

Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security

Ehab Al-Shaer

Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC, USA
ealshaer@sis.uncc.edu

Will Marrero, Adel El-Atawy and Khalid ElBadawi

School of Computing
DePaul University
Chicago, IL, USA
{wmarrero,aelatawy,badawi}@cs.depaul.edu

Abstract—Recent studies show that configurations of network access control is one of the most complex and error prone network management tasks. For this reason, network misconfiguration becomes the main source for network unreachability and vulnerability problems. In this paper, we present a novel approach that models the global end-to-end behavior of access control configurations of the entire network including routers, IPSec, firewalls, and NAT for unicast and multicast packets. Our model represents the network as a state machine where the packet header and location determines the state. The transitions in this model are determined by packet header information, packet location, and policy semantics for the devices being modeled. We encode the semantics of access control policies with Boolean functions using binary decision diagrams (BDDs). We then use computation tree logic (CTL) and symbolic model checking to investigate all future and past states of this packet in the network and verify network reachability and security requirements.

Thus, our contributions in this work is the global encoding for network configurations that allows for general reachability and security property-based verification using CTL model checking. We have implemented our approach in a tool called ConfigChecker. While evaluating ConfigChecker, we modeled and verified network configurations with thousands of devices and millions of configuration rules, thus demonstrating the scalability of this approach.

I. INTRODUCTION

The network behavior depends not only on the protocol implementation but also on the configuration that might control the protocol operation. Network traffic from source to destination passes through many different network devices such as routers, firewalls, NAT, and IPSec gateways. Each of these devices operates based on protocols and locally configured access control policies that match incoming traffic against filtering rules and determine the appropriate action. If there is a match, the corresponding action of the matching rule is performed. Forwarding, discarding or encrypting traffic are examples of rule actions in routers, firewalls and IPSec policies respectively.

This research was supported in part by National Science Foundation under Grant No. CNS-0716723. Any opinions, findings, conclusions or recommendations stated in this material are those of the authors and do not necessarily reflect the views of the funding sources.

A typical enterprise network might include hundreds to thousands of access control devices, each containing hundreds or possibly thousands of rules. Although access control policies are configured locally in isolation from each other, they are logically composed to implement global end-to-end reachability and security requirements. Manual analysis of the interaction among the policies of the many devices in the network with different syntax and semantics is infeasible. The increasing complexity of managing access control configurations due to larger networks and longer policies makes configuration errors highly likely [5]. These misconfigurations can cause major network failures such as reachability problems, security violations, and network vulnerabilities. Recent studies have found that more than 62% of network failures today are due to network misconfiguration [5]. Moreover, a recent survey by Arbor Networks shows that managing access control is one of the most challenging issues faced by ISPs. Configuration analysis is very important for debugging network problems and isolate errors due to protocol implementation bugs or configuration errors.

In the past few years, many researchers have attempted to address various challenges in network configuration. In [3], [13], [20], different techniques are presented to identify configuration conflicts between firewall and IPSec devices. Approaches for analyzing routing configuration using static or formal analysis [8], [18], and on-line debugging [5] are proposed. Top-down configuration approaches have also been proposed [17]. Unlike previous work, our system, ConfigChecker, provides comprehensive end-to-end network configuration analysis engine to model all network access control devices (routers, firewalls, NAT, IPSec gateways) and offer a rich logic-based interface for general property-based network configuration analysis.

ConfigChecker models the entire network as a state machine, where each state is defined by the location and header information of packets in the network. We model the behavior of each access control device using binary decision diagrams (BDDs) [6]. This network abstraction allows the use of symbolic model checking techniques [7] to verify

reachability and security properties written in CTL [9]. Unlike previous work, ConfigChecker considers techniques to obtain an expressive interface and scalable implementation. We extended CTL to provide additional operators that are particularly useful for verifying security properties. We also use efficient variable ordering in the BDD encoding to optimize BDD operations and space requirement. Thus, ConfigChecker can verify reachability, and identify security policy violations such as backdoors or broken tunnels due to, for example, errors or inconsistency between any two or more devices in the network. Therefore, the novelty of this work is the creation and the optimization of a symbolic model checker that abstracts the end-to-end network configuration behavior and using this to verify reachability and security properties. Our approach use a static analysis which assumes fairly stable network configuration but incremental/localized reconstruction of the state due to dynamic configuration is also feasible and it does not require rebuilding the entire state.

The rest of this paper is organized as follows. We first describe our network model and abstraction in Section II. Section III presents the implementation of the system. We then present ConfigChecker application in configuration verification and analysis in Section IV. In Section V, we present our evaluation framework and results. The related work is presented in Section VI. We finally present our conclusion and future remarks in Section VII.

II. THE MODEL

We model the network as a giant finite state machine. The state of the network is determined by the packets in the network. The relevant information about a packet includes the packet header information (which is what determines what various devices in the network do with the packet) together with the packet’s current location. We begin with a simple model where we abstract away details about the packet payload and focus only on the data contained in packet headers. We then extend this model to include certain payload information that results from the encapsulation performed by IPsec devices.

A. Basic Model

In the basic model, 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:

$$\sigma : \mathbf{IP}_s \times \mathbf{port}_s \times \mathbf{IP}_d \times \mathbf{port}_d \times \mathbf{loc} \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

- \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 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 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 of the packet 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). 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. Some examples of how real devices can be encoded in this way should help to illustrate our technique. However, to keep the formulas simple, the examples will contain only 2 bits for the source IP, destination IP, and location IP, and 1 bit for the source port and destination port. Real examples are similar, but with larger (32-bit or 16-bit) fields. The formulas will use the following variables:

- s_1, d_1, l_1 The high order bit in the source IP address, the destination IP address, and the location IP address respectively.
- s_0, d_0, l_0 The low order bit in the source IP address, the destination IP address, and the location IP address respectively.
- s_p, d_p The source port bit and the destination port bit respectively.
- s'_0, s'_1, d'_0, \dots means the same as the unprimed versions above, except that these variables represent the values of the bits in the next state.

The use of “location” as parameter in our abstraction allows for (1) constructing a global network abstraction of hetero-

geneous network devices, (2) better scalability as compared with fine-grain rule abstraction [20], (3) investigating device-specific quires, and (4) faster localized state rebuilding when dynamic configuration update is used.

Firewall: Assume a firewall with IP address of 1 has its outbound interface connected to a device with IP address 3. Furthermore, suppose it allows all traffic from IP address 2 to any destination as well as all traffic destined to port 1 on the device with IP address 3 coming from any source. All other traffic is blocked. Then all packets that satisfy the following formula will be forwarded to the outgoing interface:

$$(s_1 \wedge \overline{s_0}) \vee (d_1 \wedge d_0 \wedge d_p)$$

Furthermore, in the new state, the packet will look identical, except that it's location will be IP address 3. Therefore, the restriction on the new state is:

$$\bigwedge_{i \in \{0,1,p\}} (s'_i \Leftrightarrow s_i) \wedge \bigwedge_{i \in \{0,1,p\}} (d'_i \Leftrightarrow d_i) \wedge l'_1 \wedge l'_0$$

Finally, this transformation only occurs if the packet is currently at the firewall. In other words, the current location of the packet must be IP address 1:

$$\overline{l_1} \wedge l_0$$

Therefore, transition relation for this particular firewall is the conjunction of the three conditions above. In general, the filtering policy of any firewall can be converted into a formula similar to the one in the example. This formula encodes the fact that all packets accepted by the firewall get forwarded to the device connected to the outgoing interface, while there is no next state for rejected packets.

This construction assumes that we can encode a firewall's filtering policy as a Boolean formula over the bits in the packet header information. A simple (but quite large) disjunction of all the minterms representing packets that are accepted demonstrates that this is theoretically possible. In practice, a list of filtering rules is used to configure the behavior of the firewall. This list of rules can be used to directly construct the formula as demonstrated in [13].

Router: Assume a router with IP address 2 sends all packets destined for IP addresses 1 and 0 to IP address 0 (next hop), while all other packets are sent to IP address 3 as a default gateway. The logic for the routing decisions is given by the following formula:

$$(\overline{d_1} \wedge \overline{l'_1} \wedge \overline{l'_0}) \vee (d_1 \wedge l'_1 \wedge l'_0)$$

Since no packet header information changes we also have the restriction

$$\bigwedge_{i \in \{0,1,p\}} (s'_i \Leftrightarrow s_i) \wedge \bigwedge_{i \in \{0,1,p\}} (d'_i \Leftrightarrow d_i)$$

Finally, this transformation only takes place when the packet is currently at the router; therefore, the following restriction must be satisfied as well:

$$l_1 \wedge \overline{l_0}$$

The transition relation for this particular router is the conjunction of the three conditions above.

NAT device: We model a device that performs network address translation the same way we model a router, except that this device might change the packet header information. For example, suppose a NAT device with IP address 2 is hiding a service on port 1 of a device with IP address 3 behind it and the NAT device is connected to device with IP address 1 in front of it. So any traffic from IP address 3 port 1 is sent to IP address 1, but the source information is changed to IP address 2 port 0. Additionally, any inbound traffic destination IP address 2 and port 0 is really intended for the hidden service, so it is forwarded to location 3, but its header information is changed so that the destination is now IP address 3 port 1. We will assume all other traffic is blocked. The restrictions on this device look like:

$$[s_1 \wedge s_0 \wedge s_p \wedge \overline{l'_1} \wedge l'_0 \wedge s'_1 \wedge \overline{s'_0} \wedge \overline{s'_p} \wedge \bigwedge_{i \in \{0,1,p\}} (d'_i \Leftrightarrow d_i)]$$

∨

$$[d_1 \wedge \overline{d_0} \wedge \overline{d_p} \wedge l'_1 \wedge l'_0 \wedge d'_1 \wedge d'_0 \wedge d'_p \wedge \bigwedge_{i \in \{0,1,p\}} (s'_i \Leftrightarrow s_i)]$$

Once again, this only happens if the packet is at the NAT device, so we must have the additional restriction:

$$l_1 \wedge \overline{l_0}$$

The transition relation for this NAT device is the conjunction of the two conditions above.

B. Extended Model

In order to model devices that perform encapsulation, we need to extend the state space so that the packet information includes not only the top level IP header information, but also the IP headers that appear in the payload of encapsulated packets. We do this by adding extra copies of the IP header fields (source IP, source port, destination IP, and destination port), one extra copy for each additional level of encapsulation that we want to allow in the model. Additionally, each copy will need a few extra bits to encode other information that affects IPsec behavior (such tunnel vs. transport and AH vs. ESP). We also need an extra "valid" bit for each copy that records whether or not a particular IP header copy is actually being used or not. This bit will only be on if the packet has been encapsulated. Note that none of the devices discussed so far make use of these extra copies. In the extended model we would need to add further restrictions to the transition relations for these devices that ensure that the values of these extra bits are not changed by these devices.

IPSec: Our model for an IPSec device is best explained by comparing it to our model of a firewall. If a packet is not protected, then the transformation behaves much like firewall. If the packet is protected, then in addition to the normal forwarding behavior, we also modify the header information. Encapsulation is modeled by “shifting” all IP header information one level/copy deeper and setting the next available “valid bit”. The new IP header information is then assigned to the outermost header copy which is the one that is used by all devices to make routing/filtering/transformation decisions. If a packet is decapsulated, then all header information is “shifted out” one position and the deepest “valid bit” is cleared. Note that this assumes that at least one valid bit was set. If there was not valid bit set, then the packet we are trying to decapsulate is actually not encapsulated so it should be dropped. Note that some IPSec devices have a recursive semantics in that after transforming the packet, the packet is basically fed back through the IPSec device again. We easily handle this case by setting the next location of a protected packet to be the address of the same IPSec device.

To present a simplified model for the IPSec device, we will need to add copies of all the IP header bits. The simplest case is where there is only one possible level of packet encapsulation so we only need one copy of the IP header bits. We will use a “hat” mark over variable names to indicate they represent the inner nested packet header. For example, the variable \hat{d}_1' represents the value of the high order bit in the destination IP address that has been encapsulated and now resides in the packet payload in the new state. This bit plays no role in the behavior of the packet unless it is later decapsulated so that it becomes part of the actual packet header again. Suppose a device with IP address 0, sends a packet to IP address 3. The first hop lands the packet at IP address 1 which protects the packet by encapsulating it and sending it to IP address 2, which is both the next hop and the endpoint of the tunnel. The encapsulation requires that we make use of the copies of the header bits. We will use the variable v to indicate that the encapsulated bits are valid (in use). The formula for the protection of this traffic is:

$$l_0 \wedge \bar{l}_1 \quad (1)$$

$$\wedge \bar{s}_0 \wedge \bar{s}_1 \wedge d_0 \wedge d_1 \quad (2)$$

$$\wedge \bar{v} \wedge v' \quad (3)$$

$$\wedge \bigwedge_{i \in \{0,1,p\}} (\hat{s}_i' \leftrightarrow s_i) \wedge \bigwedge_{i \in \{0,1,p\}} (\hat{d}_i' \leftrightarrow d_i) \quad (4)$$

$$\wedge s'_0 \wedge \bar{s}'_1 \wedge s'_p \leftrightarrow s_p \wedge \bar{d}'_0 \wedge \bar{d}'_1 \wedge d'_p \leftrightarrow d_p \quad (5)$$

$$\wedge \bar{l}'_0 \wedge l'_1 \quad (6)$$

Note that line (1) ensures that the packet is at the IPSec device in question while line (2) ensures that the packet header satisfies the protection condition (namely a packet from IP address 0 to IP address 3). Line (3) ensures that the encapsulated bits are not currently being used while requiring that they be marked as used in the next state. Line (4) specifies that the current IP header bits should be copied to the

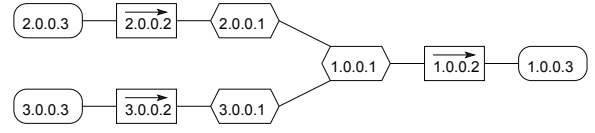


Fig. 1. Toy Network Example

loc	src	dst	loc'	src'	dst'
2.0.0.3	*	*	2.0.0.2	src	dst
2.0.0.2	*	*	2.0.0.1	src	dst
2.0.0.1	*	1.*.*.*	1.0.0.1	src	dst
2.0.0.1	*	3.*.*.*	1.0.0.1	src	dst
3.0.0.3	*	*	3.0.0.2	src	dst
3.0.0.2	*	*	3.0.0.1	src	dst
3.0.0.1	*	1.*.*.*	1.0.0.1	src	dst
3.0.0.1	*	2.*.*.*	1.0.0.1	src	dst
1.0.0.1	*	1.*.*.*	1.0.0.2	src	dst
1.0.0.1	*	2.*.*.*	2.0.0.1	src	dst
1.0.0.1	*	3.*.*.*	3.0.0.1	src	dst
1.0.0.2	2.*.*.*	1.0.0.3	1.0.0.3	src	dst

TABLE I

PORTION OF GLOBAL TRANSITION RELATION FOR FIGURE 1

encapsulated bits. Line (5) specifies what the new IP header information should look like. The packet now looks like it originated at IP address 1 and is destined for IP address 2. Finally, line (6) specifies the next hop for the packet.

More logic would have to be included for how other packets should be treated. That logic will look very much like the firewall logic where some packets are allowed to move to the next hop without any modification, while other packets are simply dropped. Finally, the dual of the rule above would need to appear in the encoding for the IPSec device with IP address 2. Instead of copying bits from the header information to the encapsulation bits and turning on the valid bit, we would copy bits from the encapsulation bits to the IP header and turn off the valid bit.

C. Model Checking Example

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. To get the transition relation for the entire network, we simply take the disjunction of the formulas for the individual devices.

Consider the toy network depicted in Figure 1. The hexagons are routers, the rectangles are firewalls, and the rounded rectangles are hosts. The arrows in the firewalls indicate in which direction filtering is performed. No filtering is performed in the opposite direction. Assume that 1.0.0.2 allows only traffic with source in the range 2.*.*.* while neither of the other two firewalls perform any filtering. Also assume the “obvious” routing tables for the routers. Then after building up the individual transition relations for the individual nodes in the network, the global transition relation would include the transitions listed in table I. For the sake of brevity, only the transitions used in our example are listed. The majority of valid transitions are missing from the table.

We can now compute the set of packets that can eventually reach the host 1.0.0.3 using symbolic model checking techniques [7]. We begin with the set of packets of interest, namely the packets at the host. This set is characterized by the logic formula $loc = 1.0.0.3$. Call this set, S_0 , the set of packets that are 0 transitions away from the goal. To compute the set S_1 (the set of packets that are one transition away from the goal, “plug in” the set S_0 into the next state variables of the transition relation and ask what assignments to the current state variables satisfy the transition relation. This gives us that S_1 is the set of packets satisfying the constraint $loc = 1.0.0.2 \wedge src = 2.*.*.* \wedge dst = 1.0.0.3$. Similarly, we do the same to compute the set S_2 (two transitions away from the goal). However, when packets at 1.0.0.1, they could have come from two different places. And indeed, when we compute S_3 , we get the formula

$$(loc = 2.0.0.1 \wedge src = 2.*.*.* \wedge dst = 1.0.0.3) \\ \vee \\ (loc = 3.0.0.1 \wedge src = 2.*.*.* \wedge dst = 1.0.0.3)$$

Continuing this process, packets in S_4 would satisfy similar formula except that the packets would have to be at location 2.0.0.2 or 3.0.0.2. Finally, S_5 would also be similar, but with location 2.0.0.3 or location 3.0.0.3. The set of all packets that could reach location 1.0.0.3 would then be $S_0 \vee S_1 \vee \dots \vee S_5$. When we discuss the specification language CTL in section IV, we shall see that this is the set of packets satisfying $\mathbf{EF}(loc = 1.0.0.3)$. However, to consider only the packets originated at 3.0.0.3, we take the intersection with the set of packets at location 3.0.0.3 as in this formula $(S_0 \vee S_1 \vee \dots \vee S_5) \wedge (loc = 3.0.0.1)$, or simply $(loc = 3.0.0.3 \wedge src = 2.*.*.* \wedge dst = 1.0.0.3)$. We can insert a filtering rule at the firewall 3.0.0.2 to prevent spoofing.

III. IMPLEMENTATION

From section II, a state in our model consists of the values of the various bits in a packet header, together with a location ID for each device. We use 104 bits to encode the basic packet header information (protocol, source IP/port, destination IP/port), 2 bits to indicate IPSec modes and 5 bits for IPSec parameters for a total 111 bits for the packet header. Recall that the extended model requires us to track the encapsulation/decapsulation of packets that is performed by IPSec devices. This requires storing the old packet header information so that it can be restored once the encapsulated packet reaches the end of a tunnel. Allowing nested tunnels will require an extra copy per level. In other words, if we allow 4 nested tunnels, then we will need to have 5 replicas of the packet headers; $5 * 111$ or 555 bits. Moreover, to keep track of which of those copies are actually being used we need a “valid bit” per copy (the outermost is always valid). We encode the location via a 32 bits for the IP address of the device for a total of 591 bits to encode the state.

Of course, it is infeasible to explicitly model a system with such high number of variables. However, by keeping track of

the states and the transition relation implicitly via formulas represented using BDDs, it is possible to perform analysis on a model so large. A BDD represents a formula as decision graph where the nodes in the graph are vertices and the edges coming out of a vertex represent the two possible Boolean assignments to that variable. Thus, a complete assignment to all variables corresponds to a path in the graph which ends in a value of true or false, which is the value of the formula when given that assignment. The size of this graph (the BDD) is extremely sensitive to the order in which the variables appear in the decision graph. We use the common interleaving heuristic which places the next state copy of a variable right next to the current state copy of the variable in the ordering. Furthermore, we try to keep variables that are related, close to each other. For this model, variables that belong to different copies of the same field are tightly related. When a tunnel is created, the current packet header copies are “pushed” one level deeper and a new packet header is created at the outer most level and the opposite when a tunnel terminates. For example, suppose a packet header consisted of three variables, x, y , and z . Also suppose we had 3 copies of the packet header to keep track of, so we now label the variables $x_0, y_0, z_0, x_1, y_1, \dots$ where x_i corresponds to the value of the variable x in the i^{th} nested copy of header information. To keep the bits of the copies near each other, we use the ordering $x_0, x_1, x_2, y_0, y_1, y_2, z_0, z_1, z_2$. If we add the next state variables as primed variables, the ordering becomes $x_0, x'_0, x_1, x'_1, x_2, x'_2, y_0, y'_0, y_1, y'_1, y_2, y'_2, z_0, z'_0, z_1, z'_1, z_2, z'_2$. For our implementation, we are using BuDDy which is a well known package that implements BDD operations and memory management. The package is implemented in C, with an added encapsulation for C++.

IV. AUTOMATED CONFIGURATION VERIFICATION AND ANALYSIS USING CONFIGCHECKER

In this section, we show two approaches to analyze network configurations: (1) verifying that the end-to-end reachability of the access control configuration is sound and complete, and (2) verifying number of end-to-end security properties to identify misconfiguration or undesired network behavior such as policy conflicts [3], [4], [13], policy violation and suboptimized configuration. For this purpose, we developed ConfigChecker Query Interface (CI) based on CTL [9] to demonstrate the effectiveness of ConfigChecker in verifying the end-to-end network configurations globally and transparently. However, developing a high-level configuration language that hides CTL primitives and offers more usable interface is not the focus of this paper but part of our future work.

A. ConfigChecker Query Interface– Extending CTL

We consider a number of end-to-end properties that we might want to verify about a network. For each property, we describe how that property can be expressed in a computation tree logic (CTL) query which we would feed to our tool. Generally, a query is a Boolean expression that is being executed against a network configuration that has been loaded

and compiled into a transition relation as shown in Section II. The Boolean expression can be defined over any of the packet header fields, and can include standard Boolean operations as well as CTL primitives. A complete and formal presentation of CTL is in [9].

CTL has a number of temporal operators that allow specifications that express future behavior. Because CTL can be used for non-deterministic models, these temporal operators can require that some future state exhibits a behavior (exists) or all future states exhibit a behavior (forall). Our models are typically deterministic; however we do use non-determinism to model the possibility of multiple routes and to model multicast. For example, intuitively a state satisfies $\mathbf{EX}(loc = 140.192.1.1)$, if there is a next state in which the packet is at location 140.192.1.1. A state satisfies $\mathbf{AX}(loc = 140.192.1.1)$ if in all next states, the packet is at location 140.192.1.1. If multiple routes or multicast were being modeled, $\mathbf{AX}(loc = 140.192.1.1)$ would insist that there is only one next hop.

Similarly, a state satisfies $\mathbf{EF}(loc = 140.192.1.1)$ if there is a path from this state along which eventually the location of the packet is 140.192.1.1. A state satisfies $\mathbf{AF}(loc = 140.192.1.1)$ if along all paths from this state, eventually the packet is 140.192.1.1. Again, $\mathbf{AF}(loc = 140.192.1.1)$ would be useful to check that if there are multiple routes, they all lead to the same destination.

Although model checking commonly searches states in future sense, we use the extension to CTL that includes operators to enable investigating states in the past. We use a subscript – with each of the operators described above to denote the identical operator but considering paths going backwards in time. These extra operators will be useful for us because many security properties have the form “if A is true now, then B must have happened in the past”. For example, \mathbf{EX}_f talks about a predecessor state (or hop) while \mathbf{EX}^f talks about a successor state. The implementation detail of the past operators are in our technical report.

In the following sections, we describe applications of ConfigChecker by referring to the queries in Table II. These queries need specific values/addresses to be customized to a certain network (e.g., ip addresses of routers, port, etc). However, as part of CI, these queries can be used iteratively over the entire value range.

B. Reachability Verification

Routinize misconfiguration is not the only cause of reachability problems. Other access control configuration in the path like firewall, NAT or IPSec might interfere with routing and also cause unreachability problems. Thus global end-to-end access control analysis is required to verify reachability. In this section, we show how ConfigChecker can prove the end-to-end reachability soundness and completeness of the access control configuration including all different devices. Thus, we do not assume all physically connected nodes should be actually connected through configuration like the case in routing-only analysis [14]. However, the valid connectivity is determined based on the Connectivity Requirement Policy (CRP) which

considers the authorization access and the physical topology. So let us assume that $\mathcal{P}_{connect}(src, dest)$ as a characterization function for all sets of allowed flows from src to $dest$ (IP and port numbers) that represents CRP.

For a basic reachability questions: $Q1$ in Table II shows whether the IP and port address ($a1$) can reach the IP and port address ($a2$). Moreover, if all types of traffic are allowed between the two hosts, the result will be the *true* value, otherwise it will be a boolean expression that represents only flows (services) that will reach the destination. Of course, we can use port numbers along with IP to query a specific flow.

Definition 1. A configuration, \mathcal{C} , is *sound* if, for all nodes u and w , all possible paths from u to w are subset of (or implies) authorized paths in $\mathcal{P}_{connect}(u, w)$. In other words, no node u can reach node w using \mathcal{C} if this is not allowed in CRP. Note that u and w can be defined based on IP and port. If the port is unspecified, all possible ports are considered in the analysis. $Q2$ in the Table II states formally this condition using CI.

Definition 2. A configuration, \mathcal{C} , is *complete* if for all nodes u and w , $\mathcal{P}_{connect}(u, w)$ subset of (or implies) all possible paths from u to w . Intuitively, if the CRP does not authorize u to reach w then there is no allowed path (no routing or packets are discarded) in \mathcal{C} from u to w . $Q3$ in the Table II states formally this condition using CI.

If soundness or completeness condition fails, then the negation of the resulting expression will give us the states of all incorrect configurations that violate the access authorization or the physical topology¹. Assuming the number of misconfigurations is not largely excessive, we can then extract every misconfiguration scenario (including flow and location) by finding a satisfying assignment using *bdd-sat* operator in BDD [15] and then fix it.

The soundness condition enforces Loop-free property. However, it is sometimes useful to verify this property for specific device configuration [14]. $Q4$ in the Table II shows the loop-free condition for a node.

C. Discovering Security and Reachability Misconfiguration

In this section, we show more examples of ConfigChecker queries to discover specific misconfiguration such as security violations or suboptimized configuration. Although some of these conditions might also unsatisfy soundness and completeness condition, it is useful to search for specific misconfiguration for debugging of network behavior

Shadowing or bogus entries: It is useful to know if an access control node has a forward entry for a flow that will never reach it. This could occur if for example there is an upstream node in the path discards a flow that is or accepted by a downstream router or firewall respectively. Another reason of this condition is the creation of bogus routing entries (i.e., invalid next-hop) due to routing protocol bugs. $Q5$ in the Table II shows how to detect his misconfiguration.

IPSec Tunnel Integrity: *Is my traffic encrypted all the way from source to destination? Are the intermediate gateways*

¹The last one could occur due to a bug in the protocol.

TABLE II

EXAMPLE OF QUERIES THAT CAN BE USED IN CONFIGCHECKER, WITH A BRIEF EXPLANATION OF EACH.

Basic reachability	
Q1:	$(src = a1 \wedge dest = a2 \wedge loc(a1)) \rightarrow \mathbf{AF}(src = a1 \wedge dest = a2 \wedge loc(a2))$ <i>Given a starting location and a flow, does packets of this flow eventually reach the destination?</i>
Reachability Soundness	
Q2:	$[loc(a1) \wedge src(a1) \wedge dst(a2) \wedge \mathbf{EF}(loc(a2))] \rightarrow \mathcal{P}connect(a1, a2)$ <i>If the src can reach the destination in configuration then it must be allowed in CRP.</i>
Reachability Completeness	
Q3:	$\mathcal{P}connect(a1, a2) \rightarrow [loc(a1) \wedge src(a1) \wedge dst(a2) \rightarrow \mathbf{EF}(loc(a2))]$ <i>if CRP allows a1 to reach a2, then there must a path in the configuration that eventually allows a1 to reach a2.</i>
Discovering routing loops	
Q4:	$loc(a1) \wedge \mathbf{EX}(\mathbf{EF}(loc(a1)))$ <i>Is there a node that can reach a1 and for the same flow it is the next hop of a1?</i>
Shadow or Bogus routing entries	
Q5:	$\mathbf{EX}(true) \wedge \neg \mathbf{EX}(true) \wedge (loc(router1) \vee loc(router2) \dots)$ <i>Given all routers, does any have a decision for traffic will never reach it from its previous hop?</i>
End-to-end integrity of single/nested or cascaded IPSec encrypted tunnel	
Q6:	$(src = a1 \wedge dest = a2 \wedge loc(a1) \wedge IPsec(encT)) \rightarrow \mathbf{AU}((IPsec(encT) \vee loc \rightarrow \mathcal{G}), loc(a2))$ <i>If the traffic is encrypted in a tunnel from the src then it will appear decrypted only at the destination or at intermediate authorized gateways (\mathcal{G}) that allow for cascaded tunnels. If $\mathcal{G} = false$, then there are no intermediate gateways and the traffic must travel through a single tunnel.</i>
Comparing configuration for backdoors or broken flows after route changes	
Q7a:	$C_{org} \triangleq [\neg multiroute \wedge src = a1 \wedge dest = a2 \wedge loc(a1) \rightarrow \mathbf{AF}(loc(a2) \wedge src = a1 \wedge dest = a2)]$
Q7b:	$C_{new} \triangleq [multiroute \wedge src = a1 \wedge dest = a2 \wedge loc(a1) \rightarrow \mathbf{AF}(loc(a2) \wedge src = a1 \wedge dest = a2)]$
Q7:	$Backdoors: \neg C_{org} \wedge C_{new}, Broken\ flows: \neg C_{new} \wedge C_{org}$ <i>what is different in the new configuration as compared with the ordinary original one. Is there any backdoor?</i>

authorized ones? In addition to reachability, ConfigChecker can be used to verify that if a given configuration also satisfies specific security properties such end-to-end encryption. It has been numerously reported that due to misconfiguration in IPSec VPN configuration a traffic might be seen as plain text although it was encrypted and tunneled from source to destination [13], [19]. Q6 in Table II verifies this property for both single and cascaded tunnels.

Discovering backdoors and broken flows: It is necessary to guarantee that changes in routing configuration as a result of, for example, link failures will not violate security properties such as allowing access that was originally denied. Correct firewall deployment when multi-routing is used can be sometimes tricky. Q7 in Table II checks if multiple paths between any two hosts exhibit different flow or security properties. We implement this by comparing the configuration of the longest-prefix route, C_{org} , with multi-route configuration, C_{new} , (a disjunctive of all possible routes). It is important to note that the backdoors exist only if C_{new} offers more accessibility than C_{org} . The opposite means reachability problems due to broken flows. Otherwise, the access control configurations are consistent before and after the change/failure.

V. EVALUATION

In this section, we show space requirements and performance evaluation of the framework with respect to the memory and time requirements. We evaluated ConfigChecker using more than 90 networks. We focus our study on the time and space for building and running the model. We study the effect

of several network configuration parameters: network size, connectivity (branching factor), number of subnets, number of configuration rules in the network, number of firewalls/IPSec devices and ACL rules. As part of our collaboration with Cisco, our BDD-based policy creation engine was tested on real network configurations. In our our previous work [2], we showed that our BDD implementation can handle up to 30K rules in a single policy efficiently.

In order to thoroughly test our framework, we have to use a large set of configuration scenarios with varying parameters and sizes. Using only real configurations will limit the number as well as the possibilities of the test scenarios. Thus, we developed our own tool to generate network topologies and valid access control configurations. Misconfigurations were manually injected for testing and evaluation purposes. The tool is given the network size, the number of components of each type (*i.e.*, routers, firewalls, host segments, etc.), and it generates the configuration files of each node, with routing tables, access-control policies, and device meta-information. A random network topology is generated as a mesh and then a hierarchical routing is constructed on this network based on the network depth (*i.e.*, average path length), average router fan-out and average number of subnets. Subnets and firewalls are then placed randomly base on the average percentage provided as an input. A set of valid IP address and netmasks are then randomly selected to generate firewall polices. The random rule generator considers various parameters including rule complexity, overlapping, and policy size as described

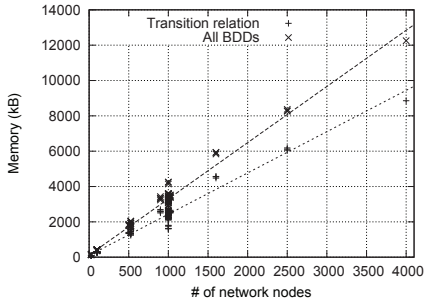


Fig. 2. Impact of network size on space requirements for a whole network.

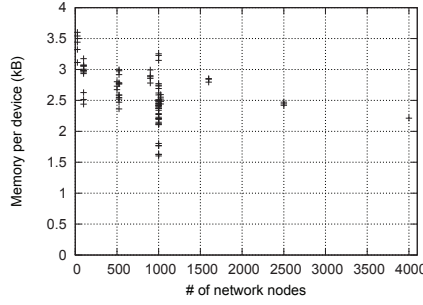


Fig. 3. Impact of network size on space requirements per device.

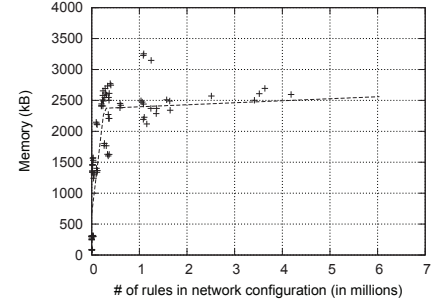


Fig. 4. Impact of network configuration size (in rule lines) on space complexity.

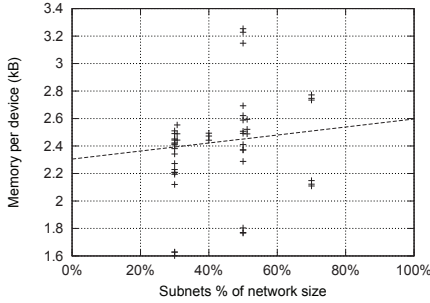


Fig. 5. Impact of the percentage of subnets on network complexity/space.

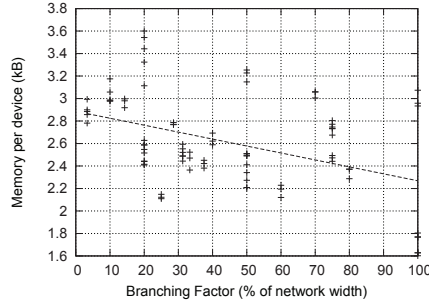


Fig. 6. Impact of the branching factor (number of outgoing links) on the network complexity.

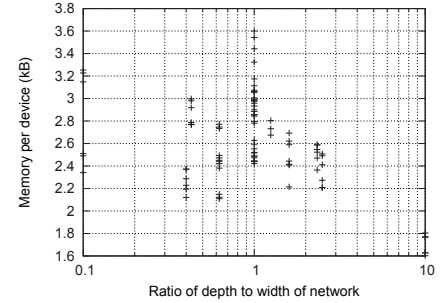


Fig. 7. Impact of the network's aspect ratio (depth by width) on memory requirements.

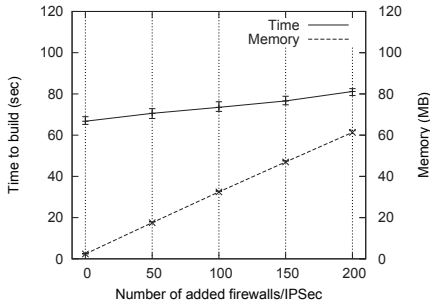


Fig. 8. Time/memory required for 3 different networks (Each has 600 routers, 400 domains), while adding 0 to 200 firewalls with same size.

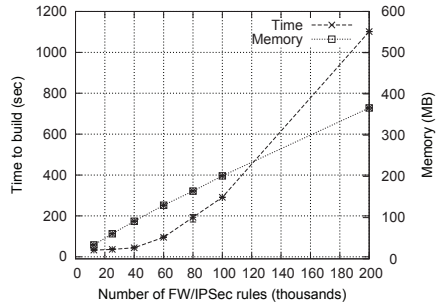


Fig. 9. Time/memory required for a network with 600 routers, 400 domains, and 50 firewalls with varying policy size.

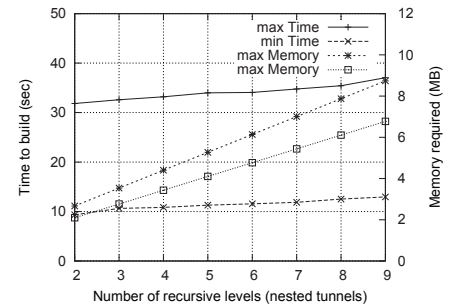


Fig. 10. Effect of supporting more encapsulation levels for IPSec tunnels. Time/memory required for two Networks of 1000 nodes each.

in [2]. Similarly, IPSec tunnels are randomly created between subnets based on the average VPN numbers, VPN mode (tunnel or transport), VPN structure (cascade or nested), and transformation type (ESP or AH). Details about the tool, its operational parameters and sample configurations are available in the projects webpage [1].

In this discussion, network size is the overall number of nodes in the network including: routers, security devices, and domains. We used networks of size that ranged from 25 nodes up to 4000 nodes, with total configuration sizes (routing tables, firewall/IPSec ACLs, etc) up to 6 million rules per network. It is worth mentioning that the execution time of all described queries is negligible, but we will discuss their complexity and provide some runtime measurements at the end of this section.

1) Impact of Network Size: We first consider the impact of the network size (or number of nodes) on the space complexity of the overall model. Space requirements are measured by the number of BDD nodes used for the model. This includes variable information and links between nodes. Each node requires 20 bytes of memory in the used implementation (BuDDy [15]). In Figure 2, we can see that the space required to compile a network is linear in its size. For 4K nodes, the space for the compiled model was less than 14MB which is very moderate considering the size. Our experiments also show that the memory requirement per device stays in a safe range (1-4K) regardless of the network size.

2) Impact of the number of Subnets: As subnets/domains defined in the network increases, the complexity of the model increases. Despite the fact that subnets are easier to represent

than other devices, we still see the dominant effect of routing information propagating in the network. Figure 5 shows the effect of subnet percentage on the space complexity for more than 20 networks of 1000 nodes each.

3) **Impact of the configuration size:** Configuration size measured in number of rules in routing and access control policies plays an obvious role in the overall network complexity. Figure 4 shows this relation clearly. However, we can see the growth stabilizing after a threshold where more configuration rules do not increase the space requirements. This behavior is due to two factors: 1) the redundancy in routing rules that manifests itself when many routing rules are propagated without being effective on the network operations, and 2) the capability of the BDD structure to merge expressions that are similar becomes more useful as policies increase in size.

4) **Impact of Branching Factor:** The number of outgoing links had a slight effect on the models used as shown in Figure 6. This can be attributed to the aggressive merging of common expressions of which the BDD structure is capable. Therefore, when a specific device has many outgoing links, only the links information is encoded. Moreover, the next-hop devices all belong to the same ring which makes their addresses differ in a few bits enhancing the possibility of further optimization by the BDD. The branching factor is given as a percentage of upstream links used out of the possible routers in the next higher level.

5) **Impact of Aspect Ratio:** For a given network, we define a network’s *depth* as the maximum number of hops between a node and the core router (*i.e.*, root router), and its *width* to be the maximum number of nodes having the same distance from the innermost core router. Figure 7 shows that the aspect ratio (*i.e.*, depth / width) has no significant effect on space requirements per device. However, one can notice that for smaller aspect ratio there is a higher variability in the memory requirements. This can be attributed to the wider range of the number of outgoing links resulting in more variability in the model and, in turn, a higher variability in complexity. In the figure, memory per device is used instead of the overall memory requirements to eliminate the effect of network size.

6) **Impact of adding filtering devices:** Firewalls and IPSec devices are expected to increase the space/time complexity as more fields are involved in the model than in routing. Figure 8 shows that as we add firewalls or IPSec devices (up to 200 added devices to a network of 1000 nodes), both space and time appear to grow almost linearly. However, the effect of increasing the policy size while fixing the number of firewalls is more evident in terms of building time. This is due to increasing potential of rule overlapping (*e.g.*, higher priority rules shadow the lower priority ones). Although this effect is usually quadratic, we reduce this effect in our implementation close to linear by exploiting the overlapping operations in BDDs. However, we can see in Figure 9 that the BDD operation is itself suffering slightly from the increased complexity of the overlapping rules as was expected but yet still in reasonable range. In fact these results match our intuition as BDD size grows linearly (in the worst case) with number of rules, and

BDD operation running time grows linearly with the BDD size. In fact, we can ideally expect logarithmic increase in BDD size as BDD optimization exploits the variable sharing.

7) **Impact of encapsulation levels:** To support devices capable of packet encapsulation (*i.e.*, IPSec), it is important to make sure the system is scalable in the number of nested tunnels. In figure 10, we show that the time and space required are linear in the number of levels (nested tunnels) for both: minimum and maximum cases. The two extreme network cases were chosen to have the same size (1000 nodes), but with varying branching density (30/device for *maximum net* and 10/device for *minimum net*).

8) **Copying expressions complexity:** In Section II, we mentioned using controlling expressions to be added to the transition relation of all devices to ensure packet information is copied as is between transitions. These expressions are independent of the network structure, size, and specific information. As in Table III, the memory needed is quite moderate, and will not pose any serious overhead when analyzing/building real networks. The table shows the size (*i.e.*, nodes in the BDD) used for each of the three expressions (“copy”, “push copy”, and “pop copy”) for different levels of encapsulation, where levels=1 means no encapsulation supported.

TABLE III
VARIABLES AND OPERATIONS COST FOR NESTED TUNNELS

Levels	1	2	3	4	5	6	7	8	9
Variables	282	500	718	936	1154	1372	1590	1808	2026
Copier	327	654	981	1308	1635	1962	2289	2616	2943
Push Copy	0	327	981	1635	2289	2943	3597	4251	4905
Pop Copy	0	327	654	981	1308	1635	1962	2289	2616

9) **Query Execution Cost:** Cost of executing most of the aforementioned queries is low in both space and time. The complexity of such queries is being delegated to the transition relation building phase, which was the focus of the results shown. This leaves the query to collect the results with minimal operations. For example, finding a routing cycle from a specific node requires a complete graph traversal in standard representation, while it needs a number of steps equal to the smallest cycle length if symbolic model checking is used.

Reachability and security queries (mentioned in Table II) when applied to a typical 1000 nodes network requires the following: queries with one step temporal relations (*e.g.*, EX, AX, etc) required a few milliseconds while queries including multi-step CTL operators (*e.g.*, EU, AF, AG, etc) required as high as 1.3 seconds for more complex examples. The space required in the highest point of a query execution was less than 5% of the overall transition relation space due to the immediate restrictions that are placed by the query definition.

10) **Impact of Dynamic Configuration:** Although our model assumes fairly stable network configuration, it can handle dynamic configuration efficiently. Intuitively, any configuration update will require rebuilding the device BDD and then update the global transition relation. The second one can be performed fairly quickly because it is literally a disjunction operation with the rest of other device BDDs. Rebuilding the BDD of a device is completely localized and it ranges from

0.01–0.05 seconds for a device with tens of thousand of rules. So for instance, the average cost of updating a single device in a network of 4000 nodes is measured as 0.13 seconds. This is considered very affordable when considering occasional configuration update such as routers, firewalls, NAT, etc.

VI. RELATED WORK

A significant amount of work was done in firewall configuration modeling and conflict analysis (e.g., [3], [11], [13], [20]). However, this work does not address reachability analysis and general property-based verification. Also, there has been considerable work recently in detecting misconfiguration in routing [5], [10], [12], [16]. Many of these approaches are specific for firewall or BGP misconfiguration. Other works made important steps towards creating general models for analyzing network configuration [8], [18]. An approach for formulating and deriving of sufficient conditions of connectivity constraints is presented in [8]. In [18], which is the closest to our approach, a graph-based approach is used to model connectivity of network configuration and set operations to perform a static reachability analysis. However, by using lessons learned in the model checking community we can do much more in both performance and application sides. Ignoring the cost of set operations, their algorithm runs in $O(V^3)$ time. If we ignore our set operations (BDD operations), our algorithm runs in $O(V)$ time. One reason for this is the fact that we consider *all* possible paths simultaneously instead of one at a time. Secondly, by adding additional BDD variables we can add other features that allow us to model and analyze more security devices and properties such as header transformations in IPsec. Finally, we make use of well studied logic (CTL) as our specification language which allows us to specify more properties and expressive queries in a more intuitive fashion than just using the language of reachability alone. Therefore, unlike the previous work that focus on end-to-end reachability properties [18], ConfigChecker can check any general properties a long the path.

VII. CONCLUSION AND FUTURE WORK

Managing the configuration of network access control devices such as routers, NAT, firewalls, and IPsec gateways is extremely complex and error-prone. This paper presents a model checker approach using Binary Decision Diagrams (BDDs) to create a unified model for network access control configuration regardless of devices function and matching semantic. We also extended CTL to provide an expressive query interface over this model for global end-to-end verification and "what-if" analysis. The model is implemented in a tool called *ConfigChecker*. We demonstrate effectiveness of ConfigChecker through the provability of soundness and completeness of the configuration reachability as well as discovering number of security violation examples such as backdoors and broken IPsec tunnels.

Our evaluation experiments using more than 90 networks of various configurations show that ConfigChecker requires less

than 5KB per device or less than 14MB in total for 4000 nodes and millions of configuration rules. We also show that the time and space complexity grow linearly in a reasonable rate with the size of the network, and dynamic configuration can be handled efficiently. The ConfigChecker implementation, query interface and the query examples stated in this paper are available for the public.

In the future work, we plan to extend our model to handle stateful devices as well as other network configuration such as QoS. We also plan to improve usability by developing a declarative high-level language to hide the CTL interface.

REFERENCES

- [1] Configchecker. <http://www.arc.cdm.depaul.edu/projects/ConfigChecker>.
- [2] E. Al-Shaer, A. El-Atawy, and T. Samak. Automated pseudo-live testing of firewall configuration enforcement. *Selected Areas in Communications, IEEE Journal on*, 27(3):302–314, 2009.
- [3] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE INFOCOM'04*, March 2004.
- [4] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications (JSAC)*, 23(10), October 2005. Nominated for Best JSAC Paper Award for year 2005.
- [5] Richard Alimi, Ye Wang, and Y. Richard 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.
- [6] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [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] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. MIT Press, 1990.
- [10] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.
- [11] M. Gouda and X. Liu. Firewall design: Consistency, completeness, and compactness. In *The 24th IEEE Int. Conference on Distributed Computing Systems (ICDCS'04)*, March 2004.
- [12] Timothy G. Griffin and Gordon Wilfong. On the correctness of IBGP configuration. In *SIGCOMM '02: Proceedings of the ACM SIGCOMM 2002 conference on Data communication*, pages 17–29, 2002.
- [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] Kirill Levchenko, Geoffrey M. Voelker, Ramamohan Paturi, and Stefan Savage. XL: An efficient network routing algorithm. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 15–26, New York, NY, USA, 2008. ACM.
- [15] J. Lind-Nielsen. The BuDDy OBDD package. <http://www.bdd-portal.org/buddy.html>.
- [16] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP misconfiguration. In *SIGCOMM '02: Proceedings of the ACM SIGCOMM 2002 conference on Data communications*.
- [17] S. Narain. Network configuration management via model finding. In *LISA*, pages 155–168, 2005.
- [18] G. G. Xie, J. Zhan, D.A. Maltz, H. Zhang, A. Greenberg, G. Hjaltmtysson, and J. Rexford. On static reachability analysis of ip networks. In *IEEE INFOCOM 2005*, volume 3, pages 2170–2183, 2005.
- [19] 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.
- [20] 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.