Demonstrating *topoS*: Theorem-Prover-Based Synthesis of Secure Network Configurations

Cornelius Diekmann, Andreas Korsten, Georg Carle

Technische Universität München {diekmann|korsten|carle}@net.in.tum.de

***Abstract*—In network management, when it comes to security breaches, human error constitutes a dominant factor. We present our tool *topoS* which automatically synthesizes low-level network configurations from high-level security goals. The automation and a feedback loop help to prevent human errors. Except for a last serialization step, *topoS* is formally verified with Isabelle/HOL, which prevents implementation errors. In a case study, we demonstrate *topoS* by example. For the first time, the complete transition from high-level security goals to both firewall and SDN configurations is presented.**

I. Introduction

Network-level access control is a fundamental security mechanism in almost every network. Unfortunately, configuring network-level access control devices still is a challenging, manual, and thus error-prone task [1]–[3]. It is a known and unsolved problem for over a decade that "corporate firewalls are often enforcing poorly written rule sets" [4]. Also, "access list conflicts dominate the misconfiguration errors made by administrators" [5]. A recent study confirms that this problem persists as a "majority of administrators stated misconfiguration as the most common cause of failure" [6]. In addition, not only is implementing a policy error-prone, but also developing it is challenging, even for experienced administrators [7].

We demonstrate our tool *topoS*: a constructive, top-down greenfield approach for network security management. *topoS* translates high-level security goals to network security device configurations. The automatic translation steps prevent manual translation errors. Furthermore, *topoS* visualizes the results of all translation steps to help the administrator uncover specification errors. In addition, since all intermediate transformation steps are formally verified, the correctness of *topoS* itself is guaranteed [8]. *topoS* is built on top of recent results of the formal methods community [7], [9], combines these results in a novel way, and transfers the knowledge to the network management community. The automated tool *topoS* is the main technical contribution of this paper.

We first give a short overview of *topoS* in Section II. Then, in Section III, we present *topoS* in detail with the help of a case study. We discuss limitations and advantages in Section IV, present related work in Section V, and conclude in Section VI.

II. Overview of *topoS*

The security requirements of networks are usually scenario-specific. Our tool *topoS* helps to configure a network according to these needs. It takes as input the high-level security requirements and synthesizes low-level security device configurations, e.g. netfilter/iptables firewall rules or OpenFlow flow table entries. It operates according to the following four-step process:

A. Formalize high-level security goals
   a. Categorize security goals
   b. Add scenario-specific knowledge
   c. ⋆ Auto-complete information
B. ⋆ Construct security policy
C. ⋆ Construct stateful policy
D. ⋆ Serialize security device configurations

All steps annotated with an asterisk are supported by *topoS*. As the ⋆-steps illustrate, once the security goals are specified, the process is completely automatic. Between the automated steps, manual refinement is possible but requires re-verification. This allows human intervention, while avoiding human error.

The automated intermediate[1] ⋆-steps are proven correct for all inputs. The proofs are verified with the interactive proof assistant Isabelle/HOL [10]. Isabelle/HOL is an LCF-style theorem prover; the correctness of derived facts is based on the correctness of a small inference kernel. This architecture is very robust and widely used for over a decade. In general, the formal methods community treats facts machine-verified with Isabelle/HOL as well-founded truth. Thus, it is guaranteed that *topoS* performs correct transformations [8]. As a side note, since the transformations are proven correct once and for all for all inputs, neither has a user to prove anything manually to use *topoS*, nor is Isabelle/HOL required to run *topoS*. The development of *topoS* started over three years ago and features, after a large rewrite, more than 10 k lines of formal proof.

We will present the steps *A* to *D* in the following section. For the sake of brevity and illustrative presentation, we only present them by example. Mathematical background has been presented in detail previously [7]–[9] and in this work, we focus on its interoperability and discuss how its underlying assumptions can be fulfilled in a real-world network. Further details, the correctness proofs, and the interplay of the individual steps can be found in the accompanying formalization and implementation of *topoS*.

[1] We did not verify the final step (i.e. serialization of security device configurations) since it is merely syntactic rewriting of the result of the previous step (c.f. Sect. III-D). Neither can we verify that the user expresses the security goals correctly. Yet, with the secure auto-completion (c.f. Sect. III-A) and the visual feedback of all intermediate results, we see reason that input errors are uncovered early.

III. *topoS* BY EXAMPLE

In this section, we demonstrate *topoS* with a small case study. The scenario was chosen because it is minimal and comprehensible, but also realistic and contains many important aspects. It runs live and is publicly available (c.f. Section Availability & Acknowledgements).

The case study is schematically illustrated in Fig. 1. The setup hosts a news aggregation web application, accessible from the Internet (*INET*). It consists of a web application backend server (*WebApp*) and a frontend server (*WebFrnt*). The *WebApp* is connected to a database (*DB*) and actively retrieves data from the Internet. All servers send their logging data to a central, protected log server (*Log*).

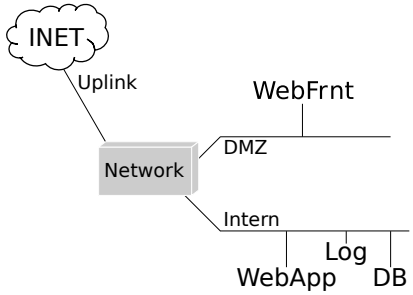

Figure 1. Network Schematic

We implemented the scenario to utilize several different protocols. A custom backend, the *WebApp* was written in `python`. The *WebFrnt* runs `lighttpd`. It serves static web pages directly and retrieves dynamic websites from the *WebApp* via `FastCGI`. All components send their `syslog` messages via UDP (RFC 5426) to *Log*.

A. Formalizing Security Goals

The security goals are expressed as security invariants over the network's connectivity structure. An invariant consists of a generic part (the semantics) and scenario-specific information. The generic part defines the type and general meaning. Our generic invariants currently defined are summarized in Table I.

Table I. GENERIC SECURITY INVARIANTS

Name	Φ	Description
Bell LaPadula	✓	Label-based Information Flow Security
Comm. Partners	✓	Simple ACLs (Access Control Lists)
Comm. With	✗	White-listing transitive ACLs
Not Comm. With	✗	Black-listing transitive ACLs
Dependability	✗	Limit dependence on certain hosts
Domain Hierarchy	✓	Hierarchical control structures
Refl	✓	Allow/deny reflexive flows. Can lift symbolic policy identifiers to role names
NonInterference	✗	Transitive non-interference properties
Security Gateway	✓	Master/Slave relationships
Sink	✓	Information sink
Subnets	✓	Collaborating, protected host groups

To construct a scenario-specific invariant, a generic invariant is instantiated with scenario-specific knowledge. This is done by specifying host attributes [7]. These invariants and the list of entities (*INET*, *WebApp*, *WebFrnt*, *DB*, *Log*) is the only input needed. For this scenario, the following four invariants are expressed, formalized in Fig. 2.

1) First, as illustrated in Fig. 1, *DB*, *Log* and *WebApp* are labeled as internal hosts. The *WebFrnt* must be accessible from outside and is thus labeled as DMZ member. This is captured in the *Subnets* invariant.
2) Next, it is expressed that the logging data must not leave the log server. Therefore, using the *Sink* invariant, *Log* is classified as information sink.
3) Using the *Bell LaPadula* invariant, it is specified that *DB* contains confidential information. Since it sends its log data to the log server, this log server is also assigned the confidential security clearance. Finally, the *WebApp* is allowed to retrieve data from the *DB* and to publish it to the *WebFrnt*. Therefore, the *WebApp* is trusted and allowed to declassify the data.
4) Finally, an access control list specifies that only *WebApp* may access the *DB*.

Subnets $\{\mathrm{DB} \mapsto \mathit{internal},\ \mathrm{Log} \mapsto \mathit{internal},\ \mathrm{WebApp} \mapsto \mathit{internal},\ \mathrm{WebFrnt} \mapsto \mathit{DMZ}\}$

Sink $\{\mathrm{Log} \mapsto \mathit{Sink}\}$

Bell LaPadula $\{\mathrm{DB} \mapsto \mathit{confidential},\ \mathrm{Log} \mapsto \mathit{confidential},\ \mathrm{WebApp} \mapsto \mathit{declassify}\ (\mathit{trusted})\}$

Comm. Partners $\{\mathrm{DB} \mapsto \mathit{Access\ allowed\ by} : \mathrm{WebApp}\}$

Figure 2. Security Invariants (Case Study)

In this example, several hosts do not have attributes assigned for all invariants. It is sufficient to supply an incomplete host attribute specification, since they are automatically and securely completed by *topoS*. Previous work [7] discusses the details.[2] Once the invariants are specified, their management scales well in the face of changes: When a new host is added to the network, issues are handled by the auto-completion: either, the new host causes a violation, which is consequently uncovered, or it can be added without any further changes. Invariants are composable and modular by design, helping structured representation and archiving of knowledge. In the worst case, inconsistent security invariants may be specified accidentally. This only results in an overly strict security policy being computed, which can be identified in the following step.

It has been shown that a special class of invariants, called Φ-structured, exhibits several nice mathematical properties [7] (c.f. Tab. I). A Φ-structured invariant asserts a predicate for every policy rule. This predicate must only depend on the sender, receiver, and their host attributes. In particular, these invariants and their derived algorithms are very efficiently computable. It is also due to the Φ-structured invariants that a maximum-permissive security policy is uniquely defined.

[2]The security of the auto-completion is guaranteed w.r.t. the provided information, i.e. the auto-completion can never lead to an unnoticed security problem, given enough information is provided. For example, information-leakage is always uncovered, given all confidential data sources are specified. However, if an administrator forgets to label a confidential data source, information leakage can occur. It is trivially possible to design explicit whitelisting invariants which auto-complete to some '*deny*' property. On the downside, this requires lots of manual configuration effort, which is avoided by the invariants utilized in this paper. Roughly speaking the auto-completion fulfills: "*the more information provided, the more secure the whole system*".

B. Constructing the Security Policy

A network's end-to-end connectivity structure, i.e. a global access control matrix, corresponds to the *security policy*. Here, we utilize the textbook definition that a policy consists of the *rules* which ensure that the network is in a secure state. In contrast, the security goals are expressed as *invariants* over the policy and reside on an higher abstraction level.

Graphically, a policy can be illustrated as a directed graph. The case study's policy, illustrated in Fig. 3, was automatically computed from the security invariants.

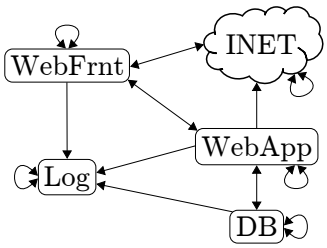

Figure 3. Security Policy

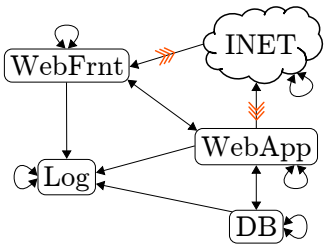

Figure 4. Stateful Policy

The algorithm to transform a set of security invariants into a policy starts with the allow-all policy and iteratively removes undesired rules. This is always possible if (and only if [7], Theorem 1) the invariants hold for the deny-all policy; a static requirement which is only to be proven once for a generic invariant. The algorithm is sound. It is also complete for the invariants utilized in this example (and for Φ-structured invariants in general [8]).

In our example, the administrator decides to manually refine the policy: there is no need for the web frontend to connect to the Internet. Therefore, this flow is prohibited. After this manual refinement, the security invariants are re-verified.

C. Constructing the Stateful Policy

The derived policy may appear adequate from a theoretical point of view but has one major problem when it comes to implementation: The *WebApp* can connect to the Internet, but the policy does not specify whether the Internet may answer this request (same for the *WebFrnt* after manual refinement). Obviously, for this scenario, answers should be permitted; otherwise, no one would be able to use the service. In contrast, the Log server uses the `syslog` protocol over UDP (RFC 5426). This protocol uses a unidirectional UDP channel and it is explicitly specified for security reasons that this is the only way the communication with the log server is permitted.

Therefore, it must be distinguished between stateful and purely unidirectional rules. We extend the security policy to additionally specify whether a flow might be stateful (i.e. answers to requests are allowed). Note that a flow with the stateful attribute might allow packets in the opposite direction of the policy rule and thus potentially violate security invariants. Defining the following two consistency criteria, the stateful attributes can be computed automatically [9]:

1) No information flow violation must occur
2) No access control side effects must be introduced

To compute the stateful policy, not only a single rule but a set S is to be upgraded to stateful rules. However, the interaction of the rules and answer paths of S must not introduce negative implications. Therefore, in particular to verify lack of side effects, all security policies derived from upgrading all *subsets* of S must be verified. A naive approach would require exponential complexity. We proved that this can be done more efficiently, particularly in linear time for Φ-structured invariants [9]. This insight provides an algorithm for computing the stateful policy from the security policy and the invariants. It is proven sound [9, Theorem 2] and complete w.r.t. the two criteria individually [9]. Multiple solutions for a stateful policy may exist; a user may set preferences.

For the case study, this results in a policy where the Internet can set up connections to the web frontend, likewise, the web backend can set up connections to the Internet. However, the logging channels are purely unidirectional UDP (stateful connections would introduce an information flow violation). We will call this the stateful policy. It is illustrated in Fig. 4.

D. Serializing Security Device Configurations

Till now, the network of Fig. 1 was considered a black box. In this section, the stateful policy is serialized to configurations for real network security devices. Though the serialization step is merely syntactic rewriting of the stateful policy, care must be taken to correctly transfer the semantics. *topoS* must fulfill the following three assumptions.

Structure The enforced network connectivity structure must exactly coincide with the policy. This requires that the links are confidential and integrity protected.

Authenticity The policy's entities must match their network representation (e.g. IP/MAC addresses). In particular, no impersonation or spoofing must be possible.

State The stateful connection handling must match the stateful policy's semantics.

One policy entity may correspond to several entities in the network. For example, deployed with load-balancing, *WebApp* corresponds a set of backend servers. In such cases, special care must be taken for reflexive policy rules. For the sake of brevity, we only present a one-to-one mapping between policy entities and their network representatives in this paper.

We present two possibilities to implement the policy.

1) Firewall & Central VPN Server: All entities connect to a central OpenVPN server which enforces the policy. Entities are bound to their policy name with X.509 certificates. Every entity sets up a layer 3 (`tun`) VPN connection with the server. The server authenticates entities by their certificate and centrally assigns IP addresses. IP spoofing over the tunnel is prevented. This provides authenticity (✓). Firewalling is applied at the server; the stateful policy is directly translated to `iptables` rules, shown in Fig. 5. With this, the stateful semantics (✓) and structure (✓) are enforced.

2) SDN: With complete control over the network, as is the case with data centers, a Software-Defined Network (SDN) may be used to implement the policy. Usually, a data center is a flat layer 2 network [11] and we need to contain layer 2 broadcasting and attacks. For this, an entity's switch port must be known. We install OpenFlow rules which prevent MAC,

```
FORWARD DROP
-A FORWARD -i tun0 -s $WebFrnt_ipv4 -o tun0 -d $Log_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebFrnt_ipv4 -o tun0 -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $DB_ipv4 -o tun0 -d $Log_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $DB_ipv4 -o tun0 -d $WebApp_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o tun0 -d $WebFrnt_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o tun0 -d $DB_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o tun0 -d $Log_ipv4 -j ACCEPT
-A FORWARD -i tun0 -s $WebApp_ipv4 -o eth0 -d $INET_ipv4 -j ACCEPT
-A FORWARD -i eth0 -s $INET_ipv4 -o tun0 -d $WebFrnt_ipv4 -j ACCEPT
-I FORWARD -m state --state ESTABLISHED -i eth0 -s $INET_ipv4 -o tun0 -d $WebApp_ipv4 -j ACCEPT
-I FORWARD -m state --state ESTABLISHED -i tun0 -s $WebFrnt_ipv4 -o eth0 -d $INET_ipv4 -j ACCEPT
```

Figure 5. VPN Server Firewall Rules (can be loaded with `iptables`)

```
# ARP Request
in_port=$port_src dl_src=$mac_src dl_dst=ff:ff:ff:ff:ff:ff arp arp_sha=$mac_src ↩
    arp_spa=$ip4_src arp_tpa=$ip4_dst priority=40000 action=mod_dl_dst:$mac_dst,output:$port_dst

# ARP Reply
dl_src=$mac_dst dl_dst=$mac_src arp arp_sha=$mac_dst arp_spa=$ip4_dst arp_tpa=$ip4_src ↩
    priority=40000 action=output:$port_src

# IPv4 one-way
in_port=$port_src dl_src=$mac_src ip nw_src=$ip4_src nw_dst=$ip4_dst priority=40000 ↩
    action=mod_dl_dst:$mac_dst,output:$port_dst

# if src (res. dst) is INET, replace $ip4_src (resp. $ip4_dst) with ⋆ and decrease the priority
```

Figure 6. OpenFlow Flow Table Template (can be loaded with `ovs-vsctl set-fail-mode $switch secure && ovs-ofctl add-flows`)

IP, and ARP spoofing. Figure 6 illustrates a template for generating a stateless rule from *src* to *dst*. The first rule allows ARP requests. Note that we rewrite the layer 2 broadcast addresses directly to the immediate receiver's address. Rule two allows the ARP responses. Both rules ensure that only valid ARP queries and responses are sent and received in the network.[3] The third rule allows IPv4 traffic. For stateful rules, the opposite direction of Fig. 6, i.e. *src* and *dst* swapped, is added. Any unmatched packets are dropped. With this set of rules, a mapping of policy identifiers to MAC and IP addresses is enforced. Also, correct address resolution is enforced (authenticity ✓). Without the ARP information leak, the desired connectivity structure (✓) is enforced. The setup does not provide stateful handling (✗) by default. However, a network firewall or SDN firewall app can provide the desired state (✓) handling.

IV. Discussion

A. Limitations

The process supported by *topoS* currently has two main limitations. First, it is completely static. For example, the mapping of policy entity names to their network representatives is done statically and manually. In general, *naming* is a complex (but orthogonal) issue. This information should usually be managed by a resource and account management system or directory service. Second, only one security device as backend is currently supported. However, related work suggests that this gap can be easily bridged, e.g. by translating to a one big switch abstraction [12], [13]. Many networks additionally employ a variety of heterogeneous, vendor-specific middleboxes. In future work, it might be worth investigating to which extent additional low-level device features (e.g. DHCP, IPv6, timeouts for stateful rules, ...) should be configurable on each abstraction layer.

[3]For the sake of simplicity, this implementation is designed such that it gets along without an SDN controller. This introduces a small hidden information flow channel (structure ✗): the ARP responses. For example in Fig. 4, *Log* may use a timing channel or the ARP `OPER` field to exfiltrate information. However, the side-channel is easily removed when an SDN controller answers all ARP requests (structure ✓); all necessary information is present.

B. Advantages

The presented process provides three novel advantages. First, it *bridges several abstraction levels* in a uniform way. The intermediate results are well-specified, which allows manual intervention, visualization, and adding features. Second, the theoretical background is *completely formally verified*. Thus, *topoS* is more than an academic prototype but a highly trustworthy tool. In addition, *topoS*'s library can be reused, extended, exported to several languages, and adapted to fit the needs of other frameworks. Finally, with the formal background, *topoS* is a first step towards high-assurance certification.

Third, *topoS* can scale to large networks w.r.t. theory (*i*), computational complexity (*ii*), and management complexity (*iii*). The theoretical foundation (*i*) scales to arbitrary networks. The computational complexity (*ii*) depends on the type of security invariants. New invariants with arbitrary computational complexity can be developed for *topoS*. However, we found that usually only Φ-structured invariants are needed, which implies the following computational complexity: $\mathrm{O}(|invariants| \cdot |entities|^2)$. An evaluation of *topoS*'s most expensive step has been presented previously [7]. Finally, (*iii*) the complexity of managing an invariant (with exception for the ACL invariants) is linear in the number entities. Due to the auto-completion, it is actually better than linear. Thus, the management complexity of *topoS* is roughly linear in the number of invariants and entities.

V. Related Work

To discuss related work, we first define four management abstraction layers to subsequently classify related work.

Security Invariants Defines the high-level security goals. Representable as predicates. For example, Fig. 2.

Access Control Abstraction Defines the allowed accesses between policy entities. Representable as access control matrix. For example, Fig. 3.

Interface Abstraction Defines a model of the complete network topology. Representable as a graph, packets are forwarded between the entity's interfaces.

Box Semantics Describes the semantics (i.e. behavior) of individual network boxes. Usually, the semantics are vendor-specific (e.g. iptables, Cisco ACLs, Snort IDS, ...).

In Fig. 7, we summarize how related work bridges the abstraction layers. Related work may bridge these layers vertically or work horizontally on artifacts at one layer. A direct arrow from the Access Control Abstraction to the Box Semantics (and vice versa) means that the solution only applies to a single enforcement box. Solutions such as Firmato and Fireman achieve more and are thus listed multiple times.

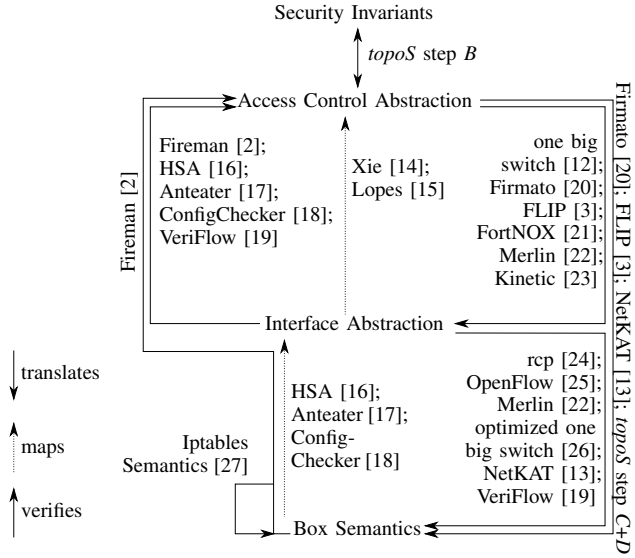

Figure 7. Four Layer Abstraction in Related Work

Firmato [20] is the work closest related to *topoS*. It defines an entity relationship model to structure network management and compile firewall rules from it, illustrated in Fig. 8. Firmato focuses on roles, which correspond to policy entities in our model. A role has positive capabilities and is related to other roles, which can be used to derive an access control matrix. Zones, Gateway-Interfaces and Gateways define the network topology, which corresponds to the interface abstraction. As illustrated in Fig. 8, the abstraction layers identified in this work can also be identified in Firmato's model. The Host Groups, Role Groups and Hosts definitions provide a mapping from policy entities to network entities, which is Firmato's approach to the naming problem. Similar to Firmato (with more support for negative capabilities) is FLIP [3], which is a high-level language with focus on *service* management (e.g. allow/deny HTTP). Essentially, both FLIP and Firmato enhance the Access Control Matrix horizontally by including layer four port management and traverse it vertically by serializing to firewall rules.

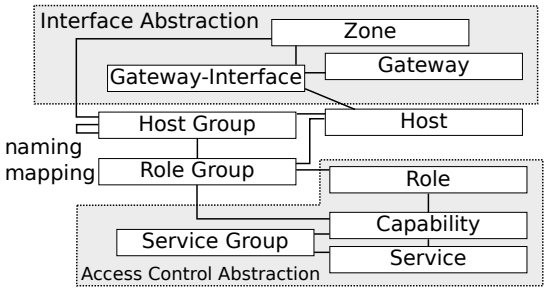

Figure 8. Firmato ERM

As illustrated in Fig. 7, Fireman [2] is a counterpart to Firmato. It verifies firewall rules against a global access policy. In addition, Fireman provides verification on the same horizontal layer (i.e. finding shadowed rules or inter-firewall conflicts, which do not affect the resulting end-to-end connectivity but are still most likely an implementation error). Abstracting to its uses, one may call rcc [28] the fireman for BGP.

Header Space Analysis (HSA) [16], Anteater [17], and ConfigChecker [18] verify several horizontal safety properties on the interface abstraction, such as absence of forwarding loops. By analyzing reachability [14]–[18], horizontal consistency of the interface abstraction with an access control matrix can also be verified. Verification of incremental changes to the interface abstraction can be done in real-time with VeriFlow [19], which can also prevent installation of violating rules. These models of the interface abstraction have many commonalities: The network boxes in all models are stateless and the network topology is a graph, connecting the entity's interfaces. A function models packet traversal at a network box. These models could be considered as a giant (extended) finite state machine (FSM), where the state of a packet is an (interface$\times$packet) pair and the network topology and forwarding function represent the state transition function [15], [29]. Anteater [17] differs in that interface information is implicit and packet modification is represented by relations over packet histories.

Most analysis tools make simplifying assumptions about the underlying network boxes. Diekmann et al. [27] present simplification of iptables firewalls to make complex real-world firewalls available for tools with simplifying assumptions.

NetKAT [13] is a SDN programming language with well-defined semantics. It features an efficient compiler for local, global, and virtual programs to flow table entries.

Craven et al. [30] present a generalized (not network-specific) process to translate access control policies, enhanced with several aspects, to enforceable device-specific policies; the implementation requires a model repository of box semantics and their interplay. Pahl delivers a data-centric, network-specific approach for managing and implementing such a repository, further focusing on things [31].

FortNOX [21] horizontally enhances the access control abstraction as it assures that rules by security apps are not overwritten by other apps. Technically, it hooks up at the access control/interface abstraction translation. Kinetic [23] is an SDN

language which lifts static policies (as constructed by *topoS*) to dynamic policies. To accomplish this, an administrator can define a simple FSM which dynamically (triggered by network events) switches between static policies. In addition, the FSM can be verified with a model checker. Features are horizontally added to the interface abstraction: a routing policy allows specifying *paths* of network traffic [26]. Merlin [22] additionally supports bandwidth assignments and network function chaining. Both translate from a global policy to local enforcement and Merlin provides a feature-rich language for interface abstraction policies.

VI. Conclusion

We presented *topoS*, a fully verified tool to manage network-level access control. It was demonstrated by example; nevertheless, the correctness proofs are universally valid and *topoS* is applicable to any larger network. The example demonstrates that a traditional network segmentation into *internal* and *DMZ* cannot cope with complex security goals and the traditional thought model of structure by IP ranges is no longer appropriate. In contrast, *topoS* only requires the high-level security goals and can automatically translate them to low-level configurations, such as firewall rules or SDN flow table entries. During the translation, all intermediate results are well-defined, accessible, and can be visualized. This provides feedback and allows manual refinement of them, including manual optimizations on lower abstraction layers. After manual refinement, re-verification is run to avoid human error. For the first time, the complete, automated, and verified transition from high-level security goals to both Firewall and SDN configurations was presented.

Availability & Acknowledgements

Our tool *topoS* and the correctness proofs can be obtained at

https://github.com/diekmann/topoS/ or [8]

The formalization of the case study is Distributed_WebApp.thy
It runs live at: http://otoro.net.in.tum.de/goals2config/

This work has been supported by the German Federal Ministry of Education, EUREKA project SASER, grant 16BP12304, and project SURF, grant 16KIS0145, and by the European Commission, project SafeCloud, grant 653884.

References

[1] F. Mansmann, T. Göbel, and W. Cheswick, "Visual analysis of complex firewall configurations," in *VizSec*. ACM, 2012, pp. 1–8.

[2] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "FIREMAN: a toolkit for firewall modeling and analysis," in *S&P*. IEEE, May 2006, pp. 199–213.

[3] B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher, "Specifications of a high-level conflict-free firewall policy language for multi-domain networks," in *SACMAT*. ACM, 2007, pp. 185–194.

[4] A. Wool, "A quantitative study of firewall configuration errors," *Computer, IEEE*, vol. 37, no. 6, pp. 62 – 67, 6 2004.

[5] H. Hamed and E. Al-Shaer, "Taxonomy of conflicts in network security policies," *IEEE ComMag*, vol. 44, no. 3, pp. 134 – 141, Mar. 2006.

[6] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *SIGCOMM*, pp. 13–24, 2012.

[7] C. Diekmann, S.-A. Posselt, H. Niedermayer, H. Kinkelin, O. Hanka, and G. Carle, "Verifying security policies using host attributes," in *FORTE*. Springer, Jun. 2014, pp. 133–148.

[8] C. Diekmann, "Network security policy verification," *Archive of Formal Proofs*, Jul. 2014, http://afp.sf.net/entries/Network_Security_Policy_Verification.shtml.

[9] C. Diekmann, L. Hupel, and G. Carle, "Directed security policies: A stateful network implementation," in *ESSS*, May 2014, pp. 20–34.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, last updated 2015, vol. 2283.

[11] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[12] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks." in *NSDI*. USENIX, 2013, pp. 1–13.

[13] S. Smolka, S. A. Eliopoulos, N. Foster, and A. Guha, "A fast compiler for netkat," in *ICFP*, Sep. 2015, pp. 328–341.

[14] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford, "On static reachability analysis of IP networks," in *INFOCOM*, vol. 3. IEEE, 2005, pp. 2170–2183.

[15] N. P. Lopes, N. Bjørner, P. Godefroid, and G. Varghese, "Network verification in the light of program verification," Tech. Rep., Sep. 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=201589

[16] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: static checking for networks," in *NSDI*. USENIX, 2012.

[17] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*, 2011, pp. 290–301.

[18] 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*, Oct 2009, pp. 123–132.

[19] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *NSDI*. USENIX, 2013, pp. 15–27.

[20] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *S&P*. IEEE, 1999, pp. 17–31.

[21] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *HotSDN*. ACM, 2012, pp. 121–126.

[22] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," *CoRR*, vol. abs/1407.1199, 2014.

[23] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *NSDI*. Oakland, CA: USENIX, 2015, pp. 59–72.

[24] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," in *NSDI*. USENIX, 2005, pp. 15–28.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[26] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *CoNEXT*. ACM, 2013, pp. 13–24.

[27] C. Diekmann, L. Hupel, and G. Carle, "Semantics-preserving simplification of real-world firewall rule sets," in *Formal Methods*, Jun. 2015.

[28] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *NSDI*. USENIX, 2005, pp. 43–56.

[29] S. Zhang, S. Malik, and R. McGeer, "Verification of computer switching networks: An overview," in *Automated Technology for Verification and Analysis*, ser. LNCS. Springer, 2012, pp. 1–16.

[30] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Policy refinement: Decomposition and operationalization for dynamic domains," in *CNSM*, Oct. 2011, pp. 1–9.

[31] M.-O. Pahl, "Data-centric service-oriented management of things," in *IM*, Ottawa, Canada, May 2015.