Evaluation of tail call costs in eBPF

Clement Joly
`clement@cloudflare.com`

François Serman
`fserman@cloudflare.com`

August 24, 2020

Abstract

This paper compares the performance of tail calls between eBPF programs before and after the optimizations introduced to mitigate spectre flaws. This was carried out for two kernel versions: 5.4 and 5.5. The latter introduces a performance optimization which removes retpoline overhead whenever possible.

Two experiments were carried out: the first one uses in-kernel testing `BPF_PROG_TEST_RUN` and the second uses kprobes and network namespaces. The conditions to trigger the optimization from kernel 5.5 were met in both cases, resulting in a drop of the cost of one tail call from 20-30 ns to less than 10 ns.

1 Introduction

Each new hardware generation brings improvements: new architecture paradigms, new features, more cache, more cores, *etc.* Software needs to be updated to leverage those new capabilities in order to introduce new features or improve performance. But because software is more flexible than hardware (*i.e* easier to update), it is sometimes used to fix hardware bugs. This was the case for recent hardware bugs known as meltdown and spectre.

Spectre is the name given to a serie of vulnerabilities leveraging hardware bugs, present on most CPU (Intel, AMD, ARM). After a 6 months embargo -allowing major operating systems to implement software mitigation-, it was publicly discosed in early 2018 [7]. Back then, the performance hit was estimated between 10-25%. The first workaround was to disable speculative execution in the BIOS, waiting for micro-codes to be published. A more general approach called *retpoline* was detailed by Google[9]. This software construct allows indirect branches to be isolated from speculative execution. It's named retpoline because it's similar to the "trampoline" technique, but a ret instruction is used to unstack the target from the stack. This mechanism was also applied to eBPF tail-calls, introducing a performance regression. Recently, an optimization was merged into the Linux kernel (in v5.5) to avoid retpoline when possible.

This paper presents the result the performance gap between those two versions: 5.4 using retpoline for tail calls, and 5.5 converting tail calls to direct calls when the verifier can ensure that a given index is constant from all program paths. Section 2 summarizes the speculative execution issue and existing counter-measures. Section 3 presents the benchmarks, and Section 4 outlines the results. In Section 5, we discuss those numbers and the expectations.

2 Context

In 2018, researchers from Google Project Zero have released a serie of side-channels attacks called meltdown and spectre[7, 5]. While meltdown allows arbitrary memory-reads (actually *melting* memory isolation provided by the hardware), spectre applies timing attack to the branch-prediction machinery of modern out-of-order executing processors called speculative execution. In a nutshell, modern CPUs execute instructions before knowing if it will be required. Without this mechanism, the processor would need to wait for prior instructions to be resolved before executing subsequent ones. The pre-computed results may be discarded if the instructions were not needed after all. In that "worst case scenario", the processor will execute the correct instruction path. But usually, the predictions are correct so the pre-computed state is commited, resulting in a higher performance and hidden latency. While speculative operations do not affect the architectural state of the processor, they can affect the microarchitectural state, such as information stored in TLBs and caches. Observing those microarchitectural state can lead to information leakage.

From a hardware perspective, the mitigation is to disable speculative execution (via BIOS/EFI settings, or micro-instructions loading).
The alternative involves software modifications; both in the kernel and in user-space applications. For instance, compilers had to be modified to add memory serialization instruction (Intel advocates `lfence`) when required, and implement the retpoline technique. JIT engine also had to be modified. `a493a87` introduced retpoline for indirect eBPF calls. Instead of reading the target location of the next program from a BPF "programs" map, a indirection layer was introduced to have the new target stored on the stack, and prevent speculative execution through the use memory fence. While some of this workaround can be considered negligible for general-purpose tasks, the Linux kernel community in general has since been obsessed with avoiding indirect calls in fast-path code whenever possible. The retpoline construction has a detrimental effect on performance. Because this patch affected the code generated by the eBPF VM, tail calls where slowed down. However, in some cases, speculative execution is not dangerous as it is not known to be usable to exploit security vulnerabilities. As a result, if those cases were treated differently, they could recover part of the performance lost due to the mitigation. One such a case is tail call known statically at compile time to be fixed [3, 4]. For instance, a given call will always read the address of the program in the fist cell of the array of the map. It does not matter whether the address in this cell is fixed or not, as these variations will be accounted for later.

In this particular but fairly common case, a patch was proposed to replace the assembly code emitted by the eBPF VM by a version without a retpoline. The virtual machine takes the value in the cell into account to execute different instruction on the CPU. If the cell is empty, a `nop` instruction is emitted. This instruction does nothing and thus has a very low cost. If however there is a value in this cell of the map, then a direct jump is written to this value. And these instructions are updated every time the value of the cell in the map changes. It is worth noting that all the previously described mechanism relies on the fact that the same cell of the map is used for the tail call. This property is identified by the verifier before loading the program. Similar improvements in some access to maps were found to result in 15 % increase in performance [6]. One the aim of the internship would be to evaluate if this improvement has a similar effect. The following section presents the benchmark we have selected to identify the performance improvement brought by kernel later than 5.5.

3 Performance case study

We focus on specific program used at Cloudflare: L4Drop [2]. It is composed of various eBPF programs enforcing mitigation rules. There programs are called *ruleset* and are linked by a tail call. To evaluate time spent in eBPF programs, two benchmarks were used.

3.1 Benchmarks

After outlining the desired measurements in the specification, it becomes possible to create the relevant benchmarks. It particular, running the measurements was automated as much as possible.

3.1.1 Benchmark 1

This first benchmark uses the benchmarking facilities of the Linux kernel (Namely BPF_PROG_TEST_RUN [1]). This benchmark runs a given XDP program 10000 times over the same packet and returns a few measures, described below. In the experiment, this was repeated 36 times. Packets are based on a sample of previous attacks. Compared to the second benchmark described in Section 3.1.2, CPU time consumption figures are stable, but it doesn't really reflects production traffic (which is more diverse).

CPU Time and Latency The fist measure is the time used to run the eBPF program on each packet. This is collected for each packet and average along with quantiles and median are reported. It is worth noting that the eBPF code is not interrupted, as it is running inside the kernel, in the eBPF virtual machine. This CPU time hence gives us the latency introduced by the l4drop system. For this measure, standard deviation is quite low, usually around $\pm 1\%$.

Throughput As the size of a given packet is known, throughput can be deduced from the previous measure, on each packet. The figures are similarly aggregated in an array. Due to the variations in size of the packet and considering that the execution time is quite low (a few dozen nanoseconds) and varies a bit when measured on an individual packet, the standard deviation is very high (usually around ± 150 %.). This measure is thus not used and not reported. The results from the second benchmark are used instead, see Section 3.1.2

Return Code The kernel facilities give the return code of the XDP program. This result is used to ensure we only collect measures like CPU time for packets going through the whole chain of rulesets. Indeed, time for dropping packets can correspond in a different work performed and would thus skew the final results. For instance, if we have three rulesets and a packet is dropped by the second ruleset, code from the third ruleset will not be executed, as dropping a packet takes effect immediately.

3.1.2 Benchmark 2

As discussed in Section 3.1.1, the first benchmark doesn't properly reflects production traffic. The second one tends to replicate production conditions. It surfaces mostly the same measures and thus can be compared to the first one. We observed less stable results, so we used this benchmark to ensure that the results were consistant with the first one. Creating this benchmark turned out to be more challenging, and we faced multiple obstacles; some are described below.

CPU Time and Latency Various methods were attempted to get execution time for the XDP programs in XDPD, including l4drop.

eBPF Map Inside Programs A fist idea was to store the the current time in nanoseconds when entering an eBPF program and then substract this time to the current time at the end of the program. This would then be stored in a eBPF map, accessible in from user space programs, where the measurement could be gathered. Two maps can be used, one accumulating time, the other counting the number of iterations. It is thus possible to extract average time of execution for the part of the program between the two calls to `ktime_get_ns`. It is worth noting that these maps are `PER_CPU`, so that mesurement is accurate even with multiple programs running concurrently on various CPU cores.

The advantage of this approach is that it is relatively straightforward to implement on a test eBPF program. It is also fairly reliable because the program cannot run without the measure-

ment being performed.
The drawbacks of this approach is that it cannot account for tail calls without significant adaptations, in particular to share a map between various programs. This would result in a loss in reliability. It also introduce a significant overhead and production programs tend to have various sections to log passed or dropped packet, characteristics, which results in additional complexity to make sure to update all values in all cases. All of this would have required significant changes to the existing l4drop code base. As a result, a small program similar using maps as described in Section eBPF Map Inside Programs was used to test another method and ensure its accuracy.

KProbes In order to gather insights on a running XDP program, we have choosen KProbes. These are probes that can be attached mainly to functions in the kernel. We had to find a "good candidate" to probe: specific functions (such as `bpf_prog_run_xdp()` `veth_xdp_rcv_skb()` or `veth_xdp_rcv_one()`) are usually inlined and cannot be kprobed. On the other hand, higher level functions are not, but there's more boiler plate code, so the results are less accurate. For this benchmark, we have placed probes on the entry and exit of `veth_poll()`. We are thus getting an estimation of the runtime of all the XDP programs (the sampler, l4drop and l4lb), along with some constant factor, due to the surrounding code. This tend to make the numbers less stable and the measure is less direct than with the first benchmark. These results were nonetheless consistant with what we got from the test program, which gave us confidence in this measurement.
In the future, as new versions of the Linux kernel get adopted more widely, the method exposed in Section 5.2 could be used to get better precision and some complementary informations.

Throughput In this second benchmark, two Linux namespaces are used, a *receiving* and *sending* namespace. The receiving namespace contains a server of the iperf [?] program, along with XDPD and thus, l4drop. In the sending namespace runs an iperf client. Iperf is a speed test tool, which generates traffic between a server and a client and reports the throughput. Even if throughput number are not always stable enough to detect small variations of performance between programs, medium or big variations are still accurately accounted for.

Return Code Return code could not be checked directly in the environment similar to production. Yet, they could be checked as dropped packets are reported by the monitoring facilities in XDPD. Also, dropping packets between the iperf client and server results in errors from this program. As a result, it is possible to make sure that no packets are dropped and that the measured values relate to the process of going through the whole pipeline of XDP programs.

4 Results

eBPF programs have the ability to tail call from one another, i.e. the execution of one program can stop at some point and continue in another eBPF program. The first improvement to be measured with our tools is the performance benefit brought by Linux kernel version 5.5. This also gives an estimation of the cost of a tail call across various kernel versions. This will be a valuable insight in the rest of our work.

4.1 Benchmark 1

For these measurements, lower is better.

4.1.1 One Rule per Ruleset

A first experiment uses rulesets of one rule each. The impact of tail calls is important, as they have a significant share of the total execution time.

On Figure 1, the purple, red and green bars represent median CPU time for a ruleset of one rule. As one can see, even though the three rules are slightly different, the execution time for each is quite consistent and stable, even across different kernel versions. The orange bar corresponds to 20 rulesets of one rule, tail calling into one another. Finally, the blue bar shows median CPU time for the same rulesets as on the orange bar, but merged into one eBPF program, without tail calls. As it was verified that the number of instructions between the orange bar and the blue bar are comparable, the difference between the two bars is exactly the time spent in the 19 tail calls.

From data points in Figure 1, it can be inferred that the cost of tail calls (the difference between the merged rulesets and the tail called rulesets, i.e. between the blue and the orange bar) is of 497 nanoseconds (ns) on the metal named testM7 with kernel 5.4 while it is reduced to 101 ns with kernel 5.5. As there are 19 tail calls, the average cost of one tail call drops to 5 ns from 26 ns, an improvement of about 81 %.

Metal testM8 has more CPU cores, with a higher frequency, as seen on Table 1. Consequently, tail calls were taking less time, at about 25 ns each. Nevertheless, the cost of one tail call drops to 5 ns with the new kernel version, an improvement of about 80 %.

4.1.2 300 Rules per Ruleset

A second experiment uses rulesets of 300 rules each. Tail calls have a smaller share of the total execution time, compared the previous experiment.

On Figure 2, the green bar represents median CPU time for a ruleset of 300 rules. If we compare, on testM8, the time spent in a ruleset of 300 rules (377 ns) and the time spent in a ruleset of one rule (29 ns), we infer that the cost of the 199 rules account for 348 ns, so the average cost of this particular type of rule is less than 2 ns. It is worth noting that this rule is not the most complicated, although it is a common one in production.

The orange bar on Figure 2 corresponds to two rulesets of 300 rules, tail calling into one another. Finally, the blue bar shows median CPU time for the same rulesets as on the orange bar, but merged into one eBPF program, without tail calls. As it was verified that the number of instructions between the orange bar and the blue bar are comparable, the difference between these two bars is exactly the time spent in the unique tail call between these two rulesets.

It can be inferred from data of the graph of Figure 2 that kernel 5.5 brings shorter runtime compared to version 5.4, although exact duration of the tail call is not always visible. For instance, on testM7, the scenario with a tail call requires 1268 ns with kernel 5.5 while it lasts for 1333 ns with kernel version 5.4. The same happens on the machine named testM8.

4.2 Benchmark 2

4.2.1 CPU Time

For these measurements, lower is better, as it is the part of the second benchmark were CPU time is discussed. Conditions are the same as the first benchmark.

General Observations The kernel probe used (`veth_poll`) returns an average time for the probed function to run. This includes the time to run the XDP code as well as the code around it in the function. These numbers are still quite stable, with a typical standard deviation around 2 %. Plus, due to the conditional code in the function, execution time tend to be centered around two values. As a result, graphs like on Figure 3 or on Figure 4 feature three groups of bars. The first group from the left corresponds to the average for values inferior to 16 000 ns (hence the name of these data, "KProbeLess16K"); the second group, values superior to this threshold and the last group is an average over the preceding values as a whole.

Machine name	**Machine from the laboratory?**	**CPU**	**Number of core (logical)**	**Frequency (GHz)**	**Maximum frequency (GHz)**	**RAM (GB)**
testM7	Yes	Intel(R) Xeon(R) Silver 4116	48	2.10	3	188
testM8	Yes	Intel(R) Xeon(R) Platinum 6162	96	1.90	3.5	188

Table 1: Machines used to run the benchmarks.

One Rule per Ruleset This is in line with the conclusions of the same paragraph for the first benchmark: tail calls are less costly with the newer kernel version. The difference between the two kernel versions is statistically significant with the T-Test: over the dataset “KProbeTotal”, the gain with kernel 5.5 is of about 20 % on testM8 and of 19 % on testM7. In both cases, the rounded p reported is 0.000.

Discrepancy Between the Two Benchmarks As there is a constant code running around the function actually executing the XDP code, the total time spent in tail calls accounts for a lower share of the total and gains over tail calls have a lower impact as a result. However, for kernel version 5.5 on testM7 in the first benchmark, XDP code take roughly 153 ns and in the second benchmark, average total time for “KProbeTotal” is roughly 19069 ns. In proportion, this should result in a much lower improvement due to tail calls. Various explanations come to mind. First, between the two kernel versions, some parts of the code could have changed resulting in performance improvements. Second, the actual time to run the eBPF code is bigger than in the first benchmark. Third, the tail call gain is bigger than in the first benchmark. This is further discussed in Section 5.2.

300 Rules per Ruleset This is again in line with the conclusions of the same paragraph for the first benchmark: tail calls are less costly with the newer kernel version. With the same statistical significance test and a rounded p of 0.000 still, a 1.4 % improvement is reported on testM7 and a 3.5 % improvement on testM8.

4.2.2 Throughput

For these measurements, higher is better, as it is the part of the second benchmark were throughput is discussed. Improvements with the new kernel version are similar to the ones seen on CPU time for this benchmark.

One Rule per Ruleset In this case, a significant drop in the difference between the case where rulesets are merged (the blue bar on Figure 5) and the case where tail calls are placed between rulesets (the orange bar on Figure 5). In other words the 20 XDP programs with tail calls (orange bar) run faster. With the same significance test, where p roughly equal 0.000, throughput is increased by 17 % on testM7 and by 23 % on testM8.

300 Rules per Ruleset In this case, the drop in the difference is slight, as one can see by comparing the blue bar on Figure 6 and the orange bar on Figure 6. In other words the XDP program runs slightly faster and, with the same signifiance test, where p roughly equal 0.000, throughput is slightly increased by 0.7 % on testM7 and 3 % on testM8.

5 Discussion

5.1 Impact

In production at Cloudflare, we typically have 4 tail calls in our XDP stack. Thus, sparing 20 ns per tail calls results in a gain of 80 ns per packet. During a DDoS attack peaking at 754 million packets per second [8], the improvement discussed in this article results in about one minute of computation saved accross Cloudflare's infrastructure for every second of the peak of the attack.

5.2 Improvements

To tackle the issues encountered, various technics could be used as a complement or replacement of what was done here. For instance, a recent addition to bpftool is the profile subcommand. It gives more detailed insights and could be used to get more precise results and hopefully solve problems encountered in Section 4.2.1. Perf was not explored at first because the trace subcommand was deemed to be too much overhead and record, to be insufficient. Deeper investigations may still be required. A final idea to explore is enabling stats on BPF programs.

6 Conclusion

In this paper, we have summarised the landscape of the spectre mitigation in XDP subsystem. Then, we have compared the performance measured between Linux kernel 5.4 and 5.4: the former includes retpoline mitigation, and the latter tries to replace them with direct calls. Those experiments have been carried out using two benchmarks. The first benchmark was used to measure the duration of tail-calls: we compared the execution time of one program doing several tail-calls with a program whose rule-set have been merged before, thus removing the tail-calls. This has demonstrated that: i. each tail call is 80 % faster if the retpolines are converted into direct calls and ii. after this optimization, the cost per tail-call is about 5 ns.
The second benchmark was used to reproduce production traffic; over a chain of two programs relatively big (as seen in production), with only one tail call between them, the gain is about 5 %.

7 Acknowledgements

The authors would like to thanks the Linux Plumbers conference for their efforts in keeping the conference during those though times. We would also like to thanks our colleagues at Cloudflare for their help and support; and the kernel "de facto" team for their reviews.

References

[1] Alexei, Starovoitov. bpf: program testing framework. `https://lwn.net/Articles/718784/`, 2017. [Online; accessed 2020-07-24].

[2] Arthur, Fabre. L4drop: Xdp ddos mitigations. `https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/`, 2018. [Online; accessed 2020-06-06].

[3] <'Cilium, Inc'>. Cilium 1.7: Hubble ui, cluster-wide network policies, ebpf-based direct server return, tls visibility, new ebpf go library, ... `https://cilium.io/blog/2020/02/18/cilium-17/`, 2020. [Online; accessed 2020-08-20].

[4] Daniel, Borkmann. [patch bpf-next v2 0/8] optimize bpf tail calls for direct jumps. `https://lwn.net/ml/netdev/cover.1574452833.git.daniel@iogearbox.net/`, 2019. [Online; accessed 2020-08-13].

[5] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[6] Nikita, V. Shirokov. Xdp: 1.5 years in production. evolution and lessons learned. `http://vger.kernel.org/lpc-networking2018.html#session-10`, 2018. [Online; accessed 2020-06-21].

[7] Graz University of Technology. Meltdown and spectre - vulnerabilities in modern computers leak passwords and sensitive data. `https://spectreattack.com/`, 2018. [Online; accessed 2020-08-20].

[8] Omer, Yoachimik. Mitigating a 754 million pps ddos attack automatically. `https://blog.cloudflare.com/mitigating-a-754-million-pps-ddos-attack-automatically/`, 2020. [Online; accessed 2020-08-17].

[9] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. *URL https://support. google. com/faqs/answer/7625886*, 2018.

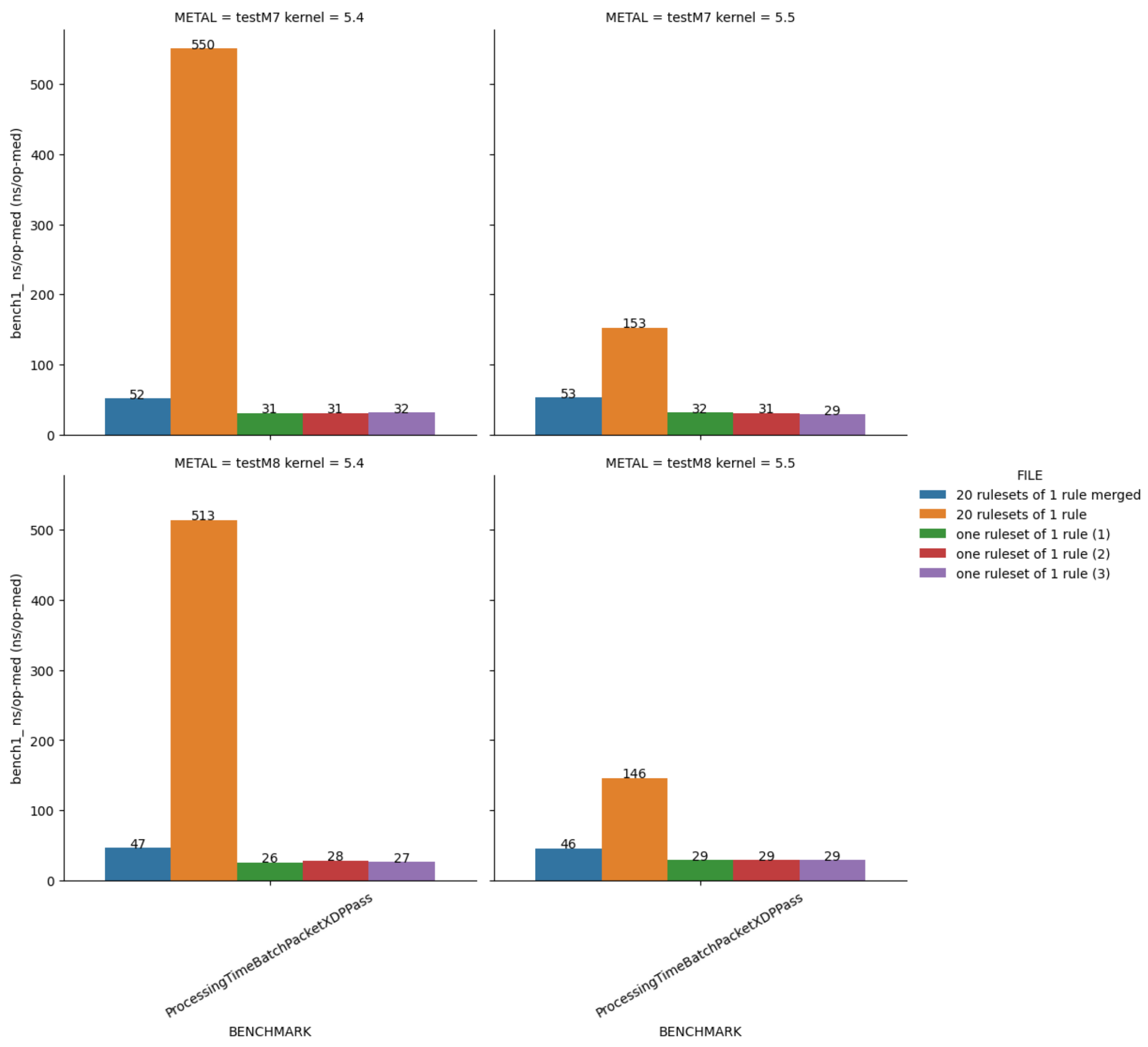

Figure 1: Benchmark 1: CPU time measured across various metals and various kernel versions, using rulesets of one rule.

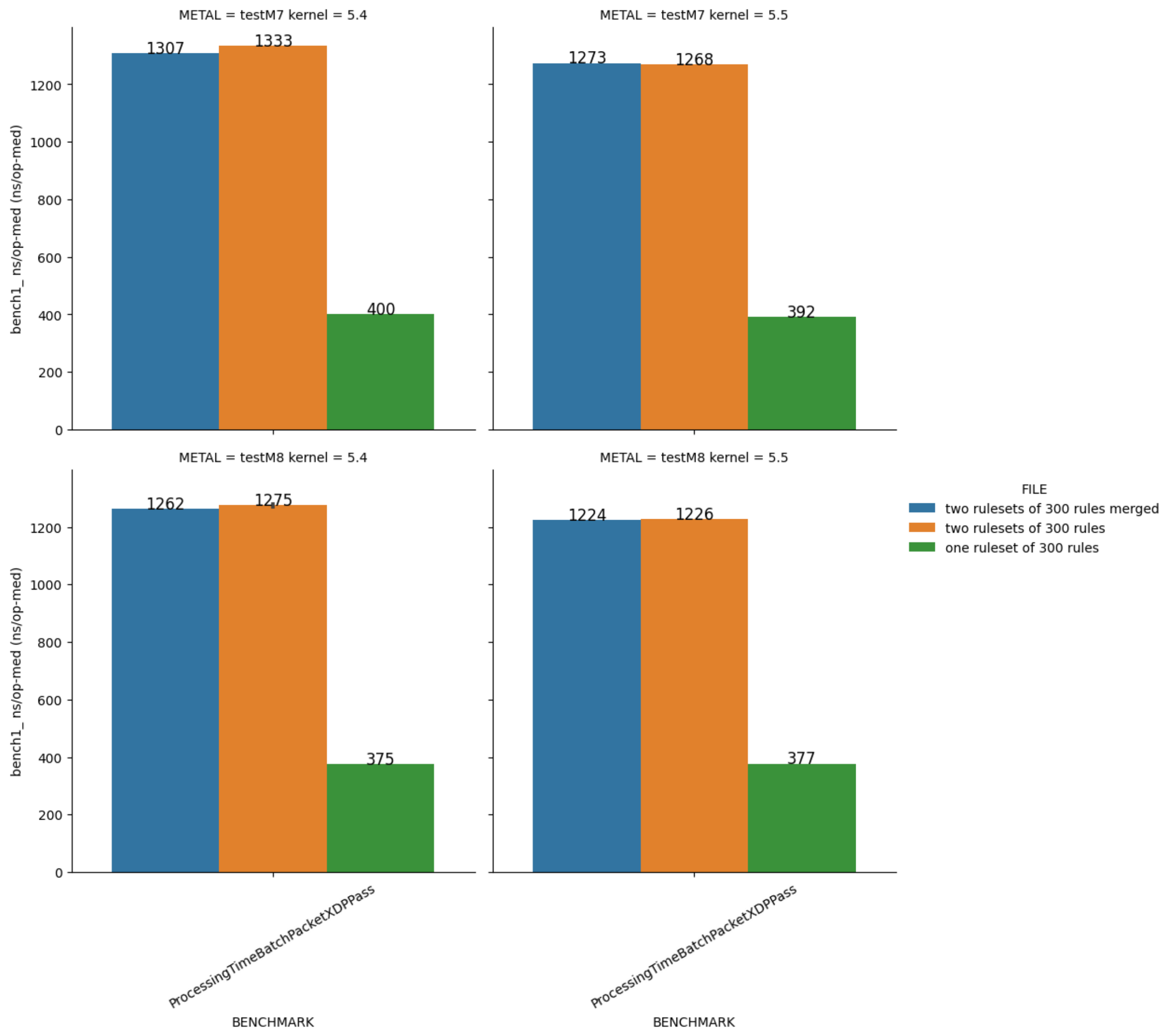

Figure 2: Benchmark 1: CPU time measured across various metals and various kernel versions, using rulesets of 300 rules.

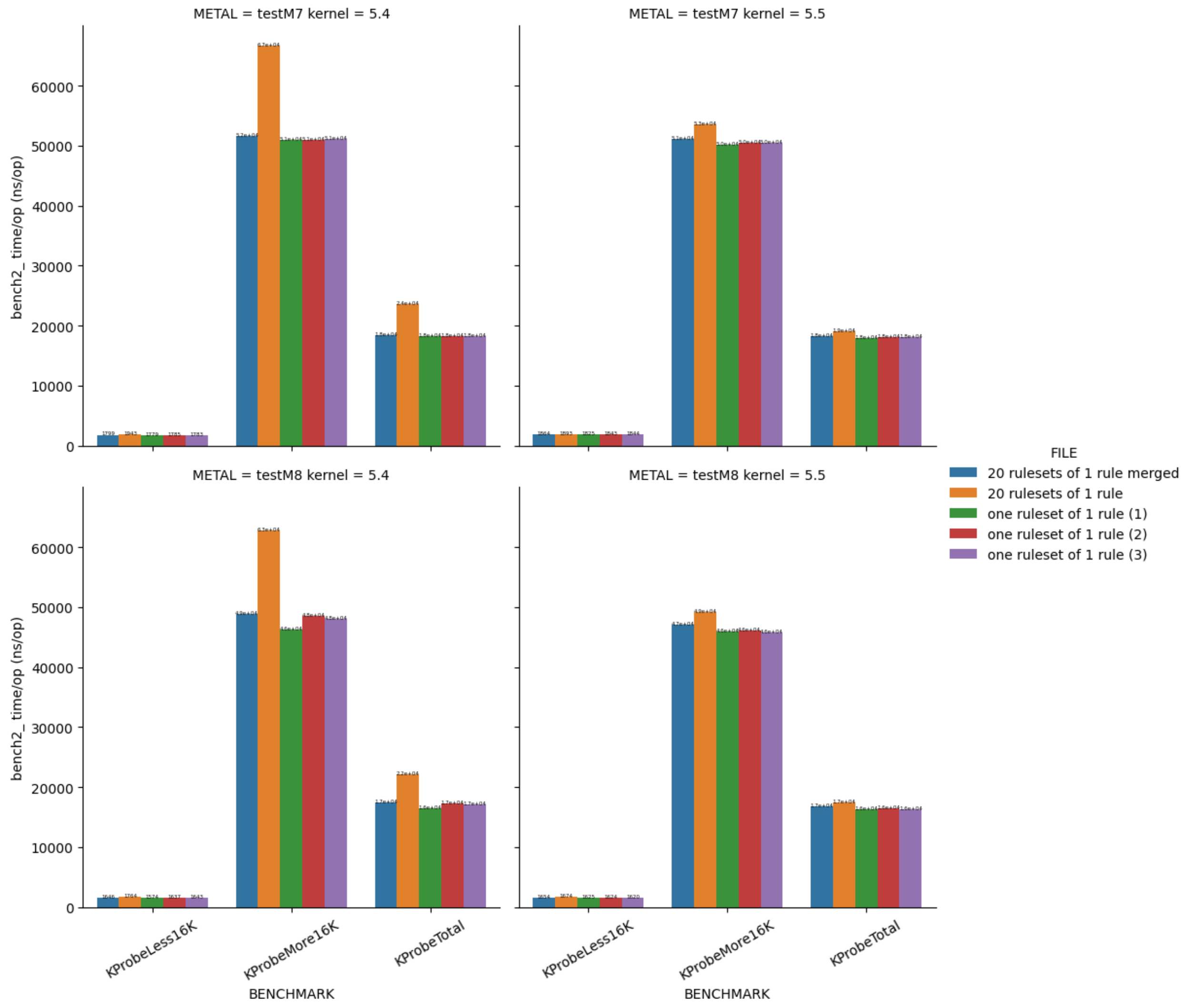

Figure 3: Benchmark 2: CPU time measured across various metals and various kernel versions, using rulesets of one rule.

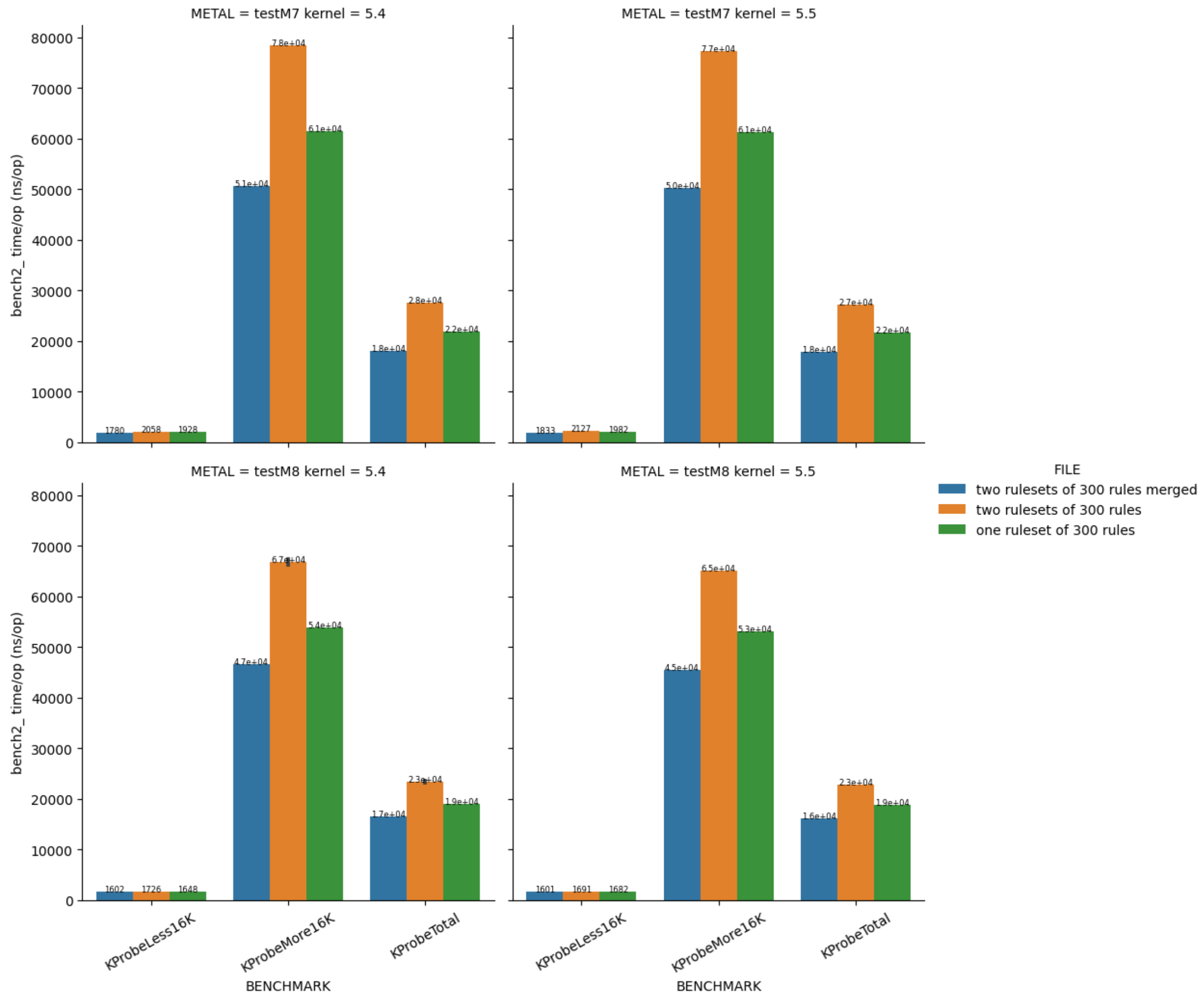

Figure 4: Benchmark 2: CPU time measured across various metals and various kernel versions, using rulesets of 300 rules.

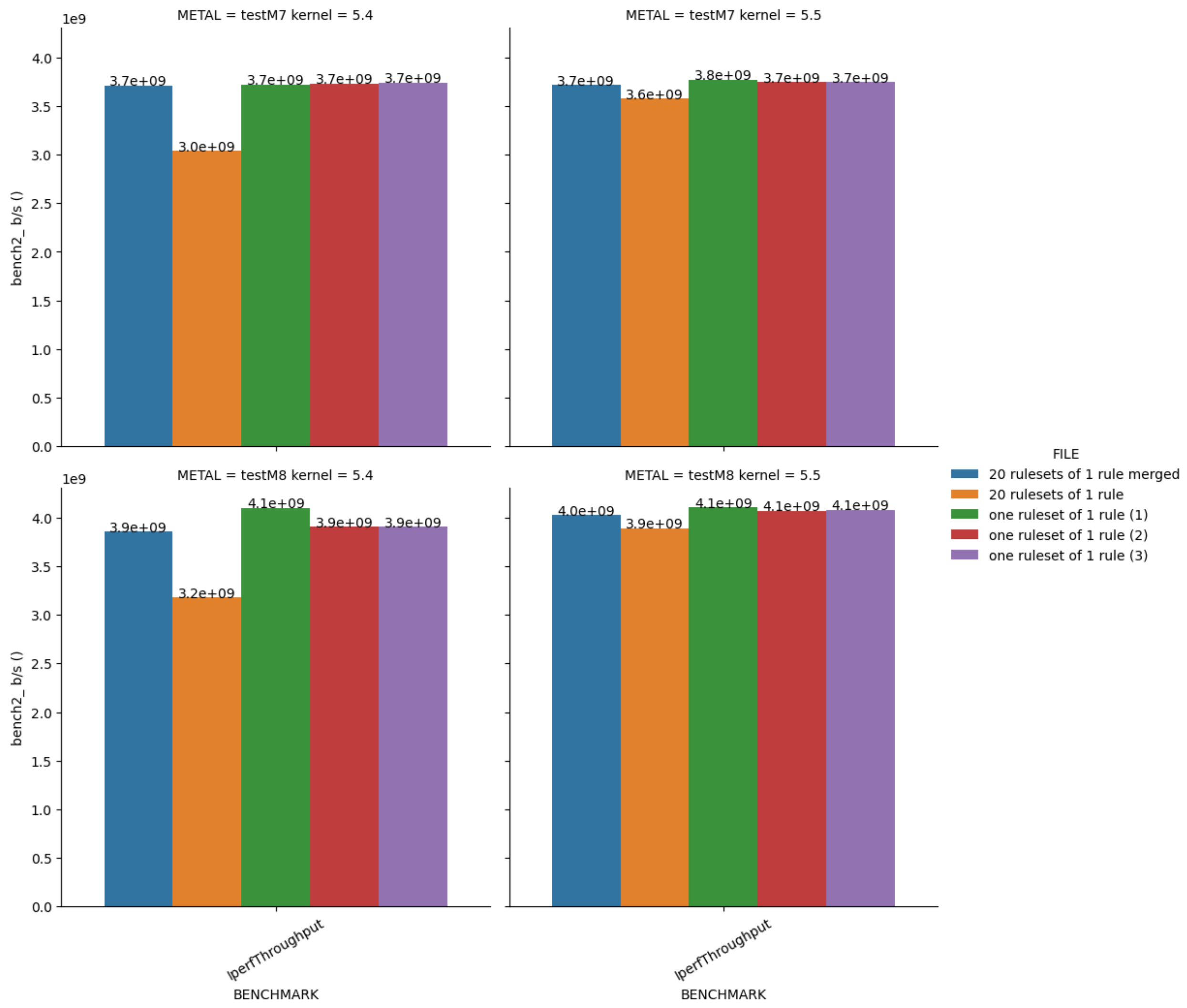

Figure 5: Benchmark 2: throughput measured across various metals and various kernel versions, using rulesets of one rule.

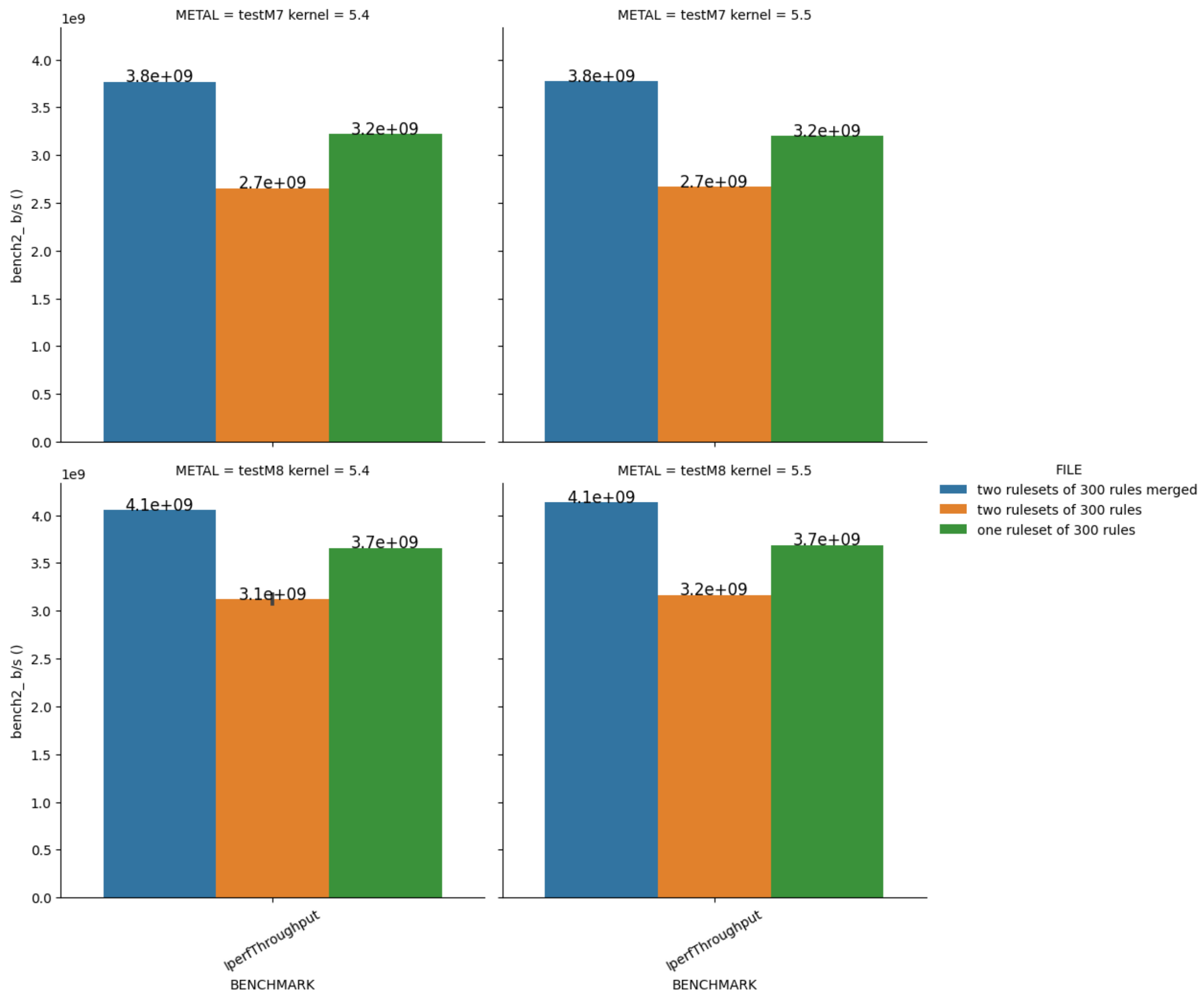

Figure 6: Benchmark 2: throughput measured across various metals and various kernel versions, using rulesets of one rule.