wBPF: Safe Kernel Extensibility with WebAssembly

Maximilien Angelo Munz Cura Haibib Kerim Ariel Reid Emma Sudo

Abstract

Custom changes to monolithic kernels often impose a heavy maintenance burden on the broader community. For many years, eBPF was the only solution: check untrusted code with a complex static verifier that occasionally accepts dangerous programs and rejects safe ones. We introduce wBPF, a mechanism to execute WebAssembly (Wasm) programs safely in the kernel by hybridizing static and dynamic analysis. Where eBPF's sole reliance on a static verifier imposes strict restrictions on developers, wBPF's static analysis can fall back on dynamic instrumentation to gracefully exit the program in cases of unsafe memory access and fuel exhaustion. wBPF accepts programs in any language, compiles them to Wasm, performs an initial pass of static analysis, adds dynamic instrumentation where static analysis is inconclusive, transpiles to C, and capitalizes on LLVM's powerful optimizing compiler to generate a kernel module. We demonstrate that wBPF can run complex workloads at near native speeds while giving developers greater flexibility.

1 Introduction

eBPF is the most widely used kernel extension tool today because of its adoption in Linux and Windows: all traffic to Netflix, the Android operating system, or Meta's datacenters passes through an eBPF program at some point. [2] Running code in the kernel is dangerous because unsafe memory accesses could leak information, corrupt data, or cause privilege escalation, and non-terminating programs could hang the entire system. Preventing these problems with only static analysis is intractable without significant restrictions on the programs themselves. wBPF combines static analysis and dynamic instrumentation, accepting a much larger range of programs than eBPF with minimal performance cost. Through wBPF, we demonstrate that it is possible to use WebAssembly as an intermediary bytecode with the same level of safety as eBPF while being more performant on heavy workloads.

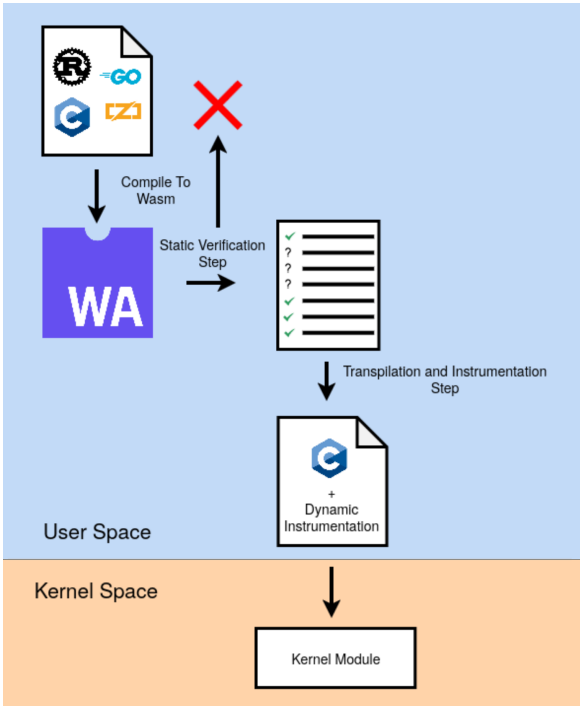

Figure 1: Overview of wBPF's pipeline for sending programs into the kernel.

2 Design and Implementation

Figure 1 presents an overview of the wBPF transformation pipeline. Initially, developers write a program to hook into a point in the kernel. They compile this program into a Wasm binary as input for wBPF. Thus, the program can be written in any programming language with existing compilers for Wasm. Then, wBPF's static verifier rejects identified unsafe memory accesses and divergence, and explicitly annotates expressions where a binary safety determination cannot be proven. Then, in the dynamic instrumentation step, wBPF converts Wasm instructions to C while injecting memory bounds checks and

fuel checks at unprovable memory accesses and termination points. We define termination points as the starting points of Wasm functions or loop instructions (a non-terminating Wasm program must have an infinite recursion in at least one of these places). Finally, wBPF produces a safe version of the program in C that the developer can call in a kernel module, which can then be hooked to a point in the kernel using the appropriate kernel APIs (e.g. `linux/tracepoint.h` or `linux/kprobe.h`).

2.1 Static Verification

While dynamic instrumentation suffices to ensure that desired safety properties are enforced, it introduces additional performance costs. Since eBPF is often used in performance-sensitive applications, achieving near native performance is necessary for wBPF. Therefore, we built a static analyzer for Wasm programs to check similar properties as the eBPF verifier (e.g., memory operations are in-bounds and all loops terminate).

However, in contrast to eBPF, we can also fall back on dynamic instrumentation for unprovable instructions, allowing us to accept a much larger set of programs while minimizing the cost of dynamic instrumentation. Mechanistically, this is accomplished by having the static analysis output a list of offsets for unprovable instructions. Dynamic instrumentation is only inserted for these instructions.

Our static verifier is a symbolic execution engine for Wasm programs that attempts to exhaustively check that possibly-problematic statements are non-problematic. In particular, we want to check that:

1. No operations trap
2. All memory operations are in-bounds
3. All loops eventually terminate

We perform this check by symbolically interpreting the Wasm binary file and exploring all equivalence classes of execution traces.

Since we have a fallback to dynamic instrumentation, there are four possible "verdicts" for all instructions:

- For all inputs, the instruction is safe
- For all inputs, the instruction is unsafe (in which case, the program can be rejected outright)
- There exists an input for which the instruction is unsafe, but inputs similarly exist for which the instruction is safe (in this case, the program can be either wholly rejected, or an assertion can be added to gracefully fail on such inputs, deferring to the developer's intentions)
- None of the above can be proven, so the instruction is dynamically instrumented

For expediency of development, we use the Z3 solver [1]. We check termination by checking the number of times `loop` instructions run (e.g., if a `loop` runs more than N times without its corresponding `end` running, we flag it as not-provably-terminating). Memory bounds checking is performed on concretized addresses. While we currently have no test cases exercising (mutual) recursion, a similar approach would be used.

Symbolic execution in Wasm has a couple of interesting points:

- Branches are easier to handle in Wasm since potential branch targets are restricted to a finite, explicit set of labels (including indirect jumps)
- Analysis is more difficult than in eBPF because we have less control over code generation. A fully general Wasm static analyzer has to work around the codegen of existing Wasm compilers; it cannot restrict its inputs as much as the eBPF verifier can.

In 10 weeks, we developed an analyzer with limited performance. We evaluate the feasibility of using symbolic execution for static analysis through the matrix multiplication and substring-search workloads.

Matrix Multiplication
A 32×32 `int32` matrix multiplication workload takes approximately 28 seconds to dispatch 242,210 instructions. This case is almost branchless and doesn't suffer from the state-forking issues mentioned in the substring-search case below. Instead, 91% of time is spent hashing memory state (as a performance optimization, if the analyzer reaches a previously-reached state, it will stop execution. In this case, the time spent is due to hashing the entire memory state at every step).

Substring-Search
Checking a simple substring-search workload (5-byte needle, 10-byte haystack) took approximately 13-15 seconds to run 814 instructions across 116 state forks. However, 97% of that time is spent on forking the solver state when resolving ambiguous conditional branches. Therefore, we believe that using a specialized solver that incurs less cost per fork or optimizing our usage of Z3 will eliminate this performance overhead.

In both of these cases, there are many opportunities for optimization (underconstrained partial execution of loop bodies, etc.). Thus, we believe it is possible to develop an efficient static analyzer for the verification step of wBPF.

2.2 Dynamic Instrumentation

wBPF's dynamic instrumentation step relies heavily on WebAssembly Binary Toolkit's `wasm2c`. `wasm2c` already emits memory bounds checks for memory accesses, but it does not handle termination checking. Therefore, we add a fuel mechanism to ensure that programs eventually terminate.

During the transpilation from Wasm to C, wBPF's modified `wasm2c` uses the list of unprovable instructions offsets from the static verification step to inject fuel consumption for loops that cannot be proven to terminate and to inject bounds checks for not verifiably safe memory accesses. Then, developers can create custom trap handlers whenever a trap is triggered during execution. `setjmp` and `longjmp` are commonly used to jump back to the host program in the case of a trap, but this functionality is not accessible in the kernel. The easiest approach would be for the trap handler to set an exit flag that signals to the wBPF program to return back to the host program and free the Wasm module.

3 Evaluation

We test the feasibility of wBPF as an eBPF replacement by answering the following questions:

- Does wBPF incur comparable runtime overhead relative to the native baseline?
- Does wBPF deliver end-to-end performance benefits for eBPF-like workloads?

3.1 Testbed

The runtime overhead experiments were run on a machine with a 4th generation Intel Xeon Platinum 8481C processor (4 cores and 8 threads) and 32 GB of RAM, running Ubuntu 25.10 (Linux 6.17.0). All experiments ran directly on Linux. We use `bpftool` to load, attach, and log eBPF programs. wBPF programs and our native baseline both take the form of a Linux loadable kernel module. We use LLVM 20.1.8 with the O3 optimization flag to compile C to Wasm, C to kernel object file (x86), and C to eBPF bytecode. eBPF bytecode is jitted to x86.

3.2 Runtime Overhead

To evaluate wBPF's runtime overhead, we use wBPF to run two microbenchmarks:

- A 32×32 32-bit integer matrix multiplication workload
- A substring-search workload that scans a 20,000-byte buffer for a 40-byte pattern using a naïve sliding-window algorithm

eBPF workloads are typically less computation-intensive than our experimental workloads because the eBPF verifier limits program complexity, and eBPF programs are often designed to run in the kernel fast path. However, these computationally heavy workloads help determine the runtime overhead incurred by each wBPF transformation step. Additionally, there are cases where running computationally intensive workloads in the kernel might be necessary but not possible with eBPF.

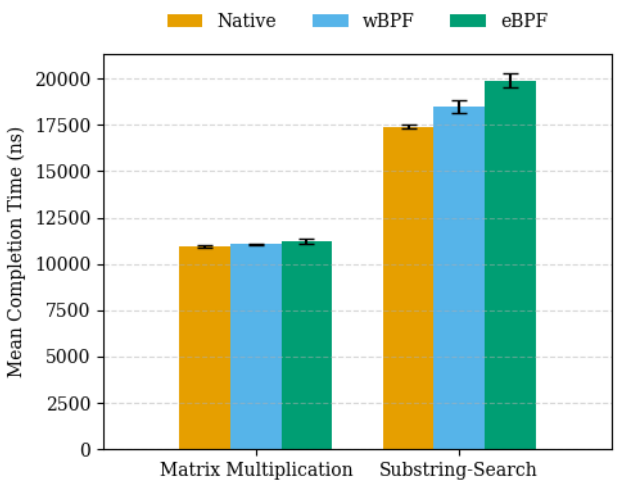

Figure 2: For the matrix multiplication and substring-search workloads, wBPF reaches near native performance and outperforms eBPF.

For example, the substring-search workload would be plausible in a packet filtering scenario.

For each workload, we start with a C function. In our native baseline, the C function is called from a kernel module. In wBPF, the C function passes through the transformation pipeline and the resulting safe C function is called from a kernel module. In both kernel modules, we measure the time taken to perform 100 iterations of the workload. In eBPF, we call the same C function, but we are limited to 50 iterations because of eBPF's 512-byte stack. All three measurements are taken using Linux's `CLOCK_MONOTONIC` clock reference. Figure 2 shows the results of the matrix multiplication and substring-search benchmarks. Note that for both workloads, the static analyzer did not identify any unprovable instructions, so wBPF does not perform any dynamic instrumentation.

With static-dynamic hybridization, wBPF is 0.9% slower than native for the matrix multiplication workload and 6.3% slower than native for the substring-search workload. This is compared to eBPF, which is 2.3% slower than native for the matrix multiplication workload and 14.3% slower than native for the substring-search workload.

We perform an ablation of the previous experiment to determine the overhead added to wBPF by dynamic instrumentation. We compare the performance of wBPF with static verification and without static verification (full dynamic verification). The results are shown in Figure 3. The substring-search workload suffers a 19.6% loss in performance with full explicit instrumentation, which aligns with expected overhead from software bounds and termination checks. In comparison, the matrix multiplication workload suffers a significantly higher 75.8% loss in performance. Upon further investigation, we discovered that the potential branches from the checks

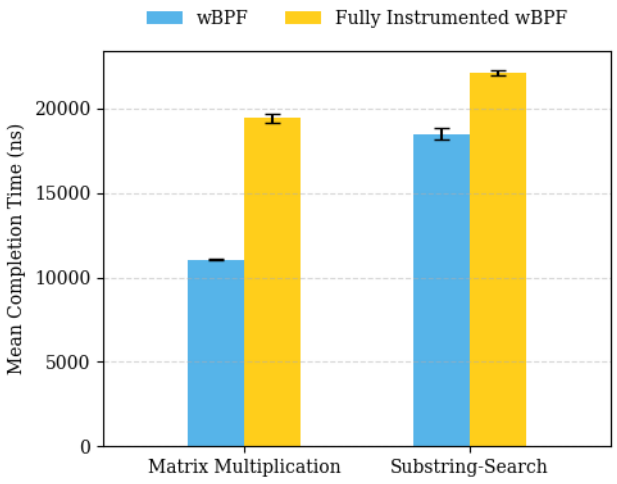

Figure 3: For the matrix multiplication and substring-search workloads, wBPF's static-dynamic hybridization approach results in large performance benefits compared to a fully instrumented version of wBPF.

were preventing the compiler from properly coalescing instructions and interfering with the CPU's branch prediction and caching. From this, we learned that the inclusion of dynamic instrumentation requires more robust communication with the compiler to avoid costly interference.

3.3 End-to-End Performance

We evaluate wBPF's end-to-end performance using a simplified implementation of `syscount`, a tool that counts system calls per process, derived from the BCC project's `libbpf-tools/syscount` observability tool. `syscount` is purely observational and attaches to the `raw_syscalls:sys_enter` tracepoint.

Further, `syscount` operates entirely within Wasm's linear memory, never needing to access arbitrary kernel virtual addresses. Wasm's memory isolation is not fighting the program's memory access needs, avoiding memory architecture mismatch overhead. Choosing `syscount` lets both implementations use the same memory access pattern: on each `sys_enter` event, read pid and process name from the current task struct, construct the composite key `{pid, syscall_nr, comm[16]}`, look up the hash table, and atomically increment the counter, inserting a new entry if the key is not yet present. This benchmark measures the overhead of a Wasm probe versus an eBPF probe, as opposed to how slowly WASM reads kernel memory.

Measurement Methodology

We measure per-probe invocation latency using a tight-loop microbenchmark (`syscall_hammer`). The harness issues $N = 10^6$ `getpid()` system calls and measures the total wall-clock round-trip duration from the `syscall` instruction in userspace to its return, using `clock_gettime(CLOCK_MONOTONIC)`:

```c
struct timespec t0, t1;
clock_gettime(CLOCK_MONOTONIC, &t0);
for (int i = 0; i < N; i++)
    syscall(SYS_getpid);
clock_gettime(CLOCK_MONOTONIC, &t1);

double elapsed_ns = (t1.tv_sec - t0.tv_sec)
    * 1e9
                  + (t1.tv_nsec - t0.tv_nsec
    );
double per_call_ns = elapsed_ns / N;
```

Listing 1: Core measurement loop in `syscall_hammer.c`

Per-call latency is reported as total elapsed time divided by N. `getpid()` is chosen because its kernel path is minimal: it reads `current->tgid` and returns immediately, so essentially all time above the unprobed baseline reflects probe overhead rather than the underlying syscall.

Each `getpid()` invocation causes the kernel to fire the `raw_syscalls:sys_enter` tracepoint, execute the attached probe to completion, and then return to userspace. The wall-clock measurement captures the full probe dispatch path end-to-end:

```
/* userspace */
syscall(SYS_getpid);
    /* kernel: sys_enter tracepoint fires */
    probe_handler();   /* eBPF or wBPF probe
     executes */
    /* kernel: getpid() reads current->tgid,
     returns */
/* userspace: call returns */
```

Listing 2: Probe execution path on each `getpid()` call (pseudocode)

Probe overhead is computed as the difference between the probed and unprobed per-call latency:

$$\text{overhead} = \bar{t}_{\text{probed}} - \bar{t}_{\text{baseline}}$$

This end-to-end measurement, the per-syscall cost a user pays for having the probe attached, is the quantity that matters most for a real deployment. Measuring only probe internals (e.g. with `rdtsc` inside the handler) would exclude probe dispatch costs inherent to each runtime, which are central to this comparison.

We measure four conditions: *baseline* (no probe attached), *eBPF* (libbpf loader, JIT-compiled), *wBPF* (stock `wasm2c`, no fuel instrumentation), and *wBPF+fuel* (fuel-instrumented `wasm2c`). Each condition runs 10 independent trials of $N = 10^6$ calls, preceded by 2 discarded warm-up trials.

	Local (CachyOS 6.18)		GCloud (Ubuntu 6.8)	
Condition	**ns**	**overhead**	**ns**	**overhead**
Baseline	276.97	—	286.71	—
wBPF	380.82	+103.8 ns	362.45	+75.7 ns
eBPF	584.95	+308.0 ns	366.82	+80.1 ns

Table 1: Per-probe latency: local machine vs. Google Cloud. eBPF variance on the local machine was high ($\sigma = 64.85$ ns, bimodal); on GCloud it was low ($\sigma = 3.94$ ns, flat).

Probes are loaded and unloaded between conditions. All measurements are run on an AMD Ryzen 9 5900HS CPU, Linux 6.18.8 (CachyOS), x86-64, with LLVM/clang as the kernel build compiler.

Reproduction on Google Cloud

To evaluate the results beyond an individual laptop, we used a Google Cloud `n2-standard-4` instance running Ubuntu 22.04 with the HWE kernel (Linux 6.8.0-1048-gcp) and reproduced the baseline, eBPF, and wBPF benchmarks. The fuel-instrumented wBPF was omitted due to build toolchain constraints with the Google Cloud instance's clang version.

Results are summarized in Table 1. Two differences are notable. First, absolute baseline latency is slightly higher on GCloud (286.7 ns vs. 277.0 ns), consistent with virtualization overhead and a different CPU microarchitecture. Second, eBPF overhead on GCloud is dramatically lower (+80.1 ns, +27.9%) compared to the local laptop (+308.0 ns, +111.2%), making eBPF and wBPF are statistically indistinguishable on GCloud (366.8 ns vs. 362.4 ns, $\Delta = 4.4$ ns, within one standard deviation of each other).

The convergence of eBPF and wBPF on GCloud is perhaps explained by the absence of BPF map contention. The GCloud instance is freshly provisioned with minimal background processes; the `BPF_MAP_TYPE_HASH` table never fills beyond a few entries, so hash chains remain short and lock contention is negligible. On the local development machine, background system activity populates the map with entries from dozens of processes, growing hash chains and increasing lookup latency. This is the likely cause of the bimodal distribution and high variance visible in Figure 6.

We present the results of the local laptop as the primary results. They represent a realistic deployment scenario rather than an artificially quiet environment where only the probe and benchmark process exist. On an idle, single-purpose machine eBPF and wBPF converge, under realistic system load the gap widens.

Results

Results are summarized in Table 2 and Figures 4–6. All figures plot 10 trials of $N = 10^6$ `getpid()` syscalls per condition on Linux 6.18.8 (CachyOS), x86-64.

Condition	**Mean (ns)**	**σ (ns)**	**vs. base**	**vs. wBPF**
Baseline	276.97	3.69	—	—
wBPF	380.82	1.52	+103.8 (+37%)	—
wBPF+fuel	400.40	12.59	+123.4 (+44%)	+19.6 (+19%)
eBPF	584.95	64.85	+308.0 (+111%)	+204.1 (+197%)

Table 2: Per-probe invocation latency. Linux 6.18.8 (CachyOS), x86-64, AMD Ryzen 9 5900HS. 10 trials × 10^6 `getpid()` calls.

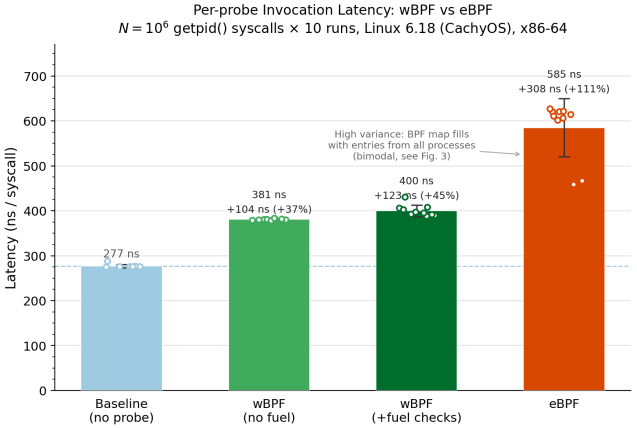

Figure 4: Absolute per-probe invocation latency for all four conditions. Bar height is the mean over 10 trials; error bars show one standard deviation. Individual per-trial measurements are overlaid as jittered points. eBPF exhibits high variance ($\sigma = 64.85$ ns) driven by BPF hash map contention (see Figure 6); wBPF and wBPF+fuel are highly consistent ($\sigma \leq 12.6$ ns).

wBPF adds 103.8 ns of overhead above the unprobed baseline, a 37.5% increase in per-call cost. wBPF+fuel adds an additional 19.6 ns for the three fuel checkpoints, bringing total overhead to 123.4 ns (+44.6%). eBPF adds 308.0 ns, 2.97× the overhead of wBPF and 2.50× the overhead of wBPF+fuel.

wBPF and wBPF+fuel exhibit low run-to-run variance ($\sigma = 1.52$ ns and 12.59 ns respectively). eBPF exhibits substantially higher variance ($\sigma = 64.85$ ns). Runs 1–2 completed at approximately 463 ns; runs 3–10 settled around 615 ns.

The 103.8 ns overhead above the bare syscall cost represents the combined cost of probe dispatch, software memory bounds checks on all WASM linear memory accesses, and five host function calls (two for process identity, two for hash table operations, one atomic increment).

Fuel instrumentation adds negligible overhead. The 19.6 ns marginal cost of wBPF+fuel over wBPF represents three `wasm_rt_consume_fuel()` calls. This is 15.9% of total wBPF+fuel overhead, and the fuel mechanism provides

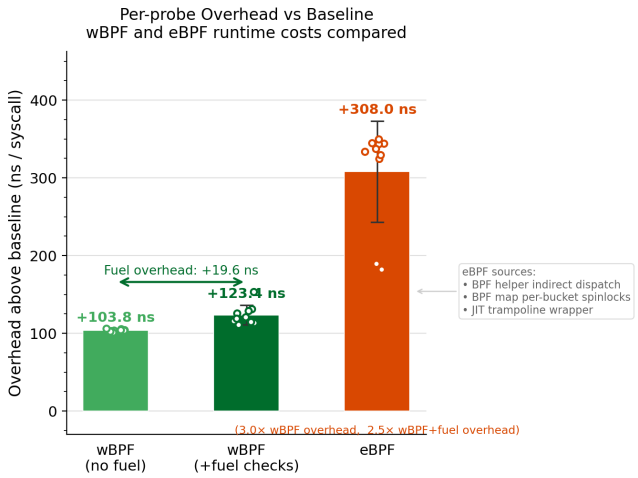

Figure 5: Per-probe overhead above the unprobed baseline. Bars reflect only the cost attributable to the probe runtime. The double-headed arrow shows the marginal cost of fuel instrumentation (+19.6 ns): three `wasm_rt_consume_fuel()` calls at function entry and two conditional branches. eBPF overhead (+308.0 ns) is 3.0× that of wBPF (+103.8 ns) and 2.5× that of wBPF+fuel (+123.4 ns).

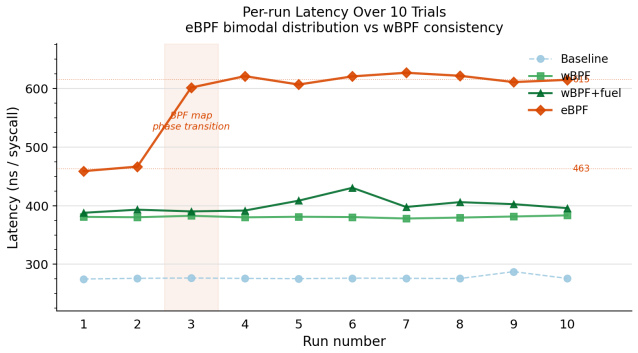

Figure 6: Per-trial latency over 10 sequential trials. The eBPF series (orange) shows a clear phase transition between trials 2 and 3: latency jumps from ≈463 ns to ≈615 ns and remains elevated. This step change reflects the `BPF_MAP_TYPE_HASH` table crossing a chain-length threshold as background processes populate it during the measurement session. wBPF and wBPF+fuel remain flat across all trials. The baseline is stable at ≈277 ns throughout.

a formal upper bound on handler execution time. Bounded execution is a strong safety property costing roughly 5% of the total syscall round-trip time above wBPF without fuel for this benchmark.

eBPF overhead is higher and less stable for this workload. The 308.0 ns overhead has two primary causes. First, eBPF helpers are dispatched through an indirect call table; each of the four helpers requires an indirect branch rather than the direct call wBPF makes to its host functions. Second, the `BPF_MAP_TYPE_HASH` map accumulates entries from all system processes throughout the measurement session. As background processes populate the map, hash chains grow and per-bucket lock contention increases. wBPF also accumulates entries in its kernel hash table, but its global `spin_lock_irqsave` dominates probe cost and scales with concurrent CPU contention (bounded by core count) rather than with total accumulated entries (unbounded). The GCloud result confirms this: on a quiet VM with a nearly empty map, eBPF and wBPF converge (366.8 ns vs. 362.5 ns). The phase transition in Figure 6 marks the local map crossing a chain-length threshold where linear traversal becomes the bottleneck.

Interpreting the comparison fairly. The overhead gap should not be interpreted as an indictment of eBPF's design. The comparison demonstrates that wBPF's hybrid model can achieve a lower and more predictable overhead for observational programs in this class. It does not claim wBPF is universally faster.

4 Future Work

One possible future direction that we are interested in is modifying wBPF's verification step to involve an automated Isabelle proof of termination and memory safety. We imagine a world where a safety policy can be defined using a lightweight domain-specific language so that Isabelle can generate a proof that arbitrary programs satisfy that policy. This proof can then be attached to the wBPF kernel module so that the kernel can quickly verify the program's safety.

Additional possible improvements include:

- Improving the static verifier and moving it into the kernel
- Exploring alternate methods of transforming Wasm code into natively executable binaries

5 Conclusion

eBPF is a flawed solution to the important task of kernel extension. In this paper, we demonstrated that wBPF is a suitable replacement for eBPF that allows developers to hook a more diverse set of programs to the kernel with minimal performance loss. This work extends beyond kernel extensions. wBPF shows that it is possible to enforce a safety policy on programs written in any programming language without impacting the developer experience.

References

[1] De Moura, L., and Bjørner, N. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis*

of Systems (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, p. 337–340.

[2] EBPF FOUNDATION. The state of ebpf.