# Database Criteria for Network Policy Chain

Anduo Wang
Temple University
adw@temple.edu

## ABSTRACT

Network policies that offer vital functionalities are often organized in a chain. Current practice either assumes proper policy chains as a prior or relies on simple syntax-based input-output analysis. This paper examines and addresses several difficulties with this approach — context-dependent policy interaction, unnecessarily coupled policies, and policies that must be jointly examined, proposing database integrity constraints as a means towards a semantic-based finer solution. Built on a unified logical framework to describe and reason about policy chains, our database solution gives (1) criteria that derive correct policy chain with a more accurate estimate of policy dependency, and (2) criteria that check and obtain atomic policy, unit of policy that is proper for policy chain.

## CCS CONCEPTS

• **Networks** → **Network management**; *Programming interfaces*;
• **Software and its engineering** → *Automated static analysis*;

**ACM Reference Format:**
Anduo Wang. 2018. Database Criteria for Network Policy Chain. In *SDN-NFV Sec'18: 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3180465.3180471

## 1 INTRODUCTION

Modern networks offer a rich set of functionalities (e.g., security and performance guarantee) through network policies. Whether these policies are deployed as functions fixed in traditional middleboxes or virtualized by software running on distinct servers, conceptually, to form a coherent network behavior, they are often organized into some form of policy chain [12, 20] — service chain for middlebox, priorities for SDN control modules. The majority of advancement has been on policy chain enforcement: assuming a proper policy chain as a prior, how to scalably deploy the policies and how to steer traffic to enforce the policy chain [10, 13, 24]. But how to arrive at a meaningful policy chain in the first place?

A straightforward solution is based on input-output dependency analysis [1, 23, 26] — a dependency is identified between a pair of service functions if the output of the first policy creates a flow space that overlaps with the input flow space of the second. The idea is to construct a chain consistent with the detected dependencies (e.g., by topological sort over the dependency graph containing all pairwise dependencies). Unfortunately, with arbitrary network policies, input-output dependency fails to give general and accurate criteria that can be used to guide policy chain construction.

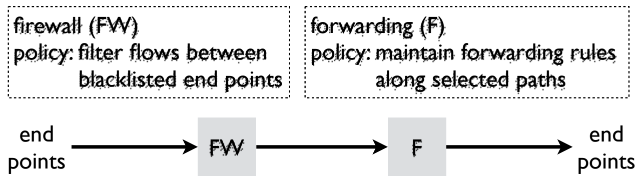

firewall (FW)
policy: filter flows between blacklisted end points
forwarding (F)
policy: maintain forwarding rules along selected paths
end points
FW
F
end points

**Figure 1: Example policy chain of firewall (FW) and forwarding (F)**

To see why input-output dependency falls short, consider the policy chain of firewall (`FW`) and forwarding (`F`) policies: `FW` blocks flows according to some access control list while `F` maintains forwarding rules along selected paths. Conventional wisdom places `FW` policy in the egress router before applying any forwarding rules on internal nodes, or equivalently on an SDN controller, a `FW` module is given a higher priority over the `F` module. This policy ordering (depicted in Figure 1) that requires traffic blocking to precede forwarding, however, cannot be recognized by the input-output dependency analysis. The difficulty is that (1) while firewall is a per-node function, the forwarding policy is a network-wide one; and (2) while firewall does generate output (packet not filtered) that overlaps with input of forwarding, so does forwarding output overlap with firewall input. In short, input-output dependency is a syntactic-based over approximation that fails to capture the semantic-based policy interaction of `F` and `FW`.

In addition to the obvious challenge of determining a proper policy ordering to ensure meaningful policy interaction, a more subtle issue is what constitutes the right unit of policy? The middleboxes that have matured [6] over the years or the more flexible units embedded in arbitrary SDN modules [15, 25] that arise in networking practice do not necessarily provide the right *units* of network policies. A middlebox policy or SDN module might contain independent sub-components (internal policies) that are unnecessarily coupled, and only at the level of those smaller internal policies does proper policy chain occur. On the other hand, a middlebox/module alone may not contain sufficient information to form a proper chain, making joint examination of multiple correlated policies necessary.

To see the need to divide policies unnecessarily coupled, consider the policy chain of firewall (`FW`) and load balancer (`LB`) in Figure 2: `FW` is the same as in the previous example while `LB` manages traffic from and to a collection of back-end servers that share a common public address. The difficulty is that the proper ordering depends on the context of the traffic: packets from the clients should follow a chain of `FW`–`LB` so that firewall can duly performs filtering, whereas

*SDN-NFV Sec'18, March 19–21, 2018, Tempe, AZ, USA*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5635-0/18/03…$15.00
https://doi.org/10.1145/3180465.3180471

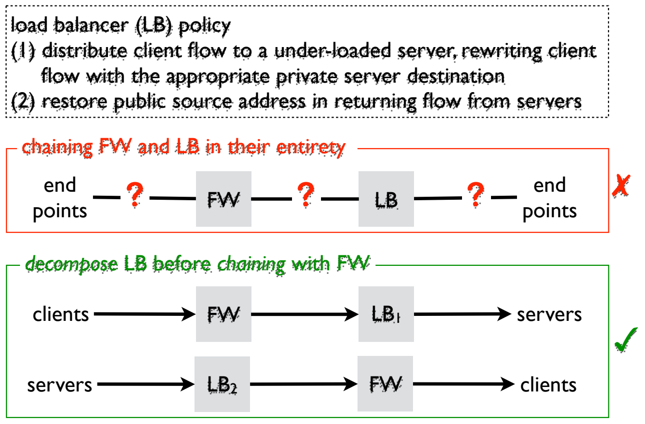

load balancer (LB) policy
(1) distribute client flow to a under-loaded server, rewriting client flow with the appropriate private server destination
(2) restore public source address in returning flow from servers
chaining FW and LB in their entirety
end points
?
FW
?
LB
?
end points
✗
decompose LB before chaining with FW
clients
FW
LB1
servers
servers
LB2
FW
clients
✓

**Figure 2: Example of policy chain requiring policy division**

returning traffic from the servers needs to restore their public address (by LB) before filtering (by FW), thus demanding a different chain LB-FW[1]. [23] called this the "decompose and re-compose" problem. We also note that the recent trend of software-centric (SDN and NFV) networking can make this problem worse: The policies (deployed middleboxes) fixed by the topology can often utilize topology dependency (viewed as a limitation) to naturally enforce the symmetrically reversing policy chain; In the new software-centric era free of this restraint, however, a meaningful policy chain solely depends on the operator who now must carefully reason about the *internals* and subtle interactions between the virtualized policies.

To see the need for joint examination, consider a network adapted from an invasion scenario in [22], as shown in Figure 3a. The three switch network (A,B,C) connects clients $H_1$,$H_2$ and servers $S_1$,$S_2$ under three policies — firewall FW forbids communication between H1 and S1, source modification S, and destination modification D. Suppose also that both S and D are compliant with FW in the sense that neither will modify a flow into one that will be blocked by FW. That is, when operating separately, both are perfectly independent of FW, and can be safely placed after FW, as shown in Figure 3b (left). Unfortunately, an implementation of this seemingly correct policy chain, as shown in Figure 3b (top-right), will allow H1 to bypass the firewall: H1 can reach S1 by sending packets destined to S2 that is allowed by FW, and manipulating the rewrites at S and D to collectively deliver the packet to S1 — S modifies the source to H2 followed by D that rewrites the destination to H1). The difficulty here is that innocent policies, when combined together (unified), can jointly produce harmful output. A proper policy chain that prevents such joint harm is shown in the bottom of Figure 3b. While this toy example is artificial in nature, we believe it reveals a neglected yet vital subtlety.

In response to these difficulties, this paper investigates database integrity constraints as a means to a more general and accurate understanding of policy chain construction. The key insight is to model the semantic of network policies as integrity constraints maintained by database query and update. This model gives a precise logical framework to describe network policies. More importantly, it gives an accurate estimate of policy dependency through the database analysis of queries independent of updates (or updates that are irrelevant to queries). This finer policy dependency analysis enables powerful correctness criteria that address all the difficulties mentioned in the above.

To summarize, our database solution gives:

**A precise logical language to describe and reason about network policy** We model a network as a database whose valid states and allowed state transition are defined by a collection of database *integrity constraints* (invariant) [11]. The network policies that determine those states and transitions are reduced to the maintenance of the integrity constraints through the unified database language of *queries* and *updates*: The query statement checks the network states for constraint violation, the update statement reconfigures network state to repair a broken constraint. Together, the query and update formulation allow us to characterize policy interaction as the database problem of determining when an update cannot affect a query (*irrelevant update*) [4, 9, 11, 16].

**A criterion to be used in determining policy chain.** We observe that, intuitively, a policy chain is meaningful if it preserves the semantics (constraint) of every member policy. Thus, the gist to correct policy chain is to take care of policy dependency when the repairing update of one policy may introduce new violation to the constraints of others. Built on this insight, we develop a criterion that reduces policy chain correctness to compliance with behavioral dependency — a more accurate estimate of policy dependency based on database irrelevant update reasoning.

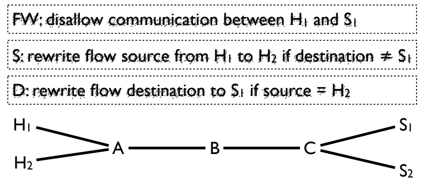
**A criteria to be used in deriving atomic policy for chaining.** We formalize network policies that are "unnecessary coupled" or "incomplete" by two novel notions — *divisibility* and *unifiability*. Under the logical framework presented in the above, divisibility and unifiability gives the right unit of constraints (policies) for policy chaining. We also sketched method to check and obtain atomicity for divisible policies. We leave the general discussion for unifiable policies to future work.

## 2 A DATABASE MODEL

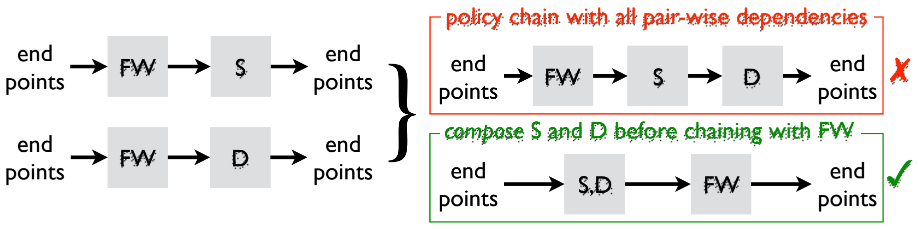
We first develop a logical framework for describing network polices. We adopt a relational data model of network and represent the entire network state — network configuration, forwarding state, policy-specific state such as access control list etc— as database tables [28]. The key insight is that, based on this relational model, network policies can be seen as database integrity constraints, that is, statements about what are that valid network state and what are the allowed transitions. The intended behavior of a network policy is then captured by database query and update that maintains the constraints.

We note that a relational data model has the advantage of being "under-specified", without forcing any particular form of abstraction on the network or the polices, making the representation extensible to future needs. It also eliminates the disparate design details (e.g., data structures, high-level language constructs) that are adopted for a particular policy — for example, forwarding policy is often

[1]In fact, performing the input-output analysis over LB and FW would result in cyclic mutual dependency.


FW: disallow communication between H1 and S1
S: rewrite flow source from H1 to H2 if destination ≠ S1
D: rewrite flow destination to S1 if source = H2
H1
H2
A
B
C
S1
S2

**(a) Example network under three policies FW,S,D**


end points
FW
S
D
end points
policy chain with all pair-wise dependencies
compose S and D before chaining with FW
S,D

**(b) A proper policy chain (right) must jointly consider the effect of S and D**

**Figure 3: Example policy chain that requires policy unification**

viewed as transformation *function* while middlebox chain is often depicted as graphs, details that are likely to be irrelevant in policy chaining.

An example network scheme [28] of the data model consists of three tables:

```
tp(sid, nid)
        # topology: edges from sid to nid
flow(fid,srcip, dstip)
        # flow requirement between srcip and
          dstip with id fid
cf(fid, sid, nid)
        # configuration (forwarding table)
```

tp is the topology table that stores link pairs (sid, nid). flow is the end-to-end flow requirements between srcip and dstip. An additional attribute fid is also introduced to uniquely identify the flow fid. For simplicity, flow table identifies nodes only by IP address, leaving out additional header fields (e.g., source MAC address, TCP source port).

**A unified database query and update language**

As shown in Figure 4, we describe a network policy by its intended behavior through a unified language of database query and update. The idea is to model a network policy by a query program and an update program that, together, maintain some invariant of concern: the *query program* checks the network states for violations of the invariant, the *update program* computes the new network for repairing the broken invariant. The query and update program can be specified by SQL statements as in [28]. In this paper, we adopt the equivalent rule-like language based on datalog because the rule form has a natural connection with formal logic — a rule has a precise interpretation as Horn clause [2] — that simplifies static analysis.

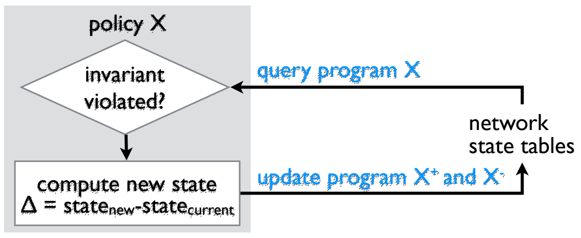


**Figure 4: Network policy as database query and update**

Take the network and the policies FW and F in Figure 1 as an example. Suppose FM has a data model of two tables: acl(srcip, dstip) that stores the pair of endpoints that are not allowed to communicate, and fw_v(fid) that contains the flow ids identifying flows that should be filtered. With these two tables and the three network base tables described in the above, we can model FW by a query program FW that checks firewall violation and an update program $FW^-$ that specifies the repairing update. In general, a policy's update program can specify insertion and/or deletion over the network base tables (denoted by + and - respectively).

```
# query program FW
r1:    fw_v(F)    :- flow(F,X,Y), acl(X,Y)

# update program FW⁻
r2:    flow(F,X,Y):- fw_v(F), flow(F,X,Y)
```

Each of the query program and update program contains a single rule r1 and r2, respectively. r1 detects flow that violated FW when the flow source and destination pair matches an acl entry. r2 is the repairing update that removes flow entries that are detected in r1. Note that rule r1 has a direct logical interpretation (Horn clause) of $\forall F$ flow(F,X,Y)∧acl(X,Y) $\implies$ fw_v(F). That is, the FW violation view fw_v is characterized by the constraint flow(F,X,Y)∧acl(X,Y). Likewise, the update (flow) computed by the head of r2 is characterized by the constraint fw_v∧flow. We call the former *invariant constraint* and the later *update constraint*. Together, these constraints can be seen as the *integrity constraint* (or *invariant*) maintained by FW. More importantly, these logical characterization enables static analysis of the dynamic behavior of network policies (§ 3).

Similarly, the data model and database program for forwarding policy is as follows:

```
# query program F
r3:    f_v(F)     :- flow(F,X,Y), path(X,Y,p⃗),
                     ¬cf(F,M,N), MN ∈ p⃗

# update program F⁺
r4:    cf(F,M,N)  :- fw_v(F), flow(F,X,Y)
```

Note that, in the body of r3, a negative literal ¬cf is used in the absence of a configuration entry (cf). In general, to be able to specify constraint violation, we add negation in the rule body.

## 3 CORRECTNESS CRITERIA

We propose to define the correctness of policy chain as a semantic-preserving property: a policy chain is correct if it respects the semantics (integrity constraints) of every member policy. Given a

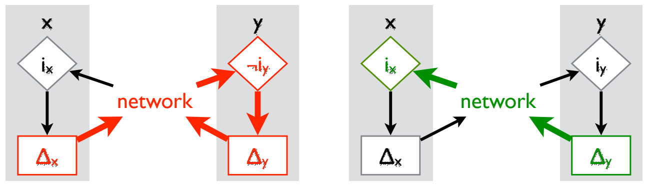


**Figure 5: Behavioral dependency `X` → `Y`**

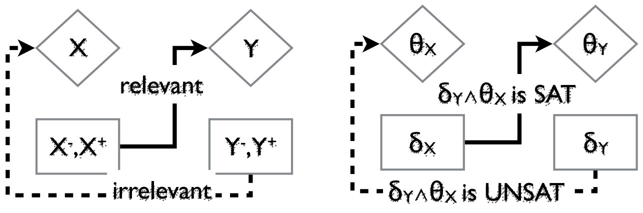


**Figure 6: Behavioral dependency as database (ir)relevant update: `X` → `Y` if (left) `X+/-` update is *relevant* to query `Y` but `Y+/-` is *irrelevant* to `X`; Or equivalently (right), if the update constraint of X ($\delta$`X`) and the invariant constraint of `Y` ($\theta$Y) is jointly *satisfiable*, but the update constraint of Y ($\delta$`Y`) and the invariant constraint of `X` ($\theta$X) is jointly *unsatisfiable*.**

set of policies each of which maintains some integrity constraint about the network, a meaningful policy chain should allow each policy to continue its constraint enforcement. Thus, the crux to policy chain is to manage policy dependency when a policy update that repairs its own invariant inadvertently affect other policies (constraints).

## Behavioral dependency

To capture policy dependency, we develop the formal notion of behavioral dependency that characterizes the effect of one policy's update on the constraint of another. As shown in Figure 5, a policy `x` depends on policy `y`, denoted by `x`→`y` (or `y`←`x`), if (1) there exists some `x` update (repair) that can transform the network into a state that violates `y`'s constraint; (2) but there does not exist any `y` update that can cause violations to `x`'s constraint. Intuitively, in the case of `x`→`y`, to restore the network into a state that satisfies both `x` and `y`, the repair update of `x` alone is not sufficient, additional "cooperating updates" from `y` is needed.

Besides, we say `x` partially depends on `y`, denoted by `x`⇝`y` if only condition (1) is known. We say `x` is independent of `y`, denoted by `x`|`y`, if neither `x` nor `y` updates will effect the other's constraint.

By formulating the dynamic behavior of a policy by a pair of database (Figure 5) query and update programs, we recast policy dependency as a database problem called (ir)relevant update [4, 16]. Essentially, as shown in Figure 6 (left): `X` depends on `Y` (`X`→`Y`) if the repairing update of X ($X^+, X^-$)is relevant to — can cause changes to the evaluation result of — the query of `Y`; but the database updates specified in $Y^+, Y^-$ is irrelevant to — will never alter — the query `X`.

Furthermore, we leverage prior work [4, 9, 16] to reduce database (ir)relevance reasoning to satisfiability analysis, shown in Figure 6 (right). An update is relevant or irrelevant to a query if the update constraint and the query constraint is jointly satisfiable (SAT) or

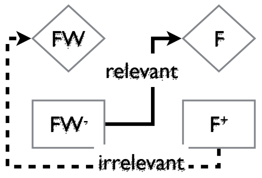


**Figure 7: Example analysis of behavioral dependency `FW`→`F`.**

not satisfiable (UNSAT), respectively. As an example, consider the policies `FW` and `F` in (Figure 1). To determine the behavioral dependency `FW`→`F`, it is sufficient to determine the (ir)relevant updates depicted in Figures 7.

To see why `FW`$^-$ is relevant to `F`, the deletion constraint for `FW`$^-$ as defined in `r2` is `fw_v(F)∧flow(F,X,Y)`, the query constraint of `F` as defined in `r3` is `flow(F,X,Y)∧path(X,Y,`$\vec{p}$`)∧¬cf(F,M,N),MN∈` $\vec{p}$. Their conjunction is satisfiable when `FW` deletes a new flow (identified by flow id `F`) that does not match any per-switch entries (`cf(F,U,V)`). To see why `F+` is irrelevant to `FW`, note that the insertion constraint defined in `r4` results in an empty set of `flow` entries. Also, the conjunction of the insertion constraint and the `FW` query constraint (defined in `r1`) is a partial evaluation of `flow(F,X,Y)∧acl(X,Y)` over the empty set which, by definition, is unsatisfiable.

## A strawman criterion

Equipped with the formal notion of behavioral dependency, we can formalize the correctness criterion for policy chain as follows:

Definition 1 (strawman criterion). *A policy chain is correct if it is compliant with all pairs of behavioral dependencies.*

It is easy to see that a policy ordering is compliant with a behavioral dependency `X`→`Y` if `X` precedes `Y`. More generally, this definition gives us a constructive method: First, build a dependency graph that contains the behavioral dependency between all pair of policies — each vertex in the graph represents a policy, and the edges denotes behavioral dependency between the two endpoints. A policy chain is then obtained by a topological sort over the graph. For example, with the behavioral dependency of `FW`→`F`, we can build the policy chain in Figure 1.

We also note that, our strawman criterion subsumes the input-output dependency method [23] in the sense that the input-output overlap dependency can always be reduced to behavioral dependency, but not vice versa. To see why, consider two arbitrary functional modules `m,n` between which a input-output dependency exists, that is the output of `m` overlaps with the input of `n`. Denote the output of `m` by $\mathtt{m}_{out}$ and input of `n` by $\mathtt{n}_{input}$, we have $\mathtt{m}_{out}\wedge\mathtt{n}_{input}$ which is satisfiable. We construct the database representation of `m,n` as follows (only show the relevant fragments of $U_m$ and $I_n$). This should be no surprise: while input-output dependency is a syntax-based over-approximation of modular interactions, semantic dependency paints a much more accurate picture.

```
#repairing updates of m (U_m)
m_out←flow(ATTS→),some_conditions


#integrity constraint of n (I_n)
←flow(ATTS→), n_input
```

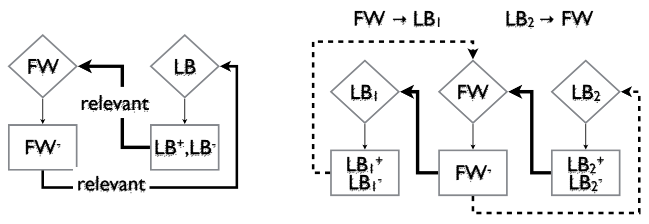


**Figure 8: Example analysis of behavioral dependency `FW`→`F`.**

## 4 CORRECTNESS CRITERIA REVISTED

**Unnecessary coupling and incompleteness? Make policies atomic!**

While the strawman approach generalizes current practice, it is not without flaws. One fundamental limitation is illustrated in the policy chain of firewall (`FW`) and load balancer (`LB`) (Figure 2). As shown in Figure 8 (left), the behavioral dependency analysis gives cyclic partial dependency. While such cycle can be triggered by conflicting policies where a policy chain compliant with the strawman criteria does not exist[2]. However, as shown in Figure 2 (bottom), a proper policy chain does exist. Only that the proper chain is constructed between `LB`'s sub-policies and `FW`. Indeed, behavioral dependency is properly identified at this sub-component level.

A dual problem is that the information contained in a policy may be inadequate. As illustrated in policies `S` and `D` in Figure 3a, behavioral analysis gives us the estimate of `S|FW` and `D|FW`. A seemingly meaningful merge of these resulted in Figure 3b (top right), however, opens a security hole. Indeed, a more sensible analysis should yield `(S;D)`→`FM` (we use `S;D` to denote the combined effect of firing policy `S` followed by `D`, i.e. rewrite source then destination), which is consistent with the policy chain depicted in Figure 3b (bottom right). In short, it is not sufficient we to examine `S` and `FW` in separate, really, only by joint examination of `S` and `D` do we have complete knowledge for analysis.

To accommodate these two problems, we develop the formal notion of policy atomicity and revise the strawman criterion accordingly.

First, we introduce the auxiliary notions of policy *division* and *unification*. Assume a collection of arbitrary policies $P$ expressed as database queries and updates, $p, q \in P$ are two arbitrary policies. We say $p$ is divisible by $q$ (or $q$ divides $p$), if a proper decomposition (for now, just understood as normal software decomposition, a formal treatment is presented in the next subsection) of $p$ into $p_1, p_2$ would result in $p_1 \rightarrow q$ and $q \rightarrow p_2$. A policy $p \in P$ is *indivisible* if there does not exists a $q \in P$ that divides $p$. We say $p$ unifies with $q$ if there exists a policy $r \in P$ such that $p \mid r$ and $q \mid r$ but $(p; q) \rightarrow r$ or $(q; p) \rightarrow r$.

Definition 2 (policy atomicity). *A policy $p$ is atomic in a collection of policies $P$ if there does not exist a policy $q \in P$ that divides or unifies $p$.*

[2]For example, a power saving policy moving traffic *off* the under-loaded path may conflict with a traffic engineering policy that moves traffic *to* the under-loaded paths.

Definition 3 (revised criterion). *A policy chain is correct if it is compliant with behavioral dependencies over all pairs of behavioral atomic policies.*

**Dividing and unifying Policies**

In this paper, we sketch a method that divides policies to break cyclic partial dependency. We leave the general treatment of unifying policies, the revised criterion relies based on atomic policy to future work.

As an example, we show how to divide `LB` by `FW`. `FW` is the same as in Figure 1. The query and update program — `FW` and `FW`$^-$, respectively — is defined in rules `r1,r2`. The `LB` policy introduces a new table `mapping(public_add,private_add)` that maps a server's public address (visible to external clients) and the various private addresses. The behavior of `LB` is as follows: the query `LB` defines flows (`lb_v`) that either have a public destination address or a private source address. These flows need to be translated, as defined in `LB`$^-$ and `LB`$^+$. These policies form cyclic partial dependencies: `FW`$\rightsquigarrow$`LB`, `LB`$\rightsquigarrow$`FW`, as shown in Figure 8 (left).

```
query program LB
r5 lb_v(F) :- flow(F,X,Y), mapping(Y,Y')
r6 lb_v(F) :- flow(F,X,Y), mapping(X',X)

update program LB-
r7 flow(F,X,Y) :- flow(F,X,Y), lb_v(F)

update program LB+
r8 flow(F,X,Y') :- lb_v(F), flow(F,X,Y),
                   mapping(Y,Y')
r9 flow(F,X',Y) :- lb_v(F), flow(F,X,Y),
                   mapping(X',Y)
```

The objective is to divide `LB` into two sub-components such that one sub-component depends on `FW` while the other is depended on, shown in Figure 8 (right). Observe that `LB`$_1$ differs from `LB`$_2$ in that `FW`→`LB`$_1$ while `LB`$_2$ →`FW`. That is, `FW`$^-$ is irrelevant to `LB`$_2$ but relevant to `LB`$_1$. Thus, the key idea is to *use* `FW`$^-$ *as a filter to divide* `LB`. More precisely, we leverage the residue method in database semantic query optimization [7], the main idea of which is to accelerates query answering by "utilizing" semantic knowledge — integrity constraint — in the database: a query is transformed into an equivalent form that embodies integrity constraint.

We take the update constraint (defines `FW`$^-$ in rule `r2`) as an integrity constraint, and embeds its positive form — `fw_v` $wedge$`flow(F,X,Y)` which expands into `acl(X,Y)`$wedge$`flow(F,X,Y))`$wedge$`flow(F,X,Y)` by rule `r1` — into `r5`-`r6`. This transforms `LB` to `LB`$_1$. The crux is that, the body of `r6` contradicts the update constraint whereas the body of `r5` subsumes the constraint, thus the transformed program `LB1` is left with `r5`.

```
query program LB1
r5 lb_v(F) :- flow(F,X,Y), mapping(Y,Y'),
```

By embedding the negative form of the update constraint in `LB`, we obtain `LB`$_2$:

```
query program LB2
r6 lb_v(F) :- flow(F,X,Y), mapping(X',X)
```

## 5 RELATED WORK

Database research has been inspiring networking practice for a long time. In the early days, network practitioners often made use of home grown database to simplify network management [5, 27]. Declarative networking [17–19] explores a more systematic use of database language (recursive datalog) to enable a more compact and higher-level specification of routing protocols, enabling rapid deployment of declarative protocols with distributed query optimization. Following this line of work, [8] extends declarative protocols to more general network management. More recently, in the software-centric era of SDN and NFV, FlowLog[21] explores the event-condition aspect of database language to provide a unified abstraction for the control-, data- planes. Deductive database language [14] also saw application in accelerating network forwarding decision computation. In addition to leveraging database as a cross-layer, distributed, higher-level, and highly efficient domain-specific language, Onix and ONOS [3, 15] utilized database concurrency control for state replication when strong consistency is needed among network components.

Unlike all previous application of database research, in this paper, rather than leveraging the database for managing data — describing, replicating, reasoning about factual data, we utilize database system for managing *semantics constraints* [11] — non-factual data of the network policies. To the best of our knowledge, this is the first work that explores database integrity constraints in networking.

## REFERENCES

[1] [n. d.]. Ph.D. Dissertation.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2620728.2620744

[4] José A. Blakeley, Neil Coburn, and Per-:1Vke Larson. 1989. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. Database Syst.* 14, 3 (Sept. 1989), 369–400. https://doi.org/10.1145/68012.68015

[5] Don Caldwell, Anna Gilbert, Joel Gottlieb, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. 2004. The Cutting EDGE of IP Router Configuration. *SIGCOMM Comput. Commun. Rev.* 34, 1 (Jan. 2004), 21–26. https://doi.org/10.1145/972374.972379

[6] B. Carpenter and S. Brim. 2002. Middleboxes: Taxonomy and Issues. RFC 3234 (Informational). (February 2002). http://www.ietf.org/rfc/rfc3234.txt

[7] Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.* 15, 2 (June 1990), 162–207. https://doi.org/10.1145/78922.78924

[8] Xu Chen, Z. Morley Mao, and Jacobus van der Merwe. 2007. Towards Automated Network Management: Network Operations Using Dynamic Views. In *Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management (INM '07)*. ACM, New York, NY, USA, 242–247. https://doi.org/10.1145/1321753.1321757

[9] Charles Elkan. 1990. Independence of Logic Database Queries and Update. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, New York, NY, USA, 154–160. https://doi.org/10.1145/298514.298557

[10] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. 2013. FlowTags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/2491185.2491203

[11] Parke Godfrey, John Grant, Jarek Gryz, and Jack Minker. 1998. Integrity Constraints: Semantics and Applications. In *Logics for Databases and Information Systems (the book grow out of the Dagstuhl Seminar 9529: Role of Logics in Information Systems, 1995)*. 265–306.

[12] Joel M. Halpern and Carlos Pignataro. 2015. Service Function Chaining (SFC) Architecture. RFC 7665. (2015). https://doi.org/10.17487/rfc7665

[13] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 239–253. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khalid

[14] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. 2014. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 203–216. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/koponen

[15] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*.

[16] Alon Y. Levy and Yehoshua Sagiv. 1993. Queries Independent of Updates. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 171–181. http://dl.acm.org/citation.cfm?id=645919.672674

[17] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/1142473.1142485

[18] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative Networking. In *Communications of the ACM*.

[19] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative Routing: Extensible Routing with Declarative Queries. *SIGCOMM Comput. Commun. Rev.* 35, 4 (Aug. 2005), 289–300. https://doi.org/10.1145/1090191.1080126

[20] Thomas Nadeau and Paul Quinn. 2015. Problem Statement for Service Function Chaining. RFC 7498. (2015). https://doi.org/10.17487/rfc7498

[21] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*. 519–531. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/nelson

[22] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. 2012. A Security Enforcement Kernel for OpenFlow Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*. ACM, New York, NY, USA, 121–126. https://doi.org/10.1145/2342441.2342466

[23] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. [n. d.]. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *SIGCOMM '15*.

[24] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 27–38. https://doi.org/10.1145/2534169.2486022

[25] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. [n. d.]. Modular SDN Programming with Pyretic. ([n. d.]).

[26] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-state Management Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 563–574. https://doi.org/10.1145/2619239.2626298

[27] Philip Taylor and Timothy Griffin. 2009. A model of configuration languages for routing protocols. In *PRESTO*.

[28] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. 2016. Ravel: A Database-Defined Network. In *SOSR*.