A Declarative Approach for Global Network Security Configuration Verification and Evaluation

Mohammad Ashiqur Rahman and Ehab Al-Shaer
Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
{mrahman4, ealshaer}@uncc.edu

Abstract— **With the increasing number of security devices and rules in the network, the complexity of detecting and tracing network security configuration errors become a very challenging task. This in turn increases the potential of security breaches due to rule conflicts, requirement violations or lack of security hardening. Most of the existing tools are either limited in scope as they do not offer a global analysis of different network devices or hard to comprehensively use because these tools are not declarative. Declarative logic programming can readily express network configurations and security requirements for verification analysis. In this paper, we use Prolog to model the entire network security configurations including topology, routing, firewall and IPSec. This is implemented in a tool called ConfigAnalyzer, which was also evaluated with large network and policy sizes. The tool allows for verifying reachability and security properties in flexible and expressive manner. It also allows for evaluating security configurations in terms of accessibilities credentials and rules.**

Keywords- declarative language, network configuration, policy verification, declarative queries, security measures

I. INTRODUCTION

As network size grow significantly in terms of number of nodes, policies and uses, the complexity of network configurations also grow exponentially. As a result, the potential of misconfiguration and security hazards spread in dimension and variety. Many network and security devices (such as routers, firewall and IPSec) work based on large policy rules distributed on the network.

A firewall is a traffic filtering device that commonly contains thousands of rules. IPSec (Internet Protocol Security) provides integrity by authentication and confidentiality of data communication over networks. IPSec devices transform (encrypt and may encapsulate) the outgoing packet, while the receiving IPSec devices detransform (decrypt and, if required, decapsulate) these transformed packets. IPSec is widely used to establish Virtual Private Networks (VPNs) between corporate networks over the Internet. Similar to firewall, an IPSec must have a policy to specify how to protect passing traffic.

Although each security device has its own local policy, policies of different devices are logically interacting to implement a global end-to-end security requirement of the network. Thus, policy inconsistencies and violations are most common misconfiguration problems in today's' networks [3, 12]. To ensure correct security policy enforcement, the firewall and IPSec configuration policies must conform to the routing topology. Due to the evolvement of wireless and mobile communications, reachability and security requirements must remain valid. For example, routers, firewalls and IPSec are different devices but they must be consistently configured to ensure reachability and security simultaneously. In each device, policy rules are required to be written in proper order and taking other policies into consideration, which are challenging and error prone tasks as the rules are written manually and locally.

Many recent studies show the application of declarative language in networking [6, 7]. Some of these studies address configuration problems, like routing, by writing query that recursively invokes itself to diagnose a problem. Declarative language is flexible to express network and easy to adopt new applications. Declarative language can be very useful to express and verify the correctness of policy configurations.

In this paper, we model the entire network configurations using Prolog, a declarative language [14]. We present a simplified but comprehensive networking model that includes routing, firewalls and IPSec polices. We write declarative representations of network topology, routing entries, firewall rules and IPSec policies. We then show how to verify reachability and security requirements across the entire network. For example, we can verify that whether a packet can reach its destination securely (authentic or confidential).

We organize the remaining of the paper as follows. In Section II, we define our declarative approach for modeling network configuration. Different verification applications based on our model are presented in Section III. We evaluate our declarative model in Section IV. We discuss the related works in Section V. Lastly, in the conclusion we write summary of our work along with further research steps.

II. NETWORK MODELING

In this section, we present the declarative modeling of basic networking and packet routing. Next, we extend the model by adding firewall and IPSec devices. We model the network using declarative logic programming language, Prolog [14]. We consider a network with two kinds of nodes: hosts and routers. The connection between any two nodes is a link. Links are bidirectional. We assume a centralized database with the configuration data of all nodes of the network. If any configuration change has occurred in a node, that will be updated in the centralized database. Our model and queries will be executed on this database.

A. Basic Model

In the basic model no security device is considered.

1) Topology:

Figure 1: A simple network.

Each participant in a network is a node. A node can be either host or router. So, we define two fact predicates: `host(N)` and `router(N)`. First predicate denotes that `N` is a host, while the second specifies `N` as a router.

We also write fact `faulty(N)` to denote a node `N` is faulty (not working). This will give the opportunity to see the effect if a particular node would fail.

We define *subnet* as a group of nodes in a simple way using two fact predicates: `subnet` and `subnetMember`. Predicates `subnet(Sn)` states that `Sn` is a subnet, while `subnetMember(N,Sn)` tells that `N` is a member of a subnet `Sn`. `N` can be a subnet (group within group) other than a node.

We define fact `link(N,T,U)` to mean that two nodes, `N` and `T`, are directly connected through a link. `U` stands for either `up` or `down`, which means the link is 'up' or 'down' respectively. Possible values for `U` are bounded by declaring, for example, `linkState(up)`. One can alter `U` of a link to see its effect on the networking. As links are bidirectional, we derive `slink(N,T,U)` from `link(N,T,U)` writing the following rules:

```
slink(N,T,U):- (link(N,T,U) ; link(T,N,U)).
```

2) Routing entries:

Each router maintains a routing table consists of a number of routing entries. We represent each routing table entry using `rtEntry(N,L,T)`. It means that router `N` forwards any packet destined to a node in `L` (a list of nodes or subnets) to node `T` (next hop). We use `rtLookup(N,D,T)` to lookup the next hop for the traffic destined to `D` from `rtEntry(N,L,T)`. It checks the routing entries and picks the first matching non-faulty next hop.

```
routeLookup(N,D,T):- rtEntry(N,DL,T),
     member(D,DL), not(faulty(T)),!.
```

3) Traffic forwarding:

```
R1.1: forward(N,S,D,D):- slink(N,D,up).
R1.2: forward(N,N,D,T):- host(N),slink(N,T,up),
        router(T).
R1.3: forward(N,S,D,T):- rtLookup(N,D,T),
        slink(N,T,up).
```

Predicate `forward(N,S,D,T)` denotes that `N` forwards any traffic between source `S` and destination `D` to `T`. From now throughout the paper we will specify any traffic or packet with its source `S` and destination `D` as `<S,D>`. R1.1 to R1.3 are the rules for packet forwarding. R1.1 works when a forwarding node is directly connected with the destination. R1.2 acts if the forwarding node is a host and it is the source of the traffic. The host forwards the packet to the router to which it is connected. It is assumed that a host is connected to only one router (gateway). R1.3 executes when `N` is a router. Note that forwarding is possible only when corresponding link is up.

4) Tracing rout and reachability:

Recursive predicate rule `findRoute(N, G, S, D, L)` is defined to find the path `L` (list of nodes) from `N` to `G` (goal node) for `<S, D>` packet. Usually `G` and `D` are same except when tunnel based transformation is applied on the packet. The predicate (R2.2) can be interpreted as `N` forwards `<S, D>` packet to `T` (a link must be from `N` to `T`) and then `T` finds the remaining path from `T` to `G`. R2.1 is the base rule, when starting and ending nodes of a path are same.

```
R2.1: findRoute(N,N,_,_,[N]):- not(faulty(N)).
R2.2: findRoute(N,G,S,D,[N|F]):- not(faulty(N)),
        forward(N,S,D,T), findRoute(T,G,S,D,F).
R2.3: traceRoute(S,D,F):- findRoute (S,D,S,D,F).
```

Predicate `traceRoute(S,D,F)` finds the communication path `F` of traffic `<S,D>` by invoking `findRoute`. We write `reachable(S,D)` using `traceRoute` to find the traffic reachability from `S` to `D`:

```
reachable(S,D):- traceRoute(S,D,_).
```

5) Example:

Here we present a simple example to explain the way `traceRoute` finds a traffic path. Consider the simple network shown in Fig 1. Here we have 4 nodes, `h1`, `h2`, `r1` and `r2`. First two nodes are hosts and the last two are routers. The links between the nodes are shown in the figure. Assume that all fact clauses are written associated to the hosts, routers and links. We have the following routing entries:

```
rtEntry(r1,[h2],r2).
rtEntry(r2,[h1],r1).
```

If we call `traceRoute(h1,h2,F)` now, it gives us the path `[h1, r1, r2, h2]` between `h1` and `h2` through the following steps of predicates' invocations:

```
traceRoute(h1,h2,F):- findRoute (h1,h2,h1,h2,F).
findRoute(h1,h2,h1,h2,[h1|F]):- not(faulty(h1)),
        forward(h1,h1,h2,T),
        findRoute(T,h2,h1,h2,F). % T is r1
findRoute(r1,h2,h1,h2,[r1|F]):- not(faulty(r1)),
        forward(r1,h1,h2,T),
        findRoute(T,h2,h1,h2,F). % T is r2
findRoute(r2,h2,h1,h2,[r2|F]):- not(faulty(r2)),
        forward(r2,h1,h2,T),
        findRoute(T,h2,h1,h2,F). % T is h2
findRoute(h2,h2,h1,h2,[h2]):- not(faulty(h2)).
```

Here `findRoute` predicate is recursively queried until it reaches its base query. After the backpropagation of the `findRoute` queries, `traceRoute(h1,h2,F)` returns true and `F` represents the path.

B. Extended Model with Security Devices

In this section, we extend the basic model by incorporating firewall and IPSec to have a complete network model. Firewalls and IPSec devices are most often implemented on routers. They check the incoming traffic and forward the traffic to next hop according to its policy. Here we consider that a firewall or an IPSec is built on a host or a router. We write following two fact predicates to identify a node `N` as firewall or IPSec respectively using `firewall(N, I)` or `ipsec(N, I)`.

The argument `I` can have any of two values: `active` and `inactive`, which denote the status of the firewall (or IPSec) that whether its security policy is in use or not. When the state

is inactive, it means that the node will deal with the traffic without applying any filtering policy. We bound the values of I by writing predicate deviceState (similar to linkState). This device state will give the opportunity to understand the situation in absence of a security device.

1) Modeling firewall policy:

Firewall works according to its *access control policy*. It matches an incoming packet with the policy and forwards the packet if policy allows; otherwise it discards the packet. We write any rule of a firewall policy in a simple form: <src><dst><act>. A particular traffic is identified by its src (source) and dst (destination) nodes, while act is any of allow (forward) or deny (discard). We write a firewall rule using firewallRule(N,S,D,A) fact predicate, which tells about the action A taken by firewall N for <S,D> traffic. Possible firewall actions are declared by firewallAction.

Firewall executes the action according to the first matching rule of its policy. Hence, that a firewall permits a particular traffic to forward be defined using the predicate rules below:

```
R3.1: matchFirewallRule(N,S,D,A):-
        firewallRule(N,S,D,A),!.
R3.2: firewallPermit(N,S,D):-
        matchFirewallRule(N,S,D,A),A=allow.
```

In R3.1, predicate matchFirewallRule finds the first matching rule's action A for a particular traffic <S, D>. Next predicate firewallPermit checks that if the matching rule allows the traffic.

2) Modeling IPSec policy:

An IPSec, like a firewall, checks incoming packets according to its access control policy. However, unlike firewalls, possible actions for an IPSec access rule are bypass, discard and protect. Actions bypass and discard are analogous to allow and deny actions of a firewall. But protect means bypass with protection by applying *transformation* on the packet according to the *mapping policy*. A mapping rule describes how the packet will be transformed and which will be the destination IPSec device. The destination IPSec node does reverse the transformation, so that the original packet is retrieved. We will use *detransformation* to cite this reverse-transformation. There are two modes of IPSec transformations: transport (only between the traffic endpoints) and tunnel (normally between two IPSec gateways). Again, any mode of transformation uses either esp (ESP: Encapsulated Security Protocol) or AH (Authentication Header protocol).

We write any access rule and its corresponding mapping rule using following two simple patterns respectively:

```
<src><dst><act>
<src><dst><ipsec-src><ipsec-dst><mode><proto>
```

Here ipsec-src and ipsec-dst stand for the source IPSec node and the destination IPSec node. Terms mode and proto denote respectively IPSec mode and protocol that are to be applied on the traffic. Predicates ipsecRule(N,S,D,A) and ipsecMapRule(N,S,D,IS,ID,M,P) are written to specify an IPSec access rule and an IPSec mapping rule respectively. Among the arguments, IS and ID stand for ipsec-src and ipsec-dst, while M and P refer to mode and proto respectively. Argument P is a list- [Pr,Pm], where Pr is the protocol name and Pm is the list of corresponding algorithms. Allowed values for A, M, Pr and Pm are declared by ipsecAction, ipsecMode, ipsecProtocol and ipsecParam. We write identical mapping rule in source and destination IPSec nodes to denote the *security association* (SA) between them.

We use a variable made up of two lists for a packet to represent any applied transformation on it. We can assume it as IPSec header. The first list is used to represent the protocol in use along with its parameters. The second list denotes the encapsulated packet's source and destination and thus indirectly used to represent the mode. If it is empty, then the transformation is transport. For multiple transformations on a packet, these IPSec headers are added one after one as a stack. This representation will be utilized to get information about the security transformations applied on a packet.

```
R4.1: matchIpsecRule(N,S,D,A):-
        ipsecRule(N,S,D,A),!.
R4.2: nextIPSecMapRule(N,S,D,ID,M,P,R):-
        ipsecMapRule(N,S,D,N,ID,M,P),
        T=[S,D,ID,M,P],not(member(T,R)).
R4.3: secAssociation(N,S,D,ID,M,P):-
        ipsecMapRule(ID,S,D,N,ID,M,P).
R4.4: transformation (N,S,D,E,E,[S,D],R):-
        not(nextIPSecMapRule(N,S,D,_,_,_,R)).
R4.5: transformation (N,S,D,E,NE,H,R):-
        nextIPSecMapRule(N,S,D,ID,M,P,R),
        secAssociation(N,S,D,ID,M,P),
        NR=[[S,D,ID,M,P]|R],
        ((M = transport,[S,D]=[N,ID],
         transformation(N,S,D,[[P,[]]|E],NE,H,NR));
         (M=tunnel,
         transformation(N,N,ID,[[P,[S,D]]|E],NE,H,NR))).
R4.6: isTransformation(N,S,N,E,E,[S,N]).
R4.7: isTransformation(N,S,D,E,E,[S,D]):-
        matchIpsecRule(N,S,D,A),A = bypass.
R4.8: isTransformation(N,S,D,E,NE,H):-
        matchIpsecRule(N,S,D,A),A = protect,
        transformation(N,S,D,E,NE,H,[]).
```

When an IPSec node receives a packet, it checks if the packet deserves any detransformation (i.e., if it is the destination IPSec of the packet). Next, it checks the access policy and takes actions accordingly on this packet. An IPSec may need to do the both. We define ipsecPermit that invokes isDetransformation and isTransformation to do the detransformation and transformation parts of the IPSec process.

There may be multiple mapping rules applicable for a packet, that is, once a packet is encapsulated, the new packet may be applicable for another rule for further transformation. Predicate transformation considers this point and so it is defined recursively (R4.4–R4.5). R4.4 is the base rule when there is no more mapping rule for the packet. The arguments E and NE in isTransformation stand for the IPSec header of the packet respectively before and after transformation. H stands for the source and destination (IP header) of the packet after transformation. If transport based transformation is executed, H is just [S,D]. Last argument R is required to find the next matching mapping rule. It is the record of already applied mapping rules. So, the next mapping rule is exclusive of those.

Predicate `matchIpsecRule` searches the first matching access rule for the IPSec, while `nextIPSecMapRule` gives the next matching mapping rules (with the consideration of `R`). Predicate `secAssociation` finds the corresponding mapping rule in the destination IPSec `ID`. Predicate `isTransformation` checks if the action is `bypass` (R4.7) or `protect` (R4.8). For the latter case, the packet is transformed by invoking `transformation`. R4.6 acts when the IPSec node is the destination of the traffic; so no transformation is applied.

```
R4.9: detransformation(N,S,N,[[P,[]]|E],NE,H,R):-
        nextIPSecMapRule(N,N,S,_,transport,P,R),
        NR = [[N,S,N,transport,P]|R],
        detransformation(N,S,N,E,NE,H,NR).
R4.10: detransformation(N,S,N,[[P,[ES,ED]]|E],NE,H,R):-
        nextIPSecMapRule(N,ED,ES,S,tunnel,P,R),
        NR = [[ED,ES,S,tunnel,P]|R],
        detransformation(N,ES,ED,E,NE,H,NR).
R4.11: detransformation(N,S,D,E,E,[S,D],R):-
        not(nextIPSecMapRule(N,D,S,_,_,_,R)).
R4.12: isDetransformation(N,N,D,E,E,[N,D]).
R4.13: isDetransformation(N,S,D,E,NE,H):-
        detransformation(N,S,D,E,NE,H,[]).
```

Predicate `detransformation(A,S,D,E,NE,H,R)` takes similar arguments as `transformation`. R4.9 and R4.10 do detransformation of the packet in the transport mode and the tunnel mode respectively. Note that, multiple detransformations may be applicable for a packet, that is, once a packet is detransformed, it may be applicable for another detransformation. Predicate `detransformation` reflects this matter. R4.11 is the end of `detransformation` when there is no (more) detransformation applicable for the packet. Predicate `isDetransformation` initiates `detransformation` in R4.13 with necessary arguments. R4.12 acts when the IPSec node is the source of the traffic.

3) Tracing rout and reachability:

In order to incorporate firewall and IPSec in tracing route, we extend `findRoute` as follows:

```
R5.1: findRoute(N,N,S,D,E,[[N,[]]]):- not(faulty(N)),
        ((not(ipsec(N,active)),E = [], N = D);
       (ipsec(N,active),ipsecPermit(N,S,D,E,[],[_,N]))).
R5.2: findRoute(N,G,S,D,E,[[N,NE]|F]):-
        not(faulty(N)),ipsec(N,active),
        ipsecPermit(N,S,D,E,NE,[NS,ND]),
        forward(N,NS,ND,T),findRoute(T,G,NS,ND,NE,F).
R5.3: findRoute(N,G,S,D,E,[[N,E]|F]):- not(faulty(N)),
        firewall(N,active),firewallPermit(N,S,D),
        forward(N,S,D,T),findRoute(T,G,S,D,E,F).
R5.4: findRoute(N,G,S,D,E,[[N,E]|F]):- not(faulty(N)),
        not(firewall(N,active);ipsec(N,active)),
        forward(N,S,D,T),findRoute(T,G,S,D,E,F).
```

Predicate `findRoute(N,G,S,D,E,F)` is modified to take one more argument `E`. It denotes the IPSec headers when the predicate is called. Argument `F` also includes more than before. `F` represents the path including IPSec header of the packet at each node. `F` shows that whether (and how) a packet was transformed when it was being forwarded. In this recursive process, R5.1 is the base rule, when the packet reaches the goal node (in plain text as `findRoute` is invoked with plain text). R5.2 and R5.3 work when the node is an IPSec and a firewall respectively. R5.4 is for a common router or a host (no security device). With the change in `findRoute` predicate, we have to modify the reachability queries (R2.3) also as follows.

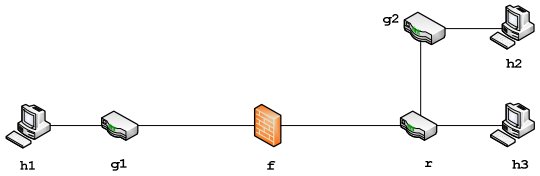

Figure 2: A network with IPSec and firewall

```
R5.5: traceRoute(S,D,F):- findRoute(S,D,S,D,F).
```

4) Example:

Let we have a network with 7 nodes and 6 links as shown in Fig. 2. Among the nodes, `h1`, `h2` and `h3` are hosts. Consider that, all `host`, `router` and `link` facts are written. We have two IPSec devices `g1` and `g2` and one firewall `f`.

Let `f` allows all traffic. For the IPSec nodes we have the following rules:

```
ipsecRule(g1,h1,h2,protect).
ipsecMapRule(g1,h1,h2,g1,g2,tunnel,[esp,[hmac,des]]).
ipsecMapRule(g2,h1,h2,g1,g2,tunnel,[esp,[hmac,des]]).
```

According to the rules we see that, the packet from `h1` to `h2` will be transformed at `g1` and detransformed at `g2`. Let us see the path that `traceRoute` finds for the packets <`h1`, `h2`> and <`h1`, `h3`>. Call of `traceRoute(h1,h2,F)` satisfies with following `F`:

```
F = [[h1, []], [g1, [[[esp,[hmac,des]], [h1,h2]]]], [f,
[[[esp,[hmac,des]], [h1,h2]]]], [r, [[[esp,[hmac,des]],
[h1,h2]]]], [g2, []], [h2, []]].
```

The `<h1,h2>` traffic is transformed (tunnel based) at `g1` (`[[[esp,[hmac,des]], [h1,h2]]]`) to a new traffic `<g1,g2>` and it is routed to `g2`. At `g2`, `<g1,g2>` is detransformed to `<h1,h2>` and then it is forwarded to `h2`. However, call of `traceRoute(h1,h3,F)` satisfies with `F` as `[h1,[]], [g1,[]], [f,[]], [r,[]], [h2,[]]]`. Empty IPSec headers in `F` mean that no transformation has been applied on the traffic. So, with the reachability `traceRoute` also provides information about the traffic's security transformation.

III. Applications

In this section we will state different queries based on our proposed declarative network model. Our main goal is to verify network security configuration by exploiting traffic paths. Firstly, we will discuss the queries that provide basic security analysis from reachability. Next we will present some query examples on security verification and evaluation.

A. Reachability Analysis with Security Constraints

Useful queries are possible to check for required security constraints based on reachability. All our analysis is based on the result of `traceRoute` query, especially on its `F` argument. `F` is examined for applied security measures on the traffic.

1) Traffic security checking:

We write `securePath` (in Listing 1) query that finds whether traffic from a particular source to destination is secured (i.e., undergone by IPSec transformation) throughout a particular path. Security is identified by IPSec transformation mode and protocol.

Query `securePath(X,Y,S,D,M,Pr)` finds whether a packet `<S,D>` passes a path from `X` to `Y` as being transformed in `M` mode and with `Pr` protocol. Predicate `findInPath` searches a node `N` in

F. This is checked by analyzing F of the traffic path. Let consider the network of Fig. 2. If we invoke securePath(g1,g2,h1,h2,tunnel,esp) it returns true, while securePath(h1,g2,h1,h2,tunnel,esp) returns false. The latter call returns false, because the <h1, h2> traffic is not transformed from h1 to g1.

```
secureLink(N,N,N,[[N,_]|F],_,_):-
        not(findInPath(N,F)).
secureLink(N,N,Y,[[N,E]|F],M,Pr):-
        findInEncHdr(M,Pr,E), F = [[T,_]|_],
        secureLink(T,T,Y,F,M,Pr).
secureLink(N,X,Y,[[N,_]|F],M,Pr):- not(N=X),
        F=[[T,_]|_], secureLink(T,X,Y,F,M,Pr).

securePath(X,Y,S,D,M,Pr):- traceRoute(S,D,F),
        secureLink(S,X,Y,F,M,Pr).
```

Listing 1: A query for secure path

We can remove the IPSec mode argument in the query securePath. The reason behind it is as follows. The mode basically depends on the transformation and detransformation ends of an IPSec operation. If the ends are same as the ends of the corresponding traffic, then transport mode transformation is normally used, since tunnel mode transformation only adds an extra header to the packet from the same source and destination. In other cases tunnel mode transformations have to be used. From now, we will not include the IPSec mode in the queries to identify the type of security.

2) All hosts' reachability:

```
canBeReached(D,Pr,[],_).
canBeReached(D,Pr,[S|HL],[S|SL]):- reachable(S,D),
        canBeReached(S,HL,SL).
canBeReached(D,Pr,[S|HL],SL):- not(reachable(S,D)),
        canBeReached(S,HL,SL).

allSrc(D,Pr,SL):- findall(S,host(S),HL),
        canBeReached(D,Pr,HL,SL).
```

Listing 2: A query to find all the reachable hosts from a particular host

We write allSrc query to find the hosts (sources) that can reach a particular node (destination) with any security constraints. allSrc (in Listing 2) first invoke findall to find the list of all hosts (HL), which are the possible traffic sources to D. Then it finds all the sources (SL) among the HL that can reach D. Security type is specified by Pr. Similarly we write allDest query to find all hosts (destinations) that are reachable by a particular host (source).

3) Group (subnet) specific queries:

```
trafficPass(_,_,_,_,[],[]).
trafficPass(X,Y, M,Pr,[[S,D]|PL],[[S,D]|TL]):-
        isSecurePath(X,Y,S,D,M,Pr),
        trafficPass(X,Y, M,Pr,PL,TL).
trafficPass(X,Y, M,Pr,[[S,D]|PL],TL):-
        not(isSecurePath(X,Y,S,D,M,Pr)),
        trafficPass(X,Y, M,Pr,PL,TL).

subnetInTrafficByPath(Sn,X,Y,M,Pr,TL):-
        findall([S,D],subnetInTraffic(Sn,S,D),PL),
        trafficPass(X,Y,M,Pr,PL,TL).
```

Listing 3: A query to find all the incoming traffic reached to a subnet

We can write the queries specific to a subnet, that is, all incoming traffics to a subnet or all outgoing traffics from the subnet.

In Listing 3, predicate subnetInTraffic checks that whether traffic <S,D> is an incoming traffic to a subnet S. Similar query can be written for subnet outgoing traffic.

4) Nested traffic security:

Nested (multiple) IPSec transformations are used to employ security in depth. So, considering nested transformations, we can query if a packet undergoes security in depth.

```
transformationDepth(N,N,N,[[N,_]|F],_,W):-
        not(findInPath(N,F)), setMax(W).
transformationDepth(N,N,Y,[[N,E]|F],Pr,W):-
        depthIpsecHdr(E,Pr,W1),F=[[T,_]|_],
        transformationDepth(T,T,Y,F,Pr,W2),
        min(W1,W2,W).
transformationDepth(N,X,Y,[[N,_]|F],Pr,W):-
        not(N=x), F=[[T,_]|_],
        transformationDepth(T,X,Y,F,Pr,W).

transformationDepthByPath(X,Y,S,D,Pr,W):-
        traceRoute(S,D,F),
        transformationDepth(S,X,Y,F,Pr,W).
```

Listing 4: A query to find the minimum number of transformations

Query transformastionInDepth (in Listing 4) finds out the minimum number of nested transformations that the <S,D> traffic experiences at a specific portion (X to Y) of the traffic path. Among the invoked predicates, depthIpsecHdr finds the number of transformations applied on a packet by analyzing its IPSec header. Predicate findInPath searches a node N in path F, while predicate min returns the minimum value of first two arguments in the third argument.

B. Security Verification and Evaluation

In this subsection we will describe the usefulness of our model for security configuration verification and evaluation.

1) Backdoor discovering:

Let h1, h2 and h3 are hosts and other nodes are routers in as shown in Fig. 3. Among the other nodes, f acts as a firewall. Consider the following routing entries for the traffic destination h2 at r1 and r3 routers:

```
rtEntry(r1,[h2],f).
rtEntry(r1,[h2],r3).
rtEntry(r3,[h2],r2).
```

We see that for traffic destined to h2, r1 has two routing entries. The first entry is the primary routing entry for h2, while the latter is the secondary routing entry. The primary entry forwards the any traffic destined to h2 to f, while the secondary entry forwards the same traffic to r3. If the next hop of the primary routing entry is not alive (faulty), then r1 will go for the latter entries. Assume, the access control rule at f for any traffic destined to d is: firewallRule(f,_,h2,deny). Now, if

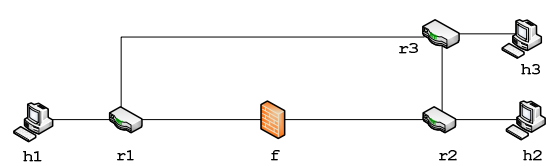

Figure 3: Backdoor

we invoke `traceRoute(h1,h2,F)`, the query is unsatisfied, since firewall rule denies the traffic. But, if we make `f` dead by invoking `faulty(f)`, `traceRoute(h1,h2,F)` becomes true. Hence, we easily understand that there is backdoor for the traffic. From `F = [h1, r1, r3, h2]`, one can figure out the backdoor. Therefore, any misplacement of firewall and routers can easily be identified. We write `backdoor(S,D,B)`, as shown in Listing 5, to automate this backdoor finding for a particular source and destination considering only firewalls. Here, argument `B` is the list of backdoor paths in from `S` to `D`. Among the predicates, `firewallInPath(L,FL)` finds the list of firewalls `FL` that exist in a traffic path `L`, `activate(FL)` makes all the firewalls in the given list `FL` to `active` while `inactivate(FL)` does the opposite. We have utilized built-in predicates `assert` and `retract` to change the facts.

```
policyImposed(S,D,L,I):- firewallInPath(L,FL),
        activate(FL), ((not(traceRoute(S,D,_)),I=y);
        (traceRoute(S,D,_), I=n)), inactivate(FL).

findBackdoor(_,_,[],[]).
findBackdoor(S,D,[[T|_]|L],B):-(S==T ; D==T),
        findBackdoor(S,D,L,B).
findBackdoor(S,D,[[T|_]|L],B):- (S\==T, D\==T),
        assert(faultD(T)), traceRoute(S,D,L1),
        policyImposed(S,D,L1,I),retract(faulty(T)),
        ((I=y, findBackdoor(S,D,L,B));
         (I=n, findBackdoor(S,D,L,B1), B=[L1|B1])).

findBackdoor(S,D,[[T|_]|L],L):- (S\==T, D\==T),
        faulty(T), not(traceRoute(S,D,_)),
        retract(faulty(T)), findBackdoor(S,D,L,B).

backdoor(S,D,B):-
        not(traceRoute(S,D,_)),
        findall(F,firewall(F,_),FL), inactivate(FL),
        traceRoute(S,D,L),findBackdoor(S,D,L,B),
        activate(FL).
```

Listing 5: A query to search backdoor between two hosts

2) Policy verification based on reachability:

It is common to see multiple IPSec transformations on the same packet. Transformation over an already transformed packet is known as nested transformation. However, nested transformation may not be valid based on security goal. Let `h1`, `h2`, `h3` and `h4` are hosts in the network shown in Fig. 4. Assume all necessary routing entries are present. Here `f` is a firewall and it allows all traffic. Now consider the following set of IPSec rules:

```
ipsecRule(x,h1,h4,protect).
ipsecMapRule(x,h1,h4,x,v,tunnel,[esp,[hmac,des]]).
ipsecMapRule(v,h1,h4,x,v,tunnel,[esp,[hmac,des]]).
ipsecRule(u,x,v,protect).
ipsecMapRule(u,x,v,u,y,tunnel,[esp,[hmac,des]]).
ipsecMapRule(y,x,v,u,y,tunnel,[esp,[hmac,des]]).
```

According to the rules, there are two nested tunnels: first

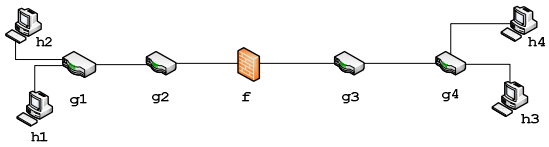

Figure 4: Invalid nested IPSec tunnel transformations

tunnel from `x` to `v` on `<h1,h4>` packet and then another tunnel from `u` to `y` on `<x,v>` packet (encapsulated `<h1,h4>` packet). The call of `traceRoute(h1,h4,F)` satisfies. However, call of `securePath(u,y,h1,h4,esp)` fails, that is, the `<h1,h4>` packet is not secure throughout the path from `u` to `y`. Let us observe `F`.

```
F = [[h1, []], [x, [[[esp,...], [h1,h4]]]], [u,
[[[esp,...], [x,v]], [[esp,...], [h1,h4]]]], [f,
[[[esp,...], [x,v]], [[esp,...], [h1,h4]]]], [v,
[[[esp,...], [x,v]], [[esp,...], [h1,h4]]]], [y,
[[[esp,...], [h1,h4]]]], [v, []], [y, []], [h4, []]]
```

We use '...' to make `F` short. From `F` it is found that, at `v` when `<u,y>` packet is reached, it is not detransformed (since no rule does match), rather it is forwarded to the destination `y`. At `y`, when the packet is detransformed, packet `<x,v>` is rerouted to `v`. Now `v` detransforms the packet and forwards `<h1,h4>` packet to `y`. Hence, ultimately plain text, `<h1,h4>` packet, is reached at `y` from `v` and so `<h1,h4>` packet does not reach `y` safely.

Policies among multiple firewalls and IPSec devices often conflict. So, desired reachability may not be possible. From `traceRoute` one can see these situations. Let, two IPSec devices `x` and `y` (in Fig. 4) are negotiated to have tunnel based transformation for `<h1,h3>` traffic from `x` to `y`:

```
ipsecRule(x,h1,h3,protect).
ipsecMapRule(x,h1,h3,x,y,tunnel,[esp,….]).
ipsecMapRule(y,h3,h1,x,y,tunnel,[esp,….]).
```

Let, firewall `f` allows `<h1,h3>` traffic, but no `<x,y>` (fall under default deny):

```
firewallRule(f,h1,h3,allow).
firewallRule(f,_,_,deny).
```

Here the invocation of `reachable(h1,h3)` query fails as the packet `<x,y>` (`<h1,h3>` is encapsulated into `<x,y>` at `x`) is discarded at `f`.

3) Security Evaluation:

We can do queries for fan-out and fan-in for a subnet. We write `fanOut(Sn,DL)` (in Listing 6) by utilizing `allDest`, which finds the outgoing traffic list `DL` from the given subnet `Sn`. Similarly, for incoming traffic we write `fanIn(Sn,SL)` by utilizing `allSrc`.

```
allReachedDest([],0).
allReachedDest([N|HL],DL):- allReachedDest(HL,DL1),
        allDest(N,DL2),append(DL1,DL2,DL).

fanOut(Sn,DL):-
        findall(H,(subnetMember(H,Sn),host(H)),HL),
        allReachedDest(HL,DL).

lowerPrivilege(Sn1,Sn2):- fanOut(Sn1,DL1),
        fanOut(Sn2,DL2), length(DL1,L1),
        length(DL2,L2), L1 < L2.
```

Listing 6: A query to compare by privilege based on outgoing traffic

These queries are useful in different ways. In order to ensure security, it is required to give minimum privilege to the hosts but not less than the essential privilege to do business requirement. Firewalls are used to control privilege. Network administrator can get an idea about the privilege level of a subnet by observing the fan-out of the subnet. One can assume that the more is the fan-out, the more is the privilege. Hence, based on fan-out, two subnets can be compared with respect to

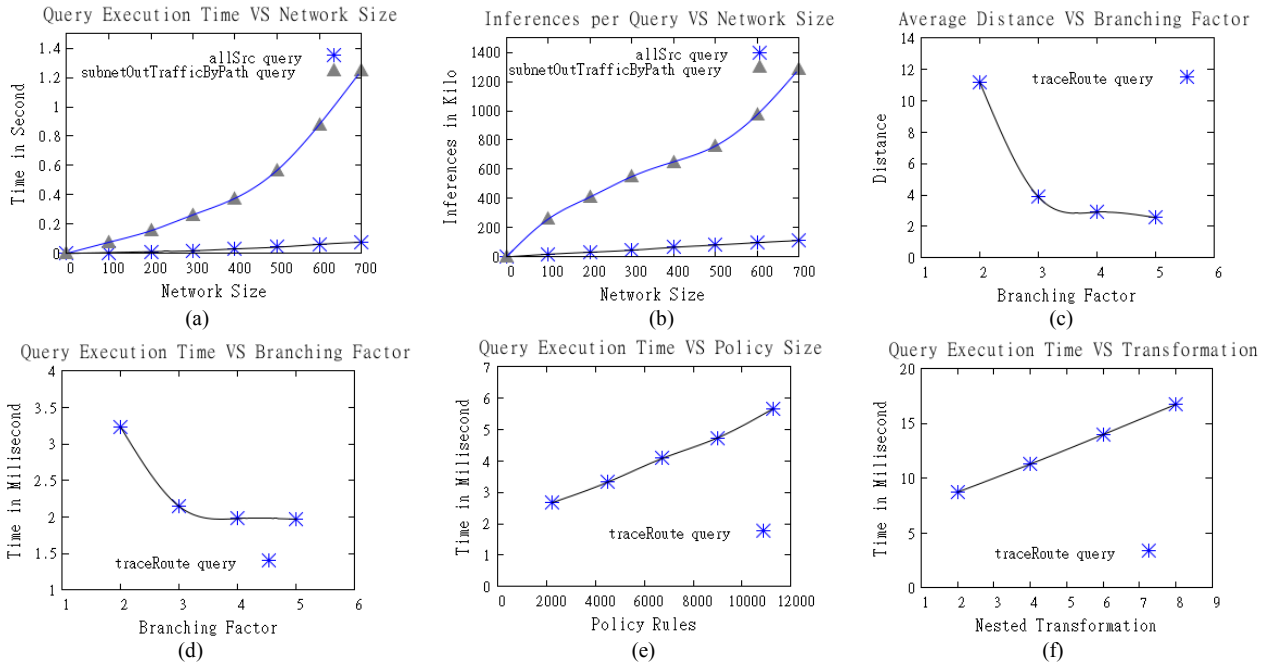

Figure 5: Impact of network size (a) on execution time and (b) on the number of inferences; impact of branching factor (c) average distance and (d) query execution time; query execution time varies with (e) policy size and (f) number of nested transformation

privilege level (`lowerPrivilege` in Listing 6). Similarly, we can assume the risk of a subnet by observing fan-in of the subnet. We can simply say that the more is the fan-in, the more is the risk. Hence, we can write query to compare risk between two subnets that one has more risk than other based on fan-in.

IV. EVALUATION

Our goal of the evaluation is to show the scalability of our declarative approach. We take execution time as the evaluation metric. We plot graphs (Fig. 5) against number of hosts, node branching factor (of core network), policy size and number of nested transformation.

We run our queries using SWI-Prolog (version 5.10.0) in Intel Core2Duo 3.33 GHz Processor with 4 GB memory under Windows 7 OS. We create the declarative network topology and security policies writing a program. We randomly create the core network with routers. Subnets are connected to the routers. Topology of each subnet is straight-forward. Each subnet consists of a number of hosts, a router, a firewall and an IPSec device. The routing entries of each router are generated following the shortest path from the router to the destination. Firewall and IPSec policies are written randomly. We run each query for minimally 3 times and take the average value.

We show the impact of network size on the execution of queries in Fig. 5(a). We consider network size as the number of hosts. Here, we only change the number of hosts but the core network remains the same. Core network consists of 20 routers. Here, we take two queries for evaluation: `allSrc` and `subnetInTrafficByPath`. These two queries find all the traffic reachable to a particular host and a subnet respectively and, hence, are suitable to check scalability in this case. We observe from the graph of Fig. 5(a) that the increase of execution time is almost linear with the increase of network size. The growth rate of the time in case of `subnetInTrafficByPath` is much higher than `allSrc`, since it searches for all the incoming traffic for a group of hosts (within the subnet) passing through a particular path. Fig. 5(b) shows the similar demonstration that the number of logical inferences grows linearly with the growth of network size.

In Fig. 5(d), we plot the growth of query execution time against the node branching factor. We take `traceRoute` as the query for evaluation and this is sufficient as it is the basis of all other queries. We keep the number of routers in the core network fixed (50 routers). The average distance between the routers changes with branching factor (Fig. 5(c)). Fig. 5(d) shows that execution time decreases with the increase of branching factor, as average distance is decreased.

We show the impact of the policy size in query execution time in Fig. 5(e). The policy size of the network is taken as the total number of policy rules written in firewalls. The number of routers is taken as 20. When the number of policy rules increases, query execution time increases proportionally. In case of nested transformation, the more is the number of nested levels, the more is the number of transformation and detransformation and so the query execution time. Fig. 5(f) justifies this. We use up to 8 consecutive IPSec devices at each subnet to create nested transformation.

V. RELATED WORKS

A good number of works has been done in the area of firewall and IPSec policy verification. Significant success has been done also in the use of declarative languages in networking. In our proposed model and tool, we do combine these two ideas. In this section, we will describe very briefly some works that are related and, also, motivator to our work.

Policy anomalies among firewalls are well discussed in [12]. Authors identify the intra-firewall and inter-firewall anomalies and present a set of techniques and algorithms for

automatic discovery of policy anomalies in centralized and distributed firewalls. A Boolean model for IPSec policy filtering rules is presented in [3] using ordered binary decision diagram (OBDD). Authors in [1] discuss about different intra-domain and inter-domain conflicts among IPSec policies. This research work is extended for an automatic IPSec policy generation framework in [2]. A network configuration tool ConfigChecker is proposed in [4] by Ehab et al. Authors here use computation tree logic (CTL) and symbolic model checking to verify traffic reachability and security requirements by investigating states of a packet in the network.

Application of declarative language in networking becomes familiar when declarative routing is proposed in [6]. Declarative routing is a flexible and extensible way to express routing protocols using database query language, Datalog. It is shown that simple recursive queries can express a variety of well-known routing protocols. NDlog (Network Datalog) is formally defined in [7] for declarative networking. Next, SeNDlog is proposed in [9, 11], which unifies NDlog with Binder (a language for distributed access control). SeNDlog provides authenticated communication among untrusted nodes. Again, the design of declarative network verification is proposed in [8, 13], where NDlog is unified with theorem prover (PVS) to verify protocol correctness.

A database-oriented declarative framework DECOR for network management and operation framework is proposed in [10]. Here the authors model router configuration along with generic network status as relational data in a database and presents rule-based language, which can easily specify and enforce management constraints. With the goal of having easy specification and automated security enforcement, a domain specific language PPL is proposed in [5]. They discuss the difficulties of writing correct security policy and propose the design of the language.

Sanjay Narain and et al. presented a declarative service grammar for creating autonomic systems in a domain of interest [15]. They considered an example virtual private network and wrote declarative rules to configure and integrate all the configuration constraints necessary for that network. They showed that, based on this declarative configuration grammar, correctness of any specification of constraints on that framework can be evaluated.

VI. CONCLUSION

In this research work, we have presented a declarative approach for network configuration verification. Researchers have already used declarative approach for modeling some network configuration aspects like routing protocols and firewalls. However, our approach is novel in applying declarative logic to comprehensively model network access control policies through variety of security devices, like, routers, firewalls and IPSec. Unlike previous work, this paper gives high attention to (1) modeling IPSec access and map polices considering transport and tunnel modes as well as AH.and ESP operations, and (2) integrating access control polices in a single framework to verify end-to-end reachability and security transformation requirements.

Our model can easily be extended for more significant uses such as risk and network vulnerability analysis. We have shown examples to illustrate the usefulness of declarative model in network security configuration verification. This framework was implemented in a tool called *ConfigAnalyzer*. Our evaluation study shows that even with growth of network size to 800 nodes and policy size to 12K rules, the analysis time remains reasonably low within few seconds. In future, this work can be extended by modeling defense-in-depth and risk analysis based on trusted zone.

REFERENCES

[1] Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, "IPSec/VPN Security Policy: Correctness, Conflict Detection and Resolution", The 2nd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), Vol 1995, 2001.

[2] Yanyan Yang , Charles U. Martel , Zhi (Judy) Fu , S. Felix Wu, "IPSec/VPN Security Policy Correctness and Assurance", Journal of High Speed Networking, Special issue on Managing Security Polices: Modeling, Verification and Configuration, 2006.

[3] Hazem Hamed , Ehab Al-Shaer and Will Marrero, "Modeling and Verification of IPSec and VPN Security Policies", IEEE International Conference in Network Protocols, 2005.

[4] Ehab Al-shaer , Will Marrero , Adel El-atawy and Khalid Elbadawi, "Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security", IEEE International Conference in Network Protocols (ICNP), 2009.

[5] Hedi Hamdi, Adel Bouhoula and Mohamed Mosbah, "A Declarative Approach for Easy Specification and Automated Enforcement of Security Policy", International Journal of Computer Science and Network Security, Vol. 8, No. 2, 2008.

[6] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan, "Declarative Routing: Extensible Routing with Declarative Queries", ACM SIGCOMM Conference on Data Communication, Philadelphia, PA, Aug 2005.

[7] Boon Thau Loo and et al., "Declarative Networking: Language, Execution and Optimization", *SIGMOD*, 2006.

[8] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau, Loo Oleg and Sokolsky Prithwish Basu, "Formally Verifiable Networking", The 8th Workshop on Hot Topics in Networks, New York, Oct 2009.

[9] Wenchao Zhou, Yun Mao, Boon Thau Loo and Martín Abadi, "Unified Declarative Platform for Secure Networked Information Systems", The 25th International Conference on Data Engineering (ICDE), China, Apr 2009.

[10] by Xu Chen and Z. Morley Mao, "DECOR: Declarative Network Management and Operation", *PRESTO*, 2009.

[11] Martín Abadi and Boon Thau Loo, "Towards a declarative language and system for secure networking", The 3rd International Workshop on Networking meets Databases, 2009.

[12] E. Al-Shaer and H. Hamed, "Discovery of policy anomalies in distributed firewalls", IEEE INFOCOM'04, Mar 2004.

[13] Anduo Wang, Prithwish Basu, Boon Thau Loo and Oleg Sokolesky, "Declarative Network Verification", The 11th International Symposium on Practical Aspects of Declarative Language (PADL), Jan 2009.

[14] SWI-Prolog: http://www.swi-prolog.org/

[15] Sanjai Narain, Thanh Cheng, Brian Coan, Vikram Kau, Kirthika Parmeswaran, William Stephens, "Building Autonomic Systems Via Configuration," The 5th Annual International Workshop on Active Middleware Services, 2003.