SymNet: Static Checking for Stateful Networks

Radu Stoenescu, Matei Popovici, Lorina Negreanu, Costin Raiciu

Department of Computer Science, University Politehnica of Bucharest
{firstname.lastname}@cs.pub.ro

ABSTRACT

Today's networks deploy many stateful procesing boxes ranging from NATs to firewalls and application optimizers: these boxes operate on packet flows, rather than individual packets. As more and more middleboxes are deployed, understanding their composition is becoming increasingly difficult. Static checking of network configurations is a promising approach to help understand whether a network is configured properly, but existing tools are limited as they only support stateless processing.

We propose to use symbolic execution—a technique prevalent in compilers—to check network properties more general than basic reachability. The key idea is to track the possible values for specified fields in the packet as it travels through a network. Each middlebox or router will impose constraints on certain fields of the packet via forwarding actions, packet modifications and filtering. The symbolic approach also allows us to model middlebox per-flow state in a scalable way.

We have implemented this technique in a tool we call SymNet and conducted preliminary evaluation. Early results show SymNet scales well and models basic stateful middleboxes, opening the possibility of analyzing complex stateful middlebox behaviours.

1. INTRODUCTION

Middleboxes have become nearly ubiquitous in the Internet because they make it easy to augment the network with security features and performance optimizations. Network function virtualization, a recent trend towards virtualizing middlebox functionality, will further accelerate middlebox deployments as it promises cheaper, scalable and easier to upgrade middleboxes.

The downside of this trend is increased complexity: middleboxes make today's networks difficult to operate and troubleshoot and hurt the evolution of the Internet by transforming even the design of simple protocol extensions into something resembling black art [5].

Static checking is a promising approach which helps understanding whether a network is configured properly. Unfortunately, existing tools such as HSA [2] are insufficient as they only focus on routers or assume all middleboxes are stateless. Checking packet forwarding alone only tells a part of the story because middleboxes can severely restrict reachability. The one common trait of most middleboxes is maintenance of per-flow state, and taking packet actions based on that state. Such middleboxes include network address translators, stateful firewalls, application-level proxies, WAN optimizers, traffic normalizers, and so on. Finally, existing tools only answer questions limited to packets; with stateful processing everywhere, we must also be able to answer questions about ***packet flows***.

In this paper we propose a new static analysis technique that can model stateful middleboxes and packet flows in a scalable way. Our solution stems from two key observations:

1. TCP endpoints and middleboxes can be viewed as parts of a distributed program, and packets can be modeled as variables that are updated by this program. This suggests that symbolic execution, an established technique in static analysis of code can be used to check networks.
2. Middleboxes keep both global and per-flow state. For instance, a NAT box will keep a list of free ports (global state) and a mapping for each flow to its assigned port (per-flow state). Modeling global state requires model-checking and does not scale. Flow-state, however, can be easily checked if we assume that flow-state creation is independent for different flows.

Starting from these two observations, we have built a tool called SymNet that statically checks network configurations by using symbolic execution and inserting flow state into packets. The tool allows answering different types of network questions, including:

- **Network configuration checking.** Is the network behaving as it should? Is the operator's policy obeyed? Does TCP get through?
- **Dynamic middlebox instantiation.** What happens if a middlebox is removed or added to the current network? How will the traffic change ?
- **Guiding protocol design.** Will a TCP extension work correctly in the presence of a stateful firewall randomizing initial sequence numbers?

SymNet is scalable as its complexity depends linearly on the size of the network. Our prototype implementation takes as input the routers and middleboxes in the network together with the physical topology. Each box is described by a Click [3] configuration. The prototype can check networks of hundreds of boxes in seconds.

We have used SymNet to check different network configurations, network configuration changes and interactions between endpoint protocol semantics and middleboxes. We discuss our findings in Section 5.

HotMiddlebox'13, December 9, 2013, Santa Barbara, CA, USA.

http://dx.doi.org/10.1145/2535828.2535835.

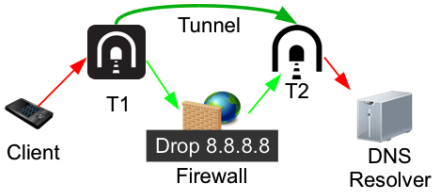


Figure 1: An example network containing a firewall and a tunnel

2. PROBLEM SPACE

Consider the example in Figure 1, where an operator deploys a firewall to prevent its customers from using public DNS resolvers. Clients, however, tunnel their traffic by routing it to T_1, which encrypts it and forwards it to T_2 that decrypts it and sends it onwards to its final destination. A number of interesting questions arise:

1. Is the firewall doing the job correctly?
2. Given the firewall configuration, which addresses are reachable via the tunnel, and via which protocols?
3. Is the payload modified en-route to the Internet?
4. What if the firewall is stateful? Do the answers to the above questions remain the same?

To address these issues, we require techniques that can model both *stateful middleboxes* as well as *TCP endpoints* and allow us to reason about the properties of different network configurations. Specifically, these techniques need to:

- **Determine the value of header fields** of a packet at different network ports. For instance, a packet accepted by the firewall should not be changed before it reaches the client.
- **Model flow state:** this allows capturing middlebox behaviour that is dependent on flow state. Such behaviour includes network address translators, stateful firewalls, tunnels, proxies and so forth.

Static analysis can be used to answer such questions. A prerequisite of static analysis is an accurate view of the network routing tables and middlebox functionalities in order to model them appropriately. With this information, it is possible to test IP reachability by tracking the possible values for IP source and destination addresses in packets, in different parts of the network [7]. However, [7] only models routers and stateless firewalls. Header Space Analysis (HSA) [2] is an extension of [7] which models arbitrary stateless middleboxes as functions which transform header spaces. The latter are multi-dimensional spaces where each dimension corresponds to a header bit.

HSA is not sufficient to answer our previously-stated questions for the following reasons: first it does not capture middlebox state, and thus cannot perform an accurate analysis on most networks as these contain NATs, stateful firewalls, DPI boxes, etc.

Second, while HSA is designed to determine what packets can reach a given destination, it is unable to efficiently examine how packets are changed by the network. Consider our example: answering the first question boils down to establishing whether the firewall reads the original destination address of the packet, as sent by the client. Given a fixed destination address d of a packet sent by the client, HSA is able to check if the firewall does indeed read d. However, it is unable to perform the same verification for an arbitrary (and a-priori unknown) destination address. Using HSA, this can only be achieved by doing reachability tests for each possible destination address d, which is exponential in the number of bits used to represent d.

Question 3 is the same as asking whether a packet can be modified by the tunnel. Assume any packet can arrive at the tunnel: this will be modelled by a headerspace containing only unknown bit values. As the packet enters the tunnel, HSA will capture the change to the outer header by setting the IP source and destination address bits to those of the tunnel endpoints; as the packet comes out, these are again replaced with unknown bit values. However, HSA is unable to infer that the latter unknown values coincide with the ones which entered the tunnel in the first place. Having a "don't know"[1] packet go in, and a similar packet go out says nothing about the actual change which occurs in the tunnel; The output is similar to that of a middlebox that randomly changes bits in the header: whenever the input is a "don't know" packet, the output is also a "don't know" packet.

AntEater [4] takes a different approach to static network modeling. It expresses desirable network properties using boolean expressions and then employs a SAT-solver to see if these properties hold. If they don't, AntEater will provide example packets that violate that property. AntEater does not model state either; additionally, its reliance on SAT-solvers makes it inapplicable to large networks.

A blend of static checking and packet injection is used to automatically create packets that exercise known rules, such as ATPG [8]. Another hybrid approach uses HSA to dynamically check the correctness of updates in SDN networks [1]. Both these tools inherit the drawbacks of HSA: they cannot model stateful middleboxes.

3. SYMBOLIC NETWORK ANALYSIS

SymNet relies on symbolic execution—a technique prevalent in compilers—to check network properties such as TCP connectivity between specified endpoints, the existence of loops, reachability of a certain node, etc.

The key idea underlying SymNet is to treat sets of possible packets as *symbolic packets* or *flows* and network devices as transformation functions or *rules*. As packets travel through the network, they are transformed much in the same way symbolic values are modified during the symbolic execution of a program. Flows consist of variables which are possibly bound to expressions. The latter have a two-fold usage. On one hand they model the header fields of a packet. On the other, they are used to keep track of the device state. Thus, stateful devices such as NATs, tunnels, (stateful) firewalls, proxies, etc. can be accurately modeled as transformations which operate both on header as well as on state variables. The set of reachable packets from a given source to a destination, as well as the detection of loops are achieved by computing the least fix-point of an operator which aggregates all transformation functions modeling the network. This is achieved in linear time, with respect to the network size.

Modeling packets. Following HSA (Headerspace Analysis) [2], we model the set of all possible headers having n fields as an n-dimensional space. A flow is a subset of such a

[1]packet header with unknown address field values

space. HSA assigns to each header bit one of the values *0*, *1*, *x*. The latter models an unknown bit value. Unlike HSA, we introduce symbolic expressions as possible values assigned to header fields. The most basic (and useful) such expression is the variable. Thus, we replace HSA assignments such as $bit_i = x$ by $bit_i = v$, where v is a variable[2]. Even without any additional information regarding the value of v, the latter assignment is more meaningful, as the unknown value of bit i can be properly referred (as v), and also used in other expressions. For instance, Question 1 from Section 2 can now be answered by assigning a variable v_d to the destination address set by the client, and checking if the incoming flow at the firewall also contains v_d as destination address.

Let $\mathbb{V}ars$ and *Expr* designate finite sets whose elements are called (header) *variables* and *expressions*, respectively. In this paper we only consider expressions generated by the following BNF grammar:

$$expr ::= c \mid v \mid \neg expr$$

where $c \in Expr$ and $v \in \mathbb{V}ars$. Let $C \subseteq \mathcal{P}(\mathbb{V}ars \times Expr)$. C models a *compact space* of packets. The set $U \subseteq \mathcal{P}(C)$ models an arbitrary space of packets, that is, a reunion of compact spaces. An element $f \in U$, $f \neq \emptyset$, models a *flow* on a given port from the network and is of the form $f = \{cf_1, \ldots, cf_n\}$ where each $cf_i \in C$ is a compact space. f models the subspace $cf_1 \cup cf_2 \ldots \cup cf_n$ of U. Whenever a flow is a compact space (i.e. it is of the form $\{cf\}$), we simply write cf instead of $\{cf\}$, in order to avoid using double brackets. We write $f|_{v=c}$ to refer to the flow obtained from f by enforcing that v has value c and $f|_v$ to refer to the flow obtained from f where v has no associated value.

Example 3.1 (Flow). *The flow* $f_1 = \{(port, p_1), (IP_{src}, 1.1.1.1), (TCP_{src}, v_1), (IP_{dst}, 2.2.2.2), (TCP_{dst}, 80)\}$ *models the set of all packets injected at port* p_1 *which are sent to an app running at device* 2.2.2.2 *on port* 80.

Modeling state. Unlike HSA, which models stateless networks only, SymNet can model stateful devices by treating per-flow state as additional dimensions in the packet header. State is not explicitly bound to devices, rather "pushed" in the flow itself. The simplest example is a stateful firewall that only allows outgoing connections: as the symbolic packet goes out to the Internet, the firewall pushes a new variable called *firewall-ok* into the packet. At the remote end, this variable is copied in the response packet. On the reverse path, the firewall only allows symbolic packets that contain the *firewall-ok* variable.

Such state modelling scales well, as it avoids the complexity explosion of model-checking techniques. However, our model makes a few strong assumptions. First, we assume each flow's state is independent of the other flows, so flow ordering that not matter—in the firewall example this is obviously true, as long as the firewall has enough memory to "remember" the flow. Second, we completely bypass global variables held by the middleboxes, and assume these do not (normally) affect flow state. Packet-counters and other statistics normally do not affect middlebox functionality, so this assumption should hold true.

[2]Also, variable-value pairs need not refer to individual bits of the header. For instance, an IP address *a.b.c.d* can be modeled in the standard way as a sequence of 32 bits, but also as a string "`a.b.c.d`". The representation choice is left to the modeler.

Modeling the network. The network is abstracted as a collection of *rules* which can be (i) matched by certain flows and (ii) whenever this is the case can modify the flow accordingly. Formally, a rule is a pair $r = (m, a)$ where $m : U \to \{0, 1\}$ decides whether the rule is applicable and $a : U \to U$ provides the transformation logic of the rule. We occasionally write $r(f)$ instead of $a(f)$ to refer to the application of rule r on flow f.

Example 3.2 (Rule, matching, application). *The rule* r_1 *takes any packet arriving at port* p_1 *and forwards it on port* p_2. r_1 *is applicable on* f_1 *and once applied, it produces the flow* $f_2 = f_1|_{port=p_2}$. *Formally,* $r_1 = (m_1, a_1)$, *where* $m_1(f) = 1$ *if* $(port, p_1) \in f$ *and* $m_1(f) = 0$ *otherwise, and* $a_1(f) = f|_{port=p_2}$.

The rule r_2 *models the behaviour of a NAT device: the source IP and TCP addresses are overridden by those of the NAT. Also, the flow will store the device state, i.e. the original IP and TCP addresses, using the (state) variables* IP_{nat} *and* TCP_{nat}. $r_2 = (m_2, a_2)$ *where* $m_2(f) = 1$ *is a constant function and:*

$$a_2(f) = f|_{IP_{nat}=f(IP_{src}), TCP_{nat}=f(TCP_{src}), IP_{src}=v_3, TCP_{src}=v_4}$$

In order to build up more complex rules from simpler ones, we introduce the *rule composition operator* $\circ$, defined as follows: $(m, a) \circ (m', a') = (m_c, a_c)$ where $m_c(f) = m(f) \wedge m'(a(f))$ and $a_c = a'(a(f))$. m_c verifies if a flow is matched by the first rule, and whenever this holds, if the subsequent transformation also matches the second rule. a_c simply encodes standard function composition. Thus, rule $r_{nat} = r_1 \circ r_2$ combines the topology-related port-forwarding with the actual NAT behavior. Rule composition is an essential means for building rules in a modular way, and which is also suitable for merging configuration files.

We define a *network function* $NF : U \to \mathcal{P}(U)$ with respect to a set R of rules, as: $NF(f) = \bigcup_{r \in match_R(f)} r(f)$, where $match_R(f)$ are the rules from R which match f.

Reachability Given a flow f where $port = p_s$ and a destination port p_d, reachability is the problem of establishing the set of packets which can reach p_d if f is sent from p_s. In what follows *NF* is a network function.

Solving reachability amounts to exploring all network transformations of f, that is, applying all rules from *NF* which match f and recursively applying the same procedure on all flows resulted from the (previous) rule application(s). Formally, this amounts to the application of the operator $OP_A : \mathcal{P}(U) \to \mathcal{P}(U)$, defined with respect to *NF* and a set of flows A, as follows:

$$OP_A(X) = \begin{cases} A & \text{if } X = \emptyset \\ A \cup \bigcup_{f \in X} NF(f) & \text{otherwise} \end{cases}$$

Consider the following (infinite) sequence:

$$X_0 = \emptyset,\ X_1 = OP_{\{f\}}(X_0),\ X_2 = OP_{\{f\}}(X_1)\ \ldots$$

Note that X_1 is the singleton set containing the initial flow f. X_2 is the set of all flows which result from the application of all matching rules on f. In other words, X_2 contains all flows which are reachable from f in a single step. Similarly, each X_i is the set of flows which are reachable from f in $i-1$

Algorithm 1: Reach(NF,f,p_d)

```
1 X = ∅, Y = {f}, R = ∅;
2 while X ≠ X ∪ Y do
3 |   X = X ∪ Y, Y' = ∅;
4 |   for f ∈ Y do Y' = Y' ∪ NF(f)   Y = Y';
5 end
6 for f ∈ X do
7 |   if (port, p_d) ∈ f then R = R ∪ {f}
8 end
9 return R
```

steps. The *least fix-point of* $OP_{\{f\}}$, that is, the smallest set X_i, such that $X_{i+1} = OP_{\{f\}}(X_i) = X_i$, contains all flows which are reachable at all ports from the network.

Algorithm 1 describes our reachability procedure. At lines 2-7, the set X contains all previously computed flows, or is $\emptyset$ if the current iteration is the first one. Y contains precisely those flows which have been computed in the previous step. The loop will build up new flows from the former ones (lines 4-6), until the set of all flows computed at the current step ($X \cup Y$) coincides with the set computed in the previous step (X). Finally, in lines 7-9, the flows reaching p_d will be extracted from the fix-point.

PROPOSITION 3.3. *For any network function NF and set A,* OP_A^{NF} *has a least fix-point.*

PROOF. According to Tarski's Theorem [6], it is sufficient to show that OP_A is monotone, that is $X \subseteq Y$ implies $OP_A(X) \subseteq OP_A(Y)$. Assume $X \subseteq Y$. Then $\cup_{f \in X} NF(f) \subseteq \cup_{f \in Y} NF(f)$ and thus we also have that $OP_A(X) \subseteq OP_A(Y)$. □

PROPOSITION 3.4. *The algorithm Reach is correct.*

PROOF. *Reach* computes the least fix-point of $OP_{\{f\}}$, and thus the set of all flows reachable in the network. The proof is done via structural induction and is straightforward. Termination is guaranteed by the existence of a fix-point for OP, via Proposition 3.3. □

In the absence of loops, the fix-point computation is $O(P * |NF|)$, where $|NF|$ is the number of rules from the network, and P is the number of network ports.

Loop detection *Topological loops* are identified by flow histories of the type $hph'p$, where p is a port and h, h' are (sub)histories. They are not necessarily infinite. For instance, in a loop where network nodes always decrease the TTL field, packets will eventually be dropped. Such a loop is *finite*. In what follows, we will focus on detecting *infinite loops* only. The principle is similar to that applied for reachability. First, we fix the set A to contain the most general flows which originate from each port: $A = \cup_{p \in Ports}\{(port, p)\}$. Second, we introduce an additional rule r_{loop} handling a variable *History* which stores the sequence of ports explored up to the current moment, in each flow. All rules are subsequently composed with r_{loop}. Thus, the application of each rule also updates the *History* variable. Third, when computing the least fix-point of OP_A we shall compare flows $f|_{History}$ and $f'|_{History}$ instead of simply f and f', to ensure the monotonicity of OP_A. Finally, a loop is identified by any two flows f and f' in the least fix-point of OP_A, such that the history of f is hp, that of f' is $hph'p$ and $f'|_{History}$ is more general than $f|_{History}$. In other words, there is a topological loop at port p, and the set of packets reaching p the second time is guaranteed to match the same network rules as the first time.

Modelling a NAT. The previously introduced rule r_{nat}, applied NAT transformations on all incoming packets from port p_1, and forwarded them on port p_2. In what follows we continue Example 3.2 and illustrate how reachability can be used to establish whether communication at the TCP layer is possible between two devices (*client*, *server*) separated by a NAT. The network configuration is shown in Figure 2(a). The client is modelled by the rule which injects the flow $f_0 = f_1|_{port=p_0}$ (on port p_0). f_0 models the set of packets generated by *client* and which are destined for *server*. We model topology links as rules matching and transforming the appropriate ports, and leaving the rest of the flow variables unchanged. Finally, the behaviour of the NAT for packets arriving at port p_2 is modelled by a rule which looks up the state variables IP_{nat} and TCP_{nat} and restores their content to the header variables IP_{dst} and TCP_{dst}. The sequence of flows and how they are transformed is depicted in Figure 2(a). Note that all outgoing flows from port p_2 will store the NAT's state, which is further used once a response from the server arrives at the same port.

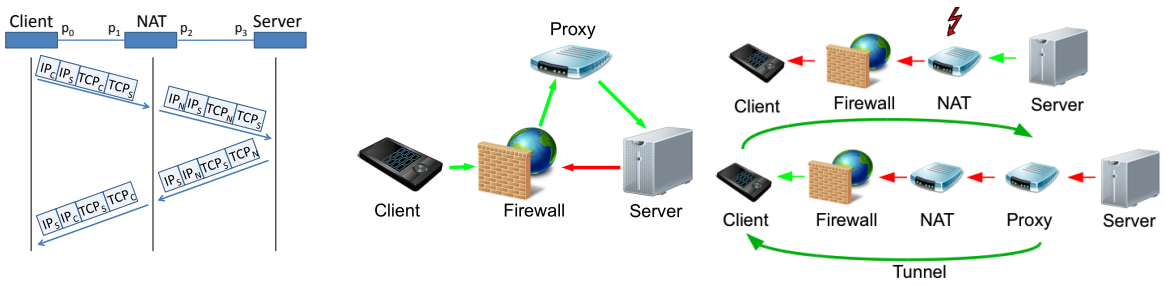
Using reachability to check (stateful) network properties. One interesting property is whether or not a header field of a packet sent from a source port p_s can be read at a destination port p_d. Question 3 from Section 2 boils down to this property, which can be checked by creating a flow f where the header field variable at hand is bound to an unused variable (i.e. the field can have any possible value), running reachability with f from p_s to p_d, and checking if the header field variable binding remains unchanged. Question 2 from Section 2 can also be answered similarly. In this case, we simply look at the expression bound to the destination address of the flow which is reachable at p_d.

TCP connectivity between p_s and p_d can be checked by (i) building up a flow f where TCP and IP source and destination variables are bound to unused variables and the variable modelling the SYN flag is set to true, (ii) building a "response rule" at p_d that swaps the IP addresses and TCP ports in the packet, and also sets the ACK flag. (iii) performing reachability from p_s with flow f to p_s, and examining the reachable flow at p_s to test if the IP addresses and ports are mirrored (i.e. the destination address of the outgoing flow is the source address of the incoming flow). If this is the case, TCP connectivity is possible. This is how we answer Questions 1 to 4 at the transport level.

4. IMPLEMENTATION

Our implementation of SymNet has two components. The first is developed in Haskell, and allows building up abstract models of a network configuration and/or topology. The second component, developed in Scala, uses Antlr to parse network configurations specified using the Click[3] language and generates abstract Haskell models. Symbolic execution is performed in Haskell on such models following the method described in the previous section. [3]

[3]The combination of two languages has the advantage that model generation and symbolic execution have no interde-



(a) Modelling a NAT (b) Enterprise network (c) Inbound connectivity in cellular networks

Figure 2: Simple network configurations to be checked with SymNet

5. EVALUATION

Our evaluation of SymNet focuses on several questions. First, we would like to check whether SymNet works correctly and gives appropriate answers in simple topologies that we can reason about. Second, we are interested in understanding how to model per-flow state for specific middleboxes. Finally, we want to see if SymNet can help guide protocol design decisions.

In our experiments, we use header variables for the IP and TCP source and destination address fields, the flags field (i.e. SYN/FIN/ACK), as well as the sequence number, segment length and acknowledgment number fields.

Enterprise network. Consider a typical small enterprise network running a stateful firewall and a client (C) of that network that uses a proxy (Figure 2(b)). All the client's requests pass through a stateful firewall. The proxy forwards the traffic to the server, as instructed by the client. To model connection state changes of the firewall, we insert a firewall-specific variable that records the flow's 5-tuple as the SYN flow goes from the client to the server.

We consider two behaviors at the proxy. If the proxy overwrites just the destination address of the flow, the server will reply directly to the client and the returning flow will not travel back through the proxy. As the flow arrives at the firewall, the firewall state (saved in the flow) does not match the flow's header. The output of SymNet in this case is the empty flow: there is no TCP connectivity between C and S, despite the fact that a flow from C arrives at S.

If the proxy also changes the source address of the flow, the reverse flow will traverse it and allow it to perform the reverse mapping. In this case, connectivity is possible.

Tunnel. A client machine wants to do a DNS lookup using the Google Public DNS resolver (8.8.8.8). Its operator disallows external resolvers by deploying a firewall that explicitly forbids packets to 8.8.8.8, as shown in Figure 1.

We would like to know how the tunnel should be configured to allow the client to send packets to 8.8.8.8. To answer this question, we bind the IP addresses of the tunnel endpoints to unitialized variables—i.e. they can take any value. Next, we run a reachability test from the client to the input port of the box T_2. At T_2, the constraints on the IP destination address give us the answer: 'Not "8.8.8.8"'.

Hence, as long as T_2's address is different from 8.8.8.8, the flow will reach T_2. The listing below shows the output of SymNet (the client uses the IP "141.85.37.151" and DST address "8.8.8.8"). Port numbers are omitted to improve readability. Note the variables named "tunnel-...": these record the original header values, allowing SymNet to perform the operations needed at tunnel exit.

```
[("SRC-ADDR" := CVar "Var-1") ^ "DST-ADDR" :=
    And (CVar "Var-2") (Not (CVal "8.8.8.8")) ^
"tunnel-SRC-ADDR" := CVal "141.85.37.151" ^ "
    tunnel-DST-ADDR" := CVal "8.8.8.8")]
```

By running reachability one step further, after the traffic exits the tunnel, we see that the IP addresses in the packets are exactly the same as set by the source: hence, the source can reach 8.8.8.8:

```
[("SRC-ADDR" := CVal "141.85.37.151") ^ "DST-
    ADDR" := CVal "8.8.8.8" )]
```

Cellular connectivity. A mobile application wishes to receive incoming TCP/IP connections (e.g. push notifications). However, the network operator runs a NAT and a stateful firewall. We use SymNet to check for connectivity (Figure 2(c), top) between the server and the mobile application. This fails because the NAT does not find a proper mapping of it's state variables in the flow.

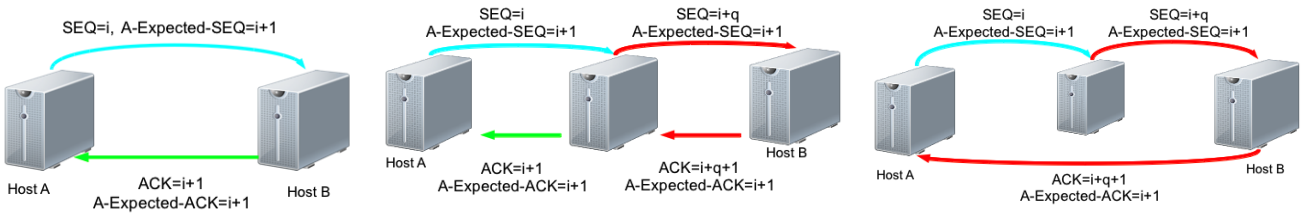
The standard solution to this issue is to use a proxy server outside the NAT, to which the client establishes a long running connection. The proxy then forwards requests to the mobile, as shown in Figure **??**, bottom.

We check this configuration in two steps. First, we model opening a connection from the mobile client to the proxy. As the firewall allows this connection, the TCP flow has all the state variables pushed by the stateful devices: firewall, NAT and proxy server. In step two, the outside connectivity request is tunneled to the mobile using the previous TCP flow as outer header. SymNet shows that connectivity is now possible: the flow has bindings describing an existing connection that can be matched by the firewall. The bindings of the source IP address and TCP source port point to the proxy. The true identity of the remote end is hidden by the proxy—rendering the firewall useless because it cannot filter blacklisted IP addresses.

Modeling middleboxes that change TCP sequence numbers. In this example we model common firewalls that randomize the initial sequence number of TCP connections, and modify all sequence and acknowledgment numbers afterwards. ISN randomization is done to protect vulnerable endpoint stacks that choose predictable sequence numbers against blind in-window injection attacks.

Does TCP still function correctly through such middleboxes? We first model the case when there is no middle-

pendencies. Thus, our approach can be naturally extended from Click to any network specification language.



(a) Direct communication (b) Bidirectional sequence number changes (c) Unidirectional sequence number changes

Figure 3: Modeling sequence number consistency in TCP

box (Figure 3(a)). Besides the regular TCP addresses, we also model sequence number and acks. After sending a segment with a sequence number, the host expects an ACK for that sequence number plus 1. When running reachability, we model this state by pushing a control variable called 'Expected-ACK' into the symbolic packet generated by A.

As this packet reaches B, the latter issues a new symbolic packet with the SYN/ACK flags set, containing the TCP and IP addresses from the original packet but with their values switched. The packet issued by B also includes a new value for SEQ representing B's initial sequence number and two new bindings: the ACK field and its own variable describing the Expected-ACK from its peer. We ommit SEQ in the ACK packet to ease readability.

When A receives the SYN/ACK packet, it checks to see if the Expected-Ack matches the ACK received. If it does, A will generate the third ACK. Finally, B will match its Expected-ACK and the ACK variable. A match is found and the flow will hold state corresponding to the newly established connection. One can observe that the flow may hold state variables with the same meaning but corresponding to different entities, Expected-ACK in this case. A simple solution is to incorporate an identifier of the device within the variable name, for example: 'A-Expected-ACK'.

In Figure 3(b) we add a box that performs ISN randomization. This box will add some value to the SEQ field of packets in one direction and subtract it from the ACK in the reverse direction. After running the same tests as before, we can observe that TCP/IP connectivity is still possible: at each side the values of the symbolic variables Expected-ACK and ACK match.

Finally, if the return traffic $B - A$ does not pass through the middlebox, the test fails: B sets the ACK field to 1 + SEQ, but SEQ has already been altered by the middlebox. Hence, A can not match Expected-ACK and ACK.

Scalability. To validate our complexity analysis, in Figure 4 we plot the time needed to check increasingly large networks. The results confirm that SymNet checking time scales linearly with the size of the network.

6. CONCLUSIONS

Middleboxes make networks difficult to debug and understand, and they are here to stay. To escalate the problems, recent industry trends advocate for software-only middleboxes that can be quickly instantiated and taken down. Existing tools to analyze networks do not model stateful middleboxes and do not capture even basic network properties.

In this paper we have shown that modeling stateful networks can be done in a scalable way. We model packet

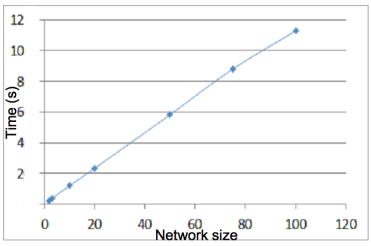


Figure 4: SymNet checking scales linearly

.

headers as variables and use symbolic execution to capture basic network properties; middlebox flow state can also be easily modeled using such header variables. We have proven our solution is correct and its complexity is linear.

We have implemented these algorithms in SymNet, a tool for checking stateful networks. SymNet takes middlebox descriptions written in Click together with the network topology and allows us to model a variety of use-cases. SymNet scales well, being able to check a network containing one hundred middleboxes in 11s.

Acknowledgements

This work was funded by CHANGE, a research project funded by the European Union's Seventh Framework Programme.

7. REFERENCES

[1] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, nsdi'13, pages 99–112, Berkeley, CA, USA, 2013. USENIX Association.

[2] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[3] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[4] Haohui Mai and et al. Debugging the data plane with anteater. In *Sigcomm*, 2011.

[5] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[6] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

[7] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On static reachability analysis of ip networks. In *Proceedings of Infocom*, 2005.

[8] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CONEXT*, CoNEXT '12, 2012.