Rust-based Drivers & Verified Rust Applications on seL4

Robbie VanVossen

DornerWorks

seL4 Summit 2025

Introduction

- Many seL4-based systems have high-assurance requirements
- Often, high confidence needs to be extended into some user space applications
- How to increase user application confidence?

    1. Use memory safe & type safe languages
    2. Test the applications thoroughly
    3. Formally verify the applications

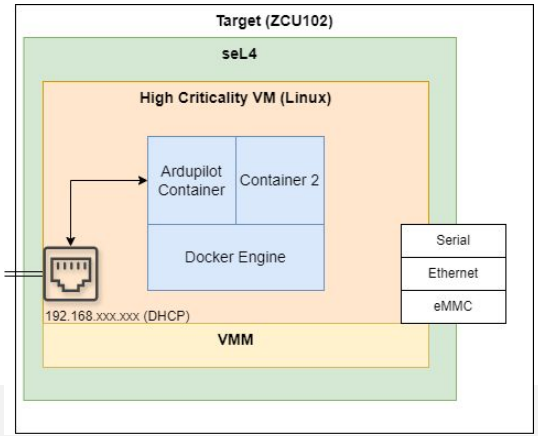
- Layering these approaches:
    - Improves confidence
    - Makes each subsequent approach easier

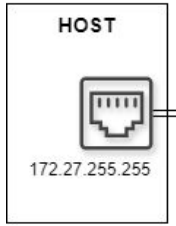
INSPECTA Tools

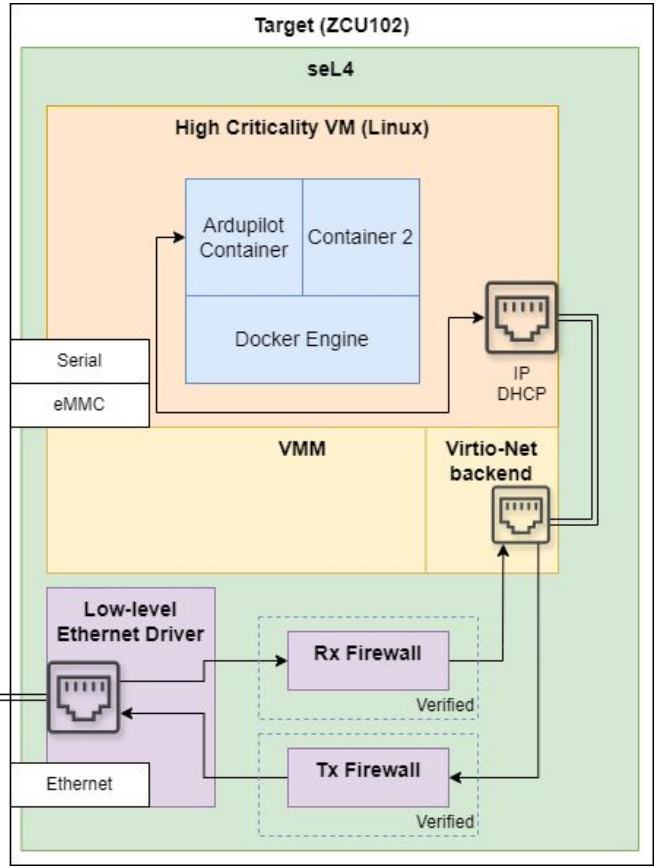
- HAMR & Verus
- HAMR uses system modelling and code generation to create skeletons and glue code
  - Automates and connects system architecture to the application implementation
- Verus is a tool for verifying the correctness of code written in Rust
- Architectural contracts are used to generate application level contracts in Verus
  - The user can then connect those to the manually developed Verus specs to verify the component and show that the contracts are met.
- Architectural contracts are also used to generate executable version of the contracts
  - This enables property-based testing

Use Case Architecture

- Legacy application run in the VM (autopilot)
  - Security is an after-thought
- Security requirements start getting addressed/changed during development
- Implement as many security features outside of the VM to give clear separation between
  - Application development
  - Security features







Low-level Ethernet Driver

- Developed without INSPECTA tools, but then ported into the generated architecture
- Implemented in Rust
- Mostly gives us memory safety, however
  - 10 blocks of manually written unsafe blocks: Pointer arithmetic and dereference for MMIO and DMA regions
- Utilized seL4 foundation libraries to reduce user effort
  - Microkit lib provides a convenient macro to translate memory region symbols (defined in microkit system file) into mutable, NonNull pointers
  - Traits from foundation libs and third-party libs allow for standardized interfaces and the use of already implemented features
- Difficult to test since it is mainly interacting with hardware

Firewalls

- Developed with INSPECTA tools
- Implemented in Rust within the HAMR-generated architecture
  - Few dependencies and no manually written unsafe code:
    - Good base-line confidence through memory & type safety
- Gained further confidence by implementing unit tests
  - ~95% code coverage!
  - Some downsides:
    - No real traceability from unit tests to requirements
    - Required some function mocking/stubbing
- Good initial target for verification

Verification approach

- There are different policies for data flowing into the VM vs data flowing out of the VM
  - Implemented as 2 separate components
  - Use the same library for parsing
- Requirements relate the policy to each relevant byte in an ethernet frame
  - Allows us to reason about the policy in relation to an ethernet frame
- Write architectural contracts which codify the natural language requirements

1.3 Rx_firewall: Copy through allowed udp port frames (*RC_INSPECTA_00-HLR-13*)))

The firewall shall copy a frame from an input port to its output port's message if that frame has a wellformed ethheader, the ethernet type is IPv4, the IPv4 packet is wellformed, the IPv4 packet uses the UDP protocol, and the UDP port is in the UDP port whitelist.

- An ethernet header is wellformed if the ethernet type is valid and the destination address is valid.
  - The ethernet type is valid if bytes 12-13 of the frame are 0x0800 or 0x0806 or 0x86DD.
  - The destination address is valid if bytes 0-5 of the frame are not 0x0000000000.
- An IPv4 packet is wellformed if the IPv4 protocol is valid and the IPv4 length is valid.
  - The IPv4 protocol is valid if byte 23 of the frame is 0x00 or 0x01 or 0x02 or 0x06 or 0x11 or 0x2B or 0x2C or 0x3A or 0x3B or 0x3C.
  - The IPv4 length is valid if bytes 16-17 of the frame are <= 9000.
- An IPv4 packet uses the UDP protocol if byte 23 of the frame is 0x11.
- The UDP port is in the whitelist if bytes 36-37 of the frame are one of the following:
  - [68]

A maximum IPv4 length of 9000 is selected since it is the standard JUMBO frame size for Maximum Transmission Unit (MTU). In most cases it will be closer to 1500.

1.4 Rx_firewall: Do not copy disallowed frame (*RC_INSPECTA_00-HLR-15*)))

The rx firewall shall not copy any frame originating from an input port to its output port if it does not match a valid frame as defined in the other HLRs.

Formalization of requirements as contracts

Guarantee Contract clause per requirement

Rx_firewall: Copy through allowed udp port frames (*RC_INSPECTA_00-HLR-13*)))

The firewall shall copy a frame from an input port to its output port's message if that frame has a wellformed ethheader, the ethernet type is IPv4, the IPv4 packet is wellformed, the IPv4 packet uses the UDP protocol, and the UDP port is in the UDP port whitelist.

```
compute

  guarantee hlr_05_rx0_can_send_arp:
    ((HasEvent(EthernetFramesRxIn0) &&
      (HasEvent(EthernetFramesRxOut0) &

  guarantee hlr_06_rx0_can_send_ipv4_tcp:
    ((HasEvent(EthernetFramesRxIn0) && valid_ipv4_tcp_port(EthernetFramesRxIn0)) ->:
      (HasEvent(EthernetFramesRxOut0) && (EthernetFramesRxIn0 == EthernetFramesRxOut0)));

  guarantee hlr_13_rx0_can_send_ipv4_udp:
    ((HasEvent(EthernetFramesRxIn0) && valid_ipv4_udp_port(EthernetFramesRxIn0)) ->:
      (HasEvent(EthernetFramesRxOut0) && (EthernetFramesRxIn0 == EthernetFramesRxOut0)));

  guarantee hlr_15_rx0_disallow:
     (HasEvent(EthernetFramesRxIn0) && !allow_outbound_frame(EthernetFramesRxIn0)) ->:
       (NoSend(EthernetFramesRxOut0));

  guarantee hlr_17_rx0_no_input:
     (!HasEvent(EthernetFramesRxIn0) ->: NoSend(EthernetFramesRxOut0));
```

GUMBO Spec Functions

Spec functions codify the byte specification from the requirements

1.3 Rx_firewall: Copy through allowed udp port frames (*RC_INSPECTA_00-HLR-13*)))

The firewall shall copy a frame from an input port to its output port's message if that frame has a wellformed ethheader, the ethernet type is IPv4, the IPv4 packet is wellformed, the IPv4 packet uses the UDP protocol, and the UDP port is in the UDP port whitelist.

- An ethernet header is wellformed if the ethernet type is valid and the destination address is valid.
  - The ethernet type is valid if bytes 12-13 of the frame are 0x0800 or 0x0806 or 0x86DD.
  - The destination address is valid if bytes 0-5 of the frame are not 0x0000000000.
- An IPv4 packet is wellformed if the IPv4 protocol is valid and the IPv4 length is valid.
  - The IPv4 protocol is valid if byte 23 of the frame is 0x00 or 0x01 or 0x02 or 0x06 or 0x11 or 0x2B or 0x2C or 0x3A or 0x3B or 0x3C.
  - The IPv4 length is valid if bytes 16-17 of the frame are <= 9000.
- An IPv4 packet uses the UDP protocol if byte 23 of the frame is 0x11.
- The UDP port is in the whitelist if bytes 36-37 of the frame are one of the following:
  - [68]

A maximum IPv4 length of 9000 is selected since it is the standard JUMBO frame size for Maximum Transmission Unit (MTU). In most cases it will be closer to 1500.

1.4 Rx_firewall: Do not copy disallowed frame (*RC_INSPECTA_00-HLR-15*)))

The rx firewall shall not copy any frame originating from an input port to its output port if it does not match a valid frame as defined in the other HLRs.

```
def valid_ipv4_udp(frame: RawEthernetMessage): Base_Types::Boolean :=
  frame_is_wellformed_eth2(frame) &&
    frame_has_ipv4(frame) &&
    wellformed_ipv4_frame(frame) &&
    ipv4_is_udp(frame);

def valid_ipv4_udp_port(frame: RawEthernetMessage): Base_Types::Boolean :=
  valid_ipv4_udp(frame) && frame_has_ipv4_udp_on_allowed_port_quant(frame);
```

GUMBO Spec Functions

Spec functions codify the byte specification from the requirements

1.3 Rx_firewall: Copy through allowed udp port frames (*RC_INSPECTA_00-HLR-13*)))

The firewall shall copy a frame from an input port to its output port's message if that frame has a wellformed ethheader, the ethernet type is IPv4, the IPv4 packet is wellformed, the IPv4 packet uses the UDP protocol, and the UDP port is in the UDP port whitelist.

- An ethernet header is wellformed if the ethernet type is valid and the destination address is valid.
  - The ethernet type is valid if bytes 12-13 of the frame are 0x0800 or 0x0806 or 0x86DD.
  - The destination address is valid if bytes 0-5 of the frame are not 0x0000000000.
- An IPv4 packet is wellformed if the IPv4 protocol is valid and the IPv4 length is valid.
  - The IPv4 protocol is valid if byte 23 of the frame is 0x00 or 0x01 or 0x02 or 0x06 or 0x11 or 0x2B or 0x2C or 0x3A or 0x3B or 0x3C.
  - The IPv4 length is valid if bytes 16-17 of the frame are <= 9000.
- An IPv4 packet uses the UDP protocol if byte 23 of the frame is 0x11.
- The UDP port is in the whitelist if bytes 36-37 of the frame are one of the following:
  - [68]

A maximum IPv4 length of 9000 is selected since it is the standard JUMBO frame size for Maximum Transmission Unit (MTU). In most cases it will be closer to 1500.

1.4 Rx_firewall: Do not copy disallowed frame (*RC_INSPECTA_00-HLR-15*)))

The rx firewall shall not copy any frame originating from an input port to its output port if it does not match a valid frame as defined in the other HLRs.

```
def frame_is_wellformed_eth2(frame: RawEthernetMessage): Base_Types::Boolean :=
  valid_frame_ethertype(frame) && valid_frame_dst_addr(frame);

def valid_frame_ethertype(frame: RawEthernetMessage): Base_Types::Boolean :=
  frame_has_ipv4(frame) || frame_has_arp(frame) || frame_has_ipv6(frame);

def valid_frame_dst_addr(frame: RawEthernetMessage): Base_Types::Boolean :=
    !((frame(0) == u8"0") &&
      (frame(1) == u8"0") &&
      (frame(2) == u8"0") &&
      (frame(3) == u8"0") &&
      (frame(4) == u8"0") &&
      (frame(5) == u8"0"));

def frame_has_ipv4(frame: RawEthernetMessage): Base_Types::Boolean :=
    frame(12) == u8"8" && frame(13) == u8"0";

def frame_has_ipv6(frame: RawEthernetMessage): Base_Types::Boolean :=
    frame(12) == u8"134" && frame(13) == u8"221";

def frame_has_arp(frame: RawEthernetMessage): Base_Types::Boolean :=
    frame(12) == u8"8" && frame(13) == u8"6";
```

Generated Verus Spec

```
pub open spec fn valid_ipv4_udp(frame: SW::RawEthernetMessage) -> bool
{
  Self::frame_is_wellformed_eth2(frame) && Self::frame_has_ipv4(frame) &&
    Self::wellformed_ipv4_frame(frame) &&
    Self::ipv4_is_udp(frame)
}

pub open spec fn valid_ipv4_udp_port(frame: SW::RawEthernetMessage) -> bool
{
  Self::valid_ipv4_udp(frame) && Self::frame_has_ipv4_udp_on_allowed_port_quant(frame)
}
```

```
ensures
  // BEGIN MARKER TIME TRIGGERED ENSURES
  // guarantee hlr_05_rx0_can_send_arp
  api.EthernetFramesRxIn0.is_some() && Self::valid_arp(api.EthernetFramesRxIn0.unwrap()) ==>
    api.EthernetFramesRxOut0.is_some() &&
      (api.EthernetFramesRxIn0.unwrap() == api.EthernetFramesRxOut0.unwrap()),
  // guarantee hlr_06_rx0_can_send_ipv4_tcp
  api.EthernetFramesRxIn0.is_some() && Self::valid_ipv4_tcp_port(api.EthernetFramesRxIn0.unwrap()) ==>
    api.EthernetFramesRxOut0.is_some() &&
      (api.EthernetFramesRxIn0.unwrap() == api.EthernetFramesRxOut0.unwrap()),
  // guarantee hlr_13_rx0_can_send_ipv4_udp
  api.EthernetFramesRxIn0.is_some() && Self::valid_ipv4_udp_port(api.EthernetFramesRxIn0.unwrap()) ==>
    api.EthernetFramesRxOut0.is_some() &&
      (api.EthernetFramesRxIn0.unwrap() == api.EthernetFramesRxOut0.unwrap()),
  // guarantee hlr_15_rx0_disallow
  api.EthernetFramesRxIn0.is_some() && !(Self::allow_outbound_frame(api.EthernetFramesRxIn0.unwrap())) ==>
    api.EthernetFramesRxOut0.is_none(),
  // guarantee hlr_17_rx0_no_input
  !(api.EthernetFramesRxIn0.is_some()) ==> api.EthernetFramesRxOut0.is_none(),
```

Ethernet Frame Parser Library

```
impl EthFrame {
    pub fn parse(frame: &[u8]) → (r: Option<EthFrame>)
        requires
            frame@.len() ≥ TCP_TOTAL,
            frame@.len() ≥ UDP_TOTAL,
            frame@.len() ≥ ARP_TOTAL
        ensures
            valid_arp_frame(frame) == res_is_arp(r),
            valid_ipv4_frame(frame) == res_is_ipv4(r),
            valid_ipv6_frame(frame) == res_is_ipv6(r),
            valid_tcp_frame(frame) == res_is_tcp(r),
            valid_udp_frame(frame) == res_is_udp(r),
            valid_tcp_frame(frame) ⟹ tcp_port_bytes_match(frame, r),
            valid_udp_frame(frame) ⟹ udp_port_bytes_match(frame, r),
            valid_ipv4_frame(frame) ⟹ ipv4_length_bytes_match(frame, r),
    {
        let header: EthernetRepr = EthernetRepr::parse(frame: slice_subrange(slice:
        ↳ frame, i: 0, j: EthernetRepr::SIZE))?;
```

```
pub open spec fn valid_ipv4_frame(frame: &[u8]) → bool
{
    net::frame_dst_addr_valid(bytes: frame@)
    && net::frame_is_wellformed_eth2(frame)
    && net::frame_ipv4(frame)
    && net::wellformed_ipv4_frame(frame@)
}

pub open spec fn valid_udp_frame(frame: &[u8]) → bool
{
    valid_ipv4_frame(frame) && net::ipv4_is_udp(frame: frame@)
}

pub open spec fn res_is_udp(r: Option<EthFrame>) → bool
{
    r.is_some() && r.unwrap().eth_type is Ipv4 &&
        r.unwrap().eth_type→Ipv4_0.protocol is Udp
}

pub open spec fn udp_port_bytes_match(frame: &[u8], r: Option<EthFrame>)
{
    net::spec_u16_from_be_bytes(frame@.subrange(36, 38)) ==
        r.unwrap().eth_type→Ipv4_0.protocol→Udp_0.dst_port
}
```

```
pub struct EthFrame {
    pub header: EthernetRepr,
    pub eth_type: PacketType,
}

pub enum PacketType {
    Arp(Arp),
    Ipv4(Ipv4Packet),
    Ipv6,
}

pub struct Ipv4Packet {
    pub header: Ipv4Repr,
    pub protocol: Ipv4ProtoPacket,
}

pub enum Ipv4ProtoPacket {
    Tcp(TcpRepr),
    Udp(UdpRepr),
    HopByHop,
    Icmp,
    Igmp,
    Ipv6Route,
    Ipv6Frag,
    Icmpv6,
    Ipv6NoNxt,
    Ipv6Opts,
}
```

```
impl EthernetRepr {
    pub const SIZE: usize = 14;
    /// Parse an Ethernet II frame and return a high-level representation.
    pub fn parse(frame: &[u8]) -> (r: Option<EthernetRepr>)
        requires
            frame@.len() >= Self::SIZE,
        ensures
            valid_arp_frame(frame) == (r.is_some() && r.unwrap().ethertype is Arp),
            valid_ipv4_frame(frame) == (r.is_some() && r.unwrap().ethertype is Ipv4),
            valid_ipv6_frame(frame) == (r.is_some() && r.unwrap().ethertype is Ipv6),
    {
        let dst_addr = Address::from_bytes(slice_subrange(frame, 0, 6));
        if dst_addr.is_empty() {
            return None;
        }
        let src_addr = Address::from_bytes(slice_subrange(frame, 6, 12));
        let ethertype = EtherType::from_bytes(slice_subrange(frame,12,14))?;

        Some(EthernetRepr {
            src_addr,
            dst_addr,
            ethertype,
        })
    }
```

```rust
pub enum EtherType {
    Ipv4 = 0x0800,
    Arp = 0x0806,
    Ipv6 = 0x86DD,
}

impl EtherType {
    pub fn from_bytes(bytes: &[u8]) -> (r: Option<EtherType>)
        requires
            bytes@.len() == 2,
        ensures
            frame_arp_subrange(bytes@) == (r.is_some() && r.unwrap() is Arp),
            frame_ipv4_subrange(bytes@) == (r.is_some() && r.unwrap() is Ipv4),
            frame_ipv6_subrange(bytes@) == (r.is_some() && r.unwrap() is Ipv6),
        {
        let raw = u16_from_be_bytes(bytes);
        EtherType::try_from(raw).ok()
    }
}
```

```rust
pub open spec fn frame_ipv4_subrange(frame: Seq<u8>) -> bool
{
    frame =~= seq![8,0]
}
```

```
impl TryFrom<u16> for EtherType {
    type Error = ();

    fn try_from(value: u16) -> (r: Result<Self, Self::Error>)
        ensures
            match value {
                0x0800 ⇒ r.is_ok() && r.unwrap() is Ipv4,
                0x0806 ⇒ r.is_ok() && r.unwrap() is Arp,
                0x86DD ⇒ r.is_ok() && r.unwrap() is Ipv6,
                _ ⇒ r.is_err(),
            }
    {
        match value {
            0x0800 ⇒ Ok(EtherType::Ipv4),
            0x0806 ⇒ Ok(EtherType::Arp),
            0x86DD ⇒ Ok(EtherType::Ipv6),
            _ ⇒ Err(()),
        }
    }
}
```

RxFirewall

```
if let Some(frame) = api.get_EthernetFramesRxIn0() {
    if let Some(eth) = Self::get_frame_packet(&frame) {
        if can_send_packet(&eth.eth_type) {
            api.put_EthernetFramesRxOut0(frame);
        }
    }
}
```

```
pub fn get_frame_packet(frame: &SW::RawEthernetMessage) -> (r: Option<EthFrame>)
    requires
        frame@.len() == SW_RawEthernetMessage_DIM_0
    ensures
        Self::valid_arp(*frame) == firewall_core::res_is_arp(r),
        Self::valid_ipv4_udp(*frame) == firewall_core::res_is_udp(r),
        Self::valid_ipv4_tcp(*frame) == firewall_core::res_is_tcp(r),
        Self::valid_ipv4_tcp(*frame) ==> firewall_core::tcp_port_bytes_match(frame, r),
        Self::valid_ipv4_udp(*frame) ==> firewall_core::udp_port_bytes_match(frame, r),
{
    let eth = EthFrame::parse(frame);
    if eth.is_none() {
        info("Malformed packet. Throw it away.")
    }
    eth
}
```

```
pub open spec fn packet_is_whitelisted_udp(packet: &PacketType) -> bool
{
    packet is Ipv4 &&
        packet->Ipv4_0.protocol is Udp &&
        seL4_RxFirewall_RxFirewall::ipv4_udp_on_allowed_port_quant(packet->Ipv4_0.protocol->Udp_0.dst_port)
}

fn can_send_packet(packet: &PacketType) -> (r: bool)
    requires
        config::tcp::ALLOWED_PORTS =~= seL4_RxFirewall_RxFirewall::TCP_ALLOWED_PORTS(),
        config::udp::ALLOWED_PORTS =~= seL4_RxFirewall_RxFirewall::UDP_ALLOWED_PORTS(),
    ensures
        ((packet is Arp) ||
            packet_is_whitelisted_tcp(packet) ||
            packet_is_whitelisted_udp(packet)
        ) == (r == true),
{
    match packet {
        PacketType::Arp(_) => true,
        PacketType::Ipv4(ip) => match &ip.protocol {
            Ipv4ProtoPacket::Udp(udp) => {
                let allowed = udp_port_allowed(udp.dst_port);
                if !allowed {
                    info("UDP packet filtered out");
                }
                allowed
            }
            Ipv4ProtoPacket::Tcp(tcp) => {
```

```
fn port_allowed(allowed_ports: &[u16], port: u16) -> (r: bool)
    ensures
        r == allowed_ports@.contains(port),
{
    let mut i: usize = 0;
    while i < allowed_ports.len()
        invariant
            0 <= i <= allowed_ports@.len(),
            forall |j| 0 <= j < i ==> allowed_ports@[j] != port,
        decreases
            allowed_ports@.len() - i
    {
        if allowed_ports[i] == port {
            return true;
        }
        i += 1;
    }
    false
}

fn udp_port_allowed(port: u16) -> (r: bool)
    ensures
        r == config::udp::ALLOWED_PORTS@.contains(port),
{
    port_allowed(&config::udp::ALLOWED_PORTS, port)
}
```

TxFirewall

```
if let Some(frame) = api.get_EthernetFramesTxIn0() {
    if let Some(eth) = Self::get_frame_packet(&frame) {
        if let Some(size) = can_send_packet(&eth.eth_type) {
            let out = SW::SizedEthernetMessage_Impl {
                sz: size,
                message: frame,
            };
            api.put_EthernetFramesTxOut0(out);
        }
    }
}
```

```
fn get_frame_packet(frame: &SW::RawEthernetMessage) → (r: Option<EthFrame>)
    requires
        frame@.len() == SW_RawEthernetMessage_DIM_0
    ensures
        Self::valid_arp(*frame) == firewall_core::res_is_arp(r),
        Self::valid_ipv4(*frame) == firewall_core::res_is_ipv4(r),
        Self::valid_ipv4(*frame) ⟹ firewall_core::ipv4_length_bytes_match(frame, r),
{
    let eth = EthFrame::parse(frame);
    if eth.is_none() {
        info("Malformed packet. Throw it away.")
    }
    eth
}
```

```
fn can_send_packet(packet: &PacketType) → (r: Option<u16>)
    requires
        (packet is Ipv4) ⟹ (firewall_core::ipv4_valid_length(*packet))
    ensures
        (packet is Arp || packet is Ipv4) == r.is_some(),
        packet is Arp ⟹ (r == Some(64u16)),
        packet is Ipv4 ⟹ (r == Some((packet→Ipv4_0.header.length + EthernetRepr::SIZE) as u16)),
{
    match packet {
        PacketType::Arp(_) ⇒ Some(64u16),
        PacketType::Ipv4(ip) ⇒ Some(ip.header.length + EthernetRepr::SIZE as u16),
        PacketType::Ipv6 ⇒ {
            info("IPv6 packet: Throw it away.");
            None
        }
    }
}
```

Verification Results

- Due to significant testing, verification only revealed one new bug.
- A potential overflow was found
  - Packet size reported for an IPv4 packet was unbounded.
  - This resulted in a change to requirements, specification, and code to limit this size.
- I spent 94 hours on the verification effort. 6 of those were paired programming with a formal methods expert.
  - Also, an initial, unrefined version of the GUMBO contracts and the ideas for the verification approach were provided by KSU
    - These were invaluable to getting started
  - I had no formal methods experience prior to this
- Some code was made more complicated because Verus did not already supply specification/proofs for some standard libraries

Property-Based Testing

An alternative to handwritten unit tests

- HAMR generates executable code from the specification for property-based testing
- 12 hours of effort to write input array generation strategies
  - This effort was probably so low because I already went through verification effort
    - The code and specification are already trusted
  - No function mocking/stubbing required!
- Usually gives 100% code coverage
  - Because it is randomized, I don't always get 100%
  - Could tweak number of tests or weightings for more consistency
- All tests passing != all contracts are being met
  - All tests passing + 100% code coverage != all contracts are being met
    - Some improvements still needed
- A method to test out the specification and code
- Can be used on its own or with verification to increase confidence

Input Strategies

```rust
fn ipv4_strategy() -> impl Strategy<Value = Vec<u8>> {
    ipv4_protocol_strategy().prop_flat_map(|proto| {
        let proto_packet = match proto {
            // Tcp
            0x06 => tcp_strategy().boxed(),
            // Udp
            0x11 => udp_strategy().boxed(),
            _ => default_packet_strategy().boxed(),
        };
        (
            ipv4_length_strategy(),
            proto_packet,
            proptest::collection::vec(any::<u8>(), 40),
        )
            .prop_map(move |(length, proto_pack, mut v)| {
                copy_u16(&mut v[2..=3], length);
                v[9] = proto;
                v.splice(20..20 + proto_pack.len(), proto_pack);
                v
            })
    })
}
```

```rust
fn ipv4_protocol_strategy() -> impl Strategy<Value = u8> {
    prop_oneof![
        4 => Just(0x00), // HopByHop
        4 => Just(0x01), // Icmp
        4 => Just(0x02), // Igmp
        10 => Just(0x06), // Tcp
        10 => Just(0x11), // Udp
        4 => Just(0x2b), // Ipv6Route
        4 => Just(0x2c), // Ipv6Frag
        4 => Just(0x3a), // Icmpv6
        4 => Just(0x3b), // Ipv6NoNxt
        4 => Just(0x3c), // Ipv6Opts
        1 => any::<u8>(),
    ]
}

fn ipv4_length_strategy() -> impl Strategy<Value = u16> {
    prop_oneof![
        40 => (0u16..=9000),
        1 => (9001u16..),
    ]
}
```

```rust
fn udp_port_strategy() -> impl Strategy<Value = u16> {
    prop_oneof![
      1 => Just(68),
      4 => any::<u16>(),
    ]
}

fn udp_strategy() -> impl Strategy<Value = Vec<u8>> {
    (
        udp_port_strategy(),
        proptest::collection::vec(any::<u8>(), 20),
    )
        .prop_map(|(port, mut v)| {
            copy_u16(&mut v[2..=3], port);
            v
        })
}
```

Conclusion

- Layering security strategies increases confidence and can make subsequent strategies easier
- The use of the INSPECTA tools, such as HAMR and Verus, make formal methods approachable to systems engineers and can improve development time
- A modelling workflow allows for:
  - Formalization of requirements to architectural contracts
  - Generation of infrastructure/architectural code, verification contracts, and test infrastructure
  - Nice separation between architecture responsibilities and application responsibilities
  - Better reconfiguration/re-use of components
    - Can be seen with how easy it was to implement 2 different firewall policies
  - Obvious traceability from requirements → contracts → verification/tests
- Would like to continue to apply this approach to the driver and more applications

Questions?

See the opened code/architecture/requirements/specification

- Requirements:
  - https://github.com/loonwerks/INSPECTA-models/blob/dw/firewalls-verified/open-platform-models/isolate-ethernet-simple/requirements/Inspecta-HLRs.pdf
- Model:
  - https://github.com/loonwerks/INSPECTA-models/blob/dw/firewalls-verified/open-platform-models/isolate-ethernet-simple/aadl/SW.aadl
- Ethernet Frame Parser Library:
  - https://github.com/loonwerks/INSPECTA-models/tree/dw/firewalls-verified/open-platform-models/isolate-ethernet-simple/microkit/crates/firewall_core
- Rx Firewall:
  - https://github.com/loonwerks/INSPECTA-models/tree/dw/firewalls-verified/open-platform-models/isolate-ethernet-simple/microkit/crates/seL4_RxFirewall_RxFirewall
- Tx Firewall:
  - https://github.com/loonwerks/INSPECTA-models/tree/dw/firewalls-verified/open-platform-models/isolate-ethernet-simple/microkit/crates/seL4_TxFirewall_TxFirewall