Checking-in on Network Functions

Zeeshan Lakhani
Carnegie Mellon University
Pittsburgh, Pennsylvania
zlakhani@cs.cmu.edu

Heather Miller
Carnegie Mellon University
Pittsburgh, Pennsylvania
heather.miller@cs.cmu.edu

Abstract

When programming network functions, changes within a packet tend to have consequences—side effects which must be accounted for by network programmers or administrators via arbitrary logic and an innate understanding of dependencies. Examples of this include updating checksums when a packet's contents has been modified or adjusting a payload length field of a IPv6 header if another header is added or updated within a packet. While static-typing captures interface specifications and how packet contents should behave, it does not enforce precise invariants around runtime dependencies like the examples above. Instead, during the design phase of network functions, programmers should be given an easier way to specify checks up front, all without having to account for and keep track of these *consequences* at each and every step during the development cycle. In keeping with this view, we present a unique approach for adding and generating both static checks and dynamic contracts for specifying and checking packet processing operations. We develop our technique within an existing framework called NetBricks and demonstrate how our approach simplifies and checks common dependent packet and header processing logic that other systems take for granted, all without adding much overhead during development.

CCS Concepts

• **Networks → Programming interfaces**.

Keywords

network functions, design-by-contract, software verification

1 Introduction

Writing network functions (NFs) today is as capable as ever, with numerous frameworks and domain-specific languages to choose from. Some target development ease, reusable abstractions, or a familiar programming model that is in vogue within software development at large, while others stress performance guarantees or deployment at scale. Irrespective of which framework or model is used, NFs tend to be comprised of code exercising arbitrary logic and domain knowledge that only network programmers or administrators would know, or *should* know.

Consider the payload length field of an IPv6 header, a field whose value is dependent upon the consequence of processing and manipulating the rest of the packet it's a part of. If an extension header [6] is added to or removed from the packet, or a layer 4 protocol's payload is modified in any way, then this payload length field must be incremented or decremented accordingly. Other "middleboxes" downstream in the network will apply functionality based on the value held in this field—without calculating the length of the rest of the packet—or just drop the packet outright if it's wrong. Handling this effect is often taken for granted, a piece of arbitrary logic that network programmers have to remember to apply and validate at different steps in a function pipeline or at egress. For example, in the still widely used [9, 18, 24] packet processing system Click [21], a *CheckIP6Header* module provides validation on the payload length field via:

```
if(ntohs(ip->ip6_plen) > (plen - 40))
  goto bad;
```

While this code does do a "check" on the field, it hard codes the value as part the conditional check instead of using a constant or variable to better express meaning (*40* is the fixed size of an IPv6 header). Additionally, if the check is invalid, *goto bad* executes a jump, leaving very little in the way of failure handling and unambiguous messaging. Besides a few per-field validations within the element file, the module does not account for related changes downstream when composed with other modules or the possibility of extension headers or variable-length fields. The functionality expressed in this snippet need not be so unwieldy, as validations should be a first-class part of programmable network architecture.

In this paper, we present a novel approach that clearly describes and validates these arbitrary effects via the addition and generation of both static checks and dynamic, runtime contracts for specifying conditional dependencies in common packet processing actions. Our work makes use of three well-known programming paradigms.

ANRW '19, July 22, 2019, Montreal, QC, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6848-3/19/07…$15.00
https://doi.org/10.1145/3340301.3341131

Static Assertions and Types Our prototype is incorporated within a framework built using the Rust programming language, which emphasizes a strong, static type system with first-class polymorphism [32] (parametric and ad-hoc). Headers within a packet are explicitly-typed, e.g. *Ipv4Hdr* or *Ipv6Hdr* for instance, and contain associated types [8] that define which header(s) can precede it, e.g. an IPv6 Header relies on an Ethernet header existing before it within a packet. We leverage the type system along with the concept of static assertions [19] to provide compile-time checking for a subset of network function components, including constant checks at the call site of these functions and ensuring that packet order and definitions adhere to specification.

Design by Contract We take inspiration from D's contract system [12], whose design was inspired by contract schemes [25] where contracts are provided and run during testing, debugging, and development stages—the design phase—but are usually omitted for release builds in order to maximize performance. In using this style of contracts, programmers are able to code, test, and simulate NF's around *pre* (ingress) and *post* (egress) invariants, checking which conditions must hold as packets are transformed by functions. These contracts allow developers to identify the intentional consequences of their packet processing algorithms.

Code Generation Though contract programming aids in checking and asserting if specifications and dependencies between operations hold, these checks are only (recommended to be) provided during time of development or within testing environments. Additionally, we do not want developers to have to sprinkle contracts throughout their NFs or implement logic to traverse or backtrack from one header, payload, or set of bytes to another just for the sake of validation. Instead, our approach allows developers to specify a set of dependencies and conditions up front via macros [20], which in turn get translated into contracts.

We develop our technique as an extension to the NetBricks framework and programming model [29], illustrating the efficacy of our approach through two examples: 1) updating the IPv6 payload length of a packet in the context of changes to an extension header; and 2) transitioning an invalid TCP request (based on an MTU—Maximum Transmission Unit—threshold), into that of an ICMPv6 *Packet Too Big* response [5]. We evaluate our prototype by examining syntax additions, compilation times, and possible runtime overheads. In Section 6, we discuss our examples within the context of a couple real-world implementations, Onos and Facebook network code, where our approach could be beneficial.

2 Motivation

Choosing between NF architectures and/or network programming languages has become a non-trivial process: What kind of programming paradigm should one choose for packet processing, e.g. functional, dataflow, or imperative? Should the framework support the OpenFlow protocol or be composed of its own data plane and control plane layers? What are the most important facets of the system or application: performance, usability and reconfiguration, reliability? There are many choices and abstractions to deliberate on, see sections 6 and 1, yet most only provide a subset of safety, design benefits, or degrees of freedom for which kinds of applications can be executed. We illustrate two specific challenges in defining a better way forward.

The Limits of Correctness In the interest of handling correctness and ensuring network programs satisfy specification, there are several efforts which have experimented with verifying networking constructs. For example, a language like NetKat [3], based on proven semantic and type theoretic foundations, provides static checking for reachability, guarantees non-interference between programs, and supports first-class primitives for the filtering, modifying, and transmitting of packets. However, though powerful, NetKat is limited in what logic it can check for and what protocols and actions it can support, as all programs must conform to the OpenFlow flow table—its compilation target. While OpenFlow *is* used in practice, its model is limited in terms of interface, protocol, and field support, especially for newer, experimental features. Due to this coupling with OpenFlow, along with a lack for handling arbitrary logic in packet processing, NetKat does not present a generalized solution.

Arbitrary Logic & Variable-length Data As described in Section 1, many network programs contain operational logic that's only applied based on the IETF or similar specifications they conform to. Some even define their own inspired-by protocols without a formalized spec [17]. One major component of the IPv6 protocol specification that has been left unsupported by many NF frameworks is that of IPv6 extension headers. Traffic containing such a header is usually dropped in practice and considered a "threat to the Internet" [15]. In skipping support for extension headers, packet-processing paradigms can avoid dealing with variable-length data—the specs of these headers contain fields with variable-byte-sized data—and complex header chaining dependencies, as these headers can be stacked upon each other to no end. However, as unique applications for programmable networks that make use of these extensions are constantly being explored [7], we must provide programmatic abstractions for adhering to the conditions of these protocols while also being amenable to new, experimental ones down the line whether they're used in industry or proposed in research.

3 Kinds of Contracts

3.1 Design by Contract

The Eiffel programming language made design by contract first-class, focusing primarily on how runtime contracts can be turned on for monitoring and testing situations so that developers can “sit back, and just watch their contracts be violated” [26]. The key idea behind the approach is that elements of a software system collaborate with each other on the basis of mutual obligations and benefits, driven by dependencies and related components in the system. These contracts are usually separated into *pre* (input/ingress) and *post* conditions (output/egress), where invariants can be asserted on for incoming and outgoing data accordingly.

In our system, design by contract-styled assertions help programmers articulate what the values of fields in a header should be equal to, bound by, approximate to, or how these values may have shifted during packet transformation (e.g. swapping of MAC addresses). From a processing perspective, the input precondition runs when the packet enters a NF and the postcondition runs as the packet is exiting the function.

3.2 Static Assertions

Static assertions, popularized in the C, C++, and D languages, allow for compile-time assertions of statically defined expressions, e.g. constants, statics. Beyond just checking for specific values, static assertions can be used to enforce fields on *struct* types and check if a pointer’s underlying value is the same when coerced to another type. NF programs tend to be comprised of many constants referring to values derived from specifications. For example, the IPv6 minimum MTU value is 1280 [6], but is actually 1294 in practice when the Ethernet header is included. Our approach can check this caveat statically at the call site where the NF is defined—not where it’s instantiated—via compile-time assertions in our prototype for constant checking. Additionally, thanks to *conditional compilation* (see 4.1 for more information), static assertions remain in release binaries.

3.3 Static Order-Persevering Headers

With our approach implemented in NetBricks, we were given a head start toward better validation mechanics with a strong, static type system and framework for programming NFs in a map-reduce fashion. To add packet headers in NetBricks, you define a *struct* with the appropriate fields, as you would do in C or P4 for example. All structs must implement a trait[1] containing an *associated type* that is defined as **PreviousHdr**:

```
impl EndOffset for Ipv6Hdr {
    type PreviousHdr=EthHdr ;
    fn offset(&self) -> usize { 40 }
}
```

When parsing a packet within an NF, the order is guaranteed by the defined *PreviousHdr*. Given any other order (e.g. parsing an IPv6 header after a ICMPv6 header), the type checker will throw a compile-time error. In our prototype, we leverage this statically-defined order mechanism on headers (4) to ensure that incoming and outgoing packet header ordering is preserved according to encoded expectations.

[1]Traits are used to define shared behavior in Rust, similar to interfaces in other languages [8].

4 Implementation

We have developed a prototype[2] that extends the NetBricks programming model via macro-based metaprogramming with very little additional syntax. Instead of having to manually incorporate or implement all of the contract methodologies described in section 3 throughout a NF code base, our contracts extension can be used gradually, i.e. on certain NFs and not others, as well as retroactively on existing NFs with just a few easy steps: (*i.*) import our **check** library into an NF module; then (*ii.*) identify an NF to validate, and mark it; and then (*iii.*) specify contracts at the beginning and end of a NF based on properties that the developer wants to uphold.

Once introduced, these contracts rewrite NFs to include mechanisms for storing runtime info (used for checking outgoing packets), generating validations, assertions, logging facilities, and flag checks for conditional compilation.

Initialization As seen in Figure 1, the *check* attribute macro (surrounded by brackets) is responsible for three steps in the contract generation process. Firstly, it identifies that the developer wants this NF to be “checked,” which means it can be used gradually over time. Secondly, by designating that this function has contract-checking **on**, we are then able to parse specific keywords, i.e. *pre*, *post*, in the figure that we want to rewrite and generate assertions from. Lastly, it performs a series of initialization operations, including turning-on specialized logging facilities and lazily instantiating a runtime hashmap that’s used to store all the headers as part of the input packet to create a mirror of the contents of the packet entering the NF. Building this map allows us to store header information for tracing, analysis, and further checks throughout the processing lifecycle, all the while producing a series of iterative steps to parse through the packet header-by-header, based on the order specified by the code.

Macro Expansion The generation of code from macros *ingress_check!* and *egress_check!* occurs prior to the NF program being subjected to Rust’s type-checker, i.e. occurring in a separate compilation step. Of note, the order key in both the pre and post assertions, specified by the developer, allows us to match on the header-order within the contents of the packet itself, as all parsing of headers requires explicit type annotations in NetBricks under the hood. If the expected order does not match up on either ingress or egress checks, a compile-time error is thrown (as per 3.3).

Ingress and Egress Contracts Figure 1 exhibits how contracts are extended into an NF. This example checks if an incoming TCP packet is beyond the valid MTU threshold,

[2]Openly accessible as a branch on Github.

```
#[ check (IPV6_MIN_MTU = 1280)]
fn send_too_big {
.pre(box pkt {
    ingress_check! {
        input: pkt,
        order : [EthHdr=>Ipv6Hdr=>TcpHdr<Ipv6Hdr>],
        checks: [( payload_len[Ipv6Hdr] , >,
                   IPV6_MIN_MTU )]
    }})
...filter/map/group_by operations...
.post(box pkt {
    egress_check! {
        input: pkt,
        order :[EthHdr=>Ipv6Hdr=>Icmpv6PktTooBig<...>],
        checks:[( checksum[Icmpv6PktTooBig] , neq,
                  checksum[TcpHdr<Ipv6Hdr>] ),
                ( payload_len[Ipv6Hdr] , ==,  1240 ),
                ( src[Ipv6Hdr] , ==,  dst[Ipv6Hdr] ),
                ( dst[Ipv6Hdr] , ==,  src[Ipv6Hdr] ),
                ( .src[EthHdr] , ==,  .dst[EthHdr] ),
                ( .dst[EthHdr] , ==,  .src[EthHdr] )]
        }})
```

Figure 1: Pre and Post contracts on MTU example

and, if so, then rewrites the packet into that of an ICMPv6 *Packet Too Big* response which gets returned to the source sender. As mentioned, the incoming and outgoing order lists reveal how the packet should be transformed throughout the main body of the function. Egress checks compare the values of fields and functions on the current, outgoing packet (left-hand side of each check) to values that are either literal integers or integer expressions, or functions or fields from the original, incoming packet (right-hand side). In this example, if it has to return to sender, this means swapping Ethernet addresses and IPv6 source and destination addresses from the original input. The checks presented here would fail or throw errors if the inner body's logic did not account for these swap operations.

4.1 Conditional Compilation

As previously mentioned, design by contract systems were devised with the intention that contracts would be applied during simulation, testing, and debugging stages of development. Our approach combines these kinds of runtime, dynamic assertions, which capture arbitrary logic and values, with static assertions and compile-time type checking. As shown in our evaluation of the runtime cost of our prototype, 5.3, runtime-checking and initialization accrue a penalty, which is manageable during NF development, not production. We leverage Rust's compile-time feature-flags [33] to only generate dynamic, runtime contracts for debug and testing modes, while ensuring all static information and assertions remain in the finalized, production program binaries.

5 Evaluation

In this section, we evaluate the possible overheads of our approach, including profiling its runtime cost by sampling the call graph during a packet's run through an NF.

Setup In our experimental setup, we ran NetBricks within an Ubuntu Docker container on a local VirtualBox VM. NetBricks uses DPDK [29] for fast packet I/O, which we have properly set up within the VM and container. We used MoonGen [10] to generate varying packet captures (pcaps) for our testing and evaluation harness. We looked at three factors in evaluating our technique for the design of NFs: (*i.*) **additional syntax** (*LoC*—lines of code); (*ii.*) **compilation-time** added to our two example NFs; (*iii.*) and **runtime overhead** of ingress and egress contract generation.

5.1 Syntax Added

LoC run	**lang**	**files**	**lines**	**code**
mtu-too-big: Contracts ON	rust	2	214	183
mtu-too-big: Contracts OFF	rust	2	189	158
mtu-too-big: Contracts ON	toml	1	19	16
mtu-too-big: Contracts OFF	toml	1	16	13
mtu-too-big: Contracts ON	total	3	233	199
mtu-too-big: Contracts OFF	total	3	205	171
Change		0	+28	+28

Table 1: Syntax additions for our MTU NF (unexpanded)

Being that most of the work in our implementation is centered around the macro generation of contracts, it's not surprising to see that our non-expanded measure of *LoC* (Table 1) is minimal. We import a few libraries (crates in the Rust ecosystem), including our **check** library, into NetBricks. The extra crates are used for logging and assertion control around error handling and operations that we can match on. Minus boilerplate, most of the additional code comes from the specifications themselves, as there is no bound on the number of possible validations that can be added.

In Table 1, we choose to show *LoC* without expansion, to faithfully represent the experience of the network function developer. At the outset of this project, we wanted to avoid altering many of the core NetBricks APIs or its existing example NFs. With our contract generation prototype, we increased our examples' programs and build configurations by an average of 23 lines, or less than 10 percent.

5.2 Compilation Times

One of the most important factors we wanted to consider was compilation time, as we did not want programmers to pay much of a penalty while developing NFs. Table 2

compile times / cargo build	**example**	**mean (s)**	**stddev (s)**	**user (s)**	system (s)	min (s)	max (s)
Contracts - Off	srv6-change-pkt	26.039	3.286	0.631	10.715	22.330	33.230
Contracts - On	srv6-change-pkt	25.099	2.398	0.549	11.697	20.238	28.220
Effect		**-0.94**	**-0.888**	**-0.082**	**+0.982**	**-2.092**	**-5.01**
Contracts - Off	mtu-too-big	21.652	2.202	0.537	9.201	18.528	25.191
Contracts - On	mtu-too-big	26.052	1.858	0.650	10.851	22.165	28.346
Effect		**+4.4**	**-0.344**	**+0.113**	**+1.65**	**+3.637**	**+3.155**

Table 2: Compile times running "cargo build" for extension header NF example and MTU NF example

compares our two example network function cases with contract generation turned on and off. For each of these runs, the build was compiled from a warm, incremental cache and then rebuilt from that cache ten times. As shown, with our contract generation system applied, the standard deviation across all builds was less than a second overall. Further, mean time even improved in the case of our extension header example. Though there is room for optimization, these results show that our technique doesn't negatively affect developers during development.

5.3 Runtime Cost

We've discussed how our objective for programmers is to be able to specify their checks up front as they build out NFs and test them. Knowing all of the initialization and setup we have to concoct on behalf of the contract engine, we were aware that the runtime cost would be problematic if the NF ran in production. But, how problematic would it be?

For this evaluation, we ran our invalid MTU example and sent a packet at a time through it, tracing the call graph throughout the function and sampling it. To trace and visualize the effect, we used the Flame Graph approach [14], popularized in industry. The graph is illustrated in Figure 2. As expected, our precondition routine takes up a majority of the function's execution time. This is mainly due to creating a copy of the incoming packet and parsing each header within it. Our egress macro, for example, does much less and spans less of the execution graph.

Further optimizing our implementation's code generation and how we store and parse the packet in the evaluated program would lessen the runtime cost of our technique in practice and eventually make it possible to include some dynamic contracts for in-production use-cases. Nonetheless, as we've stated before, to focus our technique on the design phase of NF writing, the current version of the library compiles away most of this generated code upon production builds—not slowing down the runtime.

6 Discussion and Future Work

Thus far, we've demonstrated our technique on a few simple yet practical NF examples. In this section, we discuss how our work could benefit practical programs and applications out in the wild. Then, we explore where we want to take the approach going forward.

6.1 Real-World Example: ONOS Routing

One of the cases we've evaluated includes adding additional segments to a IPv6 segment routing extension header [11]. In Section 2, we mentioned that IPv6 extension headers were difficult to handle due to their variability, causing network operators to write rules to drop packets that contain them and NF frameworks avoiding their logic altogether. ONOS [4], the open network operating system, is a controller platform supporting a wide variety of SDN use cases, including support for the *Routing* header extension. Nonetheless, the most complex logic that the header entails is that of adding and removing segments, which then triggers effects on the *Last Entry* (the index into the stack of segments) and *Segments Left* (the number of route segments remaining to be processed) fields. These triggered events provide a good story for our implementation because the ONOS class [1] does not account for changes in the Routing header stack; instead, it just works on a minimal set of fields for reading these headers along the network path. With an approach like the one we've exhibited in our paper, more assurance could be given for handling the complex logic of variable-length information.

6.2 Real World Example: Katran

Katran [30] is Facebook's Layer 4 software load balancer, built on a data plane using the eBPF VM. While Katran is deployed at scale and processes packets at high speeds, its codebase lacks specifications and checks in logic; it also includes many hardcoded values and a myriad of constants. Similar to the invalid MTU example used throughout this paper, they too have a IPv6 *Packet Too Big* response function:

```
static inline int send_icmp6_too_big(...) {
  ...
  icmp6_hdr->icmp6_type = ICMPV6_PKT_TOOBIG;
  icmp6_hdr->icmp6_mtu = htonl(
    MAX_PCKT_SIZE -sizeof(struct eth_hdr)
  )}
```

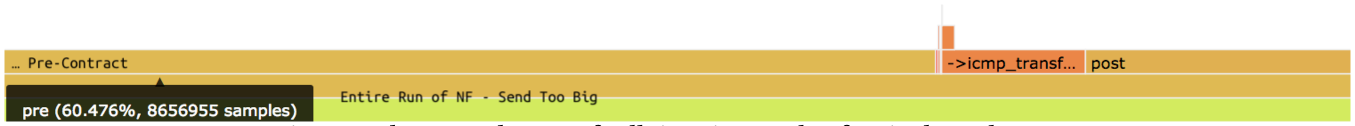

Figure 2: Flame graph trace of call time, in samples, for single-packet runs

While we expect that a test suite would capture possible bugs in arbitrary logic and pointer references, there are no abstractions within the programs themselves to ensure the validity of fields and what values they can possibly be. Furthermore, what if constants like *MAX_PCKT_SIZE* were changed within the upstream module that instantiated them? Leveraging static assertions like we do in our work could benefit functions like the one shown here.

6.3 Next Steps

6.3.1 ***Deployment Models*** Throughout this paper, we've explained how runtime assertions in our system are best used during the design phase of programming NFs. Yet within the ecosystem of network programming, our technique can be extended beyond design and integration testing. We would like to be able to *turn on* contracts automatically if our system is running within an environment acting as a simulation network, e.g. Mininet [22], or within production deployments for certain kinds of traffic, e.g. probe packets sent for NF monitoring or health checking.

6.3.2 ***Hinting and Feedback*** The major goal of our work is aiding in the development of network programs, especially those that involve arbitrary complexity and interact with changing dependencies. In our system, runtime errors look akin to typical assertion errors presented by other languages or frameworks. Even though our errors include some context—the expectation vs. what actually happened—they normally do not articulate anything related to the specification itself or what dependencies triggered it. We would like to take a page from the recent work in program slicing and compiler design (as demonstrated in languages like Elm and Rust) and provide feedback via hints to the programmer while they build out NFs during development.

Leverage static analysis of input programs In our code generation step(s), we look for a set of explicit tokens to rewrite and incorporate seamlessly within the context of a given NF. However, by adding the *check* macro to a function, we're able to walk the entire AST (Abstract Syntax Tree) of the input NF before it gets compiled, allowing us to perform static analysis on the function to find bugs [16] at compile-time. Further leveraging static analysis would allow us to limit the need for certain runtime contracts.

Interactive feedback Good feedback is crucial when an error occurs. Modern type systems provide more context to type errors (beyond just which line propagated the error itself), by suggesting more precise types for the developer to include in their programs. In designing network function paradigms, we want to build off our prototype and expand to include helpful information about where contract errors occur at the boundary and in which ways the errors may be debugged.

7 Related Work

Our approach builds upon a growing literature on contract-driven validation of programs. In Sections 1 and 3.1 we referenced our inspiration from D's contract programming model, which was itself inspired by the system developed for Eiffel. Though our approach is unique within the field of programmable networks, contract programming has gained popularity as extensions to functional programming languages like Clojure (via Spec [27]) and *Racket* [2]. Racket also includes mechanisms for generating contracts from macros.

Regarding type systems and ways to validate network programs, we've already mentioned NetKat [3], which has had follow-up pieces in literature, including probabilistic variants [31]. Recently, p4v [23] was published and is motivated by real-world examples; it attempts to find bugs in P4 programs and verify program properties by incorporating domain-specific assumptions into a constraint solver.

Languages for network function specification exist within industry, including TOSCA [28], a templating *metamodel* for network function virtualization. Also related is *Assert-P4* [13], which is a proposed approach for checking P4 programs that is also based on assertion checking. Combining assertions with symbolic execution, it finds bugs motivated by controller misconfiguration and code circumvention and gives developers that ability to specify properties about their programs. Their work is very P4-specific and does not provide examples of complex pipelines involving arbitrary dependencies, similar to the cases we've discussed in this paper.

8 Conclusion

In this paper, we provide a hybrid-approach and implementation for checking and validating the arbitrary logic and side effects typically part of network functions by combining design by contract, static assertions and type-checking, and code generation via macros. We were able to build-out and incorporate our technique within an existing network function framework, without penalizing the developer or increasing the complexity that they already have to handle. We want to explore this space further and provide better tooling and interaction models for anyone programming networks.

Acknowledgments

We want to thank Professor Justine Sherry from Carnegie Mellon University for inspiring this project while the first author was a part of her Programmable Networks class. Additionally, we're thankful for the guidance and support from Professor Aurojit Panda (New York University) for authoring NetBricks and answering related questions, and Chas Emerick for his all-around help and guidance. Finally, an extra special thank you goes out to the Occam networking team at Comcast (i.e. Peter Cline, Daniel Jin, Andrew Wang, Michael Winslow, Chris Rollins, Paul Cleary, Jon Moore) for their continued support, code, instruction, and feedback.

References

[1] 2017. Routing.java. https://github.com/opennetworkinglab/onos/blob/021d2eb175b8e46d4690cd9e1243301ddd903bcc/utils/misc/src/main/java/org/onlab/packet/ipv6/Routing.java. (2017).

[2] 2018. Racket Contracts. (2018). https://docs.racket-lang.org/guide/contracts.html

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[4] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2620728.2620744

[5] A. Conta, S. Deering, and M. Gupta. 2006. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. RFC 4443. RFC Editor. http://www.rfc-editor.org/rfc/rfc4443.txt http://www.rfc-editor.org/rfc/rfc4443.txt.

[6] Stephen E. Deering and Robert M. Hinden. 1998. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. RFC Editor. http://www.rfc-editor.org/rfc/rfc2460.txt http://www.rfc-editor.org/rfc/rfc2460.txt.

[7] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen. 2017. SRLB: The Power of Choices in Load Balancing with Segment Routing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2011–2016. https://doi.org/10.1109/ICDCS.2017.180

[8] The Rust Project Developers. 2018. The Rust Programming Language, 2st ed. https://doc.rust-lang.org/1.30.0/book/2018-edition/. (2018).

[9] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/1629575.1629578

[10] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference*.

[11] Clarence Filsfils, Stefano Previdi, John Leddy, Satoru Matsushima, and daniel.voyer@bell.ca. 2018. *IPv6 Segment Routing Header (SRH)*. Internet-Draft draft-ietf-6man-segment-routing-header-15. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-6man-segment-routing-header-15.txt http://www.ietf.org/internet-drafts/draft-ietf-6man-segment-routing-header-15.txt.

[12] D Language Foundation. 2018. Contract Programming. (2018). https://dlang.org/spec/contracts.html

[13] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3185467.3185499

[14] Brendan Gregg. 2018. Flame Graphs. (2018). http://www.brendangregg.com/flamegraphs.html

[15] L. Hendriks, P. Velan, R. d. O. Schmidt, P. de Boer, and A. Pras. 2017. Threats and surprises behind IPv6 extension headers. In *2017 Network Traffic Measurement and Analysis Conference (TMA)*. 1–9. https://doi.org/10.23919/TMA.2017.8002912

[16] David Hovemeyer and William Pugh. 2007. Finding More Null Pointer Bugs, but Not Too Many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. ACM, New York, NY, USA, 9–14. https://doi.org/10.1145/1251535.1251537

[17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*.

[18] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. 2015. Stateless Network Functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox '15)*. ACM, New York, NY, USA, 49–54. https://doi.org/10.1145/2785989.2785993

[19] Robert Klarer, John Maddock, Beman Dawes, and Howard Hinnant. 2008. Static Assertions. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1330.pdf. (2008).

[20] E. E. Kohlbecker and M. Wand. 1987. Macro-by-example: Deriving Syntactic Transformations from Their Specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 77–84. https://doi.org/10.1145/41625.41632

[21] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297. https://doi.org/10.1145/354871.354874

[22] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*.

[23] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4V: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[24] Ahmed M. Medhat, Tarik Taleb, Asma Elmangoush, Giuseppe A. Carella, Stefan Covaci, and Thomas Magedanz. 2017. Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges. *Comm. Mag.* 55, 2 (Feb. 2017), 216–223. https://doi.org/10.1109/MCOM.2016.1600219RP

[25] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40–51. https://doi.org/10.1109/2.161279

[26] Bertrand Meyer. 2018. Why not program right? (2018). https://bertrandmeyer.com/2018/05/24/not-program-right

[27] Alex Miller. 2018. spec Guide. (2018). https://clojure.org/about/spec

[28] Organization for the Advancement of Structured Information Standards. 2017. TOSCA Simple Profile for Network Functions Virtualization (NFV)âĂŤVersion 1.0. (2017). http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html

[29] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 203–216. http://dl.acm.org/citation.cfm?id=3026877.3026894

[30] Nikita Shirokov and Ranjeeth Dasineni. 2018. Open-sourcing Katran, a scalable network load balancer. (2018). https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[31] Steffen Smolka, David Kahn, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Deciding Probabilistic Program Equivalence in NetKAT. *CoRR* abs/1707.02772 (2017). arXiv:1707.02772 http://arxiv.org/abs/1707.02772

[32] Aaron Weiss. 2018. Reasoning with Types in Rust. (2018). https://aaronweiss.us/posts/2018-02-26-reasoning-with-types-in-rust.html

[33] Justin Worthe. 2018. Compile Time Feature Flags in Rust. (2018). https://www.worthe-it.co.za/programming/2018/11/18/compile-time-feature-flags-in-rust.html