Access Control for a Database-Defined Network

Noemi Glaeser[‡] Anduo Wang[*]
[‡]University of South Carolina [*]Temple University
nglaeser@email.sc.edu adw@temple.edu

***Abstract*—Software-defined networking (SDN) allows the insertion of software that manages the network through a centralized controller. While the controller improves network management through features such as network-wide and higher-level abstraction, the urgent requirement of security is still less well-studied. Ravel, a database-defined controller, like many others currently exposes all network states to its users without implementing any security measures. This position paper proposes a novel way to implement access control, a specific aspect of security for SDN, in the setting of the Ravel controller.**

***Index Terms*—Software-defined networking (SDN), databases, access control.**

I. Introduction

Sometimes referred to as the last bastion of mainframe computing, networking today is increasingly complicated and fragile. Practitioners are locked in solutions, both protocols and software, that are tightly bound to vendor-specific hardware. Additionally, every component must be individually configured at a low level, and administrators must have extensive knowledge of the network in order to enforce the desired functionality and security.

In order for networks to continue to advance, the field of networking needs to borrow a page from other computing disciplines, such as programming languages, distributed systems, operating systems, and database research. Software-defined networking (SDN) is the latest effort to leverage these disciplines towards a more manageable network.

In SDN, operators insert network management software via a central controller. One such controller is Ravel, which utilizes a standard relational database to represent and manipulate the network. An important but less-studied issue in SDN controllers is security, and Ravel is no exception. This position paper addresses a portion of the security issue. We do not offer a comprehensive solution, but focus specifically on extending Ravel to incorporate access control.

II. Background and motivation

SDN seeks to simplify network administration by introducing abstraction. This is achieved by connecting each network component to a centralized controller, reducing switches and other middleboxes to dummy pieces of hardware that only take routing commands from the controller. With this abstraction, changes to the network can be made much more easily and efficiently through the controller, without requiring extensive knowledge of the network. SDN is still a young area of research, however, and more efforts are needed before SDN can be deployed more broadly beyond the current, narrow point solutions.

Among many other implementations, Ravel proposes the use of databases as the abstraction for managing SDN. Ravel (see Figure 1) is a database-defined controller which represents the network using a standard relational database [5]. Using a database as a controller allows for abstraction in the form of views, which can be tailored to the needs of each controller application, essentially serving as an API between the controller and the apps. Using SQL triggers, Ravel can also enforce coordination between the various applications modifying the network. All this is done through a PostgreSQL interface.

By building on relational database research, Ravel deals with many of the same issues as other researchers in the field. This gives us the advantage of having a large body of previous research at our disposal do draw from for improvements, as well as an older community in addition to that of the still-budding field of SDN.

Security, although a crucial aspect of networking, has not been extensively studied in the context of SDN. Network architecture was developed in a more innocent age, but as attacks became a more pertinent threat, security measures had to be retrofitted onto the existing networks. These *post hoc* implementations gave rise to an excess of disparate devices (such as router ACLs, firewalls, NATs, and other middleboxes in addition to VLANs and other complicated link-layer technologies) which are difficult to manage and coordinate, often resulting in weaker security [1], [3]. SDN simplifies network security, but security requirements are still under development, and there is no clear consensus on what the desirable characteristics of a network are that would ensure its integrity. Like most other SDN controllers today, Ravel does not yet have any security measures in place.

One suggestion is to restrict the direction of information flow in a network. In this approach, nodes are assigned with security levels (top secret, secret, confidential, unclassified), and flows are restricted according to the security levels of the nodes they connect and/or pass through to avoid information leakage from secure to less secure switches, users, or applications.

Other efforts involved the use of access control lists (ACLs). ACLs allow a network administrator to explicitly specify what information users are authorized to access. When a user then puts in a request, only the authorized information is presented, in accordance with the ACL(s). For each rule he or she wishes to implement, the administrator must list the principal (the user the rule applies to), the object (which node/table/etc. the rule applies to), and the privilege (e.g. read, write, delete). Although at first, this seems like a reasonable expectation, such specifications are very manual and become increasingly tedious as size of the network and users grows. Even a network with only a couple thousand users quickly becomes unmanageable for the operator. This kind of explicit specification simply does not scale up [4]. In this project, we propose a novel way to define access control lists and address the access control problem for SDN for the database-defined controller Ravel.

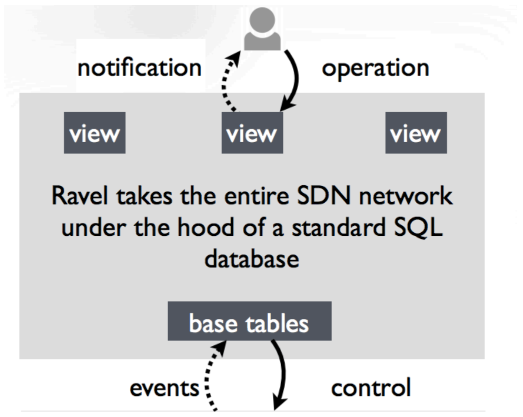

notification
operation
view
view
view
Ravel takes the entire SDN network under the hood of a standard SQL database
base tables
events
control

Fig. 1. Ravel is the realization of SDN by database.

III. Proposed solution

Our novel proposal seeks to achieve a more concise and flexible specification of access control through the use of reflective specification of access control rules. The intuition of reflective specification is the idea of stating the intent rather than the extent of rules. Consider, for instance, a large network where nodes can be leased to so-called tenants. Alice owns a portion of the nodes, and Bob owns some other nodes in the network. The active flows in the network are recorded in a **reachability matrix**, which lists the source and destination nodes of every flow. We wish to restrict the flows visible to tenants. Instead of a series of statements like "Alice can view flows that initiate in Alice's part of the network", "Bob can view flows that initiate in Bob's part of the network", and so on for every tenant, we can express the *intent* of the policy by writing, "tenants can view flows that initiate in their part of the network" [4]. This way we can restrict the information available to users while also providing a scalable security mechanism.

This content-based access control is realized through declarative queries (i.e. SQL queries). As a realization of this principle, we added this type of access control to Ravel, the aforementioned database-defined network controller, implementing these queries with PostgreSQL. We used SQL views defined in terms of the current user, effectively creating a single view that tailors itself to each distinct user.

A. Access Control for Network Resources

Users' access to network resources is based on the network **service-level agreement (SLA)**, which lists the tenants and their nodes. One resource that

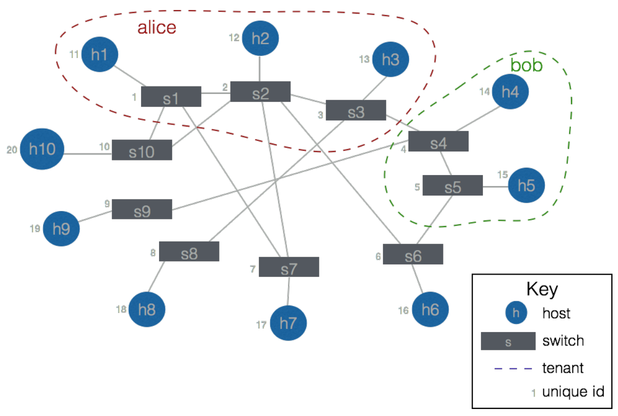
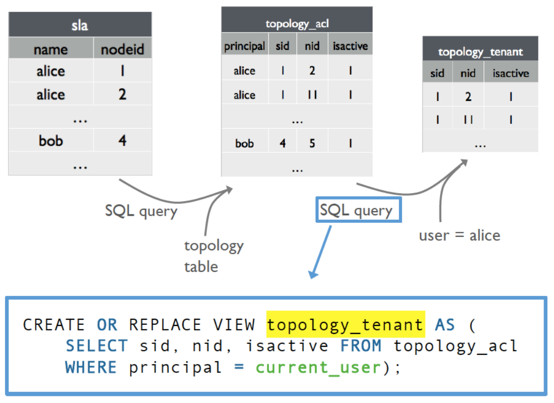


Fig. 2. Working example of a network with two tenants.



sla	
name	nodeid
alice	1
alice	2
...	
bob	4
...	

topology_acl			
principal	sid	nid	isactive
alice	1	2	1
alice	1	11	1
...			
bob	4	5	1
...			

topology_tenant		
sid	nid	isactive
1	2	1
1	11	1
...		

```
CREATE OR REPLACE VIEW topology_tenant AS (
    SELECT sid, nid, isactive FROM topology_acl
    WHERE principal = current_user);
```

Fig. 3. Access control for network resources.

tenants should only have limited access to is the topology of the network: each tenant should only be able to see his or her own nodes. In order to enforce this requirement, we first create a query that generates an **access control list** (**ACL**) for the topology table. The ACL is represented in the form of the `topology_acl` view, which queries both the SLA table and the network `topology` table:

```
CREATE OR REPLACE VIEW topology_acl AS (
  SELECT s.name AS principal, sid, nid, isactive
  FROM topology, sla s
  WHERE
    topology.sid IN (
      SELECT nodeid FROM sla WHERE name = s.name)
    AND
    topology.nid IN (
      SELECT nodeid FROM sla WHERE name = s.name)
);
```

The access control view is an intermediate step. It is important to note that it is a view, i.e. a virtual table, meaning it automatically updates when changes are made to the tables referenced in its definition. Because views are virtual, they also take up no storage space: they consist only of a query which is executed each time the view is referenced.

Next, we create the `topology_tenant` view as follows and make it accessible to all users:

```
CREATE OR REPLACE VIEW topology_tenant AS (
    SELECT sid, nid, isactive FROM topology_acl
    WHERE principal = current_user);
GRANT SELECT ON topology_tenant TO PUBLIC;
```

The power of `topology_tenant` view comes from its use of the `current_user` variable. Because users authenticate with the controller, i.e. the database, this variable is unique to every user. Thus, the `topology_tenant` view is automatically updated and tailored to each user. It contains only the entries of the `object_acl` view where the principal is the current user, thus only displaying the relevant permitted (according to the SLA) information to the current user. If we connected to the network as Alice, for instance, the `topology_tenant` view would only contain the rows from `topology_acl` where the value of the "principal" column is `'alice'` (see Figure III-A).

The same methodology was used to restrict access to the flows installed in the network, which are logged in the reachabiligy matrix (the `rm` table). Below is an example tenant view for the reachability matrix:

```
CREATE VIEW rm_tenant AS (
  SELECT fid, src, dst FROM rm WHERE rm.src IN (
     SELECT nodeid FROM sla WHERE name IN (
       SELECT p1 FROM config_sla
         WHERE p2=current_user )
     UNION
     SELECT nodeid FROM sla
       WHERE name = current_user )
  AND
  rm.dst IN (
    SELECT nodeid FROM sla WHERE name IN (
      SELECT p2 FROM config_sla
        WHERE p1=current_user )
    UNION
    SELECT nodeid FROM sla
      WHERE name = current_user )
  AND (
    rm.src IN (
      SELECT nodeid FROM sla
        WHERE name = current_user )
    OR
    rm.dst IN (
      SELECT nodeid FROM sla
        WHERE name = current_user))
```

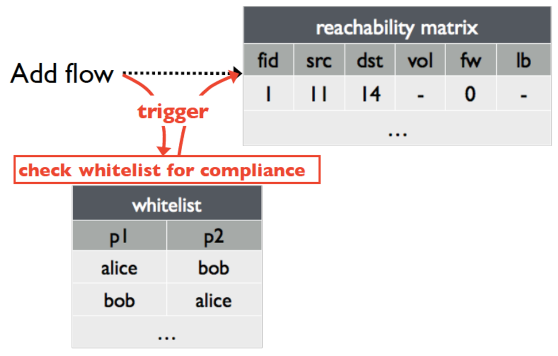


Fig. 4. Access control for network updates.

```
);
```

The reachability matrix is discussed in more detail in the following section.

B. Access Control for Network Updates

We also used SQL queries for access control on the write operations to the network. This was done through the use of triggers on the tables reflecting the network state. A **trigger** is a user-defined procedure which is automatically executed whenever a specific database condition is met (this encompasses any database events such as insertions, deletions, and updates). In this case, any insertion (e.g. a new network flow) into the network's reachability matrix `rm` causes a trigger to fire, which executes a procedure that checks the proposed insertion against the SLA and the current user.

For instance, if the user Alice attempts to insert a new flow into our working network example from Figure 2, the trigger on the `rm` table is fired, running a function which checks whether or not this flow is compliant with the network's whitelist (dictated by the SLA and the network provider). If the flow adhere's to the whitelist, it is inserted as a new entry into the `rm` table and the network itself changes accordingly. Otherwise, nothing happens. (See Figure III-B.)

C. Challenge: Controlling Network Updates

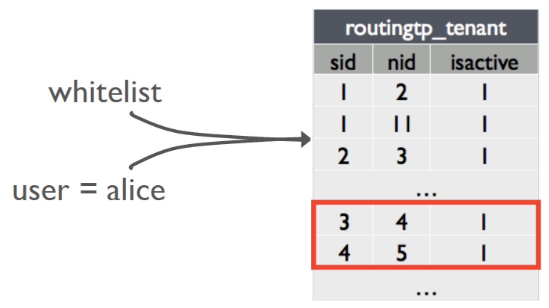
Any network update usually involves a nontrivial computation. In the case of the reachability matrix, this computation is the calculation of the path the flow will take through the network. This path should also be compliant with the SLA. In order to enforce this restriction, we define a new view called `routingtp_tenant` which, like the aforementioned views, also automatically tailors itself to each particular user. The `routingtp_tenant` view is a representation of a portion of the network's topology: it contains only the nodes of the current user and all users he or she is permitted to communicate with. In our example, Alice has been whitelisted to talk to Bob, so Alice's `routingtp_tenant` view contains her nodes and Bob's nodes (see Figure III-C).

```
CREATE OR REPLACE VIEW routingtp_tenant AS (
  SELECT sid, nid, isactive FROM topology_acl
    WHERE principal = current_user
  UNION
  SELECT sid, nid, isactive FROM topology_acl
    WHERE principal IN (
      SELECT p2 FROM config_sla
        WHERE p1 = current_user)
  UNION
  SELECT sid, nid, isactive FROM topology
    WHERE sid IN (
      SELECT sid FROM topology_acl
        WHERE principal = current_user)
    AND nid IN (
      SELECT sid FROM topology_acl
        WHERE principal IN (
          SELECT p2 FROM config_sla
            WHERE p1 = current_user))
  UNION
  SELECT sid, nid, isactive FROM topology
    WHERE nid IN (
      SELECT sid FROM topology_acl
        WHERE principal = current_user)
    AND sid IN (
      SELECT sid FROM topology_acl
        WHERE principal IN (
          SELECT p2 FROM config_sla
            WHERE p1 = current_user))
);
```

Although both this and the `rm_tenant` view definition are rather long, SQL is the most intuitive language in which to define such policies. Such a definition still defines the intent rather than the extent of the access control policy and is dynamic and updatable. Achieving the same functionality would take many more lines of code in another language such as Python, C, or Java.

Now, we can direct the path algorithm to only reference nodes contained in the `routingtp_tenant` view, thus ensuring that the path each flow takes only contains nodes from this view so the paths are also compliant with the SLA.

Using a database as the SDN controller provides



routingtp_tenant		
sid	nid	isactive
1	2	1
1	11	1
2	3	1
...		
3	4	1
4	5	1
...		

Fig. 5. Controlling network updates.

two key features that can be exploited to simplify access control: (1) users authenticate with the database upon login and (2) the database language SQL easily yields itself to the statement of concise, flexible, and expressive authorization policies. The database approach gives us higher-level and finer-grained control over database policies. Besides allowing us to specify access control rules more concisely, this system is dynamic, requiring minimal changes by the administrator upon addition of a new user.

As is often the case, implementing these features was easier said than done. It soon became clear that restricting read access had been comparatively simple once we began to implement access control for the write operation. There were always multiple approaches for every issue, and each solution had its own advantages and drawbacks. Often, simply the promise that one approach would work was not enough, since we also wanted the implementation to be conceptually clean and practicable.

When determining the route a flow should take, for instance, we needed to modify the Dijkstra algorithm in place to only take into account the allowed middleboxes, given each flow's origin and destination (for instance, the path for a flow originating from one of the admin's nodes could be calculated using the regular shortest path algorithm, but a path originating in, say, one of Alice's nodes needed to be calculated so it would only be routed through nodes to which Alice has read access. In this case, we had several options: (1) create alternate network state tables to pass to the Dijkstra function (i.e. an `rm'` table) or (2) modifying the Dijkstra function at runtime. We opted for the latter choice, modifying the trigger function to pass it the `routingtp_tenant` table, as described above, instead of the normal topology table. The former would have needlessly commplicated coordination between the routing app (which installs flows and calculates the path they take) and other applications, possibly even resulting in multiple, divergent instances of the network base tables.

The final product has been included in an experimental branch of Ravel. Users can use Ravel's existing command-line interface (CLI) to load the access control application and then add tenants, modify the SLA, change the whitelist, etc. through a customized application CLI.

Acknowledgements

This research was conducted as part of an NSF-funded Research Experience for Undergraduates (REU) held at Temple University in Philadelphia. For an in-depth demonstration of Ravel's new features, follow the readme on our GitHub: http://github.com/ravel-net/REU-access-control. There you can also find directions on how to download Ravel. Ravel uses Mininet [2], which allows users to create virtual networks within their own machines. By downloading Ravel, you can go through our step-by-step tutorial on the access control developed in this project.

References

[1] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proceedings of USENIX Security Symposium*, Vancouver, B.C., Canada, 2006.

[2] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of Hotnets*, Monterey, CA, USA, 2010.

[3] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control for enterprise networks. In *Proceedings of WREN*, Barcelona, Spain, 2009.

[4] L. E. Olson, C. A. Gunter, W. R. Cook, and M. Winslett. Implementing reflective access control in sql. In *Proceedings of DBSec*, Montreal, Canada, 2009.

[5] A. Wang, X. Mei, J. Croft, M. Caesar, and B. Godfrey. Ravel: A database-defined network. In *Proceedings of SOSR*, Santa Clara, CA, USA, 2016.