UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Verificarea statică a rețelelor de calculatoare
folosind execuție simbolică

Coordonator ştiinţific:
Conf. Dr. Ing. Lorina Negreanu

Absolvent:
Radu Stoenescu

BUCUREŞTI

Septembrie, 2014

POLITECHNICA UNIVERSITY OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Static network verification using symbolic execution

Thesis supervisor:
Conf. Dr. Ing. Lorina Negreanu

Student:
Radu Stoenescu

BUCHAREST

September, 2014

Today's networks deploy many stateful processing boxes ranging from NATs (Network Address Translation hosts) to firewalls and application optimizers: these boxes operate on packet flows, rather than individual packets. As more and more middle-boxes 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 currently only support stateless processing.

I 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 middle-box 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 middle-box per-flow state in a scalable way.

I have implemented this technique in a tool I call Veriph conducted preliminary evaluation. Early results show that Veriph works well and is able to model basic stateful middle-boxes, opening the possibility of analyzing complex stateful middle-box behaviors.

Table of Contents

Figure index

Notations and Abbreviations

CDN – Content Delivery Network
DNS – Domain Name System
HSA – Header Space Analysis
IP – Internet Protocol
NAT – Network Address Translation
NFV – Network Function Virtualization
TCP – Transfer Control Protocol
WAN – Wide Area Network

1. Introduction

Middle-boxes have become nearly ubiquitous in the Internet because they make it easy to augment the network with security features and performance optimizations. Google Global Cache [6] and Skype's “Supernodes” represent real-world examples of the proliferation of middle-boxes.

Network Function Virtualization (NFV) - a recent trend towards migrating middle-box functionality towards virtual machines hosted in data centers will further accelerate middle-box deployments as it promises cheaper, scalable and easier means of upgrading and running middle-boxes.

The downside of this trend is increased complexity: middle-boxes make 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.

Static checking is a promising approach which helps understanding whether a network is configured properly. Unfortunately, existing tools such as HSA [1,3] (Header Space Analysis) are insufficient as they only focus on routers or assume all middle-boxes are stateless, this is certainly a sever limitation since middle-boxes that are most challenging to understand and wide-spread are stateful.

Checking packet forwarding alone only tells a part of the story because middle-boxes can severely restrict reachability in terms of packet flows. The one common trait of most middle-boxes is maintenance of per-flow state, and taking packet actions based on that state. Such middle-boxes include some of the most popular such as: network address translators (NATs), stateful firewalls, application-level proxies, WAN (Wide Area Network) optimizers, traffic normalizers, and so on. Finally, existing tools only answer questions limited to packets; with stateful processing everywhere, we should be able to answer questions about connections or ***packet flows***.

This work proposes a new static analysis technique that can model stateful middle-boxes and packet flows in a scalable way. This solution stems from two key observations:

1. TCP endpoints and middle-boxes 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. Middle-boxes 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 does not scale due to path explosion [7] during the path exploration phase of symbolic execution. Flow-state, however, can be easily checked if we assume that flow-state creation is independent for different flows.

Starting from these two observations, I have built a tool called Veriph 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 properly enforced?
Are two network hosts capable of communicating by the use of TCP?

- **Dynamic middle-box instantiation**
  What happens if a middle-box is removed or added to the current network?
  How will the traffic change given this alteration of the network setup?

- **Guiding protocol design**
  Will a TCP extension work correctly in the presence of a stateful firewall randomizing initial sequence numbers?

Veriph also proves to be scalable as its complexity depends linearly on the size of the network.

One drawback of relying on symbolic execution for static verification is the exponential time upper bound [8]. Fortunately, choosing Click [2] as the language for network specification instead of a general-purpose language such as C or Java brings a series of constraints that simplify the verification problem while offering a sufficiently powerful tool for programming networks (describing links and middle-box functionality).

The current iteration of the prototype implementation takes as input the routers (their forwarding tables) and the middle-boxes in the network together with the physical topology. Each box is described by a Click configuration. The prototype can check networks of hundreds of boxes in seconds.

The criteria for choosing Click as the network specification language were:

- Expressive power – the language of choice should allow one to build sufficiently complex middle-box configurations without much effort.
- Complexity of symbolic execution – performing symbolic execution on a program written in a language that offers a high degree of freedom to the programmer (direct memory access, memory pointers etc.) yields high complexity compared to a more constraint programming model (pure functional programming).

I have used Veriph check different network configurations, network configuration changes and interactions between endpoint protocol semantics and middle-boxes.

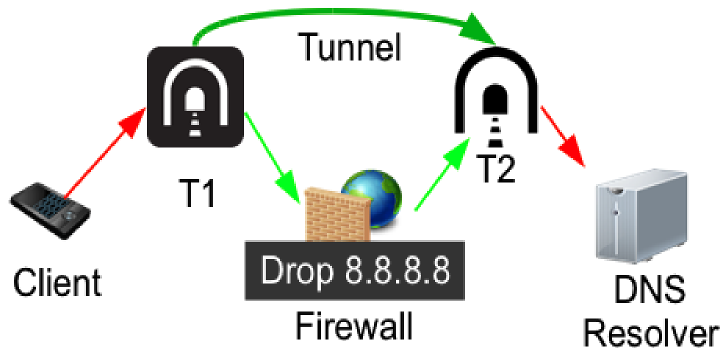


Figure 1: An example network containing a firewall and a tunnel

2. Problem Definition

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 T1, which encrypts it and then forwards it to the other end of this tunnel, T2, which decrypts the traffic and sends it onwards to its final destination. A number of interesting questions arise:

1. Is the firewall doing its 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?

2.1. Reaching a Suitable Problem Model

In order to bring relevant answers to the above questions one should rely on a model that is able to comprise and reason about:

- Agents that communicate via stateful networking protocols such as TCP.
- Network traffic described by packet flows instead of individual packets.
- Middle-boxes with behavior dependent on flow state. For instance a middle-box that runs a common NAT algorithm may chose a set of actions when confronted with a fresh (never-seen-before) network flow that is different to the actions performed when the network flow consists of packets that were already processed by the NAT.

Based on the considerations above, answering the previously stated questions should require a verification tool that can:

- **Trace the value of header fields** of a packet flow at different interest points within the network (for example before and after being processed by a given middle-box). For instance: the IP source header field of a packet accepted by the firewall should not be changed before it reaches its destination. On the contrary: a NAT middle-box should change the IP source header to one in the rage of publicly accessible addresses.
- **Model flow state.** This capability allows capturing middle-box behavior that is dependent on flow state. Such behavior includes network address translators, stateful firewalls, tunnels, proxies and so forth. This per-flow state is essentially the information based on which a NAT chooses to drop or not a packet arriving from the external network. Including this kind of information is essential for being able to build a tool that can reason about a succession of packets instead of individual ones.

2.2. Neglecting State beyond Flow-state

The previous paragraph states the minimum requirements for a model one can use to tackle the problem at hand. I argue this model should include at least the state of a network flow (what is the set of possible values for the header fields one is interested in) and per-flow state that middle-boxes require for their processing.

Beyond this, **global state** is also a factor to be considered. Or is it?

By global state we refer to state that does not pertain or can be attributed to an individual flow, but state that describes the functioning state of the network as a whole. Examples of global state are: the load factor of switch buffers, the state of the address pool in use by a NAT, the current load of a firewall.

It is clear that including global state as part of our model is going to yield better precision (we can answer a wider range of questions). But this does not come without a price: increased model complexity. In other words, one should evaluate if the added precision is satisfactory given the added complexity.

Should we trade complexity for more performance?

Let us understand what is to win and what is going to be lost before reaching the final decision.

2.2.1. Added Precision

In a the simple example of a stateful firewall, modeling global state means being able to reason about the state of its memory table and be able to make a distinction and, consequently, separately analyze: a network flow that reaches the firewall when the firewall memory is:

- Empty
- Already populated with state pertaining to previously encountered network flows, but not full.
- Full

In a more contrived example let us consider a trivial middle-box that invariably drops every fifth fresh network flow it sees.

This processing relies on updating and checking a global flow counter. For being able to model such a middle-box instance one should include in the symbolic execution model the state of this global counter. Without this added precision we would not be able to understand what is going to a network flow passing through such a middle-box – is it going to pass or is it going to get dropped?

Examples of middle-boxes that greatly depend in their functioning on global state include: cache-driven CDN middle-boxes, traffic meters, load balancers, middle-boxes with rate-limiting behavior.

2.2.2. Increased Complexity

In the above-mentioned case of analyzing a NAT middle-box, what is the added complexity?

One can observe that symbolic execution should explore at least the following paths:

- The network flow is fresh and the memory is full, thus the traffic is dropped.
- The network flow is fresh and the memory is not full: a new entry is created in the NAT table and the flow gets propagated.
- The network flow is already known (it is the response from a server in the Internet to a client behind the NAT)

Even in this trivial case of a basic NAT, results yielded by the most optimistic estimation say that we are expecting at least another branch in the path exploration performed by the symbolic execution algorithm that is at least constant a constant strictly greater than 1.

To sum up, for every middle-box that relies on global state we are expecting an increase by at least a constant factor. In the case of a network, the increase is exponential in the number of middle-boxes.

Having reached this observation, I trade the incurred lack of precision for a avoiding an exponential increase in the expected run time of the algorithm.

3. Other Approaches

3.1. Static Analysis

A prerequisite of using static analysis to answer questions such as the above is an accurate view of the network routing tables and middle-box processing 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. TCP connectivity can be understood by tracking the header values involved in the “Three way handshake” algorithm used for establishing a new connection between two network peers.

Header Space Analysis (HSA) models arbitrary stateless middle-boxes 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 middle-box state, and thus cannot perform an accurate analysis on most networks as these contain NAT hosts, stateful firewalls, DPI (Deep Packet Inspection) 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*.

Third question 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 modeled by a header-space 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” packet arriving, 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 middle-box 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 do not, AntEater will provide example packets that violate that property. AntEater does not model state either; additionally, its reliance on SAT (Boolean satisfiability problem)-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 [5]. Another hybrid approach uses HSA to dynamically check the correctness of updates in SDN (Software Defined Networks) networks. Both these tools inherit the drawbacks of HSA: they cannot model stateful middle-boxes.

At the other end of the spectrum, one can perform static analysis using model checking [9]. In this context, verifying that a given property holds with respect to a given finite-state system requires an exhaustive verification of the possible states of the system. In the field of large enterprise networks this clearly renders the problem of network verification intractable.

3.2. Dynamic Analysis

Another approach used in the field of network verification is dynamic analysis (also referred to as dynamic network testing or packet injection). This technique requires the network infrastructure that is to be tested to be on-line in an environment restricted only to testing purposes. In this context, test agents (network hosts) generate network traffic in concordance to the test cases and then gather and evaluate the results of the network processing performed by the devices subjected to testing. This closely resembles unit testing of computer software.

The downsides of this approach and the reasons why static analysis was chosen instead of dynamic testing are:

1. It requires a period during which the network infrastructure is reserved exclusively to testing. In this era in which large enterprise networks with hundreds of hosts this may not be feasible.
2. Exhaustive testing may require too many test cases.
3. Generating custom tests and targeting a specific behavior of a network usually requires an additional devoted to gathering information about the network and understanding what is the set of behaviors that should be tested and in how is the traffic going to look in order to cause such behavior.
4. The time needed for performing the tests is strongly dictated by the characteristics of the hardware.
5. This procedure requires more resources than static analysis.

4. Symbolic Execution

4.1. Introduction to Symbolic Execution

For every program, one can build a tree of possible states the program can reach given an arbitrary input. Under this assumption, every node in the tree is a possible program state and every transition between states is made by examining the outcomes of running the next program instruction (as dictated by the initial state).

By constructing and then examining such a tree, one can statically (without actually running the actual program) learn a number of properties about the program [11, 12]:

1. Provided that every in possible state (node) of the program a given property hold, one can assume that property of the program, invariant to the input of the program. For example one can assume no “division by zero error” can occur if never in the program a number is divided by a variable that may hold the value 0.
2. If a given property does not hold in case of a (or multiple nodes) node, one can trace back the program exploration path that lead to this state and understand what inputs may cause this to happen.
3. The process of constructing unit tests can be aided by automatically providing inputs that explore certain program execution paths [11], leading to a better test coverage.

Let us now step through the construction process of such a tree in case of a simple C program:

```
char i;                               (1)

if (i % 2 == 0)                       (2)
            i += 2;                   (3)
            printf(“Even\n”);         (4)
else                                  (5)
            printf(“Odd\n”);          (6)
```

Figure 2: C program subjected to symbolic analysis

At line (1), to symbol *i* are allocated 8 bits of memory. Since is not initialized one can make no assumption about the possible value of any bit. Essentially any bit can be either 0 or 1.

Next (2), the parity of *i* is checked. On the first branch, the case in which *i* is even is considered, thus one can infer that the least semnificative bit (*B8*) should be 0.

Consequently, the current execution path branches to another node (5) in which case the same bit (*B8*) has the opposite value, 0.

The first branch gets explored further by executing lines (3) and (4), while the other possible execution path, corresponding to the *else* branch explores the instructions at lines (5) and (6).

Having traced the construction algorithm for constructing a symbolic execution tree one can understand the potential of this technique for performing static verification. On the other hand, this is not a path without dangers: firstly, a high number of branches leads to an exponential number of paths being explored (phenomenon referred to as *path explosion*), secondly, every node in the tree encapsulates a complete view of the memory in use by the program; although algorithms for compressing and reusing the space demanded by this have been proposed, this still remains a consideration [11].

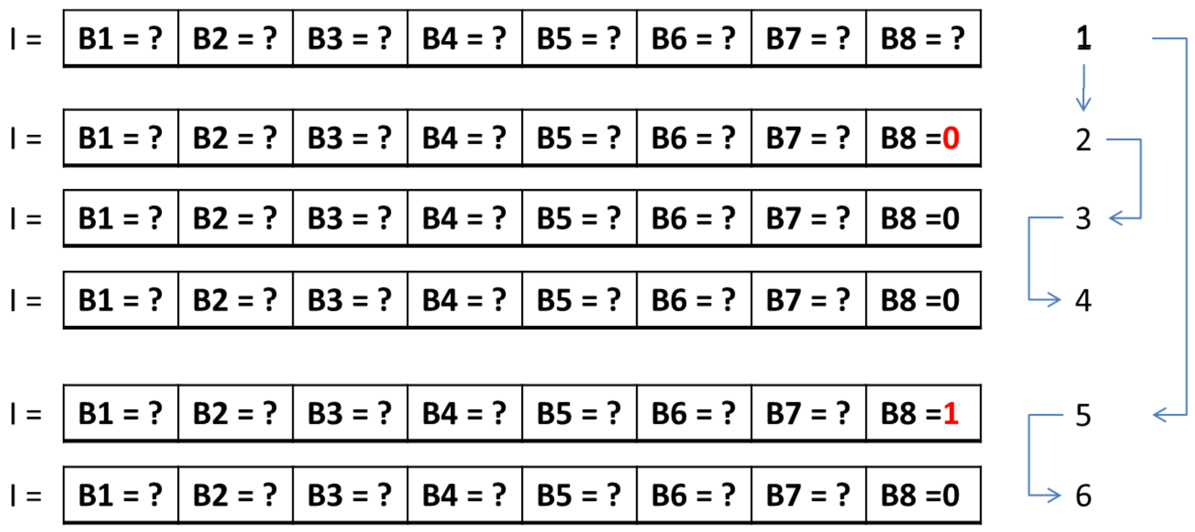

I = B1 = ? B2 = ? B3 = ? B4 = ? B5 = ? B6 = ? B7 = ? B8 = ?
I = B1 = ? B2 = ? B3 = ? B4 = ? B5 = ? B6 = ? B7 = ? B8 =0
I = B1 = ? B2 = ? B3 = ? B4 = ? B5 = ? B6 = ? B7 = ? B8 =0
I = B1 = ? B2 = ? B3 = ? B4 = ? B5 = ? B6 = ? B7 = ? B8 =0
I = B1 = ? B2 = ? B3 = ? B4 = ? B5 = ? B6 = ? B7 = ? B8 =1
I = B1 = ? B2 = ? B3 = ? B4 = ? B5 = ? B6 = ? B7 = ? B8 =0
1
2
3
4
5
6

Figure 3: Symbolic execution tree

4.2. Symbolic Execution for Network Analysis

Veriph relies on symbolic execution—a technique prevalent in compilers—to check network properties such as TCP connectivity between specified network hosts, the existence of loops, reachability of a certain host by traffic of a given nature, etc. The key idea underlying Veriph is to treat sets of possible packets as *symbolic packets* (let us simply refer to this notion as a network *flow)* and network devices as transformation functions (I shall refer to those as *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 hand, 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.

5. In-depth View of the Symbolic Execution Model

5.1. Modeling Packets

Following HSA (Header Space Analysis), we model the set of all possible headers having *n* fields as an ***n***-dimensional space. A particular flow is a subset of such a space. This approach is very similar to treating the state of a running program as a subset of the space of all its possible states, with is also ***m***-dimensional, where *m* is the number of variable identifiers defined by the program.

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[1]. 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,* an uninitialized variable), and also referred to in other expressions. For instance, Question 1 from Section 2 can now be answered by assigning a variable, let us name it 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. This would essentially state that v_d remains invariant along its pass through the network.

Let *Vars* and *Expr* designate finite sets whose elements are called (header) *variables* and *expressions*, respectively. We only consider expressions generated by the following BNF grammar:

$$expr := c \mid v \mid \neg\ expr \tag{1}$$

where $c \in Expr$ and $v \in Vars$.

Let $C \subseteq P\ (Vars \times Expr)$. *C* models a *compact space* of packets. The set $U \subseteq P(C)$ 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:

$$f = cf_1 \cup cf_2 \cup \ldots \cup cf_n \tag{2}$$

where each cf_i is a compact space.

f models the subspace of *U*. Whenever a flow is a compact space (i.e. it is at most the reunion of one compact flow), it will simply be written *cf*, in order to avoid using double brackets.

Continuing our analogy with the traditional way of conducting symbolic execution, let us examine the following code sequence, and see how it fits our model:

1

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`".

```
int i;                              (I)
if ( i != 0 )                       (II)
            i = 1;                  (III)
else                                (IV)
            i = 2;                  (V)
i++;                                (VI)
```

In this context *f*, a flow in our model represents the state of the above program as it is traced by symbolic execution:

(I) At this point *f* comprises only the symbol *i* which is bound to the expression consisting of an unbound variable, let us name it i_1.

(II) On the first branch of the *if* statement the unbounded i_1 gets bound to a possible set of values represented by the union of two compact flows: $cf_1 = \{ x \mid x < 0 \}$ and $cf_2 = \{ x \mid x > 0 \}$

(III) After the first assignment the symbol *i* is assigned a new expression consisting of a new variable, let us name it i_2 which has a domain represented by a single compact flow $cf = \{ x \mid x = 1\}$.

(IV) Consequently, on the other branch of the conditional statement, the negation of the condition holds, yielding a new possible program state, let us name it *f''* in which i_1, the expression referred to by the symbol *i* gets bound to a possible set of values defined by the compact flow $cf = \{ x \mid x = 0 \}$

(V) Following that, executing the assignment statement, symbol *i* is assigned a new expression consisting of a new variable i_2 with a domain expressed by the compact flow $cf = \{ x \mid x = 2 \}$.

(VI) At this point the set of possible states of the program has already expanded to a number of 2, one for each branch of the *if* statement. The action of the postfix increment statement has to be traced in both cases. Thus, the result consists of two possible flows *f'* consisting of the symbol *i* bound to variable i_3 with a domain consisting of $\{ x \mid x = i_2 + 1\}$ and *f''* in which case symbol *i* is also bound to i_3, defined by the same expression, the only difference being that i_2 is defined by a different set of possible values: 1 and, respectively, 2.

Let us introduce the following notations:

$$f|_{v=c} \quad (3)$$

$$f|_{v} \quad (4)$$

Notation defined by (3) refers to the flow obtained from a given *f* by enforcing that *v* has value *c*.

Consequently, notation (4) refers to the flow obtained from *f* where *v* has no associated value.

Using the above notation in the context of network flows, a flow that is known to be directed to the IP address *8.8.8.8* is defined like this: $f|_{ip_{dst}=ip_1, ip_1=8.8.8.8}$.

5.2. Modeling State

Unlike HSA, which models stateless network hosts only, Veriph can model stateful devices by treating per-flow state as additional dimensions in the packet header. This idea of breaking the boundary between values and variables pertaining to the network flow itself and variables and values used by network hosts to maintain information about their traffic stems from the observation that high-level programming languages perform the same thing when abstracting over the memory hierarchy and memory-access patterns.

In this respect we can pretend that a flow is the only resource available for storing network state, given that memory access is governed is such a way that any given network host *H* can access information held in packet header fields, together with any information it stores about this flow, beyond this no two devices H_1 and H_2 can access, share or, for that reason, modify per-flow state in a mutual fashion.

To sum up, state is not explicitly bound to devices, rather "pushed" in the flow itself.

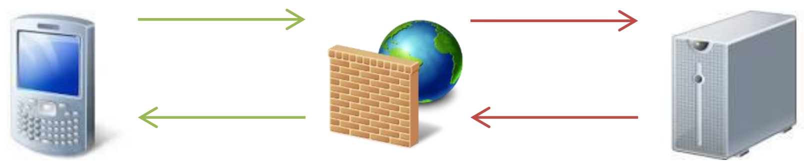


Figure 4. Communication between a client and a server through a stateful firewall

The simplest example is a stateful firewall that only allows outgoing connections, as depicted in Figure 4.

As the symbolic packet goes out to the Internet, the firewall pushes a new variable called *firewall-ok* into the packet.

A field such as that can be accessed only by the firewall, neither the client nor the server can process this field, not even acknowledge its presence.

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. This field acts essentially as a *flag* that is set en-route to the internet and is checked on the way back to the client.

This model of dealing with per-flow state scales well under the assumption that network hosts hold per-flow state that is bounded by a constant. This assumption stems from the observation that usually per-flow state is held in a key-value store in which case values are constant in space.

5.3 Assumptions about Flow-state

However, our model makes a few strong assumptions:

1. We assume each flow's state is independent of the other flows. A number of consequences arise from this assumption:

    1.1 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.

    1.2 We completely bypass global variables held by the middle-boxes, and assume these do not (normally) affect flow state. Packet-counters and other statistics normally do not affect middle-box functionality, so this assumption should hold true.

2. We are not considering boundary cases such as: memory being corrupted, memory not being sufficient and so forth. The reason for doing so is that considering such cases can yield insights about the way processing alters network traffic when devices are not working properly. The target of this piece of research does not target such cases.

5.4. Modeling the Network

Once we have cleared the way we are going to store and access network state, we should focus our attention on the way we model the behavior of network devices.

In this direction, the network is abstracted as a collection of processing *rules* which can be:

(i) Matched by certain flows

(ii) Whenever the case above, a flow is can be transformed.

In a more intuitive sense, let us pretend we want to model the behavior of the firewall in Figure 4. The two fundamental questions we should answer are the following:

(i) When is the firewall active? In other words, given the set of flows passing through a network that includes such a firewall, at a given moment in time *t* which is the subset of flows that is going to pass through (and be processed) by the firewall.

(ii) Once we know at any given time which flows are processed by which hosts, we need to conveniently include in the model the processing these devices perform.

We essentially need to answer, for each device: *"When it is active?*" and *"What 'being active' means?"*

Let us define:

F – the set of all possible network flows
P*(F) – the power set of F*

Formally, a rule is a pair:

$$r = (m, a) \tag{5}$$
$$m : F \rightarrow Boolean$$
$$a : F \rightarrow \mathrm{P}(F)$$

Each device is represented by a rule *r* defined by its pair of *m* (match) and *a* (apply) functions.

Going back to the firewall, let us see how its corresponding rule might look like:

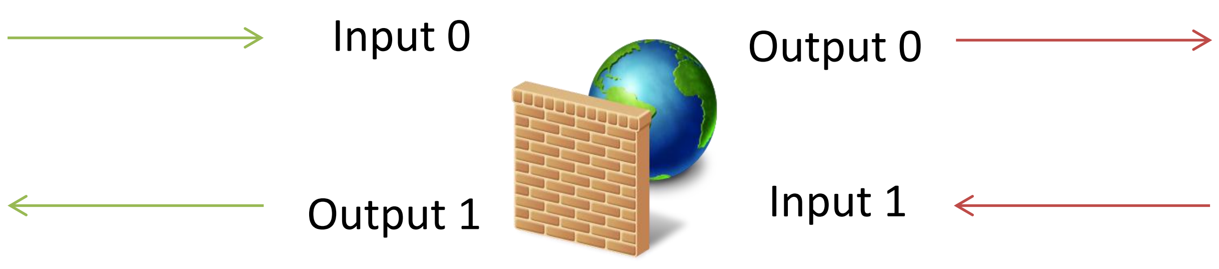


Figure 5. Stateful firewall

1. For defining the apply (*a*) we conclude that out firewall is active for processing a certain flow *f* if and only if, the port flow *f* has reached at this point in time is either *Input 0* or *Input 0*.

2. Reaching the definition of the apply (*a*) function is a little more involved: first let us acknowledge that out firewall is going to process differently two flows: one arriving on port *Input 0* and the other arriving on port *Input 1.*

In the first case, the firewall is going to store information telling him it has now "seen" this traffic. This action can be succinctly expressed:

$$a_1(f) = \{f|_{firewallOk}\} \tag{10}$$

In the second case, when a flow has arrived on port *Input 1*, the firewall is going to make a decision whether to forward or not this flow based on the previously added flow-state. If the variable *firewallOk* exists, this means this is a known flow, originating in the private network and this flow represents the server response, and should be allowed to pass. If the previous condition is not met, then a host in the internet is trying to open a new connection to a host behind the firewall which should not be allowed, thus the traffic is dropped.

The behavior I have described so far is summarized as follows:

$$a_2(f) = \begin{cases} \{f\},\ iff\ flow\ f\ contains\ variable\ firewallOk \\ \qquad\qquad \theta,\ otherwise \end{cases} \tag{11}$$

Having said this, it is more convenient and easier to grasp the functionality of the firewall in terms of rules if we split its functionality into a pair of rules *(r1, r2)*, *r1* – describing the login pertaining to the transition form $Input_0 \rightarrow Output_0$ and *r2* the other transition $Input_1 \rightarrow Output_1$.

We can break the complex functionality of middle-boxes into small units with easy to understand meaning, the example above supporting this argument.

More so, in the same way programmers isolate and encapsulate chunks of code into functions and classes for easier and better code reuse, we can do the same and build up more complex rules from simpler ones.

In our aid, we introduce the *rule composition operator* ∘, defined as follows:

$$r_1{}^{\circ}\, r_2 = r_{12} \tag{12}$$

$$\begin{aligned} r_1 &= (m_1, a_1) \\ r_2 &= (m_2, a_2) \\ r_{12} &= (m_{12}, a_{12}) \end{aligned} \tag{13}$$

$$m_{12}(f) = m_1(f)\ \wedge m_2(f) \tag{14}$$

$$\begin{aligned} a_1(f) &= \{f_1, f_2, \dots, f_n\} \\ a_{12}(f) &= a_2(f_1) \cup a_2(f_2) \cup \dots \cup\ a_2(f_n) \end{aligned} \tag{15}$$

The composition of rules *r1* and *r2* yields a rule *r12* defined by:

1. The match function *m12* that is true when both *m1* and *m2* are true.

2. The apply function *a12* that applies *a2* over the set of flows resulted by the application of apply function *a1.*

Rule composition is an essential means for building rules in a modular way, and which is also suitable for merging configuration files.

Let us define a *network function* with respect to a set *R* of rules, as:

$$NF_R : F \rightarrow \mathrm{P}(F) \qquad (16)$$
$$NF_R(f) = \cup_{r \epsilon R}\, r(f)$$

The network function, according to the definition at (16), is the function that encapsulates the logic of a complete network (as denoted by the set of rules *R*). Its result is the unification of the sets of flows produced by applying every rule *r* in the network on a given flow *f.*

5.5. Network Reachability

The problem of network reachability can be formalized like so:

Given a network in the form of a set of rules *R* and an input flow *f,* we are interested in verifying reachability of a port $p_{destination}$ by the input flow $p|_{port=p_{source}}$by *determining* the complete set of flows reaching $p_{destination}$.

Intuitively, one should start with the set of flows currently passing through the network consisting of only the input flow, an continue by applying step by step the network function *NF* on every flow until the set of possible flows remains constant (i.e. we have discovered the complete set of possible flows). At the end, one should filter the flows that did not reach $p_{destination}$ out of this set.

Formally, we introduce the definition of the *network operator* function with respect to a network *R* and a set of all discovered flows *A* as follows:

$$OP : \mathrm{P}(F) \rightarrow \mathrm{P}(F) \qquad (17)$$
$$\mathrm{OP(X)} = X \cup (\cup_{x \in X} NF(x))$$

The operator function describes the iterative process of applying the modifications described by set the set of rules *R* over all the currently discovered network flows, starting from the input flow and expanding to a set (possibly infinite) of future possible flows.

Consider the following (infinite) sequence:

$$X_1 = f_0, f_0 denoting\ the\ input\ flow \qquad (18)$$
$$X_2 = OP(X_1);\ X_3 = OP(X_2); \ldots$$

Note that X_1 is the singleton set containing the initial flow f_0; X_2 is the set of all flows which result from the application of all matching rules on f_0. In other words, it is the set of all flows which are reachable from f_0, in a single step (i.e. after a single application of the *network operator* function). Similarly, each X_i is the set of flows which are reachable from f_0 in i−1 steps.

We introduce now the *least fix-point* of the *network operator function* that is the set comprised of all the flows which are reachable in the network.

Let us refer to this set of all the reachable flows as the set *A*(all).

In the case of a finite *A,* we state the following:

$$\forall f \in A, \exists k \in N\ such\ that\ X_k = OP(X_{k-1}), f \in X_k \qquad (19)$$

$$A = \mathrm{OP(A)} \qquad (20)$$

5.5.1. Network Reachability Algorithm

Algorithm 1 describes the reachability procedure.

At lines 2-7, the set *X* contains all previously computed flows, or is ∅ 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 ∪ Y*) coincides with the set computed in the previous step (*X*). Finally, in lines 7-9, the flows reaching the destination port p_d will be extracted from the fix-point set.

```
Algorithm 1: Reach($NF$,$f$,$p_d$)

1 $X = \emptyset$, $Y = \{f\}$, $R = \emptyset$;
2 while $X \neq X \cup Y$ do
3   |  $X = X \cup Y$, $Y' = \emptyset$;
4   |  for $f \in Y$ do  $Y' = Y' \cup NF(f)$   Y = Y';
5 end
6 for $f \in X$ do
7   |  if $(port, p_d) \in f$ then  $R = R \cup \{f\}$
8 end
9 return $R$
```

5.5.2 Proof of Correctness of the Reachability Algorithm

The algorithm is correct if reaches a solution of the problem in a finite number of states.

Proposition I: *For any network function (consequently, any network in general) NF and input flow f_0, OP has a least fix-point.*

Proof: According to Tarski's Theorem [6], it is sufficient to show that OP is monotone, that is X ⊆ Y implies OP(X) ⊆ OP(Y).

Assume X⊆Y, then $X \cup (\bigcup_{x\in X} NF(x)) \subseteq Y \cup (\bigcup_{x\in Y} NF(x))$ since $\bigcup_{x\in X} NF(x) \subseteq \bigcup_{x\in Y} NF(x)$ and thus we also have that OP(X) ⊆ OP(Y).

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

5.6. Loop Detection

Network loops represent an ever present threat to modern networks. Even though algorithms and network configurations have evolved towards a state in which loops may not be able to threaten the correct operation of a given network, their presence can definitely hinder the performance of a network.

Many algorithms include mechanisms for detecting loops, for instance the TTL (time to live) field set to 0 may indicate a loop in the routing path. We will refer to such loops that are detectable by the algorithms employed by the network as *finite loops.*

For a certain device *d* to be able to detect such loops, it is necessary that for two distinct occurrences p_1 and p_2 of the same packet *p* on the same port of the device d: p_1 should differ from p_2, with respect to at least one packet field.

In what follows, we will focus on detecting *infinite loops*, those that do not satisfy the assumption above.

For instance, Figure 6 depicts two network hosts that forward network packets from one to the other:

(i) Device A decreases the TTL field of every packet is sees.

(ii) Device B increases the TTL field of every packet is sees.

It is clear to see now that both A and B are going to be presented with the same packet *p* over and over again. The safeguard against loops employed by the IP protocol is not going to work anymore and the packet will never get dropped unless one of the devices shuts down.

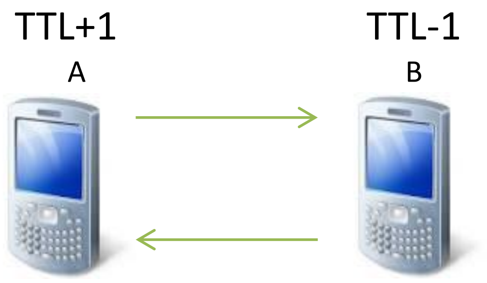


Figure 6. Infinite network loop

For tackling the case of *infinite loops*, we should make the observation that if either A or B can maintain the history of packets it has seen, it would be able to understand that any future occurrence of a packet coming from its peer is a clone of a previous one, and thus it could simply drop the clone. Unfortunately this is not feasible in practice since it clearly surpasses the capabilities of the current (and possibly near-future) hardware.

All is not lost because this solution may yield satisfying results in the case of static analysis.

A working infinite loop detection algorithm stems from the following observation:

For any given flow f that is reachable from the input flow f_0 is a finite sequence of applications of the network function *NF*:

$$F_i = NF(f_i) \quad (21)$$

where f_0 is the input flow itself, $\exists k,\ f \in F_k$, and $\forall k, f_{k+1} \in F_k$.

In other words, for any reachable flow f_k, there is a sequence of ancestors: f_0, f_1, …, f_k that lead to the occurrence of f_k.

Proposition II: Having stated that, for any pair of flows f_k and f_j, $j > k$, satisfying the condition that $f_k \subseteq f_j$ (i.e. f_k is an ancestor of f_j that is more particular), the sequence of flows ranging from f_k to f_j is going to reoccur, being generated by f_j.

Since the reachability algorithm already considers all the ancestors of newly discovered flows, one should simply augment it and, at every step, check if Proposition II holds. Whenever this is the case a loop is detected.

6. Modeling a Concrete Middle-Box (NAT)

Let us further evolve our understanding of how modeling middle-boxes according to the previously defined model is being carried out.

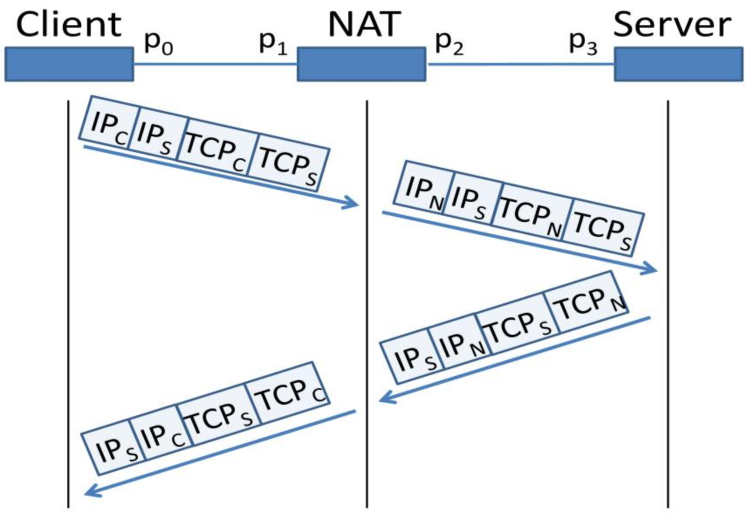


Figure 7. Modeling a NAT

In the following case, we target a NAT device that performs *port address translation (PAT)* in order to allow clients in a private network access the internet.

The functioning of such a device is described as follows:

1. A network flow originated by a client in the private network is directed to a server in the internet. This flow is described by:

$$f|_{ip_{source}=ip_C, ip_{destination}=ip_s, tcp_{source}=tcp_c, tcp_{destination}=tcp_s} \tag{22}$$

One thing to note is ip_C, ip_s, tcp_c, tcp_s, which represent the IP and TCP addresses of the client and the server, are variables that may or may not have actual expressions bound, essentially we do not infer anything about their values, except from the existence of such values.

2. Once it passes through the NAT en-route to the server, the source TCP port is assigned a new random value (NAT_{port}) and the source IP address is assigned a routable address in use by the NAT (NAT_{ip}).

$$f|_{ip_{source}=NAT_{ip}, ip_{destination}=ip_s, tcp_{source}=NAT_{port}, tcp_{destination}=tcp_s} \tag{23}$$

Furthermore, the information used to forward incoming packets back to the client (which, under normal circumstances, would have populated the NAT memory) it is now stored by the flow.

$$f|_{ip_{source}=ip_C, ip_{destination}=ip_s, tcp_{source}=tcp_c, tcp_{destination}=tcp_s, \boldsymbol{oldip_{source}=ip_c}, \boldsymbol{oldtcp_{source}=tcp_c}}$$

3. When a response comes back from a host in the internet, the NAT verifies the presence of the symbols $oldip_{source}, oldtcp_{source}$ which dictate whether or not this is a response to a request that already passed through the NAT.

If this is not the case, the flow is dropped.

Otherwise, $ip_{destination}, tcp_{destination}$ are restored to those of the client, that were referenced so far by the fields $oldip_{source}, oldtcp_{source}$.

The request response cycle ends and our symbolic execution algorithm can successfully trace the evolution of this flow through the network.

A number of observations are worth to be mentioned here:

1. The algorithm did not rely on actual values for performing the analysis; it was enough to know that values for TCP and IP addresses were present once the request was issued and that they were switched in the response from the server.

2. The correct functioning of the NAT was based on the assumption that it was configured with a routable address and that there is a TCP port available to be allocated for this new network flow, their concrete values were not important.

3. Because network devices operate at tremendous speeds and because numerous algorithms are hardware-implemented, it is safe to assume that the space reserved for state saved by the device is bound by a constant.

4. The algorithm should enforce that state saved by a certain device (such as: $oldip_{source}, oldtcp_{source}$) is not modified by other devices. In our case we address this issue by attributing every device a unique identifier which is then used when flow-state is accessed or modified.

7. Using Reachability to Check Network Properties

One interesting property, worth of investigation, is whether or not the value of a given header field of a packet processed by an arbitrary host *s* can be observed not to have been modified en-route to a destination *d*.

Question 3 from Section 2 (referring to the invariance of a given header field between two hosts) boils down to this property, which can be checked by creating a flow *f* where the header field variable being discussed is bound to an unused variable (i.e. the field can have any possible value), running reachability with *f* from *s* to *d*, and checking if the header field variable binding remains unchanged.

Furthermore, Question 2 from Section 2 can also be answered similarly. In this case, we simply evaluate the expression bound to the destination address of the flow which is reachable at the destination *d*. This set of possible values that verify the expression represents the set of reachable destination addresses.

TCP connectivity between two hosts can be checked by:

(i) Constructing a flow *f* where TCP and IP source and destination variables are bound to unused variables and the variable representing the *SYN* flag is set to the constant expression *true*

(ii) Building a "response rule“ that represents a server that accepts any connection request, swaps the IP addresses and TCP ports in the packet, and also sets the *ACK* flag to *true*

(iii) Performing reachability from a client with flow *f* to the destination server.

(iv) Examining the reachable flow back at the client 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.

8. Implementation

I have implemented and tested Veriph in Scala.

The inputs of the application are:

1. A description of the already existing network infrastructure (the system that is going to host newly deployed middle-boxes).
2.1. A *Click* file representing a new middle-box that is to be deployed if all the static checks pass.
2.2 For the above middle-box, a set of reachability checks that are going to be performed is supplied.

The output of the application signals if the checks had passed and was safe for the middle-box to be deployed.

The application runs along the following steps:

1. If the middle-boxes that are to be analyzed belong to a larger network, this model is loaded in the form of a series of Click configuration files.
2. The Click configuration file of the middle-box that is going to be analyzed is also loaded.
3. A Click parser is used (written using ANTLR) to build the abstract model of the network as a whole (together with the new middle-box).
4. Based on this model reachability checks are performed until one fails or all have passed.
5. Judging on the result of the previous step, the decision of whether it is safe to deploy the middle-box or not is made.

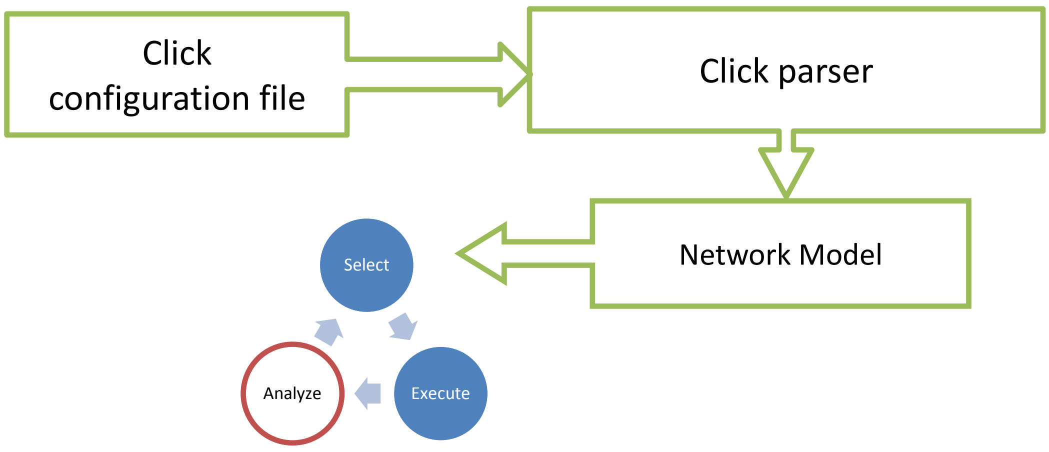


Figure 8. Implementation overview

8.1. What is Click ?

I have stated that Veriph assumes that network and middle-box configurations come in the form of Click configuration files.

Click is a solution for describing network processing (i.e. actions performed by network hosts and the links between them). Click offers advantages based on two considerations:

1. The Click specification language.

Click proposes a domain-specific language for network specification that allows a user to build increasingly complex networks by interconnecting already existing basic processing blocks (such as IP filtering, packet queuing, header rewriting blocks).

An example of a trivial Click configuration is:

FromDevice(eth0) -> Print(ok) -> Discard;

This consists of a host composed of three basic blocks:

a) FromDevice – a block that pulls traffic from an operating system network device (in this case the Ethernet card 0).
b) Print – a block that prints and then forwards any traffic.
c) Discard – the block that drops any traffic received.

The arrow (->) denotes a link between two blocks. These elements allow traffic forwarding between devices and thus the composition of Click basic elements.

To sum up the behavior of this configuration: it captures traffic from *eth0* interface, it echoes this traffic to the console, later this all the traffic is dropped.

Click offers a succinct and expandable, yet powerful and rigorous means of defining network processing and connections between hosts.

2. Click elements are implemented in C++ and offer the potential of being deployed conveniently in high-performance environments [15].

Click offers valuable means of specifying and actually running network appliances, which explains why it was chosen as input for this tool.

9. Evaluation

The evaluation of Veriph targets several questions.

First, we would like to know whether Veriph runs correctly and gives appropriate answers in cases simple topologies that are a commonality in today's networks.

Second, we are interested in understanding how to model per-flow state for specific middle-boxes.

Finally, we want to see if Veriph help guide protocol design decisions.

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

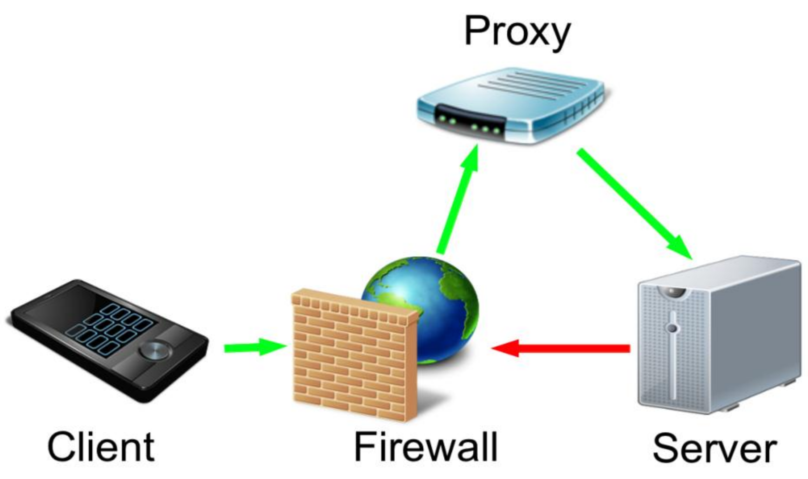


Figure 9. Enterprise network

9.1 Enterprise network

Consider a typical small enterprise network running a stateful firewall and a client of that network that uses a proxy (Figure 9).

All the client's requests pass through a stateful firewall.

The proxy forwards the traffic to the server, as instructed by the client.

To account connection state storage performed by 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 next consider two possible behaviors at the proxy:

If the proxy overwrites just the IP 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 within the flow) does not match the flow's header. The output of Veriph this case is the empty set: there is no TCP connectivity between client and server, despite the fact that a flow from client arrives at the server and the server responds to this request.

If the proxy also changes the source address of the flow, the returning flow will pass through and allow it to perform the reverse mapping. In this case, the flow is routed through the proxy. Using Veriph one can acknowledge that TCP connectivity between the client and the server is feasible.

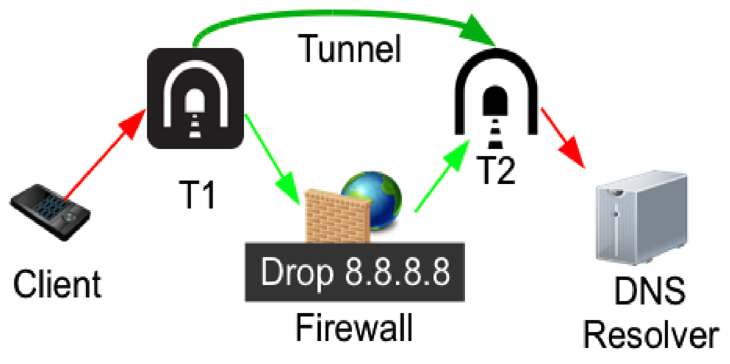


Figure 10: An example network containing a firewall and a tunnel

9.2. Tunnel

A client host is about to do a DNS lookup using the Google Public DNS resolver (8.8.8.8). The network operator's security policy disallows external resolvers by deploying a firewall that explicitly forbids packets sent to IP address 8.8.8.8, as shown in Figure 10.

We would like to know how the tunnel should be configured to allow the client to send packets to its initial destination, 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 host T_2. At this point, the constraints on the IP destination address lead to the answer: '⌐ 8.8.8.8'. In other words, any address different from 8.8.8.8.

Hence, as long as the IP destination address is different from 8.8.8.8, the flow will reach . 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:

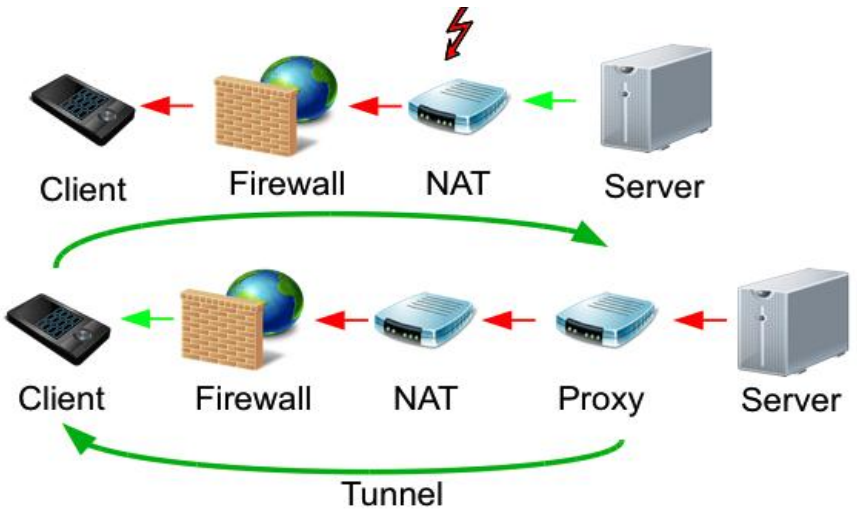


Figure 11. Inbound connectivity in cellular networks

9.3. 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 Veriph check for connectivity (Figure 11, top) between the server and the mobile application.

This fails because the NAT does not find the proper state variables in the flow, those that should have been set once a request leaves the private network.

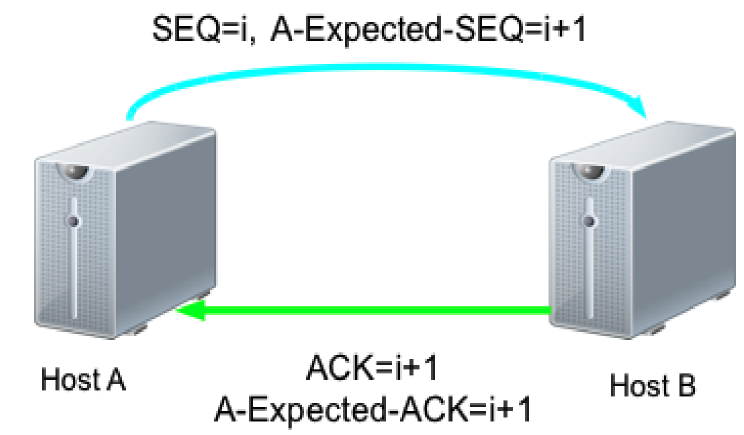
The standard solution to this issue is to use a proxy server outside the NAT, to which the client establishes a long running connection. The proxy then forwards requests to the mobile, as shown in Figure 11, bottom.

We check this configuration in two steps:

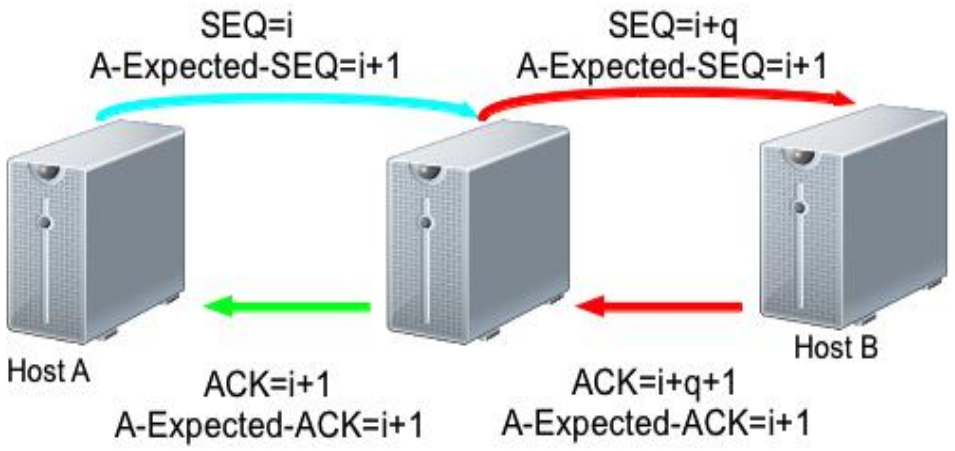
First, we model opening a connection from the mobile client to the proxy. As the firewall allows this connection, the TCP flow has all the state variables pushed by the stateful devices: firewall, NAT and proxy server.

In step two, the outside connectivity request is tunneled to the mobile using the previous TCP flow as an outer header. Veriph proves 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.

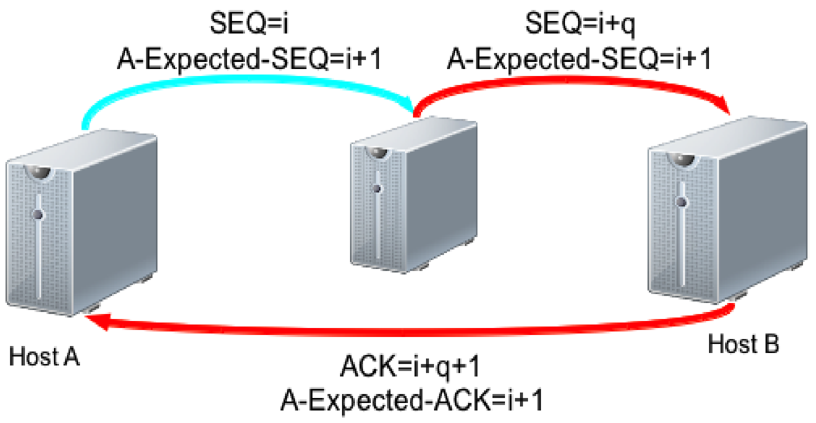
More so, the true identity of the remote end is hidden by the proxy—rendering the firewall useless because it cannot filter blacklisted IP addresses.



(a) Direct communication



(b) Bidirectional sequence number changes



(c) Unidirectional sequence number changes

Figure 12: Modeling sequence number consistency in TCP

9.4. Modeling Middle-boxes 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 (Initial Sequence Number) 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 middle-boxes?

We first model the case when there is no middle-box (Figure 12, top). Besides the regular TCP addresses, we also model sequence number and the ACK field.

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 flow 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. I omit the sequence number in the ACK packet from the figure 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 12 (bottom) 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, suppose the return traffic *B−A* does not pass through the middle-box (Figure 12, middle). This time, the test fails: B sets the ACK field to 1 + SEQ, but SEQ has already been altered by the middle-box. Host A cannot match Expected-ACK and ACK for validating the response.

10. Scalability

To validate the complexity analysis, in Figure 13 I plot the time needed to check increasingly large networks. The results confirm that Veriph time scales linearly with the size of the network, under the assumptions previously mentioned in Section 5.2 and 5.3.

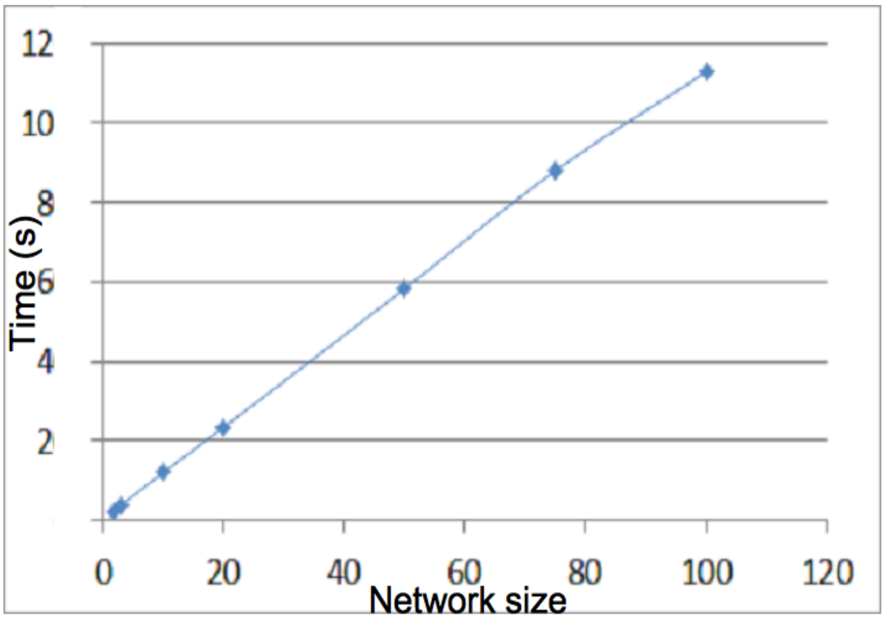


Figure 13: Veriph scales linearly

11. Conclusions

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

This work shows that modeling and statically analyzing stateful networks can be done in a scalable way. We model packet headers as variables and use symbolic execution to capture basic network properties; middle-box flow state can also be easily modeled using such header variables. We have proven our solution is correct and its complexity is linear.

I have implemented these algorithms in Veriph, a tool for checking stateful networks. Veriph takes middle-box descriptions written in Click together with the network topology and allows us to verify and understand a variety of use-cases.

BIBLIOGRAPHY

[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] 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

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

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

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

[6] Google Global Cache https://peering.google.com/about/ggc.html

[7] Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs - Krishnamoorthy, S. ; Dept. of Electr. & Comput. Eng., Virginia Tech, Blacksburg, VA, USA ; Hsiao, M.S. ; Lingappan, L. - Test Symposium (ATS), 2010 19th IEEE Asian

[8] Software Dataplane Verification - Mihai Dobrescu and Katerina Argyraki, École Polytechnique Fédérale de Lausanne, NSDI 2014

[9] Wiley Encyclopedia of Computer Science and Engineering - 2007 John Wiley & Sons, Inc.

[10] Alfred Tarski. A lattice-theoretical fix-point theorem and its applications. 1955.

[11] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2008.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI), 2005.

[13] The Definitive ANTLR 4 Reference, Terence Parr, Pragprog Publishing 2007

[14] SymNet: Static Checking for Stateful Networks: Radu Stoenescu, Matei Popovici, Lorina Negreanu, Costin Raiciu, Hotmiddleboxes 2013

[15] ClickOS and the Art of Network Function Virtualization - Joao Martins and Mohamed Ahmed, NEC Europe Ltd.; Costin Raiciu and Vladimir Olteanu, University Politehnica of Bucharest; Michio Honda, Roberto Bifulco, and Felipe Huici, NEC Europe Ltd.