The Mathematical Foundations for Mapping Policies to Network Devices

Dinesha Ranathunga[1], Matthew Roughan[1], Phil Kernick[2] and Nick Falkner[3]

[1]*ARC Centre of Excellence for Mathematical and Statistical Frontiers,*
School of Mathematical Sciences, University of Adelaide, Adelaide, Australia
[2]*CQR Consulting, Unley, Australia*
[3]*School of Computer Science, University of Adelaide, Adelaide, Australia*

Keywords: Network-security, Zone-Conduit Model, Security Policy, Policy Graph.

Abstract: A common requirement in policy specification languages is the ability to map policies to the underlying network devices. Doing so, in a provably correct way, is important in a security policy context, so administrators can be confident of the level of protection provided by the policies for their networks. Existing policy languages allow policy composition but lack formal semantics to allocate policy to network devices.

Our research tackles this from first principles: we ask how network policies can be described at a high-level, independent of vendor and network minutiae. We identify the algebraic requirements of the policy-mapping process and propose semantic foundations to formally verify if a policy is implemented by the correct set of policy-arbiters. We show the value of our proposed algebras in maintaining concise network-device configurations by applying them to real-world networks.

1 INTRODUCTION

Managing modern-day networks is complex, tedious and error-prone. These networks are comprised of a wide variety of devices, from switches and routers to middle-boxes like firewalls, intrusion detection systems and load balancers. Configuring these heterogeneous devices with their potentially complex policies manually, box-by-box is impractical.

A better approach is a top-down configuration where device configurations are derived from a high-level user specification. Such specifications raise the level of abstraction of network policies above device and vendor-oriented APIs (*e.g.,* Cisco CLI, OpenFlow). Doing so, provides *a single source of truth*, so, security administrators for instance, can easily determine who gets in and who doesn't (Howe, 1996).

In recent years, several research groups have proposed high-level policy specification languages for both Software Defined Networking (SDN) (Soulé et al., 2014; Reich et al., 2013; Foster et al., 2010; Prakash et al., 2015) and traditional networks (Bartal et al., 2004; Guttman and Herzog, 2005). A common requirement in these languages is the ability to map the abstract policies to the underlying physical network. Doing so accurately, is essential to

- enforce policy correctly; and
- track policy changes per network device, so, redundant policy updates can be avoided.

But, mapping policies to the physical network devices has challenges

- policy needs to be *decoupled* from the network. A policy shouldn't need to change with vendor, or every time IP address changes occur;
- the mapping must be *formally verifiable*. Precise and unambiguous mathematical semantics eliminate wishful thinking pitfalls in deploying policies to networks. So, security administrators have assurance, for instance, that their intended policies are enforced by the correct firewalls; and
- policies can have complex semantics including node and link properties.

We propose a generic framework to map policies to network devices algebraically. We illustrate it's use by considering security policies, because incorrect deployment of these policies in domains such as Supervisory Control and Data Acquisition (SCADA) networks can result in catastrophic outcomes including the loss of human lives! However, the principles involved, can easily be extended to policies involving traffic measurement, QoS, load balancing and so on.

In developing our algebras, we have derived the properties and constraints of sequentially- and

Ranathunga, D., Roughan, M., Kernick, P. and Falkner, N.
The Mathematical Foundations for Mapping Policies to Network Devices.
DOI: 10.5220/0005946201970206
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*, pages 197-206
ISBN: 978-989-758-196-0

parallelly-composed policies for various policy contexts. These policy-composition semantics must be preserved, when mapping policy to devices, to ensure correct deployment. Using an algebraic framework also makes the policy-mapping process efficient.

We will demonstrate the use of our proposed algebras in deploying security policies on real SCADA networks. Particularly, we will show it's value in maintaining a clear, concise set of firewall configurations. Our approach allows administrators to conduct "what if" analysis by changing policy and/or network topology and observe their effect on the network devices required to implement the changes.

2 BACKGROUND AND RELATED WORK

"The advantages of implicit definition over construction are roughly those of theft over honest toil."

Bertrand Russel

The quote is salient because network installations are commonly built from the *bottom-up*, *i.e.,* a network-device is purchased, and configurations written. The policy is the result of the configuration, which is the consequence of a purchasing decision. So, the policy is implicitly defined as a result of many small decisions that interact in complex ways. Instead, best-practice guides (*e.g.,* Byres et al., 2005) suggest designing the policy first, and only then determining how to implement it.

Solutions that employ a *top-down* network configuration have been proposed (*e.g.,* Soule et al., 2014; Anderson et al., 2014; Bartal et al., 2004). They allow management of a single network-wide policy (*i.e.,* source of truth). These policies should be high-level: *i.e.,* decoupled from network and vendor intricacies, to capture policy intent and not the implementation.

Capturing intent has several benefits: policy can be distinguished from network to assist with change management; accurate comparison of organisational policies to industry best-practices can be made to evaluate compliance; and policy semantics can be expressed without network minutiae like IP addresses. But, most research towards high-level policy languages (Bartal et al., 2004; Cisco Systems Inc., 2014; Soulé et al., 2014), still requires these minutiae to be specified in-policy, to implement policy on a network.

If the high-level policy definition is built on formal mathematical constructions, then there are no implicit properties and it provides a truly sound foundation for everything that follows. The formalism would allow construction of complex and flexible policies and support reasoning about the policies. For instance, we could precisely compare a defined policy with industry best-practices in (Byres et al., 2005) for compliance and reduce network vulnerability to cyber attacks (Ranathunga et al., 2016a).

It is equally important to map policy to devices using a formal approach. For instance, we can be confident of the blanket of protection provided for our network if and only if we could prove that an intended security policy is implemented by the correct set of network firewalls. Some top-down configuration languages (Bartal et al., 2004; Prakash et al., 2015; Cisco Systems Inc., 2014) allow creation of network-wide high-level policies, but lack means to allocate policy to network devices in provably correct way.

NetKAT (Anderson et al., 2014) is a SDN programming language for specifying and reasoning about network behaviour. An implementation has been developed for NetKAT to handle high-level policies based on virtual network topologies (Smolka et al., 2015). The implementation uses an extension of Binary Decision Diagrams to generate OpenFlow table entries from high-level policy. In another approach, the SOL framework (Heorhiadi et al., 2016) uses a path-based abstraction to capture optimisation requirements of SDN applications. The abstraction allows definition of valid paths via predicates (*e.g.,* those that include a firewall). In contrast to these works, our aim is to develop an algebraic framework to map high-level policy to network-devices in both SDN and traditional networks.

We have proposed (Ranathunga et al., 2016a) a high-level security policy specification that is network and vendor independent for top-down configuration of firewalls. The language semantics allow these policies to be easily understood by humans. We investigate here, the underlying requirements for mapping these policies to network devices.

3 ABSTRACT POLICIES

Abstractions are key to constructing high-level policies. A good policy abstraction should capture the underlying concepts naturally and concisely. For instance, a policy may be arbitrated using one or more network devices. A good abstraction should decouple *what* is arbitrated from *how* it is arbitrated.

A simple abstraction that is commonly used to decouple policy from the network is *(endpoint-group, edge)* (Bartal et al., 2004; Prakash et al., 2015). An *endpoint* groups elements with common physical or logical properties: *e.g.,* a subnet, a user-group, a collection of servers, *etc.* An *edge* specifies the relationship between the endpoints, *i.e.,* it describes what communication is allowed between the endpoints.

Consider the example network in Figure 1(a), our high-level policy might be "we want to measure all HTTPS flows from $S1$ to $S4$". This intent can be expressed using the (endpoint-group, edge) abstraction as `p:=S1` $\rightarrow$ `S4 : collect https`, where $p \in \mathcal{P}$ is an element of the possible space $\mathcal{P}$ of policies.

We now want to map the policy to enforcing devices (*i.e.,* arbiters). It looks naively easy. Just implement the policy on arbiter F. However, the problem is in general more complicated. For example, consider policy from $S1$ to $S2$. If arbiter A captures flows described by policy p_A and likewise p_B, then the combination is $p_{A \cap B}$ because the policy expresses that flows be collected but load balancing across the two paths could result in either being used by a given packet.

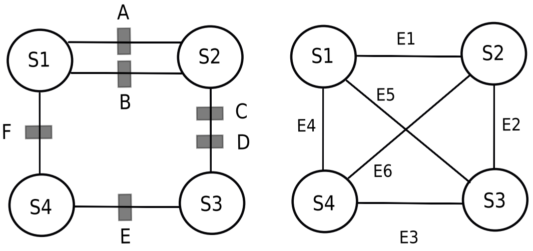
For another example, consider policy from $S2$ to $S3$. If arbiters C and D capture flows described by policy p_C and p_D respectively, then the combination would yield their *union* because both arbiters in the path are used for flow collection. So, when mapping policies to network devices, they must be composed in a way that preserve the topology-specific semantics.

We will, in general, denote the policy composition resulting from parallel routes as $p_A \oplus p_B$ and that resulting from serial routes as $p_A \otimes p_B$. In this example, $p_A \oplus p_B = p_{A \cap B}$ and $p_C \otimes p_D = p_{C \cup D}$. Note though, the meaning of $\oplus$, $\otimes$ are policy-context dependent.

The problem of mapping our high-level policy p (from $S1$ to $S4$) to network devices, may be even more complicated than simply implementing the policy on F. What happens if F fails? Surely we would still want to collect HTTPS flows that traverse the redundant paths from $S1$ and $S4$ (*e.g.,* path via $S2$ and $S3$). So, p must also be mapped to the policy arbiters along the path $S1 - S2 - S3 - S4$, to cater for this redundancy. But then, should we map the policy to all of these arbiters or a subset of them?

Given the parallel routes between $S1$ and $S2$ it is easy to see that we need to map p to both A and B to preserve the semantics of $\oplus$. But, when arbiters are in series (*e.g.,* C, D) we have the choice of mapping policy to both or just one to preserve the semantics of $\otimes$. Mapping to both may be unnecessary and inefficient and arbiters may have limits on their capabilities.

The policy arbiters may not always be in series or parallel. As we have outlined in our earlier work (Ranathunga et al., 2015), when a star-topology is involved, the question of how to map policy to topology gets interesting. Loosely mapping policy to arbiters using a *hyper-edge* adds to the complexity of the policy-graph and decreases it's precision. For instance, it allows a $1:n$ relationship between hyper-edges and policies, so, we must track the policy between every endpoint-pair in a hyper-edge to correctly map it's policy to network devices.



(a) Network topology. (b) Policy graph of (a).

Figure 1: Example network consisting of four endpoints ($S1 - S4$) and six policy arbiters $A, B, ..., F$ (a). The corresponding policy graph is shown in (b) with these endpoints and policy edges ($E1, E2, ..., E6$) between them.

We can overcome these issues using a simple-edge mapping instead (Ranathunga et al., 2015). The policy graph is then simplified and it's precision is increased. The $1:1$ relationship between edges and policies now implies we need to only track a single policy per edge, to map policy accurately to devices.

We may also need to consider paths that consist of one or more intermediate endpoints, when implementing policy between an endpoint pair. For instance, consider implementing a security policy f :=`S1` $\rightarrow$ `S3: http` in the network in Figure 1(a). Once implemented on the arbiters (*i.e.,* firewalls), the policy should allow HTTP traffic flow from `S1` to `S3`. But, the policy can only be deployed correctly if endpoints `S2,S4` can route or forward HTTP traffic.

This ability of an endpoint to route or forward traffic can be restrictive. For instance, in a security policy context, an endpoint can group systems (*e.g.,* hosts) with similar security requirements (ANSI/ISA-62443-1-1, 2007). So, high-risk systems can be grouped in to one endpoint and only *safe* traffic that originate and/or terminate at the endpoint is permitted by the security policy. Enabling this endpoint to transit-traffic could potentially expose the high-risk systems within to cyber attacks. So, an endpoint's *traffic transitivity* capability must be considered in constructing valid device-paths and must be captured explicitly in the mapping process.

The intent of our high-level policy p is to collect HTTPS flow data from $S1$ to $S4$. Similarly, traffic measurement policies can be defined between any pair of endpoints in the network, so the corresponding (endpoint-pair,edge) tuples collectively construct a *policy-graph* (Figure 1(b)). Each (logical) edge in this graph maps to a physical network path comprising of policy arbiters and zero or more intermediate endpoints. For instance, policy-edge $E4$ in Figure 1(b) implements our policy p.

So far, we have described several key requirements of a policy-to-device mapping process. Next, we illustrate these using more detailed examples involving policies found in practice.

3.1 Quality of Service (QoS) Policies

QoS policies can provide bandwidth guarantees for certain types of traffic. The policy arbiters here would be QoS capable routers or switches. Imagine we need to provision a minimum bandwidth of 100MB/s for HTTP traffic flow from $S1$ to $S4$ in Figure 1(a). The high-level policy intent can be expressed as `min(S1 → S4 : http, 100MB/s)`. Similarly, QoS policies between any endpoint-pair can be expressed using the policy-graph in Figure 1(b).

Parallel and serial (QoS-device) topologies also have an impact on the end QoS policy. Consider the parallel topology between $S1$ and $S2$ in Figure 1(a), if we assume p_A and p_B provide bandwidth guarantees of $B1$ and $B2$ respectively, then with load balancing, the resultant QoS policy (p_R) can provide a total bandwidth guarantee $p_R = p_A \oplus p_B = sum(B1, B2)$.

When the devices are in series, the resultant policy provides a bandwidth guarantee given by $p_C \otimes p_D = min(B3, B4)$ where $B3, B4$ are the bandwidth guarantees provided by p_C and p_D.

3.2 Security Policies

We consider here access-control policies in a network. The policy arbiters would be traffic filtering devices (*e.g.,* firewalls, SDN switches). The endpoints could be zones or user groups. Imagine we want to enable only SSH traffic from $S1$ to $S4$ (Figure 1(a)). The high-level policy can be expressed as `S1 → S4: ssh` with an implicit `deny-all` in place.

When the traffic filtering devices are in parallel (*e.g.,* topology between $S1$ and $S2$ in Figure 1(a)), the resultant security policy (p_R), has meaning — *all packets that can possibly be allowed through* — and is the union of the packet sets allowed by the individual devices. We take union conservatively because intrusions and attacks are usually carried out through permitted traffic. So, if p_A and p_B allows packet sets Q and T respectively, then $p_R = p_A \oplus p_B = p_{Q \cup T}$.

When the devices are in series, the resultant policy permits the *intersection* of the packet sets V, W allowed by the policies of C, D respectively, *i.e.,* $p_R = p_C \otimes p_D = p_{V \cap W}$.

With security policies, it is often useful to have endpoints that group systems with similar security requirements. Doing so, allows to define a single policy for all members of an endpoint: a clean and concise abstraction of network security. But, a generic (endpoint-pair, edge) abstraction cannot capture such network-security specific concepts precisely.

So, we need concrete definitions for what an endpoint and an edge means for each policy context to capture policy-specific intricacies. We will show later how to define these concretely for security policies.

3.3 Traffic Measurement Policies

Traffic measurement policies help implement, for instance, NetFlow (v9) on the network in Figure 1(a). The policy arbiters $A, B, ..., F$ here, would be NetFlow capable devices (usually routers or switches). The endpoints could be subnets or zones in the network.

When we considered the parallel routes between $S1$ and $S2$, the resultant policy (p_R) has meaning — *the flows that are guaranteed to be captured by the topology (without sampling)* — and constitutes of the intersection of the packet sets of A and B. So, if p_A and p_B capture packet sets Q and T respectively, then $p_R = p_A \oplus p_B = p_{Q \cap T}$.

We also showed when the devices are in series, the resultant policy captures the union of the packet sets, *i.e.,* $p_R = p_C \otimes p_D = p_{V \cup W}$.

Thus, policies can be composed using the generic semantics $\oplus$ and $\otimes$ in different policy contexts. The actual meaning of these operators is policy-type dependant. For instance, with QoS policies, $\otimes$ represented *minimum* while here it has meaning of *union*.

Policy Mapping Vs Routing: Our discussion above on mapping policies to network devices, relates to network paths or routes. This is no accident. Our target problem has many parallels with routing.

But, the aim in routing is to determine the path that optimises a given path-metric (*e.g.,* shortest-path routing finds a minimum-distance path between endpoints). Our target problem is different: we need to determine the arbiters in a network a given policy should be implemented on. So, constructing all feasible paths is crucial because, for instance, a security policy between two endpoints can only be correctly implemented (*e.g.,* to block all Telnet traffic) if all redundant paths between them are taken into account.

In routing, the number of endpoints (*e.g.,* individual gateways) can be high for a large network, so, distributed means to computing a solution is essential (requiring de-centralised protocols like OSPF and BGP). In contrast, we expect a relatively low number of endpoint groups when mapping policies to devices. As we will show in 5, a low endpoint-count makes our policy-mapping algorithm computationally tractable. So, a logically-centralised implementation can be considered.

Current meta-routing algebras (Dynerowicz and Griffin, 2013), allow to provably choose paths that optimise various path-metrics. These algebras support specification of link properties, but, not the specification of node properties such as traffic transitivity.

We described earlier, the need to define endpoints and edges concretely to incorporate the intricacies in each policy context. We illustrate the idea next using security policies and their concretised (endpoint-pair, edge) abstraction: *the Zone-Conduit model*.

The Zone-Conduit Model: Lack of internal-network segmentation is a key contributor to the fast propagation of threats and attacks in a network (Byres et al., 2005). So, ANSI/ISA have proposed the Zone-Conduit model as a way of segmenting and isolating the various sub-systems in a SCADA network (ANSI/ISA-62443-1-1, 2007).

A *zone* is a logical or physical grouping of an organisation's systems with similar security requirements so that a single policy can be defined for all zone members. A *conduit* provides the secure communication path between two zones, enforcing the policy between them (ANSI/ISA-62443-1-1, 2007). A conduit could consist of multiple links and firewalls but, logically, is a single connector.

The Zone-Conduit model is a concrete instance of the (endpoint-pair, edge) abstraction for high-level security policy specification. Zones and conduits define endpoints and edges concretely. These definitions allow important network-security characteristics such as a single zone-policy, to be captured concisely.

The ISA Zone-Conduit model in its original description lacks precision for policy specification. We use the extensions proposed in (Ranathunga et al., 2015) to increase its precision, *e.g.,* we add Firewall-Zones to specify firewall-management policies.

Zone-Conduit Policies: A conduit policy is an ordered set of rules $[p_1, p_2, ..., p_n]$ that act on packet sequences to accept, deny, or in some cases, modify them. In our related work (Ranathunga et al., 2016a), we have identified properties and constraints required in a Zone-Conduit based policy description to make the rules implementation- and order-independent.

To summarise, we adopt a security whitelisting model, *i.e.,* we restrict policies to express positive abilities[1] and deny all inter-zone flows that are not explicitly allowed (Ranathunga et al., 2015). Doing so, renders the rule order irrelevant and allows consistent conversion of policy to heterogeneous firewalls. Thus the underlying vendor can change without requiring policy alterations. By being explicit, we also prevent services being accidentally enabled implicitly.

Directed- vs Undirected-Conduit Policy: The policy on an undirected-conduit can be expressed using two *directed-conduit* policies. Directed-conduits are important in understanding how policy should be implemented on device interfaces. For instance, an undirected conduit can only map a security policy to a firewall interface. But, to implement the policy correctly, we additionally need to know if it should be implemented inbound or outbound on these interfaces. A directed-conduit provides this directionality.

However, analysis using directed-conduits can also lead to problems. For instance, directed-conduit paths that may seem feasible may require traffic to traverse a firewall interfaces twice (illustrated in detail in the longer version of the paper). We invalidate such directed-conduit paths via a mapping $h : F \to W$ where $F = \{directed\ firewalls\}$ and $W = \{firewalls\}$. A directed-firewall in the Zone-Conduit graph $G = (Z, C)$ is defined as

Definition 1 (Directed firewall)**.** *A* directed firewall *t_{ij} is a firewall $t \in W$ that filters traffic on directed-conduit $(i, j) \in C$.*
Then we can check if the directed-firewalls in a path map to the same physical firewall (*i.e.,* $h(A_{13}) = h(A_{31}) = A$) and deem that path invalid.

4 MAPPING ALGEBRA

We outline here, our proposed algebra to map policy to network devices by first making the distinction between *primary* and *secondary* policy-edges.

4.1 Firewall-path Construction

A policy-edge can be classified as primary or secondary based on how it arbitrates policy. A primary-edge enforces policy using arbiters (*e.g.,* firewalls) only. A secondary-edge enforces policy using arbiters and one or more endpoints (*e.g.,* a zone).

We demonstrate the idea using the simple Zone-Conduit graph $G = (Z, C)$ in Figure 2(a). The firewall composition of each primary-conduit in the model is shown (Figure 2(b)). The example secondary-conduit C_{13} filters traffic flow from zone `1` to `3`, using several directed-firewalls and transit zones `2`, `4`. The set of all directed-conduits are given by $DC = \{C_{ij} \mid (i, j) \in C\}$.

A policy-graph is essentially an automation that moves traffic packets from one endpoint to another using policy-edges. So, regular expressions (the natural language of finite automata), can capture the packet-processing behaviour of this model; a path encoding is a concatenation of directed-devices (*i.e.,* policy-arbitration steps *pq*) and a set of paths is encoded as

[1] Refers to the ability to initiate or accept a traffic service.

a union of paths. Past work (Anderson et al., 2014) has shown, all single-path encodings stem from the Kleene star operator (*) on the set of directed-devices.

In a security policy context, the automation means that a single firewall-path encoding is a concatenation of directed-firewalls. Each path depicts a sequence of traffic filtering steps and is an element of F^*.

But, the Zone-Conduit model is a logical representation, so, every firewall-path encoding in F^* may not be valid. We define single firewall-path concatenation to filter-out invalid paths

Definition 2 (Single firewall-path concatenation)**.** *Take* $a \in F^*$ *s.t.* $a = t_{d_1d_2}t_{d_2d_3}t_{d_3d_4}...t_{d_nd_{n+1}}$ *where* $t_{d_id_j} \in F$. *Also take* $b \in F^*$ *s.t.* $b = s_{g_1g_2}s_{g_2g_3}s_{g_3g_4}...s_{g_mg_{m+1}}$ *where* $s_{g_ig_j} \in F$. *Then firewall-path concatenation from* $F^* \times F^* \to F^*$ *is*

$$ab = \begin{cases} t_{d_1d_2}...t_{d_nd_{n+1}}s_{g_1g_2}...s_{g_mg_{m+1}} \text{ if } d_{n+1} = g_1 \\ \quad \text{and } g_i \neq d_j;\ \forall i,j,\ i > 1 \\ \quad \text{and } h(t_{d_id_j}) \neq h(s_{g_kg_l});\ \forall i,j,k,l \\ \phi,\ \text{otherwise.} \end{cases}$$

and $a\phi = \phi a = \phi;\ \forall a \in F^*$.

Concatenation defined above is a binary operation that constructs only elementary firewall-paths. These paths do not allow traffic to traverse a particular firewall interface more than once and also prohibit traffic flow through a non-transit zone (*e.g.,* Firewall-Zone).

Consider two directed-firewalls X,Y with policies p_X and p_Y that accept packet sets R and T. The resultant policy of the concatenated firewall-path XY is that of sequential firewalls and can be denoted as $(p_X \otimes p_Y)(s) = p_{R\cap T}(s)$ where s is a packet sequence. This result is policy-context dependent, for instance, with traffic measurement policies, the outcome is similar to that of serial arbiters: *i.e.,* $p_X \otimes p_Y = p_{R\cup T}$.

We define a set of directed-firewall paths as a union of elements in F^*. Again, consider our directed-firewalls X,Y from before. If these described two distinct paths, then the resultant policy of the path union $X \cup Y$ is that of parallel firewalls, *i.e.,* $(p_X \oplus p_Y)(s) = p_{R\cup T}(s)$ where s is a packet sequence. This result is also context dependent, so, for traffic measurement policies, the outcome is similar to that of parallel arbiters, *i.e.,* $p_X \oplus p_Y = p_{R\cap T}$.

We also extend concatenation in Definition 2 to $S = \{\text{Power-set of } F^*\}$

Definition 3 (Multiple firewall-path concatenation)**.** *Take* $a,b \in S$ *s.t.* $a = \{a_0,a_1...,a_x\}, b = \{b_0,b_1,...,b_y\}$ *where* $a_i,b_j \in F^*$. *Then firewall-path concatenation from* $S \times S \to S$ *is given by*

$$ab = \{a_ib_j\};\ \forall ij$$

Definition 3 allows to construct all possible firewall-path sets from the union of elements in S. Then $(S,\cup,\cdot,\hat{0},\hat{1})$ is an idempotent semiring with

$\hat{0} = \phi$; empty set; and

$\hat{1} = \{\varepsilon\}$; empty-string set where ε is the identity element of the concatenation operation.

The properties of the operators $\cup$ and $\cdot$ actually dictate rules for firewall-path construction. So, these semantics must be preserved when composing firewall-paths. For instance, $\cup$ is commutative while $\cdot$ is not. So, the order of the directed-firewalls matter, when constructing a single firewall-path, but, are irrelevant when constructing multiple paths.

Similarly, we can construct single and multiple device-paths for other policy contexts and obtain the semiring result for $(S,\cup,\cdot,\hat{0},\hat{1})$ per security policies.

4.2 Mapping Policy to Arbiters

We described how the semiring $(S,\cup,\cdot,\hat{0},\hat{1})$ constructs sets of device-paths between policy-graph end-points. The sequential (*i.e.,* $\otimes$) and parallel (*i.e.,* $\oplus$) policy-composition operators in 3 can now construct the policies of these paths. Assume we have to implement a high-level policy p_{ij} on arbiters q_{kl} that lie in the paths from i to j. All applicable device-paths from i to j can be constructed as per 4.1 and given by $S_{ij} = \{q_{a_1a_2}q_{a_2a_3}...q_{a_{n-1}a_n}, ..., q_{b_1b_2}q_{b_2b_3}...q_{b_{m-1}a_m}\}$. Then, the high-level policy p'_{ij} derived from the individual arbiter policies p'_{kl} is

$$\begin{aligned} p'_{ij} \quad = \quad & (p'_{a_1a_2} \otimes p'_{a_2a_3} ... \otimes p'_{a_{n-1}a_n}) \\ & \oplus (..................................) \\ & \oplus (p'_{b_1b_2} \otimes p'_{b_2b_3} ... \otimes p'_{b_{m-1}b_m}). \qquad (1) \end{aligned}$$

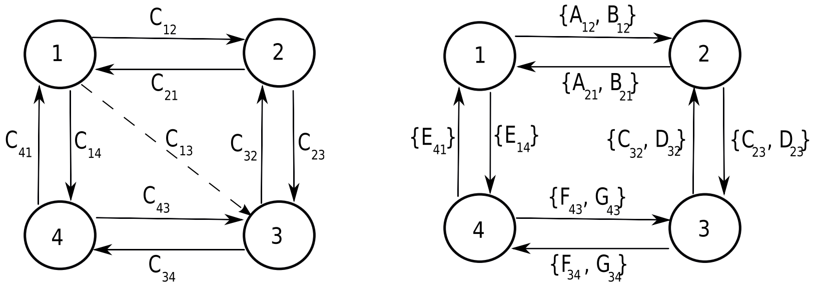
Mapping the intended high-level policy p_{ij} to the arbiters is now a matter of finding p'_{ij} for all arbiters such that $p'_{ij} = p_{ij}$. But deriving such a mapping is non trivial because $\oplus$ and $\otimes$ have policy-context dependent meanings. For instance, with security policies $\oplus$ means *union* and $\otimes$ means *intersection*. So, a simple solution that supports defence in depth would be to implement the access-control policy p_{ij} on every sequential firewall across all paths.

For another instance, $\oplus$ means *summation* and $\otimes$ means *minimum* in QoS policies. So, a required bandwidth guarantee p_{ij} can be split across multiple paths with sequential arbiters in each path guaranteeing only a portion of the total bandwidth.

Irrespective of the policy context, the underlying requirement when mapping policy to network devices is to adhere to the semantics of 1.

4.3 Computation of All Firewall-paths

We reduced the policy-to-device mapping problem to the semantics of 1 in the previous section. We now



(a) Zone-Conduit model with primary and secondary conduits.

(b) Firewall-paths of the primary-conduits in (a).

Figure 2: Zone-Conduit model depicting primary-conduits C_{ij} and a secondary-conduit C_{13} enabled by the transit zones 2 and 4 are shown in (a). The firewall-paths of the primary-conduits are shown in (b).

develop an algorithm to compute the device-paths of 1 efficiently. Again, we demonstrate the idea using security policies and the Zone-Conduit model.

We can represent the primary-conduit firewall-paths using a generalised Adjacency matrix A. Here, $A(i,j)$ is the firewall-path of primary conduit $C_{ij} \in DC$. For our example in Figure 2(b), $A =$

$$\begin{array}{cccc} Z_1 & Z_2 & Z_3 & Z_4 \\ \end{array}$$
$$\begin{bmatrix} \{\varepsilon\} & \{A_{12}, B_{12}\} & \phi & \{E_{14}\} \\ \{A_{21}, B_{21}\} & \{\varepsilon\} & \{C_{23}, D_{23}\} & \phi \\ \phi & \{C_{32}, D_{32}\} & \{\varepsilon\} & \{F_{34}, G_{34}\} \\ \{E_{41}\} & \phi & \{F_{43}, G_{43}\} & \{\varepsilon\} \end{bmatrix} \tag{2}$$

Given such a matrix A, the solution to the problem of finding the set of all valid firewall-paths between zones is a matrix A^* such that

$$A^*(i,j) \quad = \quad \{\text{valid primary- and secondary-conduit firewall-paths from zone i to j}\} \tag{3}$$

We developed the following theorem to compute A^*, inspired by algorithms in meta-routing (Dynerowicz and Griffin, 2013).

Theorem 4 (A^* calculation)**.** *A^* can be calculated using the right iteration algorithm*

$$A^{<k+1>} = (A^{<k>}T \cup I)A \text{ where } A^{<0>} = I.$$

T and I are the zone-transitivity matrix and the multiplicative-identity matrix (of semiring $(S, \cup, \cdot, \hat{0}, \hat{1})$) respectively.

Proof. See the longer version (Ranathunga et al., 2016b) of this paper. □

The zone-transitivity matrix T is defined as

$$T(i,j) = \begin{cases} \{\varepsilon\} \text{ if } i = j \text{ and transitivity}(i) = 1 \\ \phi, \text{ otherwise.} \end{cases} \tag{4}$$

and I is the multiplicative-identity matrix

$$I(i,j) = \begin{cases} \hat{1} \text{ if } i = j \\ \hat{0}, \text{ otherwise.} \end{cases} \tag{5}$$

For bounded semirings we only iterate $n-1$ times to converge to A^*, where n is the number of nodes in the Zone-Conduit model, *i.e.,* $A^* = A^{<n-1>}$.

We have defined T, A^* and the right-iteration algorithm for a security-policy context, but these can equally be defined for other policy contexts.

If we apply our algorithm in 4 to the example in Figure 2(b), $n = 4$, so, $A^* = A^{<3>}$ and for simplicity assume that all zones are transitive, then

$$T = \begin{bmatrix} \{\varepsilon\} & \phi & \phi & \phi \\ \phi & \{\varepsilon\} & \phi & \phi \\ \phi & \phi & \{\varepsilon\} & \phi \\ \phi & \phi & \phi & \{\varepsilon\} \end{bmatrix} \tag{6}$$

and we see that $I = T$ in this instance.

We calculate $A^* = A^{<3>} =$

$$\begin{bmatrix} \{\varepsilon\} & \eta & \kappa & \theta \\ \mu & \{\varepsilon\} & \nu & \beta \\ \gamma & \lambda & \{\varepsilon\} & \rho \\ \xi & \delta & \zeta & \{\varepsilon\} \end{bmatrix} \tag{7}$$

where for instance

$$\kappa \quad = \quad \{A_{12}C_{23}, A_{12}D_{23}, B_{12}C_{23}, B_{12}D_{23}, E_{14}F_{43} \\ E_{14}G_{43}\} \tag{8}$$

All valid firewall-paths from zone 1 to 3 are in κ.

Let's now assume that zone 4 is non-transitive, then $transitivity(4) = 0$, we re-calculate $A^* =$

$$\begin{bmatrix} \{\varepsilon\} & \{A_{12}, B_{12}\} & \eta & \theta \\ \{A_{21}, B_{21}\} & \{\varepsilon\} & \{C_{23}, D_{23}\} & \mu \\ \gamma & \{C_{32}, D_{32}\} & \{\varepsilon\} & \rho \\ \xi & \delta & \zeta & \{\varepsilon\} \end{bmatrix} \tag{9}$$

where for instance

$$\eta = \{A_{12}C_{23}, A_{12}D_{23}, B_{12}C_{23}, B_{12}D_{23}\} \tag{10}$$

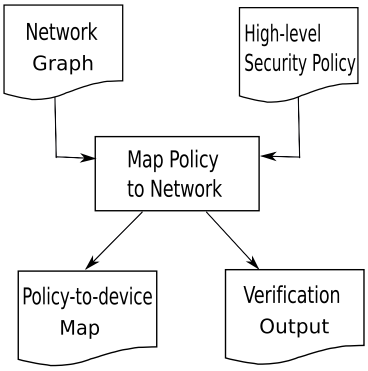


Figure 3: Policy to network-device mapping process.

The updated firewall-paths from 1 to 3 are in η. In comparison to 8, we see that the paths via zone 4 (*i.e.,* $E_{14}F_{43}, E_{14}G_{43}$) have now been removed as the zone is no longer transitive. A^* can similarly be calculated for other policy contexts to determine all valid device-paths between endpoint pairs.

We describe next our implementation of the algorithm in 4. The implementation allows the use of these algebras to map policy to real network devices.

5 IMPLEMENTATION

Our policy mapping system is depicted in 3, and we outline it's details below. The system is currently implemented in Python, and allows mapping of high-level policies written in our own policy specification language, to network devices. We will make the system open source in the near future.

High-level Security Policy: The topology-independent policy input file created using our high-level policy specification language.

Network Topology: The input network topology described in the XML-based graph file format *GraphML*. The file holds information of all devices in the network and their interconnections. The crucial aspects are the details of the topology near the policy arbiters (*e.g.,* firewalls).

Map Policy to Network: Compiles high-level policy to an Intermediate-Level (IL), generates the policy graph (*e.g.,* Zone-Conduit model) of the input network and computes A^* as per 4.3 to map high-level policy to the devices (*e.g.,* firewalls) in the network.

Policy-to-device Map: The primary output depicting policy breakdown by device, interface and direction.

Verification Output: Secondary output specifically suited for verification (*e.g.,* policy errors).

Our system implements a high-level policy on a network instance by coupling the two. We outline here, the computation of A^* to map policy to network devices (see the longer paper version for a detailed implementation). We illustrate our system using security policies and the Zone-Conduit model, but it can likewise be used in other policy contexts.

5.1 A^* Calculation

We compute A^* using the algorithm in 4. A centralised implementation is used (given typically low n), but, it could be distributed across multiple nodes performing parallel computations.

Our implementation algorithm has time complexity $O(n^4)$ (Ranathunga et al., 2016b). So, specifying policy per individual host (*i.e.,* large n) in top-down configuration makes policy mapping extremely inefficient. More efficient is to create network groups (*e.g.,* by subnet) and specify policy between them, so, a reasonably low value can be maintained for n. For instance, we will see in 6 that in a SCADA network typically $n < 25$.

We considered all valid paths between zone-pairs in calculating A^*. The decision allows us to permit or block traffic along all possible communication paths between a zone pair, providing redundancy and defence-in-depth in the network. We also map policy uniformly to every firewall in a single-path, further boosting defence-in-depth. These decisions collectively create a robust defence against cyber attacks.

But, in other policy contexts, it may be useful to select a subset of all valid paths or just a single path (*e.g.,* shortest path) instead. Doing so, could improve the time complexity of the algorithm in 4 (*e.g.,* shortest paths yield $O(n^3)$). This path pruning can be done, for instance, by incorporating a sparse matrix in the algorithm. In some contexts, it can be efficient to apply policy only on edge arbiters of a path (*e.g.,* when enabling traffic measurements).

6 A SERIES OF CASE STUDIES

We now show the use of our developed algebras, through real SCADA-firewall configuration case studies summarised in 1. The data was provided by the authors of (Ranathunga et al., 2015).

The seven Systems Under Consideration (SUCs), involve various firewall architectures and models. We use them to demonstrate several properties, most notably that the computational complexity of our policy-to-device mapping algorithm is tractable.

An important feature depicted in 1 is the number of security zones in each network. This number is

Table 1: SCADA case study summary adapted from (Ranathunga et al., 2015) (# ACL rule allocation error).

SUC	Fire-walls	Zones	Cond-uits	Max. hosts	ACLs	Average rules per ACL	Incorrect firewall#	Incorrect interface#	Incorrect direction#	Run-time (s)
1	3	7	11	67580	8	237	15	13	19	40
2	6	21	81	2794	12	16	3	2	5	70
3	4	10	17	886	8	6	2	1	4	43
4	3	9	16	2038	3	80	5	12	13	61
5	3	12	19	2664	12	677	15	8	26	47
6	3	13	21	3562	8	1034	21	15	19	63
7	6	15	22	3810	17	724	9	5	17	49

small (*i.e.,* ≤ 21) relative to the maximum (potential) number of hosts per network (*i.e.,* ≤ 67580).

This is to be expected, a zone groups a set of hosts or subnets with identical policies. If every host had a distinct policy then a large number of firewalls would be needed to enforce a real separation between the hosts, making it impractical. By grouping hosts into zones, we reduce policy complexity, so their specification becomes easier and less error-prone.

We identified incorrectly assigned ACL rules in each case study by parsing the firewall configurations as per (Ranathunga et al., 2015). We then classed the errors into three groups: *incorrect-firewall, incorrect-interface and incorrect-direction* errors (1). Incorrect-firewall errors are ACL rules that are assigned to the wrong firewall to begin with, *i.e.,* the desired traffic filtering could not be achieved by placing the ACL rule in any of that firewall's interfaces. Incorrect-interface errors are ACL rules that are assigned to the correct firewall but to the wrong firewall-interface, *i.e.,* the desired traffic filtering could not be achieved by assigning the rule inbound or outbound of that firewall-interface. Incorrect-direction errors comprise of ACL rules that are assigned to the correct firewall and firewall-interface, but in the wrong direction (*e.g.,* outbound instead of inbound).

As (1) suggests, on average there were 10 ACL rules allocated to the wrong firewall, 8 rules allocated to the wrong firewall-interface and 15 rules allocated in the wrong interface-direction, per case study. We automatically mapped the high-level policy in each case to it's network using our system. There were zero incorrectly allocated policy rules, when the policy was mapped to the firewalls using our algebras! Through correct policy deployment, we reduce vulnerability of these SCADA networks to cyber attack, preventing potentially-catastrophic outcomes.

7 CONCLUSIONS

There are various obstacles that hinder the precise mapping of policies to network devices. Most prominent is the lack of decoupling between policy and network which makes policy sensitive to network-intricacies and vendor changes. To compound the problem, policies can also have complex semantics including node and link properties.

Our research addresses these challenges and proposes a mathematical foundation for mapping policies to network-devices. We use it to deploy real-world security policies to network firewalls provably correctly, so, that administrators can be confident of the protection provided by their policies for their networks.

ACKNOWLEDGEMENTS

This project was supported by an Australian Postgraduate Award, Australian Research Council Linkage Grant LP100200493 and CQR Consulting.

REFERENCES

Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. (2014). NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126.

ANSI/ISA-62443-1-1 (2007). Security for industrial automation and control systems part 1-1: Terminology, concepts, and models.

Bartal, Y., Mayer, A., Nissim, K., and Wool, A. (2004). Firmato: A novel firewall management toolkit. *ACM TOCS*, 22(4):381–420.

Byres, E., Karsch, J., and Carter, J. (2005). Good practice guide on firewall deployment for SCADA and process control networks. *NISCC*.

Cisco Systems Inc. (2014). *Cisco Virtual Security Gateway for Nexus 1000V Series Switch Configuration Guide*. San Jose, CA 95134-1706, USA.

Dynerowicz, S. and Griffin, T. G. (2013). On the forwarding paths produced by Internet routing algorithms. In *ICNP*, pages 1–10.

Foster, N., Freedman, M. J., Harrison, R., Rexford, J., Meola, M. L., and Walker, D. (2010). Frenetic: a high-level language for OpenFlow networks. In *ACM PRESTO*, pages 21–27.

Guttman, J. D. and Herzog, A. L. (2005). Rigorous automated network security management. *IJIS*, 4:29–48.

Heorhiadi, V., Reiter, M. K., and Sekar, V. (2016). Simplifying software-defined network optimization using SOL. In *USENIX NSDI*, pages 223–237.

Howe, C. D. (1996). *What's Beyond Firewalls?* Forrester Research, Incorporated.

Prakash, C., Lee, J., Turner, Y., Kang, J.-M., Akella, A., Banerjee, S., Clark, C., Ma, Y., Sharma, P., and Zhang, Y. (2015). PGA: Using graphs to express and automatically reconcile network policies. In *ACM SIGCOMM*, pages 29–42.

Ranathunga, D., Roughan, M., Kernick, P., and Falkner, N. (2016a). Malachite: Firewall policy comparison. In *IEEE ISCC*.

Ranathunga, D., Roughan, M., Kernick, P., and Falkner, N. (2016b). The mathematical foundations for mapping policies to network devices, http://arxiv.org/abs/1605.09115. *Technical Report*.

Ranathunga, D., Roughan, M., Kernick, P., Falkner, N., and Nguyen, H. (2015). Identifying the missing aspects of the ANSI/ISA best practices for security policy. In *ACM CPSS*, pages 37–48.

Reich, J., Monsanto, C., Foster, N., Rexford, J., and Walker, D. (2013). Modular SDN programming with Pyretic. *USENIX login*, 38(5).

Smolka, S., Eliopoulos, S., Foster, N., and Guha, A. (2015). A fast compiler for NetKAT. In *ACM SIGPLAN*, pages 328–341.

Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. (2014). Merlin: A language for provisioning network resources. In *ACM CoNEXT*, pages 213–226.