Automated Verification of Customizable Middlebox Properties with Gravel

Kaiyuan Zhang *Danyang Zhuo*[†] *Aditya Akella*[#] *Arvind Krishnamurthy* *Xi Wang*
University of Washington [†] *Duke University* [#] *University of Winconsin-Madison*

Abstract

Building a 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 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 45% of the elements can be automatically verified and an additional 33% 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 middlebox implementations. Many of these bugs [8–12] directly lead to system failure or information leaks. Worse still, malformed packets can trigger some of these bugs and expose severe security vulnerabilities.

Given the importance of building functionally correct middleboxes, researchers have turned to formal verification and have made significant progress [14, 40]. 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 [40]). 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) [14] or on proving equivalence to pseudocode-like low-level specifications [39, 40].

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. First, we want verification to work on real-world "almost unmodified" middlebox implementations without requiring manual proofs. Second, we want developers to be able to express and verify high-level properties directly translated from RFCs (e.g., RFC5382 [29] 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 [30, 34]. Specifically, we would like to use symbolic execution to automatically encode a middlebox implementation and its high-level specification using satisfiability modulo theories (SMT) and then use 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 [23]) 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 reach of automated verification through symbolic execution. Thus, one goal of this paper is to identify domain-specific analyses that enable symbolic execution to exploit these characteristics and distill SMT encodings for middlebox implementations.

We begin by studying whether we can use automated verification on existing software middleboxes. We perform a systematic study on all 290 Click elements and 56 Click configurations (≈60K lines of code) in Click's official repository to test whether they are suitable for automated verification. We find that a baseline symbolic executor can derive symbolic expressions for 45% of the elements and 16% of the configurations. We then introduce a set of domain-specific static analyses and code modifications (such as replacing element state by SMT-encoded abstract data types) to enable the symbolic execution of a more substantial fraction of Click elements. These techniques allow us to symbolically execute an additional 33% and 50% of elements and configurations, respectively.

Encouraged by the results of the empirical study, we designed and implemented Gravel, a framework for automated software verification of middleboxes written using Click [23]. Gravel provides developers with programming interfaces to specify high-level trace-based properties in Python. Gravel symbolically executes the LLVM intermediate representation

compiled from an element's C++ implementation. Gravel then uses Z3 [38] to verify the correctness of the middlebox without the burden of manual proofs.

We then evaluate Gravel by porting five Click middleboxes: MazuNAT, a load balancer, a stateful firewall, a web proxy, and a learning switch. We verify their correctness against high-level specifications derived from RFCs and other sources. Only 133 out of 1687, 63 out of 1151, 63 out of 1447, 50 out of 953, and 0 out of 594 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, 39, and 91 lines of code. Our evaluation shows that Gravel can avoid bugs similar to those found in existing unverified middleboxes. Finally, we show that the code modifications do not degrade the performance of the ported middleboxes.

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 290 Click elements and 56 Click configurations[1] in Click's official repository [23]. In this section, we first explain what is automated verification and then describe ways to enhance the effectiveness of automated verification for middleboxes. Finally, we show that 78% of Click elements and 66% of Click configurations are amenable to automated verification after some limited modifications to the code.

2.1 Automating verification using symbolic execution

A well-established approach to software verification 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 [7], Isabelle [32], and Dafny [24], are highly expressive, allowing mathematical reasoning in high-order logic. However, the verification process is mostly 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 [40] shows a 10:1 proof-to-code ratio.

Recently, researchers have started exploring the feasibility of automating the verification process through exhaustive symbolic execution, which encodes the middlebox implementation into a symbolic expression that can be checked against a high-level specification. This style of software verification reduces the developers' manual proof effort and has already been used successfully to verify file systems [34] and operating systems [30]. However, this style is more limited than deductive verification, putting restrictions on the programming model. For example, Hyperkernel requires loops in its system call handlers to have compile-time bounds on their iteration counts.

[1] Our empirical study focuses on the Click elements and configurations that process packets in a run-to-completion model.

```
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.

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 as $f : \mathbb{S} \times \mathbb{P} \mapsto \mathbb{S} \times \mathbb{P}$, 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. Symbolically executing this code snippet generates the following symbolic expression:

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

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 we can automatically generate the above expression. To ensure automated verification, when the developer verifies the above expression against a specification using an off-the-shelf theorem solver (such as Z3 [38]), the solver needs to be able to solve it efficiently.

A program is suitable for **automated verification** if:

1. Symbolic execution of the program halts and yields a symbolic expression.
2. The resulting symbolic expression 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 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 bit vectors and equality with uninterpreted functions,

a solver can tackle the expression efficiently [30].

2.2 Baseline effectiveness of symbolic execution

We now study the feasibility of automating verification by examining middleboxes written using Click. We perform an empirical study on both Click elements and Click configurations, which are datagraphs formed by composing Click elements. This study allows us to measure both the fraction of Click code and the fraction of automatically verifiable Click programs. To perform this empirical analysis, we implement a baseline symbolic executor to analyze whether an element or a configuration satisfies the conditions mentioned above. Since elements are C++ classes, the symbolic executor first analyzes all the member fields to determine whether the state could be encoded with SMT (Condition 2). It then performs symbolic execution over the compiled LLVM byte code[2] of the element to check if Condition 1 is met. However, since both conditions are undecidable, we choose to use the following two conservative criteria. (In fact, we describe in the subsequent section how we augment our baseline symbolic executor with domain-specific extensions.)

Absence of pointers in element state. When the symbolic executor analyzes each of an element's members, it checks whether the element state can be expressed solely by bit vectors and uninterpreted functions. Though one could use bit vectors to encode the entire memory into a symbolic state, it would be difficult to efficiently solve expressions containing such a symbolic state due to the sheer size of the search space. Therefore, we choose 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 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 bit vectors. Of course, such criteria introduce false negatives, for example, using pointers to access a bounded data structure (e.g., fixed-size array).

Absence of loops and recursions. To determine whether Click elements' execution is bounded (Condition 1), the symbolic executor invokes the packet processing code using a symbolic element state and a symbolic packet content. The symbolic executor detects potential unbounded execution by searching for loops and recursive function calls and only performs execution on those elements that do not contain them. The symbolic executor performs the check by comparing each jump/call target with the history of executed instructions.

Table 1 shows the results of running this baseline symbolic executor. We found that 130 of the existing Click elements (45%) are suitable for automated verification. Among the ones that failed our test, 143 elements failed because of pointers, and 78 elements failed because of unbounded execution. 61 of the elements have both pointers and unbounded execution. A Click configuration is amenable to automated verification if and only if all the Click elements in the configuration can be automatically verified. Among the 56 configurations in the official Click repository, only 9 out of the 56 Click configurations (16%) are suitable for automated verification.

[2] We chose to use LLVM byte code rather than C++ abstract syntax tree as the former makes it easier to reason about the control flow by eliminating C++ related complexities (e.g., function overloading and interface dispatching).

2.3 Enhancing symbolic execution

We now augment our baseline executor with additional techniques that aid symbolic execution. We also examine the impact of performing a small number of code modifications to make the middleboxes amenable to automated verification. Some of the techniques described below are broadly applicable but are likely more effective for middlebox programs that operate on packet data with well-defined protocol specifications. The remaining techniques are domain-specific analyses that are suitable only for packet processing code.

Code unrolling. When detecting a backward jump, the symbolic executor unrolls the loop and executes its loop body. The executor keeps count of how many times it executes the backward jump instruction and raises an error if the number goes beyond a pre-defined threshold. This technique is useful when the source code has loops with a static number of iterations or loops whose iteration count is a small symbolic value, as would be the case for code that processes protocol fields of known size.

Pointer analysis to detect immutable pointers and static arrays. In general, we can classify the use of pointers into three categories: pointers to singleton objects, pointers corresponding to arrays, and pointers used to build recursive data structures. These use cases introduce two distinct challenges in the symbolic execution of Click code with pointers. First, the symbolic executor needs to determine whether two pointers point to overlapping memory regions and update the symbolic state of elements correctly irrespective of which pointer is used for the update. Second, when pointers are used to implement recursive data structures, such as linked list or tree, the data structure access often involves loops whose iteration counts depend on the symbolic state of the elements. Our symbolic executor first identifies how pointers are used and then uses the appropriate technique for symbolic execution.

We first use an analysis pass to identify immutable pointers by checking which of the pointer fields in a Click element remain unmodified after allocation. At the same time, we determine which of the other program variables serve as possible aliases for a given pointer field. Further, for pointers pointing to an array of data items, the symbolic executor also performs a static bounds check on accesses performed using the pointers to ensure that all accesses are within allocated regions. By doing so, the symbolic executor can prove an invariant that accesses performed using the array pointer do not touch other memory regions.

After performing these analyses, the symbolic executor

limits itself to handling accesses through immutable and unaliased pointers that refer to either singleton objects or arrays. For each pointer referring to a singleton object, the executor associates a corresponding symbolic value. For each pointer referring to an array of data, the symbolic executor uses an uninterpreted function in SMT to represent the contents of the array. The symbolic executor uses uninterpreted functions that map array index (64-bit integer) to bytes (8-bit integer) to model the content of the array. We choose to use this offset-to-bytes mapping as the unified representation for both array and packet content since reinterpreting a sequence of bytes in memory as a different type is a common practice in packet processing (e.g., parsing packet header, endianness conversion). We record updates to the array as a sequence of (possibly symbolic) index/value pairs. Since the functions are "uninterpreted", they model all possible values of the array data. Compared with bit vectors, representing states with uninterpreted functions makes symbolic execution scale to larger state size [6, 34].

Our symbolic executor does not handle pointers that are used to build a recursive data structure, such as a linked list, except in the case of certain abstract data types for which we are able to provide SMT encodings (as discussed next).

SMT encodings of commonly used abstract data types. Our next technique avoids the symbolic execution of the data structure implementation by hiding the implementation under a well-defined data structure interface. This technique allows us to integrate implementations that may contain unbounded loops or recursive data structures into our analysis. When performing the symbolic execution, we can simply provide an encoding in SMT 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 the explosion of the state space; 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.)

Replace element state with abstract data types. With the SMT encoding of common data types, another technique we could apply is to modify the element implementation by replacing its states with the data types mentioned above. This process requires the developer to inspect how the packet processing code uses a specific element state. If all the accesses performed on the state can be modeled using the interface of a data type with SMT encoding, we could replace the state with the SMT-encoded counterpart and run the symbolic execution on the modified implementation instead.

Consider the `CheckIPAddress` element (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

```
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 to remove the usage of pointers and loops.

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. `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, it may flag out-of-bound memory access. (2) If the executor tries to unroll the loop, it faces a path explosion problem as the number of iterations in the loop can be large.

To make this element meet the conditions for automated verification, we can modify its implementation, as shown in Figure 2. This change is based on the observation that the way `bad_src_` and `num_bad_src_` are used complies with the `HashSet` interface. The change replaces the pointer-size pair `bad_src_` and `num_bad_src_` with a `HashSet`. Besides that, the "for" loop to check whether the source IP is in the bad IP address list is also replaced with a `find` method call. The code changes remove both the use of pointers and unbounded loops. Since the semantics of `HashSet` and its `find` interface is modeled with SMT, we can symbolically execute the element.

Concretization of control flow structures. Middleboxes perform packet classification based on the value of specific fields in the packet header. Packet classification is implemented using finite-state machines, and it is often optimized by statically compiling the classification rules into a state machine model that is stored in memory. When processing an incoming packet, the classifier performs state transitions using the rules until the state machine reaches one of the end states. If the values of the state transition table are abstract, then the classification process would appear to be unbounded.

We address this issue and enable the symbolic execution of the classification tasks. We load the Click configuration containing concrete classification rules and run the state machine creation code of the classification element. We then ingest the raw bytes representing the transition rules into symbols with concrete values. We use symbolic execution to verify that the

Technique	# Elements	# Conf.
Unmodified	130 (45 %)	9 (16 %)
Code unrolling	138 (48 %)	9 (16 %)
Fix-sized array detection	185 (63 %)	9 (16 %)
SMT-encoded abstract datatype	218 (75 %)	13 (23 %)
Replacing with abstract datatype	222 (77 %)	15 (27 %)
Concretization	226 (78 %)	37 (66 %)

Table 1: Number of Click elements and configuration that can be symbolically executed.

contents of the memory region representing the state transition table remain unchanged during program execution. We then symbolically execute the packet classification code but replace the symbolic transition rules with the concrete values identified in the first step. The executor can thus process the packet classification code within a statically bounded number of steps.

2.4 Overall effectiveness of symbolic execution

We now repeat our analysis of Click elements and configurations after enhancing our symbolic executor with these additional techniques. Table 1 shows the result. Our techniques improve the fraction of elements that can be symbolically executed from 45% to 78%. The fraction of Click configurations that are suitable for automated verification improves from 16% to 66%.

Our symbolic executor cannot handle 22% of the Click elements. Among the 64 unsupported elements, 19 of them could not be symbolically executed because there are loops that traverse the payload of the packets (e.g., `AES` element for encryption). Another 26 elements use customized data structures that contain pointers that can not be modeled with SMT. One such example is `LookupIP6Route` element that uses a match table with longest prefix matching as opposed to a traditional exact match hash table. 11 elements contain loops whose number of iterations is based on the current (symbolic) element state. For example, the `AggregateFilter` element, which aggregates incoming packets according to their header values, has to loop over a queue to determine which aggregation group a packet should belong to. 8 elements have pointer accesses that are deeply coupled with the rest of the code that replacing with abstract data types is not feasible. For example, `IP6NDSolicitor` uses a set of linked lists to handle the response messages of the neighbor discovery protocol.

Three approaches can potentially improve Gravel's ability to verify more Click elements automatically. The first approach is to model more data structures using SMT. Currently, Gravel only supports `HashMap`, `HashSet`, and `Vector`. The second approach is to allow developers to write annotations to rule out part of the implementation that is not relevant to the specification. For example, if the developers only want to prove that the `AES` element does not change the TCP header of the packet, the symbolic executor can skip over the loop that traverses the packet payload. The third approach is to use an interactive theorem prover (e.g., Coq [7], Dafny [24]) to verify the correctness of element-level implementations. These interactive theorem provers can verify higher-order logic than what SMT can verify. For example, more sophisticated data structures such as priority queues or an LRU cache could be more easily verified with the help of an interactive prover.

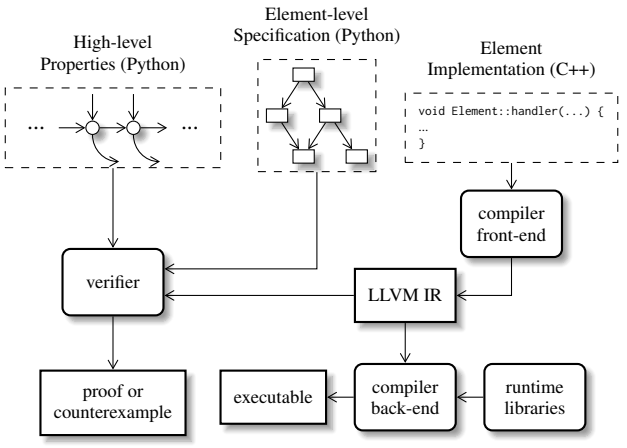

Figure 3: Development Flow of Gravel. Top three boxes denote inputs from middlebox developers; rounded boxes denote compilers and verifiers of Gravel; rectangular boxes denote intermediate and final outputs.

3 The Gravel Framework

Gravel is a framework for specifying and verifying Click [23]-based software middleboxes. It aims to verify high-level properties, such as a load balancer's connection persistency, against a low-level C++ implementation. Gravel uses symbolic execution to translate the C++ implementation into a symbolic expression automatically, and it uses the techniques described in the previous section to enhance the effectiveness of symbolic execution. In this section, we describe how Gravel allows developers to specify the desired high-level properties using Python code and a domain-specific library containing verification primitives. In Section 4, we describe how we check whether the symbolic expression derived from the implementation provides the desired properties.

3.1 Overview

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

1. Click configuration, which is a directed graph of elements.
2. A set of high-level middlebox specifications.
3. Element-level specifications for all Click elements used in the configuration.[3]

[3]Gravel provides specifications for commonly used elements.

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 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 many input and output ports through which elements can be connected with others and transfer packets. The directed graph from a 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.

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

Gravel also requires a specification for each Click element. The element-level specification describes each element's private state and packet processing behavior. The element-level specification provides a simplified description of an element's behavior and omits low-level details such as performance optimizations. Gravel again 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 a 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 the Click configuration or the implementation of the elements, Gravel outputs a counterexample that contains element states and an incoming packet that makes the middlebox violate its specification.

3.2 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 on the packet. ToyLB resembles popular Layer-3 load balancer designs used by large cloud providers [16, 19].

The ToyLB middlebox is decomposed into five elements, as shown in Figure 4. When there is an incoming packet, it first goes through two header-checking elements, `CheckIPHeader` and `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` rewrites 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 backend 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 in ToyLB.

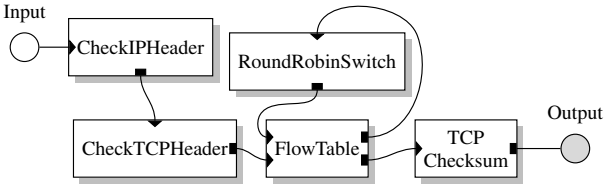

Figure 4: Breakdown of ToyLB's functionalities into packet-processing elements.

We next 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, in §4, we show how Gravel performs verification.

3.3 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 packets or passage of time. The time event can be used to implement garbage collection for middlebox states. For each state transition, the middlebox may also send packets out.

Gravel provides a specification programming interface, embedded in Python, for developers to specify high-level properties. Developers can use the interface to describe middlebox behavior over a symbolic event sequence. (See Appendix A.)

Packets in Gravel's high-level specification are expressed using key-value map abstraction, where the keys are the name of 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 IP addresses in the packet header).

Gravel provides three kinds of core interfaces (see Appendix A) in its high-level specification: (1) a set of `sym_*` functions that allow 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 takes as input the current state of the middlebox and the incoming packet/time event, and returns an (optional) output packet and the resulting middlebox state 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 and (2) connection persistence. We 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 liveness property is abstract, given that it does not say anything about the states of the middlebox. We don't even formulate the set of data structures used by `ToyLB`. This brevity is indeed the benefit of using high-level specifications. 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 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 it 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 complexity 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 forwarding condition that should be held during the time window. The `steer_to` function defined below 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`.

```
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))
```

Then, for the base case of induction, we specify that once ToyLB forwards a packet of a particular TCP connection to a backend, subsequent packets from the same connection received within a period *WINDOW* will be forwarded to the same backend. Similar to the formulation of the liveness property, the following 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`.

```
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)))))
```

In addition to the base case invariant, the specification includes two inductive cases showing that processing an additional event (e.g., a packet from a different connection or time event) does not change the forwarding behavior. The two inductive cases specify that the invariant `steer_to(...)` holds on the middlebox states when processing packets or time events if the timestamp is before the expiration time.

```
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)))
```

3.4 Element-level Specifications

Verifying high-level specifications directly from low-level C++ implementations is hard because of the gap in their semantics. Similar to all the seminal work [20, 30, 34] in software verification, we break down the verification process using refinement. Gravel requires the developer to give specifications of each element. As long as the element-level specifications capture the behavior of their corresponding elements' implementation, we can simply use the element-level specifications to prove the high-level specifications. Compared to deductive verification, this incurs a lower verification effort because element-level specifications are short (§5). Element-level specifications can be reused across different middleboxes. 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. 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 bit vectors; (2) maps from one finite set to another (e.g., a 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 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 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 bit widths. 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`.

```
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.

Event handlers. 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. Besides that, an optional time event handler can also be specified. In Gravel, the two event handlers are defined as functions with the following signatures:

```
flowtable_process_packet(state, pkt, in_port) → actions
flowtable_process_time(state, timestamp) → actions
```

The return value of each event handler (`actions`) 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)
```

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 us 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`.

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 Action(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. ToyLB's complete element-level specifications are in Appendix B.

Summary: Overall, we presented an example of (1) how to specify high-level trace-based middlebox properties, and (2) how to write element-level specifications to make verification modular. We provide a framework for developers to articulate complex trace-based properties. These high-level properties are implementation-independent. Element-level specifications decouple verification problem into two orthogonal problems: that element-level specifications conform to the high-level properties and that elements' implementations comply with their element-level specifications.

4 Verifier Implementation

Gravel proves the middlebox properties with two theorems:

THEOREM 1 (Graph Composition). The element-level specifications, when composed using the given Click configuration, complies with 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 the event sequence specified in high-level specifications. Second, Gravel checks whether the high-level specifications hold on the resulting state and outgoing packets from 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 the symbolic states of all elements in the middlebox. Remember that the high-level specification describes required middlebox behavior on an event sequence. The goal of the symbolic execution is to reproduce the sequence symbolically. For example, if the high-level specification contains an incoming packet, Gravel generates a symbolic packet at the source element of the directed graph. This symbolic packet, when processed by the first element of the graph, can trigger handlers of other downstream elements, which 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. Gravel provides this information 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 uses a 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. Gravel addresses this issue by setting a limit on the number of elements traversed by the symbolic executor. 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. It exposes a similar set of interfaces as Click configuration language so that developers could port existing code into the verifier. 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 for each event handler's compiled LLVM intermediate representation. Second, Gravel checks if the element's specification holds on the symbolic expression.

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 memory access in LLVM bytecode. As mentioned in §2.3, to bound the

symbolic execution step and state size, abstract data structures are executed by using their abstract SMT model instead of actual 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 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. 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 using an SMT solver. Gravel concludes that the packet 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 entries, the refinement of the element is proved.

Gravel's element refinement verifier is implemented in C++ using the LLVM library. The verifier invokes LLVM library's IR parser and reader to load and symbolically execute the compiled LLVM bytecode of each Click element. 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? Can Gravel scale to verify the 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 five Click applications to Gravel. For each application, we choose a set of high-level middlebox-specific properties either by formalizing them directly or extracting them from existing RFCs. We use Gravel to verify that these properties hold. Gravel also verifies the low-level properties, such as memory safety and bounded execution.

		LOC	Verif. Time (s)	LOC changed
MazuNAT	Impl	1687	–	133
	Spec (element)	443	64.60	–
	Spec (high-level)	177	3.78	–
Firewall	Impl	1151	–	63
	Spec (element)	73	32.30	–
	Spec (high-level)	70	0.67	–
Load Balancer	Impl	1447	–	63
	Spec (element)	101	10.87	–
	Spec (high-level)	68	1.48	–
Proxy	Impl	953	–	50
	Spec (element)	92	30.63	–
	Spec (high-level)	39	0.72	–
Switch	Impl	594	–	0
	Spec (element)	131	27.73	–
	Spec (high-level)	91	1.61	–

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

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 property we verified for all five middleboxes is that the middlebox does not change the packets' payload:

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 [29].[4] These requirements are proposed to make NATs transparent to applications running behind them [17].

PROPERTY 4 (Endpoint-Independent Mapping)**.** For packets p_1 and p_2 from the same internal IP, port $(X : x)$, where

[4] We omit the set of requirements related to ICMP becuase MazuNAT does not support ICMP.

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

the NAT should guarantee that $(X_1' : x_1') = (X_2' : x_2')$.

PROPERTY 5 (Endpoint-Independent Filtering)**.** Consider external endpoints $(Y_1 : y_1)$ and $(Y_2 : y_2)$. If the 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 accordingly.

These properties are essential to ensure the transparency of the NAT and are 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 are still forwarded to the same destination.

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 the NAT. The NAT has the freedom to recycle the resources used for storing connection information after the time window expires.

Load Balancer: Besides the round-robin load balancer mentioned in §3, we also verified a load balancer using Maglev's hashing algorithm [16]. 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.

Stateful Firewall: The stateful firewall is adapted from the firewall example in the Click paper [23]. 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 [28]. Also, the firewall should garbage collect finished connections.

PROPERTY 9 (Firewall Blocks Unsolicited Connection)**.** For any connection c, no packet in c from the external network is allowed until a *SYN* packet has been sent out for c.

PROPERTY 10 (Firewall Garbage-collects Records)**.** For any connection c, no packet in c from the external network is allowed after the firewall sees a *FIN* or *RST* packet for c.

Web Proxy: The Web proxy transparently forwards all web requests 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. For web request packets, the proxy rewrites the packet header to redirect them to the proxy server. The proxy also memorizes the sender of the web request to forward the reply messages back to the sender.

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 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.

Learning Switch: The Learning switch implements the basic functionality of forwarding Ethernet frames and MAC learning. The switch learns how to send to an Ethernet address A by watching which interface packets with source Ethernet address A arrives. If the switch has not learned how to send to an Ethernet address, it broadcasts the packet to all its interfaces.

We prove the following properties about the switch.

PROPERTY 12 (Forwarding Non-interference)**.** For any Ethernet address A, the behavior of how the switch forwards packets targeting A is not be affected by packets whose source Ethernet address is not A.

PROPERTY 13 (Broadcasting until Learnt)**.** For any address A, if the switch broadcasts packets targeting A, it keeps broadcasting until a packet from A is received by the switch.

5.2 Verification Cost

To understand the cost of middlebox verification on Gravel, we evaluate the amount of development effort and 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 133 lines of code in MazuNAT to make it compatible with Gravel. The firewall and load balancer required only 63 lines

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.

of code modifications. Our proxy required 50 lines of code to be changed, and the switch requires no modification. Most of the required code changes come from the `IPRewriter` element. We had to remove the priority queue that is used for flow expiration and instead use a linear scan to expire old mappings. Other code changes include removing pointers to other elements in `FTPPortMapper`, replacing `ARPTable` in `ARPQuerier` with hashmaps, and the change of `CheckIPHeader` mentioned in §2. The specifications 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 five middleboxes. The associated developer effort is also small. For the web proxy and learning switch, it took less than one person-day for 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. Five elements (`Classifier`, `IPClassifier`, `IPRewriter`, `CheckIPHeader`, and `EtherEncap`) are reused across these middleboxes, and thus we reuse their element-level specifications.

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 did 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 `IPRewriter` element 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 [31], and the Balance load balancer [3]. 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 [13].

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 Mellanox ConnectX-3 NIC. The two machines are directly connected via a 40 Gbps link. We run the

		Throughput (Gbps)	Latency (μs)
NAT	Unverified	37.39 ($\pm$ 0.03)	14.43 ($\pm$ 0.19)
	Gravel	37.41 ($\pm$ 0.04)	15.14 ($\pm$ 0.22)
LB	Unverified	37.38 ($\pm$ 0.04)	14.82 ($\pm$ 0.23)
	Gravel	37.37 ($\pm$ 0.04)	14.86 ($\pm$ 0.20)
Firewall	Unverified	37.37 ($\pm$ 0.05)	15.21 ($\pm$ 0.20)
	Gravel	37.38 ($\pm$ 0.04)	15.11 ($\pm$ 0.24)
Proxy	Unverified	37.36 ($\pm$ 0.05)	14.54 ($\pm$ 0.19)
	Gravel	37.35 ($\pm$ 0.06)	14.35 ($\pm$ 0.18)
Switch	Unverified	37.36 ($\pm$ 0.05)	15.02 ($\pm$ 0.19)
	Gravel	37.39 ($\pm$ 0.07)	14.96 ($\pm$ 0.29)

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

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.

6 Related Work

Middlebox verification. Verifying the correctness of middleboxes is not a new idea. Software dataplane verification [14] uses symbolic execution to catch low-level programming errors in existing Click elements [23]. Our work is also based on Click, but we target high-level middlebox-specific properties, such as load balancer's connection persistency. In addition, we show that 78% of existing Click elements are amenable for automated verification with slight code modifications. VigNAT [40] proves a NAT with a low-level pseudocode specification. Vigor [39] generalizes VigNAT to a broader class of middleboxes and verifies the underlying OS network stack and the packet-processing framework. We believe it is non-trivial to extend VigNAT and Vigor to specify and verify the set of high-level trace-based NAT properties (e.g., hairpinning, endpoint-independence) Gravel can verify.

We note though that specifying the correctness of programs is a fundamentally hard problem. Gravel chooses to let developers specify high-level specifications on a symbolic trace of packets. We find specifications using Gravel's specification interface to be more abstract than psuedo-code like NAT specification in VigNAT [40]. However, even with Gravel, writing specifications is still hard. For example, specifying the connection persistency property for ToyLB requires the usage of induction (§3.3). Empirically, we find that a trace-based specification is flexible enough to express the correctness of middleboxes in the RFCs we examined.

Network verification. In the broader scope of network verification, most existing work [1, 2, 4, 15, 21, 22, 26, 27, 33, 37] targets verifying network-wide objectives (e.g., no routing loop) assuming an abstract network operation model. Gravel, along with other middlebox verification work [14, 39, 40], aims to verify the low-level C++ implementation of a single middlebox's implementation. As switches become programmable [5], researchers have built tools to debug [36], verify [18, 25] P4 programs. Similar to Gravel, this line of work relies heavily on symbolic execution. Our work targets "almost unmodified" middleboxes written in C++.

Currently, Gravel only supports verification of middleboxes implemented with Click. However, since our key observation on Click middleboxes, that the number of operations performed processing each packet is finite and small, may also hold on non-Click middleboxes, we believe that Gravel's verification techniques can also be applied on other middleboxes. For example, the eXpress Data Path (XDP) in the Linux kernel also constrains the packet processing code to be loop-free. It also only allows a limited set of data structures for maintaining global states. These properties make it seem plausible that one could apply Gravel's verification techniques to it.

SMT-based automated verification. Automated software verification using symbolic execution has recently become popular. This technique has been used to successfully verify file systems [34], operating systems [30], and information flow control systems [35]. 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-based software middleboxes 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 trace-based middlebox properties of "almost unmodified" Click applications. We ported five 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. Gravel's source code is available at https://github.com/Kaiyuan-Zhang/Gravel-public.

Acknowledgments

We thank our shepherd Bryan Parno and the anonymous reviewers for their helpful feedback on the paper. This work was partially supported by NSF (CNS-1714508) and Futurewei.

References

[1] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *POPL* (2014).

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

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

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

[5] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[6] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI* (2008).

[7] COQ DEVELOPMENT TEAM. *The Coq Proof Assistant Reference Manual, Version 8.5pl2*. INRIA, July 2016. http://coq.inria.fr/distrib/current/refman/.

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

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

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

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

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

[13] Data Plane Development Kit. https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html.

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

[15] DUMITRESCU, D., STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Dataplane Equivalence and Its Applications. In *NSDI* (2019).

[16] 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).

[17] FORD, B., SRISURESH, P., AND KEGEL, D. Peer-to-peer communication across network address translators. In *USENIX ATC* (2005).

[18] FREIRE, L., NEVES, M., LEAL, L., LEVCHENKO, K., SCHAEFFER-FILHO, A., AND BARCELLOS, M. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *SOSR* (2018).

[19] 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).

[20] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP* (2015).

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

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

[23] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *TOCS* (2000).

[24] LEINO, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Berlin, Heidelberg, 2010), LPAR'10, Springer-Verlag, pp. 348–370.

[25] 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).

[26] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking Beliefs in Dynamic Networks. In *NSDI* (2015).

[27] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the Data Plane with Anteater. In *SIGCOMM* (2011).

[28] 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).

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

[30] 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).

[31] The netfilter.org Project. https://www.netfilter.org.

[32] NIPKOW, T., PAULSON, L. C., AND WENZEL, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Feb. 2016.

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

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

[35] SIGURBJARNARSON, H., NELSON, L., CASTRO-KARNEY, B., BORNHOLT, J., TORLAK, E., AND WANG, X. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *OSDI* (2018).

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

[37] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: Scalable Symbolic Execution for Modern Networks. In *SIGCOMM* (2016).

[38] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[39] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R., RIZZO, M., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. Verifying Software Network Functions with No Verifcation Expertise. In *SOSP* (2019).

[40] 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

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.

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:

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

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)),
                   {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 [29]. 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 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))
```

Endpointer Independent Mapping (Property 4. For Property 4, the specification starts by creating two symbolic packets, p1 and p2. It then invoke the process_packet on both packets (using the same symbolic state s). After that, it asks the verifier to check if the rewritten packets sending to the external network have the same source address.

```
def to_external(p, s):
    return p.ip.dst != s.public_ip

def same_src(p1, p2):
    return And(is_tcp_or_udp(p1), is_tcp_or_udp(p2),
               p1.ip.src == p2.ip.src,
               src_port(p1) == src_port(p2))

def test_ep_independent_map(self):
    p1, p2, s = sym_pkt(), sym_pkt(), sym_state()

    out1, _ = handle_packet(s, 'from_intern', p1)
    out2, _ = handle_packet(s, 'from_intern', p2)
    o1 = out1['to_extern']
    o2 = out2['to_extern']
    verify(Implies(And(to_external(p1, s),
                       to_external(p2, s),
                       same_src(p1, p2)),
                   same_src(o1, o2)))
```

Endpoint Independent Filtering (Property 5). The high-level specification of Property 5 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_filter(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

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.

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)))
```

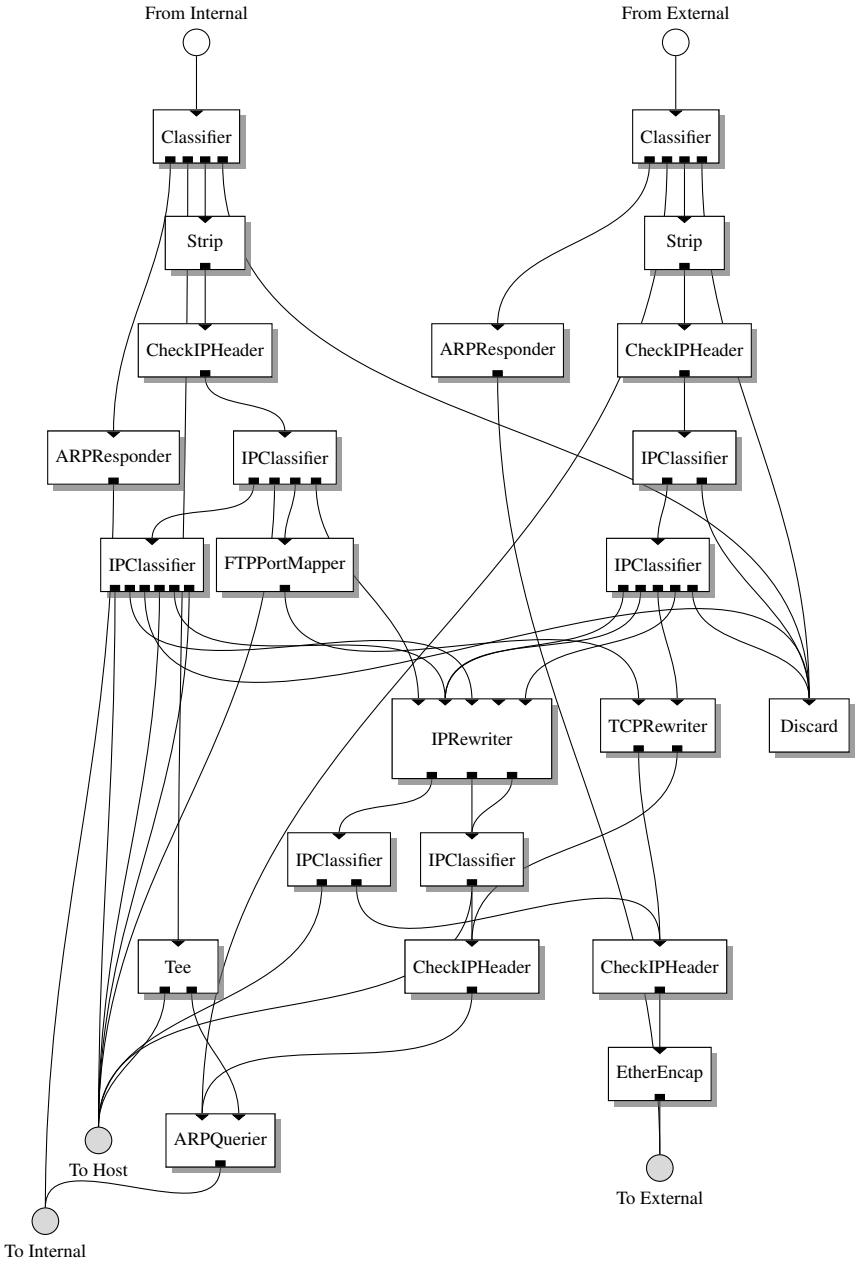

Figure 6: The directed graph of elements in MazuNAT.