Towards A Unified Modeling and Verification of Network and System Security Configurations

Mohammed Noraden Alsaleh, Ehab Al-Shaer
University of North Carolina at Charlotte
Charlotte, NC, USA
Email: {malsaleh, ealshaer}@uncc.edu

Adel El-Atawy
Google Inc
Mountain View, CA, USA
Email: aelatawy@google.com

***Abstract*—Systems and networks access control configuration are usually analyzed independently although they are logically combined to define the end-to-end security property. While systems and applications security policies define access control based on user identity or group, request type and the requested resource, network security policies uses flow information such as host and service addresses for source and destination to define access control. Therefore, both network and systems access control have to be configured consistently in order to enforce end-to-end security policies. Many previous research attempt to verify either side separately, but it does not provide a unified approach to automatically validate the logical consistency between both of them. Thus, using existing techniques requires error-prone manual and ad-hoc analysis to validate this link.**

In this paper, we introduce a cross-layer modeling and verification system that can analyze the configurations and policies across both application and network components as a single unit. It combines policies from different devices as firewalls, NAT, routers and IPSec gateways as well as basic RBAC-based policies of higher service layers. This allows analyzing, for example, firewall polices in the context of application access control and vice versa. Thus, by incorporating policies across the network and over multiple layers, we provide a true end-to-end configuration verification tool. Our model represents the system as a state machine where packet header, service request and location determine the state and transitions that conform with the configuration, device operations, and packet values are established. We encode the model as Boolean functions using binary decision diagrams (BDDs). We used an extended version of computational tree logic (CTL) to provide more useful operators and then use it with symbolic model checking to prove or find counter examples to needed properties. The tool is implemented and we gave special consideration to efficiency and scalability.

I. Introduction

Users inadvertently trigger a long sequence of operations in many locations and devices by just a simple request. The application requests are encapsulated inside network packets which in turn are routed through the network devices and subjected to different types of routing, access control and transformation policies. Misconfigurations at the different layers in any of the network devices can affect the end to end connection between the hosts them selves and the communicating services running on top of them. Moreover, applications may require to transform requests to another one or more requests with different characteristics. This means that the network layer should guarantee more than one packet flow at the same time in order for the application request to be successful. Although it is already very hard to verify that only the legitimate packets can pass through the network successfully, the consistency between network and application layer access configuration adds another challenge. The different natures of policies from network layer devices to the logic of application access control makes it more complex.

In this paper we have extended ConfigChecker [3] to include application layer access control. The ConfigChecker is a model checker for network configuration. It implemented many network devices including: routers, firewalls, IPSec gateways, hosts, and NAT/PAT nodes. ConfigChecker models the transitions based on the packet forwarding at the network layer where the packet header fields along with the location represent the variables for the model checker. We define application layer requests following a loose RBAC model: a 4-tuple of <user, role, object, action>. The request can be created by users or services running on top of hosts in the network. The services in our model can also transform a request into another one or more requests and forward them to a different destination. We have implemented a parallel model checker for application layer configuration. The transitions in the application layer model is determined by the movement of the requests between different services. However, the network and application layer model checkers are not operating separately. Requirements regarding both models can be verified in a single query using our unified query interface. Moreover, inconsistency between network configuration and application layer configuration can be detected.

The nature of the problem of verifying network-wide configurations necessitate having a very scalable system in terms of time and space requirements. Larger networks, more complex configurations, and richer variety of devices are all dimensions over which the system should handle gracefully. The application-layer access control depends on different variables than most of network layer policies. We chose to implement a parallel model checker for application access control rather than adding the application layer variables (which correspond to request fields) to the network model checker itself. This can decrease the number of system states and improve the performance. As the case in ConfigChecker, both model checkers are represented as state machines and encoded as boolean function using Binary Decision Diagrams (BDDs). We use an extended version of computational tree logic (CTL) to

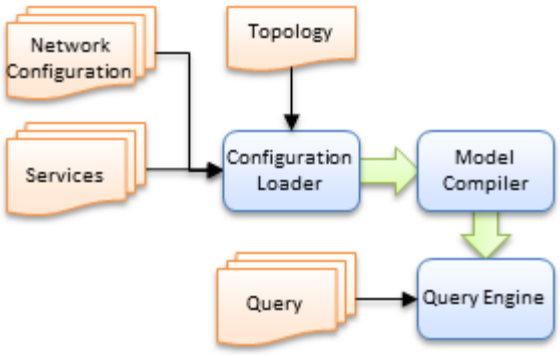

Fig. 1. A simple overview of the framework design and flow

provide more useful operators and then use it with symbolic model checking to prove or find counter examples for needed properties regarding both models.

The rest of this paper is organized as follows. We first briefly describe our framework components in Section II. We then present the model used for capturing the network and application layer configuration in Section III and Section IV respectively. Section V shows how to query the model for properties, and lists some sample queries. The related work is presented in Section VI. We finally present our conclusion and future remarks in Section VII.

II. Framework overview

The framework consists of a few key components: configuration loader, model compiler, and query engine. The duty of each component is described briefly below:

- The *Configuration Loader* parses the main configuration file that points out to the configuration files of network devices. Each file represents a device or entity (*e.g.*, firewall, router, application-layer service, etc). Each configuration file consists mainly of two sections: meta-data directives, and policy. The initial directives act as an initialization step to configure the device properties like default gateway, service port, host address, etc. The policy is listed afterwards as a simple list of rules.
- The *Model Compiler* translates the configuration into a Boolean expressions for each policy rule and builds a single expression for each device. These expressions are then combined into a single expression representing the whole network.
- The *Query Engine* is responsible for verifying properties of the system by executing simple scripts based on CTL expressions. Scripts is written using a very limited set of primitives for refining the user output, and for defining the property itself.

The model compiler component builds two separate expressions. The first represents the network layer configuration that reflects the packets forwarding and transformation through the network core and end points as described in Section III. The other expression represents the application layer configuration including services and users. This reflects how requests are forwarded and transformed in the service level. Section IV describes this process in more details. Although we can integrate the two expressions and build only one expression that accommodates for both the network and application layer configuration, we chose to split them into two expression. The variables used on each of them are generally independent except for the location variable. Building one expression takes more space because the network configuration will be duplicated for each different combination of application layer variables generating more and more states. This helps our model to scale and avoid state explosion.

III. Network Model

We model the network as a single monolithic finite state machine. The state space is the cross-product of the packet properties by its possible locations in the network. The packet properties include the header information that determines the network response to a specific packet.

A. State representation

Initially, the only information we need about the packet is the source and destination information contained in the IP header and the current location of the packet in the network. Therefore, we can encode the state of the network with the following characteristic function:

$$\sigma_n : \mathbf{IP_s} \times \mathbf{port_s} \times \mathbf{IP_d} \times \mathbf{port_d} \times \mathbf{loc} \rightarrow \{\textbf{true, false}\}$$

$\mathbf{IP_s}$	the 32-bit source IP address
$\mathbf{port_s}$	the 16-bit source port number
$\mathbf{IP_d}$	the 32-bit destination IP address
$\mathbf{port_d}$	the 16-bit destination port number
$\mathbf{loc}$	the 32-bit IP address of the device currently processing the packet

The function σ_n encodes the state of the network by evaluating to true whenever the parameters used as input to the function correspond to a packet that is in the network and false otherwise. If the network contains 5 different packets, then exactly five assignments to the parameters of the function σ_n will result in true. Note that because we abstract payload information, we cannot distinguish between 2 packets that are at the same device if they also have the same IP header information.

Each device in the network can then be modeled by describing how it changes a packet that is currently located at the device. For example, a firewall might remove the packet from the network or it might allow it to move on to the device on the other side of the firewall. A router might change the location of the packet but leave all the header information unchanged. A device performing network address translation might change the location of the packet as well as some of the IP header information. A hub might copy the same packet to multiple new locations. The behavior of each of these devices can be described by a list of rules. Each rule has a condition and an action. The rule condition can be described using a Boolean formula over the bits of the state (the parameters of the characteristic function σ_n). If the packet at the device

matches (satisfies) a rule condition, then the appropriate action is taken. As described above, the action could involve changing the packet location as well as changing IP header information. In all cases, however, the change can be described by a Boolean formula over the bits of the state. Sometimes the new values are constant (completely determined by the rule itself), and sometimes they may depend on the values of some of the bits in the current state. In either case, a transition relation can be constructed as a relation or characteristic function over two copies of the state bits. An assignment to the bits/variables in the transition relation yields true if the packet described by the first copy of the bits will be transformed into a packet described by the second copy of the bits when it is at the device in question.

B. Network devices

We integrate the policies of different network devices including firewalls, routers, NAT and IPSec gateways. The details of their policies and how they are encoded into BDDs are discussed thoroughly in [3]. However, we have modified the encoding of hosts to reflect the request transformation performed by the services running on top of each host.

The host may be configured to run one or multiple services. Each of which has its own access-control list as will be discussed in Section IV. The service configuration may also specify a set of possible request transformations where the incoming request is transformed into another (sometimes completely new) request. For example, a request to a web server can be translated into an NFS request to load a users home page. The new request will be carried through the network over packets. In our initial model the host receives packet and then forward them into the application layer within the host itself and it cannot forward it to another host. We need to modify this model so that the host will be able to forward packets to the other hosts in order to support the requests transformation performed by the services running on top of it.

IV. Application Layer Model

We also model the application layer as a finite state machine. The state space is the cross-product of the application layer request properties by its possible locations in the network. The request properties include the its fields that determine the service response to a specific request.

$$\sigma_p : \mathbf{usr} \times \mathbf{role} \times \mathbf{obj} \times \mathbf{act} \times \mathbf{loc} \times \mathbf{srv} \rightarrow \{\textbf{true, false}\}$$

usr	the 32-bit user ID
role	the 32-bit role ID which the user belongs to
obj	the 32-bit object ID
act	the 16-bit action ID
loc	the 32-bit IP address of the device currently processing the request
srv	the 16-bit service ID

In the application layer model the devices in the network are modeled by describing how they change the requests. Only the devices that operates on the application level are considered (*i.e.*, the devices who has a defined users list or services running on top of them). Here, we describe how we define access-control rights for service requests, and how we model these services and integrate them into the application layer state transition diagram.

A. Application layer access-control

In order to have a homogeneous policy definition across applications, we revert to a simplified RBAC model as a way to specify all application requests and consequently the access-control policy. As in firewall policy, the access-control list of application layer services is defined by specifying an action like (permit or deny) to the requests satisfying certain criteria. This criteria is defined using the request fields $< user, role, object, action >$ or $< u, r, o, a >$ for short. We assume that each host has a list of potential users who can use it to send requests. This list can simply be set to "any", to indicate that all defined users can access the host which enables a more powerful model for an adversary against which we want to verify the robustness of the policy. Also, another assumption is that any user can assume any role. This enables a more flexible usage of the model to incorporate more types of services. It is even possible to use one of the request fields in a slightly different meaning. For example, in a web server model; an action can be a POST or GET, the role can be a logged in versus guest visitor and the object and user will have their obvious meanings. On the other hand, for database servers, we might have users, roles, actions, and resources used in their original meaning.

When a service receives the a request from other device it first verifies it against its access-control policy. If it satisfies the access-control policy it will be forwarded to the service to be executed or transformed as shown in Fig. 2. If the request does not satisfy the access policy it will be dropped (*i.e.*, there is no valid transition in the finite state machine that goes to the required service). A policy typically is defined as a list of tuples with an assigned action:

```
user    role    resource    action      decision
;user black listing
1       *       *           *           deny
2       *       *           *           deny
;admin account
100     1       *           *           permit
;guests can only read
*       guest   1-50        3           permit
;a read only resource
*       *       60          4           permit
*       *       60          *           deny
```

As in firewall policies, we use a first-match methodology. For example, the last two rules allow read access, and then deny every other action. Also, the first few black-listing rules do not conflict with the guest account rule that appears later in the policy. From the common practices in the area of application level and RBAC policies, we believe that it should never be the case that a user or role be specified as a range. The value of $user$ and $role$ IDs are irrelevant, therefore having a range in the specification is hard to have a practical value. Although, this fact does not affect our implementation (*i.e.*, we support single values, ranges, or "any" values in all four fields of the access-control rules).

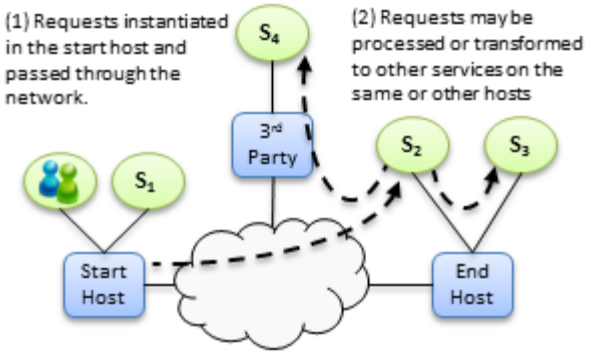

Fig. 2. Service Model. S_1, S_2, S_3, and S_4 represent services running on different hosts. The dashed lines represent application requests. Requests are subjected to the access control list of the target service.

B. State representation

Requests that pass the access-control phase are forwarded to the execution phase of the service. We have simplified the execution phase to one of two options as suggested in Fig. 2. A request can be executed on the service itself, which means that the request life-time ends at this phase, and no further events are triggered. The other possibility is that the service transforms the request into another form by modifying one or more fields (*i.e.*, user, role, object or action) and sends it to another service running on the same or a different host. For example, a request to a web server can be translated into an NFS request to load a user's home page. Each request transformation is associated with a packet flow in the network level, the host should be able to send the appropriate packet to its gateway based on the service transformation.

Only the network devices that support application layer services represented by hosts in our model are included in the application layer model. The requests that leave a host may come from two sources: either a user operating directly on the host or through a service, or a request is transformed from another one. We do not require the destination of each request to be defined in the configuration. We assume that any request instantiated in any host can be directed to any service in the network whose access control list allows the request to pass. To build the transition relation for each host in the network we need the following inputs.

- The set U of users who can access the host. Each user is represented by its unique ID in the system. It can also be expressed as a <user, role> pair.
- The set T of possible request transformations that can be performed in the host. The configuration may specify the exact ID and location of the target service, or it can be anonymous.
- The set P of access control policies for each service running in the network. These policies are encoded as BDD expressions before building the transition relation of the application layer. Each policy in the set corresponds to a particular service ID (The port number of the service can be used as its ID) running on a particular host.

Lets assume that the list U_H of <user, role> pairs represents the users who can access the host H. We need first to encode the possible states that result from having these users on the host H (recall that the state is the product of the request properties <user, role> in this case by the location in which the request exists).

$$U_{BDD} = \bigvee_{i \in U_H} (\mathbf{usr} = u_i \wedge \mathbf{role} = r_i \wedge \mathbf{loc} = H) \quad (1)$$

where u_i and r_i are the user and role IDs of the item i in the users list U_H. To find the transitions we need to find out which services can be reached starting from the states defined in the expression U_{BDD} (*i.e.*, any service whose access-control policy allows defined requests to pass).

$$T_u = \bigvee_{i \in indices(P)} (U_{BDD} \wedge P(i) \wedge \mathbf{loc}' = l_i \wedge \mathbf{srv}' = s_i) \quad (2)$$

where $P(i)$, l_i, and s_i are the policy, location, and the service ID of the target service i respectively. The variables $\mathbf{loc}'$ and $\mathbf{srv}'$ represents the location and service ID variables in the next state of the transition.

The expression T_u does not include the transitions that results from request transformation by the service running on the host. The following represents the transition that result from one transformation performed by the service i.

$$\begin{gathered} \mathbf{usr} = u_i \wedge \mathbf{role} = r_i \wedge \mathbf{obj} = o_i \wedge \mathbf{act} = a_i \\ \mathbf{loc} = H \wedge \mathbf{srv} = s_i \\ \mathbf{usr}' = u'_i \wedge \mathbf{role}' = r'_i \wedge \mathbf{obj}' = o'_i \wedge \mathbf{act}' = a'_i \wedge P'(s'_i) \\ \mathbf{loc}' = l'_i \wedge \mathbf{srv} = s'_i \end{gathered} \quad (3)$$

The values $< u_i, r_i, o_i, a_i >$ are the properties of the initial request and the values $< u'_i, r'_i, o'_i, a'_i >$ represent the properties of the transformed packet. The values $P'(s'_i)$, l'_i and s'_i are the policy, location and the service ID of the target service to which the request should be transformed. Note that we use $P'(s'_i)$ instead of $P(s'_i)$ to indicate that the transformed request (and not the initial request) should pass the target service access-control policy in order to complete the transition. The disjunction of all the transitions caused by all the possible transformations along with the expression T_u calculated earlier formulate the total transition relation of the host H.

V. Querying the Model

A query in our system takes the form of a Boolean expression that specifies some properties over packet flows, requests and locations, with temporal logic criteria specified using CTL operators. By evaluating the given expression in the context of the built state machine (*i.e.*, states and transitions), we obtain the satisfying assignments to that expression represented in the same symbolic representation as the model itself. The simplest form for the result can be the constant expression "true" (*e.g.*, the property is always satisfied), or "false" (*e.g.*, no one violates the required property), or can be any subset of the space that satisfies the property (*e.g.*, only flows with port 80, or traffic that starts from this location, etc).

In the following subsections, we will go over a few examples for reachability and security properties. For each, we show how to construct the query (*i.e.*, the Boolean expression), and what the results look like. Moreover, we discuss how to write the script that extracts the results and data fields in the intended format that makes sense to every specific query. The aim of this presentation is to show the applicability of the system to many types of properties, as well as showing the expressive power of the model.

A. Model Checking

We have described how to construct a transition relation for each device in the network. Each such transition relation describes a list of outgoing transitions for the device it models. The formulas are constructed with the requirement that the current location be equal to the device being modeled, so these transitions can only be taken when a packet or a request is at the device. To get the transition relation for the entire network, we simply take the disjunction of the formulas for the individual devices. This is applied on both models (The network and the application layer models). The current location of a packet or a request will match the location of one device in the network at most, and so only its transitions will apply.

Recall that this global transition relation is a characteristic function for transitions in the model. If we substitute the values for a packet that is in the system into the current state variables of the transition relation, what we are left with is a formula describing what the possible next states of that packet look like. We have all the machinery to perform symbolic model checking. We use BDDs for all the formulas described above and we use standard model checking algorithms to explore the state space and compute states that satisfy various CTL properties. The BuDDy BDD package provides all the required operations (including quantification). For a much more complete description of symbolic model checking, the reader is encouraged to see [7].

The network layer model checker and the application layer model checker are encoded separately. Each of them is working on different variables. However, we may need to verify some requirements using both of them together. Although the application layer requests are transmitted from an application to another, the requests are encapsulated inside network packets. We do not require to have static one-one mapping between each request and the packet flow that should be used to transfer the request. The mapping can be expressed in the query itself by specifying precise network packet characteristics <protocol, source ip, destination ip, source port, destination port>. For this purpose we use the variable **loc** which is common between the two models and has the same meaning. Fig. 3 shows how the two models are used together to verify the requirements.

B. Query structure and features

The query in our model checkers retrieves the states that satisfy a given condition. The condition is expressed by restricting some variables in the model checker to a given value or using CTL operators to express a temporal condition. We also need to specify what information to be retrieved about the states which satisfy the query (*i.e.*, a list of variables to be retrieved). An example query can look like this:

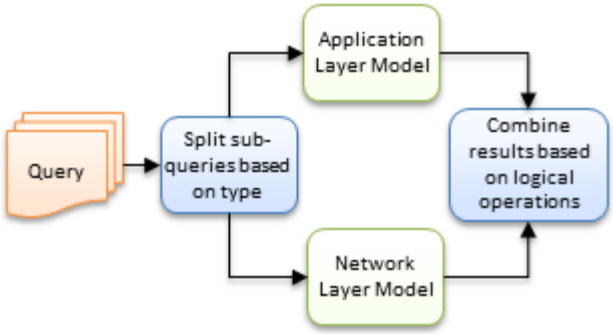

Fig. 3. Using two models to run a query.

$$Q3 = [loc(10.12.13.14) \wedge EF((\neg loc(10.0.0.0/8)) \wedge (\neg Q2))]$$
$$Q3 : extractField\ loc\ dport$$
$$Q3 : listBounded\ 20\ loc\ dport$$

The query $Q3$ is defined by the given expression (*i.e.*, what flows are in a given location $(10.12.13.14)$ and in the future will be outside of the domain $(10.0.0.0/8)$ and do not satisfy a previously defined query $(Q2)$. The second and third lines are used to format the result of the query. The second line tells the query engine that we are only interested in the variables loc and $dport$. The third line specifies that we need to display only the first 20 satisfying assignments. If there is no satisfying assignment for the given query nothing will be returned.

To handle the queries on both the network layer and application layer models, we introduced the concept of sub-query. Each sub-query is applied on one model. The application-layer sub-query should not include variables related to network layer model such as source or destination addresses and port numbers and vice versa. A query can include one or more sub-queries based on the following cases.

- It can include only one sub-query. In this case the query is applied only on the appropriate model. The query engine detects the appropriate model based on the variables used.
- It can include more than one sub-query of the same type linked by the logical operators such as AND, OR, $IMPLIES$, etc. In this case all the sub-queries are executed on the appropriate model and the final result is calculated by applying the specified operation. The results of the different sub-queries in this case are identical in terms of the number and type of the variables returned. The linking operation can be directly applied.
- It can include multiple sub-queries of different types linked by logical operators. In this case, the results of the different types of sub-queries have different variables and we cannot apply the linking operation directly. The

location variable (**loc**) is common between the two models and it has the same meaning and value for the same device. For different types of sub-queries we apply the logical operations based on the location variable only. For example, if an application-layer sub-query is combined with a network-layer sub-query by the AND operation, we calculate the result of both sub-queries and then calculate the intersection between the location values in both results. Only requests and packet flows whose location falls within the intersection are returned.

The result of a query is a list of states that satisfy the query expression. In our model we may have two types of results. The first is a set of network-layer states each represented as a packet flow characteristics and location. The second is a set of application-layer states each represented as a request characteristics and location. The existence of these two types depends on the types of sub-queries included within the query script.

C. Example properties

Property 1: Conflicting network and application access-control

(a): Given a user location and userID, does the current configuration allows the user to access the server machine, while the application layer access-control blocks the connection?

The query shown in the Table I specifies the initial properties of flows with certain user information (*e.g.*, a specific source IP "$user\_addr$", and user identifier in the application layer request) and targeted towards a service residing elsewhere (*i.e.*, $server$, $port'$). If there is an inconsistency in the configuration the query returns a list of requests that cannot access the specified service and another list of packet flows that can eventually reach the host network layer. We can see that the query combines two different types of sub-queries and restricts the location to a particular source machine. Each sub-query is surrounded by angle brackets "[ ]".

(b): Does the current configuration blocks the user's access to the server machine through network layer filtering, while the application's access-control layer permits such connection?

As in the pervious property (a), we try to see if a request that is permitted by the application layer access control will never reach the service. This means that somewhere before reaching the server hosting the service there is a network layer device blocks the traffic, or fails to route it correctly. We use the application layer model to find those requests that can pass from the source to a particular service, and we use the network model to find if the underlaying packet flows who should carry the requests are allowed to flow from the source to the appropriate destination.

Property 2: Can a user access a resource under different credentials, if he is prohibited from accessing it under his original identity?

In this property, we check if a certain user can masquerade under another identity to access a resource. This forms a back-door to this specific object-action pair. A straight forward example can be a user accessing an NFS server for which he does not have access via a web-server who can retrieve the content in the form of web pages. This can be achieved by an improper request transformation in a service which should not be reachable by the specified user. This is defined formally by evaluating the expression specifying which users cannot access an object, while it can be accessed eventually if the constraint on the user identity is removed.

Property 3: What access rights does an object require?

(a): What roles can user u use to access object o?

It is sometimes essential to know what roles can a user manifest when accessing a specific object, or a group of objects. The query consists of checking the space of requests that can pass through the network and RBAC filtering to reach our object of interest. By restricting the user part of the space, we get the possible roles that can be used. We can also, restrict the action if needed. An addition that can prove practically useful is to add another restriction of origin of request: by filtering the location and source address from which the request originated from other than that of the server. In Table I we show only the condition part of the query neglecting the result format part, we can specify to return only the roles and/or any other fields in the results.

(b): Which users can access object o via a role r?

This is a similar query to the previous one. This query concerns the different users who can access a given object in a certain capacity. For example, we might want to know who can access a critical file as an administrator. Also, we can add extra restrictions to see who can access this object for writing rather than just reading.

Property 4: Is there any conflicts within the application layer access-control?

(a): Is there any inconsistencies in allowed actions for a specific object?

Such conflicts can arise within the same policy or cross policies. For example, if a user is granted the write access to an object then, most probably, read access should be allowed as well. This query is application dependent, and priority between actions has to be specified explicitly (*e.g.*, 'delete' $>$ 'write' $>$ 'read'). We write the query for a service to check if it is possible for some user/role to reach the service via a higher action, but not with a lower one. We represent the general form for such query in Table I, the profiles $[high\ security\ requirements]$ and $[low\ security\ requirements]$ can be replaced with any combination of constraints on the request fields. For example, to compare rights for reading and writing, the high security profile may be $[obj(o) \wedge act(wr)]$ and the low security profile may be represented as $[obj(o) \wedge act(rd)]$ for the particular object (o).

(b): Are role-role relations consistent?

As in the pervious property (a), we might need to verify that the order of role privileges is maintained. In other words, a

Property	Query expression form
P1 (a):	*Requests reaching the host but not the service it is running.* $loc(user\_addr) \wedge [src(user\_addr) \wedge dest(server) \wedge dport(port') \wedge EF(loc(server))] \wedge [usr(userID) \wedge \neg EF(loc(server) \wedge srv(port'))]$
P1 (b):	*Requests reaching the service but can't reach the host itself.* $loc(user\_addr) \wedge [usr(userID) \wedge EF(loc(server) \wedge srv(port'))] \wedge$ $[src(user\_addr) \wedge dest(server) \wedge dport(port') \wedge \neg EF(loc(server))]$
P2:	*Backdoor: A user is denied direct access to a service, but can use another service to indirectly access it.* $loc(user\_addr) \wedge [usr(userID) \wedge \neg EF(usr(userID) \wedge obj(o) \wedge loc(server) \wedge srv(port')) \wedge EF(usr(\neg userID) \wedge obj(o) \wedge loc(server) \wedge srv(port')] \wedge [EF(loc(server) \wedge dport(port'))]$
P3 (a):	*What roles and actions can a user use to access a specific object from outside the server domain?* $\neg loc(server) \wedge [EF(loc(server) \wedge srv(port') \wedge usr(u) \wedge obj(o))] \wedge$ $[\neg src(server) \wedge EF(loc(server) \wedge dport(port'))]$
P3 (b):	*What users can access a given object?* $\neg loc(server) \wedge [\neg src(server) \wedge EF(loc(server) \wedge dport(port'))] \wedge$ $[EF(loc(server) \wedge srv(port') \wedge role(r) \wedge obj(o) \wedge act(a))]$
P4:	*Is there any inconsistency between rights of low and high privilege requests?* $EF(loc(server) \wedge srv(port') \wedge [high\ security\ requirements])$ $\wedge \neg EF(loc(server) \wedge srv(port') \wedge [low\ security\ requirements])$

TABLE I
EXAMPLES FOR REACHABILITY AND SECURITY PROPERTIES

more powerful role should be always capable of performing all actions possible for a weaker role. For example, an administrator should perform at least everything doable by a staff member, and guests should never have more access to other roles. It is defined by checking if the space of possible actions-over-objects that can be performed by $role1$ but not $role2$ is empty (given that $role1 < role2$). In this case the high and low security profiles can be represented as $[role(role1) \wedge obj(o) \wedge act(a)]$ and $[role(role2) \wedge obj(o) \wedge act(a)]$ respectively.

VI. RELATED WORK

There have been significant research effort in the area of configuration verification and management in the past few years. We can classify the work in this area into two main approaches: top-down and bottom-up. The top-down approaches [20], [5] create clean-slate configurations based on high-level requirements. However, the bottom-up approaches [13], [1], [24] analyze the existing configuration to verify desired properties. We focus our discussion on bottom-up approach as it is closer to our work in this paper.

There has been considerable work recently in detecting misconfiguration in routing and firewall. Many of these approaches are specific for BGP misconfiguration [9], [17], [11], [4]. [23], [13], [1], [24] focused on conflict analysis of firewalls configuration. A BDD-based modeling and taxonomy of IPSec configuration conflicts was presented in [2], [13]. FIREMAN [24] uses BDD to show conflicts on Linux `iptables` configurations. In [19] and [21], the authors developed a firewall analysis tool to perform customized queries on a set of filtering rules of a firewall. But no general model of network connections is used in this work.

In the field of distributed firewalls, current research mainly focuses on the management of distributed firewall policies. The first generation of global policy management technology is presented in [12], which proposes a global policy definition language along with algorithms for verifying the policy and generating filtering rules. In [6], the authors adopted a better approach by using a modular architecture that separates the security policy and the underlying network topology to allow for flexible modification of the network topology without the need to update the security policy. Similar work has been done in [14] with a procedural policy definition language, and in [16] with an object-oriented policy definition language. In terms of distributed firewall policy enforcement, a novel architecture is proposed in [15] where the authors suggest using a trust management system to enforce a centralized security policy at individual network endpoints based on access rights granted to users or hosts. We found that none of the published work in this area addressed the problem of discovering conflicts in distributed firewall environments.

A variety of approaches have been proposed in the area of policy conflict analysis. The most significant attempt for IPSec policy analysis is proposed in [10]. The technique simulates IPSec processing by tracking the protection applied on the traffic in every IPSec device. At any point in the simulation, if packet protection violates the security policy requirements, a policy conflict is reported. Although this approach can discover IPSec policy violations in a certain simulation scenario, there is no guarantee that it discovers every possible violation that may exist. In addition, the proposed technique only discovers IPSec conflicts resulting from incorrect tunnel overlapping, but do not address the other types of conflicts

that we study in this research.

Other works attempt to create general models for analyzing network configuration [8], [22]. An approach for formulating and deriving of sufficient conditions of connectivity constraints is presented in [8]. The static analysis approach [22]is one of the most interesting work that is close to ConfigChecker. This work uses graph-based approach to model connectivity of network configuration and use set operations to perform static analysis. The transitive closure, as apposed to a fixed point in our approach, is computed. Thus, it seems that all possible paths are computed explicitly. In addition, considering security devices and properties, providing a rich query interface based on our CTL extension, and utilizing BDDs optimization are major advantages of our work. Anteater [18] is another interesting tool for checking invariants in the data plane. It checks the high-level network invariants represented as instances of boolean satisfiability problems (SAT) against network state using a SAT solver, and reports counterexamples for violations, if exist.

Thus, in conclusion, although this body of work has a significant impact on the filed, it is either provide limited analysis due to restriction on specific network or application. Unlike the previous work, our work offers a global configuration verification that is comprehensive, scalable and highly expressive.

VII. Conclusion

We presented an extension to the ConfigChecker tool to incorporate both network and application configurations in a unified system across the entire network. Our extended system models the configuration of various devices in the network layer (hubs, switches, routers, firewalls, IPsec gateways) and access control of application layer services including multiple-level of request translation. Network and system configuration can be modeled together and used to verify properties using CTL-embedded functions translated into Boolean operations. We show that we can separate variables in two model checkers to reduce the state space and required resources. Yet, both models can be used to run combine queries.

Our future work includes enhancements in the model's performance for even faster execution and lower the construction time. Also, we plan to extending the supported devices, and node types to add more virtual devices and compound devices that can incorporate multi-node functionality as in some modern network-based devices. Moreover, a user interface for facilitating interactive execution of queries as well as updating and editing the configurations for a more practical deployment patterns for the tool. We will also try to find a practical mapping scheme between application requests and corresponding packet flows to automatically detect the flows required to communicate a request between different services.

References

[1] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE INFOCOM'04*, March 2004.

[2] E. Al-Shaer and H. Hamed. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3), March 2006.

[3] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP*, pages 123–132, 2009.

[4] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 111–122, New York, NY, USA, 2008. ACM.

[5] H. Ballani and P. Francis. Conman: a step towards network manageability. *SIGCOMM Comput. Commun. Rev.*, 37(4):205–216, 2007.

[6] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Journal of Information and Computation*, 98(2):1–33, June 1992.

[8] R. Bush and T. Griffin. Integrity for virtual private routed networks. In *IEEE INFOCOM 2003*, volume 2, pages 1467– 1476, 2003.

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

[10] Z. Fu, F. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu. IPSec/VPN security policy: Correctness, conflict detection and resolution. In *Policy'2001 Workshop*, pages 39–56, January 2001.

[11] T. G. Griffin and G. Wilfong. On the correctness of IBGP configuration. In *SIGCOMM '02: Proceedings of the ACM SIGCOMM 2002 conference on Data communication*, pages 17–29, 2002.

[12] J. Guttman. Filtering posture: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy*, pages 120–129, May 1997.

[13] Hazem Hamed, Ehab Al-Shaer and Will Marrero. Modeling and verification of IPSec and VPN security policies. In *IEEE International Conference of Network Protocols (ICNP'2005)*, Nov. 2005.

[14] S. Hinrichs. Policy-based management: Bridging the gap. In 15^{th} *Annual Computer Security Applications Conference (ACSAC'99)*, pages 209–218, December 1999.

[15] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing a distributed firewall. In 7^{th} *ACM Conference on Computer and Comminications Security (CCS'00)*, pages 190–199, November 2000.

[16] I. Luck, C. Schafer, and H. Krumm. Model-based tool assistance for packet-filter design. In *IEEE Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, pages 120–136, January 2001.

[17] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM '02: Proceedings of the ACM SIGCOMM 2002 conference on Data communications*, pages 3–16, New York, NY, USA, 2002. ACM.

[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. *SIGCOMM Comput. Commun. Rev.*, 41(4):290–301, Aug. 2011.

[19] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *IEEE Symposium on Security and Privacy (SSP'00)*, pages 177–187, May 2000.

[20] S. Narain. Network configuration management via model finding. In *LISA*, pages 155– 168, 2005.

[21] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[22] G. G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *IEEE INFOCOM 2005*, volume 3, pages 2170– 2183, 2005.

[23] Y. Yang, C. U. Martel, and S. F. Wu. On building the minimum number of tunnels: An ordered-split approach to manage ipsec/vpn tunnels. In *In 9th IEEE/IFIP Network Operation and Management Symposium (NOMS2004)*, pages 277–290, May 2004.

[24] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE Symposium on Security and Privacy (SSP'06)*, May 2006.