补了现有工具的不足，提供了一个高效、全面的容器安全监控解决方案。

第 2 章 技术理论

2.1 设计思想

2.1.1 系统架构设计

系统由两个主要模块组成：eBPF 采集模块和算法处理模块。两个模块之间通过 Kafka 消息队列进行数据交互，通过 HTTP 协议进行控制交互。将数据处理模块在另外的机器上部署，以尽可能降低系统对机器性能的影响。系统架构如下图所示：

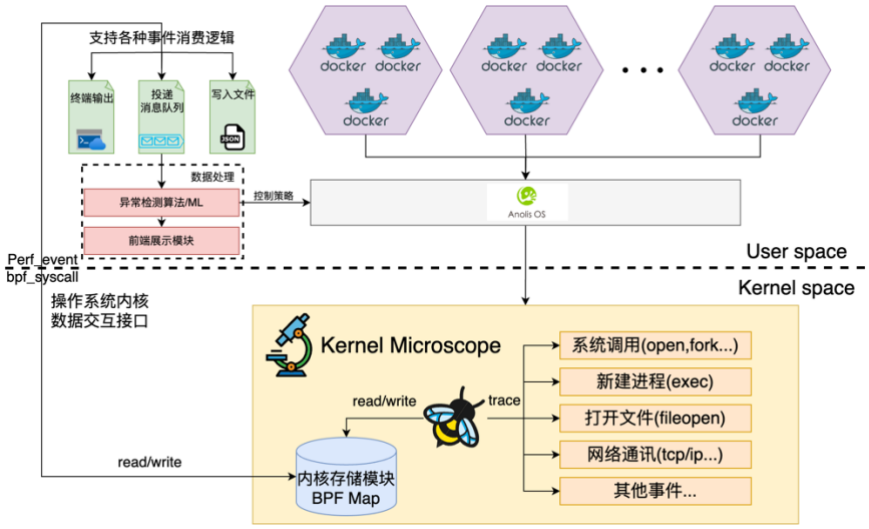

图 2 -系统架构图

1. eBPF 采集模块：部署在龙蜥操作系统中，负责从容器中收集各种行为特征数据，包括系统调用频率、系统调用序列、文件访问、网络通信、IO 吞吐、内存利用率、CPU 利用率等。该模块利用 eBPF 技术，实现高效的内核级数据采集，确保数据的实时性和准确性。

2. 算法处理模块：部署在任意 Linux 服务器中，接收 eBPF 采集模块发送的数据，通过 Kafka 消息队列进行数据交互，利用机器学习和深度学习算法对数据进行处理和分析，自动识别异常行为。该模块还通过 HTTP 协议与采集模块进行控制交互，以实现对数据采集过程的动态调整和优化。

2.1.2 研究与开发方法

我们对市面上多款 eBPF 开发工具进行了详细对比，最终选择了 Cilium/ebpf[5]库 TODO 引用。以下是对比结果：

框架	优点	缺点
iovisor/bcc	- 使用Python作为eBPF前端语言，功能丰富，简单易上手。 - GitHub 20K Stars，优秀的社区生态。 - 对其熟悉，**队伍中有BCC库的贡献者。**	- 使用BCC实现的eBPF程序在运行时需要庞大的Python运行时依赖（包括Clang、LLVM...），不利于部署。 - 每次运行时需要使用Clang重新编译ebpf内核态程序，性能开销大。 - Python会将程序错误推迟到运行时再暴露，不利于系统级工具的开发。
libbpf/libbpf	- 使用C/C++/Rust作为eBPF前端语言，高性能。 - libbpf是一个C语言库，伴随内核版本分发，由内核开发人员进行维护，可靠。 - 支持CO-RE、BTF等高级特性。	- C/C++生态较差，无好用的第三方库（如docker库、Kafka库...）。 - Rust开发难度高。
cilium/ebpf	- 纯Go实现的ebpf框架，继承了Golang的优秀特性，编译产物无外部依赖。 - 社区活跃。 - 支持CO-RE、BTF等高级特性。 - docker也是由Golang语言实现的。在"基于eBPF的容器异常行为检测"课题中，使用Golang语言能够高效的与docker后台守护进程dockerd进行通讯，获取容器信息。	- 无。

表 3 - eBPF 开发工具对比表

综上所述，我们最终选择了 Cilium/ebpf 库作为开发框架，并基于 Cilium/ebpf 库实现了一套高度抽象的 eBPF 开发框架，能够较为方便的添加功能。

2.1.3 开发思想与实现

本项目的开发思想围绕着高度抽象封装、性能优化以及数据处理与展示三个核心理念展开。通过这些设计理念，我们实现了一个高效、可靠的容器异常行为检测系统，该系统在功能和性能上都达到了预期目标，并在实际应用中表现出色。要点如下：

1. 高度抽象封装：我们对 eBPF 开发框架进行了高度抽象封装，使得为 eBPF 采集模块添加新功能变得更加简便。具体来说，开发者只需实现 eBPF 内核态程序，并在用户态程序中实现相应的接口即可。这种设计大大简化了开发流程，提高了开发效率，减少了开发过程中可能出现的错误和复杂性。此外，抽象封装还使得系统更具灵活性和可扩展性，能够快速适应不断变化的需求和技术进步。

2. 性能优化：在设计过程中，我们特别注重性能优化，以确保采集框架对系统性能影响最小。通过精细化的资源管理和优化算法，我们将 CPU 占用控制在 5%以内，保证了被监控容器的正常流畅运行。这种低开销的设计不仅提高了系统的整体性能，还确保了在高负载环境下的稳定性，使得系统能够在实际应用中长时间稳定运行。

3. 数据处理与展示：算法处理模块利用先进的机器学习和深度学习技术，实现了高准确率的异常行为检测。我们选择合适的指标作为输入，能够有效检测出各种不同类型的异常行为。这些指标包括系统调用频率、系统调用序列、文件访问、网络通信、IO 吞吐、内存利用率、CPU 利用率等。为了方便用户理解和分析检测结果，我们还借助 Grafana 等 UI 系统，实现了各种数据的汇总展示。通过清晰的可视化界面，用户可以直观地查看监控数据和异常检测结果，了解系统运行状态和潜在风险。

2.2 技术路线

本系统通过 eBPF 技术在操作系统内核层面对容器行为特征进行细粒度采集，并采用智能算法模块与检测任务相结合的方式，进行高效、精准的异常检测。数据采集与检测分离，确保低性能损耗，并使用 Grafana 搭建监控大屏，实现直观、可配置的 Web 界面展示。本系统主要使用了如下技术：

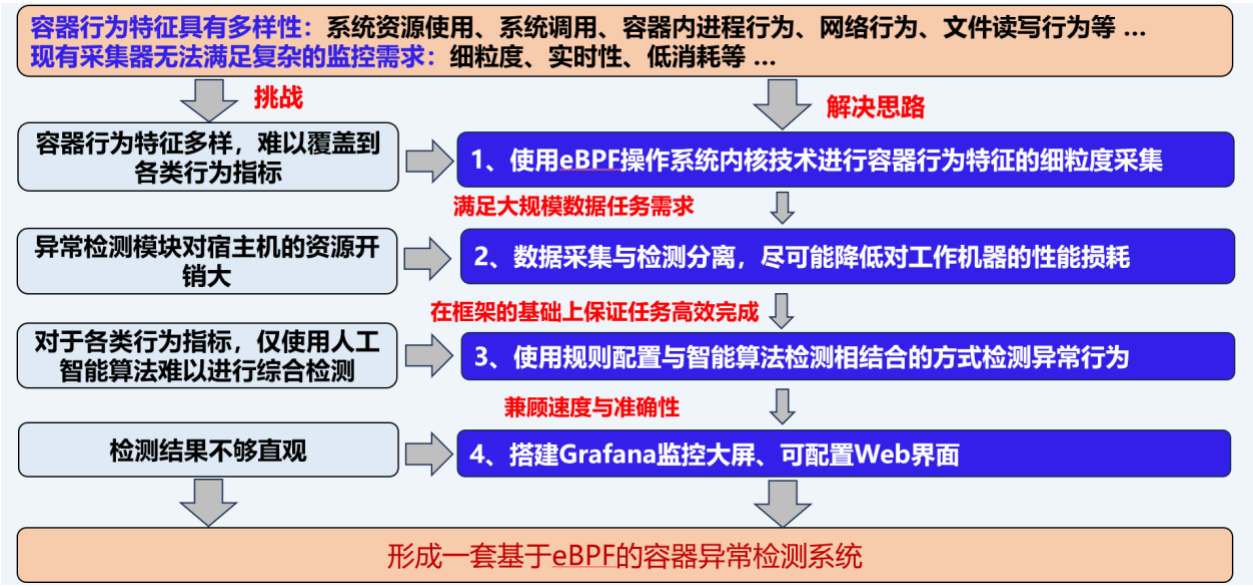

图 4 -技术路线图

1. eBPF（Extended Berkeley Packet Filter）是一种在 Linux 内核中运行的强大工具，能够以低开销的方式动态追踪和分析系统行为。通过 eBPF，开发者可以编写小型程序，直接运行在内核态，以捕获系统调用、监控网络流量、检查文件访问等操作。该技术的高性能和灵活性使其成为实现实时监控和异常检测的理想选择。

2. Golang（Go 语言）是一种静态类型、编译型的编程语言，因其简洁的语法、高效的并发处理能力和强大的标准库，广泛应用于云计算、微服务和容器化应用中。在本项目中，Golang 不仅用于实现用户态程序与 eBPF 内核态程序的交互，还负责处理数据采集、算法处理等后台服务的逻辑。Golang 的高性能和易用性，使得系统在保持高效的同时，具备良好的可维护性和可扩展性。

3. Kafka 是一种分布式流处理平台，擅长处理大规模的实时数据流。它通过发布-订阅模式，提供了高吞吐量、低延迟的数据传输能力。在本系统中，Kafka 用作 eBPF 采集模块和算法处理模块之间的消息队列，确保数据高效、可靠地传递。Kafka 的使用，不仅简化了模块间的解耦，还提高了系统的整体性能和可扩展性。

4. K-medoids 算法可以使用任意类型的距离度量，包括非数值数据的度量方式，系统调用数据可能包含时间戳、字符串或分类特征，K-medoids 可以灵活选择合适的距离度量方法进行聚类和异常检测。

通过这些技术的结合，本系统能够高效地实现容器行为的实时监控、异常检测和数据可视化，满足了各类高安全性和高性能需求的应用场景。

2.3 代码原创说明

本软件 eBPF 数据采集模块参考了 Linux[6]内核源码，eBPF 内核态程序调用了 libbpf 开源库，eBPF 用户态程序调用了 Cilium/ebpf 开源库；本软件算法模块采用的是 K-means 聚类算法的思想，代码均为原创。本项目使用了 kafka,vue.js,Prometheus,Grafana 等开源项目。

第 3 章 软件介绍

3.1 软件功能介绍

本系统由两部分构成：

1. 部署在龙蜥操作系统中的 eBPF 采集模块，采集操作系统中容器的运行时行为信息。工作容器运行在这个系统中。
2. 部署在任意 Linux 服务器中的算法处理模块，负责除 eBPF 以外的所有处理逻辑。

两个模块之间通过 kafka 消息队列进行数据交互，通过 http 协议进行控制交互。

系统功能如下：

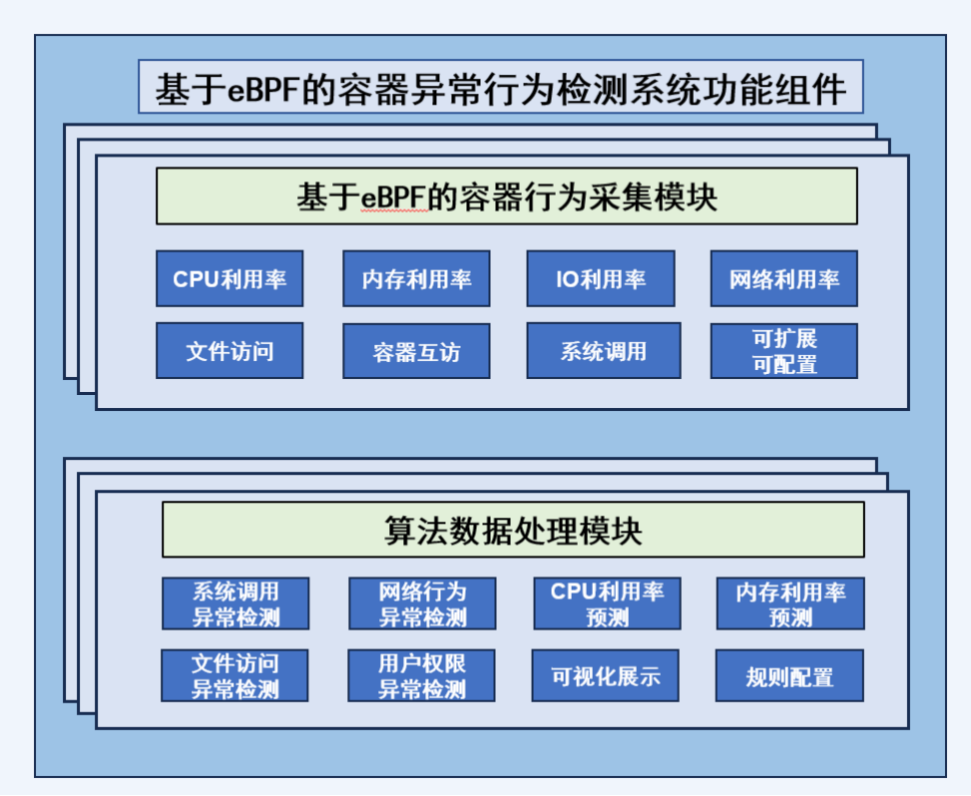

图 5 -系统功能图

3.1.1 eBPF 采集模块功能介绍

eBPF 采集模块主要实现了如下功能：

1. 采集系统资源使用情况，使用传统的采集工具。赛题导师在答疑群中说明了可以使用传统工具。

图 6 -答疑群聊天记录图

2. 采集系统调用事件，使用 btf_raw_tracepoint eBPF 程序类型。btf_raw_tracepoint 与 raw_tracepoint 类似，但是可以使用 BTF 信息来解析 tracepoint 的参数。基于此，我们使用了 eBPF tailcall 机制，对于特殊的系统调用进行特殊处理，核心代码如下：

```
struct {
  __uint(type, BPF_MAP_TYPE_PROG_ARRAY);
  __type(key, uint32_t);
  __type(value, uint32_t);
  __uint(max_entries, 1024);
  __array(values, int());
} syscall_enter_tail_table SEC(".maps") = {
    .values =
        {
            [59] = &sys_enter_execve, // sys_enter_execve是函数指针，用于处理特殊系统调用
            [83] = &sys_enter_mkdir,
            ...
        },
};
SEC("tp_btf/sys_enter")
int BPF_PROG(sys_enter, struct pt_regs *regs, long syscall_id) {
  struct task_struct *curr_task = (struct task_struct *)bpf_get_current_task();
  if (get_task_level_core(curr_task) == 0) {    // 非容器进程
    // level 0 means the task is in the root pid namespace
    return 0;
  }
  struct syscallcntkey key = {
      .syscall_id = syscall_id,
      .pid = bpf_get_current_pid_tgid() >> 32,
  };
  get_cid_core((struct task_struct *)bpf_get_current_task(), &key.cid); // 获取cid
  u64 *sys_cnt = bpf_map_lookup_elem(&syscall_cnt, &key); // 获取计数器指针
  sys_cnt++;
  bpf_tail_call(ctx, &syscall_enter_tail_table, syscall_id);// 尾调用
  return 0;
}
```

3. 采集文件访问事件，使用内核源码中/fs/namei.c/do_filp_open()内核函数作为 Hook 点，理由如下：函数稳定，自 2.6.12 版本之后（12 年）再无修改；无内联优化，函数符号

稳定；参数含义明确，其返回值是 file 类型的指针，指向本次打开的文件控制块。核心代码如下：

```c
SEC("kprobe/__tcp_transmit_skb")
int BPF_KPROBE(kprobe_tcp_transmit_skb) {
  if (get_task_level_core((struct task_struct *)bpf_get_current_task()) == 0) {
    return 0;
  }
  ...
  struct cnetwork_event *event =
      bpf_ringbuf_reserve(&containernet_rb, sizeof(struct cnetwork_event), 0);
  u16 family = BPF_CORE_READ(sk, __sk_common.skc_family);
  if (family == AF_INET) { // 一定成立
    event->daddr = BPF_CORE_READ(sk, __sk_common.skc_daddr);
    event->saddr = BPF_CORE_READ(sk, __sk_common.skc_rcv_saddr);
  }
  // 网络事件过滤
  u64 tmpa = 1ll * event->daddr << 32 | event->saddr;
  u64 tmpb = 1ll * event->saddr << 32 | event->daddr;
  if ((bpf_map_lookup_elem(&cnetwork_banned, &tmpa) == NULL) &&
      (bpf_map_lookup_elem(&cnetwork_banned, &tmpb) == NULL)) { // 不在规则集中
    bpf_ringbuf_discard(event, 0);
    return 0;
  }
  ...
  event->flag = 0;  // send
  return 0;
}
```

4. 采集网络互访事件，默认容器间通讯是使用基于 IPv4 的 tcp 协议。使用内核源码中/net/ipv4/tcp_output.c/__tcp_transmit_skb()&&net/ipv4/tcp_ipv4.c/tcp_v4_do_rcv()内核函数作为 tcp 收包、发包的 hook 点，理由如下：函数稳定，自 2.6.12 版本之后（12 年）再无修改；无内联优化，函数符号稳定，参数含义明确；相较于网络通讯本身的开销，tcp 协议栈中 eBPF 程序的开销可以忽略不计。 核心代码如下：

```c
SEC("kretprobe/do_filp_open")
int BPF_KRETPROBE(kretprobe_do_filp_open, struct file *filp) {
  ...
  struct file *fi = filp;
  u8 *fsnamep = (u8 *)BPF_CORE_READ(fi, f_inode, i_sb, s_type, name); // 获取文件系统名称
  bpf_core_read_str(&event->fsname, FSNAME_LEN, fsnamep);

  struct dentry *cur_dentry = (struct dentry *)BPF_CORE_READ(fi, f_path.dentry);
  int offset = 0;
  for (int i = 0; i < FILE_MAXDEPTH; i++) {// 按层次拼接文件路径
    if (cur_dentry == NULL) break;
    get_dentry_name_core(cur_dentry, event->filename[i]);
    if (event->filename[i][0] == 0) break;

    cur_dentry = (struct dentry *)BPF_CORE_READ(cur_dentry, d_parent);
  }
  ...
  return 0;
}
```

3.1.2 算法处理模块功能介绍

算法处理模块主要实现以下功能：

1. 基于 K-Medoids 的系统调用异常检测算法

针对监测到的数据参数复杂、种类繁多的问题，我们采用 K-Medoids 算法来对 Syscall id 进行异常检测。因为 K-medoids 算法可以使用任意类型的距离度量，包括非数值数据的度量方式，系统调用数据可能包含时间戳、字符串或分类特征，K-medoids 可以灵活选择合适的距离度量方法进行聚类和异常检测。

而我们选择 k-medoids 而不是 k-means 的原因：K-medoids 选择的是数据集中的实际点作为中心点（medoid），而不是计算均值。这意味着即使存在异常值，它们也不会极端地影响 medoid 的位置。且 k-means 对初始簇中心的选择非常敏感，不同的初始选择可能导致不同的结果，可能陷入局部最优解；而 k-medoids 对初始 medoid 的选择相对不太敏感，算法的稳定性较好，在多次运行异常检测任务时，K-medoids 可以提供更一致的结果，有助于建立稳定的异常检测系统。

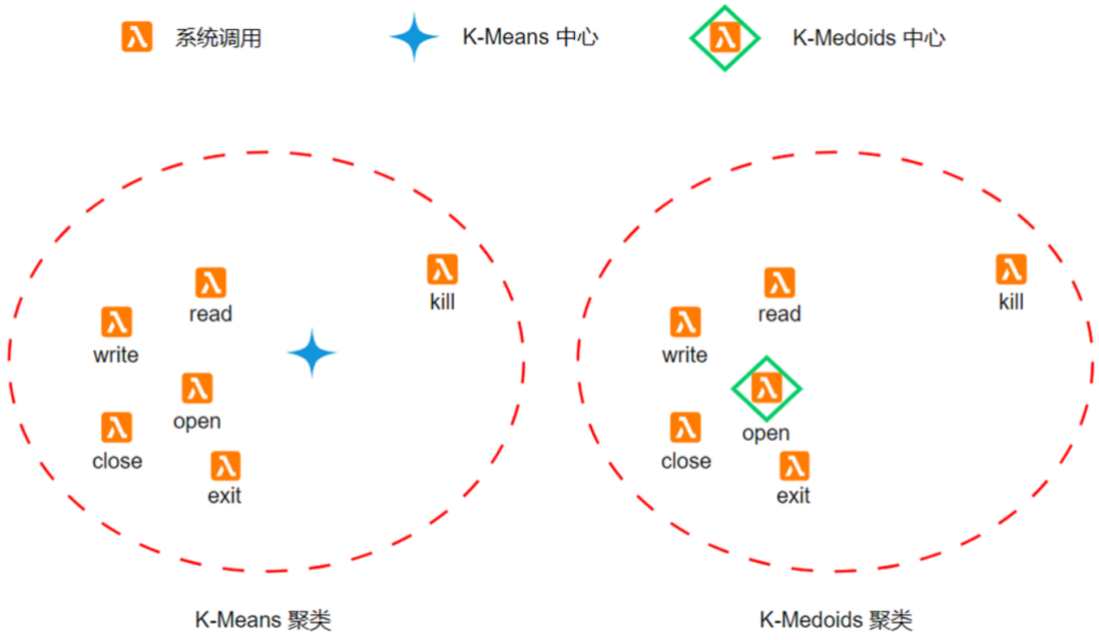

图 7 – k-means, K-medoids 算法对比图

我们的解决方案是每个系统调用均有独属 `id`，使用算法对检测到的系统调用 `id` 进行聚类，得到每个数据点所属的聚类以及聚类的中心点，设置阈值判断距离异常大的数据

点，随后将筛选的异常数据与黑名单中的系统调用 id 进行比对识别，输出异常的系统调用操作，提示容器异常

算法流程如下图所示：

改进的 K-medoids 算法

输入：函数样本集$\{P_{i,1}, P_{i,2}, \dots, P_{i,j}\}$

输出：簇的中心点$M_i = \{M_{i,1}, M_{i,2}, \dots, M_{i,K}\}$和簇集合$S_i$

1. **for** k = 1 to j **do:**
2. 随机选择 k 个数据点作为初始簇中心$M_i = \{M_{i,1}, M_{i,2}, \dots, M_{i,k}\}$
3. **repeat**
4. **for** m = 1 to j **do**:
5. **for** j = 1 to k **do:**
6. 计算$d(P_{i,m}, M_{i,j})$
7. **end for**
8. 将$P_{i,m}$分配到距离最近的簇中心点所在的簇
9. **end for**
10. **for** j = 1 to k **do:**
11. 选出新的簇中心$M'_{i,j}$
12. **if** $M'_{i,j} \neq M_{i,j}$**then:**
13. $M_{i,j} = M'_{i,j}$ //更新簇中心点
14. **else:**
15. $M_{i,j} = M_{i,j}$ //保持当前簇中心点不变
16. **end if**
17. **end for**
18. **until** 当前簇中心点均未更新
19. 根据欧氏距离法选择最佳k值
20. 输出簇的中心点$M_i = \{M_{i,1}, M_{i,2}, \dots, M_{i,K}\}$和簇集合$S_i$

第 4 章 软件测试

4.1 软件测试说明

被观测示例系统用于演示 eBPF 采集模块和异常检测算法模块的功能，其涉及原则为功能清晰明了；提供后台接口，能够模拟各种容器异常行为。其拓扑如下：

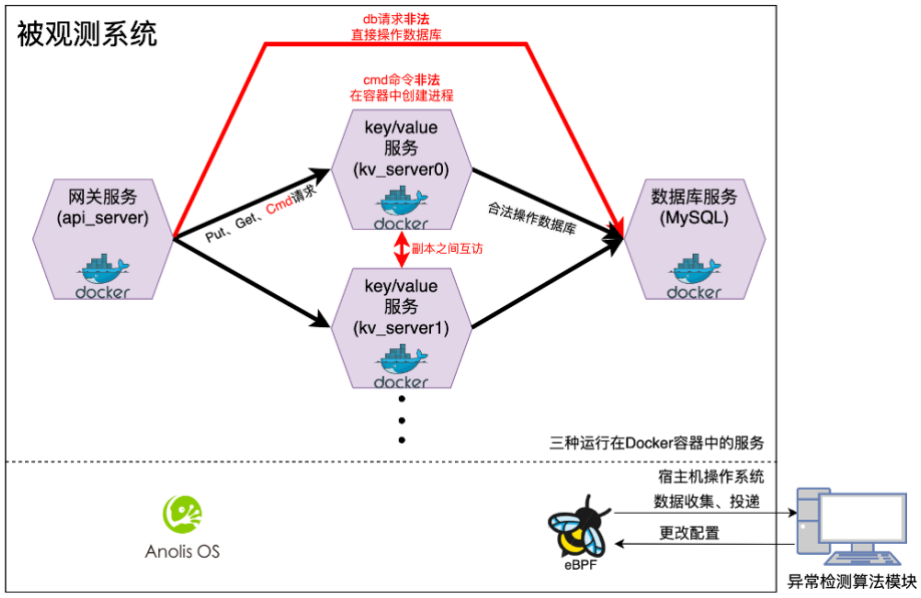

图 8 - 被观测示例系统拓扑图

宿主机使用 Anolis OS 8，在其中运行 3 种 docker 容器服务以及 eBPF 数据采集器：

- kv_server 容器：提供了三个 http 接口。Put、Get 接口用于向数据库中写入和查询 key-value 键值对；Cmd 接口会在容器内部新建进程，执行高危命令。
- MySQL 容器：提供持久化存储服务。
- api_server 容器：提供网关服务用于流量转发，同时提供了后台 db 接口直接访问数据库，进行未经授权的容器互访。

为确保软件功能的正常工作，我们采用了多种测试手段，涵盖了功能测试、性能测试和稳定性测试。

首先，在被观测的示例系统中，我们预留了后台接口，用于手动触发容器的各种异常行为。这些异常行为包括可疑的系统调用、未经授权的容器互访、异常进程的创建、以及异常的资源使用量等。通过这些预留接口，我们能够模拟和触发各种潜在的异常行为，并观察监控系统是否能够准确捕获和报告这些异常，从而验证系统的功能有效性。

其次，为了评估系统在真实负载下的性能，我们实现了一套加压工具，对系统进行压力测试。该工具通过大量模拟请求和操作，对系统进行高强度负载测试，并记录在此过程中 eBPF 采集模块的 CPU 开销。通过这种方式，我们得出了系统在高负载环境下的资源消耗情况，验证了系统在性能方面的表现。

最后，我们对系统进行了 24 小时的长时间压力测试，以测试其稳定性。在这段时间内，系统持续处理大量数据和请求，并保持对容器行为的实时监控。通过监测系统在长时间高负载下的运行状态，我们能够评估系统的稳定性和可靠性，确保其在实际应用中能够长期稳定运行。

综合以上测试手段，我们对系统的功能、性能和稳定性进行了全面验证，确保其能够在实际应用中高效、可靠地运行。

4.2 软件测试结果

在将系统部署之后，通过预留的 cmd、db 接口触发容器的各种异常行为，均能被识别到。如图所示：

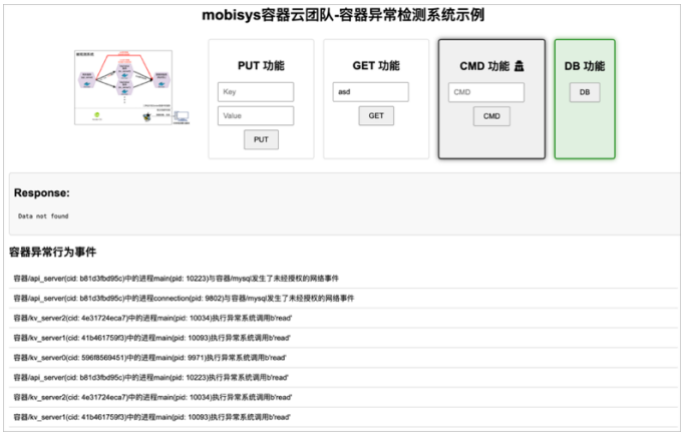

图 9 – 被测试系统前端示意图

同时，Grafana 监控看板正常工作，并且能捕获到符合预期的负载变化：

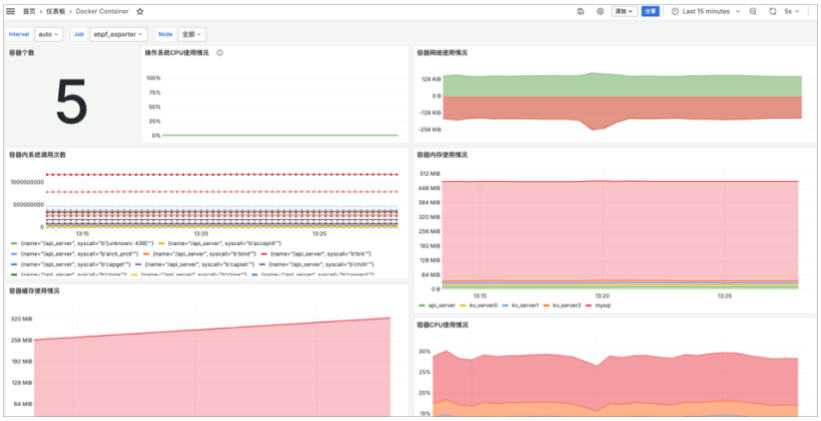

图 10 – Grafana 监控看板示意图

在容器工作负载占用 CPU 80%时，eBPF 采集框架仅占用 1.42%：

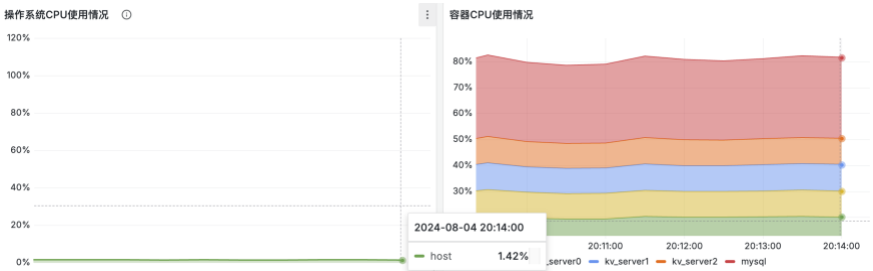

图 11 – CPU 占用情况示意图

第 5 章 实现难点说明

本项目在实现过程中遇到了以下难点：

1. 识别容器内进程
    a) 如何在操作系统内核层面识别容器内进程？
        i. 解决思路：能使用 eBPF 采集操作系统内容器行为的基本前提是容器虚拟化技术共用操作系统内核。在 eBPF 程序的视角中，容器内的程序与宿主机操作系统中的进程无异。因此，如何在 eBPF 程序中识别出容器进程是十分重要的。经过研究，容器技术基于 Linux 操作系统中的 namespace、cgroup 实现。基于此，在进程控制块 task_struct 中通过进程所属的 namespace id 能够判断其是否是容器进程；通过 cgroups 字段能够解析出其所属 cgroup id。
2. 采集文件访问事件
    a) 需要在 eBPF 程序中处理文件路径，需要拼接字符串。
        i. 解决方法：在 eBPF 内核态程序中实现了一套简单的字符串处理工具，能够通过 eBPF 验证器验证。
    b) 文件路径是变长字符串，无法通过 bpf 验证器。
        i. 解决方法：对文件路径的每一层单独存储，将原始数据发送至用户态处理。
3. 采集网络互访事件
    a) TCP 连接状态复杂，难以隔离出真正的通讯报文。
        i. 解决方法：增加 sock_alloc 内核函数作为 hook 点，捕获到容器内进程的网络 socket 创建事件，后续的收发数据包事件均以此为基础。
    b) 将所有进程间通讯的网络事件都采集的开销太大，无法接受。
        i. 解决方法：需要在内核态程序中精准识别出未经授权的容器互访，将规则进行编码并存放在 bpf map 中，用户态程序维护规则集，内核态读取规则集。
4. eBPF 采集模块 CPU 占用过高
    a) 在将 eBPF 采集模块部署后通过加压发现 CPU 占用在 15%左右，无法接受
        i. 解决方法：对 eBPF 采集器进行性能分析、调优，在关键函数处加入缓存优化。最终将 CPU 占用降低至 5%左右，符合赛题要求。

第 6 章 总 结

本项目顺利完成了赛题所要求的全部任务，覆盖了所有指标，甚至超额完成了既定目标。在整个开发过程中，我们克服了各种技术难题，并最终实现了一个高效、全面的容器异常行为检测系统。项目的成功得益于指导老师杜军朝教授的悉心指点和团队成员的共同努力。

6.1 任务完成情况

1. 指标覆盖：项目严格按照赛题要求，完成了所有功能模块的设计和实现，包括容器行为的实时监控、异常检测、数据可视化等关键环节。我们通过 eBPF 技术收集了容器的各种运行时行为特征数据，包括：系统调用频率、系统调用序列、容器间网络互访、IO 吞吐、内存利用率、CPU 利用率等行为特征和指标特征。通过这些数据，我们采用人工智能算法自动识别具有异常行为的容器，包括可疑的系统调用、未经授权的容器互访、容器内异常进程的创建、异常的资源使用量等。所有赛题指标均被覆盖，系统在性能、可靠性、准确性等方面达到了预期要求，并在一些方面实现了超额完成。

2. 性能：采集框架具有可忽略的性能开销，实测 eBPF 采集模块的 CPU 占用被控制在 5%以内，保证被监控容器的正常流畅运行。

3. 可靠性：系统经过严格测试，在各种高负载和复杂场景下均表现稳定。

4. 准确性：通过机器学习和深度学习算法，系统能够高准确率地检测各种异常行为，选择合适的指标作为输入，有效识别不同类型的异常。

赛题要求	指标描述	完成情况
可扩展的数据采集框架	容器的行为特征：系统调用频率、系统调用序列、文件访问、网络通信； 容器的指标特征：CPU利用率、内存利用率、网络使用率 可扩展能力：用户只需实现eBPF内核态程序和用户态数据类型接口，即可添加新的数据采集类型	**超额完成**
可忽略的性能开销	CPU占用控制在10%以内：在容器CPU利用率80%的情况下，采集框架的性能开销低于5%。	**超额完成**
准确的检测算法	1. 通过黑名单规则识别未经授权的容器互访、容器内异常进程的创建； 2. 通过利用机器学习（K-medoids、LGBM）、深度学习(LSTM)实现容器异常行为检测，包括可疑的系统调用、异常的资源使用量。	**超额完成**
清晰的数据、检测过程展示界面	搭建Grafana看板和可配置Web界面，用户可以通过Web页面查看容器行为特征、指标特征以及异常情况。	**超额完成**

表 12 – 赛题要求完成情况表

6.2 比赛收获

1. 校赛第一：在首届西电研究生操作系统开源创新大赛（校赛）中，我们的项目获得了一等奖。这一荣誉不仅是对我们团队工作的认可，也激励我们在未来的科研和项目开发中继续探索和创新。

2. 深入理解 eBPF 技术：通过本次项目，团队成员对 eBPF 技术有了更深入的理解，使用更加熟练，为未来的技术研发打下了坚实基础。

3. 实际应用：我们在实验室中的其他项目和科研工作中也开始应用本次比赛项目的功能，提升了整体项目的技术水平和应用效果。

技术细节：

4. 参考资料：项目参考了 Linux 内核源码，并在此基础上进行了 eBPF 程序设计。

5. 编程语言：我们使用了 Golang、C、Python 等编程语言来实现系统的各个模块，前端部分则使用了 Vue 框架。

6. 开源库和工具：项目中使用了 Linux 内核提供的 libbpf 库，Cilium 公司开源的 go-ebpf 库，以及 Kafka、Grafana 等开源项目，极大地提升了项目的开发效率和功能完备性。

6.3 鸣谢

特别感谢指导老师杜军朝教授在整个项目开发过程中的悉心指导和宝贵建议。此外，我们还要感谢以下开源项目和贡献者：

- Linux 社区：提供了强大的内核功能和参考资料。

- Cilium 公司：开源了 go-ebpf 库，为我们的项目提供了重要支持。

- Kafka 和 Grafana 社区：提供了高效的数据处理和可视化工具。

6.4 展望未来

虽然项目已经取得了一定的成果，但我们不会止步于此。未来，我们计划继续优化系统性能，增强功能，进一步提升容器安全监控的效率和准确性。我们期望该项目能够在更多实际应用中发挥作用，为更多的用户提供可靠的安全保障。

通过本次比赛，我们不仅收获了技术上的提升，更收获了团队合作的宝贵经验。我们将继续努力，争取在未来的科研和项目开发中取得更大的突破。

参 考

[1] **CVE-2024-21626**. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-21626

[2] **eBPF**. https://ebpf.io/

[3] **Falco**. https://falco.org/

[4] **cAdvisor**. https://github.com/google/cadvisor

[5] **Cilium/ebpf**. https://github.com/cilium/ebpf

[6] **Linux**. https://github.com/torvalds/linux