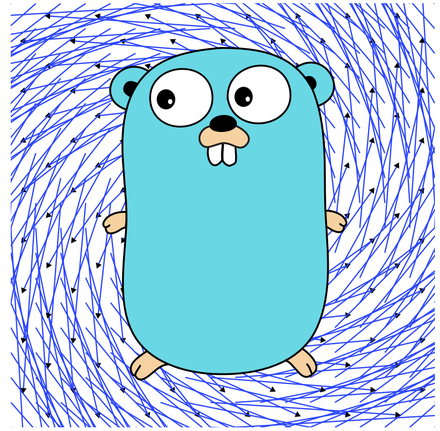


# Flowd-go: Flowd but in Go

## Why rewrite it?

**Pablo Collado Soto <pablo.collado@uam.es> - 03/12/2024**

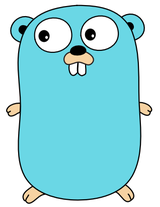
# Who are you again?


- I'm **Pablo Collado** Soto, an engineer working for **ATLAS**.
- Part of my **work** revolves around **managing** a **TIER-2** in Madrid.
- I **attended CHEP** 2024 and sat for **Andrew Bohdan's talk** on Monday...
- ... and **loved** the idea of the **SciTags** initiative!
- Given **`flowd`** leverages **eBPF** I decided to **exercise** my **eBPF-fu** too.
- I've grown **fond** of **Go**, so it was a **no brainer** in the end...
- **Best** of all, folks at **SciTags** would maybe **find** my work **useful**!
- So... we set out to **replicate `flowd`** in Go, hence **`flowd-go`** was born.

# What makes flowd-go different?

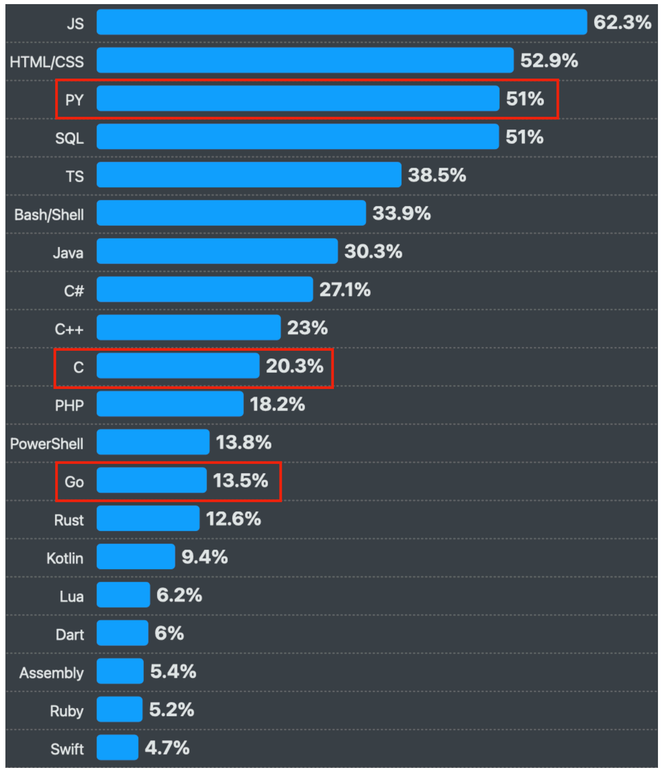
- **Two things** really:
  - It's **written** in **Go** instead of **Python** and so we get:
    - **Built-in concurrency** based on the abstraction of channels.
    - **Statically compiled** binaries.
    - Easier and more **direct access** to **kernel APIs**.
    - 'Free' **cross compilation**.
  - We leverage **libbpf** instead of **BCC**:
    - We **needn't recompile eBPF** programs at **runtime**!

# Is Go really that great?



- It **has** its **flaws**, but I really **like it**!
- Key **benefits** include:
  - **Static typing**.
  - Great **standard library** and **toolchain**.
  - Fast **execution speed**:
    - **Comparable** to **Rust's** and **C++'s**!
  - **Docker/k8s/eBPF** are **written** in **Go**.
  - Great **interop** with **C**.

Stack Overflow 2024 Survey Section 2.1



# And what's libbpf now?


- Put simply, it's a **C-based** library **handling** the **lifecycle** of **BPF** objects.
- The user/developer provides a precompiled program and **`libbpf`**:
    1. **Loads** the program into the kernel.
    2. **Verifies** the correctness of the program.
    3. **Attaches** the program to the specified hook.
- **Best** feature by far is **Compile Once:Run Everywhere** (CO:RE) **support**!
- We'll be using a **Go wrapper** around libbpf: **`libbpfgo`**.

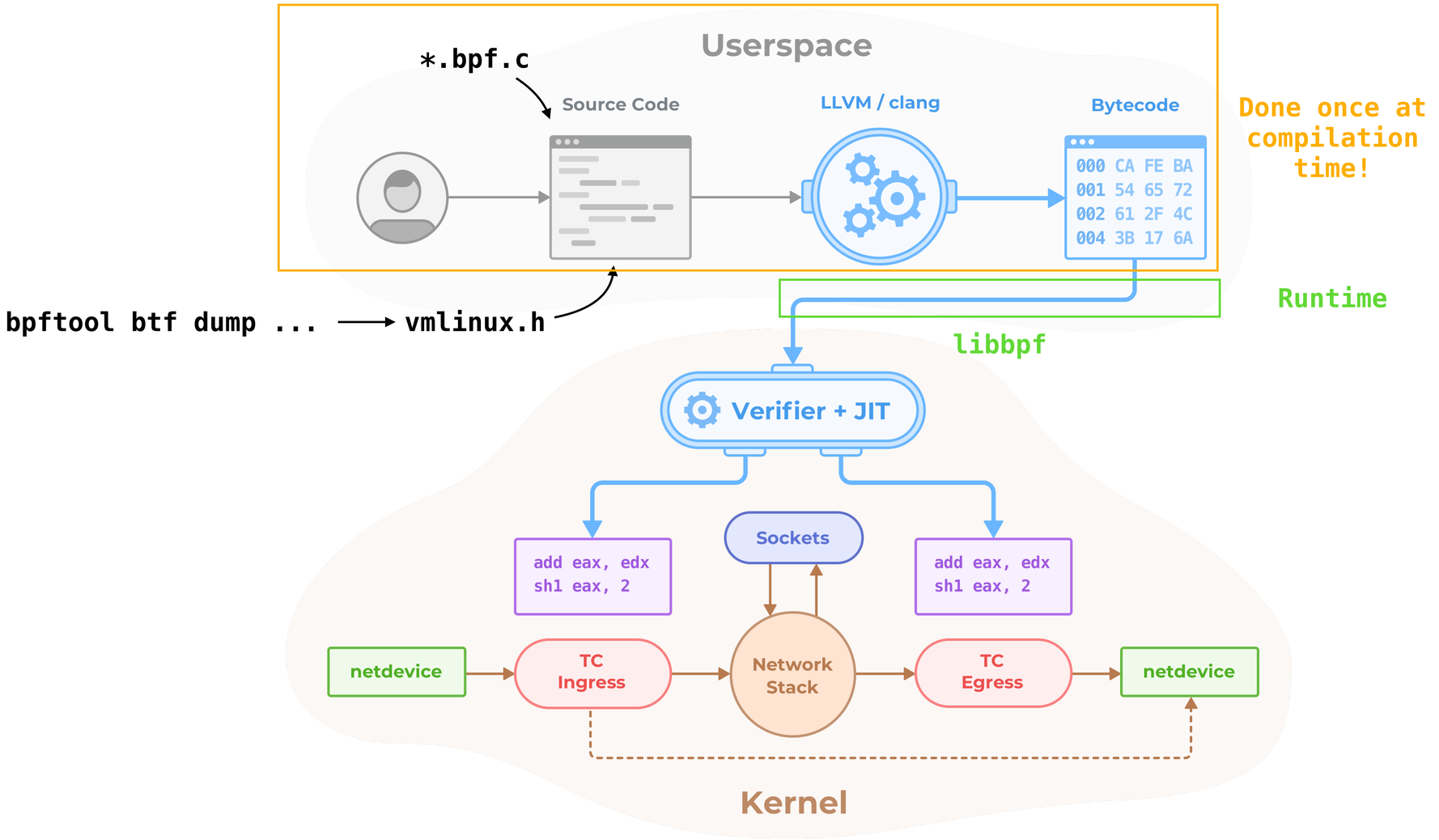
# Wait, is it BPF or eBPF?


- The **Berkeley Packet Filters** (BPF) **facility** belongs to the **Linux** kernel.
- The initial implementation (classic) **cBPF** was **superseded** by (extended) **eBPF**.
- **eBPF** comes with **more**, **wider registers**, different **flow control**, plus:
  - It is **designed** to be **JITed**, which...
  - ... allows **Clang/LLVM** to **generate** optimised **eBPF** code!
- Nowadays **eBPF** and **BPF** are usually **interchangeable**.
- Be sure to read the **kernel documentation**!

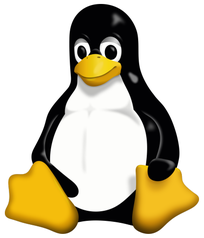
# What was flowd's approach to eBPF?

- **Flowd** leveraged the **BPF Compiler Collection** (BCC).
- Put simply, **BCC** hinges on **compiling** the **BPF** program at **runtime**:
  - It **imposes Clang/LLVM** as a **dependency** on the end user.
  - The only **available interfaces** are **Python** and **Lua**.
  - Several **C types abstract** away the actual **kernel definitions**...
- BCC's **Python interface** has been **deprecated**: they're **migrating** to **libbpf**!
- **BCC**-based solutions are **still** in **production** at **Netflix** and co.!

# What does it look like in the end?



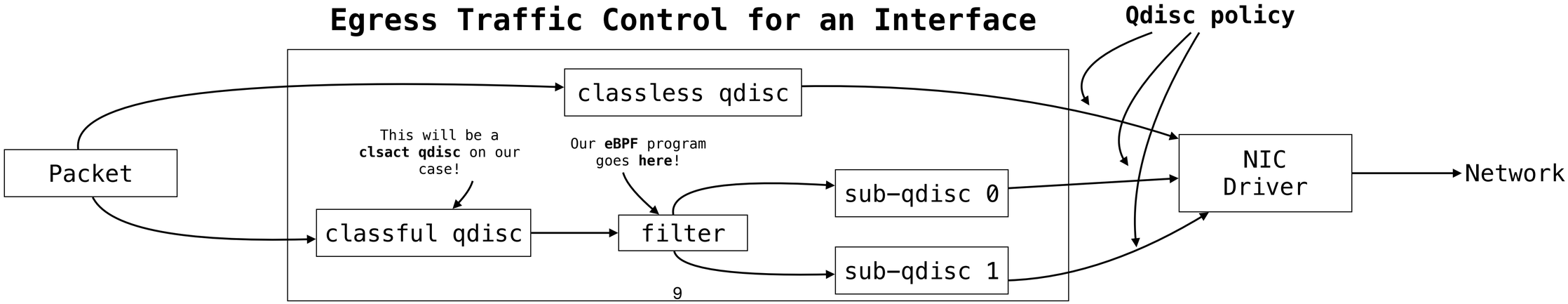
# A 'deeper' look at TC



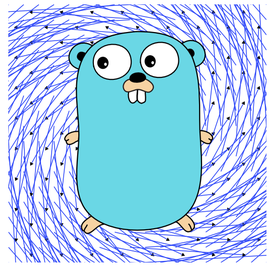
- **Traffic control** (TC) is **complex** and is controlled **hierarchically**:
  - Each **NIC** has a queue discipline (**qdisc**) associated to it.
  - The **kernel** selects **packets** from these **queues** to **send** them.
  - **Qdisc policies** determine whether a **packet** can **egress** (i.e. be chosen).
  - Some **qdiscs** are **classful** and can contain **sub-qdiscs**.
    - In that case, a **filter** '**decides**' which **sub-qdisc** a packet is assigned to.
    - The **filter** can also **apply** an **action** determining the 'fate' of a packet.
  - Starting on **kernel 4.1** BPF programs can leverage a **direct-action method**.

This site is well worth a read for more info on the topic!

**No** need for **sub-qdiscs**! We get **rid** of a ton of **complexity**!

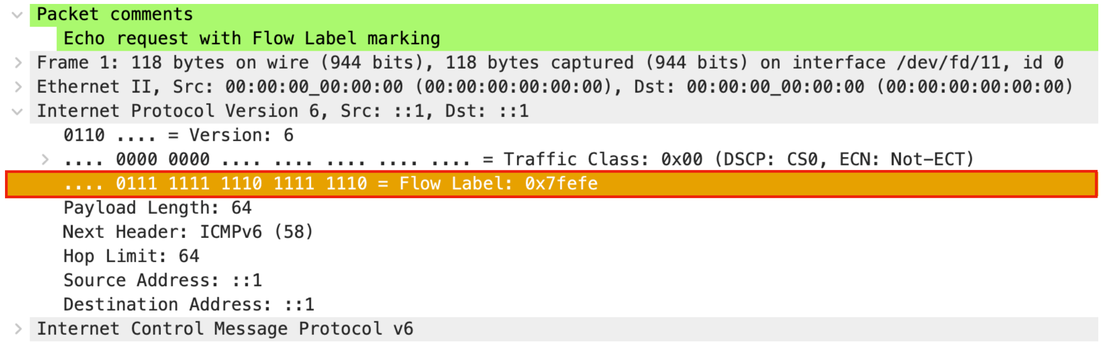


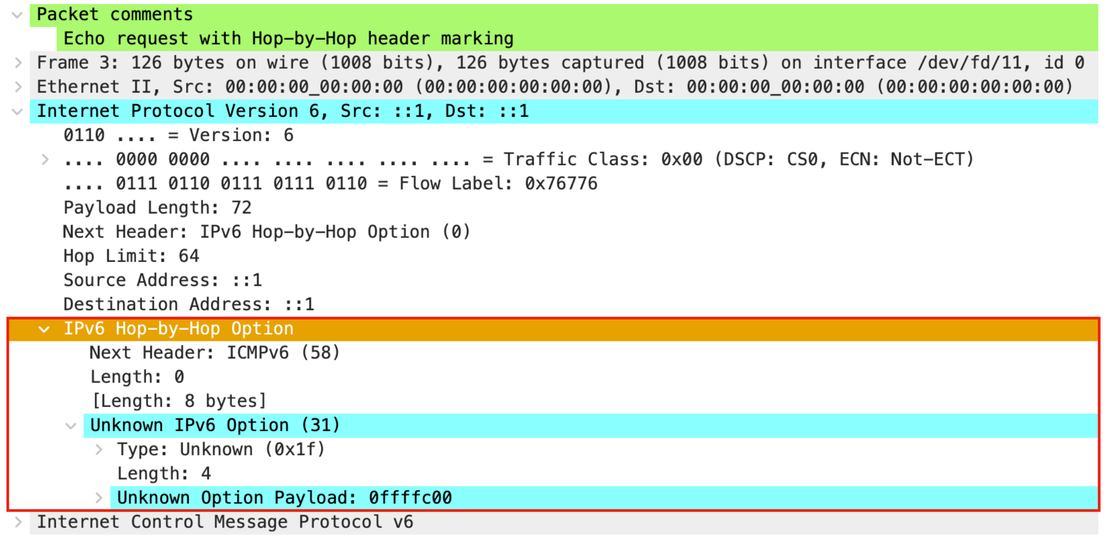
# What's working now?

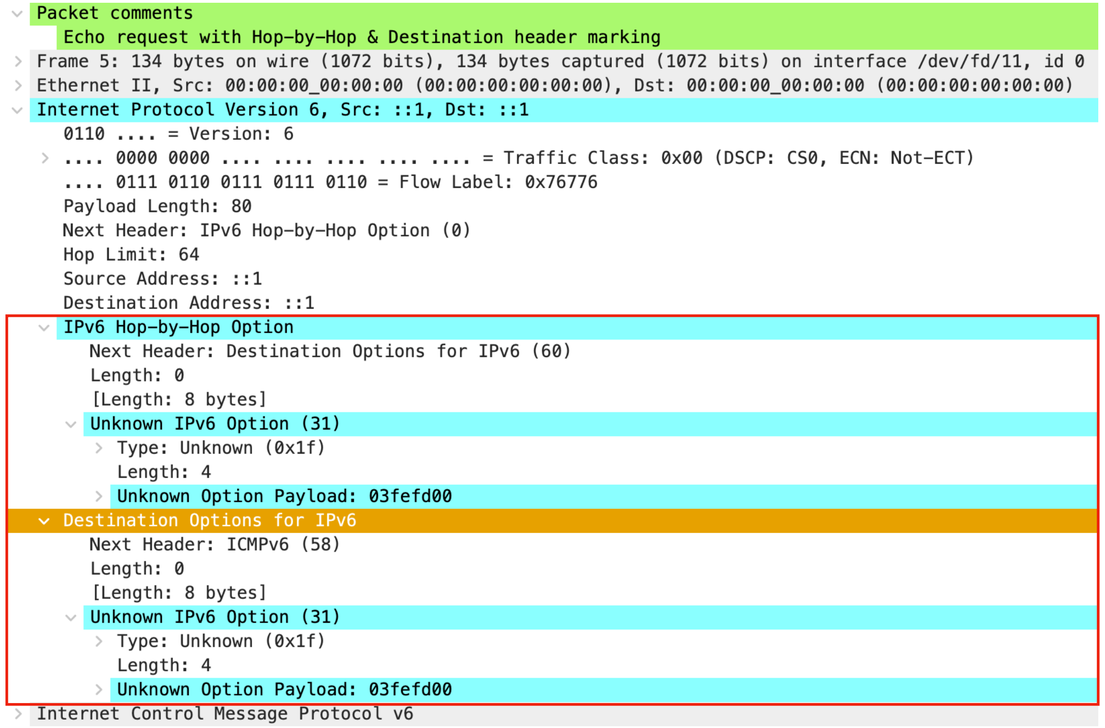


- **Plugins**:
  - New **REST API** plugin based on the performant **Echo framework**.
  - **Kernel** data **gathering** through native **sock_diag(7)** and **netlink(7)** calls.
  - **Named Pipes** based on **mkfifo(3)**.
- **Backends**:
  - **Flow-label**, **Hop-by-Hop** and **Destination Options** IPv6 marking.
  - **UDP fireflies**.
- Tested on **AlmaLinux 9.4** and **Fedora 35** (kernels **5.14.0** and **6.11.3**).

# Wireshark doesn't lie!



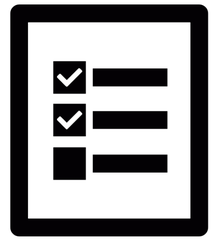
# Project structure and integrations

- A **single**, large (yet simple) **Makefile** handles everything, including:
  - Project **compilation** for several **targets**, including **eBPF** programs.
  - **eBPF program** is **macro-controlled**: marking strategy, debugging...
  - **RPM packaging**: check the **RPM Packaging Guide**!
- User documentation: **markdown manpages** (parsed with pandoc).
- We provide a **Docker** image for **CI/CD** and local **development**.
- **SystemD integration** through a simple unit file.
- **JSON-based configuration** with sane defaults.

# How are we working?


- Development is carried out on **scitags/flowd-go**.
- **TODOs** are translated into **issues** and **branches** are created off of them:
  - Sometimes **quick fixes** are pushed on **main** ¯_(ツ)_/¯.
  - Feature **branches** are usually **rebased**, but **merging** is **okay** too!
- At some point **main** gets **tagged**: that's a **release**!
- Release **assets** include **RPMs** and **binaries** for different architectures.
- **CI/CD** pipelines are **almost there**!
- Any **suggestions** for the **workflow** are **welcome**!

# TODOs



- Make sure we're **1:1 compatible** with **flowd**.
  - How can we **automate** these **checks** in the **future**?
- Apply **CO:RE** primitives in the **eBPF** program.
- **Bundle libbpf statically** as **explained by libbpfgo**.
- Add **unit tests** (and a **testing strategy** while we are at it).
- **Compare performance** with **flowd**'s.
- **Help appreciated**: **Go** is very **easy** to pick up!

# Closing thoughts

- **Flowd-go** offers a new **alternative** for **SciTags**.
- Key **functionality** is **already** in **place**.
- A **ton** of **stuff** left **to do** too!
- **Scalability assured** given **Go**'s builtin features.
- **Super glad** the **SciTags** initiative **found** this **useful**.

# Thank you for your time!

# Questions or suggestions?

# Backup

# On BPF_PROG_TYPE_SCHED_LS

- Our eBPF programs are of type **BPF_PROG_TYPE_SCHED_LS**:
  - Put simply, they **implement a TC filter**.
  - They receive an **__sk_buff** context:
    - Through it we **read** and **modify** anything from **L2** to **L5**!
  - They **leverage** the **direct action** mode: **no** need for **additional qdiscs**!
- Be sure to check the **eBPF docs**: they're a treasure trove!
- **Linux Advanced Routing & Traffic Control** (i.e. LARTC) is a must read too!
- Also, **Brendan Gregg's** and **Andrii Nakryiko's** blogs are **full** of **BPF** stuff!

# What is eBPF like from Go?

## Program Loading

```go
func (b *EbpfBackend) Init() error {

    // Try to load the module's object
    if err := b.module.BPFLoadObject(); err != nil {
        return fmt.Errorf("error loading the BPF object: %w", err)
    }

    // Create the TC Hook on which to place the eBPF program
    b.hook = b.module.TcHookInit()
    if err := b.hook.SetInterfaceByName(b.TargetInterface); err != nil {
        return fmt.Errorf("failed to set TC hook on interface %s: %w", b.TargetInterface, err)
    }

    // Place the hook in the packet egress chain
    b.hook.SetAttachPoint(bpf.BPFTcEgress)
    if err := b.hook.Create(); err != nil {
        if errno, ok := err.(syscall.Errno); ok && errno != syscall.EEXIST {
            slog.Debug("error creating the tc hook", "err", err)
        }
    }

    // Recover the specific program (i.e. function) we'll be attaching
    tcProg, err := b.module.GetProgram(PROG_NAME)
    if tcProg == nil || err != nil {
        return fmt.Errorf("couldn't find the target program %s: %w", PROG_NAME, err)
    }

    // Prepare the options for the hook
    var tcOpts bpf.TcOpts
    tcOpts.ProgFd = int(tcProg.FileDescriptor())
    tcOpts.Handle = 1
    tcOpts.Priority = 1

    // Attach the program!
    if err := b.hook.Attach(&tcOpts); err != nil {
        return fmt.Errorf("couldn't attach the ebpf program: %w", err)
    }

    // Get a reference to the map so that we're ready when running
    bpfMap, err := b.module.GetMap(MAP_NAME)
    if err != nil {
        return fmt.Errorf("error getting the ebpf map: %w", err)
    }
    b.flowMap = bpfMap
```

## Map Interactions

```go
switch flowID.State {
case glowd.START:
    flowTag := b.genFlowTag(flowID.Experiment, flowID.Activity)

    flowHashPtr := unsafe.Pointer(&flowHash)
    flowTagPtr := unsafe.Pointer(&flowTag)
    if err := b.flowMap.Update(flowHashPtr, flowTagPtr); err != nil {
        slog.Error("error inserting map value", "err", err, "flowHash", flowHash, "flowTag", flowTag)
        continue
    }
    slog.Debug("inserted map value", "flowHash", flowHash, "flowTag", flowTag)
case glowd.END:
    flowHashPtr := unsafe.Pointer(&flowHash)
    if err := b.flowMap.DeleteKey(flowHashPtr); err != nil {
        slog.Error("error deleting map key", "err", err, "flowHash", flowHash)
        continue
    }
    slog.Debug("deleted map value", "flowHash", flowHash)
default:
    slog.Error("wrong flow state made it here", "flowID.State", flowID.State)
}
```

## Program and Hook Removal

```go
// This explicit checks aren't really needed; it's simply left here for emphasis!
if b.hook != nil {
    // Detach the program. Note ProgFd and ProgId must be set to 0 or the detachment
    // won't work...
    localTcOpts := b.tcOpts
    localTcOpts.ProgFd = 0
    localTcOpts.ProgId = 0

    slog.Debug("detaching the tc hook")
    if err := b.hook.Detach(&localTcOpts); err != nil {
        slog.Error("error detaching the ebpf hook", "err", err)
    }

    if b.RemoveQdisc {
        slog.Debug("removing the backing qdisc")
        // Explicitly ask for the backing QDisc to be destroyed.
        b.hook.SetAttachPoint(bpf.BPFTcEgress | bpf.BPFTcIngress)
    }
    slog.Debug("destroying the tc hook")
    if err := b.hook.Destroy(); err != nil {
        slog.Warn("error destroying the hook", "err", err)
    }
}
```

# How does compilation look?

## eBPF Programs

```
define targetTemplate
marker-$(1).bpf.o: *.bpf.c *.bpf.h vmlinux.h
	$(CC) $(CFLAGS) -D $(2)  marker.bpf.c -o $$@

marker-$(1)-dbg.bpf.o: *.bpf.c *.bpf.h vmlinux.h
	$(CC) $(CFLAGS) -D $(2) -D GLOWD_DEBUG marker.bpf.c -o $$@

endef

# Just check it's working. Simply uncomment the line below to get a glimose into what
# the actual dynamic targets are!
# $(foreach i,$(INDICES),$(info $(call targetTemplate,$(word $(i),$(PROG_NAMES)),$(word $(i),$(PROG_DEFS)))))

# Simply make target all depend on all the programs
all: $(foreach progName,$(PROG_NAMES),marker-$(progName).bpf.o marker-$(progName)-dbg.bpf.o)

# Generate the dynamic targets. This must be done AFTER defining the all target as
# othwerwise the default target becomes the first dynamic target. The idea for getting
# this to work is to iterate over the INDICES variable and simply index the PROG_NAMES
# and PROG_DEFS thanks to (abusing) the $(word ...) function.
# Check:
#   https://www.gnu.org/software/make/manual/make.html#Foreach-Function
#   https://www.gnu.org/software/make/manual/make.html#Text-Functions
$(foreach i,$(INDICES),$(eval $(call targetTemplate,$(word $(i),$(PROG_NAMES)),$(word $(i),$(PROG_DEFS)))))

# Generate the kernel headers
vmlinux.h:
	bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

## Flowd-go

```
# Name of the output binary
BIN_NAME = flowd-go

# Path to wher the main package lives
MAIN_PACKAGE = ./cmd

# Source files to keep an eye out for for knowing when to rebuild the binary.
# This is not really honoured at the moment, but we'll look into that at some
# point...
SOURCES = $(wildcard *.go)

# What to remove when cleaning up
TRASH   = $(BIN_DIR)/* rpms/*.gz rpms/*.rpm

# Default compilation flags.
# The `-ldflags` option lets us define global variables at compile time!
# Check https://stackoverflow.com/questions/11354518/application-auto-build-versioning
CFLAGS := -tags ebpf -ldflags "-X main.builtCommit=$(COMMIT) -X main.baseVersion=$(VERSION)"

# Path to the eBPF sources need to build flowd-go. Make will be invoked
# recursively there.
EBPF_PROGS_PATH := backends/ebpf/progs

# Simply build flowd-go
build: $(SOURCES) ebpf-progs
	@mkdir -p bin
	$(ENV_FLAGS) $(GOC) build $(CFLAGS) -o $(BIN_DIR)/$(BIN_NAME) $(MAIN_PACKAGE)

# We'll only compile the eBPF progra if we're on Linux
ifeq ($(OS),Linux)
# Recursively build the eBPF program. Be sure to check
# https://www.gnu.org/software/make/manual/html_node/Recursion.html
ebpf-progs:
	$(MAKE) -C $(EBPF_PROGS_PATH)
else
# Just provide a stub it we're not on Linux!
ebpf-progs:
endif

# Include Makefiles with additional targets automating stuff.
# Check https://www.gnu.org/software/make/manual/html_node/Include.html
include mk/*.mk
```