Enabling BPF runtime policies for better BPF management

Raj Sahu  Dan Williams

SIGCOMM eBPF, NYC
10th Sept, 2023
● BPF management is getting complicated
  ○ load privileges, monitoring BPF programs, access privileges ….

● BPF-orchestrators now exist to provide access control and lifecycle management of BPF programs across clusters.

● Load Policies: hooks, pods, signature validation

● Access Policies: map R/W
However, Operator is unaware about performance impact of loaded BPF programs on the overall system.
However, Operator is unaware about performance impact of loaded BPF programs on the overall system.

- 1 High-latency program at critical hook point, or,
- Several programs in frequently used call graph

Missing SLAs
Therefore,

Runtime estimation of BPF programs is a critical requirement
Therefore,

Runtime estimation of BPF programs is a critical requirement !!!
Outline

- Motivation
- Idea
- Challenges
- Runtime Estimator
- Evaluation
- Discussion
The Idea

- BPF-verifier emits a runtime estimation as range \([\text{best case} - \text{worst case}]\) time
The Idea

- BPF-verifier emits a runtime estimation as range [best case - worst case] time
The Idea

- BPF-verifier emits a runtime estimation as range \(\text{best case} - \text{worst case}\) time.
- Estimates are checked against admin-provided Runtime Policies (latency/hook, latency/call graph).
The Idea

- BPF-verifier emits a runtime estimation as range \([\text{best case} - \text{worst case}]\) time
- Estimates are checked against admin-provided Runtime Policies (latency/hook, latency/call graph)
- Only allowed programs will get attached.
/Outline

- Motivation
- Idea
- Challenges
- Runtime Estimator
- Evaluation
- Discussion
/Challenges

C#1: Helper functions are opaque
- BPF verifier cannot traverse through helpers
- Complex internal logic is abstracted away from a BPF developer
Challenges

C#2: Multiple program paths
- Dynamic profilers don’t guarantee completeness
- Rare but costly branches can give unexpected worst-case runtime
./Challenges

C#3 : Helper induced control-flow changes
- Loops, iterators
./Challenges

C#1 : Helper functions are opaque

C#2 : Multiple program paths

C#3 : Helper induced control-flow changes
Challenges / Key Insights

C#1: Helper functions are opaque
   Key Insight ⇒ Perform dynamic measurements

C#2: Multiple program paths
   Key Insight ⇒ Utilize verifier’s in-kernel static analysis

C#3: Helper induced control-flow changes
   Key Insight ⇒ Teach verifier about special cases
Outline

- Motivation
- Idea
- Challenges
- Runtime Estimator
- Evaluation
- Discussion
Offline measurement of helper functions

C#1: Helper functions are opaque
  Key Insight: Perform dynamic measurements

./The Runtime Estimator / Helper Timer
Offline measurement of helper functions

samples/bpf
~30 helpers
10 runs x 1000 iterations
bpf_ktime_get_ns()
Offline measurement of helper functions

samples/bpf
~30 helpers
10 runs x 1000 iterations
bpf_ktime_get_ns()

Fig: Best - Worst case helper runtimes (in ns)
The Runtime Estimator / Branch Timer

C#2: Multiple program paths
Key Insight: Utilize verifier’s in-kernel static analysis

Helper estimates + static analysis
For each branch:

- BPF verifier state tracks total cost.
- Helper call adds pre-calculated cost to the current branch.

When all branches get exhausted, overall best and worst runtime is reported.
Adjust runtime estimates for control flow changes by helpers

For bpf_loop(iter, callback_fn):

- Calculate estimates for the callback function (static)
- Read last known value of r₁ register
- Increment cost by estimate * val(r₁)
The Runtime Estimator / Example Run

Verify whether the 3 sub-components are correctly working:

```c
main():
    key = rand() % 10000
    if key>1:
        bpf_printf(key)
    else:
        bpf_loop(1000, func)

func():
    bpf_loop(100, simple)

simple():
    bpf_printf("Hello")
```
main():
    key = rand() % 10000
    if key>1:
        bpf_printk(key)
    else:
        bpf_loop(1000, func)

func():
    bpf_loop(100, simple)

simple():
    bpf_printk("Hello")

Verify whether the 3 sub-components are correctly working:
1. Identifying rare branches
./The Runtime Estimator / Example Run

Verify whether the 3 sub-components are correctly working:
1. Identifying rare branches
2. Detect helper calls and factor-in their cost

```
main():
    key = rand() % 10000
    if key>1:
        bpf_printk(key)
    else:
        bpf_loop(1000, func)

simple():
    bpf_printk("Hello")

func():
    bpf_loop(100, simple)
```
Verify whether the 3 sub-components are correctly working:
1. Identifying rare branches
2. Detect helper calls and factor-in their cost
3. Considering special cases
The Runtime Estimator / Example Run

```c
main() {
    key = rand() % 10000
    if key>1:
        bpf_printk(key)
    else:
        bpf_loop(1000, func)
}

func() {
    bpf_loop(100, simple)
}

simple() {
    bpf_printk("Hello")
}
```

Runtime Estimator

[115 - 180,000,510] ns
main():
    key = rand() % 10000
    if key>1:
        bpf_printk(key)
    else:
        bpf_loop(1000, func)

func():
    bpf_loop(100, simple)

simple():
    bpf_printk("Hello")

./The Runtime Estimator / Example Run

Runtime Estimator ➔ [115 - 180,000,510] ns
Actual Runtime ➔ 125,013,362 ns
/Outline

- Motivation
- Idea
- Challenges
- Runtime Estimator
- Evaluation
- Discussion
./Evaluation

Validating runtime estimator on sample BPF programs

Linux Kernel 5.15
sysctl kernel.bpf_stats_enabled
./Evaluation
Evaluation

- Best < Actual << Worst
./Evaluation

- Best < Actual << Worst

Harder to make runtime policies
/.Outline

- Motivation
- Idea
- Challenges
- Runtime Estimator
- Evaluation
- Discussion
Helper runtime variability:

- **Argument dependent**
  - Length of string for printk, depth of stack for get_stackid, etc.
  - Are the parameters known at verification time?

- **Resource contention**
  - BPF map based helpers use locks for concurrency-safe R/W
    - Local CPU LRU lock, LRU lock, hashtab lock, remote CPU LRU lock
  - With more concurrent access, each R/W costs higher (~4x increase for 2 CPUs)

---

Some ideas

- Port existing work of performance estimation in NFs\(^{[1,2]}\) to Linux kernel
  - Current dynamic analysis of helper faces completeness problem

- Contention-aware performance prediction in NFs\(^{[3]}\)
  - As only BPF program can access map, # of contending parties could be known at load time?

---

1. Runtime estimation of BPF programs is crucial for production servers.

2. Proposed Runtime Estimator: a hybrid approach to combine dynamic measurement of black-boxed helper functions with verifier’s static analysis of all possible branches.

3. The performance estimates were correct but challenges remain around making the estimates more accurate.
BACKUP SLIDES
/Motivation

- BPF management is getting complicated
  - load privileges, monitoring BPF programs, access privileges ….

- BPF-orchestrators now exist to provide access control and lifecycle management of BPF programs across clusters.
Offline measurement of helper functions

- samples/bpf
- ~30 helpers
- 10 runs x 1000 iterations
- `bpf_ktime_get_ns()`

**Fig**: Best - Worst - Average case helper runtimes (in ns)
For `bpf_loop(iter, callback_fn)`:

- Calculate estimates for the callback function (static)
- Read last known value of `r_1` register
- Increment cost by estimate * `val(r_1)`

```c
bpf_loop(4, function, NULL, 0);

0:  r1 = 0x4  
1:  r2 = 0x208  
3:  r3 = 0x0  
4:  r4 = 0x0  
5:  call 181  <------ bpf_loop()  
6:  r1 = r0  
7:  ....  
8:  ..
```