SymNet: scalable symbolic execution for modern networks

Radu Stoenescu, Matei Popovici, Lorina Negreanu, Costin Raiciu
University Politehnica of Bucharest
Splaiul Independentei 313, Bucharest, Romania
firstname.lastname@cs.pub.ro

Abstract

We present SymNet, a network static analysis tool based on symbolic execution. SymNet injects symbolic packets and tracks their evolution through the network. Our key novelty is SEFL, a language we designed for expressing data plane processing in a symbolic-execution friendly manner.

SymNet statically analyzes an abstract data plane model that consists of the SEFL code for every node and the links between nodes. SymNet can check networks containing routers with hundreds of thousands of prefixes and NATs in seconds, while verifying packet header memory-safety and covering network functionality such as dynamic tunneling, stateful processing and encryption. We used SymNet to debug middlebox interactions from the literature, to check properties of our department's network and the Stanford backbone.

Modeling network functionality is not easy. To aid users we have developed parsers that automatically generate SEFL models from router and switch tables, firewall configurations and arbitrary Click modular router configurations. The parsers rely on prebuilt models that are exact and fast to analyze. Finally, we have built an automated testing tool that combines symbolic execution and testing to check whether the model is an accurate representation of the real code.

CCS Concepts

•**Networks** → **Network management;** •**Software and its engineering** → *Formal software verification;*

Keywords

Data plane verification, symbolic execution, SymNet

1. INTRODUCTION

Modern networks deploy a mix of traditional switches and routers alongside more complex network functions including security appliances, NATs and tunnel endpoints. Understanding end-to-end properties such as TCP reachability or distributed firewall policy compliance is difficult before deploying the network configuration, and deployment can disrupt live traffic. Dynamic testing (packet generation and tracing) can only catch common issues (e.g. lack of connectivity) but does not scale to large networks.

Static analysis of network data planes allows cheap, fast and exhaustive verification of deployed networks for packet reachability, absence of loops, bidirectional forwarding, etc. We do not aim to verify the control plane (e.g. routing protocols, SDN controllers etc.). Control plane verification is a hard problem that includes checking the correctness of the control plane configuration [8, 9, 4], proving convergence after link additions or failures and characterizing the transient behavior until convergence is reached [2, 11]. We assume the control plane configuration is stable and the control plane has converged and analyze the resulting data plane.

All static analysis tools take as input a *model* of the processing performed by each network box, the links between boxes and a snapshot of the forwarding state, and are able to answer queries about the network without resorting to dynamic testing [23, 14, 19, 20, 21]. What is the best modeling language for networks? If possible, we should simply use the implementation of network boxes (e.g. a C program), as this is the most accurate and is easiest to use. If we view packets as variables being passed between different network boxes, static network analysis becomes akin to software testing. This is a problem that has been studied for decades, and the leading approach is to use *symbolic execution* [3].

Symbolic execution is really powerful: it explores all possible paths through the program, providing possible values for each (symbolic) variable at every point. In the context of static network analysis, the power of symbolic execution lies in its ability to relate the outgoing packets to the incoming ones: even if all the incoming packet headers are unknown, a symbolic execution engine can detect which header fields are allowed in each part of the network, which ones are invariant, and can tell *how* the modified headers depend on the input when they are changed. Unfortunately, symbolic execution scales poorly: its complexity is roughly exponential in the number of branching instructions (e.g. "if" conditionals) in the analyzed program. Applying symbolic execution to actual network code quickly leads to untenable execution times, as shown in [6]. To cut complexity, we must run symbolic execution on models of the code, rather than the code itself. While it is natural to program the models also in C, as previous works do [6, 7], we show that C is fundamentally ill suited for network symbolic execution, and the resulting models are too complex to analyze.

SIGCOMM '16, August 22-26, 2016, Florianopolis , Brazil
© 2016 ACM. ISBN 978-1-4503-4193-6/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2934872.2934881

We propose the Symbolic Execution Friendly Language (SEFL, §4), a network modeling language that we have developed from scratch to enable fast symbolic execution. We have also developed SymNet (§5), a tool that performs symbolic execution on SEFL models of network boxes. Using SymNet and SEFL, we show it is possible to run symbolic execution on large networks to understand network properties beyond simple packet reachability: safety of tunnel configurations, MTU issues and stateful processing (§8.5).

The remaining challenge is to accurately model network functionality in SEFL; this process requires expert input. We have developed parsers that take switch MAC tables, router forwarding tables or CISCO firewall configurations (adaptive security appliance) and automatically generate the corresponding SEFL models. Additionally, we have manually modeled a large subset of the elements from the Click modular router suite [17]: given a Click configuration we can automatically generate the corresponding model. Finally, we have developed a testing tool that takes SEFL models and their runnable counterparts and automatically checks that the model conforms to the actual implementation.

To evaluate SymNet, we have applied symbolic execution to understand documented middlebox interactions[18], to check our department's network and the Stanford backbone. Results show that SymNet is more powerful than existing static analysis tools, captures most real-life interactions in networks, with runtimes in the order of seconds.

2. MOTIVATING EXAMPLES

Static analysis tools are enticing because they can help network operators understand the operation of their deployed networks and inform the correct deployment of updates. Verification tools must have a few desirable properties:

- **Expressiveness** to capture a wide range of data plane processing.
- **Accuracy**: the results provided should not include false positive or false negative answers.
- **Ease of modeling** captures the difficulty of expressing data plane functionality in a way that can be statically analyzed.
- **Coverage**: verify properties beyond reachability such as packet modifications and stateful processing.
- **Scalability** to enterprise and operator networks.

Network data plane analysis is maturing, however our review of related work (§9) highlights that existing tools fail to achieve all these goals simultaneously. In this section we focus only on the most relevant related works: Header Space Analysis (HSA)[14] and Network Optimized Datalog (NOD) [19] that have evolved from research and are now being rolled into production, and traditional symbolic execution has been used to find bugs in Click elements [6]. These tools can capture reachability in L2/L3 networks, but lack accuracy and expressiveness (HSA), are very difficult to model for (NOD) or do not scale ([6]). Here we discuss a number of relevant examples that highlight these limitations.

Consider Header Space Analysis [14] (HSA), the most mature network static analysis tool today. With HSA, the

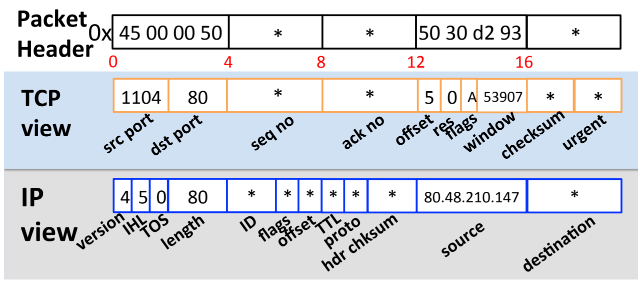


Figure 1: Different ways to interpret a packet header in Header Space Analysis.

packet header is modeled as a sequence of bits, where each bit can take values 0,1 or * (don't care). Network functions are modeled as transformations of the packet header where an input header is transformed into zero or more output headers. HSA (and static analysis in general) achieves great coverage by injecting packets with many "don't care" bits in them, and checking the behaviour of the network. One example is given in Figure 1 where a 20 byte header contains actual values for 8 bytes and don't cares for the rest.

Checking headers. All network functions verify packet headers before processing them. For instance, IP header checking code verifies the values of a few fields (e.g. IP version, header length, flags) and then computes a checksum over all the header fields and compares it to the checksum field.

Header checking is difficult in static analysis: consider the example packet header at the top of Fig. 1 containing a mix of concrete and don't care bytes. Checking whether this is a valid IP header means checking that the concrete values make sense and imposing constraints on the don't care values. To verify that the checksum is valid means to find, for every checksum, a combination of concrete header values that have that checksum. This operation is similar to finding collisions in hash functions and is intractable even for small symbolic fields. This problem affects all static analysis tools.

Once could argue that dealing with corrupt headers is not big concern for static analysis, and that we should just assume the checksum is valid. Does this assumption have any side effects? Consider the simple problem of deciding whether a given header is IP or TCP. In Fig. 1 we show the resulting header field values when we cast the symbolic header as both IP and TCP: note how all the fields have valid values. With HSA, there is *no way to decide* whether this header is an IP or TCP header. Traditional symbolic execution has the same issue: the packet is an array of bytes just as in HSA. The NOD tool models headers differently: each header field is represented as a separate variable, and a network function is a predicate that takes as parameters all the header fields in the packet. NOD can easily verify that the TCP header is different from the IP header because their number of header fields is different (10 vs. 12) in this case, but will not work in many other cases. Our evaluation in §8.2 shows that all three tools lack accuracy in recognising headers in pure symbolic packets.

Routing is the simplest network function, yet modeling it and statically analyzing it is far from trivial. A router does

```
unsigned char *ptr = &options[0];
unsigned char opcode,opsize;
while (length > 0) {
   opcode = *ptr;
   switch (opcode) {
     case TCPOPT_EOL: return True;
     case TCPOPT_NOP:
         length--;ptr++;continue;
     default:
         opsize = *(ptr+1);
         if ((opsize < 2) || (opsize > length)){
             //nop everything!
             for (i=0;i<length;i++)
                ptr[i] = 1;
             length = 0;
             continue;
         }
         switch(_options[opcode]){
           case DROP: return False;
           case ALLOW: break;
           case STRIP:
              for (i=0;i<opsize;i++)
                  ptr[i] = 1;
         }
     }
     ptr+=opsize;length-=opsize;
}
```

Figure 2: TCP Options processing code for a CISCO ASA box with default configuration.

two main jobs: IP lookup and decrement the TTL. Decreasing the TTL is difficult to express in HSA because there is no concept of a variable: we need to enumerate all concrete TTL values (0-255) and specify the concrete output (i.e. 255->254, 254->253, etc.). Both NOD and symbolic execution work with variables, and decreasing the TTL is trivial to model.

Static analysis of route lookup is not trivial when hundreds of thousands of prefixes are present in the forwarding table, as in modern core routers. HSA and NOD accomplish this task in seconds/minutes. However, symbolic execution of an IP router implemented in C is not feasible: when a packet with a symbolic destination address reaches the router, the branching factor is at least as large as the number of prefixes. To make symbolic execution tractable, we would like the branching factor to depend on the number of links of the router instead; this is feasible, but *we must write an optimized router* model *for symbolic execution* (see §7).

Modeling tunnels. Various forms of tunneling are in widespread use, sometimes deployed by different parties. Can we statically verify that such tunnels work correctly? We provide a simple example below, where E1 and E2 perform IP-in-IP encapsulation and D1 and D2 decapsulation.

$$A \rightarrow E1 \rightarrow E2 \longrightarrow D2 \rightarrow D1 \rightarrow B$$

Beyond reachability and header validation, we want to answer the following basic question: are packet contents modified across this tunnel? The answer is obviously no, but HSA cannot capture it: if the input header contains only * bits, the output will also contain * bits, but this does not imply that individual packet contents may not change. We can always feed a specific packet to the model and check it is not modified, but to ensure the invariant holds in general we must try all possible packets—this won't scale.

Length	1	2	3	4	5	6	7
Path count	3	8	19	45	106	248	510
Runtime (s)	0.2	3	20	109	501	2000	9500

Table 1: Runtime of Klee on options parsing code.

A symbolic packet is needed instead—if the symbolic packet doesn't change, the property holds regardless of its value. Symbolic execution can be used to elegantly capture the properties of tunnels, as we show in §7.

We have also modeled the simple tunnel using NOD. NOD can compute the invariant, but modeling is cumbersome and limiting in many ways. First, the models for D1 and D2 differ, despite the fact they are running the same operation. D2 takes a packet with six header fields (to "remember" the two layers of encapsulation): we cannot reuse the D1 model for D2, nor the one from E1 for E2: we need to create a new model instead. In fact, network models in NOD depend not only on the processing of the box, but also *the network topology and the processing of other boxes*. Additionally, models for boxes operating at a lower stack level must also include higher level protocol headers: for instance, a router will model not only its use of the layer 3 fields, but also the upper layer protocols such TCP, UDP or ICMP. In summary, NOD modeling is extremely cumbersome when network topologies are heterogeneous and run multiple protocols; this is not an issue for datacenters where NOD's usage is targeted, but will be a major issue in the wider Internet.

Parsing TCP Options. A network operator has deployed a firewall and wishes to know what TCP extensions are allowed through in its current configuration.

In Figure 2 we show a C code snippet taken from a firewall that processes the TCP options header field. The options field is accessible via the `options` character array, and contains `length` bytes. The middlebox we show allows a number of widely used TCP options, and drops all other options by replacing them with padding.

HSA or NOD can't model this example, but the operator could run *klee* on the middlebox code instead with a symbolic `options` field. the options field after the firewall code, the operator can tell which options are allowed through and which not. In Table 1 we present the number of resulting paths and runtime of *klee*, as we vary the length parameter whose max value is 40. Symbolic execution on C code is very expensive even on fairly simple code snippets.

2.1 Towards a Solution

Traditional symbolic execution is expressive, has great coverage but does not scale and has issues with header validation. When applying symbolic execution to C code, the number of branches in the code exponentially increases the number of paths to be explored. To make symbolic execution feasible, we need to drastically simplify the code being symbolically executed. Unfolding loops and executing both branches of an "if" instruction are techniques that reduce the complexity of symbolic execution at the cost of increased runtime [22]. Such techniques, together with simpler data structures allow verifying small pipelines of Click modular router elements in tens of minutes (Dobrescu et al. [6]).

Our target scale is two orders of magnitude higher: we aim to verify networks containing hundreds or more elements in seconds. Ideally, the number of paths explored by symbolic execution should be comparable to the number of paths in the network; in other words, the model of any network box should not produce more paths than the number of outgoing links from that box; for instance, in the TCP options code in Figure 2, we should have one or two paths at most. This property would make symbolic execution tractable even on very large networks.

To achieve this target, we rely on the key observation that each *path in network symbolic execution must be tied to an active packet* passing through the network: if a codepath does not result in packets, it should not be symbolically executed. This implies that models of network boxes should only focus on the paths that decide the fate of packets, leaving out any logging, reporting, system checks, and so forth.

The C language does not have this property: a packet is just one of many variables handled by the program, and dropping a packet does not stop the execution of the box. Another fundamental problem is the poor handling of data structures, as shown in our TCP options example.

Finally, to increase the accuracy of header verification we must depart from the byte array view of the packet header (e.g. void * in C) by adding and using header field size information (similar to array bounds checking).

3. DESIGN OVERVIEW

We need a new verification-driven modeling language that is imperative —thus easier to program with—and that allows us to harness networking domain knowledge to simplify, as much as possible, the task of the symbolic execution engine. To this end, we have designed a novel language called SEFL (Symbolic Execution Friendly Language) that makes symbolic execution a part of modeling rather than a retrospective verification and validation activity.

A major design question regards the way packets are modeled. With Header Space Analysis, headers have a fixed size, and all possible layers have to be present at all time. This is fine for L2 boxes (such as Openflow switches), but won't work in large, heterogeneous networks. Network Optimized Datalog models packets as a collection of independent "variables"; it can capture tunnels to some extent, but it does not capture the physical layout of packets, and the problems that encapsulation may bring (e.g. interpreting the wrong part of the header, not knowing the higher level protocol, etc.).

SEFL uses a packet layout that mimics real implementations. As in NOD, packet headers are variables, but each header has an absolute offset at which it is allocated. All SEFL headers must be allocated individually, and allocation and deallocation commands include the explicit size of the header field; no symbolic sizes are allowed to ensure tractability. Accesses to header fields must be aligned, otherwise errors are thrown. SEFL thus offers guaranteed *memory safety* for packet headers and simplifies the symbolic execution engine which must not test for memory errors. Memory safety also enables header validation as shown in §8.2.

By design, in SEFL a *packet is tied to an execution path*, and we use the terms path and packet interchangeably in this document. When constraints applied to header fields are unsatisfiable, the execution path fails altogether. This is in contrast with C where a packet header is just a regular variable and execution of a path terminates only on program exit. SEFL incorporates the following features:

- **Built-in map data structure.** SymNet offers a map data structure that SEFL programs can use to create or retrieve values based on concrete string keys. The map helps programmers avoid implementing complex data structures and their associated branching factor (such as those in Fig. 2).
- **Bounded loops**. The only loop instruction supported by SEFL is an iteration over a snapshot of the keys in the native map data structure. This loop can therefore be unfolded and executed without any branching.
- **Dedicated path control instructions.** SEFL allows the user to explicitly drop a packet (path) based on its contents without any branching. Multiple execution paths can be explicitly created with `fork`.

SEFL-coded models are by construction memory safe, have bounded memory usage and are guaranteed to terminate.

4. SEFL LANGUAGE

In table 3 we list all the instructions provided by SEFL, together with their parameters and description. Every instruction implicitly takes as parameter the current execution state (i.e. packet) and outputs a new execution state. The state includes header variables and map entries (called metadata) together with their values and constraints.

The `Allocate` and `Deallocate` instructions create both header fields and metadata, depending on the parameter provided. If `v` is a string, the variable is metadata and is not aligned in any way, and memory safety checks do not apply. `v` acts as a key in the map managed by SymNet. If `v` is an integer (or an expression that evaluates to an integer), it is treated like a header with the associated memory checks.

To simplify access to header fields and to enable layering, any number of *tags* can be defined. The programmer can use indexed addressing based on the tags and a fixed offset rather than absolute addresses to access a header field.

SEFL has instructions to manipulate tags. New tags can be created at absolute values (used when the packet is created), or relative to other tags (used for encapsulation of an existing packet). Tags can be defined dynamically, pointing to addresses where layer two, three and four headers start. The code below performs IP header encapsulation:

```
CreateTag("L3",Tag("L4")-160),
Allocate(Tag("L3")+96,32),   //IP src
Assign(Tag("L3")+96,
         ipToNumber("192.168.1.1")),
Allocate(IpDst,32),          //IP dst
Assign(IpDst,ipToNumber("8.8.8.8"))
```

The notation to access the IP source address field is rather wordy. To make programming easier, we have defined shorthands for all header fields that we work with: `Tag("L3")+96` becomes `IpSrc`. The code to initialize the destination address field uses this shorthand, and is also easier to read. The

Instruction	*Description*
`Allocate(v[,s,m])`	Allocates new stack for variable v, of size s. If v is a string, the allocation is handled as metadata and the optional m parameter controls its visibility: it can be global (default) or local to the current module. If v is an integer it is allocated in the packet header at the given address; size is mandatory.
`Deallocate(v[,s])`	Destroys the topmost stack of variable v; if provided, the size s is checked against the allocated size of v. The execution path fails when the sizes differ or there is no stack allocated for variable v.
`Assign(v,e)`	Symbolically evaluates expression e and assigns the result to variable v. All constraints applying to variable v in the current execution path are cleared.
`CreateTag(t,e)`	Creates tag t and sets its value e, where e must evaluate to a concrete integer value.
`DestroyTag(t)`	Destroys tag t.
`Constrain(v,cond)`	Ensures that variable v always satisfies expression cond. The execution path fails if it doesn't.
`Fail(msg)`	Stops the current path and prints message msg to the console.
`If (cond,i1,i2)`	Two execution paths are created; the first one executes i1 as long as cond holds. the second path executes i2 as long as the negation of cond holds.
`For (v in regex,instr)`	Binds v to all map keys that match regex and executes instruction instr for each match.
`Forward(i)`	Forwards this packet to output port i.
`Fork(i1,i2,i3,...)`	Duplicates the packet and forwards a copy to each output port $i1, i2$, ...
`InstructionBlock(i,...)`	Groups a number of instructions that are executed in order.
`NoOp`	Does nothing.

Figure 3: SEFL instruction set.

decapsulation code does the opposite: first, header fields are deallocated then the L3 tag is destroyed.

SEFL includes two instructions that constrain the execution of the current path, that have no direct correspondent in C. `Fail` stops the current execution path and prints an error message. `Constrain` applies a constraint to a variable, stopping the current path if the constraint does not hold. `Constrain` allows programmers to model filtering behaviour without branching. Below we show the SEFL and C code to drop non-HTTP packets.

```
Constrain(TcpDst==80)         // SEFL
if (p->dst_port==80) free(p); // C
```

The C code results in two execution paths if the dst_port field is symbolic, while SEFL only adds the `TcpDsp==80` constraint to the current path.

`If` forks the current execution state. On one path, it applies the constraint and executes `instr1`. On the "else" branch is applies the negated constraint and executes `instr2`. If more than one instruction must be executed on any branch, an `InstructionBlock` should be used that groups more instructions into a single compound instruction. If any branch is empty, `NoOp` can be used instead.

`For` iterates over the keys in the map that match a given pattern. The code does not branch: a snapshot of the keys is taken and the loop is unfolded before execution.

5. SYMBOLIC EXECUTION

Our symbolic execution tool is called SymNet and consists of 15KLOC of Scala code. To analyze a network configuration, SymNet requires as input the descriptions of all the network elements and their connections. Each network element has input and output ports, as shown in Fig. 4: input ports are shown with a triangle and output ports are shown as rectangles. Connections are unidirectional from output to input ports, so we need two pairs of ports and two links for bidirectional connectivity. Providing a model for a network element means specifying the number of inputs and output ports and associating a set of SEFL instructions to each port.

SymNet starts execution by creating an initial empty packet, with no header fields or metadata, and then executes code to create a symbolic packet of the given type (e.g. TCP). The packet is injected into an input port specified by the user, e.g. port 0 of element A in Fig.4, where SymNet executes the associated instructions.

SymNet instructions take as input an execution path and can modify its associated state, spawn new execution paths or both. The instructions on port 0 may forward the packet to one of the output ports by using the `forward` or `fork` instructions. After the output port instructions are executed, if the state is feasible and there is an outgoing link from the output port, the packet is processed at the next input port.

Values in SymNet can be concrete or symbolic; each value has a unique identifier. For each value, on each path, SymNet holds a list of constraints that apply to that value. Assignment operations modify the top of the current value stack. A path finishes execution when the `Fail` instruction is called, when a constraint does not hold on any of its variables, or when it reaches a port with no outgoing links.

SymNet is considerably simpler than tools like Klee: it does not use heuristics to prioritize paths to explore because our target is finding all the possible execution paths through the network, not just covering all instructions of the network model with at least one execution path. Additionally, SymNet (via SEFL) only supports simple expressions (referencing, subtraction, addition, negation), and this greatly reduces state representation complexity.

To better understand SymNet execution, consider the toy example in Figure 5 showing an implementation of port forwarding in element A. The code is executed when packet 1 reaches any of the input ports of A. The packet enters with header fields `IpDst` and `TcpDst` set to symbolic values. The `constrain` instruction forces packets to be destined

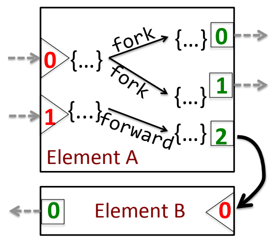
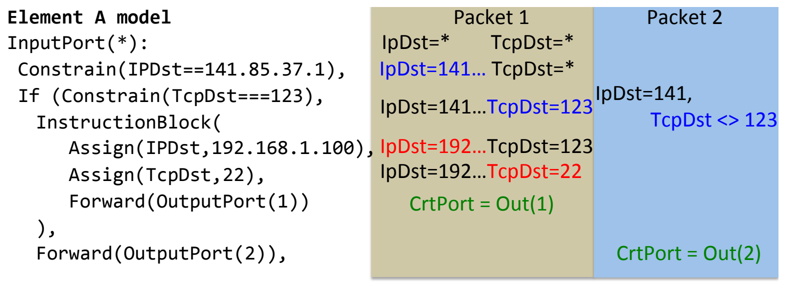


Figure 4: Network model with two elements.



```
Element A model
InputPort(*):
 Constrain(IPDst==141.85.37.1),
 If (Constrain(TcpDst===123),
   InstructionBlock(
      Assign(IPDst,192.168.1.100),
      Assign(TcpDst,22),
      Forward(OutputPort(1))
   ),
   Forward(OutputPort(2)),
```

Figure 5: Symbolic execution with SymNet. The tool keeps a per-path value stack and assignment history for each variable.

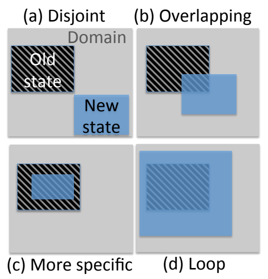


Figure 6: Loop detection algorithm example.

to a specific IP address. The `If` instruction then creates packet 2. All the state of packet 1 is replicated to packet 2 (in fact, it is shared with a copy-on-write mechanism). Next, packet 1 gets the `TcpDst==123` constraint added and checked by a constraint solver (Z3 [5], in our case). The constraint is satisfiable, so packet 1 is propagated further to the assignment instructions which rewrite both the destination address and ports of the packet before forwarding it to output port 0. Packet 1 is now done and SymNet will return to Path 2, first adding the negated constraint `TcpDst!=123` and then asking the solver if it is satisfiable. The packet goes to output port 2 and to element B (port 0).

5.1 Implementation details

SymNet enhances general purpose symbolic execution in two major ways. First, the state contains the map that stores variable names (or memory addresses) as keys and their associated *value stacks*, instead of simple values. Allocation and deallocation instructions push and pop a whole value stack. This allows programmers to quickly "mask" the current value of a variable and restore it later with ease. Secondly, SymNet keeps a complete history of the values associated with a symbol which allows it to detect network loops and to check whether fields change across network hops.

Symbolic execution is an established field, and SymNet implements the standard evaluation algorithm that hasn't changed much during the years. Advances are achieved in two areas: solving path constraints and state representation. The set of path constraints grows as new `Constrain` instructions are executed. When a new constraint is added, the previous set of constraints has already been verified to be satisfiable on the current path, and there is no need to be checked again. We track the evolution of the path constraint set and use caching to ensure each constraint is verified exactly once by Z3. Constructing and validating a Z3 solver context is a frequent, expensive operation that is an ideal candidate for optimization. To reduce the frequency of Z3 invocations, SymNet coalesces consecutive `constrain` instructions when this is safe to do.

To reduce memory overhead, SymNet uses a copy-on-write mechanism to allow paths to share state. Whenever a branching instruction is executed, the resulting paths share a read-only version of the original path state; any updates to the state are only allowed after the state is duplicated.

Finally, path state must be deleted when the associated symbols are out of scope. We have implemented a reference counting - based garbage collection mechanism to clean such state. There is one particularity of SEFL that complicates the garbage collection: the possibility of values of going out of scope, then back in through a pair of `Allocate` - `Deallocate` instructions. To ensure correctness, our garbage collection algorithm keeps in scope the head of each value stack for each symbol.

6. DATA PLANE VERIFICATION

Reachability. It is straightforward to check reachability in a network modeled with SEFL. A symbolic packet is injected at the desired source port, and this packet is then propagated through the network by SymNet. At each port reached by the symbolic packet, we can inspect the values of and constraints on the header variables to discover which packets are allowed, what input packets can reach the output, and how the packets look like at the output, on *all the execution paths that reach that port*.

In the example in Figure 5, a single path reaches output port 1. By examining the history of this path, we can conclude the port is reachable only when the incoming packet satisfies `TcpDst==123`, and the outgoing packet will have its destination address and port overwritten. Output port 2 is also reachable by packets that satisfy `TcpDst!=123`.

Loop detection. The loop detection algorithm relies on the reachability algorithm and we run it at every port in the network. When a new port is visited, we save the current execution state (all variables and all constraints). When the same port is revisited, the current state of each variable is compared to *all its previous states*.

Figure 6 shows how the current state compares to the old ones. A loop exists only when the new state contains all possible values in the old state (case (d) in the figure). Say the set of all constraints for the old state is o, and for the new state is n. To check for loops, we invoke the solver with the constraint `!n & o`. In Fig.6, this constraint asks the solver to find a point included in the old state that is not contained in the new state. If the solver returns an example, there is no loop, otherwise a loop exists. In our figure, the solver will find counter-examples for cases (a)-(c), but not for (d).

The loop detection algorithm is generic and can capture different kinds of loops. If we apply it to the entire state, the algorithm will not capture traditional forwarding loops because the TTL field will always decrease and thus the state will be different. To capture such loops, we must apply the

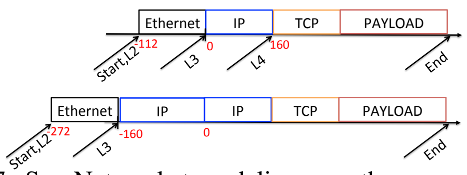


Figure 7: SymNet packet modeling uses the same physical layout as real packets.

same algorithm but only consider destination and source IP addresses when comparing states.

SEFL models (programs) are inherently bounded in space - there is no recursion or heap allocation. We have also proven that all SEFL programs have bounded execution. Loops can only appear as a result of the network topology, but are captured by our loop detection algorithm.

Tracking changes. By checking the value stack of the destination address field on output port 2, we find that it is bound to the same symbolic value that was set when the path entered the input port. In Fig. 5 the `TcpDst` and `IpDst` fields are unchanged as long as `TcpDst!=123`.

Header visibility. By analyzing the value stack of a header field at an intermediate point, we can understand whether the value read is the same as that set by the source or seen by the destination. Visibility tests allow us to check whether firewalls and endhosts see the same headers.

Header memory safety. When creating or destroying header fields, accesses are indexed through tags. If the tags are set incorrectly, of if the program wrongly assumes the location of headers, the execution path will fail. By accessing header fields, SEFL programs implicitly validate that the packet contains fields at the expected offsets, and that the header is indeed the expected one; otherwise Symnet with generate errors and the execution path will fail. This has allowed us to catch various encapsulation problems in buggy models, and a VLAN encapsulation bug described in [18].

7. MODELING NETWORKS WITH SEFL

To make static analysis tractable, we do not verify actual protocol implementations and rely, as all previous works, on data plane models that are easier to check. There is, however, no guarantee that the model is a faithful representation of the real functionality. If the two differ, the results of static analysis may not be useful.

We recognize this gap and aim to bridge it in two ways. First, we have developed an automated testing framework that checks the model against the real implementation (see §8.4) that has helped us to catch many bugs in our models. Secondly, we have quantified the properties that can be captured by traditional symbolic execution on real code and Symnet on a model of TCP options parsing in 8.3.

In the remainder of this section, we discuss a series of increasingly complex models to highlight the power and usability of SEFL. We begin with more detail on how we model packets in Figure 7. At the top of the Figure a TCP packet is encapsulated with IP and Ethernet headers. Packets always have the Start and End tags set; all the other tags are set as packets move through the (modeled) stack. Layer tags are always allocated relative to other tags; the start and end tags start at 0 by convention when a symbolic packet is created. The bottom packet in Figure 7 is an IP-in-IP encapsulated packet. The L4 tag is not set in this case; this will be set only in the IP decapsulation code. Any accesses to L4 fields before the L4 tag will fail, stopping the associated path. SEFL network models support protocol layering natively, shielding lower layer models from semantics higher up the stack.

The code below models a packet received from an Ethernet network interface; it first sets the L2 tag and only allows IP packets destined for a certain MAC address. Once the L2 tag is set, the L3 tag can be set by adding 112 bits to it:

```
InputPort(0):
  CreateTag("L2",Tag("Start"))
  Constrain(EtherDst,==00:aa:00:aa:00:aa)
  Constrain(EtherProto,==0x0800)
  CreateTag("L3",Tag("L2")+112)
  Forward(OutputPort(1))
```

Modeling switch behaviour. To model switches, we have written a parser that takes a snapshot of CISCO switch MAC tables (containing MAC, VLAN and output port tuples) and creates a SEFL model. To reduce branching, our model simply groups all MAC addresses that should be forwarded on the same output port in a single constraint. A possible switch model is the one below which we call "ingress" because the filtering code is applied on the input ports:

```
InputPort(*):
  If (Constrain(EtherDst==MAC11 |
          EtherDst==MAC12 | ... ),
     Forward(OutputPort(1)),
     If (Constrain(EtherDst==MAC21 |
           EtherDst==MAC22 | ... ),
           Forward(OutputPort(2)),
           Fail("Mac unknown")))
```

When we run the code above with a symbolic `EtherDst`, it will result in as many execution paths as the number of output ports of the switch, which is optimal. However, it will generate more constraints than needed: a path taking an `else` branch will include the negated constraints for the `if` branch together with the constraints for the current output port. To avoid these additional constraints, we can write the switch using an *egress* filtering model:

```
InputPort(*):
    Fork(OutputPort(1),OutputPort(2),...)
OutputPort(1):
    Constrain(EtherDst==MAC11 | MAC12 ...)
OutputPort(2):
    Constrain(EtherDst==MAC21 | MAC22 ...)
```

The egress model has both optimal branching and a minimum number of constraints. The egress models is correct as long as the constraints are mutually exclusive, which always holds for MAC tables due to the spanning tree algorithm.

Modeling an IP Router. At first sight, it seems we should be able to use the same approach to model an IP router, but this is not true in the following forwarding table:

Prefix		Output Interface
192.168.0.1/32	→	`If0`
10.0.0.0/8	→	`If0`
192.168.0.0/24	→	`If1`
10.10.0.1/32	→	`If1`

If we simply group the rules per output interface and apply them using `If` instructions in the order above, the resulting forwarding will not use longest prefix match for destination address 10.10.0.1, which will be forwarded wrongly on `If0`. The most obvious solution is to have one `If` instruction for each prefix and ensure that for all overlapping prefixes, more specific matches are checked first. However, this creates as many branches as the number of prefixes in the routing table. In our example we would have four branches, but for core routers this means hundreds of thousands of branches.

A better algorithm is the following. If prefix *a* is more specific than prefix *b*, create the following constraint for `b`: `!a & b`. This ensures that the more specific prefix `a` does not match. We can now group all rules that have the same output interface as in the switch case; the number of resulting paths drops from the number of prefixes to the number of links of the router, which is again optimal. We also rely on egress filtering to reduce the number of constraints.

Modeling a Network Address Translator. NATs are ubiquiously deployed as operators come to grips with the IPv4 address space shortage. NATs modify the source IP address and source port for outgoing packets and apply the reverse mapping for incoming packets. NATs are harder to model: they keep per flow state to ensure incoming traffic is only allowed if it is related to outgoing traffic the NAT has seen. In addition, the list of available ports at a NAT is a global variable, and the port assigned to a new connection will depend on many external factors, such as the number of active connections, the random number generator, and so forth.

To model the NAT we observe that the exact port number assigned by the NAT is quasi-random, and network operators treat it as such. Therefore it makes no sense to model the algorithm used to choose a port for a new connection; this would simply not scale. Instead, the newly mapped port will be a symbolic variable with allowed values in the NAT's port range. To make the NAT "remember" a mapping for the flow, we save the NAT state using metadata:

```
InputPort(0):
  Constrain(IPProto,==6) //only do TCP
  Allocate("orig-ip",32,local)
  Allocate("orig-port",16,local)
  Allocate("new-ip",32,local)
  Allocate("new-port",16,local)
  Assign("orig-ip",IpSrc) //save initial addr
  Assign("orig-port",TCPSrc) //save initial port
  Assign(IpSrc,"...")        //perform mapping
  Assign(TcpSrc, SymbolicValue())
  Assign("new-ip",IpSrc) //save assigned addr
  Assign("new-port",TcpSrc) //save assigned port
  Forward(OutputPort(0))
```

On the return path, the code restores the original mappings only if the metadata is present and matches the mapping the NAT has assigned to this flow:

```
InputPort(1):
   Constrain(IPProto,==6)
   Constrain(IpDst,=="new-ip")
   Constrain(TcpDst,=="new-port")
   Assign(IpDst,"orig-ip")
   Assign(TcpDst, "orig-port")
   Forward(OutputPort(1))
```

The NAT uses local metadata to ensure that multiple instances of the code can be run cascaded. Local metadata will ensure each NAT instance stores and retrieves its own values. Our NAT does not create any branches - the return packet is allowed if it contains the mapping, or dropped otherwise. The NAT code is a faithful model of the real thing.

The technique we used to model the NAT—storing per flow state inside the packet—we also used to model other similar boxes including stateful firewalls and firewalls that randomize the initial sequence number of TCP connections. The same technique can be applied wherever the per-flow state is independent across flows. Under this (admittedly strong) assumption, symbolic execution can verify large networks with stateful middleboxes without state explosion.

Modeling TCP options parsing. Our models for routers and switches are exact—there is no simplification compared to the real code as our model accurately mimics the behaviour of the code. To make the options parsing code symbolic-execution friendly though, we need to simplify it. The main problem with that code is that it includes a for loop with an unknown number of iterations, and the code has branches in the loop body. Simplifying the code means we cannot capture all the properties of the original code.

To understand the difference in properties that can be proven on the original C code and our SEFL model, we have performed a thorough analysis in §8.3. We find that traditional symbolic execution can prove memory safety and bounded execution on the real code, albeit only on a small subset of inputs. When answering higher level questions such as "which options are allowed through?", the answers can be wrong because only a small part of the options field can be checked in reasonable time.

We create a model of the options parsing code that is amenable to fast symbolic execution. It is based on the observation that the order in which options are placed in the options field does not matter —the code allows known options and strips everything else. This suggests that we can modify the list data structure with SymNet's in-built map. More concretely, each possible TCP option *x* (where x is in between 2 and 255) will have a corresponding metadata variable called "OPTx" that can take values 1 or 0, modeling whether that TCP option is enabled or not.The option length and body will be held in metadata variables "SIZEx" and "VALx" respectively. In a sense, our model pre-parses the byte representation of the options and stores it in the packet metadata, allowing quick access to the options.

A snippet of the firewall options code is given in Figure 8. Stripping options is simply a matter of setting the associated metadata to 0, regardless of the initial value—hence there is no branching involved. The SACK_OK is stripped only for HTTP traffic. The code then always sets the MSS option, and rewrites its value to be at most 1380.

Our model can accurately tell which options are allowed through and under what circumstances. The code has few branches and is thus cheap to symbolically execute, and captures more properties than Klee on the C code (see §8.3).

Modeling Encryption. Encrypted tunnels are being deployed

```
Assign("OPT30",ConstantValue(0)),
If(Constrain(TcpDst==80),Assign("OPT4",0),NoOp),
Assign("OPT2",1),
Assign("SIZE2",4),
If(Constrain("VAL2">1380),
    Assign("VAL2",1380),
    NoOp)
```

Figure 8: ASA options parsing code modeled in SEFL.

more and more. We need to capture two properties: first, no network box can read the original contents of the payload once it is encrypted. Secondly, if we decrypt using the same key that was used for encryption, we will retrieve the original payload. As in the NAT case, predicting the way the ciphertext will look is not important for our model. All that matters is that the original content is not available after encryption. We could use the following code snippet to encrypt with key K, where K is a parameter.

```
InputPort(0):
  Allocate("Key")
  Assign("Key",K)
  Allocate(TcpPayload)
  Assign(TcpPayload,SymbolicVariable)
  Forward(OutputPort(0))
```

The decryption only proceeds if the key matches:

```
InputPort(0):
  Constrain("Key",==K)
  Deallocate(TcpPayload)
  Forward(OutputPort(0))
```

Despite its simplicity, the code above has the two properties we seek. Any box reading the TCP payload after encryption will only see a novel unbounded symbolic variable, not the original contents. Only using the proper decryption key will retrieve the original contents.

7.1 Ready-made network models

Writing models in SEFL requires expert input. It cannot be reasonably assumed that network administrators have the time or the expertise needed to perform this task, yet they are the main beneficiaries of the tool. To make SymNet easily usable, we have created parsers that take configuration parameters and/or runtime information from well known network elements and output corresponding SEFL models. All the user has to do is place all these files in a single directory, together with a file describing the links between the boxes. Then, the user can run SymNet by specifying an input port to start the reachability and loop detection analysis.

The output of the tool is the list of explored paths in json format. For every path SymNet lists all variables and their constraints at the end of the execution as well as all the instructions and ports this path has visited. The user can check these paths either manually or with standard tools (e.g. grep) to see if certain undesired paths have been visited, etc.

Switches and routers. SymNet generates models from snapshots of switch MAC or router forwarding tables.

Click modular router configurations. We have modeled in SEFL a large subset of the elements of the Click modular router. This exercise has served two main purposes: first, it allowed us to understand whether SEFL's limited instruction set is sufficient to model a wide range of functionality. Second, we use the Click elements to build more complex boxes such as firewalls, NATs and even a CISCO application security appliance. Our parser takes the Click configuration file, generates a model for each individual element and then connects these models according to the config file.

Openstack Neutron configurations. Openstack Neutron allows cloud tenants to specify at an abstract level the networking configuration of their VMs before they are instantiated. Users can specify firewalls, router configurations, virtual links, etc. We have written an Openstack plugin that takes the router and firewall configurations and translates them into SEFL models. These could be used to check reachability before deployment, or to check that the actual deployment matches the user's intent; we are still working on integrating the results from symbolic execution back into Neutron to make it easily available to the users.

7.2 Modeling a CISCO ASA

To analyze our department's network, we must model its core device: a Cisco ASA (Adaptive Security Appliance) 5510 firewall, henceforth called ASA. It combines basic layer-2 capabilities such as switching and VLAN segmentation, with static & dynamic NAT and stateful packet inspection and filtering. How the latter is achieved, and the internal cuisine of the inspection process is the major challenge in accurately modeling ASA behavior. For instance, by default, the ASA will intercept TCP connections, and act as server until the connection is actually established. This feature protects machines behind the ASA from TCP SYN floods. However, documentation is very sparse and generic, making it difficult to understand the exact behaviour.

To understand how the ASA processes traffic, we developed a "black-box" testing environment, using the same ASA model. We connected the real ASA to two machines, generically termed *outside* and *inside*. Using Click, we ran test sequences consisting in TCP, UDP, ICMP and simple (raw) IP packets, through the ASA and observed the outputs.

The next step is to generate the model. We could generate SEFL directly, however SEFL is not executable in practice. We chose to generate a Click configuration that models the ASA instead. SymNet can execute this configuration because it has models for each element; the bonus of Click modeling is that we can potentially run the ASA in software, by simply instantiating it. After weeks of black box testing, we implemented a tool that parses the ASA configuration file and generates a Click ASA model automatically. We experimented with different ASA configurations produced by our tool, running the same test sequences through the real ASA, as well as the Click model. The default ASA configuration includes static and dynamic NAT, traffic filtering and basic TCP protocol inspection. Our model captures layer 2 and 3 behaviour, NATs. The Click configuration uses standard elements, with the exception of a new element called TCPOptions which implements the options filtering code.

The resulting (simplified) Click packet pipeline can be summarized as follows: (i) *ingress static nat*: if the packet matches an incoming static NAT rule, then it is modified accordingly; no state is preserved for such packets, (ii) *TCP in-*

spection: if the packet is the response of an active TCP connection, it is forwarded to destination directly, and appropriate dynamic NAT rules applied, if this is the case; otherwise (iii) *filtering*: appropriate filtering rules are applied; (iv) TCP connections are stored, and dynamic NAT mappings are inserted, for TCP packets; finally (v) *egress static nat*: if the packet matches an outgoing static NAT rule, then it is applied. Finally, all TCP packets are parsed by the TCPOptions element with code very similar to that in Fig.2.

8. EVALUATION

Our evaluation has two major parts. First, we want to understand whether our approach tailoring network models for symbolic execution pays dividends in terms of runtime and accuracy. Secondly, we wish to understand whether symbolic execution can help us find new types of bugs not captured by existing tools such as HSA or NOD. We explore two network deployments: our department's network containing a Cisco middlebox, one router and 15 switches and an enterprise middlebox deployment [18].

8.1 Performance evaluation

We ran experiments using our SymNet prototype on a quad-core Intel i5 machine with 8GB of RAM, testing models of network functionalities such as routers, switches, firewalls, options parsing and Click elements.

Symbolic execution of a switch model. We used a snapshot of the MAC table of the core switch in our department network (see Fig 12) with four hundred entries to model its behaviour. We built three models of the switch:

- **Basic**: a lookup table, with one entry per MAC, applied on ingress. This is the same as running a generic symbolic execution tool on switch forwarding code.
- **Ingress**: group MACs going to the same output port, apply filtering and take switch decision on input.
- **Egress**: fork traffic to all outgoing ports and apply per-port restrictions on egress.

We inject a packet with a symbolic destination MAC address and execute the switch code and forward all the resulting paths on the corresponding output ports. We measure wall-clock execution time and the number of paths, time spent in and number of calls to the constraint solver.

Figure 9 plots the SymNet runtime as we increase the number of MACs from 440 to 500,000.To generate more entries in the MAC table, we duplicate existing entries as many times as needed; each entry gets a unique destination MAC address. In all tests, more than 90% of time is spent in Z3.

The basic model generates as many paths as entries in the lookup table. Each path will have its own instance of the solver and a set of constraints, and this has a large memory overhead: it takes 10 seconds to execute a 1000-entry MAC table, and close to 8GB of RAM. Beyond this size, the RAM on our machine is insufficient.

The ingress model groups MAC addresses by output port, and results in as many paths as output ports in use on our switch (20). However, the path constraints are quite complex: the first path will contain the allowed MACs, the second path will contain its own allowed MACs and the negated constraints on the first path, and so forth. In fact, the total number of constraints grows quadratically with the number of switch ports: it takes 2 minutes to symbolically execute a switch code with 480,000 entries.

The egress model also groups MACs per output port, but avoids the negation by forking the initial flow and applying the constraints independently on each port. In this case the total number of constraints is the same as the number of entries, and this gives a big benefit in runtime: it takes only 5s to execute 480 thousand entries. This is the switch model we use in the remainder of our evaluation.

Router. We used a publicly available snapshot of the forwarding table of a core router containing 188.500 entries [12]. We generated the model by checking for overlapping prefixes and adding constraints to ensure the per-port constraints are mutually exclusive: this results in 183000 additional constraints to a total of 371.000 prefix checks (or 722.000 inequalities). Model generation takes 8 minutes.

We run tests with 1%, 33% and 100% of the prefixes by injecting an IP packet with a symbolic destination IP address into the three models: basic, ingress and egress. We provide the results in Table 2. The basic model only copes with 1% of prefixes; the ingress one also works on 33% of the prefixes but not 100%. The egress model is faster than both, and can also symbolically execute the full router in around 18s.

Performance comparison to existing tools. We seek to understand how SymNet performance compares to HSA, the most efficient static analysis tool today. We use the Stanford backbone network data [14] and run reachability from an access router to all core routers with both SymNet and HSA. The results are given in table 3 and show that SymNet is within 50% of the execution time of HSA, despite its power. In comparison, NOD is reported to be 20 times slower than HSA [19] on the same benchmark.

8.2 Accuracy of header validation

Given a purely symbolic packet of a certain size, is it possible to be interpreted as different header combinations? In our example in Fig. 1, can the given packet be mistaken as TCP or IP? To answer this question we generate symbolic packets containing up to five of the following headers in random order: Ethernet (including VLAN tags), MPLS, IPv4, IPv6 (with or without options), IPinIP, VXLAN, GRE (with or without options), TCP, UDP and SCTP.

We then check whether the resulting packet can be interpreted by SymNet in more than one ways. As SymNet checks all header accesses (an error is thrown if there is no header field allocated that starts at the given offset) and different headers have fields at different offsets, it seems unlikely that a different combination of headers will be valid. We generated approx. 400.000 possible packets, and found a single one inaccuracy: SymNet mistakes GRE+SCTP for a GRE header with options.

We performed a similar analysis for HSA, checking only the length of the entire header as means of validation. We found more than 2000 of pairs of headers having the same

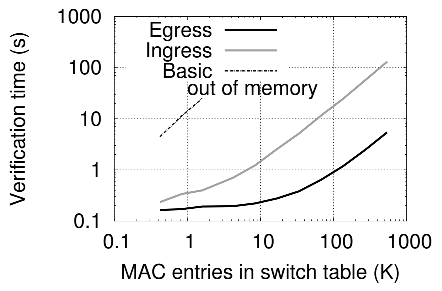


Figure 9: Symbolic execution of different switch models.

Prefixes	Basic	Ingress	Egress
1600	25s	2.1s	0.4s
62500	DNF	23.1s	5.6s
188500	DNF	DNF	18s

Table 2: Core router analysis.

	HSA	SymNet
Generation Time	3.2min	8.1min
Runtime	24s	37s

Table 3: Comparison to HSA

Property	Klee	SymNet
Runtime	1h	1s
Bounded execution	yes upto 6B	no
Memory safety	yes upto 6B	no
Invalid Length	yes upto 6B	less precise
SackOK,MSS,WScale	yes upto 6B	yes
Timestamp,Multipath	incorrect	yes
Combinations of options	incorrect	yes

Table 4: Comparison between Klee and SymNet on TCP options firewall code.

number of bits but different protocol encapsulations. We also evaluated the accuracy of NOD by using the total number of header fields for validation. We found more than 1000 of header combinations having the same number of header fields but different protocol encapsulations.

8.3 Coverage analysis of TCP options code

What properties can Klee symbolic execution capture on ASA options parsing C code, and what can SymNet do on the SEFL model? We ran experiments by creating a TCP packet with a symbolic options field and the length field set to a concrete value. We then process this packet with the options parsing code. Finally, we iterate the options field afterwards, printing a message if a specific TCP option is present. A list of the properties captured by Klee (which we stop after one hour) and SymNet is given in Table 4.

Table 1 in §2 shows that Klee symbolic execution on the C code takes more than 30 mins for only 6B of options. When options length is less than or equal to six, Klee proves that the parsing code is memory safe, guaranteed to terminate, and correctly accounts for options with invalid length. It also shows that the MSS, SackOK and WScale options are allowed alone or pairwise but not all three simultaneously. Klees reports that timestamp and multipath options are not allowed. Unfortunately, these last results are wrong when we allow full-size options: the timestamp option is allowed through when the options field is large enough to contain it. To conclude, Klee testing can give wrong results when used on small options fields, and can't be run on large ones.

SymNet runs the options code in 1s. It cannot capture memory safety and bounded execution properties of the C code, but ensures the model itself is memory safe and runs in bounded time by construction. It correctly captures all other properties, showing that the multipath option is always stripped, the MSS option is always added even if it is not present in the original packet, and that all allowed options are permitted in any combination.

SymNet achieves these results by ignoring the ordering of options and by using an abstract representation. However, ordering matters in the options parsing code: when an option has invalid format (e.g. wrong size), the code will replace all remaining bytes with NOPs, thus stripping options stored after the invalid one. The SymNet model simply marks *all existing options* in the packet as possibly removed by setting their corresponding OPT variables to new symbolic values.

8.4 Automated testing

SEFL models can be checked quickly, but they are only useful as long as they accurately reflect the processing performed by the baseline code they mimic. As modeling is manual, inadvertent errors may be introduced.

To catch such bugs, we have developed an automated testing framework that compares the model to the actual implementation (be it a Click configuration or a hardware appliance). Our automated tool is similar in principle to ATPG [24] and proceeds in the following steps:

1. We run a reachability test over the SEFL model, with a TCP/IP packet with symbolic fields. The output is a series of paths, where each path places a number of constraints on the header fields of the injected packet.
2. Pick an unexplored execution path and use Z3 and the path constraints to generate concrete values for all the header fields, resulting in a concrete packet p.
3. Packet p is injected into the running code (either the Click modular router instance or the hardware ASA box). The outputs are captured with tcpdump. We use a 1s timeout for tcpdump—if no packets are received, we conclude there is no packet reachable on output.
4. The header values from the captured packet are added as constraints at the end of the symbolic execution path. Z3 is used to check if the constraints hold; if they do not, an error report is generated.
5. Repeat from step 2, as long as there are unexplored symbolic execution paths.
6. Generate random header fields and repeat from step 3.

Our testing procedure first explores all paths resulting from symbolic execution, and then tests random inputs until the user stops the testing procedure. We deployed our tool on three machines in our local testbed: one machine generates packets according to specification, sends them to the middle machine running the code under test which sends its output to a third machine where packets are captured. The middle machine ran Click modular router configurations for most of our tests, with one exception: when testing the ASA box we replaced the middle machine with real ASA hardware configured the same way as our department's firewall. We have used testing extensively while writing our models, and it has helped us catch a series of interesting bugs, discussed below.

IPMirror. This Click element mirrors the IP source and destination addresses and transport level ports. Our model was incomplete: it only mirrored the IP addresses and not ports.

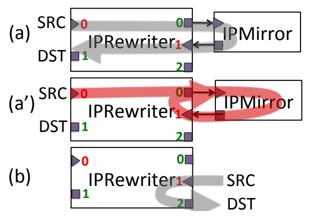


Figure 10: Testing a stateful firewall uncovered a cycle.

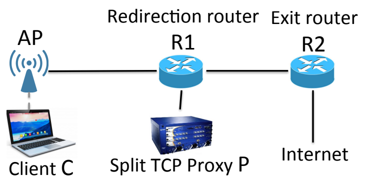


Figure 11: Split TCP Deployment, sideband mode [18].

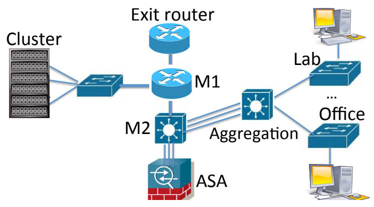


Figure 12: CS Department network modeled in SEFL and verified with SymNet

DecIPTTL. This element decrements the IP TTL and drops packets with TTL 0. The original code decreased the TTL and then constrained its value to be positive. SymNet reported a single execution path instead of the two we expected. This was a bug because the TTL is an unsigned value: when the TTL was 0 decrementing it would result in wrap-around to the higher possible TTL value, and the packet would never be dropped. The obvious fix was to place the constraint $TTL \geq 1$ first, then decrement the TTL.

HostEtherFilter only allows packets destined to a MAC address, and we were wrongly checking the ethertype field.

IPClassifier takes an input packets and forwards them on one of the output ports if the packet matching the corresponding filter. To test the classifier, we generated multiple instances using different filters and different number of output ports. We then took each instance and tested it, one output port at a time, discarding all packets from the other output ports. We found a bug where the solver was generating 0 values for several header fields (e.g. port number) and these where correctly dropped by Click. To solve this issue, we had to constrain our symbolic packet to ensure IP addresses and port numbers were valid.

IPRewriter is the Click element that allows the implementation of stateful functionality such as NATs and stateful firewalls. In our setup, the element acts as a stateful firewall where traffic from the inside network arrives on input port 0 and outside traffic arrives on input port 1 (Fig. 10).

To test the rewriter using our unidirectional testing setup we use the setup shown in Figure 10 to testing connections initiated from the inside in setup (a) or the outside in setup (b). In setup (a) we model return traffic by bouncing traffic via an IPMirror element. On output port 0 we should observe a packet with reversed IP addresses and transport ports.

When we ran SymNet we found the loop shown in Fig.10.(a'): with symbolic packets it is possible that the source and destination addresses and ports are identical and the traffic returned from the IPMirror matches the forward mapping and exits again on output port 0. The solution was to constrain the source and destination IPs to be different.

ASA model. We tested the ASA model against the ASA hardware extensively, uncovering several bugs in the process. A bug in the VLAN and ethernet decapsulation code resulted in all packets being dropped at ASA ingress. Secondly, we found that the ASA correctly allowed traffic from the higher-priority office VLAN to the lab VLAN, but wrongly dropped the return traffic. We fixed it by enhancing the ASA to behave as a stateful firewall for office to lab traffic.

8.5 Functional Evaluation

What type of properties beyond basic IP or layer two reachability can SymNet verify that are useful in practice? To begin answering this question, we turn to recent work by Le et al. that describes operational experiences learned while deploying a Split TCP middlebox in ten enterprise networks serving thousands of users [18]. We have modeled in SEFL the network topology that Split TCP uses (Figure 11). The Split TCP Proxy is deployed adjacent to router R3 which is configured to redirect traffic coming from both directions to P by rewriting the destination MAC address of the packet. Our model faithfully mimics packet processing along the whole path, including Ethernet header encapsulation and decapsulation at each hop, routing and filtering. Can SymNet discover issues that appeared in production deployment?

Asymmetric routing. We run a reachability check from C to R2, and at R2 we use IPMirror to send the traffic back to C. SymNet shows that all execution paths from C to R2 and reverse cross via P, thus the setup is correct.

MTU issues. Router R1 is configured to drop all packets with size large than 1536B. We inject a symbolic packet at C with a symbolic IP length field. At R2, the IP length field has a constraint attached: $length < 1536$.

Next, we use IP-in-IP tunneling for traffic between R1 and P. This further reduces the available MTU, and was creating difficult to debug performance problems in the actual deployment: ping and TCP connection setup worked fine, but subsequent full MTU traffic from the client was blackholed because they exceed the MTU after encapsulation [18]. Running reachability on this new setup, the new constraint applied to length becomes: $length + 20 < 1536$: the client MTU must be smaller than 1516B.

Missing VLAN tagging. In one setup, P was removing VLAN tags before processing packets, and was not adding them back before pushing packets back ro R1. This caused R1 to drop those packets because it was expecting VLAN tagging [18]. A simple reachability check quickly highlights this problem: when R1 attempts to remove the VLAN tagging it finds the wrong EtherType and drops the packet. Neither NOD nor HSA capture MTU and VLAN tagging issues.

Security Appliance. In one deployment, R2 acted as a DHCP server too, and it filtered packets where the Ethernet source address, IP source address tuple was not in its assigned leases. We modeled the DHCP assignment by using two metadata variables set by C: `origIP` and `origEther`. Both were set by the source to have the same symbolic value as the Ethernet and IP source address fields in the symbolic packet. R2 filters all packets where `origIP!=pSrc` or `origEther != EtherSrc`. We ran reachability again finding all packets were dropped by R2 because the source MAC was being modified by P and the second constraint didn't hold.

8.6 Verifying the CS Department Network

We have generated an accurate model of our department's network using switch MAC tables, router forwarding tables and a Click configuration mimicking the ASA box. The slightly simplified topology is given in Figure 12: all hosts are connected to local switches which connect to an aggregation switch. The aggregation switch connects to M2 master switch that connects to a Cisco ASA box and M1, the CS department router. L2 routing is used in most of the network: the access switches tag the traffic coming from the hosts as lab traffic (VLAN 304) or office traffic (VLAN 302). These are carried on trunk links all the way to the ASA box which is the first IP hop. A management VLAN is also configured with all switches having an interface in it. A single server called "hole" and located in the cluster has an interface in this VLAN and is used by our admin for management purposes. The network has 21 devices with 235 connected network ports. The combined MAC tables have 6000 entries, and there are 400 routing table entries.

Injecting a purely symbolic packet in the office takes 42s and results in 3000 paths: results show that the Internet and the labs VLAN are reachable via the ASA box. If we specialize the packet to have a source address in the office VLAN and a destination address in the Internet the number of valid paths drops to 50 and execution takes just 10 seconds. Next, we check TCP connectivity by further specializing the packet (IPProto field is set to 6) and adding an IPMirror element to the exit router port; this takes 1s and results in 8 valid paths.

TCP reachability is allowed, but the output shows that TCP options are tampered with: SACK is disabled for HTTP traffic (OPT5 is set to 0), and MPTCP options are removed. Our admin had no idea of this behaviour, which was enabled by the default ASA configuration. TCP options parsing is not captured by existing tools.

Next, we checked inbound reachability by injecting a purely symbolic packet at the exit router. This took 2s and resulted in 221 paths, out of which 30 were successful. Most paths were ended at the ASA box which appears to be configured correctly. However, symbolic execution showed that the management VLAN, with private *192.168.137.0/24* addresses, was accessible via router M1. A quick check on the live router showed the behaviour was true, but is it exploitable? Our ISP does not forward such traffic towards to our network. We ran another reachability test from the cluster and found that any machine from the cluster can access any of the switches, and verified manually that from the cluster we could indeed telnet into all the switches. As all of our students have cluster accounts, this is a major security risk.

9. RELATED WORK

Static network analysis is a well-established topic, with many available tools [23, 14, 20, 19, 21]. AntEater [20] models network boxes as boolean formulae.Network Optimized Datalog [19] is the most complete tool to date and relies on Datalog both for network models and policy constraints. The work of Panda et al. uses a model checker to verify networks containing stateful middleboxes [21]. Our NAT model is similar in spirit with their proposal.

		HSA	Ant-Eater	NOD	Panda	Sym-Net
Scalability		high	low	med	low	high
Coverage	Reachability	✓	✓	✓	✓	✓
	Header changes	✗	✓	✓	✓	✓
	Header visibility	✗	✓	✓	✓	✓
	Memory correctness	✗	✗	✗	✗	✓
Modeling	Model independence	✓	✓	✗	✓	✓
	Model language		Declarative	Declarative	Declarative	Imperative
Expressiveness	IP router	✓	✓	✓	✓	✓
	Dynamic tunneling	✗	✗	✗	✗	✓
	TCP options	✗	✗	✓	✗	✓
	Dynamic NATs	✗	✗	✓	✓	✓
	Encryption	✗	✗	✗	✗	✓
	TCP Segment splitting	✗	✗	✗	✗	✗
	IP Fragmentation	✗	✗	✗	✗	✗

Table 5: SymNet vs. other network verification tools.

In table 5 we provide a qualitative comparison of the most relevant network verification tools. We categorize the scalability of the tools by analyzing their runtime on enterprise-sized networks, as reported in the original papers. High scalability means runtimes of seconds, medium is minutes, while low means runtimes of hours. HSA scales very well but it does not capture many properties. SymNet scales very well on our optimized models; in general, though, its complexity can quickly run out of control if the models verified are poorly written or have inherently many branches.

Coverage examines the properties captured by the different tools; here, memory correctness is a differentiating feature of SymNet. Ease of modeling compares the task of generating models for the different tools. SEFL is the only imperative language and should thus be more familiar to and easiest to use by network professionals. Model independence means a box's model is independent of its location in the network. NOD is the only tool that doesn't have this property, which makes modeling very difficult especially in heterogeneous networks. Finally, we analyze the expressiveness of the different tools by checking their ability to analyze widely-deployed network functionalities. SymNet has the biggest coverage, however it cannot model packet splitting or coalescing (see our discussion of limitations in §10).

Symbolic execution. We are not the first to propose using symbolic execution to analyze networks. Dobrescu et al. [6] used symbolic execution to check selected Click elements' source code for bugs, aiming to prove crash-freedom and bounded execution. We have shown that using C as modeling language does not scale, and have proposed SEFL and SymNet as scalable alternatives.

Online verification. Veriflow [15] and NetPlumber [13] aim to perform live validation of all network configuration changes. They work underneath an SDN controller and verify all state updates. NICE uses symbolic model checking to verify the correctness of Openflow programs [4]. More recently, Buzz [7] uses Klee on middlebox models written in C to guide the generation of test packets for networks. SymNet is orthogonal to these works.

NetKAT[1] and Frenetic [10] are novel specification languages optimized for specifying OpenFlow-like rules in networks. SEFL is strictly more general as it can model middlebox behaviours too, not just layer two behaviour.

10. LIMITATIONS

Using SymNet for network analysis is powerful but has a few notable limitations which we discuss here. Packet processing is sequential, and the network boxes only do processing when they receive packets. This problem is not unique to SymNet: it is a general limitation of applying symbolic execution to network analysis. Parallel processing is akin to symbolic execution of multi-threaded programs, and is significantly harder because it must check all possible interleavings of the different threads [16].

A single packet is active in the network at any one time. This implies that processing that works across multiple packets, such as TCP segment splitting or coalescing, cannot be modeled. This only limits the number of in-flight packets, not the number of total packets: our TCP sender model works with a window of one packet because of this limitation, but any number of round-trip times can be simulated.

11. CONCLUSIONS

Symbolic execution is a powerful tool for network verification, but applying it to production networks is challenging.To allow scalable network symbolic execution we have proposed SEFL, a novel, minimalist, imperative language tailored by design for network symbolic execution. We have built SymNet, a fast symbolic execution tool for SEFL code.

To understand the expressiveness of SEFL, we have modeled many networking devices ranging from switches and routers to middleboxes that keep flow state and parse TCP options. We have also modeled a large subset of the elements of the Click modular router, which allows us to verify Click configurations out-of-the box. Our evaluation shows that our models have near-optimal branching factors per box and that SymNet seamlessly scales to large networks.

Finally, we have used SymNet to capture a number of middlebox behaviours described in the literature and in our own department's network. Our experience shows that SymNet catches many interesting network properties and is fast.

12. ACKNOWLEDGMENTS

We would like to thank our shepherd Ratul Mahajan for his help and the anonymous reviewers for the useful reviews that helped us improve our paper. This work was part of the Superfluidity (H2020) and Trilogy 2 (FP7) projects funded by the European Commission.

13. REFERENCES

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *POPL'14*.

[2] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, July 2002.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI'08*.

[4] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *Proc. NSDI'12*.

[5] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. TACAS'08*.

[6] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proc. NSDI'14*, NSDI'14.

[7] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proc. NSDI 2016*.

[8] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In *NSDI*, 2005.

[9] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proc. ICFP '11*.

[11] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. SIGCOMM'05*.

[12] P. Kazemian. Hassel tool and public datasets. https://bitbucket.org/peymank/hassel-public/wiki/Home.

[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI'13*.

[14] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI'12*.

[15] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proc. NSDI'13*.

[16] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS'03*.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[18] F. Le, E. Nahum, V. Pappas, M. Touma, and D. Verma. Experiences deploying a transparent split-tcp middlebox in operational networks and the implications for nfv. HotMiddlebox'15, 2015.

[19] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI'15*.

[20] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Sigcomm*, 2011.

[21] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying Isolation Properties in the Presence of Middleboxes. Tech Report arXiv:1409.7687v1.

[22] J. Wagner, V. Kuznetsov, and G. Candea. Overify: Optimizing programs for fast verification. In *Proc. HotOS'13*.

[23] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings of Infocom*, 2005.

[24] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT'12*.