杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

2024 年全国大学生计算机系统能力大赛
操作系统设计赛（华东区域赛）

Computer System Development Capability Competition

Beefine：图形化 BPF 交互与容器观测系统

队 伍 名 称 ：	霓虹 Ultra
队 伍 编 号 ：	T202410336994295
队 伍 成 员 ：	蔡龙祥 谭文轩 李睿涵
指 导 教 师 ：	刘 真 周 旭
参 赛 赛 道 ：	OS 应用开发

2024 年 12 月

目录

第 1 章　概述 3
1.1　项目背景 3
1.2　项目简介 3
1.3　项目应用场景 3
第 2 章　项目开发进度 4
2.1　功能实现进度 4
2.2　模块开发进度 5
第 3 章　整体框架及功能模块设计 6
3.1　系统整体架构 6
3.2　UI 模块（数据处理层） 7
3.3　BPF 内核模块（功能逻辑层） 8
3.3.1　基于 cilium/bpf2go 和 libbpf 8
3.3.2　基于 bpftrace 脚本（scripts 目录下） 13
3.3.3　用户态程序 15
3.4　功能模块 17
3.4.1　Docker DashBoard 18
3.4.2　Docker-Image Monitor 模块 18
3.4.3　Docker-Container Monitor 模块 19
第 4 章　项目创新点 22
第 5 章　项目测试与分析 23
5.1　Load eBPF 模块测试 23
5.1.1　ListHelpFunction 功能测试 23
5.1.2　InspectNetwork 功能测试 23
5.1.3　TraceSyscall/Exec 功能测试 24
5.2　Docker 模块测试 24
5.2.1　ImageMonitoring 功能测试 24
5.2.2　ContainerMonitoring 功能测试 26
第 6 章　项目管理与团队管理 27
6.1　Scrum 敏捷开发管理 27
6.2　工作量估算 27
6.3　项目开发流程 28
6.4　团队与人员管理 30
第 7 章　总结与展望 31
参考文献 31

第 1 章　概述

1.1　项目背景

随着容器化技术在现代软件开发和部署中的广泛应用，Docker 作为其代表性工具，已经成为开发者和运维人员的核心选择。然而，尽管容器技术极大地简化了应用的部署和管理，许多开发者对容器创建过程中的底层机制仍缺乏深入了解。例如，操作系统在处理镜像解压、文件系统挂载、网络隔离等关键步骤时的具体实现细节，往往不为人知。此外，随着 Docker Swarm 和 Kubernetes 等容器编排系统的普及，应用系统中的容器数量急剧增加，理解单个 Pod 中容器的通信方式和最小单元显得尤为重要。

为了帮助开发者更好地理解这些概念，Beefine 应运而生。

1.2　项目简介

Beefine 是一个基于 Fyne 和 Cilium eBPF 框架开发的工具，旨在通过图形化交互界面（GUI）实时监控 Docker 容器的创建过程，帮助用户深入理解虚拟化技术的核心理念和实现原理。项目不仅支持加载和管理 eBPF 程序，追踪操作系统在 Docker 操作中的行为，还计划在未来扩展到 Kubernetes 集群的 Pod 监控，为用户提供更加全面的容器化技术洞察。

Beefine 通过 eBPF 技术捕获和分析容器的系统调用行为，并借助 Fyne 提供图形化的实时反馈，最终实现以下目标：

1) **简化学习过程**：通过实时观测容器创建过程，帮助开发者更好地理解容器镜像，虚拟化技术。
2) **提高可视化交互体验**：提供直观的图形界面，展示关键的系统行为和资源变化；配置了动态加载 bpf 程序的入口，让使用者快速实践 bpf。
3) **开发工具化**：为学习者和工程师提供一个可以随时实验和验证的工具，减少操作系统实验的门槛。

1.3　项目应用场景

本项目目前专注于实时观测场景：

1) **观测 Docker 容器的创建**：Beefine 通过实时加载 eBPF 程序，从而全面捕捉 Docker 使用镜像创建容器时的操作系统行为。通过详细记录和分析在容器创建过程中的镜像拉取、文件系统设置、命名空间管理和网络配置等多个关键步骤，Beefine 为学习者和开发者提供了一个能够帮助他们深入理解容器核心机制的实验环

境，从而提升其对容器技术的掌握和应用能力。

2) **观测 Docker 容器的运行**：在容器运行阶段，Beefine 实时展示容器所处的命名空间（namespace）和控制组（cgroup）信息，并允许用户进行相应的设置。通过动态加载 XDP 程序，有效管理和分析容器网络中的数据包，确保网络通信的高效与安全。同时还支持动态观测容器内的进程信息，使用户能够实时监控容器内运行的各类进程，及时发现和解决潜在的问题，从而优化容器的运行性能和稳定性。

3) **管理动态 eBPF 程序**：Beefine 提供了一个直观的界面，允许用户轻松加载和管理 eBPF 程序，使得用户能够根据具体需求动态分析系统行为，灵活应对不同的监控和调试场景。通过简化 eBPF 程序的管理过程，Beefine 大大提升了用户在系统行为分析和性能调优方面的效率，使开发者和运维人员能够更加专注于核心任务，减少了复杂配置和操作的时间成本。

第 2 章　项目开发进度

2.1　功能实现进度

表 1　功能实现进度表

功能	完成情况	具体描述
Docker **创建过程观测**	已实现	使用 eBPF 追踪操作系统调用，捕获 Docker 使用镜像创建容器的全过程。
图形化界面	已实现	使用 Fyne 开发可交互的 GUI，包括日志查看、动态程序加载等功能。
实时 eBPF 程序加载	已实现	支持用户加载自定义 eBPF 程序，动态分析特定行为。
Kubernetes 集群观测	待开发	设计用于追踪 Kubernetes Pod 调度与运行的功能模块。
镜像与容器的性能分析	待开发	提供更多统计功能，分析资源使用情况（CPU、内存、I/O 等）。
历史数据管理	待开发	支持保存和回放观测结果，以便后续分析。
优化 GUI	待开发	增加更多交互功能，例如高级过滤、实时图表更新。

2.2 模块开发进度

表 2 模块开发进度表

模块大类	完成情况	模块具体描述
功能模块	已实现	易于使用且可交互的 bpf 载入界面，基于 golang 的 UI 库 fyne 客制化通用组件，以实现较好的复用性。
	已实现	提供模板化的框架式 bpf 编程体验，便于使用者轻松编写自己的 bpf 程序，减轻搭建系统环境的负担，支持动态加载 eBPF 程序，结合客制化的组件，可以快速观察运行结果。
	已实现	聚合基础的 docker cli 功能（镜像拉取、容器创建、状态展示），并且提供基础的 Docker 对象观测功能（可选加载 bpf 程序实时分析容器创建与运行中的系统调用、文件系统操作、网络传输、隔离控制的信息）。
	待开发	结合 prometheus 以及 echarts 等组件实现更加完善的 docker 容器内部负载观测视图。
	待开发	使观测程序不局限于操作系统，实现即使不原生支持 bpf 技术的操作系统也可以通过远程连接的方式观测远程服务器中的容器情况。
	待开发	自动化压测测试案例设计。
Docker 创建观测模块	已实现	镜像文件的拉取与解压。
	已实现	读取解析镜像文件结构打包成 runtime bundle（VFS）。
	已实现	基于 xdp 提取从网络中获取镜像信息的数据包。
	已实现	文件系统的挂载（mount）。
	已实现	容器隔离环境的设置（Namespace 和 Cgroup）。
Docker 运行观测模块	已实现	隔离文件信息。
	已实现	挂载磁盘信息。
	已实现	隔离在相同命名空间下的进程、网络信息。
	已实现	观测 CPU 信息。
	已实现	观测内存信息。
	待开发	补充提取其它命名空间下信息的功能。
	待开发	对容器的 namespace 和 cgroup 等隔离机制的交互式改造。
后续扩展场景模块	待开发	对 docker 底层 containerd、runc 等进行更细致的观测。
	待开发	Kubernetes 集群中的 pod 行为观测。
	待开发	日志数据存盘回放。

第 3 章　整体框架及功能模块设计

3.1　系统整体架构

系统开发过程中保持了一致的编码风格，以确保代码的可读性和维护性。每个功能模块的实现均基于内核的 eBPF 模块、用户应用模块与 UI 模块的协同结合，从而实现高效且模块化的系统架构。整体系统架构如下图所示。

图 1　系统总体架构图

3.2 UI 模块（数据处理层）

Beefine 的用户界面（UI）设计秉承简约原则，采用标签页（Tab）与树形目录（Tree）相结合的分页式布局，以提升用户的操作便捷性和界面的整洁性。

UI 模块基于 Go 语言的 Fyne 框架构建，充分利用了多种可复用的自定义组件。每种组件通过统一的映射（map）进行定位，避免了页面的重复创建，进而减少资源浪费，优化系统性能。代码结构遵循 Go 语言的包（package）隔离规则，每个包内的组件独立管理，确保各模块之间的相互独立与互不干扰。此外，项目设计了全局资源管理器，负责监管各包下独立组件的资源分配与管理，进一步提升系统的稳定性和可扩展性。

在全局目录树结构的管理视图中，采用两个映射（map）来表示节点之间的连接关系，作为 Fyne 库创建目录树的基础。各个视图的创建函数（Screen 函数）按照包进行相互隔离，简化了视图的管理与维护。

```go
// Watcher 目录树中的结点数据结构
/*
 *title需要是每个层级唯一的
 *Intro包含了页面的简单信息
 *View是页面canvas对象的构造函数，在运行时调用
 */
type Watcher struct {  4 usages  shoggothforever
    Title, Intro string
    View         func(w fyne.Window) fyne.CanvasObject
}

// Watchers 应用程序UI目录树中的结点信息
var Watchers = map[string]Watcher{  4 usages  shoggothforever
    "welcome":   { Title: "welcome",  Intro: "Welcome to the beefine observer",  View: welcome.Screen},
    "BPF":       { Title: "Load eBPF",  Intro: "Observe system-level activities",  View: bpf.Screen},
    "Docker":    { Title: "Docker",  Intro: "Monitor Docker activities",  View: docker.Screen},
    "imager":    { Title: "Image Monitoring",  Intro: "Monitor Docker imager creation process",  View: imager.Screen},
    "container": { Title: "Container Monitoring",  Intro: "Monitor running container performance",  View: container2.Screen},
}

// WatcherIndex 目录树UI中各个节点的连接关系
var WatcherIndex = map[string][]string{  3 usages  shoggothforever
    "":       {"welcome", "BPF", "Docker"},
    "Docker": {"imager", "container"},
}

// viewsSet 保存创建过的canvas信息，避免重复创建
var viewsSet = map[string][]fyne.CanvasObject{}  4 usages  shoggothforever
```

```go
package component

import ...

// TabManager,每个package中唯一,用于管理页面
type TabManager struct {  9 usages  shoggothforever
    TabItemsMap map[string]*container.TabItem
    Tabs        *container.AppTabs
    w           *fyne.Window
    m           sync.Mutex
}
```

```go
// tabManagerMap 键对应package，值对应package的tabManager
var tabManagerMap = make(map[string]*TabManager)  5 usages  shoggothforever

func NewTabManager(pkg string, w *fyne.Window) *TabManager {  2 usages  shoggothforever
    if _, ok := tabManagerMap[pkg]; !ok {
        tabManagerMap[pkg] = &TabManager{
            TabItemsMap: make(map[string]*container.TabItem),
            Tabs:        container.NewAppTabs(),
            w:           w,
            m:           sync.Mutex{},
        }
    }
    return tabManagerMap[pkg]
}
```

通过这种设计实现了高效的资源管理和模块隔离，确保系统在多功能扩展中的灵活性和稳定性。

3.3 BPF 内核模块（功能逻辑层）

图 2　BPF 内核模块框架图

BPF 内核模块由基础的 eBPF 内核组件（采用 C 语言实现）和用户态程序（采用 Go 语言实现）构成。该模块设计为可供用户动态加载，旨在帮助使用者深入理解 eBPF 程序的运行机制和效果。通过结合内核级别的监控与用户态的交互，BPF 内核模块为用户提供了一个直观的平台，用于观察和分析 eBPF 程序在实际操作系统环境中的表现，从而促进对 eBPF 技术的全面掌握和应用。

3.3.1 基于 cilium/bpf2go 和 libbpf

BPF 内核模块涵盖多种 BPF hook 及其相关程序，旨在为用户提供全面的 eBPF 开发与监控支持。

首先，模块展示了系统中可用的 BPF 辅助函数，旨在为初学者提供开发上的指导与支持，简化 eBPF 程序的编写过程。

其次，模块允许用户选择特定的网卡接口，实时获取通过该接口的数据包计数，并展示部分报文信息。

图 3　实时获取的数据包

这一功能不仅有助于用户监控网络流量，还能深入分析数据包的内容，从而增强对网络通信行为的理解。通过结合 bpf2go 和 libbpf，BPF 内核模块实现了高效的程序加载与管理，确保了系统资源的优化利用和稳定运行。

对于 UDP 或 TCP 报文，转换得到具体的报文首部数据结构，提取其中信息。

```
// XDP 程序：统计包数量并提取网络包信息
SEC("xdp")
int count_and_info(struct xdp_md *ctx) {
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    // 1. 包计数
    __u32 key = 0;
    __u64 *count = bpf_map_lookup_elem(&pkt_count, &key);
    if (count) {
        __sync_fetch_and_add(count, 1);
    }
    // 2. 解析以太网头部
    struct ethhdr *eth = data;
    if ((void *)(eth + 1) > data_end) {
        return XDP_PASS; // 检查包大小合法性
    }
    // 只处理 IPv4 数据包
    if (eth->h_proto != __constant_htons(ETH_P_IP)) {
        return XDP_PASS;
    }
```

```
    // 3. 解析 IP 头部
    struct iphdr *ip = data + ETH_HDR_LEN;
    if ((void *)(ip + 1) > data_end) {
        return XDP_PASS;
    }
    struct pkt_info pkt = {};
    pkt.src_ip = ip->saddr;      // 源 IP
    pkt.dst_ip = ip->daddr;      // 目标 IP
    pkt.protocol = ip->protocol; // 协议类型
    // 4. 解析 TCP/UDP 头部
    if (ip->protocol == IPPROTO_TCP) {
        struct tcphdr *tcp = (void *)ip + ip->ihl * 4;
        if ((void *)(tcp + 1) > data_end) {
            return XDP_PASS;
        }
        pkt.src_port = __constant_ntohs(tcp->source);
        pkt.dst_port = __constant_ntohs(tcp->dest);
    } else if (ip->protocol == IPPROTO_UDP) {
        struct udphdr *udp = (void *)ip + ip->ihl * 4;
        if ((void *)(udp + 1) > data_end) {
            return XDP_PASS;
        }
        pkt.src_port = __constant_ntohs(udp->source);
        pkt.dst_port = __constant_ntohs(udp->dest);
    }
    // 5. 传递网络包信息到用户空间
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &pkt, sizeof(pkt));
    return XDP_PASS;
}
```

BPF 内核模块通过加载特定的 eBPF 程序对系统中的 exec 系统调用进行全面监测。该模块在 exec 调用的入口和出口处分别设置了钩子函数，能够实时捕捉和记录程序的运行与退出事件，不仅能够展示当前系统中正在运行的进程，还能跟踪进程的终止情况，从而提供对操作系统进程生命周期的深刻洞察。

图 4　系统调用监测

下面是 BPF 程序源码。通过利用 eBPF 辅助函数获取必要的进程标识符（PID）、命令信息以及程序的 task struct。程序通过 eBPF map 接收用户态传入的容器 PID（即容器启动后启动的 sh/bash 的 PID），从而有效地过滤掉容器外的进程信息。这一机制确保了监测的进程仅限于特定容器内运行的进程，增强了监控的精确性和针对性，并提供详尽的系统行为监控数据，支持用户深入理解和优化容器化环境中的应用运行状态。

```
err := objs.CgPidMap.Put(CgMapKey, req.ContainerPid)
```

```
SEC("tracepoint/sched/sched_process_exit")
int handle_exit(struct trace_event_raw_sched_process_template *ctx)
{
    struct event *e;
    int pid, tid,prio;
    int *cg_pid;
    __u64 id, ts, *start_ts, duration_ns = 0;
    /* get PID and TID of exiting thread/process */
    id = bpf_get_current_pid_tgid();
    pid = id >> 32;
    tid = (__u32)id;
    /* ignore thread exits */
    if (pid != tid)
        return 0;
    cg_pid=bpf_map_lookup_elem(&cg_pid_map,&container_map_key);
    if (!cg_pid)
        return 0;
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    __u32 host_ppid = BPF_CORE_READ(task,real_parent,pid);
    /*compare with the container's pid*/
    if (!*cg_pid&&host_ppid != *cg_pid)
        return 0;
    /* reserve sample from BPF ringbuf */
    e = bpf_ringbuf_reserve(&rb, sizeof(struct event), 0);
    if (!e)
        return 0;
    /* fill out the sample with data */
    bpf_map_delete_elem(&cg_pid_map, &pid);
    e->prio=ctx->prio;
    e->exit_event = true;
    e->ts = bpf_ktime_get_ns();
    e->pid = pid;
    bpf_get_current_comm(&e->comm, sizeof(e->comm));
    /* send data to user-space for post-processing */
    bpf_ringbuf_submit(e, 0);



    return 0;
}
```

tracepoint/syscalls/sys_enter_mount：

```
// 捕获 sys_enter_mount 事件
SEC("tracepoint/syscalls/sys_enter_mount")
int trace_enter_mount(struct trace_event_raw_sys_enter *ctx) {
    struct mount_event *evt;
    __u32 pid = bpf_get_current_pid_tgid() >> 32;
    evt = bpf_ringbuf_reserve(&events, sizeof(*evt), 0);
    if (!evt){
        return 0;
    }
    evt->pid = pid;
    // 读取挂载参数
    bpf_probe_read_str(&evt->dev_name, sizeof(evt->dev_name), (const char*)ctx->args[0]);
    bpf_probe_read_str(&evt->dir_name, sizeof(evt->dir_name),  (const char*)ctx->args[1]);
    bpf_probe_read_str(&evt->type, sizeof(evt->type),  (const char*)ctx->args[2]);
    // 输出事件到用户空间
    bpf_ringbuf_submit(evt, 0);
    return 0;
}
```

tracepoint/syscalls/sys_enter_openat：

```
SEC("tracepoint/syscalls/sys_enter_openat")
int trace_openat(struct trace_event_raw_sys_enter *ctx) {
    struct event *e;
    const char *filename = (const char *)ctx->args[1];
    __u32 pid = bpf_get_current_pid_tgid() >> 32;
    e = bpf_ringbuf_reserve(&es, sizeof(*e), 0);
    if (!e) return 0;
    bpf_get_current_comm(&e->comm, sizeof(e->comm));
    bpf_probe_read_str(&e->filename, sizeof(e->filename), filename);
    bpf_probe_read_str(&e->operation, sizeof(e->operation), "openat");
    e->pid = pid;
    e->bytes = 0; // 初始为 0
    bpf_ringbuf_submit(e, 0);
    return 0;
}
```

tracepoint/syscalls/sys_enter_read：

```
SEC("tracepoint/syscalls/sys_enter_read")
int trace_read(struct trace_event_raw_sys_enter *ctx) {
    __u64 fd = ctx->args[0];
    __u64 bytes = ctx->args[2];
    __u32 pid = bpf_get_current_pid_tgid() >> 32;
    struct event *e;
    e = bpf_ringbuf_reserve(&es, sizeof(*e), 0);
    if (!e) return 0;
```

```
    bpf_get_current_comm(&e->comm, sizeof(e->comm));
    bpf_probe_read_str(&e->operation, sizeof(e->operation), "read");
    e->pid = pid;
    e->bytes = bytes;

    bpf_ringbuf_submit(e, 0);
    return 0;
}
```

3.3.2 基于 bpftrace 脚本（scripts 目录下）

bpftrace 是一个强大的辅助工具，允许用户通过单行代码快速启动 eBPF 程序。该工具不仅实现了专用的 bpftrace 脚本语言，还提供了丰富的内置功能，极大地简化了 eBPF 程序的开发与部署过程。借助于 bpftrace 工具，我们能够在短时间内实现大量的 hook 函数，显著提升了系统监测与分析的效率。涉及的 hook 点如下：

tracepoint:syscalls:sys_enter_clone

tracepoint:syscalls:sys_enter_unshare

tracepoint:syscalls:sys_enter_setns

tracepoint:syscalls:sys_enter_seccomp

tracepoint:syscalls:sys_enter_prctl

tracepoint:cgroup:cgroup_attach_task

tracepoint:syscalls:sys_enter_socket

tracepoint:syscalls:sys_enter_bind

tracepoint:syscalls:sys_enter_listen

tracepoint:syscalls:sys_enter_accept

tracepoint:syscalls:sys_enter_connect

tracepoint:syscalls:sys_enter_epoll_create

```
#!/usr/bin/bpftrace
// bpftrace -lv tracepoint:syscalls:sys_enter_clone
//    int __syscall_nr
//    unsigned long clone_flags
//    unsigned long newsp
//    int * parent_tidptr
//    int * child_tidptr
//    unsigned long tls
tracepoint:syscalls:sys_enter_clone
{ printf("[clone] pid=%d comm=%s flags=%x parent=%d,child=%d\n"
,pid,comm, args->clone_flags,*args->parent_tidptr,*args->child_tidptr); }
```

```
// bpftrace -lv tracepoint:syscalls:sys_enter_unshare
//    int __syscall_nr
//    unsigned long unshare_flags
tracepoint:syscalls:sys_enter_unshare
{ printf("[unshare] flags=%x\n", args->unshare_flags); }

// bpftrace -lv tracepoint:syscalls:sys_enter_setns
//    int __syscall_nr
//    int fd
//    int flags
tracepoint:syscalls:sys_enter_setns
{ printf("[setns] fd=%d, nstype=%x\n", args->fd, args->flags); }

tracepoint:syscalls:sys_enter_seccomp
/*
    int __syscall_nr
    unsigned int op
    unsigned int flags
    void * uargs
*/
{ printf("[seccomp] op=%d, flags=%x\n", args->op, args->flags); }

// bpftrace -lv tracepoint:syscalls:sys_enter_prctl
//    int __syscall_nr
//    int option
//    unsigned long arg2
//    unsigned long arg3
//    unsigned long arg4
//    unsigned long arg5
tracepoint:syscalls:sys_enter_prctl {
$opt=args->option;
printf("[prctl]comm=%s option=%d\n",comm,$opt);
}

// bpftrace -lv tracepoint:cgroup:cgroup_attach_task
/*
tracepoint:cgroup:cgroup_attach_task
```

kprobe:tcp_connect 观测 tcp 连接建立的过程状态。

```
kprobe:tcp_connect
{
    @tcp_states[1] = "ESTABLISHED";
    @tcp_states[2] = "SYN_SENT";
    @tcp_states[3] = "SYN_RECV";
    @tcp_states[4] = "FIN_WAIT1";
    @tcp_states[5] = "FIN_WAIT2";
    @tcp_states[6] = "TIME_WAIT";
    @tcp_states[7] = "CLOSE";
    @tcp_states[8] = "CLOSE_WAIT";
    @tcp_states[9] = "LAST_ACK";
    @tcp_states[10] = "LISTEN";
    @tcp_states[11] = "CLOSING";
    @tcp_states[12] = "NEW_SYN_RECV";
  $sk = ((struct sock *) arg0);
  $inet_family = $sk->__sk_common.skc_family;
  if ($inet_family == AF_INET || $inet_family == AF_INET6) {
    if ($inet_family == AF_INET) {
      $daddr = ntop($sk->__sk_common.skc_daddr);
      $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
    } else {
      $daddr = ntop($sk->__sk_common.skc_v6_daddr.in6_u.u6_addr8);
      $saddr = ntop($sk->__sk_common.skc_v6_rcv_saddr.in6_u.u6_addr8);
    }
    $lport = $sk->__sk_common.skc_num;
    $dport = $sk->__sk_common.skc_dport;
    // Destination port is big endian, it must be flipped
    $dport = ($dport >> 8) | (($dport << 8) & 0x00FF00);
    $state = $sk->__sk_common.skc_state;
    $statestr = @tcp_states[$state];
    printf("[tcp_connect] pid:%d comm:%s ", pid, comm);
    printf("saddr:%s lport:%d daddr:%s dport:%d state:%s\n", $saddr, $lport, $daddr, $dport,$statestr);
  }
}
```

3.3.3 用户态程序

用户态程序采用统一的模板设计，要求用户自行定义在载入 eBPF 程序时所需的请求结构体，并从 eBPF 程序输出的管道中获取响应结构体。通过利用 bpf2go 工具链，用户可以一键生成相关的 eBPF 用户态加载函数，简化了 eBPF 程序的集成过程。用户态程序支持与 eBPF 数据结构的高效交互，确保数据在内核态与用户态之间的顺畅传递。为了保证每个 eBPF 程序能够正常退出，系统采用闭包的形式创建具有“仅执行一次”语义的清理函数构造器，确保了资源的及时释放和程序的稳定退出。

```
//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -target bpfel -type event bpf $PACKAGE_NAME.c -- -I /sys/kernel/btf
type ${PACKAGE_NAME^}Req struct {

}
type ${PACKAGE_NAME^}Res struct {

}
func Start(req *${PACKAGE_NAME^}Req) ( chan ${PACKAGE_NAME^}Res,func()) {
  stopper := make(chan struct{})
  // Allow the current process to lock memory for eBPF resources.
  if err := rlimit.RemoveMemlock(); err != nil {
    log.Fatal(err)
  }
```

```go
  // Load pre-compiled programs and maps into the kernel.
  objs := bpfObjects{}
  if err := loadBpfObjects(&objs, nil); err != nil {
    log.Fatalf("loading objects: %v", err)
  }
    // write your link code here

    out:=Action(objs, req, stopper)
    buildClose := func() func() {
      once := sync.Once{}
      return func() {
        once.Do(func() {
          objs.Close()
          // close attach
          close(stopper)
          close(out)
        })
      }
    }
  return out,buildClose()
}
func Action(objs bpfObjects , req *${PACKAGE_NAME^}Req , stopper chan struct{}) chan ${PACKAGE_NAME^}Res{
  // add your link logic here
  out := make(chan ${PACKAGE_NAME^}Res)
  go func() {
    for {
      // write your logical code here
      select {
      case <-stopper:
        return
      default:
        time.Sleep(1 * time.Second)
      }
    }
  }()
  return out
}
```

docker cli 管理模块在 docker cli 基础上再次封装，实现了在应用程序中运行 docker 容器，管理容器状态，管理与 docker daemon 的连接复用。

```go
// GetDockerClient 返回单例 Docker 客户端
func GetDockerClient() (*client.Client, error) {  1 usage  shoggothforever
    var err error
    once.Do(func() {
        cliInstance, _ = initDockerClient()
    })
    // 检查连接是否有效
    if !isConnectionValid(cliInstance) {
        fmt.Println( a...: "Docker client connection is invalid. Reinitializing...")
        mu.Lock()
        defer mu.Unlock()
        cliInstance, err = initDockerClient()
        if err != nil { return nil, err }
    }
    return cliInstance, err
}
```

```
// ParseAndRunDockerRun 解析 JSON 并运行 docker 命令
/*
参考示例配置
`{
    "cmd" :  "/bin/sh",
    "image": "nginx",
    "name": "my-container",
    "ports": ["80:80", "443:443"],
    "volumes": ["/host/path:/container/path"],
    "env": ["ENV_VAR1=value1", "ENV_VAR2=value2"],
    "detach": true,
    "rm": true,
    "privileged": false
}`
*/
```

3.4 功能模块

图 5　功能模块框架图

3.4.1 Docker DashBoard

Docker DashBoard 模块旨在实时展示系统中 Docker Daemon 的使用情况，并帮助用户全面了解 Docker 的运行状态。

本模块重点关注以下两个方面：第一，模块能够准确获取系统中当前存在的容器和镜像的数量，提供实时的统计数据，帮助用户掌握 Docker 环境的整体规模和状态。第二，DashBoard 通过监控所有运行中容器的 CPU 和内存使用情况，生成详细的资源使用报告。通过直观的界面展示 Docker 容器和镜像的数量，以及所有运行中容器对系统 CPU 和内存资源的使用情况，从而为用户提供全面的性能监控和资源管理支持。

图 6　Docker DashBoard 模块

3.4.2 Docker–Image Monitor 模块

图 7　Docker–Image Monitor 模块

该模块专注于全面观测 Docker 基于镜像创建容器的全过程，集成了 Docker 对镜像的管理功能。用户可以通过输入镜像名称来拉取镜像，或选择系统中已存在的镜像进行使用。右侧的看板实时展示当前镜像创建的过程日志，提供直观的操作反馈。

在实现方面，模块通过日志还原 Docker 解析镜像创建容器的详细过程，这需要完全掌握 Docker 的镜像解析流程和底层实现原理，如镜像解析遵循 OCI 镜像规范，包含镜像索引、镜像清单、镜像层及镜像配置。镜像索引用于区分不同架构平台的镜像，镜像清单包含镜像内容的哈希摘要，并提供镜像的寻址方式和镜像层信息的提取方法。因此，主要涉及的系统调用包括 openat、read、write 等文件系统接口。Docker 的容器文件系统采用 UnionFS，初始时仅包含一层 rootfs，解析每一层镜像文件后，都会在 rootfs 上

覆盖新增一层。镜像配置主要包含环境变量、执行参数和存储卷等信息，Docker 通过这些配置生成相应的 OCI runtime bundle 以启动容器。

以观测文件系统为例，选择加载程序后就会调用 bpf 模块的启动函数，得到数据的输出管道，该组件也会集中管理 bpf 程序的关闭函数，保证程序能够安全卸载关闭。

```
widget.NewCheck( label: "VFS", imageSelector.chooseVFS),
widget.NewCheck( label: "mount", imageSelector.chooseMount),
widget.NewCheck( label: "network", imageSelector.chooseNetwork),
widget.NewCheck( label: "isolation", imageSelector.chooseIsolation)
```

```
// chooseMount 挂载观测文件系统相关bpf程序
func (w *ImageSelect) chooseVFS(b bool) {  1 usage  shoggothforever
    if b == true {
        w.bpfLogs.AppendLogf( format: "choose watch unionfs")
        req := image_prep.ImagePrepReq{}
        out, cancel := image_prep.Start(&req)

        w.cancelMap["chooseVFS"] = cancel
        go func() {
            mp := make(map[string]int)
            st := time.Now()
            for event := range out {
                comm := helper.Bytes2String(event.Comm[:])
                str := fmt.Sprintf( format: "pid:%d,comm:%s,operation:%s", event.Pid, comm, helper.Bytes2String(event.Operation[:]))
                if _, ok := mp[str]; ok {
                    mp[str]++
                    continue
                }
                mp[str] = 1
                w.bpfLogs.AppendLogf(str)
            }
            for log, count := range mp {
                w.bpfLogs.AppendLogf( format: "%s count:%d during %f s\n ", log, count, time.Since(st).Seconds())
            }
        }()
    } else {
        w.bpfLogs.AppendLogf( format: "cancel watch chooseVFS")
        if w.cancelMap["chooseVFS"] != nil {
            w.cancelMap["chooseVFS"]()
        }

    }

}
```

3.4.3 Docker–Container Monitor 模块

图 8　Docker–Container Monitor 模块

本模块聚焦于运行中的容器的实时数据分析，依据 OCI 运行时规范对容器运行状

态进行监测，主要观测的是处于 stopped 和 running 状态的容器，对于 running 状态的容器可以观测到更多有关的数据。容器的隔离依赖于 Linux 容器技术，Docker 则通过容器运行时来管理这些容器的生命周期和资源分配。

```
widget.NewCheck( label: "isolationInfo", cst.buildChoose(cst.chooseIsolationInfo)),
widget.NewCheck( label: "diskInfo", cst.buildChoose(cst.chooseDiskInfo)),
widget.NewCheck( label: "netInfo", cst.buildChoose(cst.chooseNetInfo)),
widget.NewCheck( label: "process", cst.buildChoose(cst.chooseProcess)),
widget.NewCheck( label: "cpu", cst.buildChoose(cst.chooseCpu)),
widget.NewCheck( label: "memory", cst.buildChoose(cst.chooseMemory)),
```

在实现方面，模块重点分析容器的命名空间和控制组信息，深入探讨处于隔离状态的容器内部进程和网络活动的具体细节。通过 eBPF 程序，模块能够动态监控和记录容器内的系统调用和资源使用情况，从而提供详尽的运行时数据。略有不同的是，部分信息无需通过 eBPF 程序即可直接从系统文件中读取，如容器的隔离信息：

```
// chooseIsolationInfo 获取选中docker容器的隔离信息
func (w *ContainersSelect) chooseIsolationInfo(b bool) {  2 usages  shoggothforever
    if b {
        w.m.Lock()
        stat, err := cli.ContainerInspect(w.currentContainer.ID)
        if err != nil { return }
        if stat.State.Pid == 0 {
            w.m.Unlock()
            w.containerLogs.AppendLogf( format: "container not start")
            return
        }
        pid := stat.State.Pid
        w.containerLogs.AppendLogf( format: "container's pid is %d", pid)
        nsPath := fmt.Sprintf( format: "/proc/%d/ns", pid)
        w.containerLogs.AppendLogf( format: "reading namespaces info: %s ", nsPath)
        // 读取 /proc/{pid}/ns 目录内容
        files, err := os.ReadDir(nsPath)
        if err != nil { return }
        for _, file := range files {
            lk, err := os.Readlink(nsPath + "/" + file.Name())
            if err != nil { return }
            w.containerLogs.AppendLogf( format: "%s\n", lk)
        }
        statusPath := fmt.Sprintf( format: "/proc/%d/status", stat.State.Pid)
        w.containerLogs.AppendLogf( format: "reading pid status: %s ", statusPath)
        data, err := os.ReadFile(statusPath)
        if err != nil { return }
        lines := strings.Split(string(data),  sep: "\n")
        for _, line := range lines {
            if strings.HasPrefix(line,  prefix: "PPid") {
                w.containerLogs.AppendLogf(line)
            }
```

```go
            if strings.HasPrefix(line, prefix: "NSpid") {
                w.containerLogs.AppendLogf(line)
            }
            if strings.HasPrefix(line, prefix: "Seccomp") {
                w.containerLogs.AppendLogf(line)
            }
        }
        cgroupPath := fmt.Sprintf( format: "/proc/%d/cgroup", stat.State.Pid)
        w.containerLogs.AppendLogf( format: "reading cgroup file: %s ", cgroupPath)
        // 读取 /proc/{pid}/cgroup 文件
        data, err = os.ReadFile(cgroupPath)
        if err != nil { return }
        w.containerLogs.AppendLogf(string(data))
        w.m.Unlock()
```

获取与容器处于同一命名空间下的所有进程信息：

```go
// getNsPeers 获取与指定 PID 和 Namespace 类型处于同一 Namespace 的进程
func (w *ContainersSelect) getNsPeersV(ctx context.Context, pid int, nsType string) {  1 usage  shoggothforever *
    // 获取目标 Namespace ID
    nsID, err := cli.GetNamespaceID(pid, nsType)
    if err != nil { log.Fatalf( format: "Failed to get Namespace ID for PID %d and type %s: %v", pid, nsType, err) }
    w.containerLogs.AppendLogf( format: "Monitoring peers in the same namespace ")
    w.containerLogs.AppendLogf( format: "Namespace Type: %s, Namespace ID: %s", nsType, nsID)
    w.containerLogs.AppendLogf( format: "PID     PPID   USER     COMMAND")
    // 使用 map 记录已发现的进程
    discoveredPeers := make(map[int]struct{})
    // 实时监控
    ticker := time.NewTicker(2 * time.Second)
    defer ticker.Stop()
    for {
        select {
        case <-ctx.Done():
            log.Println( v...: "Context canceled. Exiting.")
            return
        case <-ticker.C:
            // 检测 Namespace 中的进程
            currentPeers, err := cli.GetPeersInNamespace(discoveredPeers, nsID, nsType)
            if err != nil {
                log.Printf( format: "Error getting peers: %v", err)
                continue
            }
            // 输出新增的进程
            for _, peer := range currentPeers {
                info, err := cli.GetProcessInfo(peer)
                if err == nil { w.containerLogs.AppendLogf(info) }
            }
        }
    }
}
```

第 4 章　项目创新点

1. **客制化、可复用的 UI**

   Beefine 客制化了多个可复用的 UI，配备了快速生成 BPF 程序框架的脚本，支持使用脚本一键创建 BPF 程序模板，具有较强的拓展性，可以用于实验目的快速添加自定义的 BPF 程序并快速校验。

2. **高内聚、低耦合的并发加载 BPF 程序框架**

   对于加载 BPF 程序的设计，结合了 cilium 库，libbpf 以及 bpftrace 等工具，结合 Go 天然支持并发的特性，抽象出了一套并发加载 BPF 程序的框架，所有 BPF 程序都基于该框架设计，也实现了 UI 模块，应用模块和内核模块之间的高内聚、低耦合。

3. **一键载入 BPF 程序与内核态用户态数据交互**

   我们在限制了可以获取的函数参数信息的条件下编写能够通过 BPF verifier 的 BPF C 程序，然后结合 Go 的 BPF 辅助库实现了一键载入 BPF 程序，并且通过 BPF map、perf、ringbuffer 等数据结构实现内核态和用户态的数据交互，再结合 golang 的 UI 库搭建了可交互的桌面端程序，把底层的细节对用户隐藏，只展示用户需要的数据。

4. **通过日志还原 Docker 镜像解析过程**

   Beefine 可以通过日志还原 Docker 解析 image 创建容器的完整视图，便于使用者快速理解 Docker 解析镜像的过程，了解 Docker 底层的实现原理，镜像解析的过程，理解 OCI 镜像规范定义，包含镜像索引，镜像清单，镜像层以及镜像配置。

5. **交互式载入策略的 VFS、挂载、网络与隔离 API 的 BPF 观测**

   Beefine 实现了对 VFS、挂载、网络和隔离 API 的 BPF 观测程序交互式地载入策略，可以允许使用者按需获取镜像加载数据。

6. **基于 Linux 文件系统的实时 Namespace 内信息获取**

   我们基于 Linux 文件系统的设计结构，通过系统内部程序实现了实时获取同一 Namespace 中的 peer 信息。以 PID Namespace 为例，实现了实时获取容器中新增运行进程信息的展示，减少容器外信息的干扰。其他的 Namespace 空间也类似。

7. **实验后彻底回收容器及其他资源**

   出于实验的用途，实验过程中的产生的一切容器及其它资源都会在程序结束时彻底回收，减少使用者的心智负担，确保系统资源的高效利用。

第 5 章　项目测试与分析

5.1　Load eBPF 模块测试

点击应用左侧 Load eBPF 选项卡进入 home 界面。

5.1.1　ListHelpFunction 功能测试

如图 9 所示，在 home 界面下点击 ListHelpFunction 按钮进入子界面，点击下拉选择框，选择 syscall:选项，可发现应用展示出了 syscall 下的所有 bpf 帮助函数，同时支持复制等操作。

图 9　ListHelpFunction 测试

点击 Close Tab 按钮返回 home 界面。

5.1.2　InspectNetwork 功能测试

如图 10 所示，在 home 界面下点击 InspectNetwork 按钮进入 InspectNetwork 子界面，点击下拉选择框，选择想要追踪的网络，应用自动开始抓取数据报文。

打开终端，输入 ping www.baidu.com 命令，发现应用显示出了数据包数量和部分报文数据，同时再进行定时数据更新。

点击 Stop 按钮，结束网络数据抓包，点击 Close Tab 按钮返回 home 界面。

图 10　InspectNetwork 功能测试

5.1.3　TraceSyscall/Exec 功能测试

如图 11 所示，在 home 界面下点击 TraceSyscall/Exec 按钮进入 TraceSyscall/Exec 子界面，应用自动开始抓取调用的系统调用。

打开终端，输入命令 ping www.baidu.com 命令和 sleep 10 命令，发现应用中已经有若干条系统调用被展示出来。

图 11　TraceSyscall/Exec 功能测试

点击 Stop 按钮，结束系统调用抓包，点击 Close Tab 按钮返回 home 界面。

5.2　Docker 模块测试

点击应用左侧 Docker 选项卡，展开 Docker 模块功能，同时应用进入 DockerDashboard 界面，展示系统中的容器和镜像数量，以及统计所有运行中的容器对系统 CPU 和 memory 资源的使用情况。

5.2.1　ImageMonitoring 功能测试

如图 12(a)所示，点击应用左侧 ImageMonitoring 选项卡，进入 ImageMonitoring 子界

面，点击 SelectExistedImage 下拉选项框，选择 docker:dind，选择想要显示的信息，选中对应复选框，点击 Run 按钮，即可创建一个 Docker。返回 DockerDashboard 界面，发现新创建的 Docker 已被显示出来，如图 12(b)所示。

(a)

(b)

图 12　ImageMonitoring 功能测试

5.2.2 ContainerMonitoring 功能测试

如图 13 所示，点击应用左侧 ContainerMonitoring 选项卡，进入 ContainerMonitoring 子界面，点击下拉选择框，选中一个正在运行的容器，在终端中进入这个容器的交互式 Shell，选中任意复选框，可在左侧输出框中输出可复制的容器的相关信息；选中 netinfo 和 process 复选框，在终端进入容器输入相关命令（例如 ping 命令），应用右侧输出框输出相关信息。

图 13　ContainerMonitoring 功能测试

第 6 章　项目管理与团队管理

6.1　Scrum 敏捷开发管理

参照 Scrum 敏捷开发管理方法，结合项目和团队实际情况，本项目组的 Scrum 敏捷开发流程如图所示。

图 14　敏捷开发流程

Scrum 方法为本团队提供了一套灵活而高效的项目管理框架。本项目遵循定期的 Sprint 周期，在每个 Sprint 开始之前，本团队会举行 Sprint 计划会议，讨论并确定该周期内要完成的工作。在 Sprint 过程中，团队每天进行短暂的站立会议（Daily Stand-up），汇报进度和面临的挑战。Sprint 结束时，团队会举行 Sprint 评审会议，展示完成的工作，并收集反馈。之后，团队会进行 Sprint 回顾会议，总结经验教训，不断优化项目的工作流程。

6.2　工作量估算

本项目组组员共同参与估算。在估算前不指定谁将去开发这个需求，而是以接收需求的小组为单位集体估算，每个人提出自己的看法，大家综合，以集体智慧完成任务。在估算过程中，产品负责人不能离开，因为很多估算差异问题来自于“做什么、做到什么地步”，产品负责人需要予以解答，而不是让团队按自己的理解去做。共同估算是共同跟进的基础，若不能共同估算，则后面的“每日立会”几乎不可能正常进行，因为大家只会关心自己曾经一起分析、思考、提问、设计乃至争论过的任务进展的怎么样了，是不是和自己想象的一样。

共同估算的最佳方法是“扑克估算”，这看似很像一个小游戏，但却是 Delphi 估算的一个快速方法，同时实现了匿名性和高效性。估算扑克的使用方法：

1)　每个团队成员拿到一组卡片，包括 0,0.5,1,2,3,5,8,13,20,40,⋯,∞,共计 12 张。这

些数字使用故事点作为计量单位。

2) 产品负责人扮演阅读者的角色，他负责阅读需要估算产品 Backlog 的条目，并且询问大家是否有疑问。

3) 团队讨论这个条目。

4) 当团队理解了这个条目之后，每个团队成员按照自己的想法给出估算结果，并且选择对应的扑克出牌，估算结果不能告诉其他人，出牌时数字朝下扣在桌面上。所有人都出牌之后，阅读者向大家确认是否都已经确定估算结果，确认后，大家同时展示估算结果。

5) 团队评估不同的估算结果.一般最小和最大的两个人 PK，说出自己的观点，大家一起讨论。大致是两个方面：做什么，怎么做；产品负责人参与讨论回答做什么的问题，其他组员讨论解决怎么做的问题；讨论之后，最终团队需要达成一致。

6) 回到 2)，开始估算下一个条目。

上述估算了产品 Backlog 的故事点总数。下面需要知道每个 Sprint 的团队速率。团队速率是一个 Scrum 团队在一个 Sprint 中实际完成的故事点数，通过团队速率可以知道团队做的有多快。在以往 Sprint 执行过程中，记录每个 Sprint 的速度，作为本项目的参考。

图 15　项目周期推算示意图

6.3　项目开发流程

Beefine 项目的开发流程从 2024 年 11 月 1 日启动，至 2024 年 12 月底，历时约七周。整个开发周期被划分为多个阶段，以确保项目的有序推进和高质量交付。

第一阶段：项目启动与需求分析（11 月 1 日~11 月 3 日）

项目在启动阶段明确了目标和范围，进行了详细的需求收集与分析。团队成员确定

了项目的核心功能，包括实时监控 Docker 容器创建与运行过程、图形化用户界面（GUI）的开发以及 eBPF 程序的动态加载与管理。这一阶段为后续的设计和开发奠定了坚实的基础。

图 16　项目开发流程甘特图

第二阶段：系统设计（11 月 4 日~11 月 10 日）

在需求分析完成后，团队进入系统设计阶段。首先，设计了系统的整体架构，确保内核模块、用户应用模块和 UI 模块的协同工作。接着，详细设计了 UI 模块和 BPF 内核模块，明确了各自的功能和接口，以便于模块化开发和后续的功能扩展。

第三阶段：模块开发（11 月 11 日~12 月 5 日）

模块开发是项目的核心阶段，涵盖了多个子任务：

1) **功能模块开发**：构建基础功能，确保各模块的基本操作和交互顺畅。
2) Docker **创建观测模块开发**：实现对 Docker 容器创建过程的全面监控，包括镜像拉取、文件系统挂载和网络配置等关键步骤。
3) Docker **运行观测模块开发**：开发运行阶段的监控功能，实时展示容器的命名空间、控制组信息及网络活动。
4) UI **界面开发**：使用 Fyne 框架构建直观、交互性强的图形界面，提供日志查看、动态程序加载等功能。
5) eBPF **程序加载功能开发**：实现用户自定义 eBPF 程序的动态加载与管理，确保系统行为的灵活分析和监控。

第四阶段：测试与质量保证（12 月 6 日~12 月 15 日）

在模块开发完成后，进入测试与质量保证阶段。首先进行单元测试，验证各模块的基本功能和稳定性。随后进行集成测试，确保不同模块之间的协同工作无缝衔接。最后，进行用户验收测试，收集用户反馈并进行必要的调整和优化，确保系统满足预期需求。

第五阶段：部署与发布（12 月 16 日~12 月 22 日）

测试完成后，团队开始部署准备工作，包括环境配置、文档编写和部署脚本的完善。随后，进行正式发布，将 Beefine 工具推向用户，并确保部署过程顺利、系统运行稳定。

我们的项目管理与团队协作贯穿整个项目周期，团队通过定期的 Sprint 计划会议、每日站立会议和 Sprint 评审会议，确保项目按时推进，及时解决遇到的问题，不断优化开发流程，提高工作效率。通过上述严谨的开发流程，Beefine 项目在有限的时间内有条不紊地推进，最终实现了实时监控 Docker 容器创建与运行的核心功能，并为未来的扩展和优化奠定了坚实的基础。

6.4 团队与人员管理

表 3 团队成员分工表

团队成员	职责
蔡龙祥	系统整体架构设计 BPF 内核模块设计与开发 Docker 观测模块开发 项目部署与发布
谭文轩	BPF 内核模块设计与开发 eBPF 程序加载与管理 前后端集成 测试与质量保证
李睿涵	Docker 观测模块开发 UI 模块设计与开发 用户体验优化 文档编写与维护

第 7 章　总结与展望

本项目自立项到目前已有一个多月的时间，在这一个多月的时间里，团队共向 gitlab 仓库推送 50 余次，更新文档 10 余版，通过该项目，团队对 eBPF 的了解与掌握程度有了可观的提升。值得欣慰的是，本项目圆满完成了立项时所预定的目标，在简化学习过程、提高可视化交互体验、开发工具化等方面取得了预期成果，同时，通过规范化编程，我们设置了许多可拓展模块，使用者可根据自身学习情况定制化自己的软件体验，这无疑有助于未来的操作系统教学与实践。See the Unknown，Conquer the Unknown，我们很开心能够为这一领域添砖加瓦。

在 Beefine 的开发过程中，我们参考了许多优秀的开源项目，例如 cilium，libbpf，eunomia-bpf，在这里，我们谨向这些伟大的开发团队表达诚挚的感谢。

一个月的时间可以成熟一个人的技术，但绝不会让一个应用变得完美。在未来，团队还将致力于丰富优化本项目，增加对 Kubernetes 的支持、引入实时性能监控、对远程主机的支持、优化可视化体验，引入实时更新图表与分析报告以及对 Windows、MacOS 的支持和对 OpenEuler 等国产操作系统的专门优化。

参考文献

[1] Eunomia-bpf, “eBPF 开发教程”, [Online]. Available: https://github.com/eunomia-bpf/bpf-developer-tutorial/tree/main.

[2] XDP Project, “eXpress Data Path program written tutorial”, [Online]. Available: https://github.com/xdp-project/xdp-tutorial/tree/master.

[3] Cilium Project, “BPF Code Style Reference”, [Online]. Available: https://github.com/cilium/cilium/tree/main/bpf.

[4] Linux Kernel, “BPF Program Samples in Linux Source Code”, [Online]. Available: https://elixir.bootlin.com/linux/v6.11.5/source/samples/bpf.

[5] libbpf-bootstrap, “Libbpf BPF Programming Introduction”, [Online]. Available: https://github.com/libbpf/libbpf-bootstrap/tree/master/examples.

[6] libbpf, “Libbpf API Documentation”, [Online]. Available: https://libbpf.readthedocs.io/en/latest/api.html.

[7] Fyne Framework, “Fyne Demo Application”, [Online]. Available: https://github.com/fyne-io/fyne/tree/master/cmd/fyne_demo.

[8] Kernel.org, “BPF Verifier Rules Documentation”, [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/verifier.html.

[9] Linux Kernel Travel, “eBPF Visualization Project”, [Online]. Available: https://github.com/linuxkerneltravel/lmp/tree/develop/eBPF_Visualization.

[10] bpftrace, “BPFtrace Tools Reference”, [Online]. Available: https://github.com/bpftrace/bpftrace/tree/master/tools.