Design and Verification of Software Middleboxes using Gravel

Kaiyuan Zhang[1], Danyang Zhuo[1], Aditya Akella[2], Arvind Krishnamurthy[1], and Xi Wang[1]

[1]University of Washington
[2]University of Wisconsin-Madison

Abstract

Middleboxes play a critical role in modern networks, yet providing bug-free middleboxes implementations remains challenging. This paper presents Gravel, a framework for building formally verified middleboxes. Gravel allows developers to specify high-level middlebox-specific properties, as opposed to generic robustness and liveness conditions, and checks correctness in low-level implementations without manual proofs. Further, Gravel reduces the proof burden by breaking down middlebox functionalities into smaller packet processing elements that can be independently verified. We build a NAT, a load balancer, and a firewall with Gravel and then verify their correctness. Our evaluation shows that middleboxes built with Gravel avoid bugs similar to those found in today's middleboxes and achieve similar performance to their unverified counterparts.

1 Introduction

Middleboxes (e.g., NATs, firewalls, and load balancers) provide essential services, improving security and performance in modern networks, yet building functionally correct middleboxes remains a difficult challenge. Critical bugs have routinely been found in today's middlebox implementations. Many of these bugs [4–8] directly lead to system failure or information leaks. Worse still, some of these bugs can be exploited simply by malformed packets, exposing severe security vulnerabilities.

There has been significant advancement in middlebox verification [9, 15, 24, 26]. These efforts work with concrete middlebox implementations and focus on guaranteeing a wide range of *low-level* properties (e.g., free of crashes, memory safety, and bounded execution time); some of them take additional steps to prove functional-correctness of specific middleboxes, such as VigNAT [26]. However, they currently still have two limitations. First, they prove against pseudo code-like *low-level* specifications, but not *high-level* middlebox properties (e.g., a NAT rejects unsolicited external connections). Second, they focus on single type of middlebox, and it is unclear how to generalize their techniques to other types of middleboxes with low programming and verification effort.

This paper presents Gravel, a framework for designing, implementing, and verifying middleboxes. Gravel provides developers programming interfaces to specify high-level middlebox properties and verify them against a low-level C implementation. In addition, it aims to minimize developers' burden of writing manual proofs, ideally in a push-button fashion of verification.

A key challenge of Gravel is to support verification of high-level, middlebox-specific specifications. High-level specifications are important because they are concise and directly related to desirable middlebox properties (e.g., NAT's hairpinning [22] and load balancer's connection persistency [10]). However, they are difficult to prove because of the semantic gap between the high-level specifications and low-level implementations. Gravel provides a specification programming interface to encode middlebox-specific properties. Instead of supporting arbitrary high-level properties, Gravel restricts developers to specify properties only on a symbolic packet trace over the middlebox. This style of specification enables developers to encode high-level properties concisely and allows the specification to directly relate to the behavior of the middlebox implementations.

A second challenge is to reduce the verification effort. Middleboxes are complex software, and directly reasoning about the entire low-level C implementation is not feasible. To modularize verification, Gravel enforces a restricted programming model, similar to Click [14]. A middlebox built with Gravel consists of packet processing elements connected in a directed graph. Each element is written in C. Also, the state of each element can be expressed only in fix-sized variables or in maps. The low-level state machine specifications and implementations are provided on a per-element basis. In this way, the verification that the implementation meets the low-level state machine specification can be done on each element independently. After each element is veri-

fied, Gravel verifies the high-level specification by performing symbolic execution on the graph. While Gravel's programming model may seems restrictive, languages such as p4 [25] show that these primitives (states as maps, directed graph of elements) can be used to build a variety of powerful middleboxes [16, 20, 25].

We evaluate this programming model's flexibility by building three middleboxes: a NAT, a load balancer, and a stateful firewall. We verify their correctness against high-level specifications from RFCs and other desirable middlebox properties, for example, load balancer's connection persistency. Our evaluation shows that Gravel can avoid bugs similar to those found in existing unverified middleboxes. Our evaluation also shows that the programming and verification effort is small. Finally, middleboxes built with Gravel achieve similar performance to their unverified counterparts.

In summary, this paper makes the following contributions:

- A specification programming interface for encoding high-level middlebox properties.
- A programming framework for middleboxes that enables automated verification. The programming framework segments a middlebox into multiple packet processing elements. Each element uses a restricted set of primitives (e.g., fix-sized variables, maps).
- Three separate middlebox implementations formally verified with Gravel, including a load balancer, a stateful firewall, and a NAT.

2 Gravel Programming Framework

Gravel is a programming framework for designing, implementing, and formally verifying software middleboxes. It aims to verify a high-level, abstract middlebox specification (e.g., NAT's hairpinning) against a low-level, concrete C implementation of the middlebox. To do so, it provides both a generic interface for encoding high-level middlebox properties and for writing the C implementation. Gravel chooses to employ a Click-like [14] programming style for middleboxes. As we will show later, a Click-like decomposition of middlebox functionalities into multiple packet-processing elements enables efficient and automated verification.

Figure 1 shows the workflow of Gravel. Gravel expects three inputs: a high-level middlebox specification of several properties, a directed graph of elements with their state-machine specifications, and a set of element implementations. The middlebox implementation is essentially the directed graph and the elements' implementations. If there is any bug in the specification of an element or in the implementation, Gravel outputs a counterexample. Once the verification completes, Gravel emits an executable that contains the middlebox implementation. Gravel provides support for running the executable as a DPDK application.

This section describes how to construct a simple Layer-3 load balancer, ToyLB. ToyLB receives packets on its incoming interface and forward them to a pool of servers in a round-robin fashion. It steers traffic by rewriting the destination IP address on the packet. At a high-level, ToyLB represents popular Layer-3 load balancer designs used by large cloud providers [10, 11]. We first write the high-level specifications of ToyLB, and then decompose ToyLB into a directed graph of elements. Finally, we write the implementation of the elements used in the ToyLB.

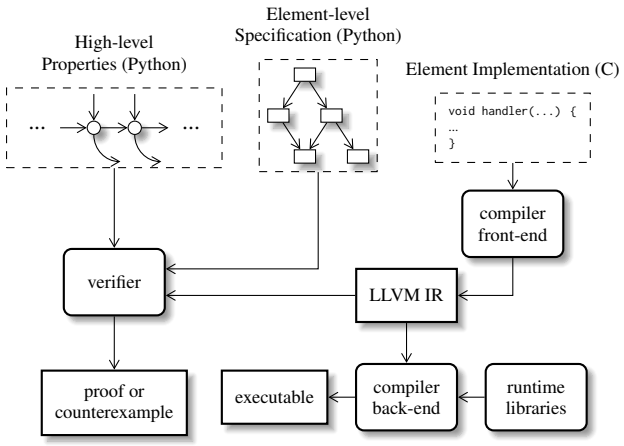


Figure 1: Development Flow of Gravel. Top three boxes denotes inputs from middlebox developers; rounded boxes denotes compilers and verifiers of Gravel; rectangular boxes denotes intermediate and final outputs.

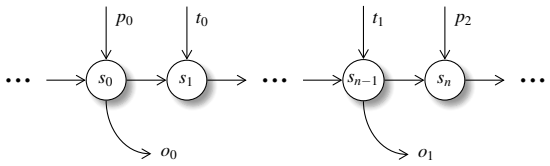
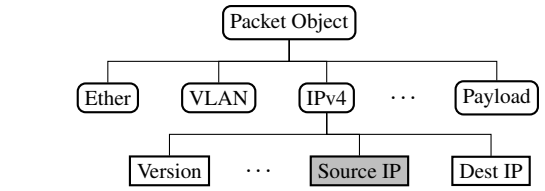


Figure 2: A sample event sequence for a middlebox. s means the states of the middlebox. p means incoming packets. o means outgoing packets. t means time events.

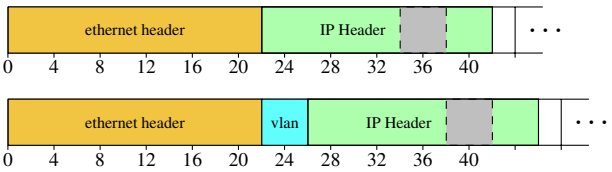
2.1 High-level Specifications

In the high-level specification, Gravel models the execution of a middlebox as a state machine. As shown in Figure 2, state transition can occur in response to external events such as incoming network packets or passage of time. For each state transition, it may also send packets out. This models the packet rewriting, forwarding and broadcasting behavior of a middlebox. In Gravel, the passage of time is modeled as a external event. Gravel's runtime triggers the time event periodically. The time event can be used to implement garbage collection for middlebox states.

For each state transition, Gravel follows a "run-to-completion" execution model: When processing a incoming packet or event, the application always runs until the packet



(a) Field access in abstract packet



(b) Field access in packet buffer

Figure 3: Accessing packet header fields. The gray parts in Figure 3b are the source IP addresses.

is fully processed by the middlebox, before handling any other incoming packet or event.[1]

To encode high-level middlebox properties, Gravel provides a set of programming interface, embedded in Python. Developers can use our specification programming interface to describe the required behavior on the symbolic event sequence (Figure 2).

In the high-level specification, Gravel uses a key-value store abstraction to represent packet header, where the keys are the name of the header fields. This abstraction make the specification concise and hides the implementation details that are less related to high-level properties (i.e., the position of source IP addresses in the packet header). Figure 3 shows the difference between the key-value store abstraction and the actual packet data layout in memory. Accessing the source IP address of an IP packet may not be as simple as a single memory read, because optional headers such as VLAN changes the position of source IP address.

Gravel provides three kinds of core interfaces in its high-level specification: (1) a set of `sym_*` functions that allows developers to create symbolic values of different types such as IP address, packet, or middlebox state; (2) middlebox's event handling functions, like `handle_packet(state, pkt)`, `handle_time(state, timestamp)`, that takes as input the current state of the middlebox and the incoming packet / time event, and returns the output from the middlebox as well as the new states after the state transition; and (3) the `verify(formula)` function call that first encodes the given logical formula in SMT and invokes the SMT solver to check if `formula` is always true. Besides that, Gravel also provides some helper functions for developers to encode high-level middlebox properties. We also include them here in the table since we use them in the next set of examples.

[1] Gravel assumes "run-to-completion" in verification, however, the Gravel's runtime might not need to enforce it if the runtime provides a transactional guarantee for packet-processing. (See §6.)

To make this concrete, we next describe how to encode two high-level properties of ToyLB using this specification programming interface. We are going to encode two load balancer properties: (1) liveness (2) connection persistency.

Let's first take a look at the liveness guarantee:

PROPERTY 1 (ToyLB liveness). For every TCP packet received, ToyLB always produces an encapsulated packet.

In Gravel, this can be specified as:

```
def toylb_liveness():
    # create symbolic packet and symbolic ToyLB state
    p, s0 = sym_pkt(), sym_state()
    # get the output packet after process packet p
    o, s1 = handle_packet(s0, p)
    verify(Implies(is_tcp(p),
                   And(Not(is_none(o)))))
```

In the liveness formulation, first we construct a symbolic packet `p` and the symbolic state of the middlebox `s0`. Then, we let the middlebox with state `s0` process the packet `p` by invoking the `handle_packet` function. After that, the state of the middlebox changes to `s1`, and the output from the middlebox is `o`. (Here `o` can be `None`, meaning the middlebox drops the packet. Function `is_none` can be used to test whether an output is `None`) Finally, the code snippet asks Gravel to verify that if the incoming packet is a TCP packet, then `o` exists (i.e., there is a outgoing packet).

Note here the formulation of the liveness property is an abstract middlebox behavior. The formulation does not say anything about what the state of the middlebox looks like. We don't even formulate the set of data structures used in the middlebox. This is indeed the benefit of using high-level specification in middlebox verification. These formulations are concise and are directly related to a desirable middlebox property.

Now, we move to a more complex load balancer property—connection persistency. This property is crucial to correct functioning of a load balancer as it ensures that packets from the same TCP connection are always forwarded to the same backend server.

PROPERTY 2 (ToyLB persistency). If ToyLB forwards a TCP packet to a backend b at time t, subsequent packets of the same TCP connection received by ToyLB before time $t + WINDOW$, where $WINDOW$ is a pre-defined constant, will also be forwarded to b.

Formulation of Property 2 is more complex than the liveness property because, as Property 2 shows, the formulation requires a condition (forwarding packet of certain TCP connection to b) to hold over all possible events sequence between time t and time $t + WINDOW$. This means we cannot formulate connection persistency as a trace containing a single event. Rather, we need to use induction to verify the property holds on an event trace of unbounded length.

Function name	Description
Core Interfaces:	
`sym_*() → SymValT`	Create a symbolic value of corresponding type
`handle_packet(s, pkt, in_port) → o1, ···, on, ns`	Handle the packet and returns the outputs and new state
`handle_time(s, timestamp) → o1, ···, on, ns`	Handle time event, return value is same as `handle_packet`
`verify(formula)`	Encode given formula and verify that a formula always holds
Helper Functions:	
`is_none(output) → Bool`	Check if an output is `None`
`payload_eq(p1, p2) → Bool`	Determine if two packets have the same payload
`from_same_flow(p1, p2) → Bool`	Determine if two packets are from the same TCP connection
`is_tcp(pkt) → Bool`	Check if a packet is TCP packet

Table 1: Gravel's specification programming interface.

Gravel specifies Property 2 as an inductive invariant. First, we formulate the packet forwarding condition that should be hold in the time window:

```
def steer_to(state, pkt, dst_ip, t):
    o0, s_n = handle_time(state, t)
    o1, s_n2 = handle_packet(s_n, pkt)
    return And(Not(is_none(o1)),
               o1.ip4.dst == dst_ip,
               payload_eq(o1, pkt))
```

The `steer_to` function defined above determines whether a packet received at time `t` will be forwarded to backend server with address `dst_ip`. The code snippet first lets the middlebox handle a time event with timestamp `t`, followed by the handling of `pkt`. We known that the packet is forwarded to `dst_ip` if the output from the packet processing is not `None` and that the resulting packet's destination address is `dst_ip`.

Then, for the base case of induction, we prove that once ToyLB forwards a packet of certain TCP connection to a backend, consecutive packets from the same connection will be forwarded to same backend if the difference in their arrival time is less than *WINDOW*.

```
def base_case():
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    o, s1 = handle_packet(s0, p0)
    dst_ip, t0 = sym_ip(), s0.curr_time()
    t = sym_time()
    ddl = t0 + WINDOW
    verify(Implies(And(Not(is_none(o)),
                       o.ip4.dst == dst_ip,
                       from_same_flow(p0, p1)),
                ForAll([t],
                       Implies(t <= ddl,
                           steer_to(s1, p1, dst_ip, t)))))
```

Similar to the formulation of liveness property, the above code snippet first creates two symbolic packets and a symbolic middlebox state, then invokes `handle_packet` to obtain the output packet as well as the new state after packet processing. After that, the code asks the verifier to prove that if `p0` is forwarded to `dst_ip`, then at anytime before the expiration time `ddl`, packet `p1` is also forwarded to `dst_ip` if it is from the same TCP connection.

After proving the base case, we also need to prove the two inductive cases showing that processing an additional event (e.g., processing a packet from a different connection, processing a time event) does not change the forwarding behavior.

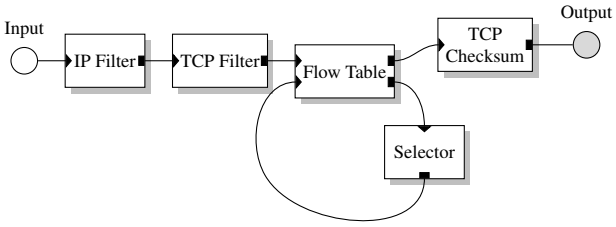


Figure 4: Breakdown of ToyLB's functionalities into 5 packet-processing elements. IP filter and TCP filter together filter out non-TCP traffic. FlowTable stores the mapping from flow signature to the destination server. Selector chooses destination server for new flows. White and shaded circles denote the source and sink of packets respectively.

```
def step_packet():
    dst_ip, p0, p1 = sym_ip(), sym_pkt(), sym_pkt()
    s0, t0, p_other = sym_state(), sym_time(), sym_pkt()
    o, s1 = handle_packet(s0, p_other)
    verify(Implies(And(steer_to(s0, p0, dst_ip, t0),
                       from_same_flow(p0, p1)),
                   steer_to(s1, p1, dst_ip, t0)))

def step_time():
    dst_ip, p0, p1 = sym_ip(), sym_pkt(), sym_pkt()
    s0, t0, t1 = sym_state(), sym_time(), sym_time()
    _, s1 = handle_time(s0, t1)
    verify(Implies(And(steer_to(s0, p0, dst_ip, t0),
                       t1 < t0,
                       from_same_flow(p0, p1)),
                   steer_to(s1, p1, dst_ip, t0)))
```

The two inductive cases proves that the invariant condition (`steer_to(...)`) holds when processing packets (`p_other`) or handling time events where the timestamp is before the expiration time.

2.2 Element-level Specifications

Other than the high-level specification, Gravel also requires the developers to decompose the middlebox into a directed

graph, similar to Click modular router [14]. In Gravel, the directed graph consists of multiple "elements", each representing an independent packet-processing or event-handling entity. Elements in Gravel can have their own private state that are exclusively managed by themselves. Each element has a set of input and output ports to receive packets from or send packets to. The graph is constructed by connecting the an output port of one element with an input port of another. Elements can then send packets to others through its output ports. Each element in Gravel has a packet handler. When a packet arrives at any of the element's input ports, Gravel invokes its packet handler to perform operations such as rewriting the packet and/or updating private states of the element. Moreover, element may also have event handler for other kind of events such as time events. Besides normal elements that performs packet processing, the directed graph also includes multiple "source" and "sink" that represents the entries where packets entering or leaving the graph. In the actual implementation, these sources and sinks represents receiving and sending packets using different physical network interfaces.

As a concrete example, ToyLB's functionality can be decomposed into 5 elements, as shown in Figure 4. When there is an incoming packet, it first goes through two filter elements (the TCP FILTER and the IP FILTER). These two filters will discard any packet that is not a TCP packet. Then the FLOWTABLE element checks whether the packet belongs to a TCP flow that has been seen by ToyLB earlier. If so, FLOWTABLE encapsulates the packet with the corresponding backend server's IP address stored in the FlowTable and send the packet to the destination server. Otherwise, the FLOWTABLE consults a SELECTOR element to decide which destination server should the new connection bind to. After the SELECTOR decides which backend server to forward the packet to, the selector notifies the FLOWTABLE of the decision. The FLOWTABLE stores the decision into its internal state and also rewrites the destination address of the packet into the destination server. For simplicity, low level functionalities such as ARP lookup are omitted for ToyLB.

Besides the directed graph, Gravel also requires the developer to give specifications of each individual element. The element-level specification in Gravel consists of two parts: the definition of abstract states that will be used by the element during execution, and a set of event handling behaviors in response to events such as time or incoming packets.

Element States The specification of a Gravel element starts with declaration of states kept by the element. To ensure efficient encoding with SMT, Gravel requires the state of elements to be finite. More specifically, elements' state in Gravel may contain: (1) a fixed size variable such as bitvector (fix-size integer); (2) a map from one finite set to another (i.e., map from IP address space to 64-bit integer). For example, in ToyLB, the state of `FlowTable` is defined as:

```
class FlowTable(Element):
    num_in_ports = 2
    num_out_ports = 2

    decisions = Map([AddrT, PortT, AddrT, PortT], AddrT)
    timestamps = Map([AddrT, PortT, AddrT, PortT], TimeT)
    curr_time = TimeT
    ...
```

The code snippet above defines three components of `FlowTable`'s state:

- `decisions` maps from a TCP connection to a backend server address. `FlowTable` identifies a TCP connection by the tuple of source and destination address and port number. This map is used to store results from consulting the `Selector` element.
- `timestamps` stores the latest time receiving a packet for each TCP flow stored in `decision`.
- `curr_time` stores the current time.

Here the types such as `AddrT` and `TimeT` are pre-defined integers of different bitwidth. Besides the states, the code also informs Gravel how many input and output ports does `FlowTable` element have by assigning to `num_in_ports` and `num_out_ports`.

Event Handlers As mentioned above, Gravel requires each element to have a handler function for incoming packets from its input ports. This packet handler needs to be specified in the element-level specification. The specification of packet handler describes operations the element should perform when handling packets. Similarly, in element-level specification, developers can also declare handlers for other events. For example, the `FlowTable` element in ToyLB has two event handlers, one for incoming packets, and the other for time events. In Gravel, the two event handlers are defined as functions with the following signature:

```
flowtable_handle_packet(old, pkt, in_port) → action_list
flowtable_handle_time(old, timestamp) → action_list
```

The return value of each event handler(`action_list`) is a list of *condition-action pairs*. Each entry in the list describes the action one element should take under certain conditions. In the python code, developers can write:

```
If(cond, { port_i : pkt_i }, new_state)
```

to denote an action that send `pkt_i` to output port `port_i` while at the same time update the element state to `new_state`. This action will be taken when condition `cond` holds. To make it concrete, let's take a look at the packet handler of `FlowTable`:

```
def flowtable_process_packet(self, old, p, in_port):
    flow = p.ip4.src, p.tcp.src, p.ip4.dst, p.tcp.dst
    # the case when flowtable has record of the flow
    known_flow = And(
        # packet is received from the network
        in_port == IN_TCP_FILTER,
        # flowtable has record of the flow
        flow in old.decisions)
```

```
    # construct the encapsulated packet
    fwd_pkt = p.copy
    fwd_pkt.ip4.dst = old.decisions[flow]
    # update the timestamp of the flow with current time
    after_fwd = old.copy()
    after_fwd.timestamps[flow] = old.curr_time

    known_flow_action =
        If(known_flow,
           {PORT_TO_EXT: fwd_pkt}, after_fwd)

    # the case when flowtable does not know the flow
    consult_selector = And(
        in_port == INPORT_NET,
        Not(flow in old.decisions))
    unknown_flow_action =
        If(consult_selector, {PORT_TO_SELECTOR: p}, old)

    # packet from the Selector
    register_new_flow = in_port == IN_SELECTOR
    # extract the new_flow
    new_flow = p.inner_ip.src, p.tcp.src, \
               p.inner_ip.dst, p.tcp.dst
    # add the record of the new_flow to FlowTable
    after_register = old.copy()
    after_register.decisions[new_flow] = p.ip4.dst
    after_register.timestamps[new_flow] = old.curr_time
    register_action =
        If(register_new_flow, {PORT_TO_EXT: p},
           after_register)

    return [known_flow_action,
            unknown_flow_action,
            register_action]
```

In the code snippet above, when receiving a incoming packet, `FlowTable` can perform one of the following actions:

- `known_flow_action`: When the packet is from input port `IN_TCP_FILTER`, and the `decision` map contains record from the connection of the packet, `FlowTable` directly rewrites the destination address and send the packet to the port connected to `TCP Checksum` element (`PORT_TO_EXT`).
- `unknown_flow_action`: If the `FlowTable` do not have the record, it forwards the packet to `Selector` element through `PORT_TO_SELECTOR`.
- `register_action`: When receiving packet from `Selector`, `FlowTable` records the destination decided by `Selector` and forward the packet to `PORT_TO_EXT`.

Similarly, `FlowTable`'s behavior in response to time changes is also specified as *condition-action*s:

```
def flowtable_process_time(self, s, time):
    new = s.copy()
    # update the "curr_time" state
    new.curr_time = time
    # records with older timestamps should expire
    def should_expire(k, v):
        return And(s.timestamps.has_key(k),
                   time >= WINDOW + s.timestamps[k])

    new.decisions = new.decisions.filter(should_expire)
    new.timestamps = new.timestamps.filter(should_expire)
    return [If(True, {}, new)]
```

When `FlowTable` got notified by a time change, it updates its `curr_time` state to the given time value, Gravel offers a `filter` interface for its map object, which takes a predicate (`should_expire`) and deletes all the entries that satisfies the requirement. For `FlowTable`, it removes all the records that were recorded for longer than a constant `WINDOWN` value.

2.3 Element-level Implementations

With the middlebox decomposed into a directed graph of elements and each of the elements formally specified, Gravel then asks the developers for the implementations of elements. Gravel's implementation of an element consists of following components:

1. *Element state definition.* A C struct defining the states that are used by the element during packet and event processing.
2. *Event handlers.* A set of callback functions that will be invoked in response to different events such as receiving a packet from an input port or handling time events.

Element State. In Gravel, developers define the element state as a C struct. Like the specification of element, Gravel requires the element state size to be bounded. More specifically, for each field in the element state struct, Gravel requires the type of the field to be one of the following:

- Fix-sized integers such as `int` `long`.
- Fix-sized buffers.
- A pointer to a data structure that complies with a map interface.

As an example, in Gravel the state of FLOWTABLE element in ToyLB is defined as:

```
struct State {
    map_t *decisions;
    map_t *timestamps;
    uint64_t curr_time;
};
```

Event Handlers. For each of the event handlers specified in the element-level specification, Gravel requires a corresponding event handler in its C implementation. In Gravel's implementation, event handler is defined as a callback function that takes as arguments the current state of the element and the information of the incoming event. When implementing each of the event handlers, Gravel provides a set of programming APIs to help manipulating packet data and updating/accessing element states, as shown in Table 2. Other than those interfaces, Gravel requires the event handlers' implementation to be self-contained, which means function calls to external libraries not known by Gravel's verifier are not allowed.

Accessing Packet Header Fields. The implementation of accessing the packet field often contains a set of branches for different cases:

Function Name	Description
`map_get(map_t *, key_t *, val_t *) → int`	perform a lookup from the given map, if key is found, returns 0 and writes value to the given pointer, otherwise return -1
`map_set(map_t *, key_t *, val_t *)`	stores the given key-value pair into the given map
`push_pkt(pkt_t *, int)`	push the packet to the given output port using the port index, can only be called at most once on each output port
`copy_pkt(pkt_t *) → pkt_t *`	copy all contents from the given packet and returns the new one
`new_pkt(sizt_t) → pkt_t *`	create an empty packet of certain size, always success

Table 2: C programming interface offered by Gravel.

```
...
struct ethhdr *eth = (struct ethhdr *)pkt;
char *end_of_eth = pkt + sizeof(struct ethhdr);
struct iphdr *ip;
if (eth->h_proto == ETH_P_8021Q) {
        ip = (struct iphdr *)(end_of_eth
                              + sizeof(struct vlan_hdr));
} else {
        ip = (struct iphdr *)end_of_eth;
}
ip->saddr = <value>;
...
```

Remember, in both the high-level specifications and the element-level specifications, packet header fields with the key-value store abstraction. To make access to header fields consistency in the specification and the implementation, Gravel requires a packet format as input, similar to that in p4 [25]. Because Gravel formulates packets as a buffer, we allow arbitrary pointer arithmetic on the packet pointer to access header fields.

Middlebox Runtime. Gravel provides a runtime built on top of DPDK framework to deploy the middleboxes. Gravel maps each of the source and sink elements to NICs in DPDK. During the execution, Gravel repeatedly polls the NICs to retrieve incoming packets and sends them to corresponding source elements. Starting from that source element, Gravel performs a breadth-first traversal by invoking the packet handler for each element and propagates its output to downsteam elements in the directed graph until the packet reaches an sink element where it got sent out by invoking DPDK's `tx` functions. Gravel also reads the system clock and invokes the time handler of elements periodically.

3 Verifier

Gravel's verifier proves the correctness of middleboxes with two theorems:

THEOREM 1 (Graph Composition). The element-level specifications, when composed using the given graph of the elements, meet the requirement in the high-level specification of the middlebox.

THEOREM 2 (Element Refinement). Each element's C implementation is a refinement of that element's specification, that is: Every possible state transition and packet processing of the C implementation must have a equivalent counterpart in the element-level specification.

Theorem 1 verifies that the composition of element-level specifications meet the requirement in the high-level specifications. Theorem 2 verifies the C implementation of each element meets its element-level specification.

3.1 Graph Composition

Gravel proves Theorem 1 using two steps. First, Gravel symbolic executes an event sequence, as specified in the high-level specifications. Second, Gravel checks whether the high-level specifications hold on the result of the symbolic execution.

Gravel performs symbolic execution on the directed graph. Before the symbolic execution, Gravel creates a symbolic state of the entire middlebox, which is a composition of symbolic states of all elements in the middlebox. [2] Remember that the high-level specification describes required middlebox behavior on an event sequence. The goal here in the symbolic execution is to reproduce the event sequence symbolically. For example, if the high-level specification contains an incoming packet, Gravel generates a symbolic incoming packet at the source element of the directed graph. This incoming packet, when processed on the first element of the graph, can trigger events in other downstream elements. These events are symbolically executed as well. If the element-level specification contains a branch (e.g., depending on the packet header, a packet can be forward to one of the two downstream elements), Gravel performs symbolic execution in a breadth-first search manner.

After performing symbolic execution for each type of events, Gravel records the updated state of each element as well as the produced packet on each output elements. These information is used by Gravel as the return value of the `handle_*` functions in the high-level specification. Gravel then invokes the functions defined in the high-level specification. Once the `verify` function is invoked, Gravel encodes the high-level specifications into SMT form and inquire the Z3 SMT solver to see if they always hold.

[2] Developers may also supply a state mapping function that maps element states to states in the high-level specifications.

Loops in the Graph Gravel allows the dataflow model to contain loops in order to support bi-directional communications between elements, such as FLOWTABLE and SELECTOR in ToyLB(§2). However, loops may introduce non-halting execution when symbolic execute the dataflow. To address this issue, Gravel sets a limit on the number of elements the symbolic execution can traverse. When the symbolic execution hits this limit, Gravel simply raise an alert and fails the verification. For example, in ToyLB, the FLOWTABLE is hit at most twice: When FLOWTABLE can not find a record of certain packet, the packet is sent to SELECTOR, which will later sent the packet back to FLOWTABLE. But upon receiving packets from SELECTOR, FLOWTABLE simply records the selected backend server into its own records and never sends that packet back to SELECTOR. Thus the maximum number of elements traversed during the symbolic execution is 6 and developers can safely use 6 as the limit for ToyLB.

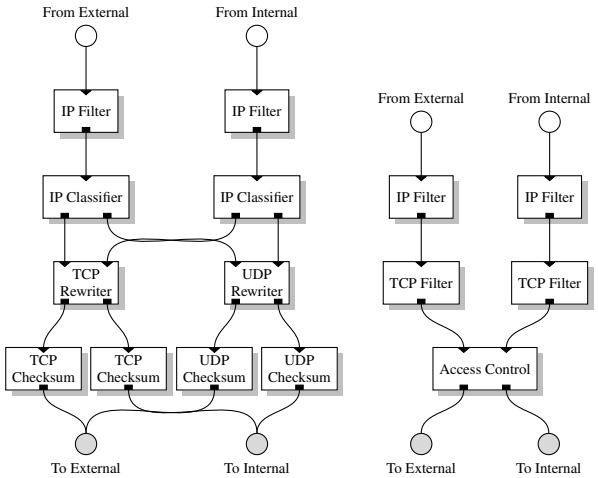
3.2 Element Refinement

The Graph Composition theorem verifies that the high-level specification holds when the low-level element specification is a faithful representation of the C implementation. The refinement theorem removes this assumption by proving that the C implementation is correct against the element specification. Gravel does the Element Refinement verification on a per-element basis. For each element, Gravel verifies that the event handlers in the C implementation perform the same state and packet modification as the corresponding handlers in the element specification. For the C implementation, Gravel uses the compiled LLVM bytecode (after compiler optimization) to mitigate impacts from compiler bugs.

Remember in the programming framework section (§2.3), state of each element is explicitly defined as a C struct. For element state accesses in the LLVM bytecode, Gravel extracts the offsets of fields in the C struct from the LLVM bytecode and use the offsets to determine which field the code is accessing. For fix-sized fields such as integers, Gravel directly map this access to the corresponding state in specification [3]. As for map data structure, Gravel use an abstract map model in SMT to hook in any invocation of map accessing functions in the C implementation and element specification.

As mentioned in §2.3, packet accesses in C implementation are memory operations instead of accessing a key-value store. Thus, we need to establish a relation between header field values in the implementation and in the specification. To do so, Gravel first computes offsets for all the header fields using the packet format specification defined in §2.3, note that these offsets are also symbolic values as they depend on the content of other packet fields. After that, Gravel extracts value of each header fields from the memory buffer of the packet. Each extracted value is then encoded into SMT formula and compared against fields from the abstract packet object using SMT solver. Gravel concludes that the packet object and memory buffer are equivalent when values of all fields are equivalent.

[3] Like the verification of the Graph Composition theorem, developers can also define a customized state mapping function.



(a) NAT (b) Stateful Firewall

Figure 5: Element graph for Applications built on Gravel

Similar to the symbolic execution used for verifying the Graph Composition theorem, Gravel also sets a limit on the maximum number of LLVM instructions that can be executed. Such limit can help Gravel detect unbounded loops from the C implementation and Gravel raises an exception when the symbolic execution exceeds the limit. At the end of the symbolic execution, Gravel gets a list of possible results. Each result contains the change of the element state and also the outgoing packet. Gravel checks whether every result meets the low-level specification.

3.3 Trusted Computing Base

The trusted computing base (TCB) of Gravel includes the verifier (including Theorem 1 and Theorem 2), the high-level specifications, the tools it depends on (i.e., the Python interpreter, the LLVM compiler framework, and the Z3 solver), and the runtime library used by the C implementation (i.e., the DPDK framework and Gravel's wrapper on top of it). Note that the specification of each element is not trusted.

4 Case Studies

We implemented three middlebox applications on top of Gravel and verified a set of high level properties for each of them. Gravel also automatically verified low-level properties such as memory safety and bounded execution.

4.1 Load Balancer

Besides the round-robin load balancer mentioned in §2, we also implemented a load balancer using Maglev's hashing algorithm [10]. Its element graph looks exactly the same as in Figure 4. The only difference is that the SELECTOR element uses consistent hashing. The load balancer steer packets by rewriting the destination IP address.

We verified connection persistency [10] for the load balancer. The goal of connection persistency is to make load balancing transparent to the clients.

PROPERTY 3 (Load Balance Persistence). For all packets p_1 and p_2 from connection c, if the load balancer steers p_1 to a backend server, then the load balancer steer p_2 to the same backend server before c is closed.

We also prove that the middlebox do not change the payload of all forwarded packets:

PROPERTY 4 (Payload Preservation). For any packet that is processed by the middlebox, the middlebox never modifies the payload of the packet.

We also verify the payload preservation property for the next two middleboxes.

4.2 Stateful Firewall

The stateful firewall built on top of Gravel keeps track of connection states between two subnetworks, the internal network and the external network. The firewall updates connection states when processing TCP control packets (e.g. *SYN*, *RST*, and *FIN* packets), and removes records for connections that are finished, disconnected.

Figure 5 shows the element graph of the stateful firewall. It reuses elements IP FILTER and TCP FILTER (both element-level specification, and element implementations) in the load balancer to filter out non-TCP traffic. It has an ACCESS CONTROL element that stores connection states and blocks unsolicited connections.

We prove that the stateful firewall can prevent packets of unsolicited connections [17] from getting into the internal network:

PROPERTY 5 (Firewall Blocks Unsolicited Connection). For all connection c, any packet p in c from the external network is not allowed until a *SYN* packet has been sent from the internal network.

PROPERTY 6 (Firewall Garbage-collects Records). For all connection c, a packet p in c from external network is blocked after the firewall sees a *FIN* or *RST* packet.

4.3 NAT

We built an NAT(Network Address Translation) middlebox. The NAT has similar functionalities as MazuNAT from Click modular router [14]. It is responsible for forwarding traffic between two network address spaces, the internal network and the external network. The NAT performs two types of packet rewriting:

1. For a packet whose destination address is the NAT, the NAT rewrites its destination IP address and port with the corresponding endpoint in the internal network.
2. For a packet from internal to external network, NAT assigns a external source IP address and port to the connection. NAT also need to keep track of assigned addresses and ports to guarantee persistent address rewriting for packets of the same connection.

Figure 5 shows the element graph of the NAT. It contains 10 elements. Note that IP FILTER and TCP CHECKSUM are reused from the load balancer.

We prove that the NAT fulfills the requirements proposed in RFC5382 [22]:

PROPERTY 7 (Endpoint-Independent Mapping). For any packets p_1 and p_2, both sent from internal address and port $(X : x)$, where

- p_1 is targeting at external endpoint $(Y_1 : y_1)$ and got its source address and port translated to $(X_1' : x_1')$
- p_2 is targeting at external endpoint $(Y_2 : y_2)$ and got its source address and port translated to $(X_2' : x_2')$

then the NAT should guarantee that $(X_1' : x_1')$ and $(X_2' : x_2')$ are always equal.

PROPERTY 8 (Endpoint-Independent Filtering). For any external endpoints $(Y_1 : y_1)$ and $(Y_2 : y_2)$. If NAT allows connections from $(Y_1 : y_1)$, then it should also allow connections from $(Y_2 : y_2)$ to pass through.

PROPERTY 9 (Hairpinning). If the NAT currently maps internal address and port $(X_1 : x_1)$ to $(X_1' : x_1')$, a packet p originated from internal network whose destination is $(X_1' : x_1')$ should be forwarded to the internal endpoint $(X_1 : x_1)$. Furthermore, the NAT also need to create an address mapping for p's source address and rewrite its source address according to the mapping.

These properties are essential to ensure transparency of NAT and is required for TCP hole punching in peer-to-peer communications.

We also prove that the NAT preserve the address mapping for a constant amount of time:

PROPERTY 10 (Connection Memorization). If at time t the NAT with state forwards packet from certain connection c, then for all states s' reachable before time $t + THRESHOLD$, where *THRESHOLD* is a pre-defined constant value, packets in c can still be forwarded.

Property 10 guarantees that the NAT can translate the address of all packets from a TCP connection consistently. The constant *THRESHOLD* defines a time window where the

TCP connection should be memorized by NAT, leaving to the actual implementation the freedom to recycle resources used for storing connection information after it expires.

5 Evaluation

This section aims to answer the following questions:

- Can Gravel's verification framework prevent bugs?
- How much run-time overhead does Gravel introduce to middleboxes?
- How much extra effort is required to verify middleboxes with Gravel?

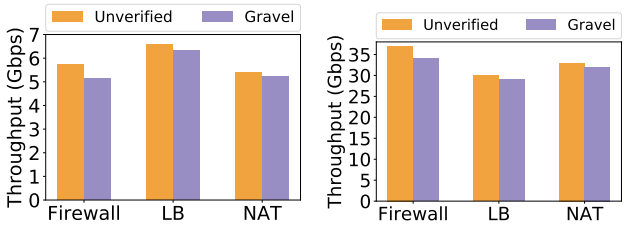
5.1 Bug prevention

To evaluate the effectiveness of Gravel's verification in preventing bugs, we manually analyze bugs from several open-source middlebox implementations. We examine bug trackers of software middleboxes with similar functionalities as those in our case studies (§4) and search the CVE list for related vulnerabilities. We inspect bug reports from the NAT and firewall of the netfilter project [19], and the Balance [2] load balancer. Since the netfilter project contains components other than the NAT and firewall, we use the bug tracker's search functionality to find bugs relevant only to its NAT and firewall components. We inspect the most recent 10 bugs for all three kind of middleboxes, and list the result in Table 3. Of the 30 bugs we inspected, we exclude 10 bugs for features that are not supported in our middlebox implementations, 3 bugs related to documentation issues, 5 bugs on command line interface, and 2 bugs on performance.

From the remaining 10 bugs, Gravel's verifier is able to catch 7 of them. Among these bugs, *Bug #12* in the load balancer and *bug #227* in the NAT can be captured by the verification of the high-level specification as they lead to the violation of Property 4 and Property 10 respectively. Other bugs involving integer underflow or invalid memory access can be captured by the C verifier. Note that there are still three bugs Gravel cannot capture, such as incorrect initialization of the system and properties that are not in our high-level specifications (e.g., unbalanced hashing). These results are consistent with our lessons learned from implementing the three middleboxes: (1) having the high-level specifications checked by the verifier makes it easy to capture bugs related to corner cases; and (2) the implementation verifier can help finding low-level bugs efficiently.

5.2 Run-time Performance

To examine the run-time overhead introduced by Gravel's run-time, we compare Gravel's implementation of the three middleboxes with unverified DPDK implementations of middleboxes with the same functionalities. We use the verified/unverified NAT implementation from VigNAT [26]. We also implement unverified counterparts of the load balancer and the stateful firewall.



(a) 200 Bytes (b) 1500 Bytes

Figure 6: Throughput of middleboxes built with Gravel, compared to their unverified counterparts. Throughput are measured with 200-byte and 1500-byte packets.

Our testbed consists of two machines each with Intel Xeon E5-2680 (12 physical cores, 2.5 GHz), running Linux (v4.4) and has a 40 Gbps Intel XL710 NIC. The two machines are directly connected via a 40 Gbps link. We run the middlebox application with DPDK on one machines and use the other machine as both the client and the server.

We measure the throughput of TCP flows using the *iperf* tool, and use *NPtcp* for measuring latency (round trip time of a 200-byte TCP message). Figure 6 and Table 4 show the results for throughput and latency. For throughput, we create 5 TCP connections between client and server to saturate the load of the middlebox. Gravel's element abstraction and directed graph model of execution introduces a maximum overhead of 3.6%, 4.1%, and 10.5% for NAT, load balancer, and firewall, respectively on throughput. Gravel also incurs a maximum of 1% overhead on latency. This shows that when executed in a single thread, Gravel's element abstraction and its runtime do not introduce much overhead to packet processing.

5.3 Development Effort

We first evaluate the amount of development effort for building verified middlebox using Gravel and then show results regarding the verification time.

Developers' Effort Table 5 shows the size of the code base of Gravel and middleboxes built on top of it. The design and implementation of Gravel's run-time system and verifier took about 5 person months. As for building middleboxes on top of Gravel, the load balancer and stateful firewall each takes one researcher one day to implement and another day to write the specification (plus fix bugs reported by the verifier). The more complex NAT middlebox takes about 5 person-days for both the specification and the implementation. As a comparison, implementing the unverified load balancer and stateful firewall each takes one person day for the implementation and about 3 hours for debugging and testing.

Middlebox	Bug ID	Description	Can be prevented?	Why/Why not?
Load Balancer	bug #12	Packet corruption	✓	high-level specification
	bug #11	Counter value underflow	✓	C implementation verifier
	bug #10	Hash function not balanced	✗	Not formalized in specification
	bug #6	throughput not balanced	✗	Not formalized in specification
Firewall	bug #822	Counter value underflow	✓	C implementation verifier
	bug #691	segfault by uninitialized pointer	✓	C implementation verifier
	bug #1085	Malformed configuration leading crash	✗	Gravel assumes correct init
NAT	bug #658	Invalid packet can bypass NAT	✓	C implementation verifier
	bug #227	Stale entries may not expire	✓	high-level specification
	bug #148	Infinite loop	✓	C implementation verifier

Table 3: Bugs from real-world software middleboxes

Middlebox		Latency(μ sec)
NAT	VigNAT (unverified)	24.15
	VigNAT (verified)	24.19
	Gravel	24.19
Load Balancer	Unverified	24.20
	Gravel	24.25
Firewall	Unverified	24.2
	Gravel	24.17

Table 4: Packet-level latency of middleboxes built with Gravel, compared to their unverified counterparts.

Component		LOC	Verif. Time (s)
NAT	Impl	737	48.08
	Spec (element)	168	2.20
	Spec (high-level)	90	/
Firewall	Impl	364	13.39
	Spec (element)	73	0.67
	Spec (high-level)	70	/
Load Balancer	Impl	704	8.70
	Spec (element)	101	1.48
	Spec (high-level)	68	/
Gravel run-time	DPDK wrappers	493	/
	Gravel API	739	/

Table 5: The lines of code and verification time on three middleboxes using Gravel.

Our experience with Gravel shows that Gravel does not introduce much overhead for implementation, especially when it is possible to reuse existing verified elements. Though it does takes more time to write the specification compared with testing and debugging, we do want to point out that the testing applied to unverified versions only ensures successful execution of our benchmark and does not cover all the corner cases.

Performance of Verifier With the per-element specification as the middle layer, Gravel's verifier can prove the middlebox applications efficiently. As shown in Table 5, most of the verification time is spent on proving the equivalence of the C implementation of each element and its specification. For all the three middleboxes, proving the correctness of both the C implementation and the element-level specification takes less than one minute each. This suggests that Gravel's two-step verification described in §3 and SMT encodings are effective.

6 Discussion

Experience with Gravel. We started the development of Gravel as effort of verifying single middlebox applications. But then later realized that many of the techniques and abstractions used there can be generalized as a framework for building middleboxes. During the course of building the three middleboxes used in our case studies(§4), we found that decomposing middlebox into multiple elements helps us reusing existing verified code more efficiently. We also find that having Gravel check the high-level specification can help us find bugs related to corner cases, for example, packets with same source and destination addresses or packets with different layouts like those with a VLAN header or extra IP options.

Expressiveness of encoding high-level middlebox properties in Gravel. The high-level middlebox properties we choose to verify in the case studies §4 are either from RFCs and from design requirement of other systems [10, 17]. In our experience, we can encode all of them in the Gravel framework concisely. There are still several properties of NAT that we have not tried to encode, such as NAT's port garbage collection feature and support for application-level gateway. These properties are stated as optional requirements in RFC 5382.

Expressiveness of the Gravel programming framework. One common problem with Gravel is that Gravel reduces the programming flexibility. Concretely, Gravel disallows usage of complex data structures such as linked list, binary trees,

etc. We do not encounter these problems in our case studies because states of middlebox can always be expressed in maps. An analogy of this is that P4 has been used for various middlebox functionalities (e.g., load balancer [16], firewall [20], NAT [25]), and the only data structures used in P4 are match-action tables.

Handling control-plane commands. Gravel's interface of event handling can be further extended to support control-plane commands. For example, network administrators may want to add a new filtering rule into a stateful firewall, or adding a new address mapping to the NAT. Currently, the DPDK runtime of Gravel only supports two kinds of events: incoming packets from the RX queue of NICs and periodical time event. However, handling control-plane command do not require a drastic change to Gravel. One possible approach is to add a RPC server into the runtime to handle different types of control-plane operations and add corresponding event handlers on elements. We choose not to support this in our prototype as we do not prove any properties relating to control-plane commands in the three middleboxes built in §4.

Transactional runtime for concurrent packet processing. In Gravel's runtime, the directed graph processes one packet at a time. This means Gravel's throughput is bottlenecked by a single core's maximum clock speed. To scale middlebox performance, there are two options: (1) statically partition the network space into multiple chunks and running each middlebox on a separate core to serve traffic for a single chunk. (2) make the runtime of the middlebox support processing multiple packets at the same time.

Thus Gravel gives developers two approaches to deploy Gravel's runtime accordingly. However, if deployed as (2), developers have to carefully adding locking mechanisms in the C implementation to avoid interleaving. For example, let's say a middlebox has two elements A and B. It process two packets, p1 and p2, at the same time. Element A processes p1 before p2, but element B processes p2 before p1. This makes the middlebox not in a well-define state, because the verifier assumes the middlebox process a packet at a time, and the middlebox must be in a state either before a packet is processed or after a packet is processed.

To eliminate the need for developers to adding locking mechanism, Gravel provides a transactional packet processing framework to intelligently order of which element processes which packet. The runtime framework allows our middleboxes to scale to 40 Gbps line rate on commodity server hardware. Verifying the transactional runtime is our future work. The evaluations shown in §5 is based on single-core performance.

7 Related Work

Middlebox verification. Verifying correctness of middleboxes is not a new idea. Software dataplane verification [9] uses symbolic execution to catch low-level programming errors in existing Click elements [14]. VigNAT [26] further proves a NAT with a low-level specification. VigNAT does not support verifying high-level NAT properties (e.g., hairpinning, endpoint-independence). Compare to these work, our work is a general framework for verifying middlebox for both low-level correctness and high-level properties.

Network verification. There is a separate line of works focusing on verifying network-wide objectives (e.g., whether an destination server is reachable from a source server given a network topology), assuming some abstract switch/middlebox operating models is a faithful representation of switch/middlebox implementations. The goal there is to construct correct SDN controller [1], correct BGP configurations [3], or correct routing table or middlebox configurations [12, 13, 21]. Gravel focuses on verifying the low-level C implementation of standalone middleboxes, rather than network-wide objectives.

Push-button verification. The push-button verification techniques used in Gravel is used in many other previously verified systems [18, 23]. Similar to Gravel, these work also advocates restricted programming model (e.g., no unbounded loop, programming in a single-core state machine paradigm) for efficient and low-effort verification. Our work is inspired by these work and introduces additional techniques, such as decomposition of middlebox for efficient verification.

Modular design for middleboxes. At high level, our modular element graph resembles Click [14]. In Click, modular design helps low-effort implementation of middleboxes because elements in the graph can be reused by different middleboxes. Gravel benefits from this design for efficient software verification.

8 Conclusion

Middleboxes play a critical role in modern networks, yet implement them bug-free is difficult. This paper presents Gravel, a framework to build formally verified middleboxes. Gravel allows developers to encode high-level middlebox properties, and then check correctness in low-level implementations in a push-button fashion. Gravel reduces proof burden by breaking down middlebox functionalities into multiple packet processing elements that can be independently verified. We build a NAT, a load balancer, and a firewall with Gravel and then verify their correctness. Our evaluation shows that middleboxes built with Gravel avoid similar bugs found in today's middleboxes and achieve similar performance of their unverified counterparts. All of Gravel's code will be publicly available.

References

[1] ARASHLOO, M. T., KORAL, Y., GREENBERG, M., REXFORD, J., AND WALKER, D. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *SIGCOMM* (2016).

[2] Balance, Inlab Networks. https://www.inlab.net/balance/.

[3] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A General Approach to Network Configuration Verification. In *SIGCOMM* (2017).

[4] CVE-2013-1138. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1138, 2013.

[5] CVE-2014-3817. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3817, 2014.

[6] CVE-2015-6271. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6271, 2015.

[7] CVE-2014-9715. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9715, 2014.

[8] CVE-2017-7928. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7928, 2017.

[9] DOBRESCU, M., AND ARGYRAKI, K. Software Dataplane Verification. In *NSDI* (2014).

[10] 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 *NSDI* (2016).

[11] GANDHI, R., LIU, H. H., HU, Y. C., LU, G., PADHYE, J., YUAN, L., AND ZHANG, M. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM* (2014).

[12] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking for Networks. In *NSDI* (2012).

[13] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI* (2013).

[14] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *TOCS* (2000).

[15] LIU, J., HALLAHAN, W., SCHLESINGER, C., SHARIF, M., LEE, J., SOULÉ, R., WANG, H., CAŞCAVAL, C., MCKEOWN, N., AND FOSTER, N. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM* (2018).

[16] MIAO, R., ZENG, H., KIM, C., LEE, J., AND YU, M. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM* (2017).

[17] MOSHREF, M., BHARGAVA, A., GUPTA, A., YU, M., AND GOVINDAN, R. Flow-level State Transition as a New Switch Primitive for SDN. In *HotSDN* (2014).

[18] NELSON, L., SIGURBJARNARSON, H., ZHANG, K., JOHNSON, D., BORNHOLT, J., TORLAK, E., AND WANG, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP* (2017).

[19] The netfilter.org project. https://www.netfilter.org.

[20] p4c_firewall. https://github.com/open-nfpsw/p4c_firewall, 2016.

[21] PANDA, A., LAHAV, O., ARGYRAKI, K. J., SAGIV, M., AND SHENKER, S. Verifying Reachability in Networks with Mutable Datapaths. In *NSDI* (2017).

[22] NAT Behavioral Requirements for TCP. Available from IETF.

[23] SIGURBJARNARSON, H., BORNHOLT, J., TORLAK, E., AND WANG, X. Push-button Verification of File Systems via Crash Refinement. In *OSDI* (2016).

[24] STOENESCU, R., DUMITRESCU, D., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Debugging P4 programs with Vera. In *SIGCOMM* (2018).

[25] switchp4. https://github.com/p4lang/switch, 2015.

[26] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A Formally Verified NAT. In *SIGCOMM* (2017).