Gravel: Automated Software Middlebox Verification

Kaiyuan Zhang Danyang Zhuo
Aditya Akella Arvind Krishnamurthy Xi Wang

Abstract

Building formally-verified software middlebox is attractive for network reliability. In this paper, we explore the feasibility of verifying "almost unmodified" software middleboxes. Our key observation is that software middleboxes are already designed and implemented in a modular way (e.g., Click). Further, to achieve high performance, the number of operations each element or module performs is finite and small. These two characteristics place them within the reach of automated verification through symbolic execution.

We perform a systematic study to test how many existing Click elements can be automatically verified using symbolic execution. We show that 47% of the elements can be automatically verified and an additional 21% of click elements can be automatically verified with slight code modifications. To allow automated verification, we build Gravel, a software middlebox verification framework. Gravel allows developers to specify high-level middlebox properties and checks correctness in the implementation without requiring manual proofs. We then use Gravel to specify and verify middlebox-specific properties for several Click-based middleboxes. Our evaluation shows that Gravel avoids bugs that are found in today's middleboxes with minimal code changes and that the code modifications needed for proof automation do not affect middlebox performance.

1 Introduction

Middleboxes (e.g., NATs, firewalls, and load balancers) play a critical role in modern networks. Yet, building functionally correct middleboxes remains challenging. 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.

Given the importance of building functionally correct middleboxes, researchers have turned to using formal verification in building middleboxes and have made significant progress [13, 26]. Crucially, these efforts tackle real middlebox implementations rather than abstract middlebox models and verify non-trivial program properties. However, just as with using software verification in other areas of computer systems, this can incur a non-trivial amount of proof effort (e.g., 10:1 proof to code ratio in VigNAT [26]). At the same time, the excessive proof effort prevents researchers from exploring verification of high-level middlebox-specific properties (e.g., a middlebox rejects unsolicited external connection). As a consequence, recent verification efforts focus either entirely on low-level code properties (e.g., free of crashes, memory safety) [13] or on proving equivalence to pseudocode-like low-level specifications [26].

In this paper, we ask whether it is possible to make software middlebox verification completely automated with minimal proof effort. In particular, our goal is two-fold: (1) we want verification to work on real-world "almost unmodified" middlebox implementations, and (2) we want developers to be able to express and verify high-level middlebox properties directly translated from RFCs (e.g., RFC5382 [2] for NAT) without writing manual proofs towards each of these properties. To deliver on these goals, we seek to replicate the automated reasoning approach used in some recent verification projects that focus on file systems and OS system calls [22, 25]. Specifically, we would like to automatically encode a middlebox implementation and its high-level specification using satisfiability modulo theories (SMT) and then use automated solvers to verify that the implementation is consistent with the specification.

Our key observation regarding the suitability of this approach is that many existing middleboxes are already designed and implemented in a modular way (e.g., Click [19]) for reusability. As they aim for high performance, the number of operations they perform on each packet is finite and small. Both characteristics place these middleboxes within the reach of automated verification through symbolic execution. The goal of this paper is to identify and address last-mile obstacles.

We begin by studying whether such automated verification can be applied to existing software middleboxes. We perform a systematic study on all 425 unmodified Click elements (≈60K lines of code) in Click's official repository to test whether they are suitable for automated verification. We find that 47% of the elements are suitable for automated verification. We then classify code patterns (e.g., unbounded loops, pointers) that can prevent automated verification, and we find that an additional 21% of Click elements can be made amenable to automated verification by modifying the interfaces by which they invoke common data structures (e.g., `Vector`,`HashSet`,`HashMap`).

Encouraged by the results of the empirical study, we designed and implemented Gravel, a framework for automated software verification of middleboxes written using Click [19].

Gravel provides developers programming interfaces to specify high-level middlebox-specific properties in Python. Gravel symbolically executes the LLVM intermediate representation compiled from an element's C++ implementation. Gravel then uses Z3 [12] to verify the correctness of the middlebox without the burden of manual proofs.

We then evaluate Gravel by porting four Click middleboxes: MazuNAT, a load balancer, a stateful firewall, and a web proxy. We verify their correctness against high-level specifications from RFCs and other desirable middlebox properties. Only 133 out of 6457, 63 out of 4294, 63 out of 6336, 50 out of 2683 lines of code need to be modified to make them automatically verifiable. The high-level specification of the middlebox-specific properties can be expressed concisely in Gravel, using only 177, 70, 68, and 39 lines of code. Our evaluation shows that Gravel can avoid bugs similar to those found in existing unverified middleboxes with minimal code modification. Finally, we show that the code modifications do not degrade the performance of the ported middleboxes.

This paper makes the following contributions:

- An empirical study to check whether existing Click elements are suitable for automated verification.
- A framework, Gravel, for automated software middlebox verification based on Click.
- Four case studies (MazuNAT, a stateful firewall, a load balancer, and a web proxy) of middlebox verification with Gravel.

2 Encoding Existing Software Middleboxes

To understand the feasibility of applying automated verification to existing software middleboxes, we perform an empirical study of all the 425 Click elements [19] in Click's official repository. In this section, we first explain what automated verification is and then describe the restrictions needed to be put on middlebox source code for automated verification. Finally, we show that 68% of Click elements are amenable to automated verification after some limited modifications to the code.

2.1 Automated verification

There are primarily two approaches to software verification. One style is deductive verification. In this style, a developer generates a collection of proof obligations from the software and its specifications. Proof assistants, such as Coq [11], Isabelle [23], and Dafny [20], are highly expressive, allowing mathematical reasoning in high-order logic. However, the verification process is largely manual, requiring significant effort from the developer to convey his/her knowledge of why the software is correct to the verification system. For example, when applied to a NAT, VigNAT shows 10:1 proof to code ratio.

```
class CntSrc : public Element {
    // omitting constructor and destructor
    Packet *process_packet(Packet *pkt) {
        if (pkt->ip_header->saddr == _target_src)
            _cnt++;
        return pkt;
    }
    IPAddress _target_src;
    uint64_t _cnt;
}
```

Figure 1: A C++ implementation of a simple packet counter.

Another style of software verification is to exhaustively explore the target software using symbolic execution. This style of software verification has recently become popular because it no longer requires developers' manual proof effort. It has already been used successfully to verify file systems [25] and operating systems [22] with minimal proof effort. However, this style is more restrictive than deductive verification, putting restrictions on the programming model. For example, Hyperkernel disallows unbounded loops in any of its system call handlers.

To see an example of symbolic execution based verification, Figure 1 shows a simple packet counter. This code increments a counter when the source IP address of a packet matches a signature (i.e., _target_src). Here we model this process_packet function in the following way:

$$f : \mathbb{S} \times \mathbb{P} \mapsto \mathbb{S} \times \mathbb{P}$$

where $\mathbb{S}$ is the set of all possible internal states (_target_src and _cnt) and $\mathbb{P}$ denotes the set of all possible packets. For simplicity, this formulation assumes that at most one outgoing packet is generated for each incoming packet. The symbolic execution over the code snippet in Figure 1 generates the following symbolic expression:

$$\begin{aligned}
&\forall s, s' \in \mathbb{S}, \forall p, p' \in \mathbb{P}\cdot \\
&\quad f(s,p) = (s',p') \Rightarrow \\
&\qquad (p' = p) \land (s'.target\_src = s.target\_src) \\
&\qquad \land (p.saddr = s.target\_src \Rightarrow s'.cnt = s.cnt + 1) \\
&\qquad \land (p.saddr \neq s.target\_src \Rightarrow s'.cnt = s.cnt))
\end{aligned}$$

This symbolic expression says that for all possible inputs, outputs and state transitions: (1) the input packet should be the same as the output packet; (2) the _target_src should not change; (3) if the packet's source IP address matches _target_src, the _cnt in the new state should be the _cnt in the old state plus 1; (4) if the packet's source IP address does not match _target_src, the _cnt should not change.

Symbolic execution alone is not enough for automated verification, it only ensures that the above expression can be automatically generated. To ensure automated verification, when the developer verifies the above expression against a specification using an off-the-shelf theorem solver, such as Z3 [12], the solver needs to be able to solve it efficiently.

A program is suitable for **automated verification** if the following two conditions hold:

(1) Symbolic execution of the program halts.
(2) After the symbolic execution, the resulting symbolic expression of program state is restricted to an effectively decidable fragment of first-order logic.

Condition 1 means the program has to halt on every possible input. Condition 2 depends on which fragment of first-order logic a solver, such as Z3, can solve efficiently. This fragment changes as solver technologies improve over time. Empirically, we know that if we can restrict the symbolic expression to only the bitvector and equality with uninterpreted functions, a solver can tackle the expression efficiently [22].

2.2 Feasibility of automated verification on Click

To measure what fraction of Click elements are suitable for automated verification, we implement a static analyzer that checks whether each element satisfies the above two conditions. We make our static analyzer strictly conservative. If the static analyzer says a Click element is suitable for automated verification, then Click element is indeed suitable for automated verification. If the static analyzer says a Click element is not suitable for automated verification, then Click element might still be suitable for automated verification with additional static analysis techniques, in which case we have a false negative.

As the two conditions are undecidable, our static analyzer checks for the following two more restrictive properties in the compiled LLVM byte code[1] of Click elements:

Absence of loops and recursion. To determine whether Click element's execution is bounded (*Condition 1*), the analyzer first uses a straightforward indicator: whether the program contains loops or recursion. Loops in C++, such as "for" loops, "while" loops, "do while" loops, and "backward goto" statements, all compile to backward jumps in LLVM. This means our analyzer first checks whether: (1) the function call graph has no cyclic dependency, and (2) there is no backward jump in the LLVM byte code. In the case of detecting a backward jump, we also analyze the code to determine whether the loop can be eliminated by unrolling (i.e., loops with a static number of iterations).

Absence of pointers in element state. For the second condition, our static analyzer checks whether the program state can be expressed solely by bit vectors and uninterpreted functions. Recall that in Click, packet handlers of each element can only access two types of program states: packet data and the element's state. Packet data, on one hand, has a relatively small size upper bound (i.e. 1500 bytes MTU for Ethernet packets). Thus, we use a bitvector to express it. Element state, on the other hand, requires greater care so that they are encoded efficiently. Though in theory, one could use bitvectors to encode the entire memory into a symbolic state. However, such an expression could not be efficiently solved due to the sheer size of the search space. Therefore, the static analyzer chooses a conservative criterion, the absence of pointers in element states, as it is easy to see that elements without pointers always have bounded state. Each element in Click is a C++ class. In C++, each class can only have a finite number of member variables, and each non-pointer variable can only consume a finite amount of memory. Thus, the state space of a Click element without pointers can always be expressed by constant size bitvectors. Of course, such criteria introduces false negative to the analyzer. For example, a program uses pointers to access a bounded data structure (e.g., fixed-size array).

[1]We chose to implement the static analyzer on LLVM byte code rather than C++ abstract syntax tree. This is because the compiled LLVM IR code makes it easier to reason about the control flow of the program by eliminating C++ related complexities (e.g., function overloading and interface dispatching).

		Loop	
		No	Yes
Pointer	No	200 (47%)	19 (4%)
	Yes	178 (42%)	28 (7%)

(a) Before

		Loop	
		No	Yes
Pointer	No	289 (68%)	13 (3%)
	Yes	95 (22%)	28 (7%)

(b) After

Table 1: Categorization of Click elements based on the boundedness of states and execution steps.

We run the static analyzer over all the 425 Click elements. Table 1a shows the results. We found that 200 of the existing Click elements (47%) can be automatically verified without code modification. Among the ones that failed our test, 206 (178 + 28) elements failed because of pointers and 47 (19 + 28) of them failed because of loops. 28 of the elements have both pointers and unbounded loops. Next, we study how these Click elements use pointers and loops, and how some slight code modifications can eliminate the usage of pointers to make them amenable for automatic verification.

There are several limitations of our static analysis:

C++ function pointers. C++ has other features that can complicate checking for the absence of loops. For example, C++ allows function pointers and virtual functions, making it impossible to reason about the control flow of the program at compile-time. Fortunately, existing Click elements do not use these features in C++. In general, these C++ features complicate verification. Our framework, Gravel, does not allow usage of function pointers or virtual functions for programming new Click elements.

```
class CheckIPAddress : public Element {
    // omitting constructor and destructor
    Packet *process_packet(Packet *pkt) {
        auto saddr = pkt->ip_header->saddr;
-       for (size_t i = 0; i < _num_bad_src; i++)
-           if (_bad_src[i] == saddr)
+       if (_bad_src.find(saddr) != _bad_src.end())
                return NULL;
        return pkt;
    }
-    IPAddress *_bad_src;
-    size_t _num_bad_src;
+   HashSet<IPAddress> _bad_src;
}
```

Figure 2: Modification of `CheckIPAddress`'s implementation.

Click Program versus Click Elements. The analysis we have done is at the level of Click elements. A Click program is a datagraph connecting these elements. Even if all the elements are free of loops, a loop can be introduced at the datagraph-level, and thus prevent automated verification for the entire Click program. Our framework rejects a Click configuration with unbounded loops. (See §4.)

2.3 Code Modification for automated verification

We propose one form of code modifications that can make a larger fraction of elements amenable for automated verification. The modification is to mask the use of pointers under well-defined data structure interfaces. Let's take the `CheckIPAddress` element as an example (Figure 2). This element serves as a source IP packet filter. Before our proposed modifications, `CheckIPAddress` stores a list of bad IP addresses (`_bad_src`). A packet is dropped if the source IP address of the packet is listed in the bad IP address list. In this element, `_bad_src` and `_num_bad_src` together represents a fixed size array containing the bad IP addresses. To check whether the source IP address of a received packet matches any address in the array, `CheckIPAddress` uses a "for" loop to go through this array to find a matching source IP address. Before modification, `CheckIPAddress` is not suitable for automated verification: (1) the size of the array that `_bad_src` is pointing to is not known by the symbolic executor, thus the executor may flag an out-of-bound memory access; (2) the symbolic executor faces a path explosion problem as the number of iterations in the loop can be very large (up to $2^{64} - 1$ iterations on a 64-bit machine).

To make this element meet the conditions for automated verification, developers can modify its implementation as shown in Figure 2. After the modification, the pointer-size pair `_bad_src` and `_num_bad_src` is replaced with an abstract data structure, `Hashset`. Besides that, the "for" loop to check whether the source IP address is in the bad IP address list is also replaced with a `find` method call. The code changes

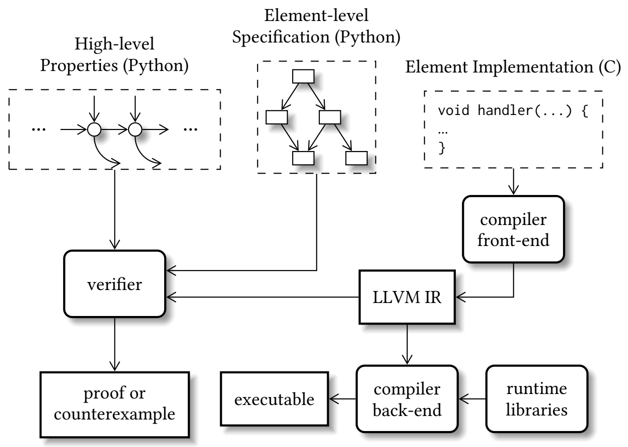

Figure 3: Development Flow of Gravel. Top three boxes denote inputs from middlebox developers; rounded boxes denote compilers and verifiers of Gravel; rectangular boxes denote intermediate and final outputs.

remove both the use of pointers and unbounded loops. Fundamentally, this approach avoids the symbolic execution of the data structure implementation by hiding the implementation under a well-defined data structure interface. When performing the symbolic execution, we can simply provide an encoding in satisfiability modulo theories (SMT) using uninterpreted function theory for common data structures, such as `HashSet`. Note that not all data structures can have their interfaces encoded in SMT. The key challenge here is to prevent state-space explosion: the size of the encoding should not depend on the actual size of the data structure. We managed to encode three commonly used data structures in Click, `Vector`, `HashSet`, `HashMap`, into SMT. (See Appendix A.)

We now investigate how many of the elements can be automatically verified with our data structure interfaces. We let the static analyzer omit elements whose internal state is stored exclusively using `Vector`, `HashSet`, `HashMap`. Table 1b shows the result. The result shows that with the code modification, 289 of Click elements (68%) are suitable for automated verification.

3 The Gravel Framework

Gravel is a framework for specifying and verifying software middleboxes written using Click [19]. It aims to verify high-level middlebox properties, such as a load balancer's connection persistency, against a low-level C++ implementation of the middlebox.

Figure 3 shows the workflow of Gravel. Gravel expects three inputs from middlebox developers:

(1) A Click configuration, which is a datagraph of Click elements.
(2) A set of high-level middlebox specifications.

(3) Element-level specifications for all Click elements used in the configuration.[2]

Like building a normal Click middlebox, Gravel first takes as input a directed graph of Click elements. In Click, a middlebox is decomposed into smaller packet processing "elements". Each element keeps its own private state that is accessible only to itself and has a set of handlers for events such as incoming packet or timer events. Elements can also have a number of input and output ports through which elements can be connected with others and transfer packets. The directed graph in Click configuration connects Click elements together to form the dataplane for packet processing. The topology of the directed graph remains unchanged during the execution of the middlebox.

Then, Gravel requires a formalization of the high-level middlebox properties. In order to check middlebox properties automatically with SMT solver, properties need to be expressed using first-order logic. In Gravel, middlebox properties are formalized as predicates over a trace of events. Gravel includes a Python library for developers to specify the middlebox-specific properties.

Similarly, for each Click element, Gravel also requires a specification. The element-level specification describes each element's private state and packet processing behavior. The purpose of having an element-level specification is to provide a simplified description of an element's behavior and to omit low-level details such as performance optimizations in the element's C++ implementation. Gravel also provides a Python library for developers to write element-level specifications.

With these three inputs, Gravel verifies the correctness of the middlebox in two steps. First, Gravel checks whether Click configuration composed using Click elements satisfies the desired high-level properties of the middlebox. A high-level property is expressed as a symbolic trace of the middlebox's behavior (in Python). Gravel verifies the high-level property by symbolically executing the datagraph of elements using element-level specifications (in Python). Then, Gravel verifies that the low-level C++ implementation of each element has equivalent behavior as the element-level specification. Gravel compiles the low-level C++ implementation into LLVM intermediate representation (LLVM IR) and then symbolically executes the LLVM IR to obtain a symbolic expression of the element. Gravel then checks whether the element-level specification holds in the element's symbolic expression. If there is any bug in Click configuration or in the element implementation, Gravel outputs a counterexample that contains the element state and incoming packet that makes the middlebox violate its specification.

[2]Gravel provides specifications for commonly used elements.

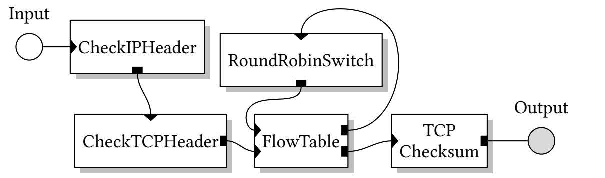

Figure 4: Breakdown of ToyLB's functionalities into packet-processing elements.

3.1 A Sample Application: ToyLB

The rest of this section describes the Gravel framework in the context of a simple running example corresponding to a Layer-3 load balancer, ToyLB. ToyLB receives packets on its incoming interface and forwards 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 resembles popular Layer-3 load balancer designs used by large cloud providers [14, 16].

The ToyLB middlebox is decomposed into 5 elements, as shown in Figure 4. When there is an incoming packet, it first goes through two header-checking elements (the `CheckIPHeader` and the `CheckTCPHeader`). These two elements act like filters and 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 sends the packet to the destination server. Otherwise, the `FlowTable` consults a `RoundRobinSwitch` scheduler element to decide which destination server should the new connection bind to. After the `RoundRobinSwitch` decides which backend server to forward the packet to, `RoundRobinSwitch` 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 further simplicity, low-level functionalities such as ARP lookup are omitted for ToyLB.

In the rest of the section, we describe how Gravel can be used to model high-level specifications of middleboxes such as ToyLB and then outline how the element-level properties are specified. Later (§4), we show how Gravel performs the aforementioned two steps of verification.

3.2 High-level Specifications

Gravel models the execution of a middlebox as a state machine. State transitions can occur in response to external events such as incoming network packets or passage of time. In Gravel, the passage of time is modeled as an external event. The time event can be used to implement garbage collection for middlebox states. For each state transition, the middlebox may also send packets out. Overall, this models packet rewriting, forwarding, and broadcasting.

Gravel follows a "run-to-completion" model: When processing an incoming packet or event, the application always runs until the packet is fully processed by the middlebox before handling any other incoming packet or event.

To encode high-level middlebox properties, Gravel provides a specification programming interface, embedded in Python. Developers can use Gravel's specification programming interface to describe the middlebox properties on a symbolic event sequence. (See Appendix A.)

A packet in Gravel's high-level specification is expressed using a key-value map abstraction, where the keys are the name of the header fields and values are the content of the fields. This abstraction makes the specification concise and hides the implementation details that are less related to high-level properties (e.g., the position of source IP addresses in the packet header).

Gravel provides three kinds of core interfaces (Appendix A) in its high-level specification: (1) a set of `sym_*` functions that allows developers to create symbolic representations of different types of states such as IP address, packet, or middlebox state; (2) middlebox's event handling functions, like `handle_packet(state, pkt)`, `handle_time(state, timestamp)`, that take 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 resulting states after a 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.

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

Let us first consider 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 processing packet p
    o, s1 = handle_packet(s0, p)
    verify(Implies(is_tcp(p), Not(is_none(o))))
```

In this liveness formulation, we first 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`. If `o` is `None`, the middlebox has not generated an outgoing packet. This high-level specification says that, if the incoming packet is a TCP packet, the middlebox has an outgoing packet.

Note that the formulation of the liveness property is abstract given that it 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 the desired middlebox properties.

Now, we move to a more complex load balancer property—connection persistency. This property is crucial to the 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, the formulation requires a forwarding requirement (i.e., the forwarding of packets of a certain TCP connection to b) to hold over all possible event sequences between time t and time $t + WINDOW$. This means that we cannot formulate connection persistency with traces containing only a single event, but, rather, we need to use induction to verify that the property holds on event traces of unbounded length.

Gravel allows us to specify Property 2 as an inductive invariant. First, we formulate the packet forwarding condition that should be held during 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 the 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 ascertain whether the packet is forwarded to `dst_ip` by checking that 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 a certain TCP connection to a backend, subsequent packets from the same connection received within a time period *WINDOW* will be forwarded to the same backend.

```
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 the 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 requires the verifier to prove that if `p0` is forwarded to `dst_ip`, then a packet, `p1`, in the same connection received any time before the expiration time `ddl` is also forwarded to `dst_ip`, assuming that the middlebox state hasn't changed from state `s1`.

In addition to requiring the base case invariant, the specification includes 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.

```
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 prove that the invariant `steer_to(...)` holds on the middlebox states when processing packets or handling time events as long as the timestamp is before the expiration time.

3.3 Element-level Specifications

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 incoming packets and time events.

Element states. The specification of a Gravel element starts with a declaration of the state associated with the element. To ensure efficient encoding with SMT, Gravel requires the state to be bounded. More specifically, elements' state in Gravel may contain: (1) fixed size variables including bitvectors; (2) maps from one finite set to another (e.g., 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
    ...
```

This part of element-level specifications define 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 addresses and port numbers. This map is used to store the results from the `Selector` element.
- `timestamps` stores the latest times at which packets were received 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 bitwidths. Besides the state, the code also informs Gravel as to how many input/output ports the `FlowTable` element has through `num_in_ports`/`num_out_ports`.

Event handlers. As mentioned above, Gravel requires each element to have a handler function for packets received from its input ports. This packet handler needs to be specified in the element-level specification. The specification of the packet handler describes the operations the element performs when handling packets. Similarly, in the element-level specification, developers can also declare handlers for other events such as time events. In Gravel, the two event handlers are defined as functions with the following signatures:

```
flowtable_handle_packet(state, pkt, in_port) → action_list
flowtable_handle_time(state, 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 an element should take under certain conditions. In the python code, developers can write:

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

```
def flowtable_process_packet(s, p, in_port):
    flow = p.ip4.saddr, p.tcp.sport, \
           p.ip4.daddr, p.tcp.dport
    # 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 s.decisions)
    # construct the encapsulated packet
    fwd_pkt = p.copy()
    fwd_pkt.ip4.dst = s.decisions[flow]
    # update the timestamp of the flow with current time
    after_fwd = s.copy()
    after_fwd.timestamps[flow] = s.curr_time
    known_flow_action =
        Action(known_flow,
               {PORT_TO_EXT: fwd_pkt}, after_fwd)
```

Figure 5: Example of an element-level action.

to denote an action that sends pkt_i to output port port_i while also updating the element state to new_state. This action will be taken when condition cond holds. To make it concrete, let's consider the packet handler of FlowTable. Upon receiving a packet, FlowTable does one of the followings:

- If the packet is from the CheckTCPHeader element, and the decisions map contains a record for the connection, FlowTable rewrites the destination address and sends the packet to TCP Checksum element, as shown in Figure 5.
- If the FlowTable does not have a record for a packet, the packet is sent to RoundRobinSwitch element.
- If the packet is sent from RoundRobinSwitch, FlowTable records the destination decided by RoundRobinSwitch and forwards the packet to TCP Checksum.

ToyLB's element-level specifications are listed in Appendix B.

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 is notified of a time change, it updates its curr_time 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 satisfy the predicate. FlowTable uses this to remove all the records that were inactive for a period longer than a constant WINDOW value.

4 Verifier Implementation

Gravel proves the middlebox properties 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). *The C++ implementation of Click is a refinement of that element's specification. That is, every possible state transition and packet processing action of the C++ implementation must have an equivalent counterpart in the element-level specification.*

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

4.1 Graph Composition

Gravel verifies the Graph Composition theorem (Theorem 1) in two steps. First, Gravel symbolically executes an event sequence specified in the high-level specifications. Second, Gravel checks whether the high-level specifications hold on the resulting state of the symbolic execution and the outgoing packets.

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 the symbolic states of all the elements in the middlebox. 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 by 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 event type, Gravel records the updated state of each element as well as the packet produced by each output element. This 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 inquires the Z3 SMT solver to see if they always hold.

Loops in the graph. Gravel allows the directed graph of elements to contain loops in order to support bi-directional communications between elements, such as `FlowTable` and `RoundRobinSwitch` in ToyLB (§3). However, loops may introduce non-halting execution when we symbolically execute the datagraph. 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 raises an alert and fails the verification. For example, in ToyLB, the `FlowTable` is hit at most twice: when `FlowTable` cannot find a record for a certain packet, the packet is sent to `RoundRobinSwitch`, which will later send the packet back to `FlowTable`; upon receiving packets from `RoundRobinSwitch`, `FlowTable` records the selected backend server into its own records and does not send the packet back to `RoundRobinSwitch`. Thus, the maximum number of elements traversed during the symbolic execution is 6, and developers can safely set 6 as the limit for ToyLB.

The graph composition verifier is implemented with 1981 lines of Python code. The verifier exposes a similar set of interfaces as Click configuration language so that developers could port existing Click elements into the verifier. Currently, the translation from Click configuration to the graph composition verifier can only be done manually. The verifier uses the Python binding of Z3 to generate symbolic packets and element states.

4.2 Element Refinement

Gravel verifies the Element Refinement theorem (Theorem 2) in two steps. First, a symbolic expression of the element is generated by symbolically execution of each event handler's compiled LLVM intermediate representation. Second, Gravel checks if the element's specification holds on the symbolic expression of the implementation.

Before performing the symbolic execution, Gravel first uses the LLVM library to extract the memory layout of the C++ class of the element, along with the types of each of its member variables. The verifier can later use this information to determine which field is accessed when it encounters a memory access instruction in LLVM bytecode. As mentioned in §2.3, in order to bound the symbolic execution step and state size, abstract data structures are executed by using their abstract SMT model instead of actual implementation code. A complete list of the data structures and interfaces replaced is given in Appendix A.

For packet content access and modification, Gravel's symbolic executor is compatible with Click's `Packet` interface. In the compiled LLVM bytecode, packet content accesses are compiled into memory operations over a memory buffer. To establish the relation between packet header fields and memory offsets, Gravel needs to extract the symbolic header field value for each output packet after the symbolic execution. To do so, Gravel first computes offsets for each header field. Note that these offsets are also symbolic values as they depend on the content of other packet fields. After that, Gravel extracts the value of each header field from the memory buffer of the packet. Each extracted value is then encoded into an SMT formula and compared against fields from the abstract packet object using an SMT solver. Gravel concludes that the packet object and the memory buffer are equivalent when values of all fields are equivalent.

At the end of symbolic execution, the verifier gets a list of ending states, along with the packets sent out at each output port and the path conditions under which it can be reached. For each entry in the list, Gravel uses Z3 to find an equivalent counterpart in the element specification. If such a counterpart exists for all of the entries, the refinement of the element is proved.

Gravel's element refinement verifier is implemented in C++ on top of the LLVM library. The verifier invokes LLVM library's IR parser to load the compiled LLVM bytecode of each Click element. To perform the symbolic execution, Gravel uses the `InstVisitor` interface to traverse through the instructions. Besides the SMT encoding of all LLVM instructions used in the compiled Click elements, the verifier also has the SMT encoding of the abstract data types as described in §2. The refinement verifier and the symbolic executor consists of 10396 lines of C++.

4.3 Trusted Computing Base

The trusted computing base (TCB) of Gravel includes the verifier (used for proving 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 Click runtime. Note that the specification of each element is not trusted.

5 Evaluation

This section aims to answer the following questions:

- How much effort is needed to port existing Click applications into Gravel? Can Gravel scale to verify existing Click applications?
- Can Gravel's verification framework prevent bugs?
- How much run-time overhead does the code modification introduce to middleboxes in order for them to be automatically verifiable by Gravel?

5.1 Case Studies

To evaluate whether Gravel can work for existing Click applications, we port four Click applications to Gravel. For each application, we choose a set of high-level middlebox-specific

Component		LOC	Verif. Time (s)	LOC changed
MazuNAT	Impl	6457	–	133
	Spec (element)	443	64.60	–
	Spec (high-level)	177	3.78	–
Firewall	Impl	4294	–	63
	Spec (element)	73	32.30	–
	Spec (high-level)	70	0.67	–
Load Balancer	Impl	4336	–	63
	Spec (element)	101	10.87	–
	Spec (high-level)	68	1.48	–
Proxy	Impl	2683	–	50
	Spec (element)	92	30.63	–
	Spec (high-level)	39	0.72	–

Table 2: Development effort and verification time of using Gravel on four Click-based middleboxes.

properties either by formalizing them directly or extracting them from existing RFCs. We use Gravel to verify that these properties hold for Click applications. Gravel also automatically verifies the low-level properties, such as memory safety and bounded execution.

5.1.1 MazuNAT

MazuNAT is a NAT that has been used by Mazu Networks. MazuNAT consists of 33 Click elements. (See Appendix C.) MazuNAT forwards traffic between two network address spaces, the internal network, and the external network. It mainly 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 going from the internal to the external network, NAT assigns an externally visible source IP address and port to the connection. The NAT also needs to keep track of assigned addresses and ports to guarantee persistent address rewriting for packets in the same connection.

One common middlebox property we have for all four applications is that the middlebox should not change the payload of the packet:

Property 3 (Payload Preservation). For any packet that is processed by the middlebox, the middlebox never modifies the payload of the packet.

For NAT-specific properties, we verified that MazuNAT meets the requirements proposed in RFC5382 [2]. These requirements are proposed to make NATs transparent to applications running behind them [15].

Property 4 (Endpoint-Independent Mapping). For packets p_1 and p_2, both sent from internal address and port $(X : x)$, where

- p_1 is targeting external endpoint $(Y_1 : y_1)$ and got its source address and port translated to $(X'_1 : x'_1)$
- p_2 is targeting 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 5 (Endpoint-Independent Filtering). Consider 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 6 (Hairpinning). If the NAT currently maps internal address and port $(X_1 : x_1)$ to $(X'_1 : x'_1)$, a packet p originated from the internal network whose destination is $(X'_1 : x'_1)$ should be forwarded to the internal endpoint $(X_1 : x_1)$. Furthermore, the NAT also needs 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 the transparency of the NAT and is required for TCP hole punching in peer-to-peer communications.

We also prove that the MazuNAT preserves the address mapping for a constant amount of time:

Property 7 (Connection Memorization). If at time t, the NAT forwards a packet from a certain connection c, then for all states s' reachable before time $t + THRESHOLD$, where $THRESHOLD$ is a predefined constant value, packets in c can still be forwarded.

Property 7 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 the resources used for storing connection information after the time window expires.

5.1.2 Load Balancer

Besides the round-robin load balancer mentioned in §3, we also verified a load balancer using Maglev's hashing algorithm [14]. Its element graph looks exactly the same as in Figure 4. The only difference is that the `RoundRobinSwitch` element is replaced by a hashing element that uses consistent hashing. The load balancer steers packets by rewriting the destination IP address.

We verified connection persistency for both of the load balancers. The goal of connection persistency is to make load balancing transparent to the clients.

Property 8 (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 steers p_2 to the same backend server before c is closed.

Middlebox	Bug ID	Description	Can be prevented?	Why/Why not?
Load Balancer	bug #12	Packet corruption	✓	high-level specification
	bug #11	Counter value underflow	✓	element refinement
	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	✓	element refinement
	bug #691	segfault by uninitialized pointer	✓	element refinement
	bug #1085	Malformed configuration leading crash	✗	Gravel assumes correct init
NAT	bug #658	Invalid packet can bypass NAT	✓	element refinement
	bug #227	Stale entries may not expire	✓	high-level specification
	bug #148	Infinite loop	✓	element refinement

Table 3: Bugs from real-world software middleboxes

5.1.3 Stateful Firewall

The stateful firewall is adapted from the firewall example in the Click paper [19]. Besides performing static traffic filtering, it also keeps track of connection states between 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 or disconnected.

We prove that the stateful firewall can prevent packets from unsolicited connections [21]. Also, the firewall should garbage collect finished connections.

Property 9 (Firewall Blocks Unsolicited Connection). For any 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 10 (Firewall Garbage-collects Records). For any connection c, a packet p in c from the external network is blocked after the firewall sees a *FIN* or *RST* packet.

5.1.4 Web Proxy

The Web proxy is a middlebox that transparently forwards all web request packets to a dedicated proxy server. When the middlebox receives a packet, it first identifies if it is a web request by checking the TCP destination port number. For web request packets, the proxy middlebox rewrites the packet header to redirect the packet to the proxy server, and also memorize the translation in order to forward the reply messages from the proxy.

We prove that the web proxy middlebox forwards packets in both directions.

Property 11 (Web Proxy Bi-directional). For a web request packet p with 5-tuple (SA, SP, DA, DP, PROTO), if the middlebox forwards p from the external network to the proxy server and rewrites the 5-tuple to (SA', SP', DA', DP', PROTO), then a packet from the reply flow with 5-tuple (DA', DP', SA', SP', PROTO) should be forwarded back to the sender.

5.2 Verification Cost

To understand the cost of middlebox verification on Gravel, we evaluate the amount of development effort and also the verification time. Table 2 shows the result.

Development effort. We find that porting existing Click applications to Gravel requires little effort and that writing specifications with Gravel are also easy. We only modified 113 lines of code in MazuNAT to make it compatible with Gravel. The firewall and load balancer required only 63 lines of code modifications. Our proxy middlebox required 50 lines of code to be changed. The specifications needed to verify those middleboxes are concise. The high-level specification is below 200 lines of code and the element-level specifications are less than 450 lines of code for all three middleboxes. The associated developer effort is also small. For the web proxy, it took less than one person-day for writing both the high-level properties and the element specifications. The load balancer and the stateful firewall each required a full day's effort in order to port them to Gravel and verify their correctness. The most complicated middlebox in our case study, MazuNAT, took about 5 person-days to port and specify.

Verification time. With Gravel's two-step verification process, Gravel's verifier can efficiently prove that the middlebox applications provide the desired properties. Most of the verification time is spent on proving the equivalence of the C++ implementation of each element and its element-level specification. Verification of the high-level specifications from the element-level specifications took less than 4 seconds for the different applications. Overall, even for MazuNAT, the overall verification time is just over a minute.

5.3 Bug prevention

When verifying MazuNAT with Gravel, we found that the original MazuNAT implementation does not possess the endpoint independent mapping property (Property 4). MazuNAT uses a 5-tuple as the key to memorize rewritten flows. This means that when MazuNAT forwards a packet coming from the external network, the packet's source IP address and source port affects the forwarding behavior, violating Property 4. To fix this, we changed the implementation of

`IPRewriter` element of MazuNAT to use only a part of the 5-tuple when memorizing flows.

To evaluate the effectiveness of Gravel at a broader scope, we manually analyze bugs from several open-source middlebox implementations. We wanted to understand whether these bugs can happen if the middlebox is built using Gravel. We examine bug trackers of software middleboxes with similar functionalities as those in our case studies (i.e., NAT, load balancer, firewall) and search the CVE list for related vulnerabilities. We inspect bug reports from the NAT and firewall of the netfilter project [3], and the Balance load balancer [1]. Since the netfilter project contains components other than the NAT and the 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 kinds 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 3 and Property 7 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).

5.4 Run-time Performance

To examine the run-time overhead introduced by the code modifications we made, we compare the performance of the middleboxes before and after the code modifications. We run these Click middleboxes on DPDK.

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 machine and use the other machine as both the client and the server.

The code modification to make these Click applications compatible with Gravel has minimal run-time overhead. We measure the throughput of 5 concurrent TCP connections using *iperf*, and use *NPtcp* for measuring latency (round trip time of a 200-byte TCP message). Table 4 shows the results. The code modifications introduce negligible overheads in terms of throughput and latency.

Middlebox		Throughput (Gbits / sec)	Latency (μ sec)
MazuNAT	Unverified	37.43	32.15
	Gravel	37.41	33.16
Load Balancer	Unverified	37.48	30.68
	Gravel	37.47	30.79
Firewall	Unverified	37.42	32.20
	Gravel	37.42	32.37
Proxy	Unverified	37.60	32.03
	Gravel	37.59	32.17

Table 4: Performance of verified middleboxes, compared to their unmodified counterparts.

6 Related Work

Middlebox verification. Verifying correctness of middleboxes is not a new idea. Software dataplane verification [13] uses symbolic execution to catch low-level programming errors in existing Click elements [19]. Gravel targets high-level middlebox-specific properties, such as load balancer's connection persistency. VigNAT [26] further proves a NAT with a low-level pseudocode specification. We believe it is non-trivial to extend VigNAT to verify the set of high-level NAT properties (e.g., hairpinning, endpoint-independence) Gravel can verify.

Network verification. In the broader scope of network verification, most existing work [9, 10, 17, 18, 24] targets verifying network-wide objectives (e.g., no routing loop) assuming an abstract switch/middlebox operation model. Gravel, along with other middlebox verification work [13, 26], aims to verify the low-level C/C++ implementation of a single middlebox's implementation.

SMT-based automated verification. Automated software verification using symbolic execution has recently become popular. This technique has been used to successfully verify file systems [25], and operating systems [22]. However, this technique usually requires a complete re-implementation of the target application because of the restricted programming model. We conduct a systematic study on (§2) whether unmodified Click elements can be automatically verified.

7 Conclusion

Verifying middlebox implementations has long been an attractive approach to obtain network reliability. We explore the feasibility of verifying "almost unmodified" software middleboxes. Our empirical study on existing Click-based middleboxes shows that existing Click-based middleboxes, with small modifications, are suitable for automated verification using symbolic execution. Based on this, we have designed and implemented a software middlebox verification framework, Gravel. Gravel allows verifying high-level middlebox properties of "almost unmodified" Click applications. We ported four Click applications to Gravel. Our evaluation shows that Gravel can avoid bugs found in existing middleboxes with small proof effort. Our evaluation also shows that the modifications required for automated verification incur negligible performance overheads. All of Gravel's source code will be publicly available online. This work does not raise any ethical issues.

References

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

[2] NAT Behavioral Requirements for TCP. Available from IETF https://tools.ietf.org/html/rfc5382.

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

[4] CVE-2013-1138. Available from MITRE, CVE-ID CVE-2013-1138., 2013.

[5] CVE-2014-3817. Available from MITRE, CVE-ID CVE-2014-3817., 2014.

[6] CVE-2014-9715. Available from MITRE, CVE-ID CVE-2015-9715., 2014.

[7] CVE-2015-6271. Available from MITRE, CVE-ID CVE-2015-6271., 2015.

[8] CVE-2017-7928. Available from MITRE, CVE-ID CVE-2017-7928., 2017.

[9] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *SIGCOMM* (2016).

[10] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A General Approach to Network Configuration Verification. In *SIGCOMM* (2017).

[11] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl2*. INRIA, July 2016. http://coq.inria.fr/distrib/current/refman/.

[12] de Moura, L., and Bjørner, N. Z3: An efficient SMT solver. pp. 337–340.

[13] Dobrescu, M., and Argyraki, K. Software Dataplane Verification. In *NSDI* (2014).

[14] 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).

[15] Ford, B., Srisuresh, P., and Kegel, D. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference, General Track* (2005), pp. 179–192.

[16] 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).

[17] Kazemian, P., Varghese, G., and McKeown, N. Header Space Analysis: Static Checking for Networks. In *NSDI* (2012).

[18] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI* (2013).

[19] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *TOCS* (2000).

[20] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. pp. 348–370.

[21] 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).

[22] 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).

[23] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Feb. 2016.

[24] Panda, A., Lahav, O., Argyraki, K. J., Sagiv, M., and Shenker, S. Verifying Reachability in Networks with Mutable Datapaths. In *NSDI* (2017).

[25] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. Push-button Verification of File Systems via Crash Refinement. In *OSDI* (2016).

[26] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., and Candea, G. A Formally Verified NAT. In *SIGCOMM* (2017).

A Gravel Programming Interface

A.1 High-level Specification Interface

Table 5 gives a list of the interfaces Gravel offers to the developers. The core interfaces of Gravel includes:

- Functions that generates symbolic value (bitvectors) of different sizes (the `sym_*` API).
- Functions that performs graph composition and returns the result of packet or event processing (`handle_*`)
- The **`verify`** function which informs Gravel's verifier the verification task to perform.

Besides the core interfaces, Gravel also provides a set of helper functions to ease the formalization effort. These functions include functions that access header fields and functions that checks whether two packets are from the same TCP flow. Table 5 also lists some examples of helper functions.

A.2 Modeling Abstract Data Structure

As discussed in §4, Gravel masks the actual C++ implementation of several data structures and replace them with an SMT encoding during the symbolic execution in order to generate SMT expressions that could be efficiently reasoned about by SMT solvers. Table 6 lists all the interfaces that Gravel's symbolic executor masks during the verification process. This section gives more details on how Gravel generates SMT encoding for these data structure interfaces in a way that the resulting formular can be effciently solved.

Unlike bounded data such as the content of a network packet or an integer field in element state, which can be encoded as a symbolic byte sequence using the bitvector theory of SMT, these data structures have a large state space. This means that encoding them with bitvectors does not results in practically solvable expression. For example, the state of a `HashMap<IPAddress, IPAddress>` could grow up to $2^{64}-1$ bytes. This sheer size makes it infeasible to be encoded using bitvectors.

Gravel's symbolic executor choose to use a different approach and represents data structures as a set of uninterpreted functions. In the aforementioned `HashMap` example, Gravel represents the map as two functions:

$$f_{contain} : \{0,1\}^{32} \mapsto \{\bot, \top\}$$
$$f_{value} : \{0,1\}^{32} \mapsto \{0,1\}^{32}$$

$f_{contain}$ maps from the key space $\{0,1\}^{32}$ to boolean space and represents whether certain key is present in the `HashMap`. Similarly, f_{value} represents the mapping between hashmap keys and the corresponding values.

Each of the data structure interfaces is also modeled by Gravel as operations performed on uninterpreted functions. For the `find(K k)` interface of `HashMap`, Gravel first gets the symbolic value representing whether the key is in the map by computing $f_{contain}(k)$. Based on the result, Gravel takes different actions:

$$\text{If } f_{contain}(k) = \top, \texttt{find(k)} = f_{value}(k)$$
$$\text{If } f_{contain}(k) = \bot, \texttt{find(k)} = \bot$$

In the actual implementation, $\bot$ is represented as `HashMap::end()`.

The `intert(K k, V v)` interface performs update on the content of the `HashMap`. In Gravel, this is modeled as creating a new set of uninterpreted functions, $f'_{contain}$ and f'_{value} such that:

$$\begin{aligned}&\forall k' \in \{0,1\}^{32}\cdot\\ &\quad f'_{contain}(k') = (f_{contain}(k') \vee (k = k'))\\ &\quad \wedge (k \neq k') \Rightarrow f'_{value}(k') = f_{value}(k')\\ &\quad \wedge f'_{value}(k) = v\end{aligned}$$

Similarly, `erase(K k)` replaces $f_{contain}$ with a new function $f'_{contain}$ such that:

$$\forall k' \in \{0,1\}^{32} \cdot f'_{contain}(k') = f_{contain}(k') \wedge (k \neq k')$$

Besides modeling interfaces from existing Click code base, Gravel also adds a set of iteration interfaces that corresponds to commonly used data structure traverse paradigms. These interfaces could be used to abstract away loops in the Click implementation and making more elements feasible for automated verification.

Gravel currently provides two interfaces for `HashMap`, `map` and `filter`. for `map` interface, Gravel takes as parameter a function g and replace f_{value} with a function f'_{value} where:

$$\forall k \in \{0,1\}^{32} \cdot f'_{value}(k) = g(k, f_{value})$$

Similarly, `filter` takes a predicate p and create a function $f'_{contain}$ such that:

$$\forall k \in \{0,1\}^{32} \cdot f'_{contain}(k) = p(k, f_{value})$$

The modeling of interfaces of `Vector` and `HashSet` are similar to the modeling of `HashMap` mentioned above. The main difference are that `HashSet` only uses $f_{contain}$ function, where as `Vector` uses a symbolic integer to denote the size of the vector and does not have a $f_{contain}$ function.

B ToyLB's Element-level Specification

This section gives a detailed description of the element-level specification of ToyLB. As mentioned in §3, element-level specification in Gravel is given as a list of "condition-action" pairs. In Gravel, developers write python functions that generates the list of possible actions for an element. For example, The `CheckIPHeader` element only forwards packets that are both IP packets and are not from a known "bad" address:

```
def checkipheader_process_packet(s, p, in_port):
    is_bad_src = p.ip.src in s.bad_src
    return [Action(And(p.ether.ether_type == 0x0800,
                       Not(is_bad_src)),
```

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 5: Gravel's specification programming interface.

```
                {0: p},
                s)]
```

Remember that the `Action` is used to create a *condition-action* entry, which denotes an action that the element takes under certain condition (§3).

Similarly, `CheckTCPHeader` filters all packets that are not TCP packets.

```
def checktcpheader_process_packet(s, p, in_port):
    return [Action(p.ip.proto == 6,
                {0: p},
                s)]
```

`RoundRobinSwitch` not only performs address rewriting for incoming packets, it also updates packet header fields and its own state:

```
def roundrobinswitch_process_packet(s, p, in_port):
    ns, np = s.copy(), p.copy()
    dst_ip = s.addr_map[s.cnt]
    ns.cnt = (s.cnt + 1) % s.num_backend
    np.ip4.dst = dst_ip
    return [Action(True, {0: np}, ns)]
```

The `FlowTable` element have a more complex specification as it takes one of three actions based on both the content of the incoming packet and its own state:

```
def flowtable_process_packet(s, p, in_port):
    flow = p.ip4.saddr, p.tcp.sport, \
           p.ip4.daddr, p.tcp.dport
    # 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 s.decisions)
    # construct the encapsulated packet
    fwd_pkt = p.copy()
    fwd_pkt.ip4.dst = s.decisions[flow]
    # update the timestamp of the flow with current time
    after_fwd = s.copy()
    after_fwd.timestamps[flow] = s.curr_time
    known_flow_action =
        Action(known_flow,
               {PORT_TO_EXT: fwd_pkt}, after_fwd)

    # the case when flowtable does not know the flow
    consult_sched = And(
        in_port == INPORT_NET,
        Not(flow in s.decisions))
    unknown_flow_action =
        Action(consult_sched, {PORT_TO_SCHED: p}, s)

    # packet from the Scheduler
    register_new_flow = in_port == IN_SCHED
    # extract the new_flow
    new_flow = p.inner_ip.saddr, p.tcp.sport, \
               p.inner_ip.daddr, p.tcp.dport
    # add the record of the new_flow to FlowTable
    after_register = s.copy()
    after_register.decisions[new_flow] = p.ip4.daddr
    after_register.timestamps[new_flow] = s.curr_time
    register_action =
        Action(register_new_flow, {PORT_TO_EXT: p},
               after_register)

    return [known_flow_action,
            unknown_flow_action,
            register_action]
```

C Verifying Properties of MazuNAT

The MazuNAT middlebox is the most complicated application Gravel verifies in the case study (§5.1). Figure 6 shows the directed graph of Click elements extracted from its configuration file.

The three properties of MazuNAT proved by Gravel are extracted from RFC [2]. They are important to provide transparency guarantees for application running inside the network. Here we give the formalization of them in Gravel using Gravel's Python interface.

Payload Preservation (Property 3). The specification of Property 3 simply says that the payload of any packet

Function name	Description
Vector<T>:	
`const T& get(unsigned int)`	Get value by index
`void set(unsigned int i, T v)`	Set i-th value of vector to `v`
`void map(void(*)(T) f)`	Apply function `f` for all value in vector
HashMap<K, V>:	
`V &find(K k)`	Lookup by key `k`
`void insert(K k, V v)`	Insert key-value pair `k, v` into the hashmap
`void erase(K k)`	Delete key `k` from the hashmap
`void map(void(*)(K k, V v) f`	Apply function `f` to all key-value pair in hashmap
`void filter(bool(*)(K k, V v) p)`	Filter key-value pairs in the hashmap with predicate `p`
HashSet<T>:	
`T &find(T v)`	Check if `v` is present in hashset
`void insert(T v)`	Insert `v` into the hashset
`void erase(T v)`	Delete `v` from the hashset
`void filter(bool(*)(T v) p)`	Filter with predicate `p`

Table 6: Data structure interfaces supported by Gravel.

forwarded by the middlebox remains the same. Note that this is a general property that can be verified on multiple middleboxes.

```
def test_payload_unchanged(self):
    p, s = sym_pkt(), sym_state()
    for source in sources:
        ps, _ = handle_packet(s, source, p)
        for sink in sinks:
            verify(Implies(Not(ps[sink].is_empty()),
                           ps[sink].payload == p.payload))
```

Endpoint Independent (Property 4). The high-level specification of Property 4 starts with creating symbolic packet `p1` and symbolic state `s`. Then it creates a new packet `p2` by replace only the source address and port with fresh symbolic values. After that the specification uses `process_packet` to get the resulting packets from processing `p1` and `p2`. Finally, we ask the verifier to check whether the resulting packets (`o1` and `o2` in the code snippet below) are sent to the same destination.

```
def test_ep_independent(self):
    p1, s = sym_pkt(), sym_state()
    ps1, _ = handle_packet(s, 'from_extern', p1)
    p2 = p1.copy()
    p2.ip.src = sym_ip()
    p2.tcp.src = sym_port()
    p2.udp.src = sym_port()
    ps2, _ = handle_packet(s, 'from_extern', p2)
    for sink in sinks:
        o1 = ps1[sink]
        o2 = ps2[sink]
        verify(Implies(Not(o1.is_empty()),
                       And(Not(o2.is_empty()),
                           o1.ip.dst == o2.ip.dst,
                           dst_port(o1) == dst_port(o2))))
```

Hairpinning (Property 6). As shown below, rather than inspecting the state of elements in MazuNAT to determine whether a address mapping is established. Gravel uses the packet forwarding behavior as the indicator. The specification says that if a packet `p1` from external network is forwarded to internal network. any packet `p2` with the same destination address and port received from internal network is also forwarded to the same destination in the internal network.

```
def test_hairpinning(self):
    p1, p2, s = sym_pkt(), sym_pkt(), sym_state()
    out1, _ = handle_packet(s, 'from_extern', p1)
    out2, _ = handle_packet(s, 'from_intern', p2)
    o1 = out1['to_intern']
    o2 = out2['to_intern']
    verify(Implies(And(p1.ip.dst == p2.ip.dst,
                       p1.ip.proto == p2.ip.proto,
                       dst_port(p1) == dst_port(p2),
                       o1.not_empty()),
                   And(o2.not_empty(),
                       o1.ip.dst == o2.ip.dst,
                       o1.tcp.dst == o2.tcp.dst)))
```

Connection memorization (Property 7). The formalization of Property 7 uses the same inductive approach as in the ToyLB example. As shown below, the specification is decomposed into a base case and two inductive cases. The base case states that when a packet from internal network is forwarded to external world by MazuNAT, the translation will be still effective within the time window `THRESHOLD`.

```
def test_memorize_init(self):
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    o, s1 = handle_packet(s0, 'from_intern', p0)
    ext_port = o['to_extern'].tcp.src
    t = s0['rw'].curr_time
    ddl = t + THRESHOLD
    verify(Implies(is_tcp(p0),
                   steer_to(c, s1, p0, ext_port, ddl)))
```

Then, the two inductive cases show that processing a packet from other flows or any time event before the end of the time window do not effect existing translation mappings.

```
def test_memorize_step_pkt(self):
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    t = sym_time()

    p_diff = sym_pkt()
    ext_port = sym_port()
    _, s1 = handle_packet(s0, 'from_intern', p_diff)
    verify(Implies(And(steer_to(c, s0, p0, ext_port, t),
                       from_same_flow(p0, p1)),
                   steer_to(c, s1, p0, ext_port, t)))

def test_memorize_step_time(self):
    ext_port = fresh_bv('port', 16)
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    t0, t1 = sym_time(), sym_time()
    _, s1 = handle_time(s0, 'rw', t1)
    verify(Implies(And(steer_to(c, s0, p0, ext_port, t0),
                       z3.ULT(t1, t0),
                       from_same_flow(p0, p1)),
                   steer_to(c, s1, p1, ext_port, t0)))
```

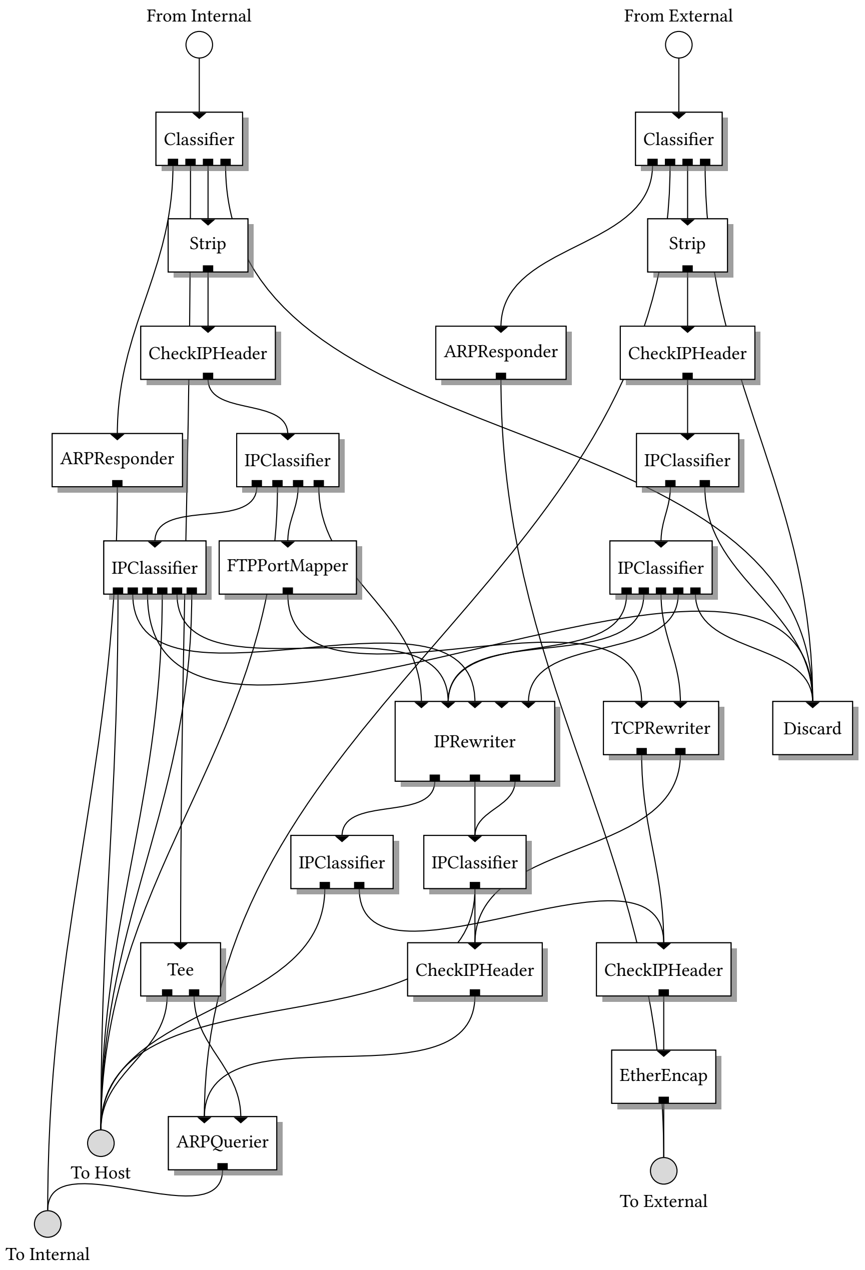

Figure 6: The directed graph of elements in MazuNAT.