Checking-in on Network Functions

by Zeeshan Lakhani and Heather Miller

@ Carnegie Mellon University
The rise of network functions?
The rise of network functions?
The rise of network functions?
The rise of network functions?
The rise of network functions?

class firewall(DynamicPolicy):

def __init__(self):
    # Initialize the firewall
    print "initializing firewall"
    self.firewall = {} 
    super(firewall,self).__init__(true)
    import threading
    self.ui = threading.Thread(target=self.ui_loop)
    self.ui.daemon = True
    self.ui.start()

def AddRule (self, mac1, mac2):
    if (mac2,mac1) in self.firewall:
        print "Firewall rule for \%s: \%s already exists" % (mac1,mac2)
        return
    self.firewall[(mac1,mac2)]=True
    print "Adding firewall rule in \%s: \%s" % (mac1,mac2)
    self.update_policy()

def DeleteRule (self, mac1, mac2):
    try:
        del self.firewall[(mac1,mac2)]
        print "Deleting firewall rule in \%s: \%s" % (mac1,mac2)
        self.update_policy()
    except:
        pass
    try:
        del self.firewall[(mac2,mac1)]
        print "Deleting firewall rule in \%s: \%s" % (mac1,mac2)
        self.update_policy()
    except:
        pass

class BlacklistDropper(Application):
    def init(self, blacklist):
        flow = self.make_wildcard_flow()
        flow["tp.dst"] = 53
        eds = self.apply_elem(flow, ["DnsDpi"])
        if(self.check_elems_installed(eds)):
            self.installed = True
            droppers = list()

    def handle_trigger(self, ed, trigger):
        if(trigger["type"] == "BlacklistedQuery"):
            src_flow = self.make_wildcard_flow()
            src_flow["nw.src"] = trigger["src.ip"]
            eds = apply_elem(src_flow, ["DropAll"])
            if(self.check_elems_installed(eds)):
                droppers.append(eds[])
Writing network functions is not “composed of nothing more than algorithms and small programs”\[1\]

- complex routing and load balancing policies
- traffic monitoring
- experimental/new specifications, protocols, and headers
- computation and aggregation
  (e.g. In-Network Computation is a Dumb Idea Whose Time Has Come)

\[1\] Cultures of programming: Understanding the history of programming through controversies and technical artifacts by Tomas Petricek, University of Kent, UK, 2019
Motivation
If I program in React, can I program a network function?
Motivation

If I program in React, can I program a network function?

How do we know what we’re doing is right?
If I program in React, can I program a network function?

How do we know what we’re doing is right?

How can we iterate?
Motivation

- Limits of Correctness
  - e.g. reliance on OpenFlow protocol
Motivation

- Limits of Correctness
  - e.g. reliance on OpenFlow protocol
  - Arbitrary (ad-hoc) Logic & Variable-length Data, e.g. Ipv6 Extensions, ndp options
  - packet length
  - failure and reconfiguration
Motivation

\begin{verbatim}
if(ntohs(ip->ip6_plen) > (plen - 40)) [2]
goto bad;
\end{verbatim}

[2] The Click Modular Router by Eddie Kohler, et. al., Laboratory for Computer Science, MIT, 1999
Two examples

**MTU: Send Too Big**
- swap ethernet addresses
- swap src/dst
- change protocol
- set mtu info
- calculate checksum

**Ipv6 Extension Headers: SRH**
Two examples

**MTU: Send Too Big**

- swap ethernet addresses
- swap src/dst
- change protocol
- set mtu info
- calculate checksum

**Ipv6 Extension Headers: SRH**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Protocol</th>
<th>RFC</th>
<th>IANA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Hop-by-Hop Options</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>43</td>
<td>Routing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>44</td>
<td>Fragment</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>50</td>
<td>Encapsulating Security Payload</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>51</td>
<td>Authentication</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>60</td>
<td>Destination Options</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>135</td>
<td>Mobility Header</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>139</td>
<td>Host Identity Protocol</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>140</td>
<td>Shim6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>253</td>
<td>Experiments/testing purposes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>254</td>
<td>Experiments/testing purposes</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Kinds of Contracts
Kinds of Contracts

focused 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”

Design by Contract

erased on release binaries
focused 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"

\[ \triangleright \text{erased on release binaries} \]

compile-time assertions for consts, statics

\[ \triangleright \text{remain in release binaries} \]
Kinds of Contracts

- **Design by Contract**
  - focused 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"
  - erased on release binaries

- **Static Assertions**
  - compile-time assertions for consts, statics
  - remain in release binaries

```rust
impl EndOffset for Ipv6Hdr {
    type PreviousHdr = EthHdr;
    fn offset(&self) -> usize { 40 }
}
```
dependencies and related components in the system. These contracts are usually separated into \textit{pre} (input/ingress) and \textit{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.
Kinds of Contracts: Design by Contract

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.
Kinds of Contracts: **Design by Contract**

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.
Kinds of Contracts: **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.
Kinds of Contracts: **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.
Kinds of Contracts: **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.
Kinds of Contracts: Static Order-Persevering Headers.

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.
Kinds of Contracts: Static Order-Persevering Headers

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.
Implementation
Implementation

prototyped as a **gradual** extension to *NetBricks* (i.e. *NetBricks: Taking the V out of NFV, OSDI 2016*)
Implementation

prototyped as a **gradual** extension to **NetBricks** (i.e. NetBricks: Taking the V out of NFV, OSDI 2016)

Focused on Zero-Copy Soft Isolation
Implementation

- prototyped as a *gradual* extension to **NetBricks** (i.e. NetBricks: Taking the V out of NFV, OSDI 2016)
- implemented as a small rust library to easily write specifications, which *generates code* for validations and assertions at compile-time

Focused on Zero-Copy Soft Isolation
Implementation

- prototyped as a **gradual** extension to **NetBricks** (i.e. NetBricks: Taking the V out of NFV, OSDI 2016)
- implemented as a small rust library to easily write specifications, which **generates code** for validations and assertions at compile-time

**Focused on Zero-Copy Soft Isolation**

**macros** turn checks into static and dynamic contracts
Implementation

Checking-in on Network Functions
In Action.

```r
#[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])
    })
  })
```
In Action.

```rust
fn send_too_big {
    .pre(box pkt {
        ingress_check! {
            input: pkt,
            order: [EthHdr=&gt;Ipv6Hdr=&gt;TcpHdr&lt;Ipv6Hdr&gt;],
            checks: [(payload_len[Ipv6Hdr], &gt;, IPV6_MIN_MTU)]
        }
    })
    ...filter/map/group_by operations...
    .post(box pkt {
        egress_check! {
            input: pkt,
            order: [EthHdr=&gt;Ipv6Hdr=&gt;Icmpv6PktTooBig&lt;...&gt;],
            checks: [(checksum[Icmpv6PktTooBig], neq, checksum[TcpHdr&lt;Ipv6Hdr&gt;]),
                     (payload_len[Ipv6Hdr], ==, 1248),
                     (src[Ipv6Hdr], ==, dst[Ipv6Hdr]),
                     (dst[Ipv6Hdr], ==, src[Ipv6Hdr]),
                     (.src[EthHdr], ==, .dst[EthHdr]),
                     (.dst[EthHdr], ==, .src[EthHdr])
        }
    })

order is checked statically via a trace of packet contents
```
Checking-in on Network Functions

order is checked statically via a trace of packet contents

pre-checks validate incoming contents and store contents @ runtime
In Action.

```rust
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], ==, 1248),
      (src[Ipv6Hdr], ==, dst[Ipv6Hdr]),
      (dst[Ipv6Hdr], ==, src[Ipv6Hdr]),
      (src[EthHdr], ==, dst[EthHdr]),
      (dst[EthHdr], ==, src[EthHdr])
    })
  })
```

- The order is checked statically via a trace of packet contents.
- Pre-checks validate incoming contents and store contents at runtime.
- Post-checks validate transformed contents against pre-check contents.
Evaluation

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.
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 Moon-Gen [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.
### Evaluation: Syntax Added

<table>
<thead>
<tr>
<th>LoC run</th>
<th>lang</th>
<th>files</th>
<th>lines</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtu-too-big: Contracts ON</td>
<td>rust</td>
<td>2</td>
<td>214</td>
<td>183</td>
</tr>
<tr>
<td>mtu-too-big: Contracts OFF</td>
<td>rust</td>
<td>2</td>
<td>189</td>
<td>158</td>
</tr>
<tr>
<td>mtu-too-big: Contracts ON</td>
<td>toml</td>
<td>1</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>mtu-too-big: Contracts OFF</td>
<td>toml</td>
<td>1</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>mtu-too-big: Contracts ON</td>
<td>total</td>
<td>3</td>
<td>233</td>
<td>199</td>
</tr>
<tr>
<td>mtu-too-big: Contracts OFF</td>
<td>total</td>
<td>3</td>
<td>205</td>
<td>171</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td>0</td>
<td>+28</td>
<td>+28</td>
</tr>
</tbody>
</table>
## Evaluation: Compilation Time

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

Due to:
- mirroring and tracing packet contents
- runtime checks
- storage overhead
Evaluation: Runtime Cost

Due to:
- mirroring and tracing packet contents
- runtime checks
- storage overhead

Design Phase
Future Work

- deployment models / running contracts in simulation / CI
  - e.g. via Mininet / Containernet
- (further) leverage static analysis of input programs
- interactive feedback (many examples in UI tooling and langs like Elm and Rust)
  - program slicing
  - refinement via domain-specific heuristics and constraint solving
In Practice

Scoped Side Effects

- cascade
- packet length
- checksum
- etc...

Typed Packets

- $\text{Envelope} : T < T : \text{IpAddress}$
- $\text{Header} : \text{TCP}$

$\lambda \rightarrow \lambda$
we need better approaches to **VERIFY** and **INTERACT** with network functions and packet processing program properties.

Here, we provide a **HYBRID-APPROACH** and implementation for **GRADUALLY** checking and validating the arbitrary logic and side effects by **COMBINING** design by contract, static assertions and type-checking, and code generation via macros.

All without **PENALIZING** programmers at development time.