Performance Contracts for Software Network Functions

Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli,
Katerina Argyraki, and George Candea

EPFL, Switzerland

Abstract

Software network functions (NFs), or middleboxes, promise flexibility and easy deployment of network services but face the serious challenge of unexpected performance behaviour. We propose the notion of a *performance contract*, a construct formulated in terms of *performance critical variables*, that provides a precise description of NF performance. Performance contracts enable fine-grained prediction and scrutiny of NF performance for arbitrary workloads, without having to run the NF itself.

We describe BOLT, a technique and tool for computing such performance contracts for the entire software stack of NFs written in C, including the core NF logic, DPDK packet processing framework, and NIC driver. BOLT takes as input the NF implementation code and outputs the corresponding contract. Under the covers, it combines pre-analysis of a library of stateful NF data structures with automated symbolic execution of the NF's code. We evaluate BOLT on four NFs—a Maglev-like load balancer, a NAT, an LPM router, and a MAC bridge—and show that its performance contracts predict the dynamic instruction count and memory access count with a maximum gap of 7% between the real execution and the conservatively predicted upper bound. With further engineering, this gap can be reduced.

1 Introduction

The goal of our work is to enable network operators and developers to predict and scrutinise the performance of software network functions without having to run them. A network function (NF) performs packet processing inside the network, such as packet forwarding, load balancing, or network address translation (NAT). NF development has been moving away from custom hardware toward software running on commodity hardware. This change increases flexibility and reduces development costs and time-to-market [22, 37, 38], but arguably makes it harder to predict the NF's performance. Unexpected NF performance behaviour makes it harder for network operators to provision their networks and exposes a new attack surface for adversaries seeking to degrade network performance.

We propose the construct of a *performance contract* for NFs. A contract $C_N^U(i)$ answers the question of what the performance of the NF N is like when processing packets from an arbitrary input packet class i, with performance measured in units of U. To illustrate, N could be a particular implementation of a router, U — the number of x86 instructions it executes per packet, and i_1 (respectively i_2) — the class of valid (respectively invalid) packets arriving at the router. The contract predicts performance in terms of human-readable expressions. These expressions are functions of what we call *performance critical variables* (PCVs), which summarise the impact of input history and configuration on the given NF's state and execution. In our example, the contract could return functions $p_1(l)$ (respectively $p_2(l)$) for valid (respectively invalid) packets, where l is the length of the IP prefix that matches the input packet's destination IP address. l is a PCV. For a NAT, a PCV could be the occupancy rate of the NAT's flow table. In this paper, we consider three performance metrics: number of executed instructions, number of memory accesses, and number of execution cycles. In general, we consider an NF implementation to be the software stack plus the hardware architecture it runs on.

Our work draws upon ideas from earlier work on analysing/predicting performance and worst-case execution time (WCET), either of software in general [24, 26, 45] or of NFs in particular [15, 32]. The way performance contracts differ from classic performance prediction and WCET analysis is that, rather than producing a performance number, they express performance as a function of critical parameters—the PCVs. This enables contracts to expose the entire range of values of the NF's performance, not just a worst-case bound, as well as explain how these values relate to different workloads. Performance contracts also strike a favourable balance between accuracy, utility, and human legibility.

We present BOLT, a technique and tool that analyses NF code, without actually running it, to generate NF performance contracts. We draw inspiration from Vigor [47], a technique for verifying that NFs written in C satisfy semantic properties and are memory-safe. Vigor assumes a clear separation of NF code into (a) a library of commonly used NF data structures, which is written and formally verified by experts, and (b) stateless NF logic that uses the library, is written by NF developers, and is verified automatically by Vigor using symbolic execution. In the same spirit, BOLT starts from manually pre-computed performance contracts for basic common data structures as a base case, and then automatically generates performance contracts for the NF code that uses these data structures. Contracts can be computed recursively for chains of NFs as well.

To help operators and developers handle the PCVs in contracts, BOLT comes with a tool we call the Distiller, which takes as input a packet trace and computes the PCV values that result from that trace being processed by the target NF.

We evaluate the accuracy and utility of BOLT-generated performance contracts for four NFs written in C using DPDK: a NAT, a Maglev [17]-like load balancer, an LPM router, and a MAC bridge. The contracts for these NFs predict the dynamic instruction count and memory access count with a maximum gap of 7% between real executions and the conservatively predicted upper bound. We explain the origin of this gap and argue that it can be reduced to 0 with further engineering. BOLT also generates contracts for the number of execution cycles, in which case it relies on a hardware model; the result is as accurate as the model allows. We close our evaluation with example use cases where BOLT uncovers performance issues and helps understand how to fix them.

In summary, we make two contributions in this paper:

- We propose the concept of a performance contract for NFs, which expresses NF performance as a function of performance critical variables.
- We demonstrate, using our BOLT prototype, that it is possible to compute performance contracts that are accurate, useful, and human-legible.

The BOLT source code is available as open source [1].

In the rest of the paper we define the performance contract construct (§2), then describe how BOLT generates performance contracts (§3) and how the BOLT Distiller helps obtain a concrete performance number from a performance contract given a particular packet trace (§4). Finally, we present BOLT's evaluation (§5), discuss its limitations (§6), present related work (§7), and conclude (§8).

2 Performance Contracts

In this section, we define the performance contract construct, and we use a running example to illustrate this definition.

2.1 Running Example: LPM Router

Algorithm 1 shows pseudocode for a simplified longest prefix match (LPM) IPv4 router that stores the forwarding table in a Patricia trie. The router first classifies packets based on whether they are IPv4 or not (line 2). Invalid packets are immediately dropped (line 6), thus incurring a constant performance cost. Valid packets lead to a lookup in the LPM data structure (line 3), which has a more complex performance profile (lines 10–17), with the number of loop iterations being data-dependent (see lines 12 and 15).

Algorithm 1: Simple LPM Router

```
1  function processPacket (packet pkt)
2  if pkt.etherType == IPv4 then
3      dst_port = lpmGet (pkt.ipv4.dst_addr)
4      FORWARD (pkt, dst_port)
5  else
6      DROP (pkt)
7  end

8  function lpmGet (bit ip[32])
9  node = lpmRoot
10 for i in 0..31 do
11     b = ip[i]
12     if exists node.children[b] then
13         node = node.children[b]
14     else
15         break
16     end
17 end
18 return node.port
```

For clarity of exposition, the running example assumes that the packet processing framework and every layer below has zero impact on performance.

2.2 Definition

A performance contract describes the performance of NF software running on a particular hardware configuration.

Contract $C_N^U : I \rightarrow F$ is a map from input classes to functions, i.e., C_N^U maps input class $i \in I$ to a function $p_i(v_1, v_2, ...) \in F$. Input class i is a specification that describes which inputs (e.g., packets) belong to that class, such as a symbolic expression for "all valid IPv4 packets without IP options." The contract's domain I spans the entire input space of the program N. Function p_i expresses the performance exhibited by N when processing an arbitrary input that belongs to class i. p_i is a function of performance-critical variables $v_1, v_2, ...$, and its value is measured in units of U. p_i can be as simple as a constant function.

In general, a performance contract can be formulated for any program/procedure P, not just an NF. The performance of P is determined by its input and its state at the time of processing the input. P's state, in turn, is determined by the inputs it has processed in the past, its configuration parameters, and its environment.

Performance-critical variables (PCVs) capture the influence on performance of anything other than P's input.

Performance is expressed through metrics, such as number of memory accesses or number of execution cycles. Performance contracts are metric-specific. An essential property of a performance contract is that, for any real execution that satisfies the contract's assumptions (configuration, input history, etc.), the measured performance is guaranteed to be no more than the metric value predicted by the contract.

A performance contract for P is a recursive composition of the performance contracts for its constitutive parts. P as used above could be a chain of NFs, one individual NF, a part of an NF, or—in the base case—a data structure method whose contract has been derived manually by an expert.

Table 1 shows two performance contracts for our example LPM router, corresponding to two performance metrics: instruction count and memory accesses. There are two input classes that emerge naturally from this NF's code structure: invalid and valid packets. To process any packet in the first class, the NF executes 2 instructions and 1 memory access. For the second class, it executes $4 \cdot l + 5$ instructions and $l + 3$ memory accesses, where l is the matched prefix length. This example ignores all layers below the NF code, so the matched prefix length l fully captures how anything other than the input packet (in particular, the configuration of the LPM table) influences performance. Hence it is the sole PCV used by both contracts.

Input Class	**Instructions**	**Memory Accesses**
Invalid packets	2	1
Valid packets	$4 \cdot l + 5$	$l + 3$

Table 1: Two stylised performance contracts for an example LPM router. PCV l is the matched prefix length.

2.3 Rationale Behind PCVs in Contracts

Designing a performance contract involves a trade-off between the level of detail that the contract exposes about the code and the contract's legibility (i.e., how easy it is for a human to parse and draw useful conclusions from).

Different users may need different balances between detail and legibility. For instance, an NF developer who uses performance contracts to debug and optimise the performance of their own code will likely want more detail than a network operator who has no access to NF implementations and uses performance contracts solely to provision their network.

We chose to express performance as a function of PCVs because this enables one to navigate fluidly this trade-off, i.e., to generate contracts that achieve different detail/legibility balance. For example, consider a hypothetical NF whose only operation is to update, for every observed packet, per-flow state stored in a hash map. The performance of this NF is determined, in a straightforward manner, by the collision rate of the hash map, which is itself determined, in a complicated manner, by the workload and hash map configuration. So, one can express the performance of this NF as a complicated function of workload and configuration, or as a simple function of the hash map's collision rate. The former arguably provides all the detail that any user of a performance contract would ever care for, but may be too much for a human to digest and draw conclusions from; the latter hides some of this detail but still provides insight into what determines NF performance and how, just at a different level. So, it is possible to generate performance contracts that favour detail or legibility by choosing the proper level of PCVs.

We are concerned about exposing, in a performance contract, implementation-specific notions like collision rates or matched prefix lengths. This may be fine for the developer who chose or implemented the NF's data structures, but awkward for an operator who knows nothing about the NF's implementation. Still, we do not think that it is possible to design meaningful performance contracts that do not leak non-trivial information about implementation. In the end, when a network operator is debugging an unexpectedly slow network device, they do end up digging into the device's implementation and trying to understand how that interacts with the given workload. A performance contract that distills how implementation affects performance into a simple expression would arguably be welcome in such cases.

If desired, operators and developers can bind the PCVs in the performance expressions to values chosen by themselves or by the BOLT Distiller. The latter, given a packet trace, computes the concrete values of the PCVs at the end of the NF's processing of that trace.

3 Generating Performance Contracts

In this section, we describe BOLT, a technique and tool that generates performance contracts. Our current prototype works with three performance metrics: number of executed instructions, memory accesses, and execution cycles.

We first provide background on the techniques BOLT employs (§3.1), then describe how to obtain contracts for data structures (§3.2), entire NFs (§3.3), and chains of NFs (§3.4). We close with a few implementation details (§3.5).

3.1 Background

The conceptually simplest way to explore all possible behaviours of a program is to execute it with every possible input. As this does not typically scale to real-world programs, a more efficient approach is to group inputs in non-overlapping classes, such that all inputs in the same class follow the same execution path through the program (hence induce the same behaviour); then we can explore induced behaviour once per input class. For example, a program that takes as input a 64-bit number and takes one of two possible actions depending on whether the number is positive or negative has 2^{64} possible inputs but only 2 input classes that induce different behaviours.

Symbolic execution (SE) [8,21] is a commonly used technique for exploring feasible execution paths of a program and identifying the input class that triggers each one. SE relies on a special program interpreter called a symbolic execution engine (SEE), which uses *symbols* to represent inputs and propagates these symbols through the program. For instance, if a program takes as input an integer x, the engine associates with x a symbol α; if the program assigns to a variable y the value $x+1$, the engine associates with y the expression $\alpha+1$. If the program branches on a symbolic value, the engine explores both paths and keeps track of the *constraints* that led down each path, such as $\alpha < 0$. The engine uses a constraint solver [13,20] to ensure that it explores only feasible paths and to identify the input class that triggers each one. However, it can typically not identify *all* the feasible paths of a program due to path explosion [6]: the number of paths is generally exponential in the number of branches in the code, and symbolic pointers make things worse, as the engine sometimes needs to concretise them, i.e., fork a new path for each possible address that a symbolic pointer may reference.

Vigor [47] leverages SE to verify semantic properties and memory safety of a stateful NF. It assumes a clear separation of NF code into: a library of common NF data structures that is written and verified by experts; and stateless NF code that uses the library and is written by NF developers. The experts that verify the library produce a semantic contract for each library method, which specifies pre- and post-conditions for the method; this is a tedious process that requires time and expertise, but it needs to be done only once per library method, hence its cost is amortised when multiple NFs use the library.

Vigor uses SE to automatically explore all the feasible paths through the stateless NF code. The Vigor toolchain includes an SEE tailored to the domain of NFs so that SE of the stateless NF code does not suffer from path explosion. Vigor automatically combines this analysis with the semantic contracts for the library methods used by the NF, and generates a proof that the NF as a whole satisfies the target semantic properties and is memory-safe. Note that Vigor's semantic contracts are unrelated to our performance contracts and do not provide any performance-related information.

BOLT reuses Vigor's toolchain and adopts a similar NF development process: A team of experts writes the library of common NF data structures and their performance contracts and symbolic models. NF developers write stateless NF code that uses this library and use BOLT to generate performance contracts for the NF.

Input Class	**Instructions**	**Memory Accesses**
Unconstrained	$4 \cdot l + 2$	$l + 1$

Table 2: Performance contract for `lpmGet`. PCV l is the matched prefix length.

3.2 Base Case: Contracts for Data Structures

The first step is to manually generate performance contracts for the parts of the code that BOLT does not analyse automatically; for our current prototype, these are all the methods for accessing data structures that keep NF state. In the same spirit as Vigor, we rely on a library of common data structures that are analysed once and then reused across multiple NFs. Table 2 shows manually generated performance contracts for the `lpmGet` method used by our LPM router. Like the performance contracts for the entire router, the ones for its data structure express performance as a function of the length of the matched prefix l, which is the only PCV.

Part of this process—perhaps the hardest one—is picking a set of PCVs so as to achieve a target balance between precision and legibility. For example, the `lpmGet` method uses pointer arithmetic (line 12), which the compiler unfolds into a series of conditional jumps; as a result, the performance of the method varies slightly, depending on whether each bit in the matched IP prefix is 0 or 1. One option is to expose each bit in the matched IP prefix as a PCV, in which case the contract precisely predicts the performance of any real executions. Another option is to assume that each bit has the value that results in the worst-case performance (essentially coalesce multiple execution paths into the one among them with the worst performance) and expose only the length of the matched prefix as a PCV; in this case, the contract predicts performance conservatively, i.e., overestimates the number of execution cycles and memory accesses. This is an example of how a higher-level PCV sacrifices a small amount of precision for a more concise, hence legible contract.

3.3 Contracts for NFs

BOLT (Algorithm 2) takes as input the NF code (line 1) and generates a special build where all calls to stateful methods are replaced at link time with calls to corresponding symbolic models (line 2). For example, in our LPM router, the call to `lpmGet` is replaced with a call to the symbolic model shown as Algorithm 3. Next, BOLT symbolically executes this special build exhaustively and obtains all feasible execution paths through the stateless NF code (line 3). For our

Algorithm 2: GetPerformanceContract

```
1  function GetPerformanceContract
   Input  : function Fn,
     <opt> map <function, model> Models
     <opt> map <function, perfContract> Contracts
   Output: perfContract Perf_Contract
2  stubbedFn := SubstituteModels(Fn, Models)
3  paths := GetAllPaths(stubbedFn)
4  Perf_Contract := $\phi$
5  foreach path in paths do
6      inputs := GetInputsForPath(path)
7      traceInstr := GetInstrTrace(stubbedFn, inputs)
8      perf := $\phi$
9      foreach instr in traceInstr do
10         if instr is a call to a stateful function fn then
11             perf += Perf(Contracts[fn], path.constraints)
12         else
13             perf += Perf(instr)
14         end
15     end
16     Perf_Contract.append(path, perf)
17 end
18 return Perf_Contract
```

Algorithm 3: `lpmGet` function model.

```
1  function lpmGet ( bit ip[32] );
2  return <new symbol>
```

LPM router, this results in 2 paths, one for valid IPv4 packets and one for invalid packets. For each execution path, BOLT also obtains *symbolic path constraints*, which consist of two categories of constraints: (1) constraints on NF inputs that cause it to go down the particular execution path and (2) constraints on the abstract state of each data structure, before and after each call to a stateful method. The second category of constraints tells BOLT how stateless and stateful code interact along the execution path.

Once it has obtained all feasible execution paths and their path constraints, BOLT analyses each path: First, it passes the path's constraints to a solver to obtain concrete inputs that exercise the path (line 6); these inputs include a packet, as well as values for any symbols generated by the symbolic models of the stateful methods. For our LPM router, one path will yield a concrete invalid IPv4 packet, while the other will yield a concrete valid IPv4 packet and a concrete port that would result from the LPM lookup (e.g., port 0). Next, for each of these concrete inputs, BOLT replays the NF execution and obtains a unique trace of machine instructions (line 7).

Finally, BOLT characterises the performance of each feasible execution path by stepping through the corresponding instruction trace: it traverses the trace, adding up the cost of each instruction (line 13), until it hits a call to a modelled method; when this occurs, it picks the right branch of the method's performance contract based on the constraints on the abstract state of the data structure (line 11). In the case of `lpmGet`, the performance contract has no branches. This will typically not be the case for more complex data structures and methods, e.g., the performance contract of a flow table *get* method will have different formulae depending on whether the flow is present or absent in the flow table. In such a scenario, BOLT uses the path constraints to pick the right formula.

3.4 Contracts for NF chains

We summarise how BOLT can be extended to generate contracts for chains of NFs. This can be useful in scenarios where one NF's worst-case performance is masked by another NF on the same chain. For example, consider the scenario where, in front of our LPM router, an operator deploys a firewall that drops all packets matching prefixes that exceed a given length. In this scenario, we will get a more accurate performance prediction by using a performance contract generated for the NF chain as a whole, than by using two separate contracts generated for each NF and adding their predictions.

We can extend BOLT for joint analysis of multiple chained NFs as follows: First, generate a performance contract for each individual NF as before. Next, pair together execution paths from two connected NFs; for each such path pair, AND together their respective path constraints and add equality constraints connecting the symbolic expression for the packet sent by the first NF to the symbol representing the packet received by the second NF. Next, use a solver to check if the paths are compatible. Finally, generate a global performance contract for the NF pair that sums up the performance for each compatible path pair, while ignoring incompatible ones.

For longer chains, rather than fully enumerating the entire combinatorial explosion of all path tuples, BOLT could piece together compatible paths across the chained NFs one at a time in sequence, following a procedure similar to joint symbolic execution [31]. This process further generalises to more complex networks, so long as the topology forms a directed acyclic graph (DAG).

3.5 Implementation Details

Instruction Replay. While replaying each execution path, we use an instruction tracer based on Intel's pin dynamic binary instrumentation tool [25] to log the x86 instructions along with memory locations touched along that path. During replay, we ensure that despite the difference between the analysed code (linked against models) and production code (linked against actual data structures), BOLT remains conservative. This is done by compiling the stateless NF code separately from the models and disabling any link-time-optimisations when linking them together. While this leads to slight over-estimations, it ensures that BOLT never under-approximates performance.

Hardware model employed. For metrics that rely on the underlying hardware (e.g., cycles), the BOLT prototype employs a simple, conservative hardware model that does not model CPU components that are either too complex or constitute trade secrets. For compute instructions, BOLT conservatively assumes the worst case performance cost of each instruction as reported in the Intel manual [2] due to the proprietary nature of out-of-order (OoO) instruction scheduling within the processor. For memory instructions, we only model the private L1 Data Caches. We do not model proprietary features such as the slice selection algorithm in the L3 cache, memory-level parallelism (MLP), or pre-fetching. Consequently, BOLT conservatively assumes that every memory access is serviced from main memory unless it can definitively prove otherwise (by tracking spatial and temporal locality of memory accesses in the L1D cache).

These proprietary features significantly improve NF performance, given that an NF repeatedly performs the same tasks (e.g., flow expiration, hash-ring traversal) in a tight loop. Consequently, our hardware model causes BOLT to over-estimate performance and in our experimental evaluation §5, we find that BOLT comes within 4× for typical workloads and 9× for pathological workloads. However, we show that it should be possible to improve the accuracy of BOLT's contracts by plugging in a better hardware model.

Including DPDK and NIC driver code. BOLT allows analysis of the NF code at two different levels of abstraction: 1) Only the NF code sitting atop DPDK or 2) the entire software stack including the NF logic, DPDK and the NIC driver.

Doing this at the level of just the NF is relatively simple. We stub out the DPDK send and receive calls and replace them with models that inject the packet symbols. We then filter the stateless instruction traces to include only the instructions between these two calls.

Building on recent work [34] that applied Vigor to the entire NF software stack, BOLT can also include DPDK and the NIC drivers in its performance predictions. The insight behind this work is that while such frameworks as a whole may be complex, simple NFs require only a small subset that primarily reads and writes to device registers. This subset has simple control flow and so can be symbexed along with the stateless NF code. In this case, we include in the trace instructions from the beginning of the driver receive function until the end of the corresponding send/drop function.

4 The BOLT Distiller

Once BOLT has built a performance contract, users can predict the performance of the NF under varying assumptions. However, performance contracts can have several hundred to a few thousand execution paths each, with their own unique assumptions. Often, it is not obvious which assumptions are reasonable or typical in the real world. To reason about this, BOLT provides an additional tool called the *Distiller*.

The Distiller takes as input the NF code and a sample of real-world traffic (as PCAP files). It feeds the traffic through the NF code and logs the values that are induced in each model parameter. For our running example, this would mean linking the stateless code with a slightly modified version of the data structure that traces the number of loop iterations that occur, logging the matched prefix length. With these traces, the Distiller computes a detailed breakdown of which assumptions hold for each packet, and how that relates to predicted performance. Note, the distiller does not affect the generated performance contract in any way; it merely tells the user which assumptions held for each packet in the given trace allowing the user to then extrapolate and identify execution paths of interest in the performance contract.

The distiller also enables users to perform a sensitivity analysis. For our example LPM, the user could, for instance, see that most packets match prefixes that are 16 to 24 bits long. Longer prefixes lead to 32% worse performance (133 vs 101 instructions) but may (hypothetically) account for only 1% of traffic.

An operator can leverage the Distiller to balance risk with resource utilisation to decide how to provision the network. A developer can understand how any assumptions that they have made regarding which scenarios are more common may be wrong, guiding further optimisation efforts. We illustrate further, the utility of the distiller in §5.2 and §5.3.

5 Evaluation

We now examine whether BOLT works, i.e., whether it produces correct performance contracts for software NFs (§5.1), and illustrate, through example use cases, how BOLT can help network operators (§5.2) and NF developers (§5.3).

5.1 Does BOLT work?

We experiment with a MAC bridge (Br), an LPM router implemented with DPDK's LPM data structure [3] (LPM), the NAT from [47] (NAT), and a Maglev-like [17] Load Balancer (LB).

Testbed. We use two directly connected servers: a device under test (DUT) and a traffic generator and sink (TG). Both servers have Intel Xeon E5-2667v2 3.3GHz CPUs with 32GB of RAM; they are connected over Intel 82599ES 10Gb

NICs. The DUT runs one of the NFs, while the TG uses MoonGen [18] to replay a PCAP file, one packet at a time, to avoid any queuing or pipelining effects.

Methodology. First, Bolt generates performance contracts for each of the 4 NFs for three different metrics — number of executed instructions, number of memory accesses and number of execution cycles. Each such contract subsumes from several hundred to a few thousand unique execution paths.

We compare the performance predicted by each contract for various input packet classes to actual measurements. We use broad input packet classes (e.g., "unconstrained traffic" or "broadcast traffic"), each of them covering thousands to millions of possible packets and exercising hundreds to thousands of execution paths in the NFs. Given these broad input classes, BOLT being conservative reports the predicted performance value of the execution path with the worst predicted performance. Said differently, BOLT predicts the worst performance that an input packet from this class could encounter.

For most packet classes, we programmatically produced a PCAP file that samples the packet class, i.e., contains a large number of packets from that packet class, and obtained measurements from our testbed. For one specific packet class (discussed below), we could not produce a representative PCAP file; in this case, to obtain ground truth, we modified the NF to synthesise the expected state (so that it did not need to be built from actual packet history). For each NF and input packet class, we measure the performance metrics predicted by BOLT: instruction count (IC), number of memory accesses (MA), and latency (cycles); IC and MA are measured using the binary instrumentation described in §3.5, while latency is measured using high precision CPU clocks (TSC) during separate non-instrumented runs.

Input packet classes. For each NF, we first consider unconstrained traffic (scenarios Br1, LPM1, NAT1, and LB1). Given this input packet class, the performance contract generated by BOLT predicts the absolute worst-case performance of the NF. For Br, NAT, and LB, which maintain and expire per-flow state, BOLT determines that the worst-case performance happens when the NF's MAC/flow table is full, all the entries have collided with each other, and all the entries are sufficiently aged so as to induce a mass expiry event that completely clears the table when the current input packet arrives. We were unable to produce a PCAP file that led to this pathological scenario, but still wanted to verify that, if this scenario did occur, the NF's performance would indeed be the one predicted by BOLT. This is why we modified the NF to synthesise the necessary state (as stated above). To generate unconstrained traffic for the LPM, we used the CASTAN framework [32] which specialises in generating adversarial workloads for NFs.

For the NFs that maintain per-flow state, we also consider

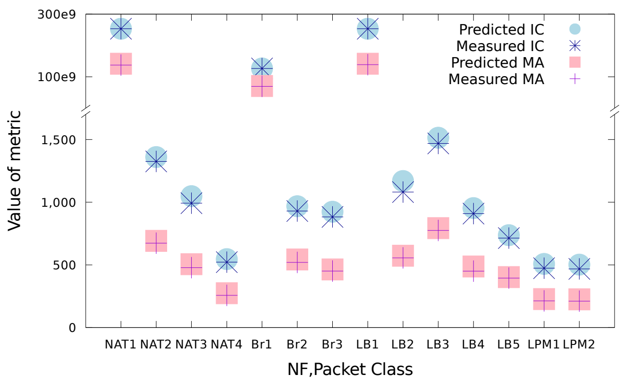

Figure 1: Accuracy of Performance contracts for multiple NFs and packet classes in terms of "Instruction Count"(IC) and "Number of Memory Accesses"(MA).

a few representative classes of input packets that do not encounter hash collisions or entry expirations. For the Bridge: broadcast (Br2) and unicast (Br3) packets. For the NAT: packets arriving from the internal network that belong to new (NAT2) and established (NAT3) connections, and packets arriving from the external network that do not belong to an established connection and are dropped (NAT4). For the Load Balancer: packets arriving from the external network that belong to new flows (LB2), existing flows with unresponsive (LB3) and live (LB4) backend servers and heartbeat packets from backend servers (LB5).

The LPM uses DPDK's two-tiered lookup table, which is structured such that any packet with a matched prefix of $\leq$ 24 bits incurs exactly one lookup and all other packets incur exactly two lookups. Hence, any input packet class where the packets are constrained to matched prefixes of $>$ 24 bits can incur the same performance as unconstrained traffic. In addition, we consider input packets that are constrained to matched prefixes of $\leq$ 24bits (LPM2).

Results for hardware-independent metrics. Figure 1 shows the results for the metrics IC and MA, and we see that BOLT predicts them accurately, with a maximum over-estimation of 7.5% and 7.6%, respectively. It is possible that our generated test traffic may not have incurred the actual worst-case performance, hence the above numbers represent an upper bound on Bolt's over-estimation. In the pathological scenarios that correspond to unconstrained traffic (Br1, NAT1, LB1) for NFs that maintain and expire per-flow state, both the predicted and the actual performance is 8 orders of magnitude worse than in the other scenarios (in these extreme scenarios, a packet could take over a minute to be processed). Even so, BOLT's IC and MA predictions are accurate (and conservative) with maximum over-estimation 2.36% and 3.03%, respectively. The over-estimation in IC and MA predictions comes from two sources: (1) imprecision introduced when we coalesce execution paths within the

NF+Class	Predicted Bound	Measured Cycles	Ratio
NAT1	591,948,908,371	65,217,699,390	9.08
NAT2	7,401	2,376	3.11
NAT3	5,142	1,789	2.87
NAT4	2,956	884	3.34
Br1	295,984,939,878	32,383,472,634	9.14
Br2	7,329	2,013	3.64
Br3	7,383	1,808	4.08
LB1	591,969,879,756	66,062,284,173	8.96
LB2	5,299	2,386	2.22
LB3	8,108	2,541	3.19
LB4	4,300	2,310	1.86
LB5	4,837	2,079	2.33
LPM1	1,419	967	1.46
LPM2	1,015	545	1.86

Table 3: Accuracy of execution cycle performance contracts for multiple NFs and packet classes.

stateful performance contract, and (2) small differences between the analysed code (linked against models) and the production build (linked against real data structure implementations).

Results for our hardware-dependent metric. Table 3 shows the results for the number of execution cycles; BOLT predicts them conservatively, within a factor of 4.08× for typical workloads and 9.26× for the pathological (unconstrained) workloads. This is not surprising, given our simplistic, conservative hardware model.

If we had a more accurate hardware model, we would expect BOLT's results to be more accurate too. To validate this hypothesis without investing significant time in a sophisticated model, we performed a simple experiment.

We used three simple programs that traverse a non-contiguously allocated linked list (P_1), a linked list allocated in a contiguous chunk of memory (P_2), and an array (P_3), respectively, and had BOLT compute their performance contracts. P_1 lacks opportunities for MLP or prefetching, and BOLT's latency prediction was within 5% of the measured value. P_2 benefits from prefetching but not MLP, and BOLT's prediction was 6× higher than measured. P_3 has ample opportunity for both prefetching and MLP, and BOLT's predicted latency was 9× greater than measured latency. These measurements indicate that the more the hardware behaves like the model, the more accurate BOLT becomes. In future work, we plan to bring the model closer to real hardware.

5.2 NF operator use-cases

In this subsection, we illustrate the utility of performance contracts for NF operators and answer the following questions: Can performance contracts enable NF operators to 1) Understand the NF's performance for a variety of workloads, in particular, when the NF is under attack? 2) Reason about the performance of a sequence of NFs, in order to design and configure NF chains that meet their performance targets?

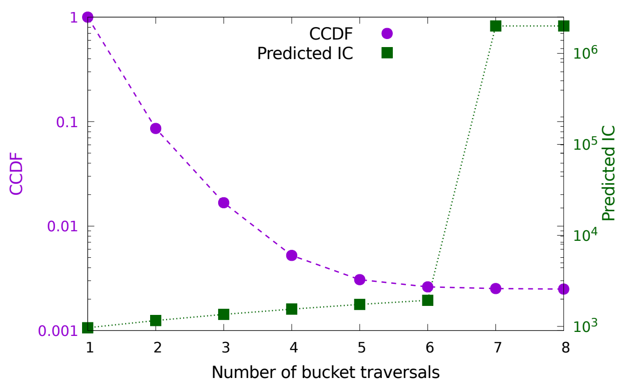

Figure 2: Predicted latency as a function of hashring bucket traversals, alongside the CCDF of traversals for a uniform random workload. The Distiller allows the operator to make an informed choice regarding where to position the threshold.

Understanding the performance of the NF under attack. Performance contracts that detail the performance of every feasible execution path through the NF code can be of particular utility to network operators for reasoning about the performance of their NF, when under attack. We use the MAC bridge as a motivating example to illustrate this use-case.

The bridge uses a MAC learning hash-table that defends itself from collision attacks by incorporating a random key in the hash algorithm. If the number of buckets traversed in the hash-table during a `put` operation exceeds a certain threshold, the key is renewed and the table re-hashed accordingly. This rehashing is designed to be a defence from attackers that know the hashing algorithm, but not the random key. However, this rehashing is particularly expensive and results in a performance cliff (Table 4).

Given its performance cost, the rehashing mechanism should be used only when a deliberate attack is suspected. The threshold that triggers the re-hashing should be carefully picked to avoid it occurring under normal circumstances. In such a scenario, the contracts and the Distiller enable the operator to easily understand the risks and trade-offs involved.

Figure 2 shows the analysis generated by the Distiller. The CCDF shows that less than 0.2% packets incur more than 6 traversals under a uniform random test workload. Setting the threshold to 6 results in the performance prediction shown in the overlaying line. The instruction count is predicted to always be less than $1939 = (144 \times 5 + 50 \times 6 + 918)$ for typical traffic.

Ability to reason about the performance of a network. Typically, operators deploy NFs in chains with packets being processed by each NF in a sequence. In these scenarios, the worst-case for one NF can often be masked by another,

Traffic Type	Instructions
Known Source MAC	$245 \cdot e + 144 \cdot c + 36 \cdot t + 82 \cdot e \cdot c + 19 \cdot e \cdot t + 882$
Unknown Source MAC; No Rehashing	$245 \cdot e + 144 \cdot c + 50 \cdot t + 82 \cdot e \cdot c + 19 \cdot e \cdot t + 918$
Unknown Source MAC; Rehashing	$245 \cdot e + 144 \cdot c + 50 \cdot t + 124 \cdot o + 82 \cdot e \cdot c + 19 \cdot e \cdot t + 14 \cdot t \cdot o + 984069$

Table 4: Bridge performance contract. Instructions are described as a function of the number of expired MAC entries (e), the number of hash collisions (c) and bucket traversals (t) incurred in the hash table, and its occupancy (o).

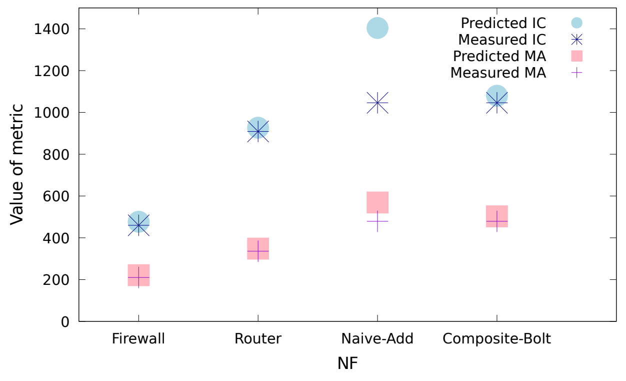

Figure 3: Composite NF of firewall + IP router. Naive-Add represents the predictions obtained from trivial addition of the individual contracts, while Composite-Bolt represents the contract for the chain produced by Bolt. Bolt's predictions are more accurate since it correctly takes into account the inter-NF dependencies.

resulting in tighter bounds for the aggregate than the sum of the parts (§3.4).

To evaluate BOLT's ability to reason about such chains, we sequence together a firewall and a static IP router. Unlike our example LPM, this router can process IP options (particularly the timestamp option [36]), but doing so can be expensive, as can be seen in the contract in Table 5b. We set up the firewall (contract shown in Table 5a) to drop any packets with any IP options included.

The combined contract for the sequence, shown in Table 5c, accurately reflects the best of both NFs. Packets with IP options are quickly dropped, incurring no further cost, and the remaining packets go through the fast path on the router.

The composition gap is further illustrated in Figure 3, the worst-case performance prediction for the composite NF is more accurate than that which would be obtained by naïvely adding their individual performance contracts.

5.3 NF developer use-cases

We now illustrate the utility of performance contracts for NF developers and answer the following questions: Can performance contracts enable NF developers to 1) Identify design flaws that lead to the NF performing poorly for certain workloads/packet traces? and 2) Pick appropriate data structure implementations when multiple implementations exist?

Debugging configuration bottlenecks. The performance contracts generated by BOLT provide the developer insight into possible performance bottlenecks. We illustrate this utility with a performance bug we found in VigNAT [4].

When dealing with traffic with high churn, VigNAT consistently incurred significant latency ($> 3\mu s$) but only for around 1.5% of packets, as can be seen in the latency CCDF in Figure 4. Building the performance contract for VigNAT (Table 6) allowed us to realise that such long tails were likely an artifact of a large number of flows expiring at once since the PCV "e" is dominant (by an order of magnitude) in the performance contract. This is further corroborated by the Distiller (Table 7), as the number of expired flows follows a similar distribution and coincides with the worse performance predictions.

As it turns out, VigNAT was inadvertently batching flow expiry. This was due to VigNAT time-stamping flows only at the granularity of a second. As a result, any packet arriving at the change of the second on the clock would induce the expiry of all of the flows that were supposed to expire during the entire previous second.

After we increased the granularity of the timestamp, ensuring a more uniform expiry of flows, VigNAT no longer exhibited such a long tail. The resulting change can be seen both in the latency CCDF (Figure 4) and Table 8 . The median per-packet latency rises since more packets are affected by flow expiry; however, the long tail has been eliminated.

Picking the appropriate data structure implementation. Often, developers need to make a choice between multiple implementations of a data structure that can deliver varying performance depending on the characteristics of the incoming traffic. In such scenarios, the predictive power of BOLT greatly simplifies this decision and lessens the need for more elaborate A/B testing.

We illustrate this utility using two implementations of the port allocator for a NAT (Allocator A & Allocator B) which differ in subtle ways. Note, that the difference in performance cannot always be captured in the big-O performance specification: both allocators are O(1) in the common case but have different constant factors in different scenarios. Allocator A, implemented as a doubly-linked list has similar constants for allocating & de-allocating a new port, regardless of churn or flow-table occupancy. Allocator B, implemented using an array and a singly linked list, has a similar

Traffic Type	Instructions
No IP options	477
IP Options	298

(a) Firewall.

Traffic Type	Instructions
No IP options	603
IP Options	$79 \cdot n + 646$

(b) Static Router.

Traffic Type	Instructions
No IP options	1053
IP Options	298

(c) Firewall+Router chain.

Table 5: NF performance contracts for the Firewall, the Static Router, and a combination of the two in a chain. Instructions are described as a function of the number of IP options in the packet (n).

Traffic Type	Instructions
Invalid packets (dropped)	$359 \cdot e + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 425$
Known flows (forwarded)	$359 \cdot e + 30 \cdot c + 18 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 1030$
New external flows (dropped)	$359 \cdot e + 30 \cdot c + 18 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 528$
New internal flows; table full (dropped)	$359 \cdot e + 30 \cdot c + 18 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 639$
New internal flows; table not full (forwarded)	$359 \cdot e + 30 \cdot c + 44 \cdot t + 80 \cdot e \cdot c + 38 \cdot e \cdot t + 1316$

Table 6: VigNAT performance contract. Instructions are described as a function of the number of expired flows (e) and the number of hash collisions (c) and bucket traversals (t) incurred in the hash table.

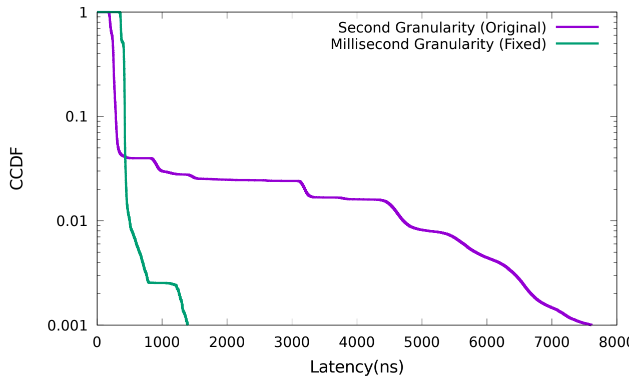

Figure 4: CCDF of packet latencies. Second granularity refers to the NF from [47]. Millisecond Granularity is the NF after the timestamp granularity was increased.

constant to allocator A for de-allocation but trades off a faster allocation at low flow-table occupancies for a much slower allocation at high flow-table occupancies.

The performance contracts capture this trade-off precisely. By specifying the characteristics of the expected traffic, the developer gains complete insight into the expected performance of the NF for each data structure under consideration. Figure 5 shows how the performance contracts precisely encapsulate the performance difference between the two implementations. Figures 6 & 7, show that the performance predicted by the contract is reflected in the performance of the NAT, for each scenario. In scenarios with a large number of long-lived flows (low churn), Allocator A outperforms Allocator B by approximately 33%, while in high churn scenarios with few, short-lived flows, Allocator B outperforms Allocator A by approximately 10%. BOLT predicts a performance difference of 30% and 8% in the two scenarios, respectively.

Number of Expired Flows	Probability Density(%)
0	98.459
$1-63$	0.0066
64	0.93
65	0.6
66+	0.0044

Table 7: Distiller report for expired flows for VigNAT for uniform random traffic, clearly indicating batching.

Number of Expired Flows	Probability Density(%)
0	16.1
1	83.6
2	0.28
3+	0.02

Table 8: Distiller report for expired flows for VigNAT for uniform random traffic after the timestamp granularity was increased. Clearly flows are expired more uniformly

6 Limitations

In this section, we describe the limitations of our current BOLT prototype.

Since BOLT builds upon Vigor [47], it requires NFs to be written with a clean stateless/stateful separation and to use a library of pre-analysed data structures. An NF that does not follow this design cannot be analysed accurately by BOLT.

The current BOLT prototype does not extend to multi-threaded NFs with shared state. We expect the biggest challenge here to be the generation of performance summaries for concurrent data structures that properly account for the effects of cross-core interference.

The current BOLT prototype works for NFs that are im-

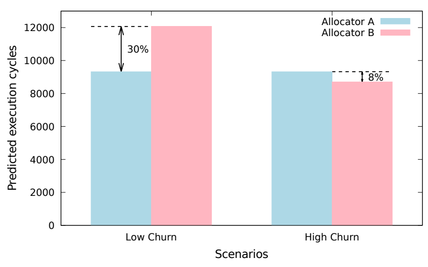

Figure 5: NAT latency predictions with both allocators in different scenarios.

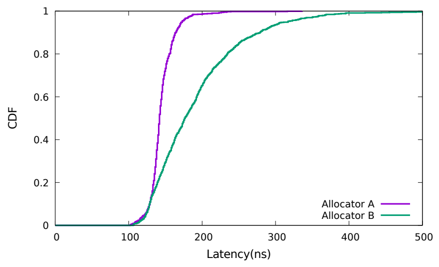

Figure 6: Allocator A outperforms Allocator B in scenarios with low churn.

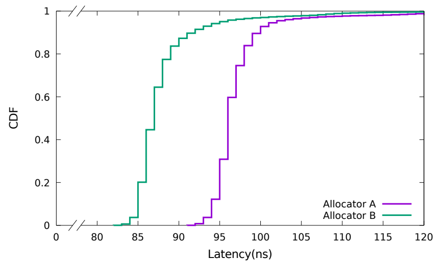

Figure 7: Allocator B outperforms Allocator A in scenarios with high churn.

plemented using the DPDK [16] framework, using a single processor core. Since DPDK is a kernel bypass framework, BOLT does not concern itself with the Linux kernel. With DPDK, best practices dictate pinning the NF to a core guaranteeing exclusive access to the L1 cache. An NF that performs Linux system calls during packet processing or that does not have exclusive non-preemptible use of a CPU core cannot be analysed accurately by BOLT.

As mentioned previously, BOLT is accurate for metrics that are independent of the underlying hardware. The slight over-estimation (7% error) arises from two sources: 1) coalescing two or more execution paths into the most expensive of them all in stateful performance summaries and 2) disabling link-time optimisations to ensure no under-approximation of performance due to differences between the analysed and production code. From our experience, the first factor is dominant and can be reduced to 0 if we choose to expose additional PCVs that are less intuitive. For hardware dependent metrics, BOLT is only as accurate as the hardware model it employs. The simple, conservative hardware model described in §3.5 leads to a 4× over-estimation for typical workloads and 9× over-estimation for pathological workloads. Additionally, it does not account for scenarios in which multiple co-located NFs contend for resources such as memory controllers [42] since it assumes a constant performance cost for memory accesses. We plan to improve the hardware model in future work.

The performance summaries for the stateful data structures were generated manually, by studying the assembly code. While the effort can be amortised across all NFs that use the data structure, the approach is laborious and potentially error-prone. In future work, we plan to automate this.

BOLT currently requires access to the NF source code, though we argue that this is not fundamental and merely an artifact of our current research prototype. Having access to the source significantly simplifies the manual analysis of the stateful code and allows us to attach semantic meaning to PCVs. That said, much of the needed information could also potentially be gleaned from debug information that compilers optionally include in binaries or otherwise deduced via reverse engineering. Likewise, access to the source facilitates attaching semantic meaning to different traffic classes in the stateless code and also gives us more freedom in preparing analysis builds for the SEE. While our current SEE analyses LLVM bit-code, which requires a special build process, other engines [11] can directly analyse X86 binaries and our stateful/stateless separation could be rendered as separate object files. We leave redesigning our system around such considerations to future work.

BOLT currently quantifies performance in terms of three metrics (IC, MA, cycles) that provide a concrete first step into understanding NF performance. Nevertheless, we plan to extend BOLT to reason about more commonly used metrics such as throughput and end-to-end latency. We expect the major challenge to arise from modelling of the PCIe bus, the NIC and queueing delays (for latency) and modelling instruction and memory level parallelism (for throughput).

7 Related Work

Performance evaluation and diagnosis. There exists a large body of work that focuses on generating and analysing adversarial workloads that attack software performance. [5, 12, 39] describe manually generated, adversarial attacks on specific data structures and network applications (e.g., IDS). Others generate these workloads automatically: [33, 35, 41] use fuzzing to find bottlenecks in individual methods and data structures, [29] automatically detects certain complexity attacks in web services and [32] automatically generates adversarial inputs for NFs. All of these systems, focus only on adversarial workloads, while performance contracts characterise performance in the face of any arbitrary workload, whether typical, ideal or adversarial.

Others focus on deriving formal upper bounds on performance: Worst-Case Execution Time (WCET) Analysis [45]. Again, as with adversarial workloads, this only looks at one aspect of the performance profile: the absolute worst-case. These techniques are popular in the real-time systems domain where performance guarantees are a part of functional correctness. However, because of this requirement, real-time systems tend to avoid dynamic data structures and input-dependent memory accesses, aspects that are commonplace in NFs. Though not primarily designed as a WCET analysis tool, BOLT can also be used to deduce worst-case bounds (when generating contracts without any assumptions).

Performance side-channel attacks have also been analysed using static analysis techniques. [7] uses symbolic analysis to identify cost differentials between execution paths, rather than to predict absolute performance.

Another large body of work predicts performance for large-scale data analytics applications [30, 43]. This work solves a different problem from Bolt and operates on a different scale with different challenges and effects to account for. Rather than predicting performance for varying workloads on a given hardware platform, this work focuses on predicting performance for a given workload but on varying hardware configurations.

Other instances of previous work address the same problem as Bolt, but operate under different assumptions and thus tackle different challenges. Like Bolt, [28] makes parametric performance predictions, but for probabilistic programs that implement randomised algorithms where the challenge lies in the probabilistic reasoning, rather than cache-effects and other low-level details. TiML [44] also similarly predicts performance but requires applications to be implemented in the TiML functional language which uses type annotations to enable more rigorous reasoning about performance bounds. While Bolt assumes that NFs use a pre-analysed data structure library, it permits developers to continue to use low-level languages like C, as is the norm for NFs. Bolt also reasons about low-level hardware behaviour, such as cache effects.

Runtime performance analysis monitors systems during execution to identify performance issues; we focus here on work related to NFs. NFVPerf [27] passively monitors traffic to find hardware and software bottlenecks in software NFs. PerfSight [46] extracts low-level performance information from software data planes and allows operators to find which network functions are responsible for performance issues. FlowTags [19] modifies middleboxes to tag sent packets and use tags from received packets to enforce network-wide policies, including performance.

Program Analysis Applied to NFs. Several instances of prior work have proposed using static analysis to help understand, debug, and verify software NFs. StateAlyzr [23] statically analyses NFs to identify their per-flow and global state to enable efficient state migration and redistribution. Other approaches, use symbolic execution to find bugs or formally verify correctness. [9, 10, 48] leverage this technique to automate bug finding and test-case generation. [40] symbolically executes NF models to reason about network properties like reachability and loops. [14, 47] use exhaustive symbolic execution to formally verify functional correctness.

8 Conclusion

In this work, we propose the notion of performance contracts for NFs. Performance contracts precisely characterise NF performance for any arbitrary incoming workload, whether typical, ideal or adversarial. We express performance contracts in terms of Performance Critical Variables (PCVs) which succinctly characterise how NF state and configuration parameters affect the performance of the incoming packet. Additionally, we present and evaluate BOLT, a tool that automatically derives performance contracts with accurate performance predictions from the NF code. Finally, we walk through a series of use-case scenarios that illustrate how network operators and NF developers can use BOLT to understand and mitigate NF performance unpredictability.

References

[1] Bolt Project Repository. https://github.com/bolt-perf-contracts/bolt.

[2] Intel®64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[3] LPM library - data plane development kit 18.08.0 documentation. https://doc.dpdk.org/guides/prog_guide/lpm_lib.html.

[4] VigNAT source code repository. https://github.com/vignat/vignat.

[5] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Making DPI engines resilient to algorithmic complexity attacks. *IEEE/ACM Trans. on Networking*, 2016.

[6] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[7] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu. Symbolic path cost analysis for side-channel detection. In *Intl. Symp. on Software Testing and Analysis*, 2018.

[8] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[9] M. Canini, D. Kostic, J. Rexford, and D. Venzano. Automating the testing of OpenFlow applications. *Intl. Workshop on Rigorous Protocol Engineering*, 2011.

[10] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test openflow applications. In *Symp. on Networked Systems Design and Implem.*, 2012.

[11] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[12] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symp.*, 2003.

[13] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[14] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Symp. on Networked Systems Design and Implem.*, 2014.

[15] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Symp. on Networked Systems Design and Implem.*, 2012.

[16] DPDK: Data plane development kit. https://dpdk.org.

[17] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Symp. on Networked Systems Design and Implem.*, 2016.

[18] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A scriptable high-speed packet generator. In *Internet Measurement Conf.*, 2015.

[19] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Symp. on Networked Systems Design and Implem.*, 2014.

[20] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification*, 2007.

[21] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[22] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 2015.

[23] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr. In *Symp. on Networked Systems Design and Implem.*, 2016.

[24] M. Kuhnemann, T. Rauber, and G. Runger. A source code analyzer for performance prediction. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[26] K. Meng and B. Norris. Mira: A framework for static performance analysis. In *2017 IEEE International Conference on Cluster Computing*, 2017.

[27] P. Naik, D. K. Shaw, and M. Vutukuru. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *IEEE Conf. on Network Function Virtualization and Software Defined Networks*, 2016.

[28] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Intl. Conf. on Programming Language Design and Implem.*, 2018.

[29] O. Olivo, I. Dillig, and C. Lin. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Conf. on Computer and Communication Security*, 2015.

[30] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Symp. on Operating Systems Principles*, 2017.

[31] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing protocol implementations for interoperability. In *Symp. on Networked Systems Design and Implem.*, 2015.

[32] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *ACM SIGCOMM Conf.*, 2018.

[33] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conf. on Computer and Communication Security*, 2017.

[34] S. Pirelli, A. Zaostrovnykh, and G. Candea. A formally verified NAT stack. In *ACM SIGCOMM Workshop on Kernel-Bypass Networks*, 2018.

[35] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Real-Time Systems Symp.*, 1998.

[36] RFC 781. https://tools.ietf.org/html/rfc781, 1981.

[37] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Symp. on Networked Systems Design and Implem.*, 2012.

[38] V. Sekar and P. Maniatis. Verifiable resource accounting for cloud computing services. In *Cloud Computing Security Workshop*, 2011.

[39] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a NIDS. In *Annual Computer Security Applications Conf.*, 2006.

[40] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM Conf.*, 2016.

[41] L. D. Toffola, M. Pradel, and T. R. Gross. Synthesizing programs that expose performance bottlenecks. In *Intl. Symp. on Code Generation and Optimization*, 2018.

[42] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling slos in network function virtualization. In *Symp. on Networked Systems Design and Implem.*, 2018.

[43] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Symp. on Networked Systems Design and Implem.*, 2016.

[44] P. Wang, D. Wang, and A. Chlipala. Timl: A functional language for practical complexity analysis with invariants. 2017.

[45] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.

[46] W. Wu, K. He, and A. Akella. PerfSight: Performance diagnosis for software dataplanes. In *Internet Measurement Conf.*, 2015.

[47] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea. A formally verified NAT. In *ACM SIGCOMM Conf.*, 2017.

[48] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Intl. Conf. on Emerging Networking Experiments and Technologies*, 2012.