Carleton UNIVERSITY

Canada's Capital University

Host-Based Anomaly Detection with Extended BPF

William Findlay

Supervisor: Dr. Anil Somayaji

April 10th, 2020

Carleton UNIVERSITY

Canada's Capital University

Research Questions

- **Can we write IDS software in eBPF?**
  - Spoiler: Yes.
- **How does eBPF compare with kernel-based IDS implementations?**
- **How far can we take this?**
  - I have some thoughts on this (later)

Carleton UNIVERSITY
Canada's Capital University

Why eBPF?

- **Good performance***
  - Keep up with kernel-based implementation
- **Broad scope**
  - Trace userspace, kernelspace, hardware, sockets, packets (incl. **before** kernel networking stack!)
- **Low opportunity cost**
  - No custom kernel required
  - Forward compatibility
- **Production-safe**
  - No kernels were harmed in the making of this software

*we will revisit this soon!

What is eBPF?

Carleton UNIVERSITY
Canada's Capital University

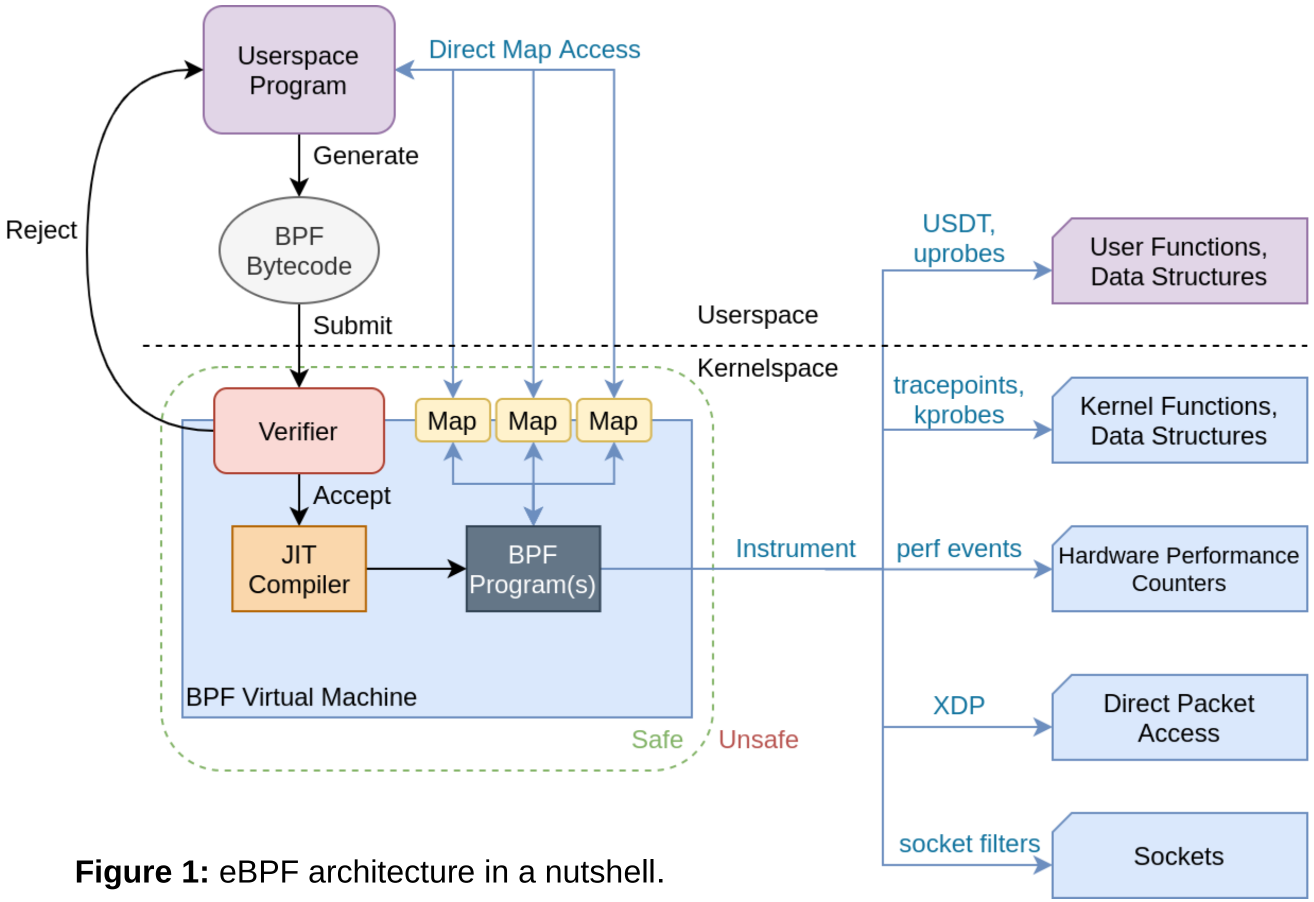


Figure 1: eBPF architecture in a nutshell.

What is eBPF?

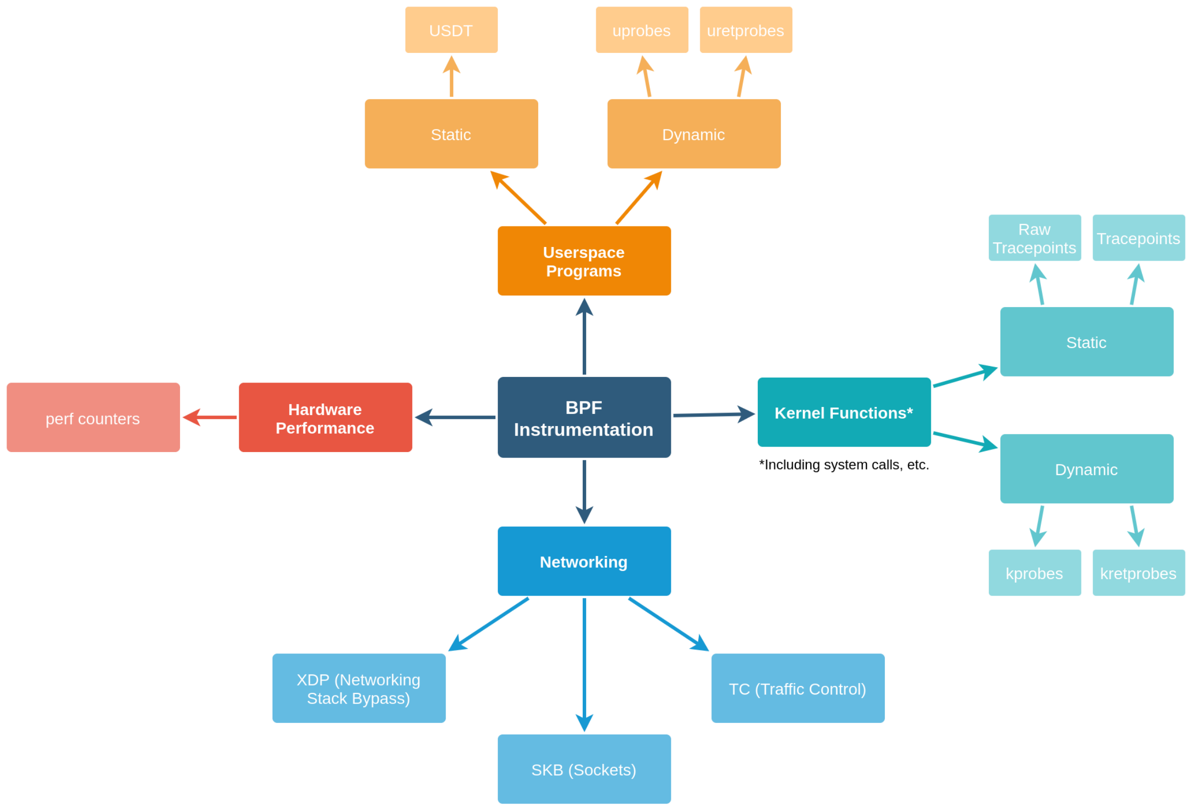


Figure 2: BPF program types and use cases.

What is eBPF?

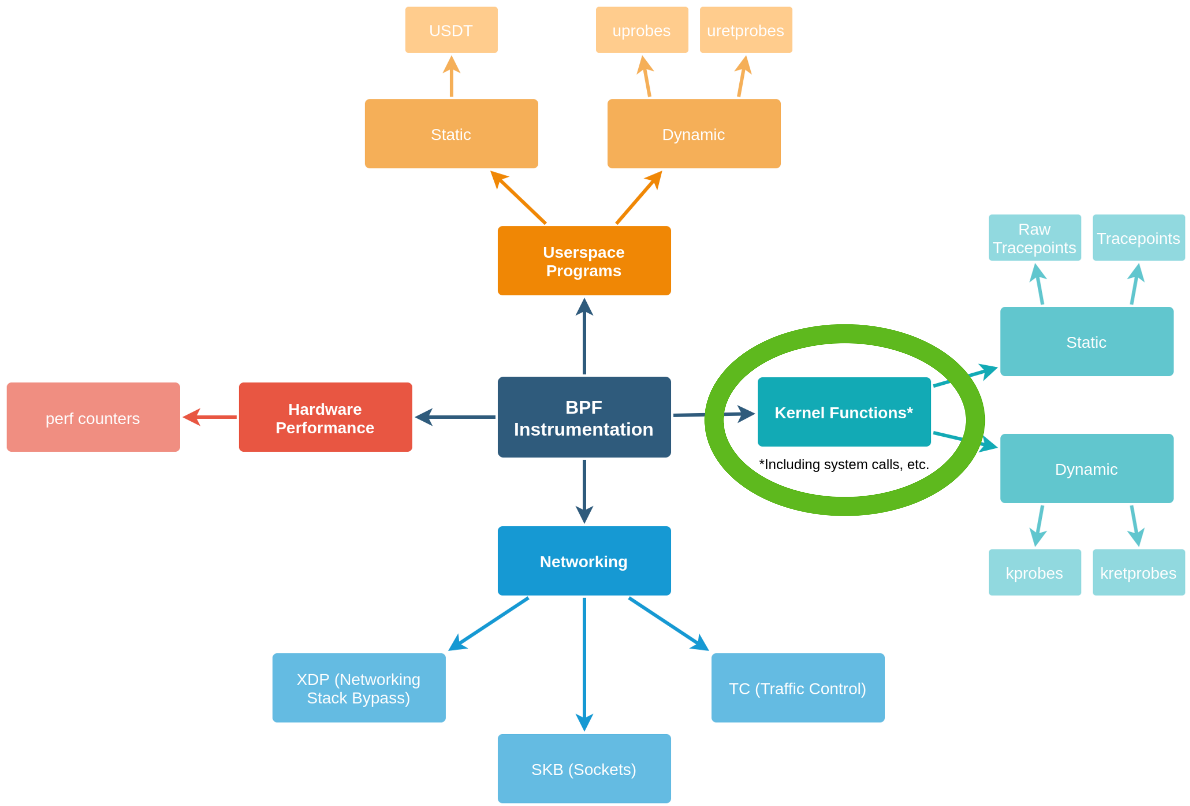


Figure 2: BPF program types and use cases.

Tracepoints vs Kprobes

- **Tracepoints**
  - Stable API
  - Limited number of these
    - *(getting better)*
    - *(1,872 of them on Linux 5.5)*

- **Kprobes**
  - Unstable API
  - But, can trace (almost) any kernel function

```
TRACEPOINT_PROBE(raw_syscalls, sys_enter)
```

↓

All syscall entrypoints

```
kprobe__get_signal(...)
```

↓

All signal handler invocations

tl;dr? Use tracepoints when you can, kprobes otherwise

Back to the Future

- **ebpH**
  - "Extended BPF + Process Homeostasis"
  - 20 year old technology...
  - Re-written using modern technology

Table 1: Comparing ebpH and pH.

System	Implementation	Portable	Production Safe	Low Mem. Overhead	Low Perf. Overhead	Detection	Response
pH	Kernel Patch	✗	✗	✓	✓	✓	✓
ebpH	eBPF + Userspace Daemon	✓	✓	✗	✓	✓	✗

Back to the Future

- **ebpH**
  - "Extended BPF + Process Homeostasis"
  - 20 year old technology...
  - Re-written using modern technology

Table 1: Comparing ebpH and pH.

System	Implementation	Portable	Production Safe	Low Mem. Overhead	Low Perf. Overhead	Detection	Response
pH	Kernel Patch	✗	✗	✓	✓	✓	✓
ebpH	eBPF + Userspace Daemon	✓	✓	✗	✓	✓	✗

Carleton UNIVERSITY

Canada's Capital University

Back to the Future

- **ebpH**
  - "Extended BPF + Process Homeostasis"
  - 20 year old technology...
  - Re-written using modern technology

Table 1: Comparing ebpH and pH.

System	Implementation	Portable	Production Safe	Low Mem. Overhead	Low Perf. Overhead	Detection	Response
pH	Kernel Patch	✗	✗	✓	✓	✓	✓
ebpH	eBPF + Userspace Daemon	✓	✓	✗	✓	✓	✗

ebpH in Detail

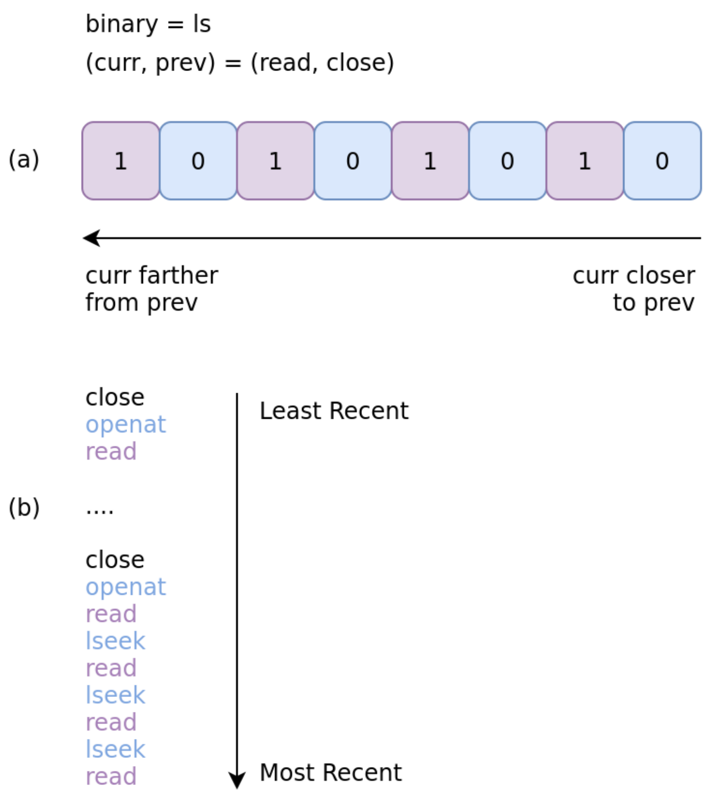


Figure 3: Example (read, close) lookahead pair from ls.

Same idea as pH:

1) **Trace all system calls**
2) **Build a profile of lookahead pairs from system calls**
3) **Gather enough data → Normal profile**
4) **Flag new lookahead pairs as anomalies**

ebpH in Detail

Carleton UNIVERSITY

Canada's Capital University

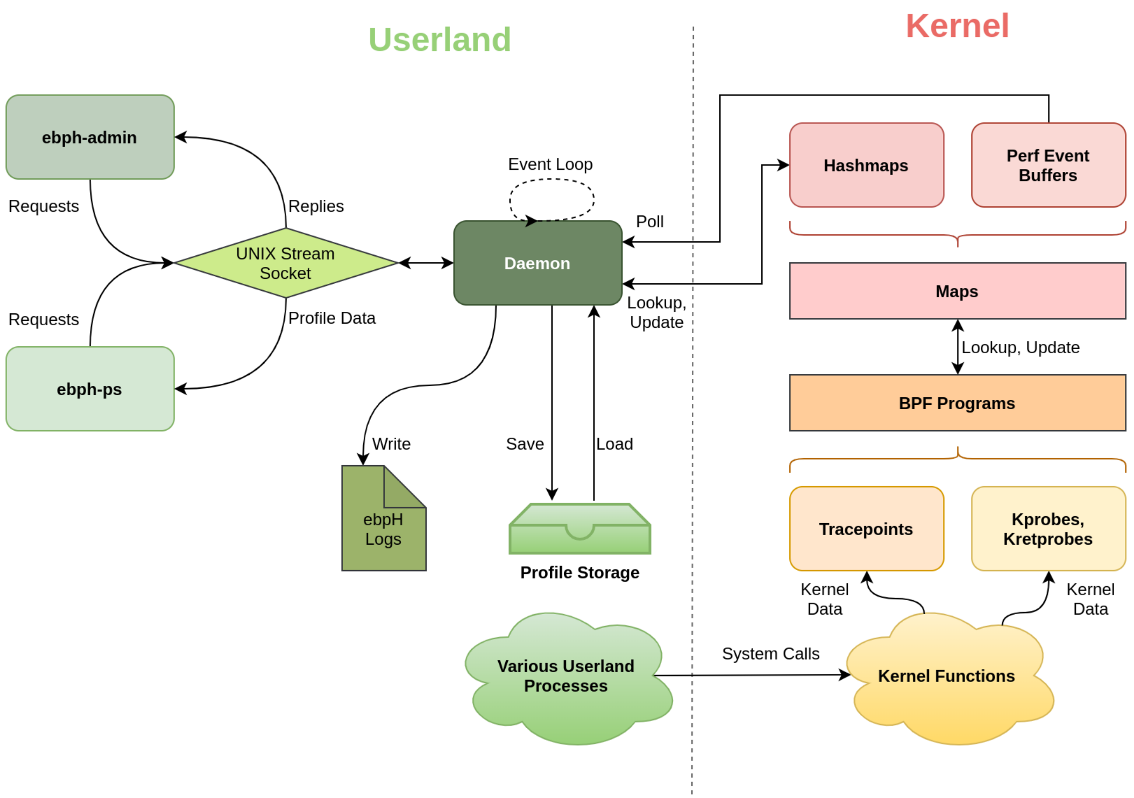


Figure 4: ebpH architecture in a nutshell.

Performance Analysis

Benchmarks

- lmbench OS suite (micro)
  - *System call overhead*
  - *Process creation overhead*
  - *IPC overhead (signals, UDS, pipes)*
- x11perf (micro)
  - *Syscall-heavy application overhead*
- bpfbench (macro, ad-hoc)
  - *Real world system call overhead*

Feedback welcome here

Performance Analysis

Carleton University
Canada's Capital University

Table 2: Systems used for benchmarking tests.

System	Description	Specifications	
`arch`	Personal workstation	Kernel	5.5.10-arch1-1
		CPU	Intel i7-7700K (8) @ 4.500GHz
		GPU	NVIDIA GeForce GTX 1070
		RAM	16GB DDR4 3000MT/s
		Disk	1TB Samsung NVMe M.2 SSD
`bronte`	CCSL workstation	Kernel	5.3.0-42-generic
		CPU	AMD Ryzen 7 1700 (16) @ 3.000GHz
		GPU	AMD Radeon RX
		RAM	32GB DDR4 1200MT/s
		Disk	250GB Samsung SATA SSD 850
`homeostasis`	Mediawiki server	Kernel	5.3.0-42-generic
		CPU	Intel i7-3615QM (8) @ 2.300GHz
		GPU	Integrated
		RAM	16GB DDR3 1600MT/s
		Disk	500GB Crucial CT525MX3

lmbench: System Calls

Figure 5: Various system call overheads.

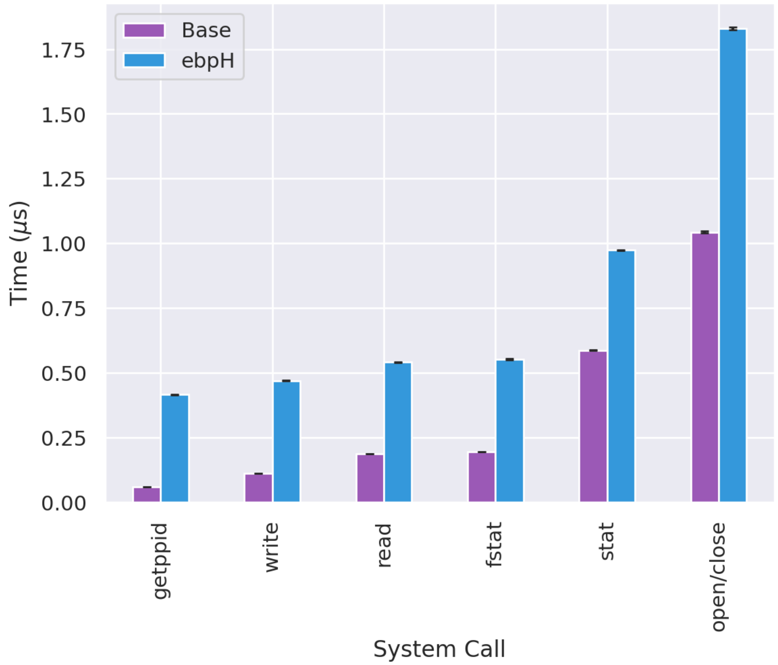
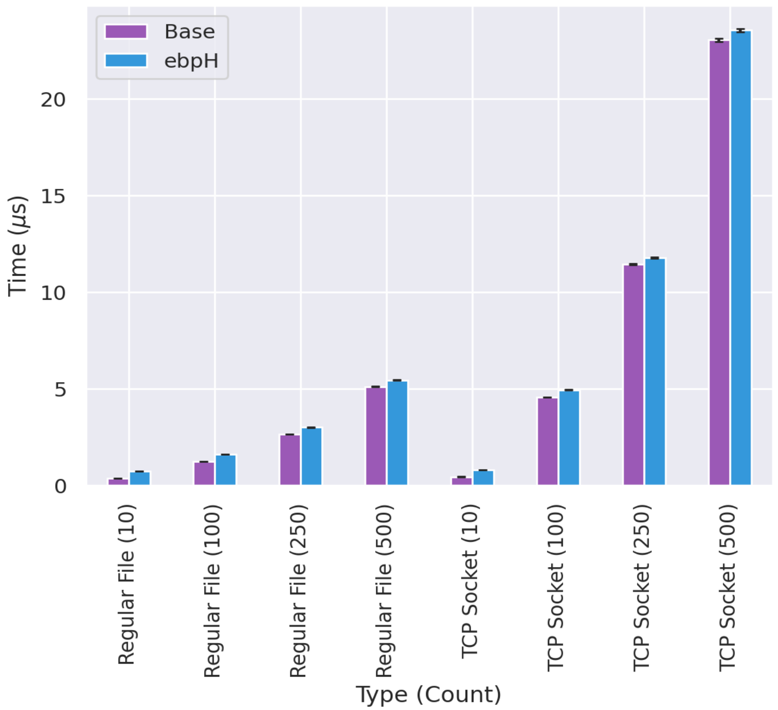


Figure 6: `select(2)` (blocking) system call overhead.



*All error bars show standard error

lmbench: System Calls

Table 3: Various system call overheads.

System Call	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`getppid`	0.058 (0.0023)	0.416 (0.0157)	0.357811	614.784969
`write`	0.111 (0.0039)	0.469 (0.0168)	0.357955	321.179901
`read`	0.187 (0.0064)	0.540 (0.0185)	0.353581	189.189001
`fstat`	0.194 (0.0062)	0.552 (0.0171)	0.357821	184.176095
`stat`	0.587 (0.0146)	0.973 (0.0250)	0.386082	65.765787
`open/close`	1.043 (0.0348)	1.830 (0.0567)	0.787454	75.509370

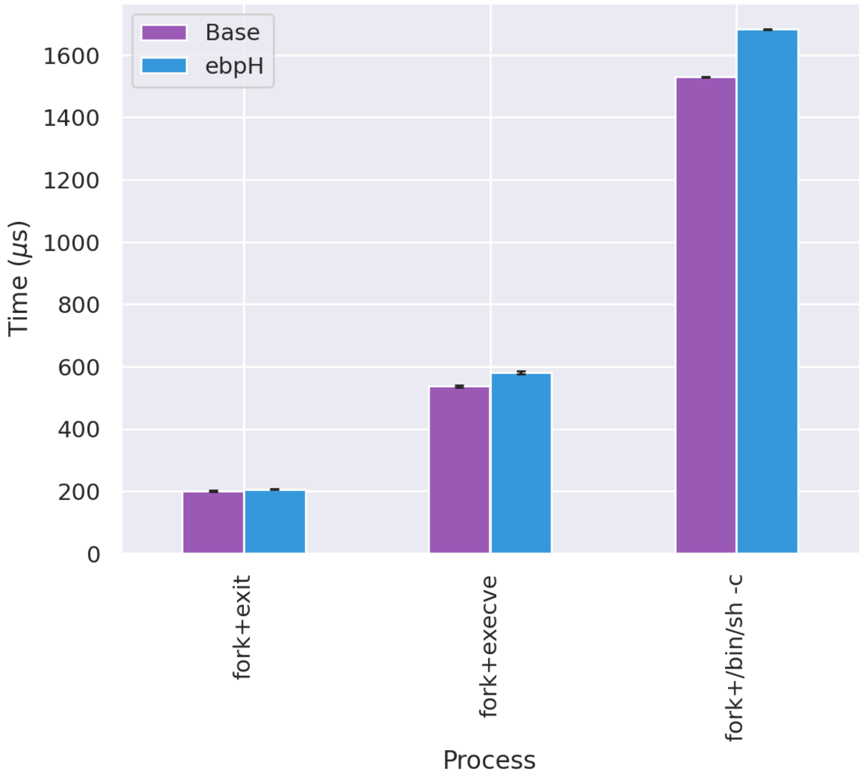
Table 4: `select(2)` (blocking) overhead.

Type	Count	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`Regular File`	10	0.362 (0.0128)	0.723 (0.0282)	0.360632	99.565990
`Regular File`	100	1.231 (0.0372)	1.596 (0.0443)	0.365494	29.699868
`Regular File`	250	2.639 (0.0799)	2.996 (0.0956)	0.356587	13.510287
`Regular File`	500	5.091 (0.1183)	5.426 (0.1490)	0.335187	6.584345
`TCP Socket`	10	0.436 (0.0144)	0.796 (0.0267)	0.360081	82.674990
`TCP Socket`	100	4.547 (0.1258)	4.928 (0.1792)	0.380938	8.378431
`TCP Socket`	250	11.433 (0.3849)	11.766 (0.3369)	0.332886	2.911606
`TCP Socket`	500	23.028 (0.8414)	23.530 (0.9567)	0.501917	2.179609

*Standard deviations in parentheses

Imbench: Process Creation

Figure 7: Process creation overhead.



*All error bars show standard error

lmbench: Process Creation

Table 5: Process creation overhead.

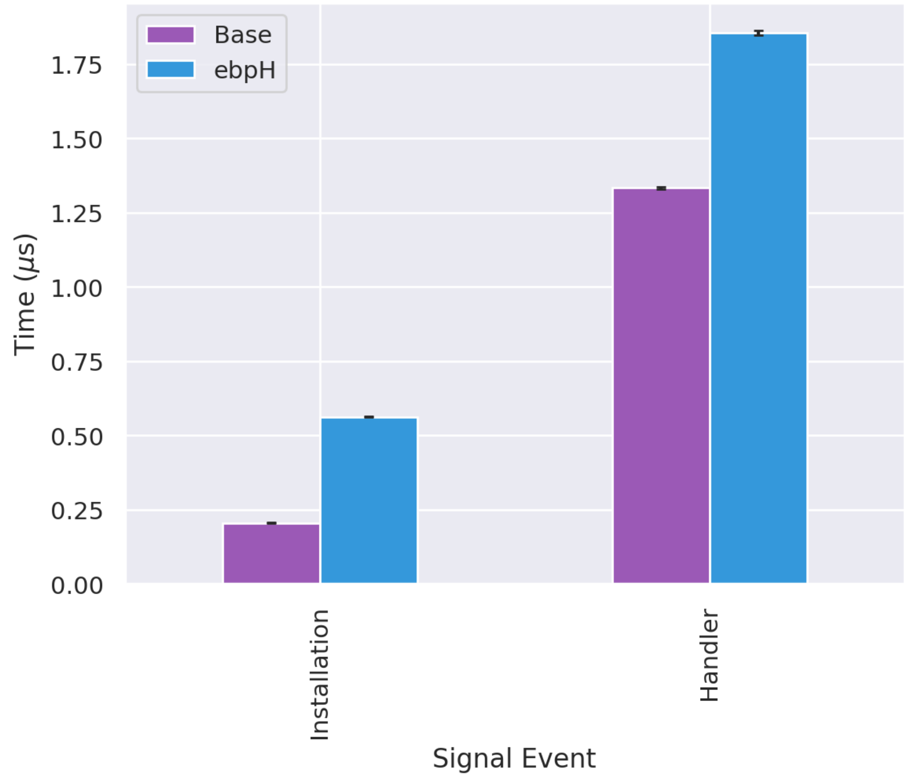
Process	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`fork+exit`	200.503 (17.3410)	205.998 (11.2935)	5.494621	2.740415
`fork+execve`	536.914 (30.5695)	580.532 (47.9242)	43.617913	8.123821
`fork+/bin/sh -c`	1529.053 (20.5609)	1682.445 (13.9791)	153.392500	10.031866

*Standard deviations in parentheses

Imbench: Signal Handlers

Canada's Capital University

Figure 8: Signal handler creation and invocation overheads.



*All error bars show standard error

lmbench: Signal Handlers

Table 6: Signal handler creation and invocation overheads.

Type	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`Installation`	0.205 (0.0061)	0.562 (0.0177)	0.357275	174.179379
`Handler`	1.333 (0.0420)	1.855 (0.0750)	0.522106	39.179999

*Standard deviations in parentheses

lmbench: Signal Handlers

This is just a short system call!
`(rt_sigaction(2))`

Table 6: Signal handler creation and invocation overheads.

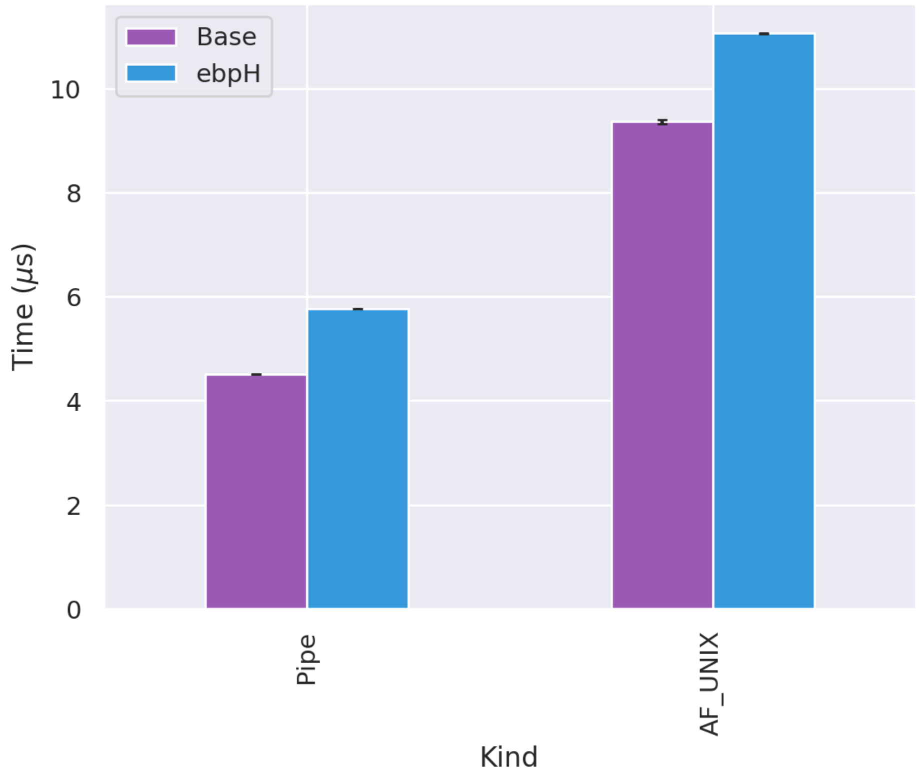
Type	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`Installation`	0.205 (0.0061)	0.562 (0.0177)	0.357275	174.179379
`Handler`	1.333 (0.0420)	1.855 (0.0750)	0.522106	39.179999

*Standard deviations in parentheses

lmbench: UDS, Pipes

Carleton UNIVERSITY
Canada's Capital University

Figure 9: IPC (UDS and pipes) overheads.



*All error bars show standard error

lmbench: UDS, Pipes

Table 7: IPC (UDS and pipes) overheads.

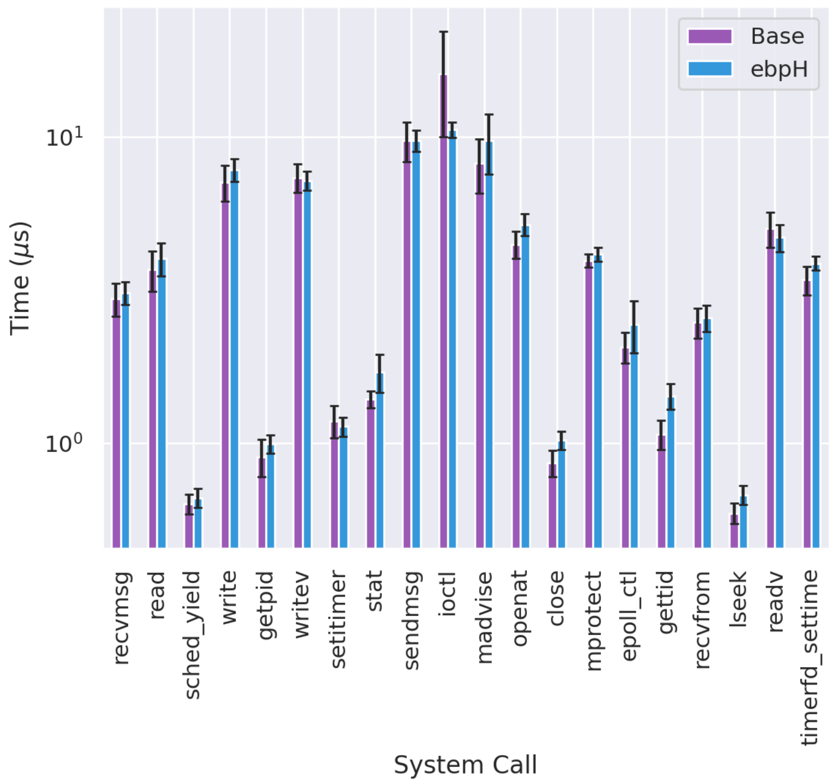
Type	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
Pipe	4.510 (0.0236)	5.768 (0.0394)	1.257634	27.886271
AF_UNIX	9.367 (0.3300)	11.067 (0.1340)	1.699890	18.148105

*Standard deviations in parentheses

x11perf

Carleton UNIVERSITY
Canada's Capital University

bpfbench: arch

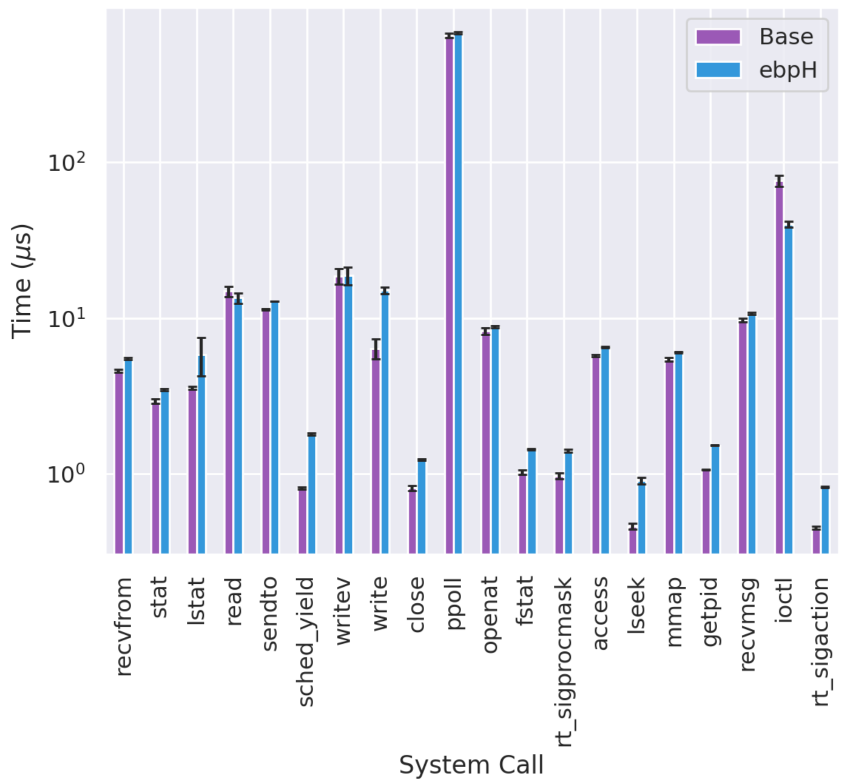


*All error bars show standard error
**Time scale is logarithmic

bpfbench: arch

*Standard deviations in parentheses

bpfbench: homeostasis



*All error bars show standard error
**Time scale is logarithmic

bpfbench: homeostasis

*Standard deviations in parentheses

bpfbench: bronte

*All error bars show standard error
**Time scale is logarithmic

bpfbench: bronte

*Standard deviations in parentheses

Cool New(ish) eBPF Features

- **`bpf_signal`**
  - Real-time signals from kernelspace (*instantaneous*)
  - SIGKILL, SIGSTOP, SIGCONT… you name it
  - Linux 5.3
- **`bpf_signal_thread`**
  - Like `bpf_signal` but target a specific thread
  - Linux 5.5
- **`bpf_override_return`**
  - Targeted error injection
  - Whitelisted kernel functions only :(
  - Linux 4.16

Future Work: Responding to Attacks

- **Add system call delays**
  - `bpf_signal` → send SIGSTOP and SIGCONT for delays
- **Add execve abortion**
  - `bpf_override_return` → target execve implementation

Recall this table:

System	Implementation	Portable	Production Safe	Low Mem. Overhead	Low Perf. Overhead	Detection	Response
pH	Kernel Patch	✗	✗	✓	✓	✓	✓
ebpH	eBPF + Userspace Daemon	✓	✓	✗	✓	✓	✓

Future Work: Saving on Memory Overhead

- **Current map allocation is too granular**
  - One big map for profiles, one big map for processes
- **Solution: use new map types**
  - `LRU_HASH` → smaller map, discard least recently used entries
  - HASH_OF_MAPS → nested maps for lookahead pairs (sparse array)

Recall this table:

System	Implementation	Portable	Production Safe	Low Mem. Overhead	Low Perf. Overhead	Detection	Response
pH	Kernel Patch	✗	✗	✓	✓	✓	✓
ebpH	eBPF + Userspace Daemon	✓	✓	✓	✓	✓	✓

Conclusion

- **ebpH:**
  - is as fast as the original implementation.
  - supports most of the original functionality.
  - can be made even better, using new eBPF features.
- **Future of ebpH?**
  - Ecosystem of BPF programs
  - All talking to each other, sharing information about diff. parts of system
  - Beyond just system call tracing
- **Future of eBPF in OS security?**
  - We are going to be seeing a lot more of this.
  - eBPF keeps getting better and better.
  - Replacing many in-kernel implementations with something safer, with less opportunity cost.

Some Links

https://github.com/iovisor/bcc

https://github.com/willfindlay/honors-thesis

https://github.com/willfindlay/ebph

PRs welcome!

Thank you!