Carleton UNIVERSITY

Canada's Capital University

Host-Based Anomaly Detection with Extended BPF

William Findlay

Supervisor: Dr. Anil Somayaji

April 10th, 2020

eBPF at a High Level

- **Inject user-specified code into the kernel**
- **BPF code runs in kernelspace, can instrument essentially all system behavior**
- **This sounds a lot like a kernel module…**
  - Key difference? Safety.
- **Before they can run in the kernel, BPF programs are statically verified**

Carleton UNIVERSITY
Canada's Capital University

eBPF in Industry

- **Performance monitoring**
  - Netflix
  - Facebook
  - Google
  - … many others
- **Established tools**
  - bcc-tools (over 100 performance monitoring / visibility tools)
- **Network security**
  - Cloudflare's DDoS mitigation stack

What Makes eBPF Good for Security?

- **A lot of security is about what we can see**
  - eBPF lets you see everything about your system
  - … and it can do this with crazy low overhead
- **Before eBPF, system introspection came at a cost**
  - Speed
  - Scope
  - Production safety
- **eBPF can do everything, without the speed / scope / safety trade-off**
  - Although eBPF comes with its own nuances (more on this later)

Carleton UNIVERSITY

Canada's Capital University

eBPF Architecture

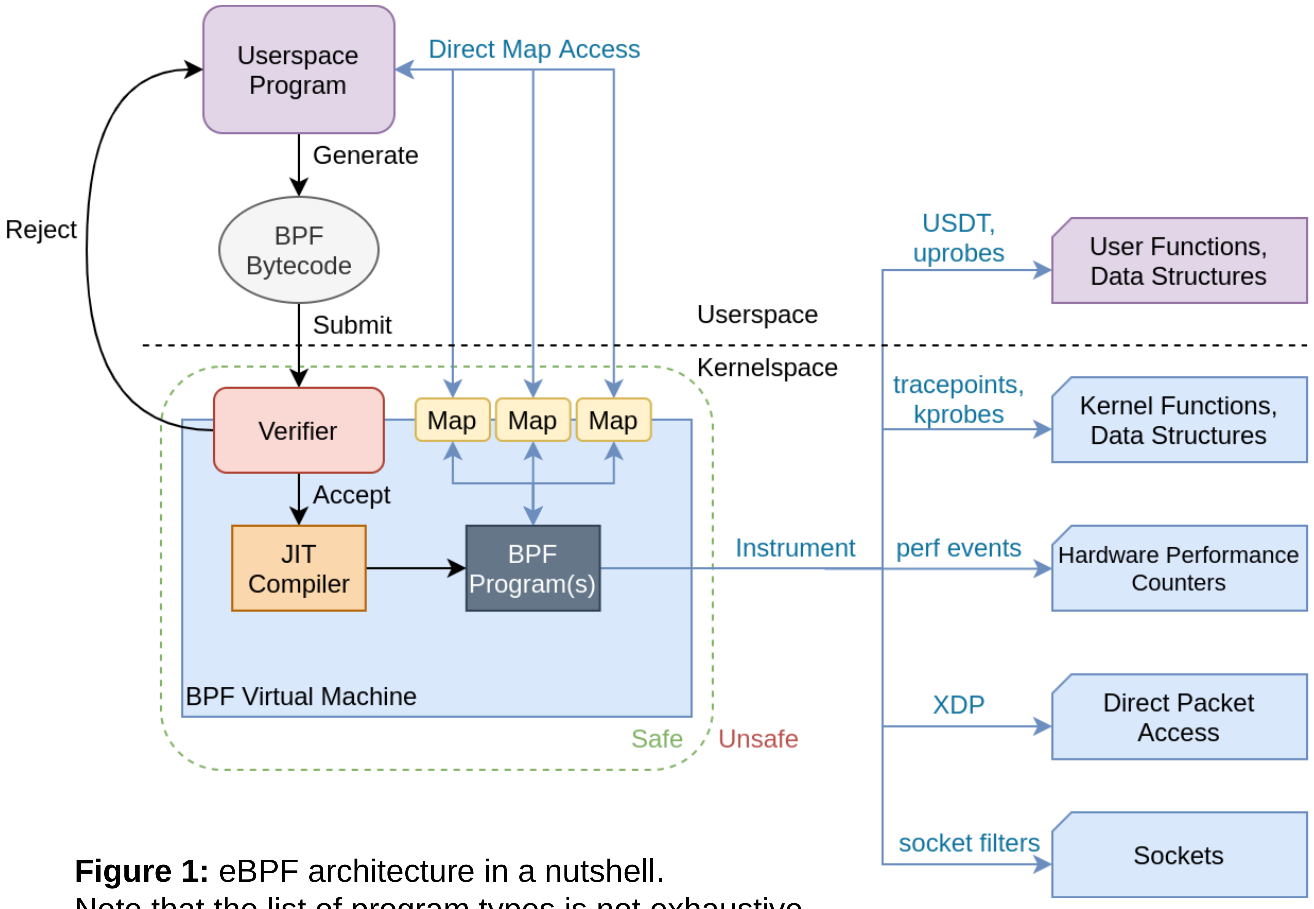


Figure 1: eBPF architecture in a nutshell.
Note that the list of program types is not exhaustive.

The Verifier in Detail

- **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?**
  Limitations + simulation + static analysis
  - 512 byte stack space
  - No unbounded loops
  - Max 1 million BPF instructions per program
  - No buffer access with unbounded induction variable
  - Etc.

BPF Programs Still Can Be Complex

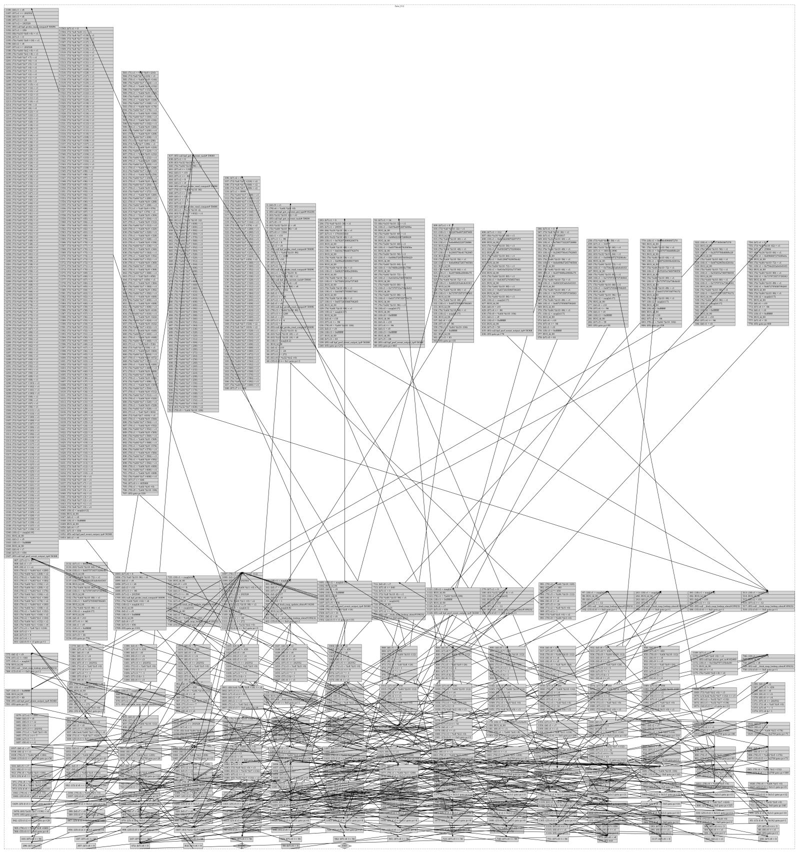


Figure 2: Instruction flow graph of ebpH's sys_exit tracepoint.

- **ebpH's sys_exit tracepoint**
  - bpftool + graphviz osage
  - 1,574 BPF instructions
  - 1,930 machine instructions
- **BPF programs can interact with each other**
  - Direct map access
  - Tail calls

Carleton UNIVERSITY

Canada's Capital University

Process Homeostasis

- **Early anomaly detection system by Anil Somayaji**
- **The idea:**
  - Instrument system calls to build per-executable behavioral profiles
  - Delay anomalous system calls proportionally to recent anomalies
- **Problems?**
  - Implemented as a kernel patch
  - Need to make crazy modifications for it to work
  - Patch the scheduler, write in assembly language, etc.
  - Not production-safe
  - Not portable

Carleton UNIVERSITY
Canada's Capital University

ebpH: 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

ebpH: 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: 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

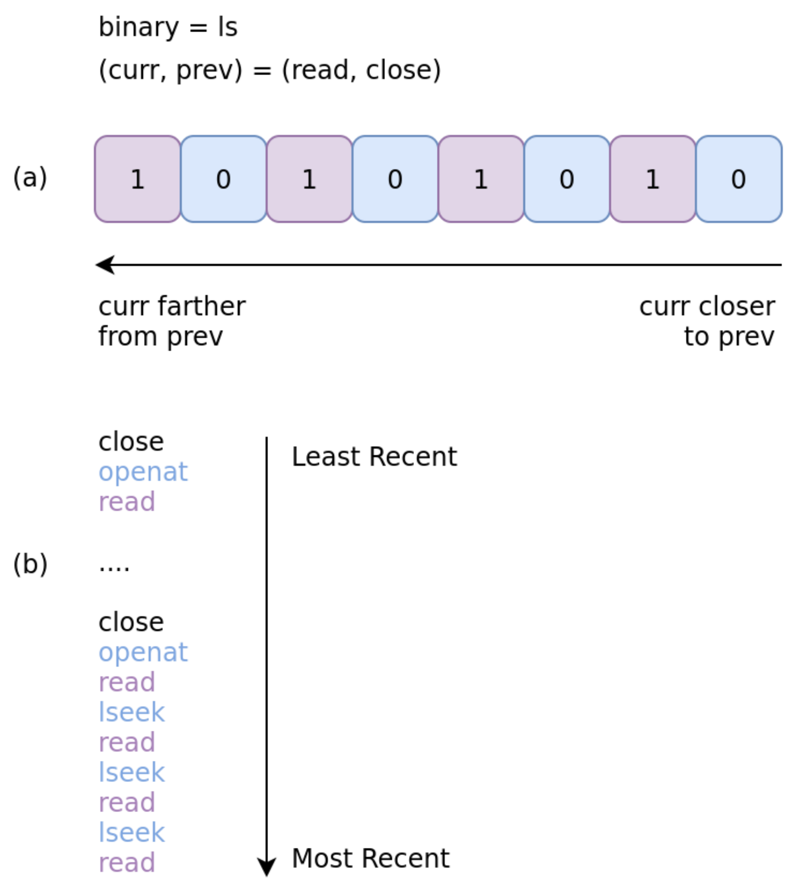


Figure 3: Example (read, close) lookahead pair from ls.

Same idea as pH:

- **Trace system calls**
- **Build profile of lookahead pairs**
- **Gather enough data**
- **Flag new lookahead pairs as anomalies**

eBPF makes this safe.

How ebpH Collects Data

- **Tracepoints (static kernel tracing)**
  - Instrument system calls
  - Instrument scheduler
- **Kprobes (dynamic kernel tracing)**
  - Instrument signal delivery
- **Uprobes (dynamic user tracing)**
  - Instrument libebph.so
  - Allow user to issue commands to ebpH's BPF programs

ebpH Architecture

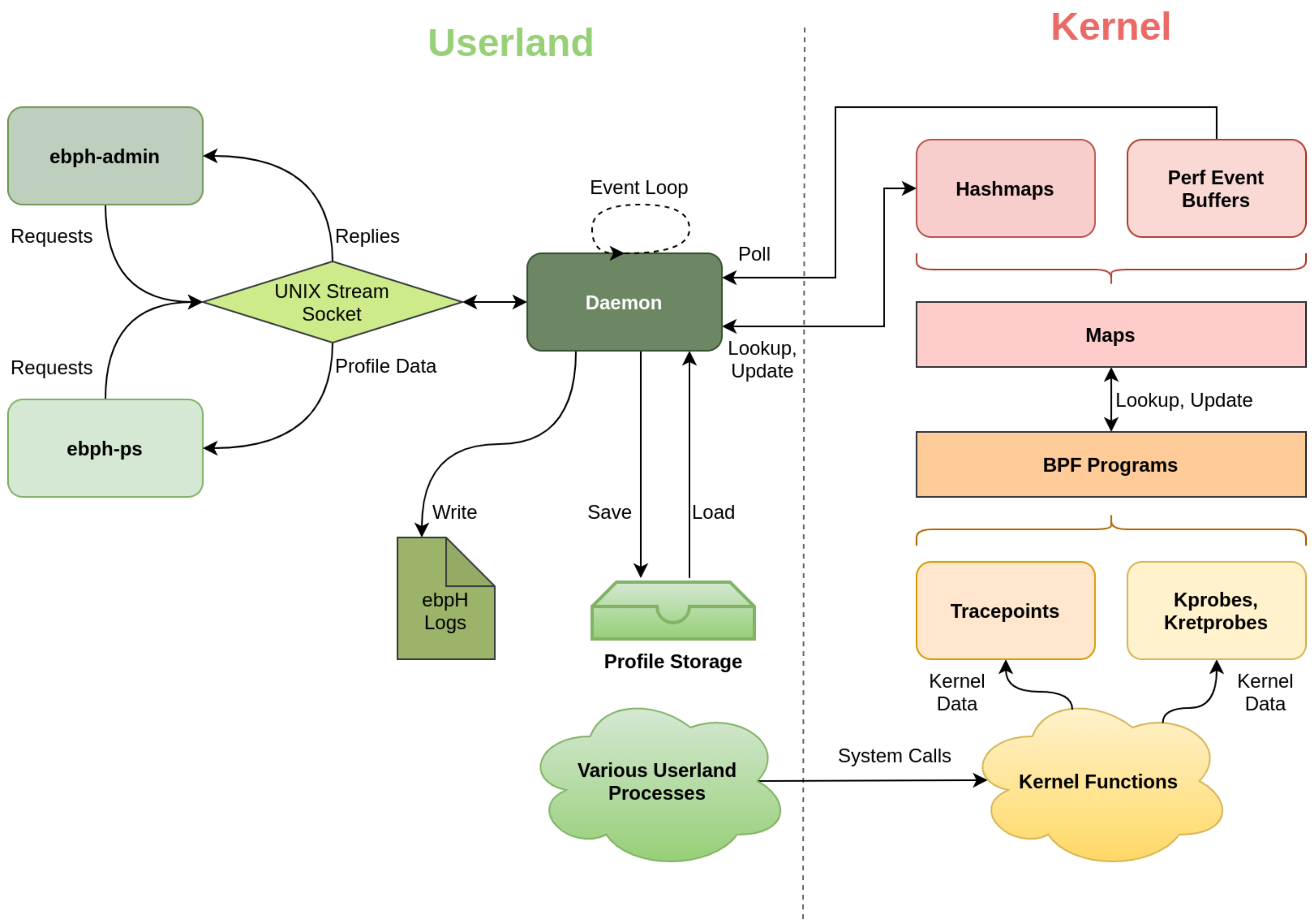


Figure 4: ebpH architecture in a nutshell.

Performance Analysis

- **How does ebpH overhead compare with pH?**
- **Benchmarks**
  - lmbench OS suite (micro)
    - *System call overhead*
    - *Process creation overhead*
    - *IPC overhead (signals, UDS, pipes)*
  - Kernel compilation benchmarks (micro)
    - *How does ebpH perform on real tasks?*
  - bpfbench (macro, ad-hoc)
    - *Real world system call overhead*
    - *Most frequent system calls in practice*

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

lmbench Results

- **Short system calls**
  - getppid(2): 614% overhead
    - *Almost no kernelspace runtime*
  - stat(2): 65% overhead
    - *More significant kernelspace runtime*
- **Long system calls**
  - select(2)
  - As high as 99%
  - But as low as 2%

Figure 5: System call overheads.

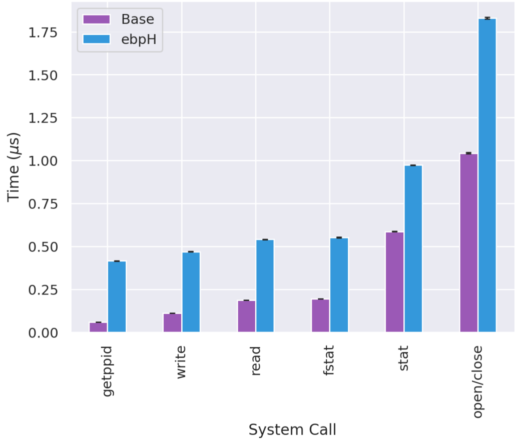
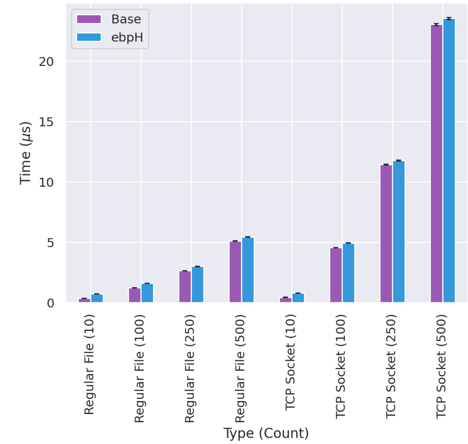


Figure 6: Various select(2) system call overheads.

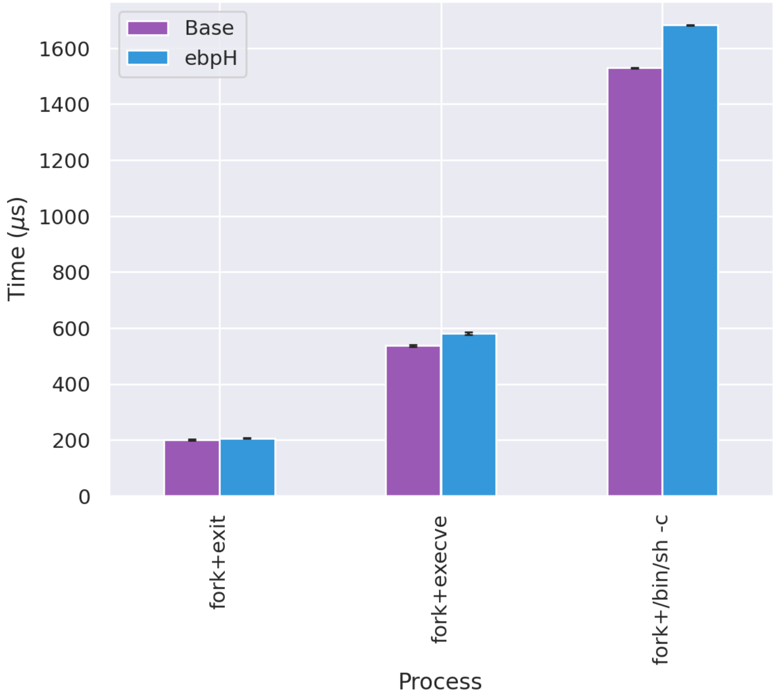


*Error bars show standard error.

lmbench Results

- **Process creation**
  - fork+exit:
    - 2.7% overhead
  - fork+execve:
    - 8.1% overhead
  - fork+/bin/sh -c:
    - 10% overhead

Figure 7: Process creation latency results. Least to most complex.



*Error bars show standard error.

Kernel Compilation Results

Carleton University
Canada's Capital University

- **Kernel compilation**
  - CPU-intensive task
  - A lot of userspace time
  - Still many system calls
    - *Over 176 million*
- **ebpH performs remarkably well here**
  - 10% kernelspace overhead
  - 0.3% userspace overhead
  - under 1% real overhead

Table 3: ebpH kernel compilation overheads. Tests were run using 16 logical cores.

Category	T_{base} (s)	T_{ebpH} (s)	Diff. (s)	% Overhead
System	1525.412 (1.7603)	1687.833 (8.0621)	162.421667	10.647727
User	12333.737 (27.8529)	12370.957 (4.1244)	37.220000	0.301774
Elapsed	915.173 (3.9876)	924.032 (1.1194)	8.858333	0.967940

Table 4: Original pH kernel compilation overheads.

Time Category	Standard (s)	pH (s)	% Increase
user	728.92 (0.74)	733.09 (0.17)	0.57%
system	58.19 (0.80)	80.34 (0.17)	38.06%
elapsed	798.65 (0.87)	825.18 (1.75)	3.32%

*Standard deviations in parentheses.

bpfbench Results

- **Looked at top 20 system calls by count from three datasets**
  - arch (personal use)
  - bronte (idle)
  - homeostasis (production use)
- **Most frequent system calls have acceptable overhead**
  - Anywhere from about 5% to about 150%
- **Idle system reported significantly more overhead than the other two**
  - Lower overhead when it actually matters

Performance Summary

- **ebpH imposes significant overhead on some system calls**
  - But this is not the whole story
    - *Longer system calls means less overhead*
    - *System call overhead ≠ overall impact*
- **Impact on most frequent system calls can be much lower in practice**
- **ebpH does very well on real tasks**
  - In some cases better than the original pH
  - Slowdown is mostly imperceptible in practice

Responding to Attacks with eBPF

- `bpf_signal`
  - Real-time signals from kernelspace (instantaneously)
  - 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

Table 1A: Adding response to ebpH.

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)

Table 1B: Fixing ebpH's memory overhead.

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

What Other Security Problems Can We Solve with eBPF?

- **Anomaly detection**
  - Add more sources of data?
  - No reason to stop at system calls
- **DDoS mitigation**
  - Cloudflare is doing this with eBPF/XDP
- **Increasing visibility of attacks / misuse**
  - ebpH does a bit of this
  - bcc tools are great for this
    - e.g. capable(8), eperm(8), setuids(8), execsnoop(8), etc.

What Other Security Problems Can We Solve with eBPF?

- **Sandboxing?**
  - Externally enforcing seccomp rules with eBPF?
  - bpf_signal could do this easily
- **Name something you want to trace**
  - eBPF can do it
  - And it can do it safely and with excellent performance
- **ebpH is just the beginning**
  - Uses a small fraction of eBPF's capabilities

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

Carleton UNIVERSITY
Canada's Capital University

Some Links

https://github.com/iovisor/bcc

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

https://github.com/willfindlay/ebph

PRs welcome!

Thank you!