Carleton
UNIVERSITY

Canada's Capital University

Host-Based Anomaly Detection with Extended BPF

William Findlay

Supervisor: Dr. Anil Somayaji

April 10th, 2020

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)

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?

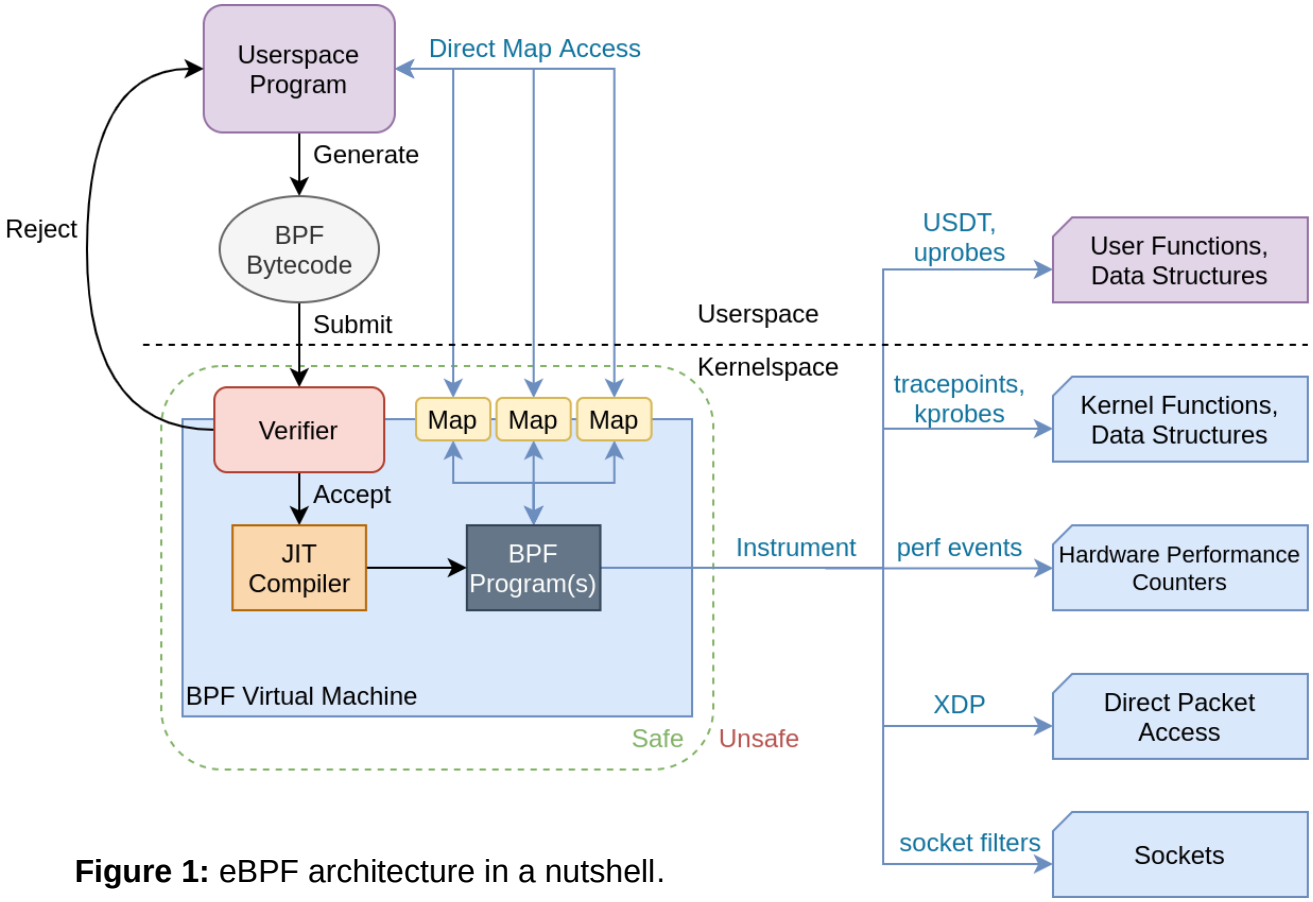
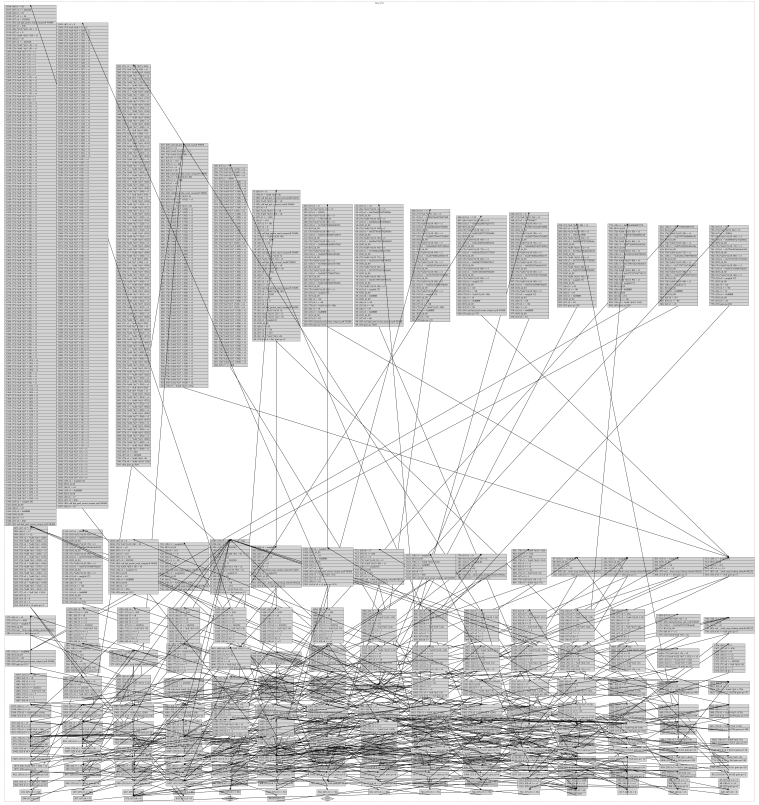


Figure 1: eBPF architecture in a nutshell.

What is eBPF?

- **eBPF verifier**
  - Ensure BPF program will not crash the kernel
  - 10,000 lines of C code in kernel
  - BPF system call traps to verifier on PROG_LOAD
- **How to guarantee safety?**
  Severe limitations + simulation.
  - 512 byte stack space
  - No unbounded loops
  - Max 4 million BPF instructions per program
  - No buffer access with unbounded induction variable
  - Etc.

BPF Programs Still Can Be Complex



- **ebpH's sys_exit tracepoint**
  - bpftool + graphviz osage
  - 1,574 BPF insns
  - 1,930 machine insns
- **Current hard limit:**
  - 4 million BPF insns

Figure 3: Instruction flow graph of ebpH's sys_exit tracepoint.

What is eBPF?

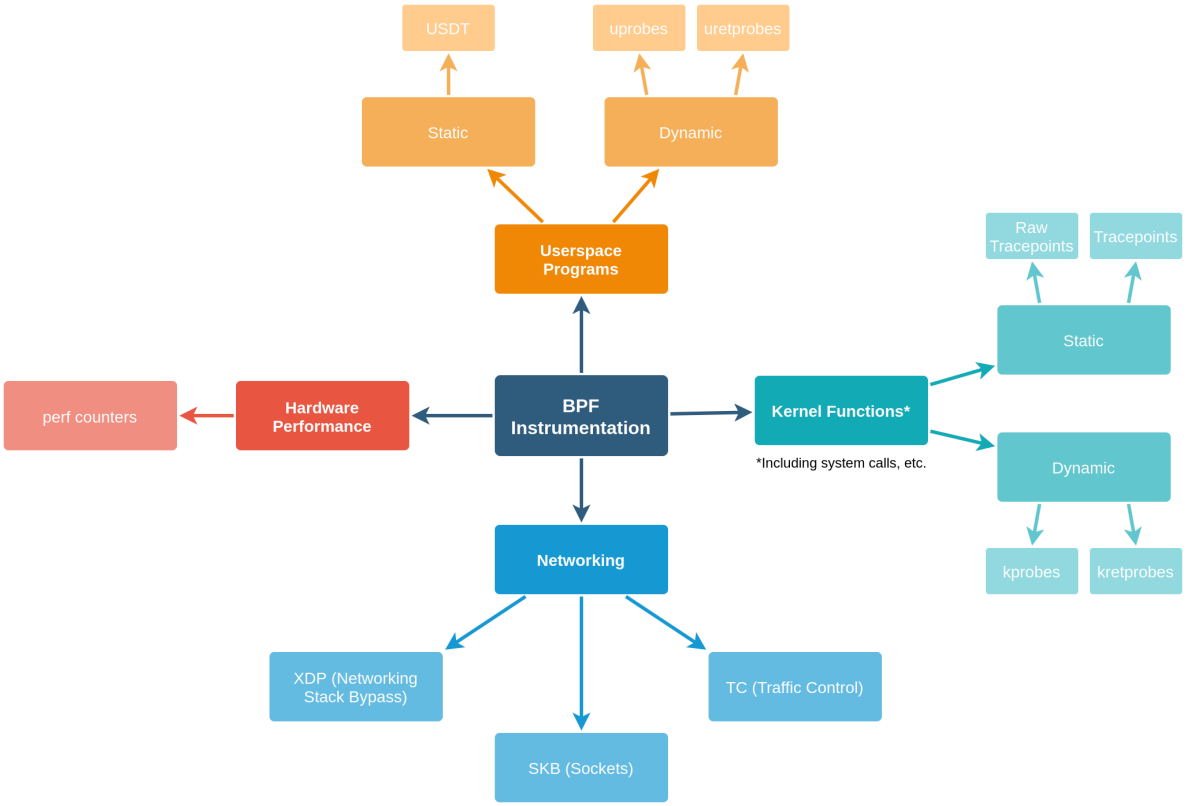


Figure 2: BPF program types and use cases. Not an exhaustive list, but representative of the most common program types.

What is eBPF?

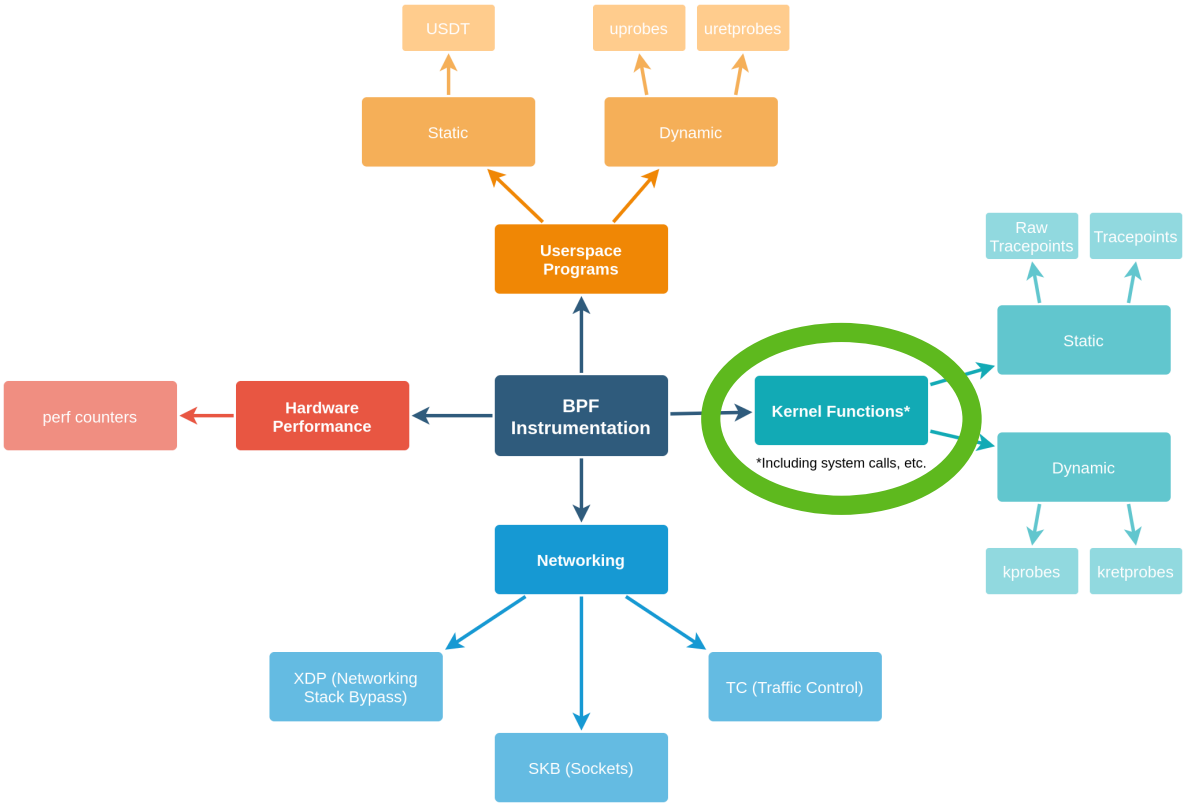


Figure 2: BPF program types and use cases. Not an exhaustive list, but representative of the most common program types.

What is eBPF?

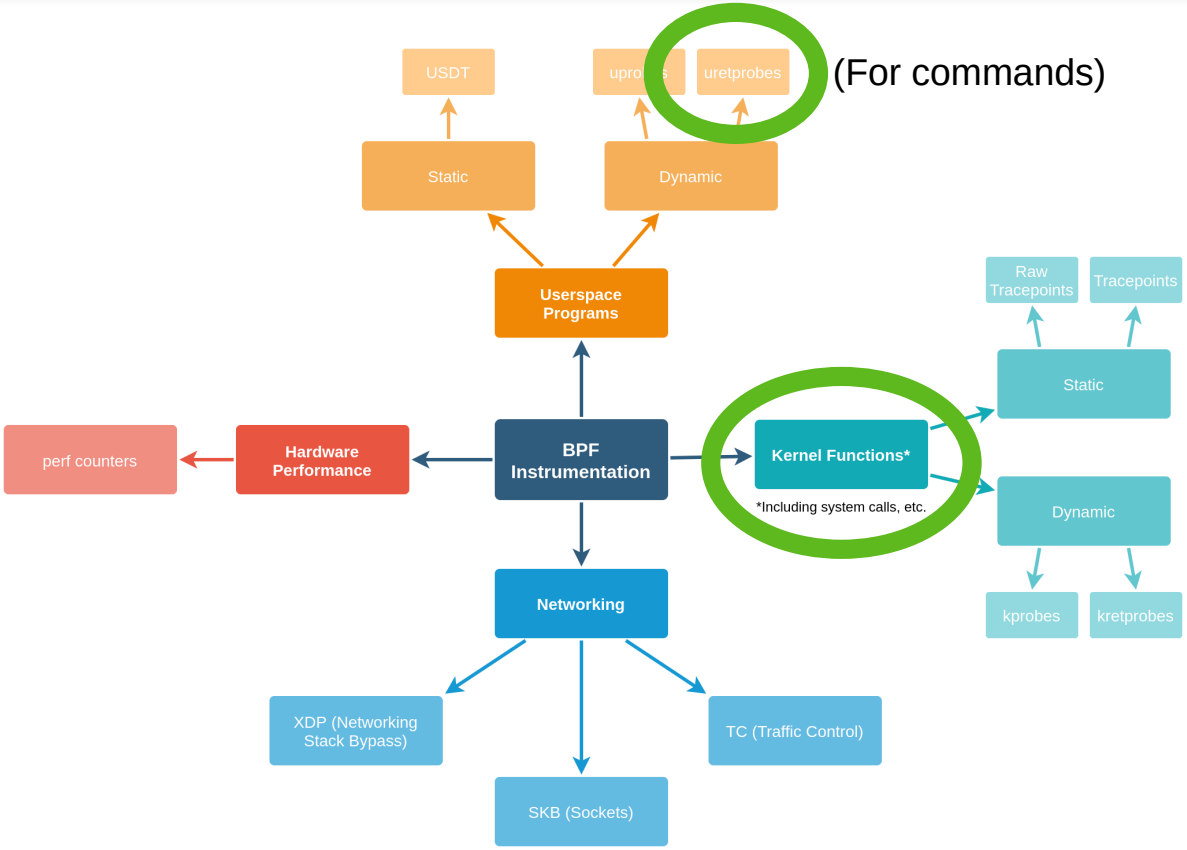


Figure 2: BPF program types and use cases. Not an exhaustive list, but representative of the most common program types.

Tracepoints vs Kprobes

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

`TRACEPOINT_PROBE(raw_syscalls, sys_exit)`

All syscall returns

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

`kprobe__get_signal(...)`

All invocations of `get_signal()` in the Kernel

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	✓	✓	✗	✓	✓	✗

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

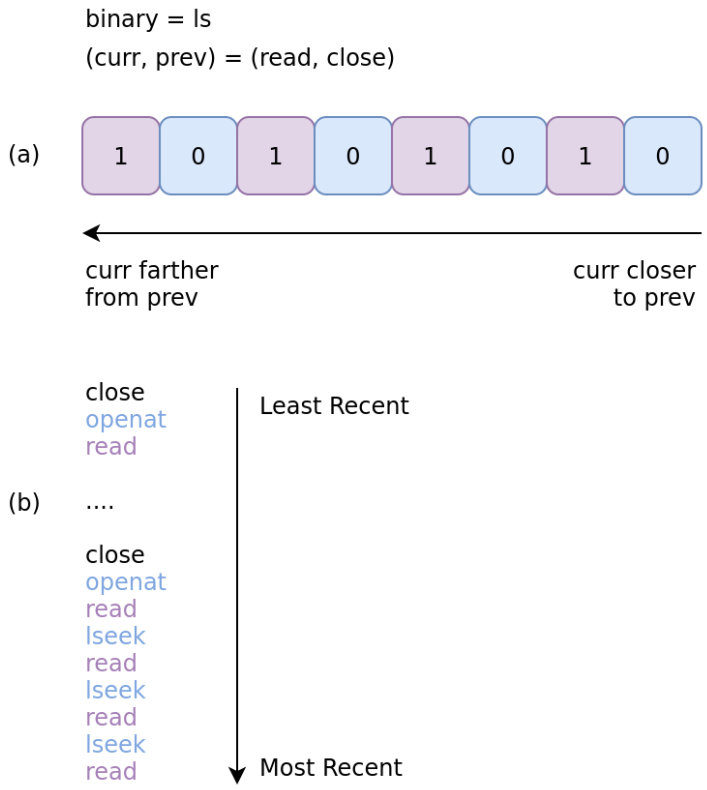


Figure 4: Example (read, close) lookahead pair from ls.

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

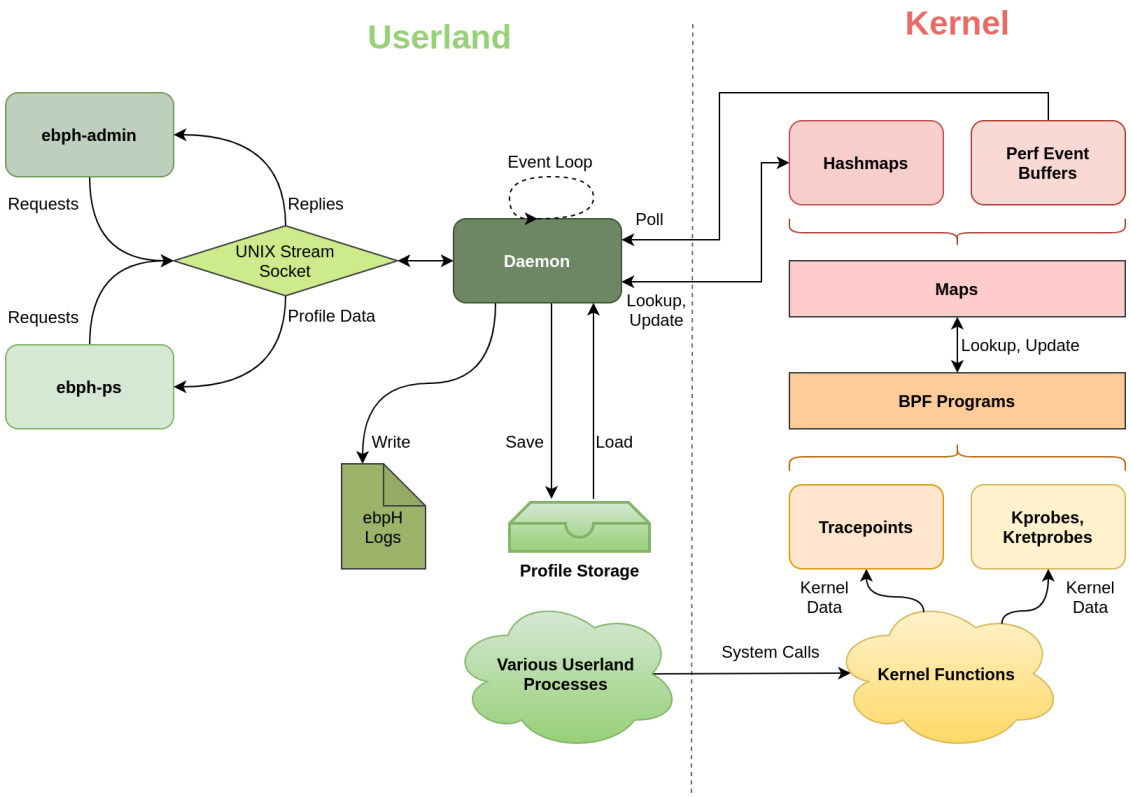


Figure 5: ebpH architecture in a nutshell.

Performance Analysis

- **Benchmarks**
  - lmbench OS suite (micro)
    - *System call overhead*
    - *Process creation overhead*
    - *IPC overhead (signals, UDS, pipes)*
  - bpfbench (macro, ad-hoc)
    - *Real world system call overhead*
- **Feedback welcome here**

Performance Analysis

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

Imbench: 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: System Calls

Figure 6: Various system call overheads.

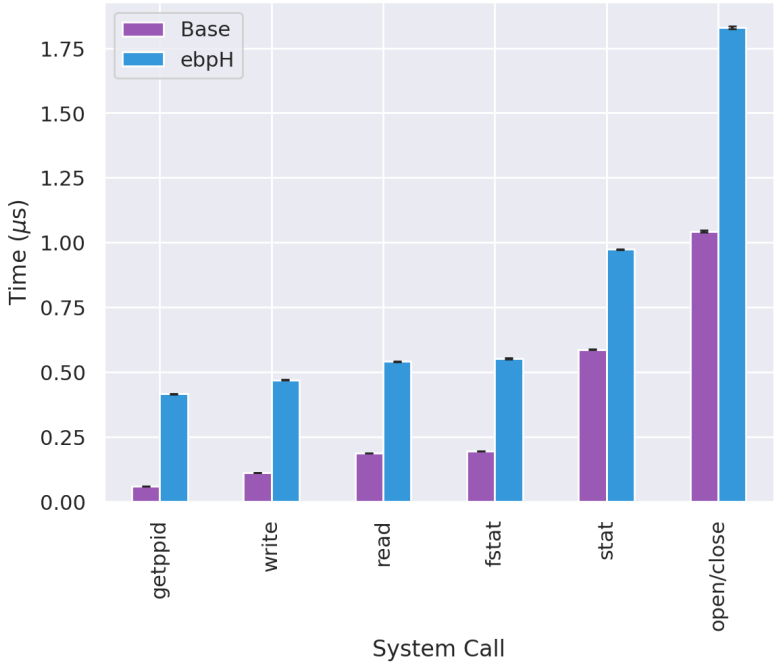
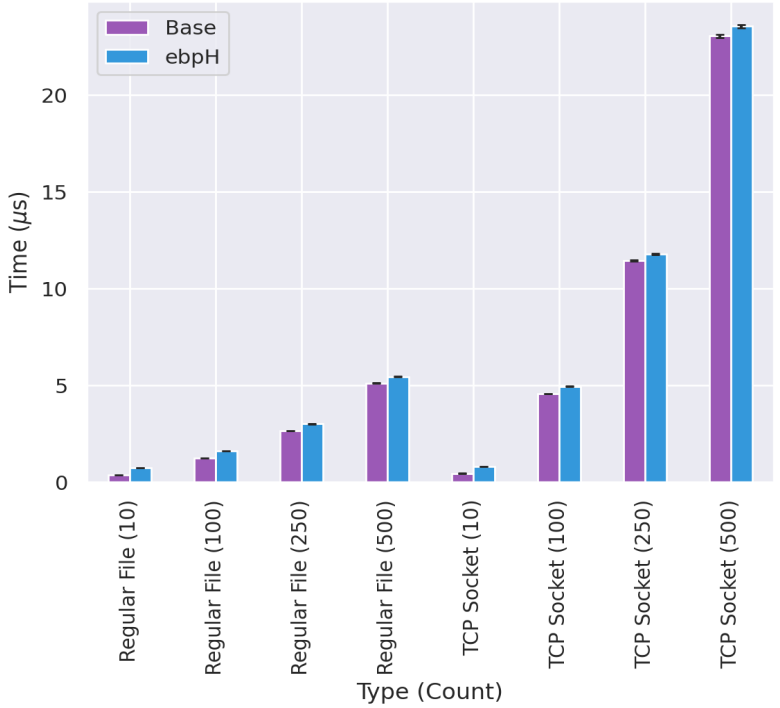


Figure 7: `select(2)` (blocking) system call overhead.



*All error bars show standard error

Imbench: Process Creation

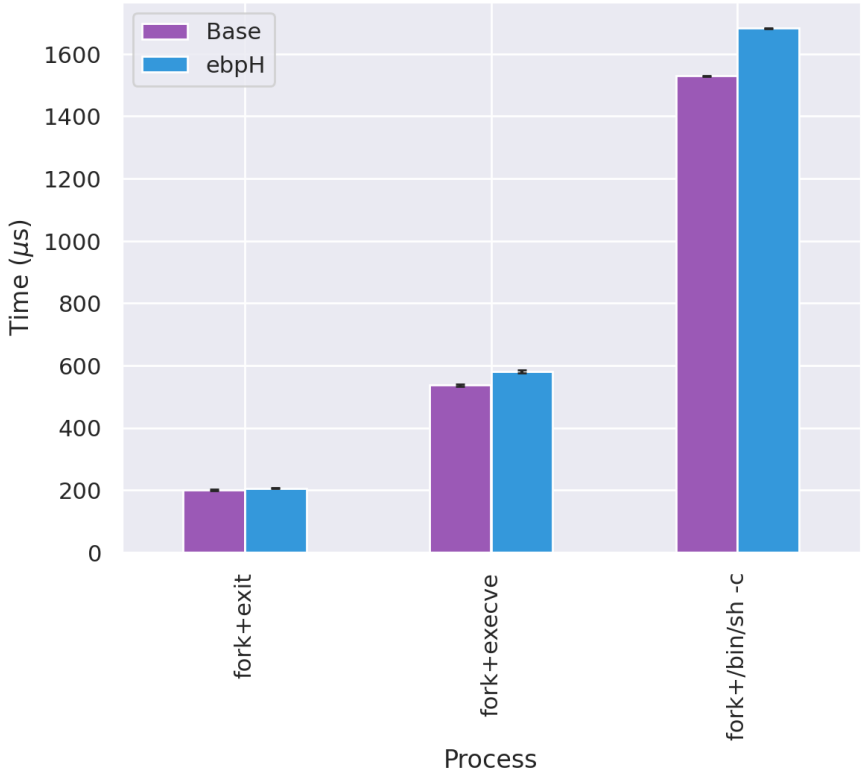
Table 5: Process creation overhead.

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: Process Creation

Figure 8: Process creation overhead.



*All error bars show standard error

Imbench: 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

Imbench: Signal Handlers

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

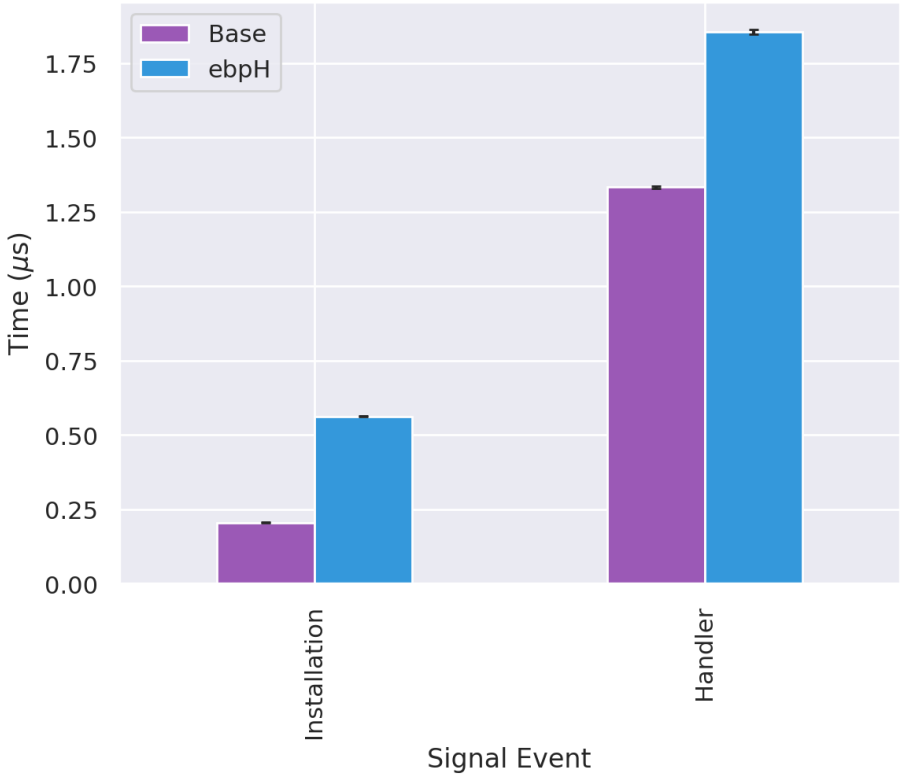
Table 6: Signal handler creation and invocation overheads.

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

Imbench: Signal Handlers

Figure 9: Signal handler creation and invocation overheads.



*All error bars show standard error

Imbench: UDS, Pipes

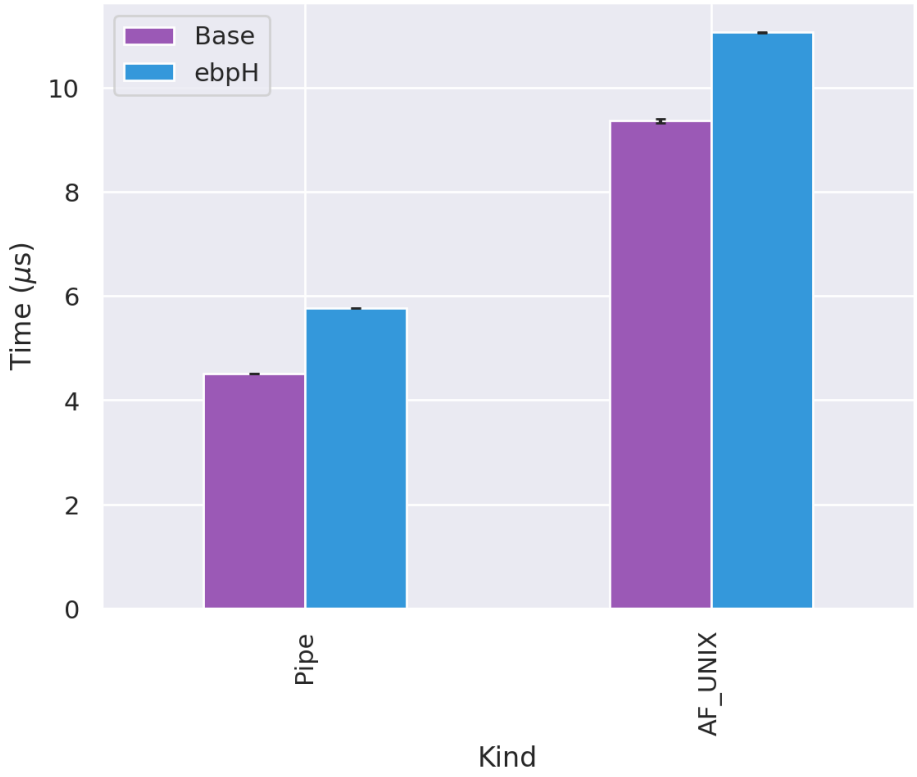
Table 7: IPC (UDS and pipes) overheads.

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

Imbench: UDS, Pipes

Figure 10: IPC (UDS and pipes) overheads.



*All error bars show standard error

bpfbench: arch

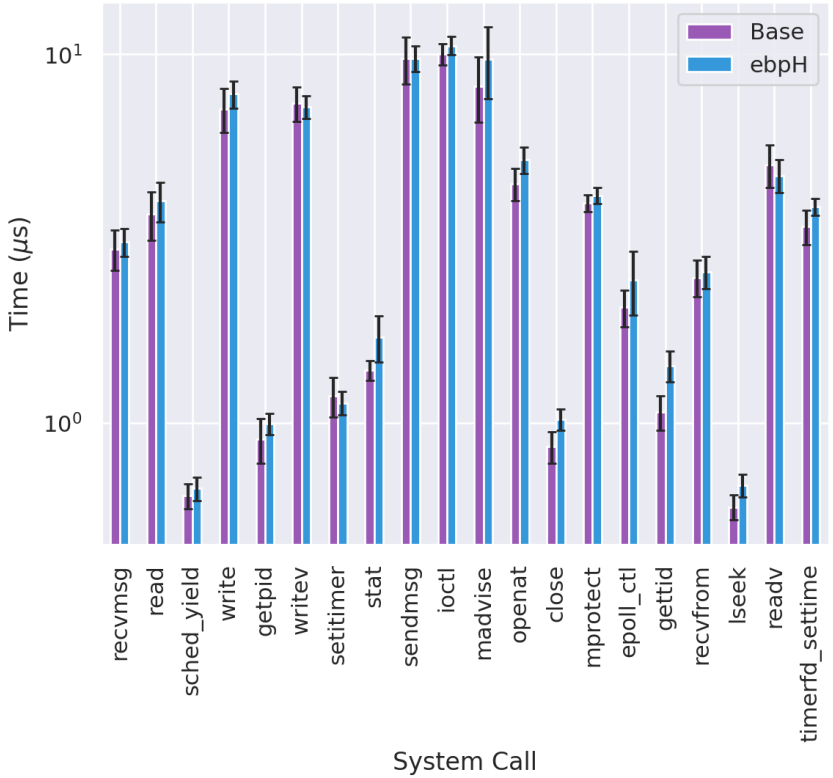
Table 8: Top 20 system call overheads by count in the arch-3day dataset.

System Call	Count	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
recvmsg	1581788435	2.964 (1.2181)	3.101 (0.8429)	0.137	4.631
read	559839058	3.684 (1.8254)	4.005 (1.5622)	0.321	8.713
sched_yield	557122573	0.634 (0.1591)	0.663 (0.1554)	0.030	4.697
write	422940971	7.108 (3.2055)	7.808 (2.0907)	0.700	9.851
getpid	357026536	0.903 (0.4142)	0.994 (0.2102)	0.091	10.111
writev	297206760	7.358 (2.5764)	7.193 (1.6120)	-0.165	-2.239
setitimer	246047072	1.181 (0.4767)	1.133 (0.2606)	-0.048	-4.094
stat	233556922	1.389 (0.2925)	1.706 (0.7753)	0.317	22.817
sendmsg	176379278	9.729 (4.7259)	9.744 (2.4594)	0.014	0.148
ioctl	101386112	10.028 (2.0969)	10.558 (1.9098)	0.530	5.285
madvise	58208675	8.194 (5.4377)	9.717 (6.8087)	1.524	18.599
openat	47987948	4.453 (1.4879)	5.174 (1.3422)	0.721	16.201
close	43600631	0.862 (0.2795)	1.022 (0.2182)	0.159	18.478
mprotect	39922557	3.950 (0.6694)	4.142 (0.6807)	0.191	4.844
epoll_ctl	39571935	2.058 (0.7853)	2.441 (1.5081)	0.383	18.585
gettid	39531180	1.068 (0.3854)	1.426 (0.4344)	0.358	33.466
recvfrom	39304167	2.478 (0.9323)	2.567 (0.8106)	0.089	3.603
lseek	25079617	0.591 (0.1506)	0.677 (0.1540)	0.086	14.535
readv	22072501	5.009 (2.1978)	4.692 (1.5339)	-0.318	-6.339
timerfd_settime	21359440	3.406 (1.2268)	3.866 (0.6438)	0.459	13.485

*Standard deviations in parentheses

bpfbench: arch

Figure 11: Top 20 system call overheads by count in the arch-3day dataset.



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

bpfbench: homeostasis

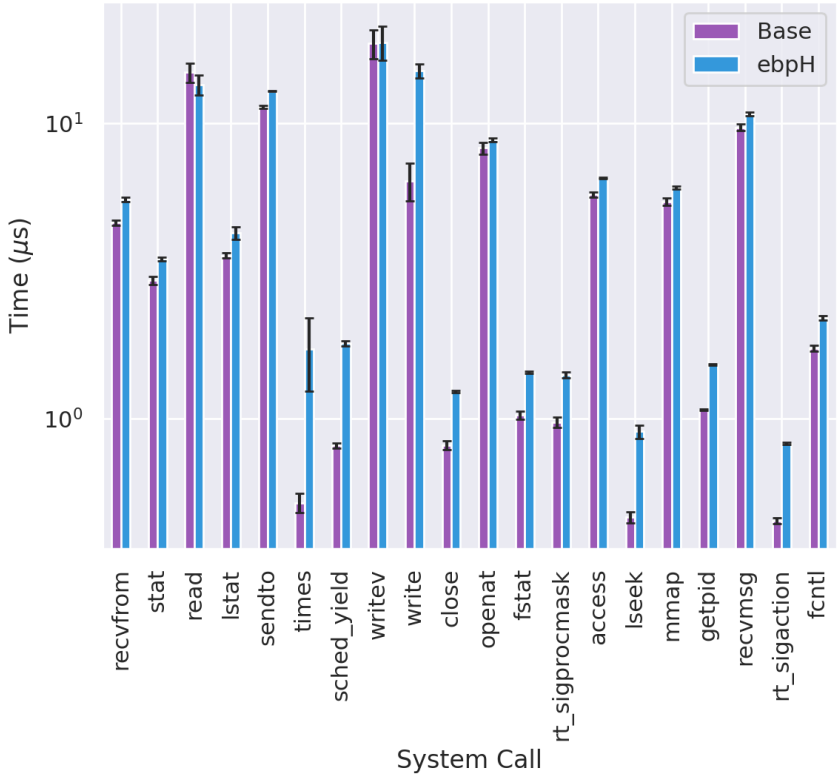
Table 9: Top 20 system call overheads by count in the homeostasis-3day dataset.

System Call	Count	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
recvfrom	22311033	4.599 (0.3375)	5.505 (0.2877)	0.906	19.705
stat	16578427	2.929 (0.3409)	3.467 (0.1842)	0.538	18.362
read	12023727	14.824 (3.8911)	13.446 (3.5872)	-1.378	-9.296
lstat	11953074	3.560 (0.2693)	4.238 (0.6952)	0.678	19.052
sendto	11070639	11.330 (0.4644)	12.800 (0.2013)	1.469	12.969
times	9085541	0.518 (0.1353)	1.712 (1.6402)	1.194	230.587
sched_yield	7951003	0.810 (0.0547)	1.795 (0.1257)	0.986	121.760
writev	6195312	18.546 (7.1331)	18.717 (8.5272)	0.171	0.925
write	6102652	6.362 (3.2141)	15.007 (2.7556)	8.645	135.890
close	5701429	0.811 (0.0969)	1.236 (0.0387)	0.425	52.405
openat	4985018	8.221 (1.3344)	8.762 (0.3944)	0.541	6.578
fstat	4196505	1.025 (0.1166)	1.434 (0.0439)	0.409	39.878
rt_sigprocmask	3746321	0.969 (0.1343)	1.405 (0.1113)	0.436	44.988
access	3302798	5.721 (0.3865)	6.499 (0.1562)	0.778	13.599
lseek	3010731	0.463 (0.0679)	0.903 (0.1624)	0.440	95.068
mmap	2438472	5.424 (0.5294)	6.022 (0.2403)	0.598	11.024
getpid	2063206	1.068 (0.0177)	1.520 (0.0247)	0.452	42.350
recvmsg	1745288	9.652 (0.8664)	10.700 (0.5092)	1.048	10.857
rt_sigaction	1308454	0.450 (0.0363)	0.822 (0.0272)	0.372	82.691
fcntl	828473	1.727 (0.1388)	2.189 (0.1229)	0.462	26.749

*Standard deviations in parentheses

bpfbench: homeostasis

Figure 12: Top 20 system call overheads by count in the homeostasis-3day dataset.



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

bpfbench: bronte

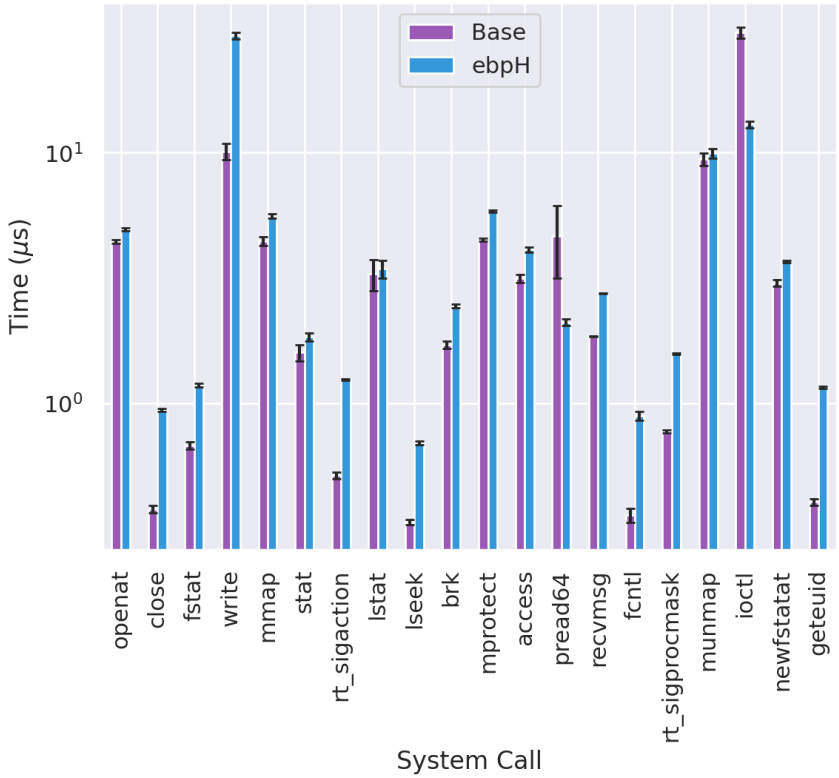
Table 10: Top 20 system call overheads by count in the bronte-7day dataset.

System Call	Count	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
openat	280543846	4.429 (0.4253)	4.945 (0.3266)	0.516	11.656
close	167111207	0.380 (0.0606)	0.943 (0.0663)	0.563	148.153
fstat	159725956	0.680 (0.1099)	1.185 (0.0953)	0.504	74.141
write	45140161	10.116 (3.6480)	29.324 (4.8695)	19.208	189.887
mmap	37679318	4.439 (0.8666)	5.606 (0.6333)	1.168	26.304
stat	21164450	1.597 (0.6161)	1.843 (0.3344)	0.246	15.388
rt_sigaction	17243143	0.517 (0.0789)	1.245 (0.0579)	0.728	140.751
lstat	16056909	3.284 (2.2767)	3.446 (1.4707)	0.163	4.951
lseek	15244865	0.337 (0.0427)	0.697 (0.0548)	0.360	107.050
brk	14775427	1.719 (0.2968)	2.447 (0.2186)	0.728	42.317
mprotect	12492042	4.502 (0.2475)	5.836 (0.3272)	1.334	29.623
access	10748181	3.159 (0.5712)	4.113 (0.5105)	0.955	30.229
pread64	9851780	4.652 (6.6537)	2.110 (0.3353)	-2.542	-54.639
recvmsg	5677371	1.853 (0.0300)	2.746 (0.0361)	0.893	48.199
fcntl	5201423	0.359 (0.1118)	0.892 (0.1908)	0.533	148.594
rt_sigprocmask	4311686	0.774 (0.0583)	1.585 (0.0601)	0.811	104.842
munmap	3261382	9.409 (2.7439)	9.965 (2.3270)	0.556	5.906
ioctl	2691686	30.088 (7.2827)	12.949 (1.9564)	-17.139	-56.964
newfstatat	2634685	3.020 (0.4457)	3.687 (0.2255)	0.667	22.090
geteuid	2035211	0.405 (0.0576)	1.161 (0.0718)	0.756	186.801

*Standard deviations in parentheses

bpfbench: bronte

Figure 13: Top 20 system call overheads by count in the bronte-7day dataset.



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

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!