Enabling BPF runtime policies for better BPF management

Raj Sahu   Dan Williams

SIGCOMM eBPF, NYC
10th Sept, 2023

VIRGINIA TECH

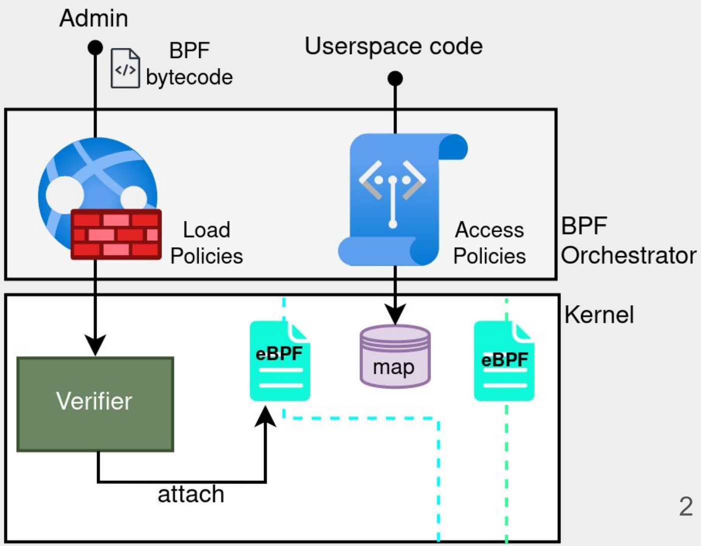
./Motivation

- 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.

- Load Policies : hooks, pods, signature validation

- Access Policies : map R/W



./Motivation (cont.)

However,

Operator is unaware about performance impact of loaded BPF programs on the overall system.

./Motivation (cont.)

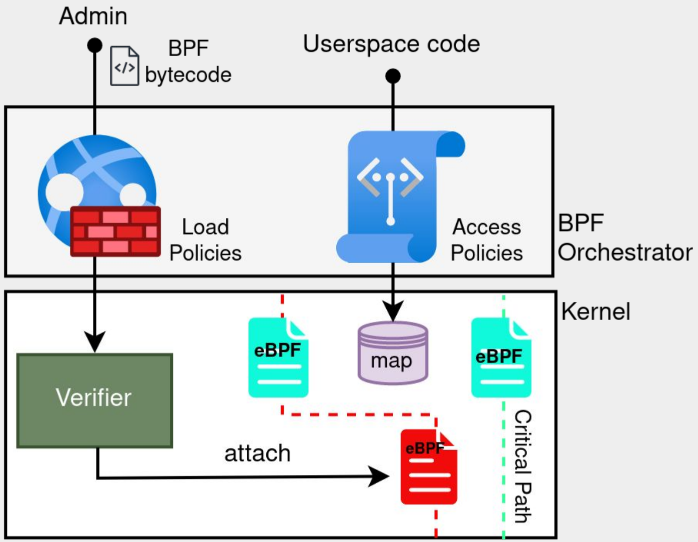
However,

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



./Motivation (cont.)

Therefore,

Runtime estimation of BPF programs is a critical requirement

./Motivation (cont.)

Therefore,

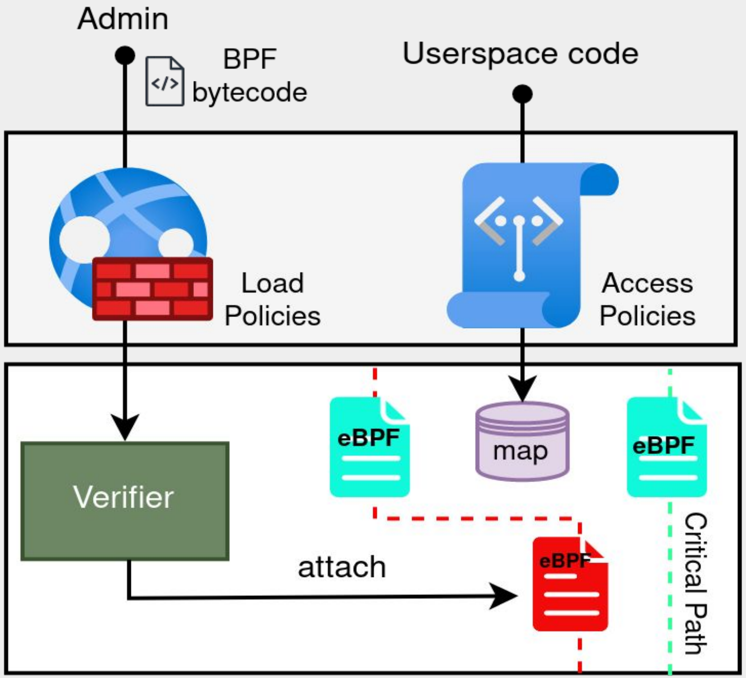
Runtime estimation of BPF programs is a critical requirement **!!!**

./Outline

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

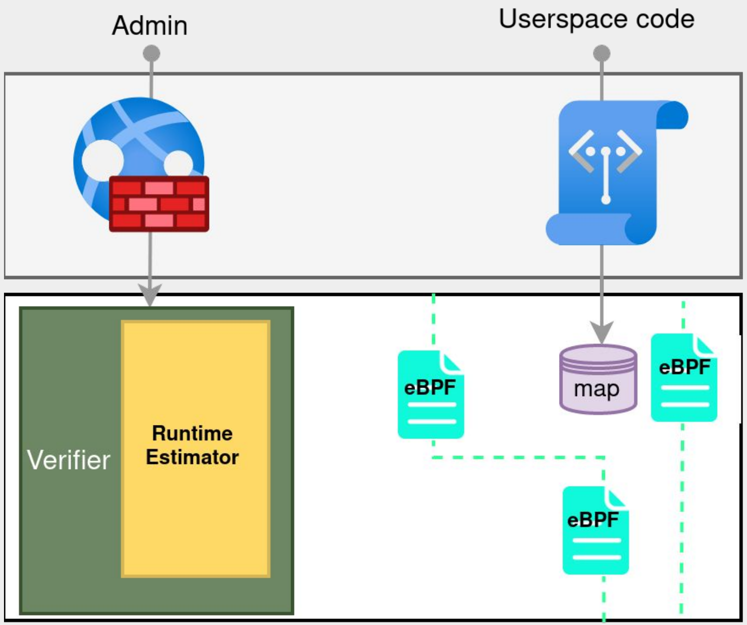
./The Idea

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



./The Idea

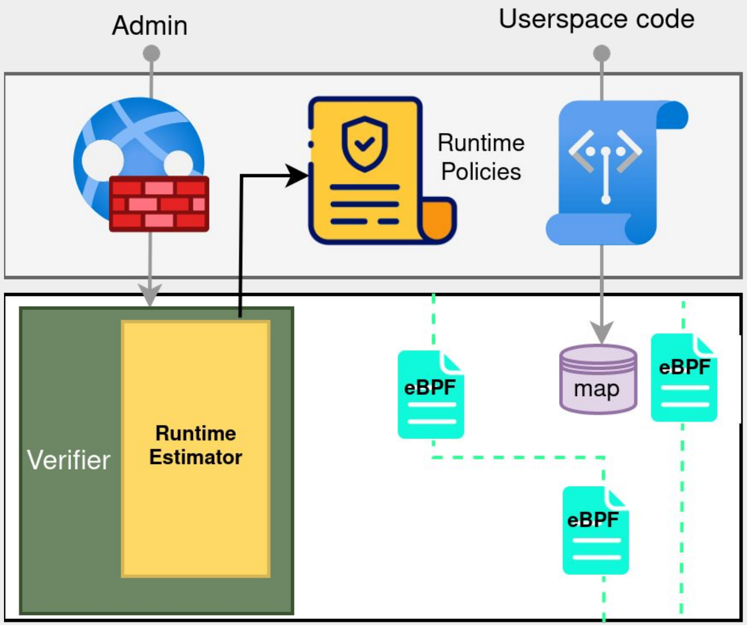
- BPF-verifier emits a runtime estimation as range *[best case - worst case]* time



./The Idea

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

- Estimates are checked against admin-provided Runtime Policies (latency/hook, latency/call graph)

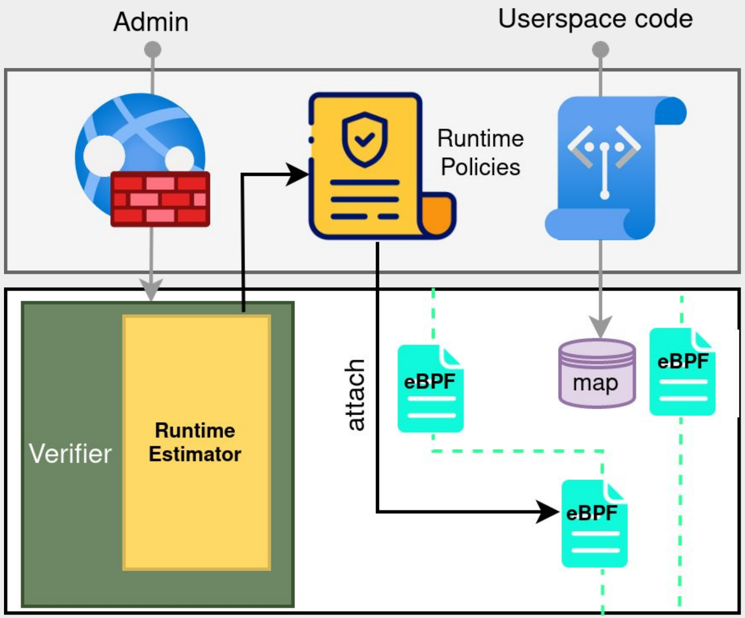


./The Idea

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

- Estimates are checked against admin-provided Runtime Policies (latency/hook, latency/call graph)

- Only allowed programs will get attached.



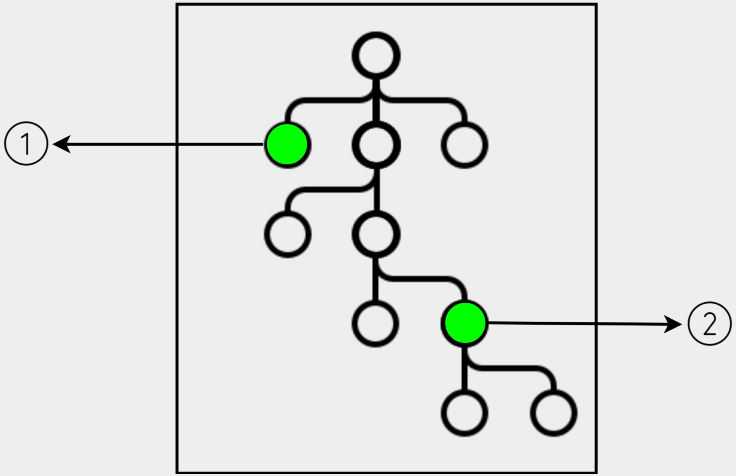
./Outline

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

./Challenges

C#1 : Helper functions are opaque

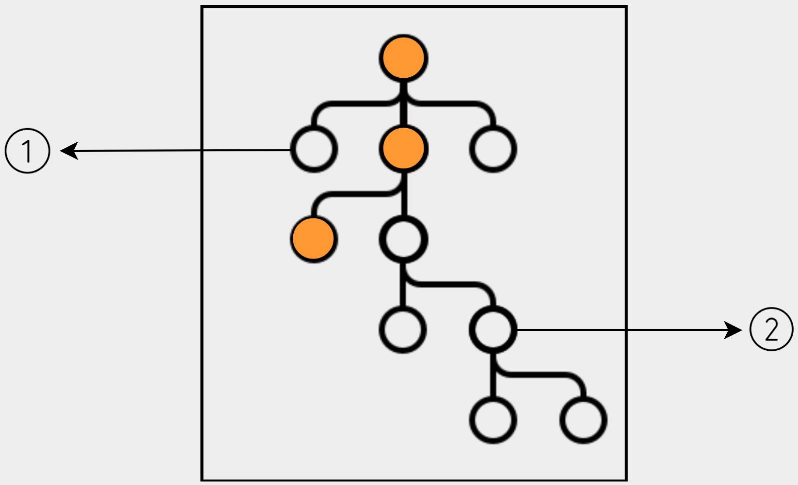
- BPF verifier cannot traverse through helpers
- Complex internal logic is abstracted away from a BPF developer



./Challenges

C#2 : Multiple program paths

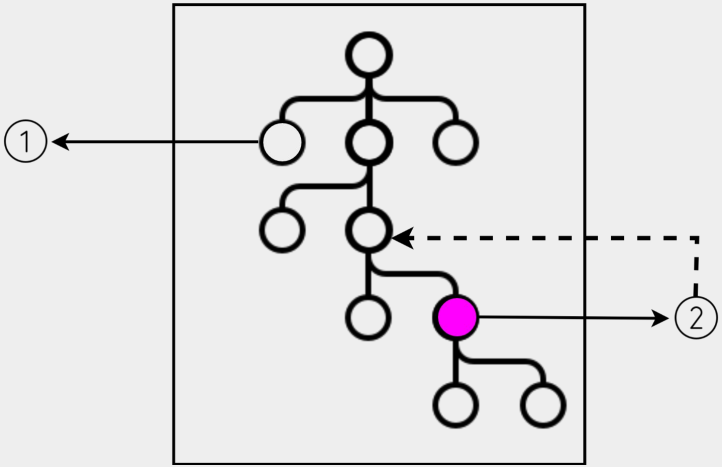
- Dynamic profilers don’t guarantee completeness
- Rare but costly branches can give unexpected worst-case runtime



./Challenges

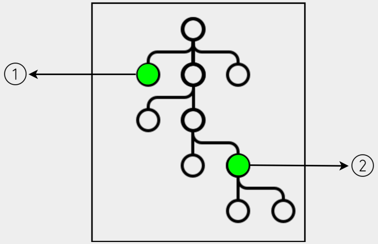
C#3 : Helper induced control-flow changes

- Loops, iterators

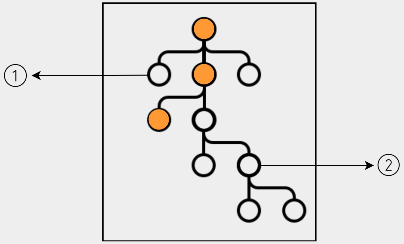


./Challenges

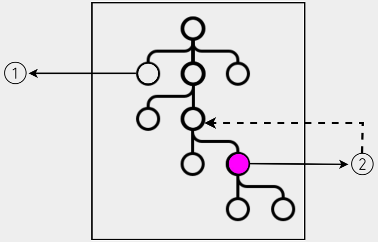
C#1 : Helper functions are opaque



C#2 : Multiple program paths



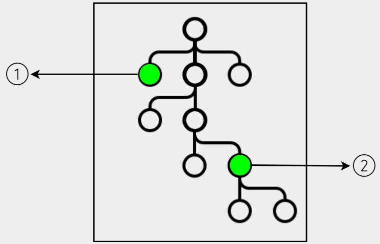
C#3 : Helper induced control-flow changes



./Challenges / Key Insights

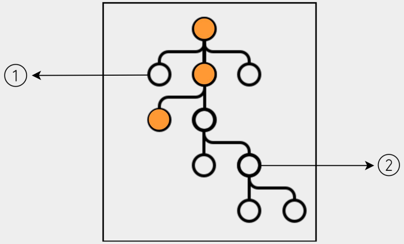
C#1 : Helper functions are opaque

Key Insight ⇒ Perform dynamic measurements



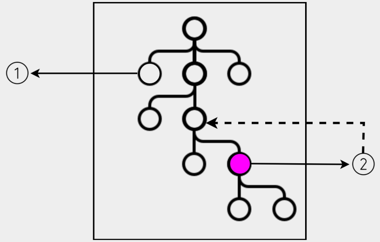
C#2 : Multiple program paths

Key Insight ⇒ Utilize verifier's in-kernel static analysis



C#3 : Helper induced control-flow changes

Key Insight ⇒ Teach verifier about special cases

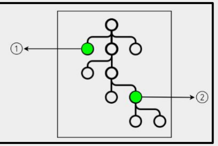


./Outline

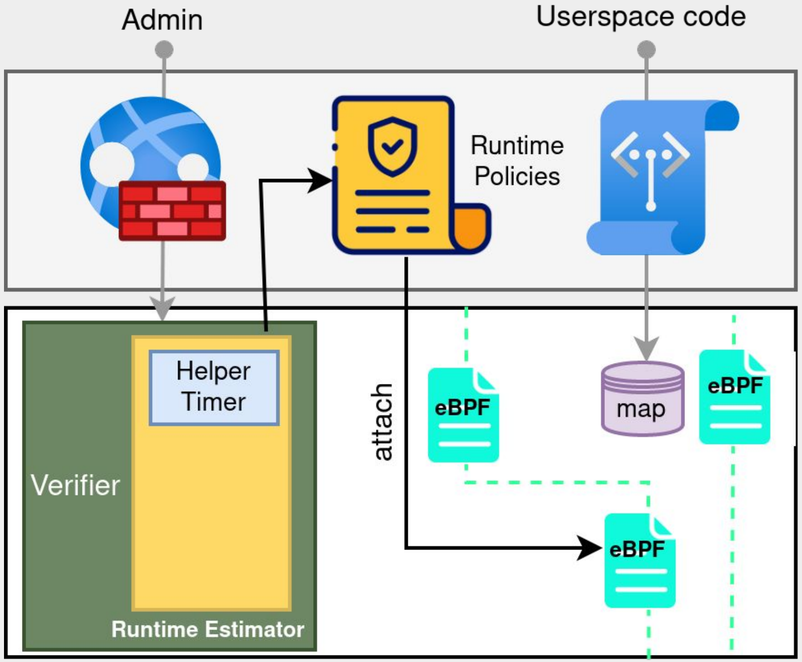
- Motivation
- Idea
- Challenges
- Runtime Estimator
- Evaluation
- Discussion

./The Runtime Estimator / Helper Timer

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



Offline measurement of helper functions



…/Helper Timer

Offline measurement of helper functions

⬇️

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

…/Helper Timer

Offline measurement of helper functions

⬇️

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

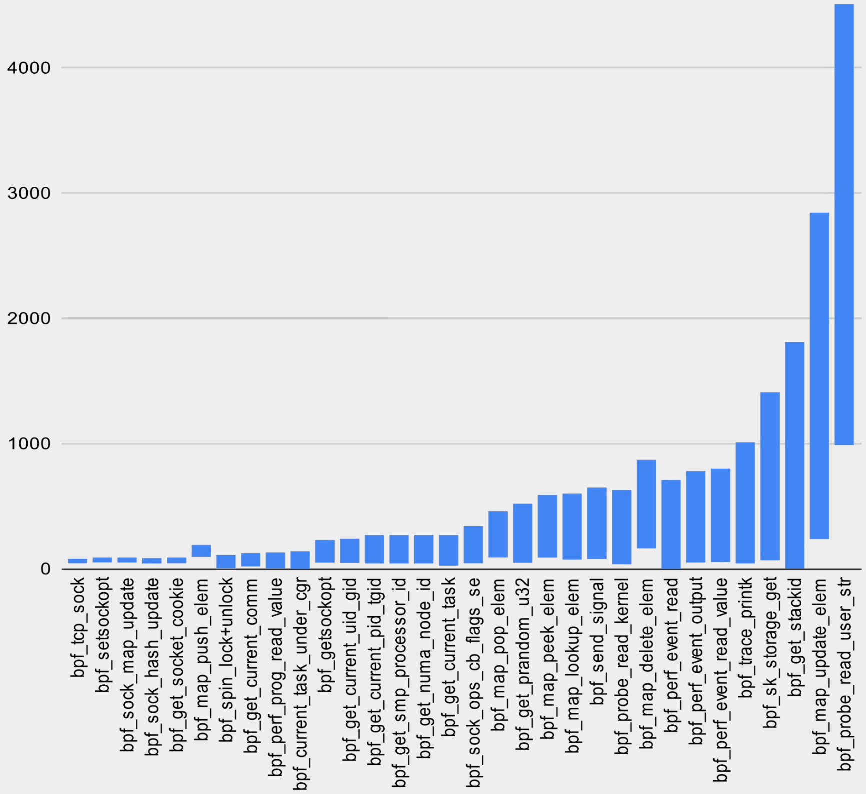
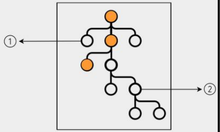


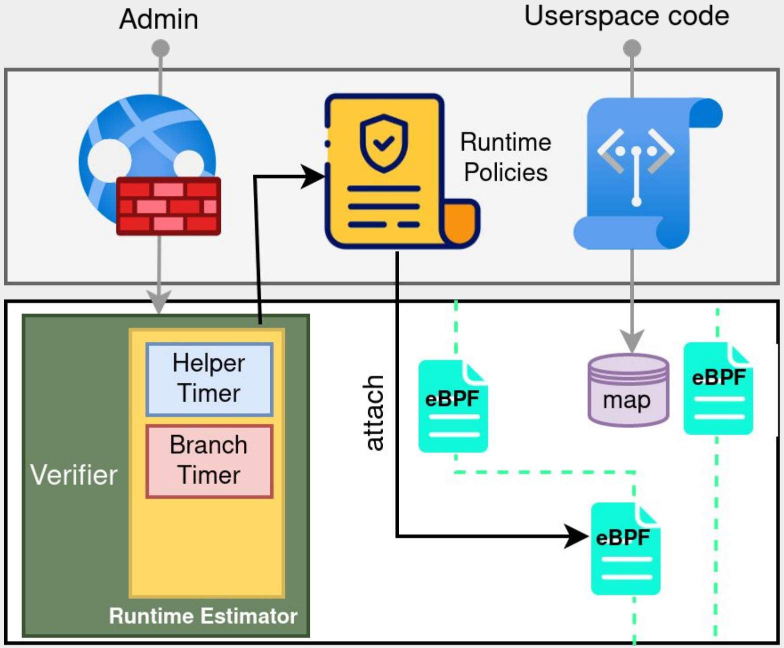
Fig : Best - Worst case helper runtimes (in ns)

./The Runtime Estimator / Branch Timer

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



Helper estimates + static analysis

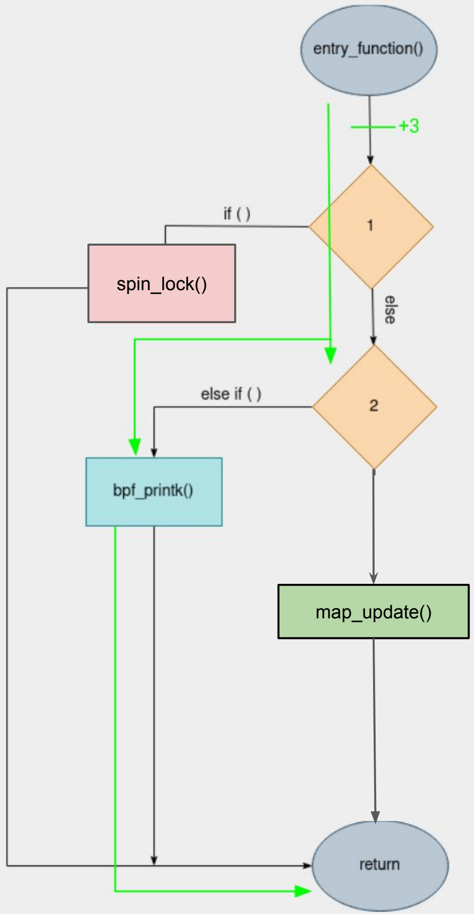


…/Branch Timer

For each branch :

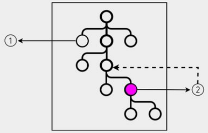
- BPF verifier state tracks total cost.
- Helper call adds pre-calculated cost to the current branch

When all branches get exhausted, overall best and worst runtime is reported.



./The Runtime Estimator / Special-case Handler

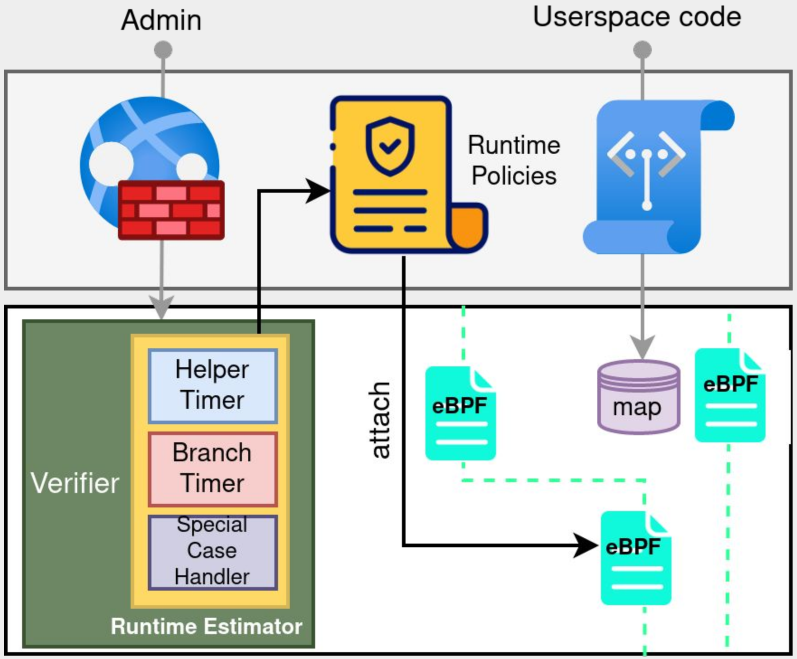
C#3 : Helper induced control-flow changes
Key Insight ⇒ Teach verifier about special cases



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 r_1 register
- Increment cost by estimate * val(r_1)



./The Runtime Estimator / Example Run

```
simple():
        bpf_printk("Hello")
```

```
func():
        bpf_loop(100, simple)
```

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

Verify whether the 3 sub-components are correctly working :

./The Runtime Estimator / Example Run

```
simple():
        bpf_printk("Hello")

func():
        bpf_loop(100, simple)

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

Verify whether the 3 sub-components are correctly working :

1. Identifying rare branches

./The Runtime Estimator / Example Run

```
simple():
        bpf_printk(“Hello”)
```

```
func():
        bpf_loop(100, simple)
```

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

Verify whether the 3 sub-components are correctly working :

1. Identifying rare branches
2. Detect helper calls and factor-in their cost

./The Runtime Estimator / Example Run

```
simple():
    bpf_printk("Hello")
```

```
func():
    bpf_loop(100, simple)
```

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

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

```
simple():
        bpf_printk("Hello")

func():
        bpf_loop(100, simple)

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

Runtime Estimator → [115 - 180,000,510] ns

./The Runtime Estimator / Example Run

```
simple():
        bpf_printk("Hello")
```

```
func():
        bpf_loop(100, simple)
```

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

Runtime Estimator → [115 - 180,000,510] ns

Actual Runtime → 125,013,362 ns

./Outline

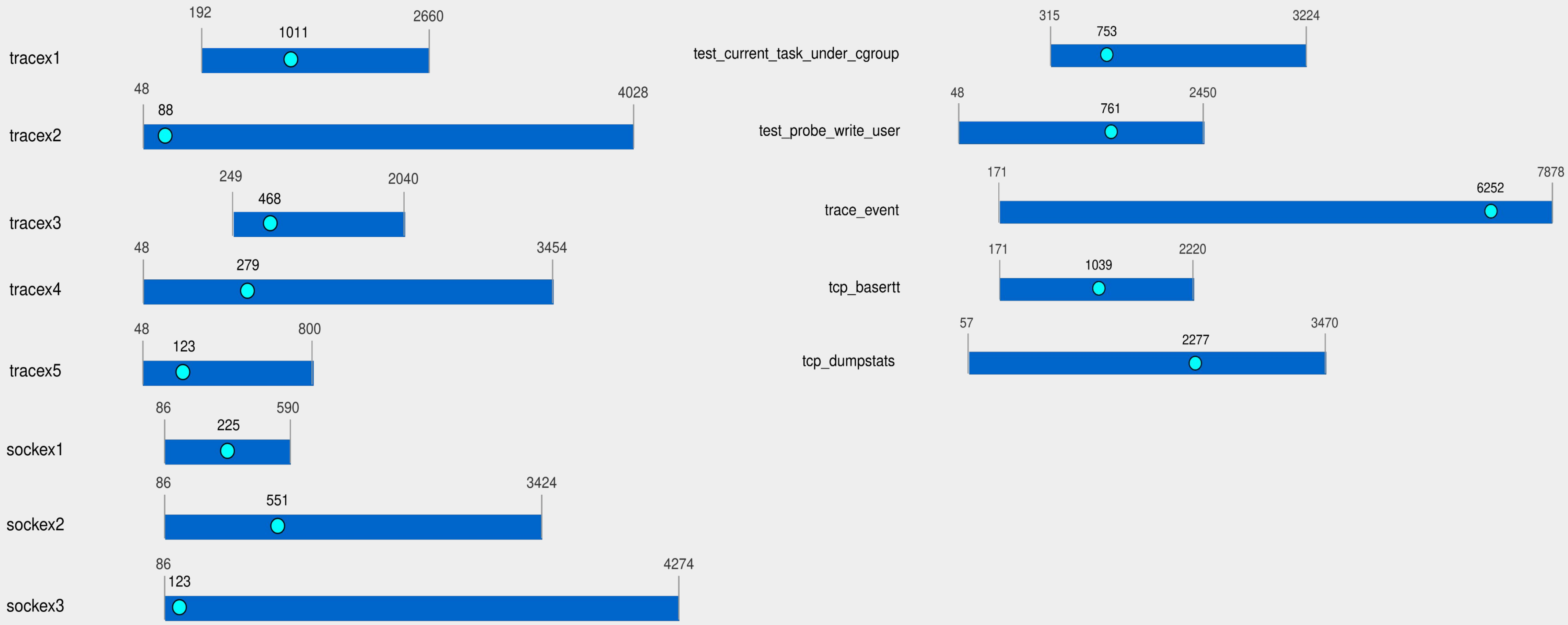
- Motivation
- Idea
- Challenges
- Runtime Estimator
- Evaluation
- Discussion

./Evaluation

Validating runtime estimator on sample BPF programs

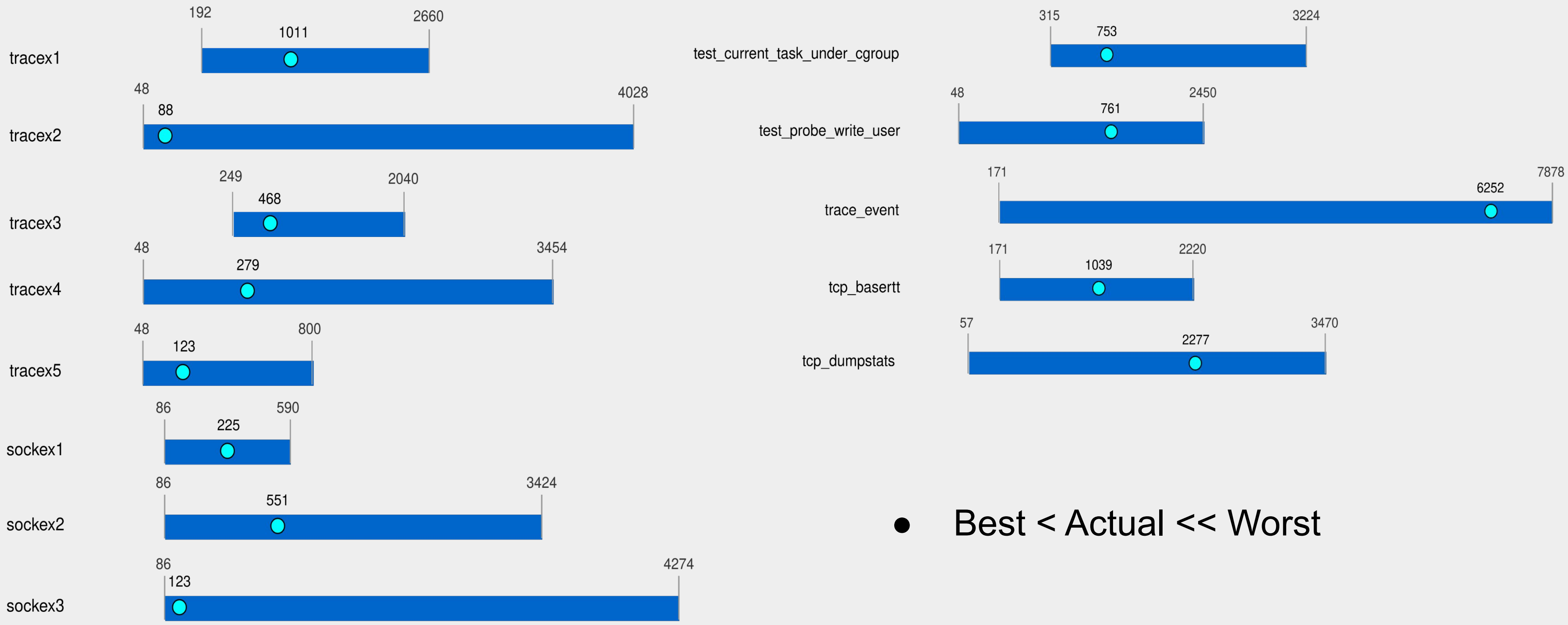
Linux Kernel 5.15
sysctl kernel.bpf_stats_enabled

./Evaluation



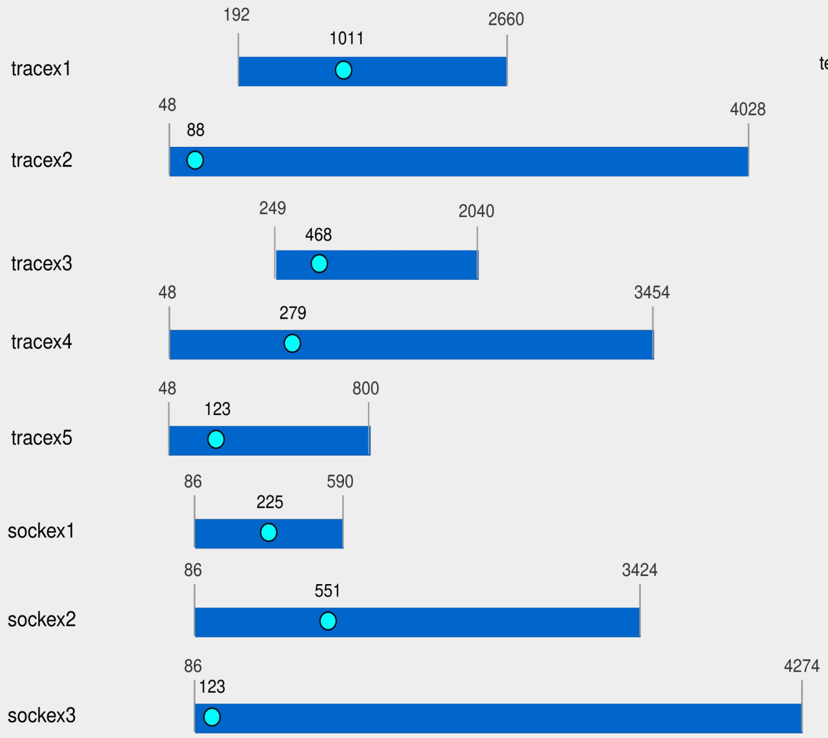
tracex1
192
1011
2660
tracex2
48
88
4028
tracex3
249
468
2040
tracex4
48
279
3454
tracex5
48
123
800
sockex1
86
225
590
sockex2
86
551
3424
sockex3
86
123
4274
test_current_task_under_cgroup
315
753
3224
test_probe_write_user
48
761
2450
trace_event
171
6252
7878
tcp_basertt
171
1039
2220
tcp_dumpstats
57
2277
3470

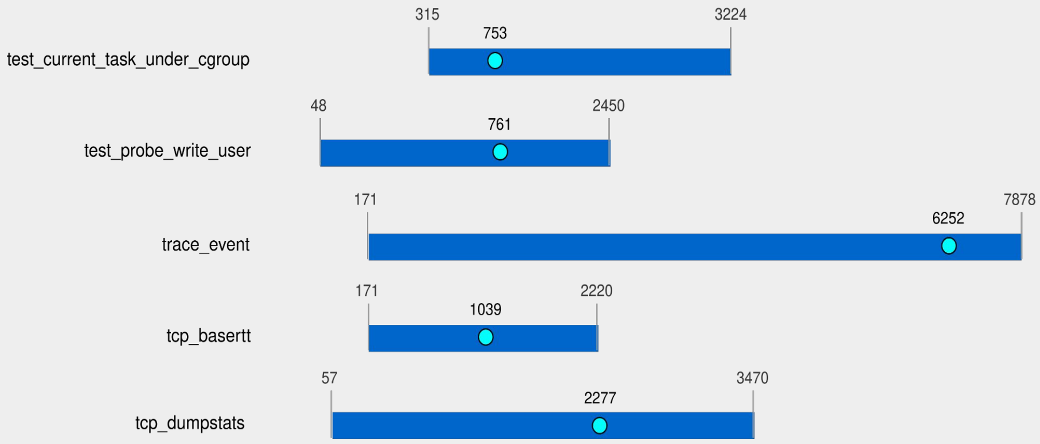
./Evaluation



- Best < Actual << Worst

./Evaluation





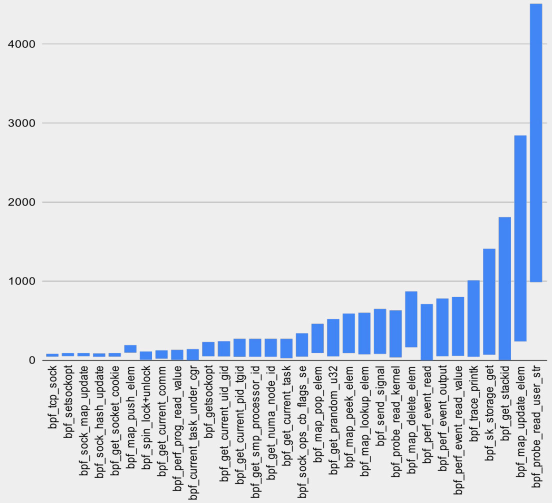
- Best < Actual << Worst

Harder to make runtime policies

./Outline

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

./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[1]

          ⇒ With more concurrent access, each R/W costs higher (~4x increase for 2 CPUs)

1. https://www.kernel.org/doc/html/next/bpf/map_hash.html

./Discussion

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. Iyer, Rishabh, et al. "Performance contracts for software network functions." 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019.
2. Iyer, Rishabh, Katerina Argyraki, and George Candea. "Performance interfaces for network functions." 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). 2022.
3. Manousis, Antonis, et al. "Contention-aware performance prediction for virtualized network functions." Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication. 2020.

./Summary

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.

THANK YOU

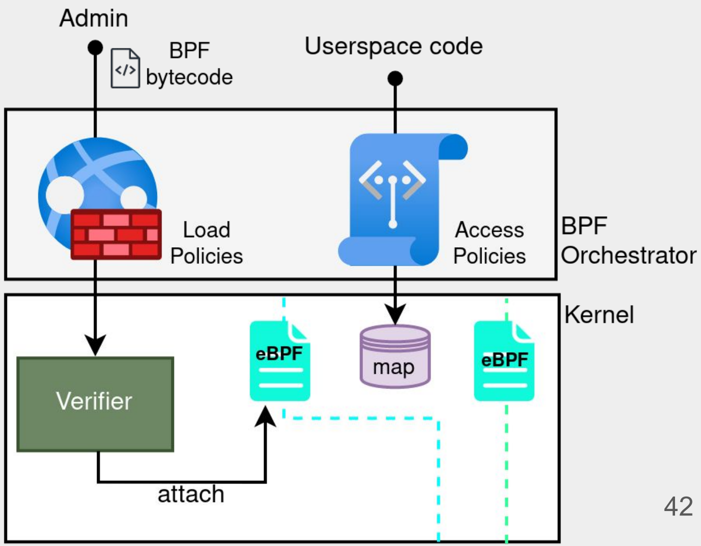
Raj Sahu
raj.sahu@vt.edu

Dan Williams
djwillia@vt.edu

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.



…/Helper Timer

Offline measurement of helper functions

⬇️

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

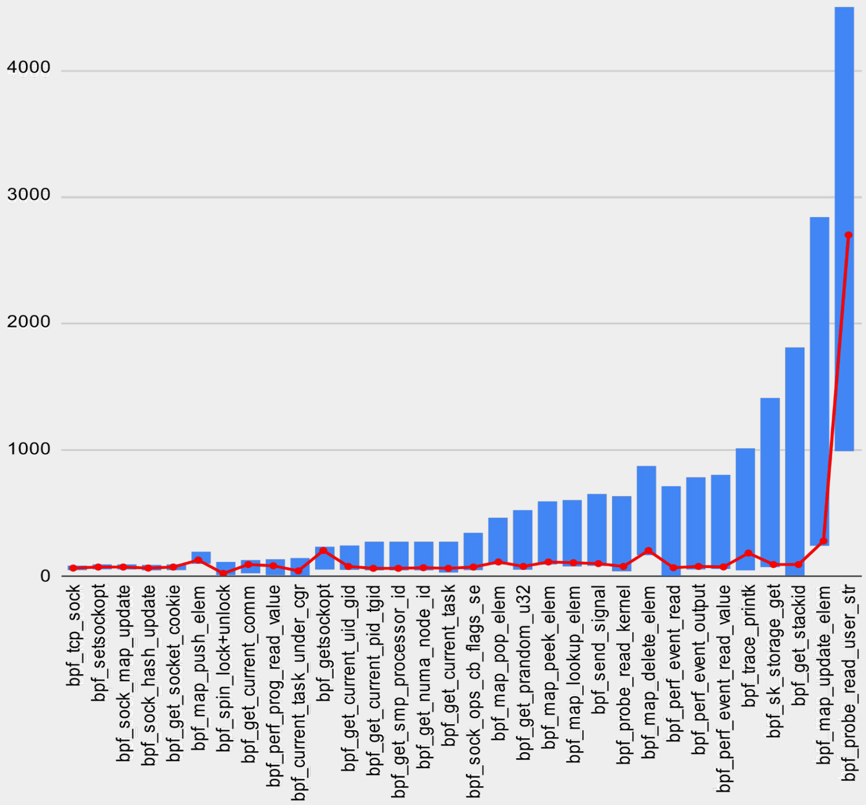


Fig : Best - Worst - Average case helper runtimes (in ns)

./The Runtime Estimator / Special-case Handler

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)

```
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:      ..
```