1 Introduction

Semantic interfaces (e.g., abstract classes, specifications, header files, documentation) succinctly describe a program's externally visible functional behavior, enabling engineers to use the system productively. This makes it possible for programmers to use a lot of third-party code and makes infrastructure operators comfortable with deploying software they did not write.

We do not know of an equivalent construct for describing performance behavior in a way that is simultaneously succinct, precise, complete, and human-readable. Engineers reason about performance in terms of envelopes (e.g., "runs in $O(n)$ time") and benchmarks, which implies that they deploy their system without understanding the entire spectrum of performance it can exhibit. As a result, untested inputs can exercise mysterious code paths that lead to unexpected performance behavior [4, 36, 39] and a perpetual need to fix performance bugs [35, 43].

In this paper, we explore the idea of a *performance interface*: a description of a system's performance behavior that is simultaneously succinct, precise, and human-readable. What should such an interface look like? Like a good semantic interface, it should be "much smaller and simpler than the code" [48], so it must abstract away certain details—but which ones? Performance problems often lie in low-level implementation details as well as the code's interaction with the environment (e.g., specifics of the underlying hardware's cache hierarchy). Is it possible to capture all the relevant performance behaviors of a system while being "much smaller and simpler" than the system itself?

We propose that the performance interface of a system be a *program* that accepts the same inputs as the system and outputs how long the system would take to process the given input[1]. A performance interface has a *resolution*, which quantifies the smallest change in performance that it specifies (e.g., 50 ns, 1 mem-op)—the coarser the resolution, the simpler the interface. We distinguish a *deployment-specific* interface from a *general-case* one: The former is much simpler and of greater interest to an operator, who wants to understand the system's performance behavior in her specific environment, while the latter is most useful to developers. This distinction, along with resolution, makes it possible to have performance interfaces that capture only those behaviors that are relevant to the case at hand. In other words, the two concepts enable abstraction of performance behavior.

We concretize our proposal in the context of network functions (NFs), i.e., load balancers, firewalls, NATs, etc. NFs are typically on the critical path of serving a user request and often face unpredictable traffic coming from the outside world. For instance, any packet that enters a service provider's data center traverses at least one load balancer/reverse proxy and typically also a firewall—the latency that each NF adds to the packet directly impacts the user-perceived latency. A recent survey [54] of network operators found NF performance degradation to be a frequent pain point, and such performance bugs to be among the hardest to diagnose.

To make NF performance interfaces useful today, we developed PIX (**P**erformance **I**nterface e**X**tractor). PIX takes as input NF code written in C and outputs general-case performance interfaces in the form of small Python programs that it can then specialize into deployment-specific interfaces for individual deployments. PIX currently supports three latency-related metrics: number of instructions, number of memory operations, and number of CPU cycles. For each metric, PIX outputs one set of Python programs; each set contains one Python program per relevant range of resolutions. All PIX-extracted performance interfaces are specific to the CPU's ISA. Further, the interfaces for CPU cycles are specific to

[1] In this paper we focus on system latency, not throughput.

the micro-architecture of the underlying hardware and assume that the NF does not contend for hardware resources with other processes (i.e., assume either smart process co-location [12, 31, 52] or process isolation, using techniques such as cache partitioning [77]). Under the covers, PIX employs symbolic program analysis techniques to reason about the NF's performance behaviors.

We evaluate PIX on 12 open-source NFs, including the Katran load balancer [71] used at Facebook, the Natasha NAT [58] used at Scaleway and the XDP packet filter from the Cilium project [14]. All 12 NFs were written using either the Linux kernel's eBPF XDP [82] framework or the DPDK [21] kernel-bypass framework, two of the most popular ways to develop high-performance NFs. Our evaluation shows that the extracted performance interfaces are accurate yet orders of magnitude simpler than the code, and take minutes to obtain. We show how performance interfaces extracted by PIX can be used to identify performance regressions, diagnose and fix performance bugs, and identify the latency impact of NIC offloads.

In summary, we make two contributions in this paper:

- We propose the concept of performance interfaces, which leverages the notions of performance resolution and deployment-specific interfaces to enable abstraction of performance behavior.
- We demonstrate that it is feasible to build a tool that automatically extracts performance interfaces from NF code, and that these interfaces can be accurate-yet-simple enough to help understand and debug performance.

In the rest of the paper, we describe how we think a performance interface should look like (§2). Then, we describe PIX (§3) and use it to evaluate the feasibility and utility of performance interfaces for NFs (§4). Finally, we discuss how PIX can generalize to systems beyond NFs (§5), related work (§6), and conclude (§7).

2 Performance Interfaces

In this section, we present our proposal for performance interfaces, and describe how we envision them being used.

Target audience: We target two categories of audience for any system: The *developers* write the code for the system and are familiar with its low-level implementation details, but not necessarily with all possible performance behaviors it can exhibit. The *operators* did not write the code but instead seek to use/deploy/build on top of the system in their respective environments. They are unfamiliar with and do not necessarily want to understand its low-level details. Further, unlike the developers who care about the system's performance in all settings, they care primarily about its performance in their specific use-case/deployment. These categories can vary from system to system—the developer of an application A might themselves be building upon on a network stack B, making them an operator for that stack.

Design goals: We envision that a "performance interface" must describe the system's externally visible *performance* behaviors, just as a semantic interface describes a system's externally visible functionality [48].

The primary challenge in summarizing performance is that systems typically expose a greater variety of performance behaviors than semantic ones. Hence, a performance interface that perfectly predicts every possible performance behavior would likely be so complex that it wouldn't deserve to be called an interface.

We look for a compromise, i.e., a way to summarize performance that achieves a good balance between the following properties: (1) **Accuracy**, i.e., the ability to summarize performance completely (for every possible input) and precisely (with a small error). (2) Simplicity, i.e., being **smaller** than the code and as **abstract** as possible—summarize performance in terms of primitives appropriate for a semantic interface of the system, and reveal implementation details only when necessary.

We also aim for (3) **Portability**. A system's performance may depend significantly on its environment (e.g., workload, hardware). For instance, adversarial traffic causing L3 cache misses can degrade NF latency by 3× [64]. The interface should make it easy to quantify the impact of a particular environment on performance, enabling porting of the interface across deployments.

State of the art: Today, performance is typically summarized through upper bounds—Big-Oh notation or worst-case execution time [79]—and statistics (e.g., x-th percentile latency). These descriptions maximize simplicity at the cost of accuracy—there are many inputs for which they do not provide accurate predictions.

We draw inspiration from two recent proposals that describe a system's performance behavior as performance annotations [69] and performance contracts [41] respectively. Freud [69] describes a method's performance as a performance annotation: a set of ⟨input/global-variable constraints, performance formula⟩ tuples, and each formula is a mathematical function of the method's input and/or global variables. Bolt [41] describes an NF's latency as a "performance contract:" a set of ⟨input constraints, performance formula⟩ tuples, where each formula is a function of the system input and "Performance-Critical Variables" (PCVs).

Since we reuse the idea of PCVs from performance contracts [41], we elaborate upon them here. A PCV is a parameter that captures the influence on performance of all factors other than the input packet (e.g., NF configuration, state built up by prior packets, hardware characteristics, etc). A PCV is not always an explicit variable in the NF implementation, rather it can be an implicit "ghost" variable [29, 32]. For instance, if an NF employs a hash table, a PCV could be the "number of collisions" encountered by the current packet—this ghost variable allows latency to be expressed as a function of, among other things, the number of collisions. Independent prior work [37, 38] on symbolic bounds has also argued for the use of a PCV-like abstraction to succinctly summarize the performance behavior of stateful programs.

However, neither annotations nor contracts were designed to be "performance interfaces". Contracts sacrifice simplicity for accuracy—they include a list of input constraints, which can be as many as the number of execution paths through the system and reveal low-level implementation details *even when unnecessary*. Annotations do not meet our accuracy goal as they do not capture how performance depends on state built from past inputs, e.g., the contents of a hash table or a hardware cache (Appendix A).

Definition: The *performance interface* of a program P with procedures $p_1, p_2, \ldots$ is a program $S_P = \{p'_1, p'_2, \ldots\}$. A procedure $p'_i \in S_P$ takes the same inputs as the corresponding $p_i \in P$ and returns the performance of executing p_i. This return value corresponds to a performance metric (e.g., # of `x86` instructions, # of CPU cycles). The *resolution* r of S_P is the smallest difference in performance that S_P can specify: if $\mathcal{P}(p_i(I))$ is p_i's performance given input I, then $|p'_i(I) - \mathcal{P}(p_i(I))| < r, \quad \forall p_i, I$.

A performance interface can be for the "general case" or specific to a deployment.

In a *general-case performance interface*, the procedures p'_i compute performance as a function of PCVs [41]. PCVs ensure that the interface can describe the performance of each p'_i in full generality, i.e., for arbitrary workloads and hardware configurations.

A *deployment-specific performance interface* is simpler than the general-case one and does not contain PCVs. Instead, procedure p'_i returns performance as a statistic (e.g., median, max, 99$^{\text{th}}$ percentile), computed for a given joint probability distribution of the PCVs that describes P's environment for a particular deployment. In this work, an NF's deployment environment is defined by its configuration read at startup, a representative workload, and the specific hardware it runs on.

Example: We illustrate with an example implementation of a MAC learning bridge (Fig. 1) that uses a fast MAC table, implemented in hardware, and a slow software-based table, based on a cuckoo hash table. Table 1 shows the performance cost of this implementation's procedures in terms of executed lines of pseudocode (LOP), a performance metric we use for illustration only. For now, we assume these costs, we elaborate on how they are obtained in §3.

```
void bridge(pkt* p, time_t now) {
  expire_stale_ports(now);
  if (invalid_hdr(p)) {
      DROP(p);
      return;
  }
  /* Learning source MAC addr */
  if(!slow_MACtable_get(p->src_mac, &p->port))
     slow_MACtable_put(p->src_mac,&p->port);
  else
     slow_MACtable_update(p->src_mac, now);
  /* Forwarding based on dest MAC addr */
  if (fast_MACtable_get(p->dst_mac,&out_port))
     FORWARD(p,out_port);
  else if (slow_MACtable_get(p->dst_mac,&
    out_port))
     FORWARD(p,out_port);
  else
     BROADCAST(p,p->port);
}
```

Figure 1. Example implementation of a MAC learning bridge

Operation	**Performance [LOP]**
`expire_stale_ports()`	40 + 60× n_stale
`invalid_hdr()`	5
`DROP`	1
`FORWARD`	60
`BROADCAST`	200
`fast_MACtable_get()`	10
`slow_MACtable_get()`	50
`slow_MACtable_update()`	70
`expire_stale_ports()`	40 + 60 × `n_stale`
`slow_MACtable_put()`	110 + 80 × `n_evicted` + 120 × `occ` × `rehashing`

Table 1. General-case performance of procedures called by the code in Fig. 1. Two have non-constant performance: expiring learned ports is linear in the number of stale ports, and doing a `put()` in the cuckoo hash table depends on the number of keys that must be evicted and whether rehashing is necessary.

Fig. 2 shows two performance interfaces of this implementation. Since it exposes a single procedure, the performance interface also has a single procedure. The resolution of the performance interfaces is $r = 50$ LOP.

The general-case interface gives performance as a function of 4 PCVs: number of stale flows (`n_stale`), hash-table occupancy (`occ`), number of hash-table evictions triggered by this input (`n_evictions`), and whether rehashing is needed (`rehashing=`1 if yes, 0 otherwise). Since the performance metric LOP is independent of the underlying hardware, all 4 PCVs are specific to the

```
def perf_bridge_gc(p,now):
  # Metric: LOP, Resolution: 50
  # NF state: slow_MACtable, fast_MACtable

  if invalid_hdr(p):
    return 46 + 60* n_stale
  if fast_MACtable_get(p->dst_mac) or
     slow_MACtable_get(p->dst_mac):
    return 280 + 60* n_stale + 80*
     n_evictions + (120* occ) * rehashing
  else
    return 445 + 60* n_stale + 80*
     n_evictions + (120* occ) * rehashing
```

```
def perf_bridge_ds(p,now):
  # Metric: LOP, Resolution: 50
  # Statistic: 50th percentile
  # NF state: slow_MACtable, fast_MACtable

  if invalid_hdr(p):
    return 106 #(46+60)
  if fast_MACtable_get(p->dst_mac) or
     slow_MACtable_get(p->dst_mac):
    return 340 #(280+60)
  else
    return 505 #(445+60)
```

Figure 2. General-case (left) and deployment-specific (right) performance interfaces for the bridge (Fig. 1). Each return value in the latter is the median LOP executed for the assumed PCV distribution.

bridge's implementation. If the bridge stored the MAC table using a binary tree instead of a cuckoo hash table, the interface would describe performance using different PCVs (`tree_depth` instead of `rehashing`).

The deployment-specific interface gives the median latency for a deployment where the expected workload is such that 50% of input packets encounter no hash collisions and expire ≤ 1 stale ports. The interface produces concrete numbers corresponding to this deployment-specific PCV distribution. Note, the deployment-specific interface does not restrict the inputs (e.g., the types of packets), it only instantiates the PCVs.

This performance interface captures all the performance behaviors of the bridge that are externally visible at resolution r=50. It is accurate, in that it correctly predicts performance (at the given resolution) for every possible input. It is smaller and simpler than the implementation: each procedure considers only three operations (invalid header check, fast table lookup, and slow table lookup), since these are the only ones that affect performance at r=50. Unlike the general-case interface, the deployment-specific interface makes assumptions about the expected workload.

Why represent the interface as a Python program? We believe that an interface that presents performance like the system itself—through code that branches on the input—is more intuitive than a list of input constraints for developers and operators. We chose Python due to its ubiquitous use [33].

Resolution: Often, developers and operators do not care about certain performance differences, either because they do not affect their performance targets, or because they are masked by the environment. For example, developers building minute-scale applications may not care about µs-scale variability in the networking stack, while those building µs-scale ones typically do.

The notion of resolution enables the developer/operator reading the interface to choose between multiple levels of abstraction (trading off accuracy for simplicity) in a controlled manner. A performance interface at a specified resolution only differentiates between input classes whose performance differs by more than the resolution—implementation details that cause variability relevant to the specific developer/operator are abstracted away. In our bridge example, a performance interface with a resolution of 1 LOP must report the performance of each forwarding behavior separately; an interface with resolution >= 45 LOP can abstract away the difference between a fast and slow lookup, and an interface with resolution >= 115 LOP can abstract away the difference between a successful and unsuccessful lookup.

Picking the right resolution: We envision developers/operators picking their respective resolutions based on the performance variability they are willing to tolerate in their deployment scenarios. In §3, we show how PIX goes a step further for those unsure of the "right" resolution, by identifying a minimal set of resolution thresholds that yield all the possible different performance interfaces. This is possible since the performance interface can only elide each implementation detail at a distinct resolution threshold, which results in it not changing between two such thresholds. In our bridge example, {1, 20, 45, 115, 210} is such a minimal set of resolution thresholds, i.e., other resolutions don't yield different interfaces (e.g., the interface at $r = 50$ is identical to that at $r = 46$). By identifying these resolution thresholds, PIX enables developers and operators to easily pick the resolution (and corresponding interface) that achieves the desired trade-off between accuracy and simplicity.

Deployment-specific interfaces: We chose to have separate general-case and deployment-specific interfaces to provide a different balance between accuracy and simplicity for operators and developers respectively.

General-case interfaces are meant for developers. Developers cannot always predict where/how their code will be deployed, and are hence often interested in the performance of their system when deployed in arbitrary environments. The general-case interface provides them with such a description by summarizing the impact of the environment on the system's performance using PCVs. While PCVs do reveal implementation details (e.g., `n_evicted`, `rehashing` reveal the use of a cuckoo-hash table), these details *are necessary* to summarize performance *for an arbitrary workload*, so they must be represented in the general-case interface.

We designed the deployment-specific interface for operators. Since operators are unfamiliar with the system's implementation and only care about the system's performance behavior in their particular deployment environment, the deployment-specific interface does away with the hard-to-understand PCVs by instantiating them with a distribution specific to that deployment. This enables the deployment-specific interface to summarize performance in an NF-generic way—any NF would normally involve a header check and state lookups—and be understood by almost any NF operator. Of course, it does reveal one important aspect of the implementation, namely the distinct fast and slow tables. However, this aspect (which would have no place in a semantic interface) is crucial to any bridge operator interested in performance.

That said, we do not envision the separation between the general-case and the deployment-specific interfaces being set in stone—developers may refer to the deployment-specific interface to understand performance in the face of specific workloads, while operators may refer to the general-case interface to understand performance beyond their expected workload.

3 Extracting Performance Interfaces

We now describe PIX, which takes as input an NF implemented in C and automatically extracts performance interfaces in the form of Python programs.

We designed PIX to meet two goals: (1) *miniminal developer effort*: developers/operators should not need to write performance test suites or proof lemmas, and (2) *allow for proprietary NFs*: NF vendors typically provide operators with only binaries [55]; it's ok for them to provide a performance interface, but not source code.

Fig. 3 presents an overview of PIX. The NF developer gives the PIX back-end the NF source, augmented with a few single-line annotations akin to instantiating a type in a higher-level language. PIX combines this with a pre-analysis of the data structures used by the NF and extracts the general-case interfaces for all meaningful resolution ranges. The NF operator provides the PIX front-end with an NF binary and general-case interface (provided by the NF developer), along with a (set of) packet trace(s) that represent the expected workload in their deployment. From these, PIX extracts the deployment-specific interfaces for all meaningful resolution ranges. NF developers/operators can also query PIX with a specific resolution, to get the interface at that resolution.

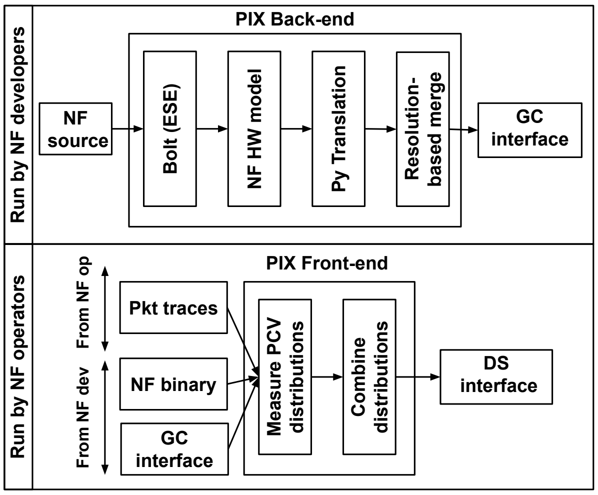

Figure 3. Overview of PIX. GC and DS refer to general-case and deployment-specific respectively.

Limitations and Assumptions: The PIX back-end uses exhaustive symbolic execution (ESE) [45] to automatically analyze the NF code. For this to work, the NF needs to be single-threaded, all its loops except the top-level event loop must have statically computable bounds, and it must keep all history-dependent state in data structures with clear interfaces. PIX cannot extract performance interfaces for NFs that do not meet these requirements.

Many (not all) data-plane NFs meet these requirements. For instance, many NFs are written using the eBPF [82] framework as stateless modules that keep their state in cleanly separated, kernel-maintained eBPF maps [24]. Other examples include recently proposed NF frameworks that build upon the DPDK kernel-bypass framework [21] like FastClick [5] and Vigor [83], which impose the use of a specific set of well-separated data structures to store NF state. Counterexamples include Intrusion Detection Systems (IDSes) and TCP-terminating NFs; in general PIX cannot extract interfaces for them and we describe this limitation in more detail in §5.

PIX-extracted interfaces only summarize the processing latency for each packet and do not reason about queuing latencies. Reasoning about these latencies would require PIX to reason about multiple inputs together, and for this, we need to employ techniques more sophisticated than ESE [83]. Reasoning only about processing latency allows PIX to avoid reasoning about load-based variability since processing latency (unlike queueing latency) does not vary with load.

To capture how hardware affects performance with reasonable accuracy, PIX assumes that the NF runs pinned to a core and does not significantly contend for hardware resources, e.g., due to smart process isolation [12, 31, 52, 77]. We believe network operators keen on predictable performance are likely to employ such techniques.

Implementation: PIX builds on the KLEE symbolic execution engine [9]. We extended KLEE with 4629 lines of C++ code to implement the first two steps of the back-end. We synthesize the Python-based interfaces using 1825 lines of OCaml. We implemented resolution-based merging and the PIX front-end in 1221 lines of Python.

3.1 Extracting general-case interfaces

We now describe how the PIX back-end extracts general-case interfaces from NF source code.

Step 0: Pre-processing: This step outputs the performance of each execution path of the NF in terms of hardware-independent metrics as a function of PCVs specific to the NF's implementation (we refer to these PCVs as *hardware-independent PCVs* henceforth). PIX currently supports two hardware-independent metrics—instruction count and memory-access count. We call this Step 0 because it is not part of our contribution, we largely reuse the approach and tool from Bolt [41].

Bolt relies on the observation that data-plane NFs tend to use the same, relatively few data structures, mainly hash tables/maps and buffers/rings. One can therefore collect these data structures in a library, have an expert "pre-analyze" them once, and then amortize this analysis cost across all NFs that use the library. Bolt's pre-analysis consists of two manual tasks for each call in the library's interface: (1) identify the PCVs relevant to that call, and (2) write a simple symbolic model of the call. Such manual effort is reasonable because it is a rare effort (e.g., once per update to the Linux kernel's eBPF maps) and it is done by the maintainer of the data structure library instead of its users. To illustrate, there were 34 new commits in Linux's eBPF maps last year [23] while the Cilium project [14] alone—just one among hundreds of projects that leverage eBPF maps—had an order of magnitude more commits during that same period [13]. Further, independent prior work [38] has observed that most data structures require only a "few" PCVs, and identifying them is "straightforward". Our experience as the "experts" for this work corroborated this observation—identifying PCVs required only single-line loop annotations which took ≤ 1 person-hour for someone familiar with the data structure code.

The Bolt tool takes as input the NF source code, as well as the symbolic models and loop annotations for the state-accessing calls made by the NF; and outputs the performance of each execution path through the NF as a function of the hardware-independent PCVs.

In Step 0, PIX uses the Bolt tool as stated above, and also automatically instruments the NF code such that it can log the values of the hardware-independent PCVs for each input packet encountered.

Step 1: NF-domain hardware model: This step characterizes the performance of each execution path of the NF in terms of hardware-dependent metrics (CPU cycles), by introducing hardware-dependent PCVs; i.e., PCVs that capture the interaction between NF and hardware.

PIX uses the notion of a CPI (Cycles Per Instruction) stack [27] to compute the number of CPU cycles of an execution path. A CPI stack breaks down the average CPI for a program executing on a given microprocessor into a base CPI plus various CPI components that reflect "lost" cycle opportunities due to miss events such as branch mispredictions and cache/TLB misses. In general, replicating a perfect CPI stack is infeasible—it is equivalent to analyzing each execution path to the depth provided by a cycle-accurate simulator.

We leverage NF-domain knowledge to eliminate CPI components and pick only the necessary set of hardware-dependent PCVs. When an NF runs pinned to a core and with limited contention for hardware resources, the dominant hardware factor that influences its performance is the last-level cache (LLC) [20, 52, 77]. Hence, PIX introduces only two hardware-dependent PCVs—*base_CPI* and *LLC_miss_latency*—and expresses a path's CPU cycle count as $instructions \cdot base\_CPI + LLC\_misses \cdot LLC\_miss\_latency$. Note, while PIX uses the same two PCVs for all NFs, the values of these PCVs vary with each <*NF, HW*> pair (§3.2). To track possible LLC misses, PIX leverages taint-analysis [70] to identify independent heap accesses specific to the current input; it then branches on each such access, with one outcome being an LLC miss and the other an LLC hit.

Step 2: Python program: The previous steps specify an NF execution path as a set of symbolic constraints on the input packet and symbols arising from calls to data structures; this step translates these constraints into human-readable python code and outputs a general-case performance interface of the NF with a resolution of 1.

PIX translates symbolic constraints on the input packet using knowledge of the header format of the popular networking protocols (e.g., IPv4, TCP, QUIC). For instance, the constraint $pkt[23:24] == 6$ on a non-tunnelled IPv4 packet is translated to $pkt.isTCP$.

```
1 # Developer annotation:
2 DS_INIT(&map,"macTable","ethaddr", struct
      eth_addr,"port", int);
3
4 # Starting condition derived from implem:
5 if bpf_map.unnamed_symbol
6 # Transform based on called library function
7 if bpf_map.contains(pkt[7:12])
8 # Transform based on developer annotation
9 if macTable.contains(pkt.src_mac)
```

Figure 4. Example of PIX's constraint rewriting.

PIX translates symbols arising from calls to data structures using call context and developer-provided annotations (one annotation per instantiated data structure). Fig. 4 illustrates such a translation: Line 2 shows a developer's annotation for a data structure of type `map`: it indicates that this NF uses this map as a `"macTable"`, which maps `"ethaddr"` keys to `"port"` values; these are human-friendly terms chosen by the developer to help the generation of simple performance interfaces. Line 5 shows a constraint derived from the NF code that concerns this map. Line 7 shows how PIX rewrites this constraint because it knows that this is a call to `bpf_map_lookup_elem()` with an argument corresponding to bytes $7-11$ of the input packet. Line 9 shows how PIX further rewrites the constraint because the developer's annotation enables PIX to identify the given bytes as the input packet's source MAC address.

The annotation on Line 2 is the only annotation that the NF developer needs to provide. We believe such one-line annotations are reasonable since they are similar to instantiating a type in a higher-level language.

Step 3: Resolution-based merging: This step uses the notion of resolution to simplify the performance interface: First, it calculates the maximum performance impact of each constraint, i.e., the maximum performance difference between two execution paths that only differ w.r.t this constraint. The set of distinct "maximum performance impacts" forms the minimal set of resolution thresholds. Second, it eliminates all constraints with an impact smaller than the target resolution.

3.2 Extracting deployment-specific interfaces

To extract a deployment-specific interface, the PIX front-end takes as input the NF binary and its general-case interface[2], provided by the NF developer/vendor; along with a (set of) deployment-specific packet trace(s), provided by the NF operator. It then runs the NF binary using the packet trace(s) as input, infers the deployment's PCV distributions, and instantiates the deployment-specific interface. Running the NF allows PIX to extract accurate deployment-specific interfaces since it can precisely measure the performance impact of the NF's environment as opposed to modeling it.

PIX infers three PCV distributions per NF, deployment:

Hardware-independent PCVs: PIX leverages the instrumentation introduced in Step 0 to measure the values of these PCVs encountered by each packet in the provided trace(s). It then computes a joint probability distribution of these PCVs, since they tend to be highly correlated (e.g., in Table 1, `n_stale` and `n_evictions` are both functions of `occ`).

Base CPI: PIX measures this using hardware performance counters [75] available on all major processors today. Since the packet trace(s) may not exercise all execution paths, PIX assumes the same base-CPI distribution across all paths, and it provides warnings if it detects significant differences (e.g, some paths use expensive x86 instructions, like integer divide, while others don't). We think this is a reasonable assumption because the base CPI is only a function of the instruction mix (it does not include any miss events). In §4.1, we experimentally validate this.

LLC miss latency: Measuring the distribution of LLC miss latency ideally requires sophisticated NF-specific testing [64], to account for the NF's particular instruction- and memory-level parallelism. PIX avoids this because it targets NFs that keep all their state in a relatively small set of pre-analyzed data structures. For each data structure, we craft a microbenchmark that triggers LLC misses.[3] PIX estimates the LLC-miss-latency distribution of each data-structure call in a given deployment, by running the corresponding microbenchmark on the deployment's hardware. In §4.1, we experimentally show that our approximation, performs well in practice (avg. error of $<$ 10%). Note, our approximation concerns the *latency* introduced by LLC misses, *not the number* of LLC misses—the PIX back-end tracks LLC misses per path in Step 1.

Finally, PIX instantiates each formula in the general-case interface with these inferred distributions to compute the requested latency statistic (e.g., 50^{th} percentile in Fig. 2). We show examples of deployment-specific interfaces and their distributions in §4.

[2] The operator cannot be certain that this general-case interface is accurate for the production binary, but we do not see this as a barrier to adoption: operators routinely deploy NF binaries while relying only on non-attested configuration interfaces and vendor manuals [55].

[3] The expert must do this once per data structure, like the pre-analysis.

Framework	NF	Functionality
eBPF XDP	Katran LB	Per-flow state, per-VIP state, consistent hashing, IPv6, ICMP, QUIC, tunneling
	Cilium filter	Longest prefix matching, IPV6
	CRAB LB	Read-only state
	hXDP firewall	Per-flow state
DPDK	Natasha NAT	Per-flow state, handles fragmentation, UDPLite, ICMP, ARP
	Maglev LB	Per-flow state, consistent hashing
	VigNAT	Per-flow state, header rewriting
	Bridge	Packet duplication
	Router	Longest prefix matching
	Policer	Per-flow state, fine-grained timing
	DPDK NAT	Per-flow state, header rewriting, cksum offload
	DPDK firewall	Per-flow state

Table 2. Network functions used to evaluate PIX.

4 Evaluation

In this section, we address two main questions: (1) does PIX extract good performance interfaces, and (2) can performance interfaces make NF developers and NF operators more productive? To answer the former, we quantitatively evaluate the complexity of PIX-extracted interfaces, their accuracy, and the time it takes to obtain them (§4.1). We find that they are one to two orders of magnitude simpler and more accurate than prior work. To answer the latter question, we show how developers can use PIX-extracted interfaces to catch performance regressions and fix performance bugs (§4.2). We then show how operators can use interfaces to pick the NF variant best suited for their target hardware and to perform root-cause diagnosis of performance anomalies (§4.3).

We evaluate PIX on 12 dataplane NFs that cover a wide variety of functionality and network protocols (Table 2). These include the Katran load balancer used in production at Facebook [71], the Natasha NAT used in production at Scaleway [58], the XDP packet filter from the Cilium project [14] and an implementation of Google's Maglev load balancing algorithm [25]. The NFs were written using DPDK [21] and eBPF XDP [82], arguably the two most popular frameworks today for building high-performance software NFs. VigNAT, Policer, Router and Bridge come from the Vigor project [83], the CRAB load balancer from [46], and the hXDP firewall from [8]. The Vigor and eBPF NFs are written in the commonly used stateless/stateful split model, which makes them amenable to exhaustive symbolic execution. We modified Natasha and DPDK NAT to also have such a clean split; this took ~3 person-days per NF.

The performance metrics we use for DPDK-based NFs are `x86` instruction count, `x86` memory access count, and `x86` CPU cycles (thus wall-clock time). Note, PIX is not specific to x86 and can just as easily predict the corresponding metrics for another ISA (e.g., ARM) if the PIX front-end is given the corresponding binary. For eBPF NFs, we only analyze the NF itself, and not the eBPF maps that are part of Linux, so we only report hardware-independent metrics.

NF	Implementation		HW-independent interface (PIX)		HW dependent interface (PIX)		Bolt contract	
	LOC	CC	LOC	CC	LOC	CC	LOC	CC
Natasha	2932	192	1.8%	8.9%	2.8%	15.1%	17.4%	97.3%
Maglev	3168	29	0.9%	37.9%	1.6%	65.5%	2.1%	82.7%
VigNAT	2770	22	0.7%	36.3%	0.9%	52%	1.8%	81%
Bridge	2837	219	0.5%	2.7%	2.1%	10.5%	22.7%	98.6%
Router	1260	17	0.4%	17.6%	1.0%	29.4%	3.0%	82.3%
Policer	2466	16	0.4%	31.2%	0.6%	37.5%	1.4%	81.2%
DPDK FW	2508	21	0.8%	38%	1.0%	45%	2.0%	85.7%
DPDK NAT	1780	35	0.6%	27%	0.9%	39%	4.5%	80%
Katran	2661	3226	2.8%	0.8%	-	-	363%	100%
Cilium filter	784	42	3.2%	14.3%	-	-	10.7%	100%
CRAB	437	4	2.0%	100%	-	-	2%	100%
hXDP FW	312	33	3.8%	15.1%	-	-	30%	100%

Table 3. Complexity of extracted interfaces and Bolt contracts vs NF implementation. "(x%)" means "x% of implementation". For each NF, the complexity is calculated for an interface with resolution equal to 10% of the maximum latency variability the NF can exhibit. Since Bolt computes the worst-case performance for HW-dependent metrics (not shown), the numbers are identical to those for HW-independent metrics.

Our testbed consists of two directly connected servers: a device under test (DUT) and a traffic generator and sink (TG). The servers are identical, with an Intel Xeon E5-2667 v2 processor @ 3.30 GHz, 32 GB of DRAM, and Intel 82599ES 10-Gbps NICs. The DUT runs one of the NFs and measures the performance, while the TG uses MoonGen [26] to generate traffic.

4.1 Does PIX Work?

In this section, we show that extracted interfaces are 1−2 orders of magnitude simpler than both the NF implementations and the equivalent Bolt contracts (§4.1.1). Their accuracy is 100% for reasonable resolutions, while at the finest resolution they are still practical and considerably better than Bolt, the state of the art (§4.1.2). Extracting a performance interface typically takes minutes (§4.1.3).

4.1.1 Are performance interfaces user-friendly?

To evaluate the "human palatability" of the performance interfaces, we (1) measure their complexity in terms of both lines of code (LOC) and cyclomatic complexity (CC) [78], and (2) evaluate whether the primitives exposed by the performance interfaces are those that NF developers and operators are familiar with.

Table 3 compares the complexity of the PIX-extracted interfaces and the Bolt contracts, measured as a fraction of LOC and CC of the implementation. In a nutshell, the extracted interfaces have 26−210× fewer LOC than the corresponding implementations and are 3−124× less cyclomatically complex, ignoring CRAB, which is already simple to start with. The performance resolution allows PIX-extracted interfaces to be 2.3−129× shorter than the Bolt contracts, and 2.1−124× less cyclomatically complex, by abstracting away irrelevant details. The more complex an NF, the higher this reduction in complexity, which argues for the real-world utility of performance interfaces.

Fig. 5 illustrates the impact of varying resolution on the complexity of Katran's performance interface. At the finest granularity, Katran's instruction-count interface, like the Bolt contract, is fairly complex (LOC=9675, CC=3226 independent paths). Since no two packets in Katran can incur an instruction count that differs by more than 854 instructions (number determined by PIX and verified by us), for resolutions above 854 the interface becomes a simple upper bound. In between these two extremes, we see how low-level details get abstracted away—for instance, at resolution=50 instructions, we see a 125× drop in complexity (LOC=75, CC=26). The Bolt contract, however, lacks the notion of resolution and thus remains 3.6× longer and just as cyclomatically complex as the implementation.

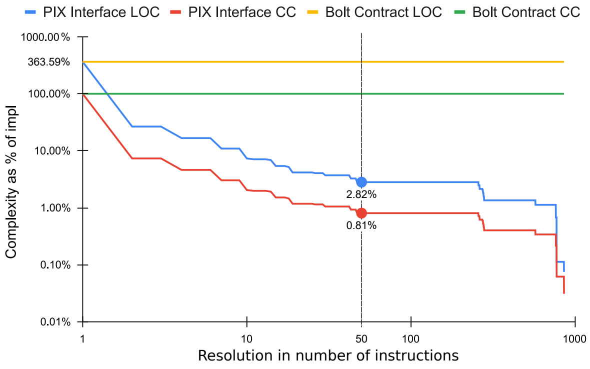

Figure 5. Impact of varying resolution on the size (LOC) and complexity (CC) of Katran's performance interface.

We conclude that PIX-extracted performance interfaces are significantly simpler than the NF implementations, which argues for them making it easier to understand performance behaviors by reading the interface than reading the code. The notion of resolution succeeds in abstracting a performance interface, giving the reader a knob with which to control the amount of detail contained in the interface.

```
def perf_vignat_gc(pkt):
  # Perf metric: x86 instructions
  # Resolution: 10
  # NF state:
  #   flowtable
  # PCVs:
  #   e - expired flows
  #   t - bucket_traversals
  #   c - hash_collisions

  x = 19*e*t + 40*e*c + 227*e + 123

  if not (pkt.is_IP) or not(pkt.is_TCP or pkt.
    is_UDP):
    return x + 7
  else:
    if pkt.port != internal_network_port:
      if flowtable.contains(pkt.flow):
        return x + 289
      else:
        return x + 68
    else:
      if flowtable.contains(pkt.flow):
        return x + 18*t + 30*c + 395
      else:
        return x + 31*t + 30*c + 547
```

Figure 6. Extracted general-case interface for VigNAT.

Another aspect of palatability is how familiar the interface looks to a human reader. To illustrate this, we show an example of the general-case interface for VigNAT in Fig. 6, restricted to TCP/UDP packets for space considerations. The interface is a succinct, self-descriptive Python program. The conditions in `if` statements are expressed in terms of fields in the input packet header (e.g., `pkt.port`) or semantic operations on data structures (e.g., `nat_flowtable.contains`), which are primitives we expect both developers and operators to understand. Being a stateful NF, VigNAT's performance is influenced by NF state, and the interface reflects this via PCVs, documented in the header. Bolt, on the other hand, does not translate low-level details and exposes primitives such as the starting condition on line 4 of Fig. 4. While such details are understandable to the NF's developer, they make the contract hard to read for those unfamiliar with the code.

Finally, we illustrate the impact of deployment-specific instantiation of interfaces on their palatability. Fig. 7 shows the interfaces for VigNAT's 50th and 95th percentile latencies and the distribution underlying them, for a particular <*workload, HW*> pair. The deployment-specific instantiation turns each formula (expressed in terms of PCVs in the general-case interface) into concrete values specific to the environment and workload, thus tailoring the interface to an operator's needs. The latency CDF also enables interested operators to understand how VigNAT's percentile latency varies.

4.1.2 Accuracy of performance interfaces

We now evaluate the prediction error of PIX-extracted interfaces, i.e., the difference between the latency predicted by the interfaces and the measured latency.

To do so, we use PIX to extract interfaces for all 8 DPDK NFs[4] for two hardware-independent metrics (`x86` instructions and memops) and one hardware-dependent one (`x86` cycles). For each NF, we instantiate two deployment-specific interfaces corresponding to two very different deployments—typical traffic representative of university networks [6] and adversarial traffic that seeks denial-of-service [64]. The above deployments represent opposite ends of the spectrum for *absolute* NF latencies [64]—e.g., adversarial traffic incurs 2.1× greater latency than typical traffic in VigNAT. To instantiate each deployment-specific interface, we use PCAP traces of 100M packets each. These traces are similar to what an operator could obtain with `tcpdump` on their domain gateway and are not specific to any particular NF implementation.

[4] PIX does not support HW-dependent metrics for eBPF NFs

```
def perf_vignat_ds(pkt):
  # Metric: CPU cycles, Resolution: 200
  # Percentile: 50
  # NF state - flowtable

  if flowtable.contains(pkt.flow):
    return 301
  else:
    if pkt.port != internal_network_port:
      return 92
    else:
      return 558
```

```
def perf_vignat_ds(pkt):
  # Metric: CPU cycles, Resolution: 200
  # Percentile: 95
  # NF state - flowtable

  if flowtable.contains(pkt.flow):
    return 395
  else:
    if pkt.port != internal_network_port:
      return 97
    else:
      return 1037
```

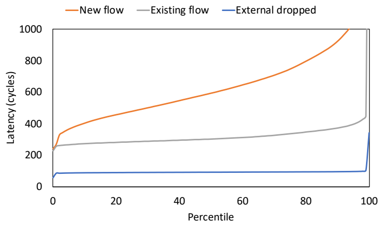

Figure 7. Common-case interfaces for VigNAT (50th and 95th percentile) and the latency CDF (resolution=200 cycles).

For ground-truth measurements, we manually generate synthetic packet traces for each <*NF, deployment*> pair akin to Scaleway's NAT test suite [57]. We playback these traces against the NF and measure the latency of each packet (the ground truth). Note, the synthetic traces are only used to measure the ground truth and not for predicting performance, thus avoiding any overfitting.

To compare to Bolt [41], the closest prior work, we use their published code [1]. We run each deployment trace through the Bolt distiller, which computes the PCVs and concretizes the performance contracts. For a comparison to Freud [69], please see Appendix A.

We present here the prediction error for the 50th percentile, 90th percentile and 99th percentile latencies (which is the point at which PIX's limitations become evident). Appendix B provides the details for the entire spectrum. We compute all prediction errors by subtracting the relevant statistic of the measured latency distribution from that of the predicted latency distribution. The results reported are at resolution 1, where PIX does the worst.

50th **percentile (median) latency:** Table 4 describes the maximum and average error for median latencies across all NFs for each metric and deployment regime. Note, despite the *absolute* NF latency differing widely, PIX's prediction accuracy is similar for both deployments, showing that the PIX front-end correctly instantiates each deployment-specific interface.

	Instr's error		**Memops** error		**Cycles** error	
	Typ	*Adv*	*Typ*	*Adv*	*Typ*	*Adv*
PIX	1.8% (1.5%)	1.7% (1.2%)	4% (1.6%)	3.7% (1.5%)	26% (11%)	24% (9%)
Bolt	7.5% (3.7%)	7.6% (4.0%)	7.5% (3.7%)	7.6% (4.0%)	308% (164%)	186% (103%)
PIX improvement	4.1× (2.4×)	4.4× (3.3×)	1.8× (2.3×)	2.0× (2.6×)	11.8× (14.9×)	7.7× (11.4×)

Table 4. Max (average) median latency prediction error for PIX and Bolt for typical (Typ) and adversarial (Adv) traffic.

We find that, even in the worst case for PIX (i.e., finest resolution), the error for hardware-independent metrics is ≤4%, which is small enough to make PIX practical. At any reasonable resolution, the error vanishes and PIX becomes 100% accurate. PIX outperforms Bolt by 4.4×. For the CPU cycles, PIX has a maximum error of 26%. This is due to the overhead of the instrumentation used to measure the CPI and LLC miss latencies. Nevertheless, PIX's accuracy is an order of magnitude better than Bolt's since PIX reasons about hardware performance as a distribution, while Bolt only models the worst case.

90th **percentile latency:** Table 5 describes the prediction error for 90th percentile latencies. The results are similar to those for median latency with PIX outperforming Bolt by up to an order of magnitude. This is once again due to PIX reasoning about each of the PCVs as a distribution, while Bolt only models the worst-case.

	Instr's error		**Memops** error		**Cycles** error	
	Typ	*Adv*	*Typ*	*Adv*	*Typ*	*Adv*
PIX	1.4% (0.9%)	1.2% (0.9%)	3.2% (1.4%)	2.8% (1.2%)	22% (10%)	19% (7%)
Bolt	5.9% (2.4%)	5.3% (3.1%)	6.6% (2.9%)	6.1% (3.2%)	234% (122%)	153% (94%)
PIX improvement	4.2× (2.6×)	4.4× (3.4×)	2.0× (2.1×)	2.1× (2.6×)	10.6× (12.2×)	8× (13.4×)

Table 5. Max (average) prediction error for 90th percentile latencies for typical (Typ) and adversarial (Adv) traffic.

99th **percentile latency:** PIX cannot accurately predict the latency at the very end of the tail (nor can Bolt). PIX's predictions have an error of ≤61% (average 22%), while Bolt's predictions have an error of ≤45% (average 14%).

It is interesting to note that PIX *underestimates* the 99th percentile latency while Bolt *overestimates* it; this contrasting behavior is due to the different hardware models underlying the two tools. PIX underestimates the 99th percentile latency since its simple hardware model (*instructions * CPI + LLC_misses * miss_latency*) is invalid at this percentile where other hardware aspects also impact latency significantly. Bolt, on the other hand, overestimates the 99th percentile latency since its hardware model is designed to estimate the absolute worst-case latency. However, PIX's simple hardware model enables it to accurately predict performance at all percentiles except the tail (details in Appendix B); a task that Bolt's worst-case-only model is incapable of.

4.1.3 Time to extract performance interfaces

Table 6 shows the time it takes PIX to extracts the general-case interfaces for all the NFs in this evaluation. We believe that these numbers make it feasible to incorporate performance interfaces extraction part of the regular NF development cycle, e.g., as part of continuous integration.

NF	Natasha	Maglev	VigNAT	Bridge	Router	Policer	DPDK FW	DPDK NAT	Katran	Cilium filter	CRAB	hXDP FW
PIX	15	5	4	17	0.73	3	4	6	32	0.43	0.15	0.23
BOLT	6	4	2	7	0.35	1.7	2	3	28	0.26	0.1	0.13

Table 6. Time, in minutes, for PIX and Bolt to extract the general-case interfaces and contracts, respectively.

The time required to obtain the deployment-specific interface is largely a function of the time required to run the provided workload. In our experiments, we ran PCAP files with 100M packets, and it took PIX <= 5 mins to generate the deployment-specific interface for a given <*workload, HW*> pair from the general-purpose interface, regardless of NF. We conclude that PIX fulfills the portability requirement (§3) well: operators can download an NF with its general-case interface, provide a PCAP file specific to their deployment, and PIX quickly produces the deployment-specific interface.

4.2 Are interfaces useful to NF developers?

In this section, we present two workflows that NF developers can use to understand (§4.2.1) and debug (§4.2.2) the performance behavior of their code.

4.2.1 Flagging performance regressions

Programmers often introduce involuntarily performance regressions. Using performance test suites to catch such regressions is not easy, because they require environment setup, are fragile, and take long to run. We show here how a developer or a tool can instead compare the performance interface before and after a commit to identify performance regressions more quickly, conveniently, and precisely than with a performance test suite.

We wrote a script that retrieves each Katran commit and uses PIX to extract the corresponding instruction-count interface, at resolution=1. For each pair of commits a and b, there is a corresponding pair of interfaces S_a and S_b. The script finds the maximum latency (in terms of LLVM instruction count) predicted by each of the two interfaces and compares the two. We report LLVM (not eBPF bytecode) instructions since PIX builds on KLEE which interprets LLVM IR. Reporting eBPF instructions would require us to build on a tool that interprets eBPF bytecode (e.g., Serval [59])—this is an engineering task we leave to future work. We run PIX on all commits to the eBPF portion of Katran's code.

Table 7 shows the commits where a performance regression occurs. Over the past three years, the maximum latency for new flows regressed by 14.6%.

Commit ID	Perf before [LLVM instr's]	Perf after [LLVM instr's]	Performance regression [%]
Orig commit	-	1771	-
873d0501695c	1765	1896	7.42%
39e58b530a8a	1896	1914	0.95%
458aa0907b68	1914	1933	0.99%
15f81d0e7ec6	1930	1946	0.83%
74c3338c2f7e	1952	1983	1.59%
d0790d3a3823	1983	2030	2.37%
All commits	1771	2030	14.62%

Table 7. Perf regressions in Katran (handling new flows).

We imagine using this workflow as part of continuous integration (CI) to automatically identify unintended performance regressions. The CI system can present to the developer a before-and-after comparison of performance that directly highlights for which classes of inputs the regression occurs and what the magnitude of the regression is. Compared to performance tests, this workflow consumes less developer time and fewer resources and offers better completeness.

4.2.2 Fixing performance bugs

By helping developers understand the code's performance more quickly and deeply, interfaces can help fix performance bugs. We illustrate this with two examples of performance bugs in the map used by Vigor NFs [50].

The top of Fig. 8 shows a snippet of the performance interface of the `contains` operation in libVig's `map`.

```
if map.contains(key): # --- BEFORE ---
  if not(cached(key)):
   # Warning: 2*t integer divides
   return (4*t)*miss_latency + (21*t+27)*CPI
....
if map.contains(key): # --- AFTER ---
  if not(cached(key)):
   return (1*t)*miss_latency + (18*t+27)*CPI
....
```

Figure 8. Interface for `map_contains()` before and after the bug fix. `t` is the PCV for traversals in the hash ring.

The first red flag is the warning issued by PIX itself, based on tracking of expensive x86 instructions that adversely impact CPI. Looking for integer divides in the `map` code, we found that, on each traversal, it uses two costly modulo operations. To fix the issue, we replaced them with one bitwise `and`.

The second red flag is that each traversal requires 4 independent heap accesses (`4*t`). It turns out that `key` metadata is being stored in four distinct arrays of `int` elements. Our fix was to encapsulate `key`'s metadata in a single `struct` and use a single array with elements of this `struct` type. The rest remained unchanged.

Table 8 shows the impact of our fixes, based on Vigor's benchmarks: the two fixes, together, improve NF latency by 22% on average, and throughput by 19%.

NF	Throughput [Mpps]			Change	Latency [ns]			Change
	Orig	Fix 1	Fix 2		Orig	Fix 1	Fix 2	
VigNAT	3.88	4.36	4.68	20.62%	317	276	236	25.55%
Bridge	3.05	3.59	3.62	18.69%	410	332	323	21.22%
Maglev	2.58	2.86	3.04	17.83%	482	423	391	18.88%

Table 8. Throughput and latency of three NFs using `map`, shown before/after each performance bug fix.

4.3 Are interfaces useful to NF operators?

Operators typically care about how an NF performs in their specific deployment, not in general for everyone's deployment. We show how operators can use performance interfaces to pick the NF variant best suited to their hardware (§4.3.1) and to do a root-cause diagnosis of deployment-specific performance anomalies (§4.3.2).

4.3.1 Which NF variant for my NIC?

Modern NICs provide the ability to offload specific tasks (like checksums and encryption) to specialized hardware. It is therefore useful to know which variant of an NF takes max advantage of the offloads available on a NIC.

Fig. 9 shows the interfaces for two variants of a NAT, and the interaction with checksum offload on Mellanox ConnectX-4 [15] and Intel Ixgbe [40] NICs. The formally verified VigNAT does not do any offloading, whereas DPDK NAT does. The strings in the `if` conditions on lines 3 and 6 are identical to the one used by the NIC driver to identify itself [22]. The interface also shows the difference in latency: Ixgbe requires the software to compute a pseudo-header checksum, whereas ConnectX-4 allows full offload, so it has lower latency.

Based on this performance interface, an operator can make an informed deployment decision: if using Ixgbe NICs, choosing the verified VigNAT makes sense; else, it's a trade-off to make carefully.

4.3.2 Why do I get bad performance?

NFs running in production can face workloads that trigger surprising performance degradation. To address such anomalies, operators must first diagnose the root cause, and this often takes a lot of work.

PIX helps the search for a root cause by providing a list of possible explanations for the observed performance, ranked by likelihood. Given a problematic workload and an NF (or its general-case interface), PIX instantiates the PCVs in a deployment-specific manner and then measures the distributions for each PCV and the NF latency. It then ranks the PCVs based on the correlation between the latency distribution and that of the PCV (using least-square fit linear regression).

```
# Snippet from VigNAT interface
if flowtable.contains(pkt.flow):
 return  18*t + 30*c + 518  # No offload
else:
  ....
# Snippet from DPDK NAT interface
if flowtable.contains(pkt.flow):
 if(NIC_family == "net_mlx5"):
  return  18*t + 30*c + 265 + cksum_offload()
 else:
  if(NIC_family == "net_ixgbe"):
   return 18*t + 30*c + 478 + cksum_offload()
  else:
   return 18*t + 30*c + 564
else:
  ....
```

Figure 9. Interfaces for VigNAT and DPDK NAT: VigNAT does checksums in software, while DPDK NAT offloads checksums to the NIC as much as possible.

To illustrate this workflow, we refer to three performance bugs that span both hardware and software root causes, shown in Table 9. The first bug occurs due to the uniform random workload causing hash collisions in a widely used hash function [42] used by Bridge; typical workloads with Zipfian distributions do not suffer from hash collisions. The second bug is caused by VigNAT's batches expiry of flows, which results in a latency spike that only becomes evident for traffic with high churn. The third bug occurs when the active flowtable in Maglev overflows the last-level cache of the server; this makes the latency spike be highly dependent on LLC configuration.

Bug	Root cause	Identified as most-likely cause?
Spike in median latency of Bridge for uniform random workload	`hash-collisions`	Yes
Spike in tail latency of VigNAT due to high churn	`expired-flows (batched)`	Yes
Spike in median latency of Maglev on a particular x86 server	`active-flowtable-size`	Yes

Table 9. Performance bugs used for root-cause diagnosis.

For each bug, we generated a workload that triggers it and provided the PCAP file to PIX, along with the general-case interface of the corresponding NF. For each bug, PIX correctly reported the culprit PCV as the most likely root cause. Of course, PIX can only track bugs that arise from PCVs it accounts for. It would be unable, for instance, to identify the root cause for a latency spike due to LLC evictions caused by a noisy neighboring process, since PIX does not account for contention.

This example illustrates how PIX can help focus the operators' attention on likely explanations for the performance they observe, thereby reducing the amount of work needed to find the root cause.

In conclusion, our evaluation shows that PIX is practical: the complexity of extracted interfaces is significantly lower than the NF implementation, their accuracy is high, and the time taken to extract them is reasonable. Further, NF developers and operators can use these interfaces to identify performance regressions, diagnose and fix performance bugs, and pick the NFs that are best suited to their hardware.

5 Does PIX Generalize?

In this section, we explore how PIX can generalize in two directions: (1) programs other than NFs, that are nevertheless still amenable to ESE; and (2) NFs that are not amenable to ESE. Overall, we find that the design of PIX—split into a modular back-end and front-end that produce general-case and deployment-specific interfaces, respectively—enables generalization by adapting just the necessary modules in the PIX pipeline.

Beyond NFs: We have successfully applied PIX to the OpenSSL library, to uncover digital side-channels, and to eBPF extensions for user-space file systems.

Extracting interfaces for finding digital side-channels required modifying only PIX's hardware model (i.e., step 1 in the back-end). Implementing a new model focused on sources of constant-time violations (using the exhaustive list in [3]) took us 2 person months. We ran PIX on 12 cryptographic primitives from OpenSSL 3.0 [61] and found a constant-time violation in the AES cipher unpadding function. This violation was acknowledged by the OpenSSL maintainers [62]. We have submitted a pull request [63] that has undergone multiple rounds of review and is in the final stages of getting merged.

Our experience with OpenSSL reinforced our belief (from §4.2.1) that a tool that automatically extracts performance interfaces would be of great use to developers. For example, we learned that the violation we uncovered had been latent since OpenSSL 1.1.1 because the developer "just reused the code" and had somehow been missed despite the extremely thorough code reviews that OpenSSL goes through. If performance interfaces of the OpenSSL code were extracted regularly, e.g., as part of continuous integration, it is unlikely that this violation would have persisted for this long.

Extracting interfaces for eBPF file system extensions was more straightforward since the code is similar to that of eBPF NFs. Here, we only had to add translation rules (step 2 in the PIX back-end) corresponding to the supported system calls. This took 4 person-days, after which PIX was able to automatically extract interfaces for the extFUSE extensions [7].

Code not amenable to ESE: To evaluate the limits of PIX's ESE-based approach, we used PIX on Snort [73], a popular IDS that independent prior work has shown to not be amenable to ESE [53, 81]. Our results corroborated those from prior work; while PIX did extract performance interfaces for the networking stack and all detection rules that look only at packet headers, attempting to extract a complete interface caused PIX to time out. Extracting an interface from Snort with PIX requires either that we modify its code to cleanly separate the stateful components, or that we replace Bolt in the PIX back-end with a manual theorem prover.

6 Related Work

We compared PIX to Bolt [41] w.r.t the design (in §2) and results (in §4.1). We do not do so again here.

Here we provide a qualitative comparison of PIX against Freud, Appendix A provides a detailed quantitative one. Freud treats code almost as a black box and relies on developers to provide comprehensive performance test suites in order to guide the exploration of performance behaviors and ensure prediction accuracy. PIX is white-box because it analyzes source code. Analyzing the source ensures that PIX can analyze the system once, and instantiate the interface for different deployments, while Freud users must re-run the tool for each new <workload, hardware> pair. Lastly, Freud's performance formulae are limited to being expressed in terms of program variables, but the performance of stateful code typically depends on (implicit) PCVs instead [37, 38].

Performance upper bounds and adversarial workloads: Worst-Case Execution Time (WCET) Analysis derives formal upper bounds on performance; [79] provides an overview of the state of the art. These bounds are particularly popular in the domain of real-time and safety-critical systems where performance guarantees are a part of functional correctness. While WCET only looks at one aspect of the performance profile—the absolute worst-case—performance interfaces characterize performance in the face of any arbitrary input, whether typical, ideal, or adversarial. Further, to enable stringent upper bounds, real-time systems tend to avoid dynamic data structures and input-dependent memory accesses—aspects that are commonplace in NFs.

Considerable prior work focuses on generating and analyzing adversarial workloads that attack software performance [2, 18, 49, 60, 64, 65, 67, 72, 76]. As with WCET, all of this work focuses only on worst-case inputs, while interfaces reflect the entire performance profile.

Performance profilers: Traditional profilers [51] measure the execution cost (e.g., running time, executed instructions, cache misses) of a piece of code. Trend Profiling [34], Algorithmic Profiling [85] and Input-Sensitive Profiling [16, 17] take this one step further: by extracting a cost function defining the relationship between input size and execution cost. However, like Freud, these tools treat the code as a black-box and require developers to provide comprehensive performance test suites to guide the exploration of performance behavior.

Performance analysis for SmartNIC-based NFs: Krude et al. [47] use SMT solvers to analyze NF code written for processor-based SmartNICs and provide lower bounds on throughput. Focussing solely on throughput lower bounds results in their approach being limited to analyzing worst-case latency, much like WCET. Clara [68] uses machine learning to analyze NF code written in C to identify "effective porting strategies" that result in low latency when the NF is ported to a SmartNIC. Unlike PIX that focusses on accurately predicting the NF latency, Clara focusses on identifying how the NF implementation can make best use of the SmartNIC hardware (e.g., accelerator usage, NF state placement strategies, etc).

Program analysis for NF code running on commodity hardware: Several instances of prior work have proposed using program analysis to help understand, debug, and verify the semantic behavior of software NFs [10, 11, 19, 44, 66, 74, 84, 86]. PIX builds upon the experience of all of this prior work, but analyzes NF performance.

NF performance monitoring and diagnosis: Several instances of prior work [28, 35, 56, 80] diagnose performance issues such as packet drops or low throughput in NF deployments. Such work is complementary to PIX since it helps diagnose performance issues once they occur in production, while PIX provides a summary of NF performance before the NF is deployed.

7 Conclusion

We proposed the notion of a *performance interface*—a program that accepts the same inputs as the system and outputs the latency incurred by the given input. For the interface to be simultaneously simple, accurate and human-readable we proposed (a) the notion of a *performance resolution* to eliminating unnecessary details, and (b) separate *deployment-specific interfaces* to tailor the interface to particular <workload, environment> pairs.

We described a tool (PIX) that automatically extracts performance interfaces from NF implementations. and evaluated it on 12 NFs, including several used in production. Our results show that PIX is practical—the complexity of extracted interfaces is significantly lower than the NF implementation, their accuracy is high, and the time to extract them is reasonable. Finally, we show how NF developers and operators can use these interfaces today, to identify performance regressions, diagnose and fix performance bugs, and pick the NFs that are best suited to their hardware.

8 Acknowledgements

We thank our shepherd Theo Benson and the anonymous OSDI, SOSP and NSDI reviewers for their detailed feedback that significantly improved the paper. We are also grateful to the many people who provided helpful feedback on drafts of the paper at various stages—Solal Pirelli, Arseniy Zaostrovnykh, Marios Kogias, Adrien Ghosn, Can Cebeci, Yugesh Kothari, Ayoub Chouk, Johannes Kinder, Jonas Wagner, Ed Bugnion and James Larus.

References

[1] Bolt source code. https://github.com/bolt-perf-contracts/bolt.

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

[3] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. Verifying constant-time implementations. In *USENIX Security Symp.* (2016).

[4] WSJ: Facebook, google and apple hit by unusual outages. https://www.wsj.com/articles/facebook-and-instagram-suffer-lengthy-outages-11552539752.

[5] Barbette, T., Soldani, C., and Mathy, L. Fast userspace packet processing. In *ACM/IEEE Symp. on Architectures for Networking and Communications Systems* (2015).

[6] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conf.* (2010).

[7] Bijlani, A., and Ramachandran, U. Extension framework for file systems in user space. In *USENIX Annual Technical Conf.* (2019).

[8] Brunella, M. S., Belocchi, G., Bonola, M., Pontarelli, S., Siracusano, G., Bianchi, G., Cammarano, A., Palumbo, A., Petrucci, L., and Bifulco, R. hxdp: Efficient software packet processing on FPGA NICs. In *Symp. on Operating Sys. Design and Implem.* (2020).

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

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

[11] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE way to test openflow applications. In *Symp. on Networked Systems Design and Implem.* (2012).

[12] Chen, S., Delimitrou, C., and Martinez, J. F. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2019).

[13] Commits to the eBPF code in the Cilium project. https://github.com/cilium/cilium/commits/master/bpf.

[14] Cilium Project. https://cilium.io.

[15] Mellanox ConnectX-4 Network Adapter Cards. https://downloadcenter.intel.com/download/14687.

[16] Coppa, E., Demetrescu, C., and Finocchi, I. Input-sensitive profiling. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

[17] Coppa, E., Demetrescu, C., Finocchi, I., and Marotta, R. Estimating the empirical cost function of routines with dynamic workloads. In *Intl. Symp. on Code Generation and Optimization* (2014).

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

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

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

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

[22] Ethtool Driver Identifier. https://docs.huihoo.com/doxygen/linux/kernel/3.7/include$_2$uapi$_2$linux$_2$ethtool$_8$h$_s$ource.html#l00085.

[23] Commits to eBPF maps in the Linux Kernel. https://github.com/torvalds/linux/commits/master/kernel/bpf.

[24] eBPF maps. https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf$_m$aps.html.

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

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

[27] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. A performance counter architecture for computing accurate CPI components. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2006).

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

[29] Filliâtre, J., Gondelman, L., and Paskevich, A. The spirit of ghost code.

[30] Freud source code repository. https://github.com/usi-systems/freud.

[31] Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. Caladan: Mitigating interference at microsecond timescales. In *Symp. on Operating Sys. Design and Implem.* (2020).

[32] Ghost variables in software verification. http://whiley.org/2014/06/20/understanding-ghost-variables-in-software-verification/.

[33] GitHub. The 2020 state of the Octoverse. https://octoverse.github.com, Dec. 2020.

[34] Goldsmith, S., Aiken, A., and Wilkerson, D. S. Measuring empirical computational complexity. In *Symp. on the Foundations of Software Eng.* (2007).

[35] Gong, J., Li, Y., Anwer, B., Shaikh, A., and Yu, M. Microscope: Queue-based performance diagnosis for network functions. In *ACM SIGCOMM Conf.* (2020).

[36] Google cloud storage incident. https://status.cloud.google.com/incident/storage/19002.

[37] Gulwani, S. SPEED: symbolic complexity bound analysis. In *Intl. Conf. on Computer Aided Verification* (2009).

[38] Gulwani, S., Mehra, K. K., and Chilimbi, T. M. SPEED: precise and efficient static estimation of program computational complexity. In *Symp. on Principles of Programming Languages* (2009).

[39] Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Symp. on Cloud Computing* (2014).

[40] Intel Network AdIntel 82599 10 GbE Controller Datasheetapter Driver for PCIe Intel 10 Gigabit Ethernet Network Connections. https://downloadcenter.intel.com/download/14687.

[41] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance contracts for software network functions. In *Symp. on Networked Systems Design and Implem.* (2019).

[42] Java String hashCode. https://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode().

[43] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

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

[45] King, J. C. Symbolic Execution and Program Testing. *J. ACM 19*, 7 (1976).

[46] Kogias, M., Iyer, R., and Bugnion, E. Bypassing the load balancer without regrets. In *Symp. on Cloud Computing* (2020).

[47] Krude, J., Rüth, J., Schemmel, D., Rath, F., Folbort, I., and Wehrle, K. Determination of throughput guarantees for processor-based smartnics. In *Intl. Conf. on Emerging Networking Experiments and Technologies* (2021).

[48] Lampson, B. Hints and principles for computer system design. https://www.microsoft.com/en-us/research/uploads/prod/2019/09/Hints-and-Principles-v1-full.pdf, November 2020.

[49] Lemieux, C., Padhye, R., Sen, K., and Song, D. Perffuzz: automatically generating pathological inputs. In *Intl. Symp. on Software Testing and Analysis* (2018).

[50] libVig source code . https://github.com/vigor-nf/vigor/tree/master/libvig/verified.

[51] The Linux Perf Tool. https://en.wikipedia.org/wiki/Perf$_($Linux).

[52] Manousis, A., Sharma, R. A., Sekar, V., and Sherry, J. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM Conf.* (2020).

[53] Mehrotra, P., and Goswami, S. Analyzing Snort. Tech. rep., University of British Columbia, 2018.

[54] Microscope survey form and results. https://www.dropbox.com/s/66cp4k3wl8zm0q5/survey.pdf?dl=0.

[55] Moon, S., Helt, J., Yuan, Y., Bieri, Y., Banerjee, S., Sekar, V., Wu, W., Yannakakis, M., and Zhang, Y. Alembic: Automated model inference for stateful network functions. In *Symp. on Networked Systems Design and Implem.* (2019).

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

[57] Performance Tests for Natasha. https://github.com/scaleway/natasha/tree/master/test/perf.

[58] Scaleway Natasha. https://github.com/scaleway/natasha.

[59] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Symp. on Operating Systems Principles* (2019).

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

[61] OpenSSL. https://github.com/openssl/openssl.

[62] Github issue raising constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/issues/16230.

[63] Pull request to fix constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/pull/16323.

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

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

[66] Pirelli, S., Valentukonytė, A., Argyraki, K., and Candea, G. Automated verification of network function binaries. In *Symp. on Networked Systems Design and Implem.* (2022).

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

[68] Qiu, Y., Xing, J., Hsu, K.-F., Kang, Q., Liu, M., Narayana, S., and Chen, A. Automated smartnic offloading insights for network functions. In *Symp. on Operating Systems Principles* (2021).

[69] Rogora, D., Carzaniga, A., Diwan, A., Hauswirth, M., and Soulé, R. Analyzing system performance with probabilistic performance annotations. In *ACM EuroSys European Conf. on Computer Systems* (2020).

[70] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symp. on Security and Privacy* (2010).

[71] Shirokov, N., and Dasineni, R. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer, May 2018.

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

[73] Snort. https://www.snort.org.

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

[75] Terpstra, D., Jagode, H., You, H., and Dongarra, J. J. Collecting performance data with PAPI-C. In *Workshop on Parallel Tools for High Performance Computing* (2009).

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

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

[78] Watson, A. H., and McCabe, T. J. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Computer Systems Laboratory, National Institute of Standards and Technology, 1996.

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

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

[81] Wu, W., Zhang, Y., and Banerjee, S. Automatic synthesis of nf models by program analysis. In *ACM Workshop on Hot Topics in Networks* (2016).

[82] Express data path. https://en.wikipedia.org/wiki/Express$_D$ata$_P$ath.

[83] Zaostrovnykh, A., Pirelli, S., Iyer, R. R., Rizzo, M., Pedrosa, L., Argyraki, K. J., and Candea, G. Verifying software network functions with no verification expertise. In *Symp. on Operating Systems Principles* (2019).

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

[85] Zaparanuks, D., and Hauswirth, M. Algorithmic profiling. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

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

Appendix A Using Freud on NFs

In this section, we describe our experience experimenting with Freud. We used the publicly available Freud code [30] at commit ID e6e7a91006.

Freud takes as input a binary and a test suite, and outputs an expression of performance (runtime) as a function of input and global variables. So, by design, it strikes a different generality/accuracy balance than PIX: It is more general, in the sense that it can run on any program—not just NFs that are amenable to ESE—and requires no source code and no data-structure pre-analysis. It is less accurate, in two ways: (a) It cannot reason about the performance of execution paths that are not triggered by the test suite (since it does not symbex the program, and it does not analyze the source code). (b) It cannot reason about how past inputs affect performance in stateful code (since it does not know anything about the data structures where the state is stored).

To assess Freud's generality/accuracy balance, especially in the context of NFs, we used it on three classes of programs: (a) A stateless program that spins for a period of time proportional to the input length. (b) Data structures commonly used by NFs: a longest prefix match (LPM) trie and a hash map. (c) NFs: VigNAT (academic prototype), Natasha (production NAT used at ScaleWay), and Maglev (DPDK implementation of Google's load balancer). Natasha comes with an open-source performance test suite [57], making it an ideal fit for Freud. For the remaining programs, we used as test suites the packet traces on which we evaluated PIX.

Table 10 summarizes our results, discussed below.

Freud-vanilla: First, we ran Freud on unmodified programs, and it behaved as expected: It successfully characterized the spinning program's runtime as a function of the input length, but it could not produce meaningful performance annotations for the data structures or NFs.

Freud mode	Program	Accurate annotation?
Freud-vanilla	Synthetic stateless NF	Yes
	LPM trie	No
	Hashmap	No
	Real NFs	No
Freud-nf	Synthetic stateless NF	Yes
	LPM trie	Yes
	Hashmap	No
	Real NFs	No

Table 10. Summary of our experiments with Freud.

This is normal, since, in the latter programs, runtime is a function of implicit variables that capture the interaction between current and past inputs (e.g., number of iterations of `while(bucket[i].is_full ==1)`).

Freud-nf: Next, to compare with PIX more fairly, we explicitly modified our programs to work with Freud: we identified conditions that we knew impacted performance (essentially PCVs) and manually added them as global variables (which Freud tracks). For instance, in the hashmap, we added a global variable to explicitly track the number of collisions; in the LPM trie, we added a global variable to explicitly track the depth traversed.

The results for the data structures were mixed: For the LPM trie, Freud produced an accurate performance annotation. For the hashmap, Freud mistook a correlation for a causation: when a test caused every packet to experience a collision, Freud concluded that runtime was determined by occupancy, as opposed to the number of collisions. We expect that this issue can be resolved at the cost of extra developer effort (to produce a smarter test suite).

For the real NFs, Freud could not produce meaningful performance annotations (despite our modifications to the NF source code). This is not surprising, given that Freud does not analyze the source code, hence is unable to track how a sequence of state-accessing calls affects runtime. For instance, in Maglev, known client packets that are destined to a now-stale backend-server undergo consistent hashing once again, to pick a new backend. Since Freud does not analyze the source code, it cannot track how this call sequence affects runtime, looking instead to express runtime as a function of individual variables—which does not work. We observed similar scenarios in the other NFs.

Conclusion: In its current form, Freud cannot produce accurate performance annotations for stateful NFs. To do so, it would need to track how a sequence of state-accessing calls affects performance. We think that that would necessarily require (a) some assumption about the structure of the code (akin to our clean state assumption), (b) a nuanced test suite for the NF's data structures to reveal which aspects of state affect performance (which is done, in our approach, with the manual extraction of PCVs during pre-analysis), and (c) leveraging call context. We think that adding these elements to Freud would bring it very close to PIX; we expect it would achieve similar accuracy, but at the cost of its current generality.

Appendix B Accuracy of performance interfaces

This section provides more detailed answers to the following questions: (1) What is the prediction error for both PIX and Bolt for each individual NF? (2) What is the prediction error for both PIX and Bolt as a function of the percentile latency?

Prediction error for individual NFs: Table 11 provides detailed per-NF results for PIX's prediction accuracy for hardware-independent metrics, i.e., x86 instruction count and x86 memory accesses. We see that the results are similar across all the NFs, with PIX consistently outperforming Bolt.

NF	Spade Prediction Error		Bolt Prediction Error	
	x86 instructions	x86 mem-ops	x86 instructions	x86 mem-ops
VigNAT	1.2%	1.3%	3.1%	4.0%
Bridge	0.8%	1.1%	3.6%	3.6%
Maglev	1.1%	1.1%	5.1%	4.2%
Router	1.7%	3.9%	6.8%	6.1%
Policer	1.4%	1.7%	4.2%	5.1%
Natasha	2.6%	3.2%	5.1%	5.6%
DPDK NAT	0.9%	1.1%	2.3%	2.9%
DPDK FW	1.1%	1.4%	2.7%	3.7%

Table 11. Prediction error for median latency for x86 instruction count and memory accesses for all 8 DPDK NFs in the deployment characterized by the typical traffic. The numbers for adversarial traffic are similar.

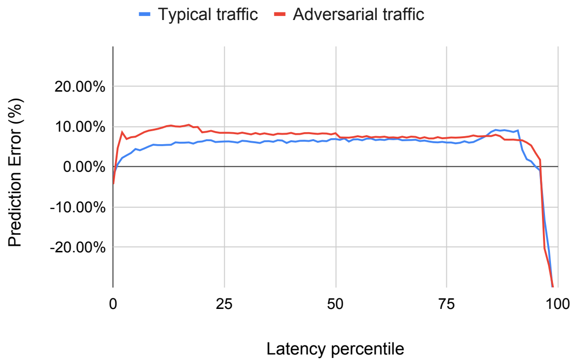

Figure 10. PIX's average prediction error for CPU cycles across two deployments as a function of the percentile latency

Prediction error as a function of the percentile latency: Fig. 10 illustrates PIX's average prediction error across all 8 NFs for each deployment. First, we see that the average error shows similar trends across deployments, proving that PIX characterises the deployment-specific workload correctly. Second, we see that for both deployments the average error is more or less stable at

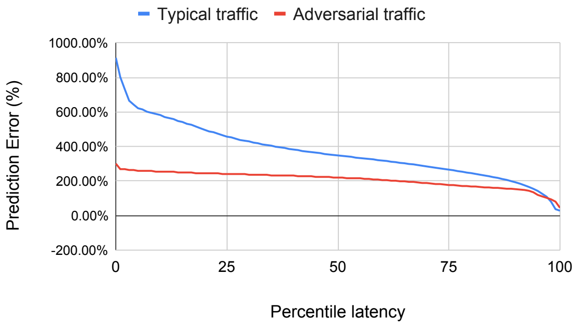

Figure 11. Bolt's average prediction error for CPU cycles across two deployments as a function of the percentile latency

around 8% up to the 95th percentile showing that for these percentiles, PIX characterises the interaction of the NF with the hardware correctly. Lastly, we see that at the tail, the prediction errors become negative. This is due to the fact that the simple HW model that PIX employs (*instructions* * *CPI* + *LLC_misses* * *miss_latency*) is invalid at the tail, where other hardware aspects kick in.

Fig. 11 illustrates Bolt's average prediction error across all 8 NFs for each deployment. Bolt estimates only worst-case latency and this is evident in the results—note the change in scale on the y-axis from Fig. 10. For all percentiles except the tail, Bolt is widly inaccurate with errors up to 900%. On the other hand, Bolt does not underestimate latency at the tail since it accounts for myriad worst-case scenarios in the underlying hardware worst-cases that PIX ignores to ensure accuracy across the remainder of the spectrum.