SymPerf: Predicting Network Function Performance

Felix Rath, Johannes Krude, Jan Rüth, Daniel Schemmel,
Oliver Hohlfeld, Jó Á. Bitsch, Klaus Wehrle
Communication and Distributed Systems, RWTH Aachen University
{rath,krude,rueth,schemmel,hohlfeld,bitsch,wehrle}@comsys.rwth-aachen.de

ABSTRACT

The softwarization of networks provides a new degree of flexibility in network operation but its software components can result in unexpected runtime performance and erratic network behavior. This challenges the deployment of flexible software functions in performance critical (core) networks. To address this challenge, we present a methodology enabling the prediction of runtime performance *and* testing of functional behavior of Network Functions. Unlike traditional performance evaluation, e.g., testbed testing or simulation, our methodology can characterize the Network Function performance for *any* possible workload only by code analysis.

CCS CONCEPTS

• **Networks → Network performance analysis**; *Middle boxes / network appliances*; *In-network processing*; Programmable networks;

KEYWORDS

NFV, Symbolic Execution, Measurements, Reliability, Performance

ACM Reference Format:
Felix Rath, Johannes Krude, Jan Rüth, Daniel Schemmel, Oliver Hohlfeld, Jó Á. Bitsch, Klaus Wehrle. 2017. SymPerf: Predicting Network Function Performance. In *Proceedings of SIGCOMM Posters and Demos '17, Los Angeles, CA, USA, August 22–24, 2017,* 3 pages.
https://doi.org/10.1145/3123878.3131977

1 INTRODUCTION

Current research in SDN and NFV exemplifies the resurgence of network softwarization and stands to fulfill the promise of active networks by enabling flexible in-network processing for control and data plane tasks. Contrasting this flexibility is 1) an erratic latency being added to each flow that is now processed in software rather than on dedicated, purpose-built hardware, and 2), a loss in reliability, as software running on generic platforms is much harder to test comprehensively. It is noteworthy that current implementations of even simple on-path Network Functions (NFs) face *both* of these problems to *unknown* degrees.

To address this dual challenge, we present *SymPerf*, a methodology that accurately predicts runtime and behavior of an NF purely by automated reasoning about its implementation. SymPerf requires no testbed emulations for known workloads and can *fully* characterize NF performance for any input—even for the rarest corner case. This work seeks to create *pre-deployment* confidence in the runtime behavior of NFs by working towards addressing the following questions that network operators face when designing and deploying NFs:

Implementation quality: Does the NF lead to an unwanted state, e.g., an infinite loop? Can unexpected input cause forbidden behavior, such as buffer overflows?

Performance: How much latency and jitter is added per packet? Will the NF sustain line rate? What is the worst processing time for the NF and when—state, sequence of packets—does it occur? How are execution times distributed for a random, known or adversarial workload? How many NFs can be executed on a single machine?

SIGCOMM Posters and Demos '17, August 22–24, 2017, Los Angeles, CA, USA

2 PREDICTING NF BEHAVIOR

Our approach aims to predict the performance of an NF from its underlying source code, as illustrated in Fig. 1, beginning with enumerating all possible execution paths. In our scheme, we use Symbolic Execution (SE) [2] in step ① to generate a complete execution tree. SE has already proven itself in the networking domain [3, 5] and allows us to categorize all possible inputs in such a way that each possible execution path maps to exactly one distinct input category. Since SE originates from automated software testing, it is also able to find a broad range of software defects, such as buffer overflows. If a bug is found, the developer is given a test case triggering the bug, to iterate on the NF (step ②).

Each path through the execution tree is necessarily also a pass through the NF under scrutiny. Next, in step ③, we walk the execution tree generating one instruction chain per path, in effect mapping each possible input category to its corresponding CPU instruction chain. In order to calibrate our prediction for a target platform, we perform microbenchmarks measuring the costs of instructions. Instead of measuring time durations, we use CPU cycles, which are a metric independent of frequency scaling. During the calibration, which has to be done once for each platform, like our testbed running Linux on an Intel i7-870, we store the results in a per platform instruction cost database (cf. Fig. 1, step ④). Note that this is only done once per target.

Combining the NF instruction chains with the appropriate calibration database (step ⑤), we can predict runtimes for each path. This is already enough to find, e.g., the path that consumes the most CPU cycles, i.e., the worst-case path through the NF. The results from the prediction can also be combined into a general overview, assuming a uniform distribution over all paths, as is done in the example presented in the next section. Alternatively, we can categorize packets from a captured trace, which gives a traffic-dependent prediction without needing a full testbed.

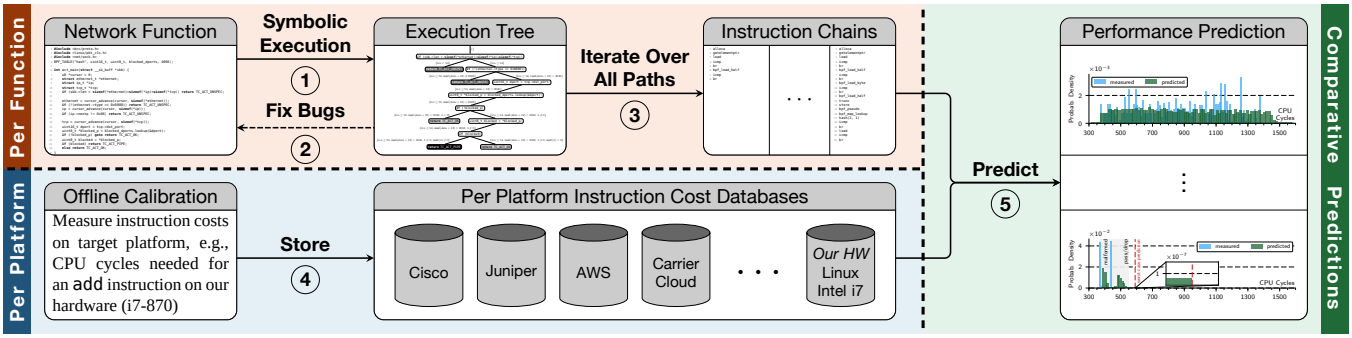
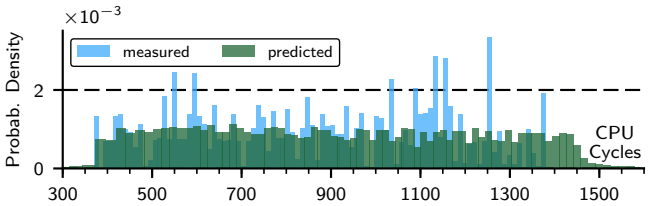
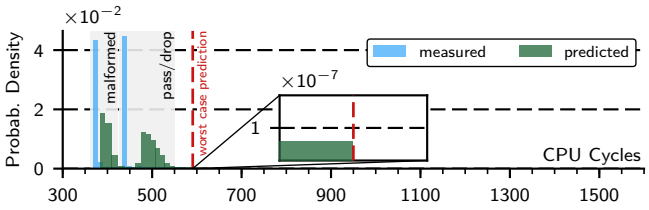


Figure 1: SymPerf architectural overview and Network Function performance prediction workflow.



(a) Histograms for the array-based implementation.



(b) Histograms for the hashmap implementation.

Figure 2: Histograms for predicted and measured runtime distributions of two TCP firewall implementations.

3 FIREWALL PREDICTION EXAMPLES

We apply SymPerf to predict the runtime performance of two simple Berkeley Packet Filter (BPF) firewalls. BPF can run on the data path [1], is available in Linux and already found industry-level deployment (e.g. [4]). Since BPF NFs are small but performance critical, they are a natural showcase for our methodology's usefulness.

We use our methodology to predict the performance of an array-based and a hashmap firewall implementation. Both implementations perform the same task, i.e., dropping packets addressed to a list of TCP ports. The first implementation sequentially scans an array for blacklisted ports, while the second does a single hashmap lookup instead. To create an instruction cost database (see Fig. 1), we calibrate our approach for BPF instructions executed on our testbed hardware: an off-the-shelf Intel CPU running Linux.

First, we compare the *predicted* to the *measured* performance of the array firewall over all possible paths (assuming each path to be equally likely) in Fig. 2(a). To obtain measured performance figures of the firewall, we generate packets in a testbed triggering all paths and measure the consumed CPU cycles until each path is hit at least 10^6 times. We applied a 0.5% error margin for outlier removal. Note the heavy variability in the NF performance, which is no surprise, as some ports are at the far end of the array and matched last, while other are at the start and matched earlier. Also, observe that our prediction *closely matches* measured performance.

Similarly, we compare the predicted to the measured performance of the hashmap firewall in Fig. 2(b). We observe that the variance is significantly lower than for the array firewall, reflecting the $O(1)$ runtime of hashmap lookups. This shows that our methodology is able to differentiate the runtime performance of the two implementations of the same NF. We further show that our methodology can predict the *worst-case runtime performance* of the NF. That is, we correctly predict that no path in this firewall will ever take longer than 591 CPU cycles. Thus, we are already able to make *informed decisions* about the quality of both firewalls.

We are even able to distinguish different paths through the same NF with our methodology. Fig. 2(b) clearly distinguishes paths that exit early as the packet is malformed (e.g. invalid TCP), as well as paths executing the core firewall. We observe that our predictions are off by few CPU cycles, caused by advanced CPU features that cannot easily be captured during calibration (e.g., superscalar execution). However, SymPerf correctly predicts the relative performance of both program parts. Also, predicted performance is more conservative than measured, yielding a useful upper bound.

Preliminary results show that SymPerf accurately predicts the runtime performance of BPF NFs for *all* possible execution paths. This enables network engineers to evaluate and compare performance of different NFs without performing testbed evaluations.

4 CONCLUSION

This paper discusses SymPerf, a new approach for analyzing and predicting NF performance. We show the potential of Symbolic Execution not only for testing but also for quantitative NF performance prediction and implementation comparison, including the expected and worst case. We currently focus on further improving this prediction as well as the resilience of our calibration. Overall, this will lead to safer NFs with predictable performance.

ACKNOWLEDGEMENTS

This work has been partly supported by the European Research Council (ERC) under the EU's Horizon2020 research and innovation programme grant agreement n. 647295 (SYMBIOSYS) and by DFG within SPP 1914 (Cyber-Physical Networking).

REFERENCES

[1] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. 2016. InKeV: In-Kernel Distributed Network Virtualization for DCN. *ACM SIGCOMM Computer Communication Review* 46, 3.

[2] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

[3] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[4] Cilium Authors. 2017. Cilium: Helping Linux Secure Microservices. https://www.cilium.io/ [accessed 2017-07-13].

[5] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: scalable symbolic execution for modern networks. In *Proceedings of the ACM SIGCOMM 2016 Conference*. ACM, 314–327.