Trident: Toward a Unified SDN Programming Framework with Automatic Updates

Kai Gao
Tsinghua

Taishi Nojima
Yale

Y. Richard Yang
Yale/Tongji

ABSTRACT

Software-defined networking (SDN) and network functions (NF) are two essential technologies that need to work together to achieve the goal of highly programmable networking. Unified SDN programming, which integrates states of network functions into SDN control plane programming, brings these two technologies together. In this paper, we conduct the first systematic study of unified SDN programming. We first show that integrating asynchronous, continuously changing states of network functions into SDN can introduce basic complexities. We then present Trident, a novel, unified SDN programming framework that introduces programming primitives including stream attributes, route algebra and live variables to remove these complexities. We demonstrate the expressiveness of Trident using realistic use cases and conduct an extensive evaluation of its efficiency.

CCS CONCEPTS

• **Networks → Programming interfaces**; **Network management**; *Middle boxes / network appliances*;

KEYWORDS

SDN, Network functions, Network programming, Stream attributes, Live variables, Route Algebra

ACM Reference Format:
Kai Gao, Taishi Nojima, and Y. Richard Yang. 2018. Trident: Toward a Unified SDN Programming Framework with Automatic Updates. In *SIGCOMM '18: ACM SIGCOMM 2018 Conference, August 20–25, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3230543.3230562

1 INTRODUCTION

Bringing together software-defined networking (SDN) and network functions (NF) is an essential step toward highly programmable networking. In particular, SDN introduces the ability of logically centralized, global network routing; network functions (*e.g.*, deep packet inspection, DPI), on the other hand, introduce the ability of network elements conducting general-purpose, programmable packet processing beyond the network layer. Only by combining SDN and network functions can one realize programmable, cross-layer networking. The importance of integrating SDN and network functions has motivated multiple studies (*e.g.*, [1–7]).

Unified SDN programming, which integrates information extracted by network functions, such as application layer HTTP headers, into SDN programming, is an approach to bringing SDN and network functions together. By exposing such information, which we refer to as *network function state information* or *network function state* for short, to SDN programming, unified SDN programming can bring many benefits such as adaptive, cross-layer SDN control.

Despite the benefits, realizing unified SDN programming, however, is non-trivial, due to 3 basic programming complexities that are not fully addressed before:

1) *How to naturally integrate network function state into SDN programming.* Current SDN programming frameworks (*e.g.*, [8–12]) make decisions on the headers of each individual packet. A network function state, on the other hand, may not appear in a single packet but span multiple packets, and hence need to be extracted by a network function using a finite state machine. For example, a DPI can extract the HTTP `URI` field [13] for an SDN program to better route traffic (*e.g.*, large files are routed differently). The extraction of the field, however, is a progress: the field can be *unknown* for a *non-deterministic* amount of time due to TCP three-way handshake and fragmentation. Existing SDN programming frameworks do not have a model to expose cross-packet, *asynchronous* state.
2) *How to flexibly construct consistent, correlated routes to utilize network function state.* One benefit of SDN is the ability to realize complex, advanced routing, such as traffic engineering routing and fast rerouting. Integrating network function states into SDN routing can exacerbate the complexity. For example, a network function may have route symmetry as a consistency requirement: both forward and return traffic go through it to update its finite state machine reliably. Computing advanced routing and at the same time enforcing consistency for network functions can then become a substantial programming complexity.

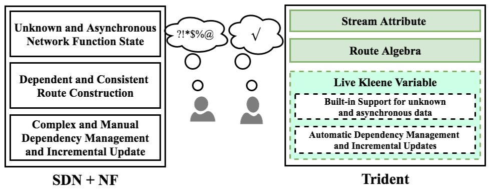

Unknown and Asynchronous Network Function State
Dependent and Consistent Route Construction
Complex and Manual Dependency Management and Incremental Update
?!*$%@
√
Stream Attribute
Route Algebra
Live Kleene Variable
Built-in Support for unknown and asynchronous data
Automatic Dependency Management and Incremental Updates
SDN + NF
Trident

Figure 1: Complexities and TRIDENT Primitives.

Although there are excellent previous studies in SDN routing (*e.g.*, [5–7, 14–19]), they are lacking in supporting efficient, flexible routing for unified SDN programming.

3) *How to handle dynamicity of unified SDN programming.* In the general case, the state of a network function can continuously change: a DPI updates the URI when a persistent HTTP connection requests the next content; a security network function updates the security level of an existing connection. When the state changes, the routes may also need to change. Thus, accurate and efficient dependency management is crucial in achieving correct, efficient utilization of network function states. Existing approaches use either manual identification of data dependencies (*e.g.*, [6, 20, 21]), which can be complex and error-prone, or simple automatic dependency tracking and complete recompilation (*e.g.*, [22, 23]), which can lead to unnecessary recomputation and large latency.

In this paper, we introduce TRIDENT, a novel SDN programming framework that addresses the preceding 3 complexities with 3 powerful, simple-to-use programming primitives. A mapping between complexities and solution primitives are shown in Figure 1. Specifically, the 3 primitives include:

1) *stream attribute* (§5), a simple abstraction for modeling cross-packet states extracted by network functions;
2) *route algebra* (§6), a simple yet powerful abstraction to flexibly express consistent, advanced routing;
3) *live Kleene variable* (§4), a unified data model and the foundation of TRIDENT, which enables built-in support for unknown and asynchronous data, as well as *semantic dependency tracking* and automatic incremental updates.

We implement ((§7) a prototype of TRIDENT and conduct an extensive evaluation (§8) to demonstrate that TRIDENT can be realized cleanly and efficiently.

2 MOTIVATION

We start with a simple example to illustrate how an SDN program may use network function state and the 3 complexities introduced in §1. The example uses a simplistic network shown in Figure 2, in which two DPIs are deployed. We use an algorithmic SDN programming model [10], due to its flexibility and simplicity. At a high level, the example SDN program routes traffic on the bypass link if the internal user accesses an HTTP URI that is not "sensitive"; otherwise, the traffic should be continuously monitored by a DPI.

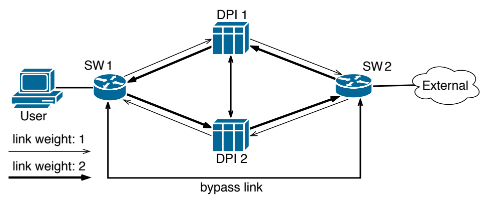


Figure 2: Example Network.

C1: Naturally Integrating Network Function State into SDN. As simple as the objective of the SDN program is, existing high-level SDN programming frameworks (*e.g.*, [8–10, 24]) typically do not support routing using layer-7 information such as "`http_uri`", as such information is unlike packet header fields, which are contained in every packet. Instead, such information can be extracted only by a network function, potentially after reassembling multiple packets.

Despite the reassembling need, one might think that the issue would be trivial to solve: network functions and SDN controllers share a common data store, and the network functions update the data store. This leads to a simple SDN program as below, where "`http_uri`" is a variable that DPIs update for each HTTP flow in the data store:

```
if (pkt.http_uri === "sensitive")) {
  // This branch can never be reached
} else {
  // Bypass link
}
```

As simple as it is, the program does not achieve the objective. When the first packet (TCP SYN) of a "sensitive" HTTP flow enters the network, a DPI cannot decide whether the packet belongs to an HTTP flow with a "sensitive" URI. Thus, the result is *unknown*. Consider two approaches handling unknown. First, assume an unblocking (*i.e.*, asynchronous) design in that a DPI assigns an initial value null for such a case. Using standard logic, the controller will evaluate the condition of the if statement as false, and hence the packet should be handled by the else branch, resulting in that the packet uses the bypass link, which has no DPI at all. With no DPI to inspect and update, "`http_uri`" will remain null, and all packets will use the bypass link. Second, assume a blocking design, in that the SDN program blocks when reading "`http_uri`", if the value is not available. However, with no route returned, the whole system blocks, with no update on "`http_uri`", leading to a deadlock.

Summary: Cross-packet, unknown and asynchronous network function states can be complex to handle.

C2: Constructing Consistent, Correlated Routes to Utilize Network Function States. Assume that one can fix the preceding complexity, so that the initial packets of each HTTP flow from the user go through a DPI. Below is a potential code segment showing route computation:

```
if (pkt.http_uri === "sensitive")) {
  // compute shortest path from src to dst, must use a DPI
} ...
```

A problem of the preceding route computation, however, is that it does not enforce routing consistency required by network functions. In particular, a network function may have symmetry as a consistent requirement, in that the return packets go through the same network function instance (*e.g.*, Pico Replication [25]). Assume directed link weights shown in Figure 2, the shortest path algorithm computes SW1→DPI1→SW2 for the forward (user to outside) packets, and SW2→DPI2→SW1 for the return (outside to user) packets, leading to inconsistency (different DPIs).

One might think that the inconsistency problem would be easy to fix: when setting SW1→DPI1→SW2 as the route for the forward packets, setting at the same time an inverse of it for the return packets. Although this works in a simple setting, in a practical setting where both a primary route and a backup route are installed to implement fast rerouting, the technique may not work: when SW1 switches the route of the forward packets from primary (SW1→DPI1→SW2) to backup (SW1→DPI2→SW2) when link SW1→DPI1 is down, there are no existing mechanisms to specify that the return packets need to automatically change route as well.
Summary: Although there are previous studies on the construction of correlated routes, for example, Genesis [16] introduces the ability to specify disjoint routes, a mechanism for systematic construction and enforcement of consistent routes supporting unified SDN programming is missing.
C3: Handling Dynamicity of Unified Programming. The preceding discussions on **C**1 and **C**2 already touch on the complexity of dynamicity: in **C**1, when a network function updates its state, the execution path of the SDN program may change; in **C**2, when a link fails, the backup routes will need to consistently replace the primary routes. In the general setting, other data can trigger updates as well.

Consider a revision of the SDN program shown below, where the program uses source IP to determine `user`, and each `user` has a customized `white_list`:

```
if (user.white_list.contains(http_uri)) {
  ...
}
```

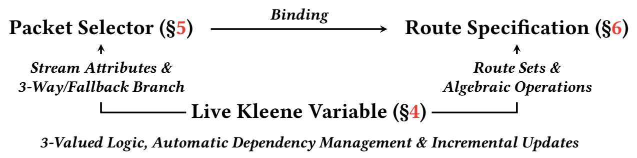
In this example, network function state (`http_uri`), configuration state (`white_list`), and network topology state can all lead to program outcome changes. Thus, identifying dependencies and ensuring the consistency between control plane state and outcomes can become extremely complex and error-prone (*e.g.*, [20, 21, 26–28]. Even with an event-driven system [6, 29], specifying the states and transitions can also be complex, because the number of states grows exponentially with the number of data.

3 OVERVIEW

With an understanding of the key complexities, we now introduce Trident. We first introduce the programming primitives in Trident that address these complexities. We then go over Trident programming workflow using an example.

3.1 Trident Programming Primitives

To understand not only how Trident programming primitives address the complexities in §2 but also how they fit into general SDN programming, consider the "match-action" paradigm of SDN programming: each SDN program need to specify (1) the selection of packets (match), (2) the action for each selection, and (3) the binding between match and action. Trident introduces (1) stream attributes to extend traditional packet selection to support network function states; (2) route algebra to specify consistent routing actions; and (3) live variables to achieve consistent binding between match and action despite dynamicity. The diagram below illustrates how Trident extends SDN programming:


Packet Selector (§5)
Binding
Route Specification (§6)
Stream Attributes & 3-Way/Fallback Branch
Route Sets & Algebraic Operations
Live Kleene Variable (§4)
3-Valued Logic, Automatic Dependency Management & Incremental Updates

Stream Attribute. Traditional SDN programming uses headers in each packet to select packets, but as we identified in **C**1, network function states can be cross-packet. Hence, Trident introduces stream attribute as an abstraction for cross-packet network function state. Hence, a Trident program can use both packet headers and stream attributes to select packets.

Specifically, a stream attribute not only exposes a network function state but also specifies the group of packets that must be sent to the network function to compute the state. For example, to specify "`http_uri`" as a stream attribute, one must specify that it takes a complete TCP flow (TCP5TUPLE); on the other hand, a stream attribute such as "`is_endhost_infected`" may need all packets from/to the same endhost. Hence, a stream attribute exposes not only the output (state) but also the input of a network function, allowing Trident to make consistent, efficient decisions.

One key complexity of exposing stream attributes, as we discussed in **C**1, is that their values can be unknown and continuously change. Hence, in Trident, stream attributes can use only unknown-conforming operations: They can be either the *IS_UNKNOWN* operator ? (examples in later sections), or unknown-enforced control structures including *3-way branch* and *fallback branch*. The examples below illustrate the two control structures and the left hand handles the example in **C**1: Now all possible values of "`http_uri`" are explicitly enumerated, but the program still has an intuitive synchronous programming structure. Trident will automatically re-execute the program as asynchronous "`http_uri`" changes arrive (see below on live Kleene variables).

```
if (pkt.http_uri === "sensitive") {
  // The true branch
} else {
  // The false branch
} unknown {
  // The unknown branch
} // 3-way branch example
```

```
iff (pkt.is_analytics_job) {
  // The true branch
} else {
  // The false branch
  // and the unknown branch
} // fallback branch example
```

Route Algebra. Trident route algebra introduces systematic route construction primitives to address **C**2. As we discussed in **C**2, a network function can have consistency requirements on how packets traverse it, but such requirements are not fully handled in existing systems. Hence, Trident route algebra introduces a grammar called *network function indicator*, which selects routes satisfying traversal requirements. Consider the example code below:

```
1  val X = SimplePath(G, H :-: DPI :-: ISP)
2  ...
3  iff (pkt.http_uri === "sensitive") {
4      val x = any((X where { capacity >= 100 Gbps }) >> X)
5      val y = inv(x)
```

The expression `H :-: DPI :-: ISP` is a network function indicator. It means that among all simple paths (*i.e.*, a link is used at most once) in graph `G`, the result `X` only contains those connecting any host (`H`) and any external Internet service provider (`ISP`), and passing exactly one `DPI`.

Trident route algebra also introduces generic, algebraic route constructions for flexible SDN routing. In the example above, L4 constructs a route `x` using 3 algebraic operators: (1) the selection operator (`where`) selects a subset of `X` only those with a capacity greater than or equal to 100 Gbps; (2) the preference operator (`>>`), which is key to implement backup routing, specifies that routes selected in step (1) are preferred than those only in the general set `X`; (3) the arbitrary selection operator (`any`) then selects one from routes given to it. Consider the example network in §2 where all links are 100 Gbps except SW1⟶DPI2, which is only 50 Gbps. Then `x` will record that SW1⟶DPI1⟶SW2 is the chosen route, if available, and SW1⟶DPI2⟶SW2 is the backup. L5 uses the inversion operator (`inv`) to specify that `y` is the inverse of `x`. It is important to note that route algebra not only computes routes but also records how routes are computed, to systematically address **C**2, as discussed next.

Live Kleene Variable. Both stream attribute and route algebra are high-level abstractions, and a reader may already wonder how they can be realized, in particular due to dynamicity. In Trident, it is through *live Kleene variables* that stream attributes and route algebra are realized. Further, live Kleene variables go beyond stream attributes and routes to provide a generic mechanism handling dynamicity. Hence live Kleene variables are foundation to address **C1** to **C3**.

The basic function of a live Kleene variable is simple: Trident keeps track of dependencies for it and automatically updates it when its dependency changes.

We use an example including both a stream attribute and a route algebra operation to illustrate the foundational role of live Kleene variables. Figure 3 shows the example. Assume that when a packet with source IP 10.0.1.5 arrives, the network function providing `is_endhost_infected` does not know whether 10.0.1.5 is infected. From the programmer's point of view, the fallback branch of the program is executed: a route `r_1 + r_2`, is constructed using the route algebra concatenation operator `+` from route `r_1` and route `r_2`, and then the Trident built-in function `bind` specifies that the packet be forwarded using the constructed route. Under the hood in Trident, both routes and bindings are live Kleene variables and hence Trident keeps track of dependencies for them, resulting in a tracking record shown at Figure 3(a). Now assume `r_1` is changed to a new value (*i.e.*, a different route), Trident will automatically update the computation, generating a new concatenated route to forward packets for 10.0.1.5, as shown Figure 3(b). Instead of route changes, consider a stream attribute change: the network function updates that 10.0.1.5 is infected. Then Trident re-executes the program, blocking all packets of 10.0.1.5. With no packets for 10.0.1.5 and if the only input that can trigger the network function to update `is_endhost_infected` is packets, then 10.0.1.5 will be blocked indefinitely. On the other hand, the network function can have an administrative interface to reset `is_endhost_infected`, for example, after an operator disinfects the endhost. The reset will then automatically trigger Trident re-execution.

```
iff (pkt.is_endhost_infected) {
  drop(pkt)
} else {
  bind(pkt, r_1 + r_2)
}
```

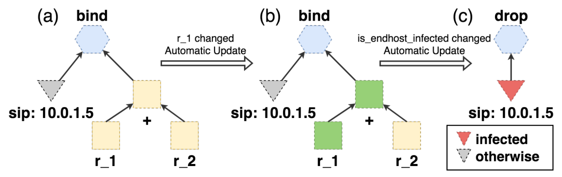


Figure 3: Packet Selector and Route as Live Variables.

3.2 Trident Programming Workflow

The preceding Trident programming primitives are generic abstractions and hence may be realizable in multiple languages or systems. Below we illustrate Trident programming using a domain-specific language embedded in a Scala-like language, to utilize its language features to improve code readability. We also choose an algorithmic SDN programming setting (*e.g.*, Maple [10] or SNAP [30]).

A complete system integrating SDN and network functions can have additional complexities such as placement and life cycle management of network functions. To focus on Trident features, we assume that the network functions are already deployed. Further, complex network functions can have complex behaviors such as reading the states of (remote) SDN controllers or other network functions. Although Trident can be extended to handle such additional cases, we focus on independent, write-only network functions, where by an independent, write-only network function, we mean one that does not read the state of either the controller or another network function.

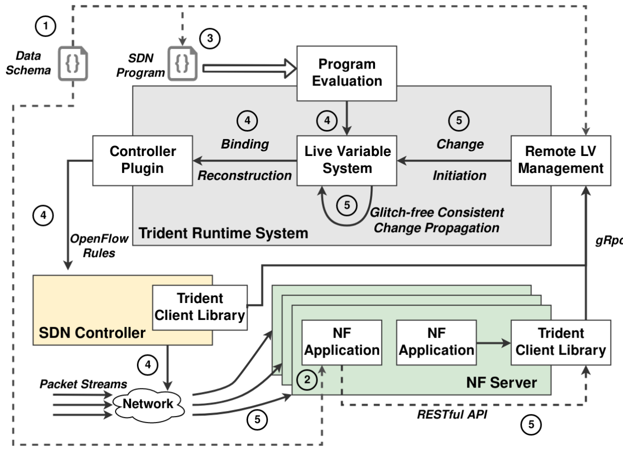


Figure 4: TRIDENT Programming Workflow.

With the preceding clarification, programming using TRIDENT is simple, with few steps, as shown in Figure 4.

Step 1: Declare Stream Attributes (Programmer). Integrating stream attributes is a key component for unified SDN programming, and in TRIDENT, using stream attributes is simple. As we discussed in §3.1, a stream attribute has both output (state) and input. Hence, a declaration of `http_uri`, as shown below, specifies that the output is a string with name “`http_uri`”, and that the responsible network function needs packets of a complete TCP flow:

```
// trident.example.DPI - object DPI
val http_uri = StreamAttribute[String]("http_uri", TCP5TUPLE)
```

Step 2: Implement Stream Attribute in Network Function (Programmer). A network function providing a stream attribute needs to be modified to continuously publish the state to TRIDENT runtime. TRIDENT provides a simple client library to simplify the publishing process, and as we will show in §8, this typically takes only a few lines of code.

Step 3: Write Unified SDN Program (Programmer). Writing a unified SDN program using stream attributes is straightforward, and below is an example program:

```
 1 // trident.example.DemoProgram
 2 object DemoProgram extends SDNProgram {
 3   import trident.example.DPI.http_uri
 4
 5   val capacity = NetworkAttribute[Link, BandwidthUnit]("capacity")
 6   val label = NetworkAttribute[Vertex, String]("label")
 7
 8   val DPI = Waypoint(V where { label === "dpi" }) expose http_uri
 9   val H = Waypoint(V where { label === "host" })
10   val ISP = Waypoint(V where { label === "isp" })
11   val X = SimplePath(G, H :-: DPI :-: ISP)
12   val Y = SimplePath(G, H :-: ISP)
13
14   override def onPacket(pkt: Packet) = program {
15     iff (pkt.http_uri === "sensitive") {
16       val x = any((Y where { capacity >= 100 Gbps }) >> Y)
17       bind(pkt, x)
18       bind(inv(pkt), inv(x))
19     } else {
20       bind(pkt, any(X))
21       bind(inv(pkt), any(X))
22     }
23   }
24 }
```

With an understanding of the primitives and logically centralized SDN programming, the program is quite straightforward: the `onPacket` function (L4) defines the behavior for each packet; L15 uses a stream attribute to select packets; L8-L12 use route algebra to construct static routes; L16 uses route algebra to construct consistent, dynamic routes; L17-L18 bind constructed routes for both traffic directions.

Step 4: Deploy Program (TRIDENT Runtime). After submitting the program to TRIDENT, the TRIDENT runtime will execute the program. The value of the `pkt` can be either reactively obtained from `packet-miss` messages or proactively constructed using *symbolic execution*. After the execution, the new bindings are added to the live variable system, and are further *translated* into datapath OpenFlow rules. A key step of the translation is to generate the “match” from the packet selector, which we will discuss in §5.1. The datapath deployment can be handled by many systems (*e.g.*, [31–33]).

Step 5: Automatic, Asynchronous Updates (TRIDENT Runtime). As packet streams enter and traverse the network according to the SDN program, network functions or switch agents update the values of certain stream attributes or network attributes, which eventually change the output of program. These asynchronous updates are sent to TRIDENT through the client library. TRIDENT runtime automatically captures the asynchronous data changes and checks whether the resulted bindings are still consistent with the program. Invalid routes and bindings are removed automatically, along with the corresponding datapath configurations. Meanwhile, the program is re-evaluated to compute new consistent bindings. Invalidation and recovery of the same stream are conducted as a single transaction to avoid inconsistency.

4 LIVE KLEENE VARIABLE

As introduced in §3, live Kleene variables (or simply *live variables*) are the foundation of the abstractions in TRIDENT. Now we formally specify the live Kleene variable in detail.

4.1 Live Variable System

Live Variable. A live variable x is specified as follows:

$$x : \langle type, value, deps, operator, operands \rangle, \quad (1)$$

where each property is defined as follows:

- *type* (T_x): the data type for x’s value, which defines the valid operations and the domain (D_x) for x’s value;
- *value* (v_x): x’s value, which is either a valid data in the domain D_x, or *unknown*, *i.e.*, $v_x \in D_x \cup \{unknown\}$;
- *deps* ($\mathcal{D}_x$): a set of live variables on which x depends. Let $\prec$ denote the dependency relationship, *i.e.*, $\forall d \in \mathcal{D}_x, d \prec x$.
- *operator* (f_x) and *operands* ($\mathcal{Y}_x$): a function which defines how x’s value is computed, and a sequence of live variables which are the input of the *operator* f_x. Let $y_{x,i}$ denote the i-th operand in $\mathcal{Y}_x$, then f_x has the following signature: $f_x : D_{y_{x,1}} \times \ldots, D_{y_{x,|\mathcal{Y}_x|}} \mapsto D_x$.

It is important to note that one can attach new properties to a live variable and enable functionalities such as efficient change propagation introduced in §7.3.

Consistency Guarantee. Let $\mathcal{X}$ denote the set of all live variables. We define two important consistency properties:

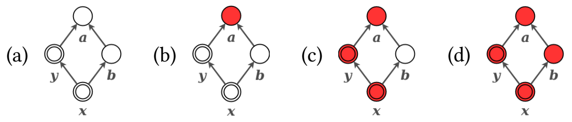
Definition 1 (Synchronized [34]). $\forall x, x' \in \mathcal{X}$ such that $x' \prec x$, x is *synchronized* with x' if and only if: *1)* $\forall d \in \mathcal{D}_x$, if $x' \prec d$, d is synchronized with x'; *2)* $\forall y \in \mathcal{Y}_x$, if $x' \prec y$, y is synchronized with x'; and *3)* $v_x = f_x(v_{y_{x,1}}, \dots, v_{y_{x,|\mathcal{Y}_x|}})$.

Definition 2 (Glitch-free [34]). $\forall x \in \mathcal{X}$, x is *glitch free* if and only if $\nexists x', y, y' \in \mathcal{X}$ such that $x' \prec y \prec x$ and $x' \prec y' \prec x$, where *1)* y is synchronized with x', *2)* y' is not synchronized with x', and *3)* x is synchronized with both y and y'.

To achieve the two properties, Trident manages the following procedures of each live variable $x \in \mathcal{X}$:

- *Dependency tracking*: to determine the dependency set $\mathcal{D}_x$, given an operator f_x and operands $\mathcal{Y}_x$.
- *Change propagation*: to determine whether v_x should be updated, and if so, compute the new value of x, given an operator f_x; operands $\mathcal{Y}_x$; a live variable $\kappa \prec x$ whose value has changed, which we call a *root cause*; and a subset of x's dependent live variables $\mathcal{D}_x(\kappa) \subseteq \mathcal{D}_x$ where $\forall y \in \mathcal{D}_x(\kappa)$, v_y is synchronized with the new v_κ.

The following example illustrates the dependency relationships and change propagation processes for four live variables, a, b, x and y, where x depends on y and b, and both y and b depend on a. In this example, a is a root cause, and x and y are exposed results. Live variables with different colors means that they are not synchronized. In this example, x and y in (a), (b) and (d) are all glitch-free.



4.2 Basic Live Variable

We now introduce three types of basic live variables, which are later extended to different programming primitives.

Simple Live Variable. A simple live variable x represents the result of an idempotent function and depends on all of its operands. When the value of a live variable $\kappa \prec x$ is updated, x's value must be recomputed in a glitch-free way, *i.e.*,:

- *Simple dependency tracking:* Given an operator f_x and operands $\mathcal{Y}_x$, $\mathcal{D}_x \leftarrow \mathcal{Y}_x$.
- *Simple change propagation (SCP):* Given an operator f_x, operands $\mathcal{Y}_x$, a root cause $\kappa \prec x$, and a set of dependent live variables $\mathcal{D}_x(\kappa)$, if and only if $\mathcal{D}_x(\kappa) = \mathcal{D}_x$, $v_x \leftarrow f_x(v_{y_{x,1}}, \dots, v_{y_{x,|\mathcal{Y}_x|}})$.

Proposition 1. SCP guarantees glitch-free consistency.

Proof. **Sketch** This can be proved by checking Definition 2 for x and its dependencies recursively. □

Remote Live Variable. A remote live variable x represents the most recent value of a remote data source, such as an internal state of a network function, and is specified as follows:

$$x : \langle type, value, deps \equiv \emptyset, operator \equiv \textbf{\textit{remote}}, \\ operands \equiv \emptyset, source, uuid\rangle, \tag{2}$$

where *type*, *value*, *deps*, *operator* and *operands* are the same as in (1), and the new properties are specified as follows:

- *source*: a sequence S_x of values provided by a remote data source, where new values are continuously appended to S_x. The domain of each value v in S_x is specified by *type*, *i.e.*, $v \in D_x \cup \{unknown\}$;
- *uuid*: a universally unique identifier (id_x) for x *i.e.*, for remote live variables x and y, $id_x = id_y \Leftrightarrow x = y$.

A remote live variable x has a special operator ***remote***, which does not take any operand; x thus does not depend on any other live variables and is managed by the rules below:

- *Remote dependency tracking*: As specified in (2), $\mathcal{D}_x \leftarrow \emptyset$.
- *Remote change propagation*: The value of x is updated whenever there is a new value in S_x, and $v_x \leftarrow S_x \downarrow$, where $\downarrow$ means fetching the last value of a sequence.

Programmers can avoid working with the *source* directly by using predefined extensions of remote live variables, *e.g.*, stream attributes (§5.1) and network attributes (§6.1).

Snapshot Live Variable. In Trident, the *operator* of a live variable can be a complex routing function. However, such a function can take a very long execution time, during which the live variable's value becomes unknown. To make the last known value always available, we introduce a snapshot live variable $\tilde{x}$, which is used to hold the last *known* value of another live variable y until y is synchronized; however, $\tilde{x}$ can have *unknown* value when the first known value of y is not yet available. $\tilde{x}$ is specified as follows:

$$\tilde{x} : \langle type, value, deps, operator \equiv :=, operands\rangle. \tag{3}$$

A snapshot live variable $\tilde{x}$ has a special operator called *snapshot assignment* (denoted as *:=*), and has exactly one operand y. $\tilde{x}$ must have the same type as y, *i.e.*, $T_{\tilde{x}} = T_y$ and is managed using the following rules:

- *Snapshot dependency tracking*: Given the operand y, if and only if y is synchronized, $\mathcal{D}_{\tilde{x}} \leftarrow \{y\}$; otherwise $\mathcal{D}_{\tilde{x}} \leftarrow \emptyset$.
- *Snapshot change propagation*: Given the operand y, if and only if y is synchronized, $v_{\tilde{x}} \leftarrow v_y$; otherwise, the value of $\tilde{x}$ does not change.

A snapshot live variable "cuts off" the dependencies conditionally to guarantee that the result is *always* glitch-free. In our examples, due to the limitation of Scala, the *snapshot assignment* is actually written as a function `snapshot`.

Example. Figure 5 includes a minimal abstract example, which involves a source S with the initial value $\langle 4 \rangle$; a remote live variable x with $id_x = 1$; a simple live variable y; and a snapshot live variable z. The right side of the figure illustrates how the values change over time. At t_4 when S has a new value, the value of x is also updated, which immediately triggers the recomputation of y. When y is not synchronized

```
S = source<Int>([4])
x = remote(S)
//x : ⟨Int, 4, ∅, remote, ∅, S, 1⟩
y = log2(x)
//y : ⟨Float, 2.0, {x}, log2, {x}⟩
z := y
//z : ⟨Float, 2.0, {y}, :=, {y}⟩
S += [5] // at t4
```

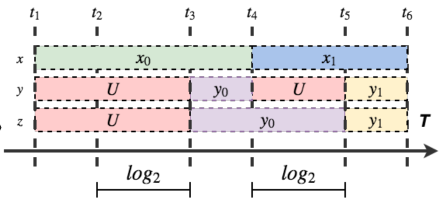


Figure 5: Example of Basic Live Variable Types.

(during the execution of the $\log_2$ function), z keeps the old value. Once y is synchronized, $v_z \leftarrow v_y$.

5 PACKET SELECTOR

On top of the live variable system, *stream attribute* and *packet selector* enable fine-grained packet selection utilizing internal states in network functions. We now provide the details.

5.1 Stream Attribute

Stream. A stream $\delta_{\mathcal{H},\Pi}$ represents a sequence of packets with common header field values. $\delta_{\mathcal{H},\Pi}$ is uniquely defined by a sequence of header fields, $\mathcal{H} \triangleq \langle h_1, \dots, h_{|\mathcal{H}|} \rangle$, and a sequence of header field values, $\Pi \triangleq \langle \pi_1, \dots, \pi_{|\Pi|} \rangle$, where $\forall i$, π_i is a valid value of h_i. $\mathcal{H}$ is called the *stream type* and can contain any standard packet header field, *e.g.*, source IPv4 address and TCP/UDP destination port number.

For a packet *pkt* and a header field h, we use $pkt.h$ to denote the value of h in *pkt*, *e.g.*, $pkt.sip$ represents the source IPv4 address of *pkt*. A packet *pkt* is in a stream $\delta_{\mathcal{H},\Pi}$, denoted as $pkt \in \delta$, if and only if $\forall i, 1 \le i \le |\mathcal{H}|, pkt.h_i = \pi_i$.

For example, consider the following two packets:

$$pkt_1 = \langle sip = 192.168.1.2, dip = 192.168.1.3, \dots \rangle,$$
$$pkt_2 = \langle sip = 192.168.1.2, dip = 192.168.1.4, \dots \rangle.$$

If $\mathcal{H} = \langle sip \rangle$, pkt_1 and pkt_2 are in the same stream, specifically, $\delta_{\langle sip \rangle, \langle 192.168.1.2 \rangle}$. However, if $\mathcal{H} = \langle sip, dip \rangle$, the two packets belong to different streams.

Stream Attribute. A stream attribute x_δ represents a specific internal state about the stream $\delta_{\mathcal{H},\Pi}$ in a network function. Let T_{sa} and n_{sa} denote the type and the name of this internal state, x_δ is specified as a *remote live variable*:

$$x_\delta : \langle T_{sa}, value, \emptyset, \textbf{remote}, \emptyset, source, uuid \equiv (n_{sa}, \mathcal{H}, \Pi) \rangle. \quad (4)$$

For example, consider an intrusion detection system which exposes a `Boolean` internal state called "is_src_infected", which indicates that the source host is diagnosed as infected. Assume host 192.168.1.2 is infected, the stream attribute of $\delta_{\langle sip \rangle, \langle 192.168.1.2 \rangle}$ has the following format:

$$x_\delta : \langle \texttt{Boolean}, true, \dots, (\text{“is\_src\_infected”}, \langle sip \rangle, \langle 192.168.1.2 \rangle) \rangle.$$

Stream Attribute Schema. As we can see, stream attributes representing the same internal state have some common properties. Thus, we use a *stream attribute schema* ϑ to uniquely represent these common properties. Specifically, ϑ is specified as a tuple: $\vartheta : \langle T_{sa}, n_{sa}, \mathcal{H} \rangle$, where T_{sa}, n_{sa} and $\mathcal{H}$ have the same meaning as in (4). The following illustrates the schema for the stream attribute "is_src_infected":

$$\vartheta : \langle \texttt{Boolean}, \text{“is\_src\_infected”}, \langle sip \rangle \rangle.$$

For simplicity, for a packet *pkt* and a stream attribute schema $\vartheta = \langle T_{sa}, n_{sa}, \mathcal{H} \rangle$, we define $pkt.\vartheta$ as follows:

$$pkt.\vartheta \triangleq \langle T_{sa}, value, \emptyset, \textbf{remote}, \emptyset, source, (n_{sa}, \mathcal{H}, \langle pkt.h_1, \dots, pkt.h_{|\mathcal{H}|} \rangle) \rangle$$

It is important to note that, by properly specifying the stream type $\mathcal{H}$ of each schema, the programmer can benefit from significantly reduced memory usage on the controller at runtime. For example, she can specify either $\mathcal{H}_1 = \langle sip, dip \rangle$ or $\mathcal{H}_2 = \langle sip \rangle$ for the schema "is_src_infected". Although $\mathcal{H}_1$ and $\mathcal{H}_2$ both guarantee that packets in the same stream share the same internal state, if she uses $\mathcal{H}_1$, TRIDENT needs to keep track of up to N distinct streams, where N is the number of all possible (sip, dip) combinations. However, if she uses $\mathcal{H}_2$, TRIDENT only keeps M distinct streams, where $M \ll N$ denotes the number of all possible *sip*.

5.2 Packet Selector

On top of the preceding data models, we now define the model of *packet selectors*, which are used to select packets into different streams based on their header field values and stream attribute values. Each packet selector $\lambda : \texttt{Packet} \mapsto \{0, 1, unknown\}$ selects packets into a stream δ such that $\delta = \{pkt | \lambda(pkt) = 1\}$, and is one of the following two types:

- a *header field packet selector*, specified as $\lambda : \langle h, op, x \rangle$, where h is a header field, x is a live variable, $op : D_h \times D_x \mapsto \{0, 1, unknown\}$, and $\lambda(pkt) \triangleq op(pkt.h, v_x)$, or
- a *stream attribute packet selector* specified as $\lambda : \langle \vartheta, op, x \rangle$, where $\vartheta = \langle T_{sa}, n_{sa}, \mathcal{H} \rangle$ is a stream attribute schema, x is a live variable, $op : D_{T_{sa}} \times D_x \mapsto \{0, 1, unknown\}$, and $\lambda(pkt) \triangleq op(pkt.\vartheta, v_x)$.

Following are examples of a header field packet selector and a stream attribute packet selector, respectively.

$$\langle sip, in, 192.168.1.0/24 \rangle, \langle http\_uri, =, \text{“www.xyz.com”} \rangle.$$

At runtime, TRIDENT translates each packet selector into proper OpenFlow match conditions, as will be discussed in §7.2. Note that a programmer can avoid working with packet selectors directly; as specified in §3.2, she can specify conditions for packet selection algorithmically and let TRIDENT automatically generate the corresponding packet selectors, which will also be discussed in §7.2.

6 ROUTE ALGEBRA

TRIDENT introduces route algebra, a simple yet powerful programming abstraction to flexibly express consistent, advanced routing. We now formally specify its details, by starting with defining basic constructs of routing on top of the live variable system. We then explain *network function indicators* and *algebraic operators*.

6.1 Route Object

We first specify basic constructs of routing strategies on top of the live variable system.

Network Component. In TRIDENT, a network topology has three types of network components: a vertex u, a port p

and a link l. Trident models each network component as a *remote live variable*. Live variables for u, p, and l are denoted as x_u, x_p and x_l respectively and specified as follows:

$$x_u : \langle \texttt{Vertex}, value \equiv \check{u}, \emptyset, remote, \emptyset, source, id(u)\rangle,$$
$$x_p : \langle \texttt{Port}, value \equiv \check{p}, \emptyset, remote, \emptyset, source, id(p)\rangle,$$
$$x_l : \langle \texttt{Link}, value \equiv \check{l}, \emptyset, remote, \emptyset, source, id(l)\rangle.$$

The values of these remote live variables are assigned from domain $\{0, 1\}$, indicating whether the corresponding component is physically functioning, *e.g.*, $\check{l} = 1$ means link l is up and configured correctly.

Network Attribute. A network component live variable $x \in \{x_u, x_p, x_l\}$ has multiple *attributes*, such as node label, port statistics and link capacity, which is used to help select routes in route algebra. For simplicity, we call these attributes *network attributes*. A network attribute na_x for x is a remote live variable, whose *uuid* is determined by *1)* the *uuid* of x, and *2)* the *name* of this attribute, denoted as n_{na}.

Similar to *stream attribute schema*, we specify a *network attribute schema* θ as a tuple $\langle type, comp\_type, name\rangle$ (denoted as $\langle T_\theta, C_\theta, n_\theta\rangle$). With a supporting live variable x and a schema θ where $T_x = C_\theta$, one can obtain an attribute $na = x.\theta$ where $T_{na} = T_\theta$, $n_{na} = n_\theta$. For example, for a vertex live variable $x_v : \langle \texttt{Vertex}, 1, \ldots, \text{"openflow:1"}\rangle$ and a stream attribute schema $\theta = \langle \texttt{String}, \texttt{Vertex}, \text{"label"}\rangle$, $x_v.\theta = \langle \texttt{String}, value, \ldots, uuid = (\text{"openflow:1"}, \text{"label"})\rangle$.

Route. A route r is a path in a network and consists of a sequence of links $L_r = \langle l_{r,1}, \ldots, l_{r,|L_r|}\rangle$. It has a source port and a destination port, denoted as src_r and dst_r. Two routes r_1 and r_2 are *1) equal*, denoted as $r_1 = r_2$, if and only if $|L_{r_1}| = |L_{r_2}|$ and $\forall 1 \le i \le |L_{r_1}|$, $l_{r_1,i} = l_{r_2,i}$, and *2) equivalent*, denoted as $r_1 \sim r_2$, if and only if $src_{r_1} = src_{r_2}$ and $dst_{r_1} = dst_{r_2}$.

We say a route r is *valid* if and only if *1)* $\forall l \in L_r$, $\check{l} = 1$, and *2)* $\forall 1 \le i < |L_r|$, the destination port of $l_{r,i}$ is the same as the source port of $l_{r,i+1}$, and we use $\check{r} \in \{0, 1\}$ to indicate whether r is valid. Since the validity of a route r depends on the physical status of network components, Trident models r as a *simple live variable*. We call this live variable as a *route object*, denoted as x_r, which is specified as follows:

$$x_r : \langle type \equiv \texttt{Route}, value \equiv r, deps, operator, operands\rangle, \quad (5)$$

and can be constructed in multiple ways:

- enumerating links $l_1, \ldots, l_n$, *i.e.* $v_{x_r} = \langle l_1, \ldots, l_n\rangle$;
- concatenating two route objects x_{r_1} and x_{r_2}, denoted as $x_{r_1} + x_{r_2}$, *i.e.* $v_{x_r} = \langle l_{r_1,1}, \ldots, l_{r_1,|L_{r_1}|}, l_{r_2,1}, \ldots, l_{r_2,|L_{r_2}|}\rangle$;
- inverting a route object x_{r_1}, denoted as $\asymp r_1$, *i.e.* $v_{x_r} = \langle \asymp l_{r_1,|L_{r_1}|}, \ldots, \asymp l_{r_1,1}\rangle$ where $\asymp l$ means inverting the source and destination ports of this link.
- selecting from a *route set* Δ for a stream δ, denoted as $\psi(\Delta, \delta)$, whose details are given in §6.2.

6.2 Route Algebra

Network Function Indicator. A network function indicator (or simply an *indicator*) is a grammar that describes how a flow traverses different network functions, using *waypoints* and *patterns*. A *waypoint* represents all instances of the same network function. A *pattern* is either *1) unidirectional*, which is encoded as $\star\texttt{X}\star$ where $\texttt{X} \in \{-, >\}$ represents the traversal rules only in one direction, or *2) bidirectional*, which is encoded as $\textbf{:}\texttt{X}\textbf{:}$ where $\texttt{X} \in \{-, >, <, =\}$ represents the traversal rules for the same flow in both the forward and reversed direction. Consider two consecutive network functions in a chain, denoted as waypoints a and b. If $X \in \{<, =\}$, a flow can use different instances of b in its life cycle. If $X \in \{>, =\}$, the return flow can use a different instance of a. All patterns in an indicator are either unidirectional or bidirectional.

An indicator has the following format:

$$w_1 \; rp_1 \; w_2 \; \ldots \; w_{|\mathcal{W}|-1} \; rp_{|\mathcal{W}|-1} \; w_{|\mathcal{W}|},$$

which interleaves a waypoint sequence $\mathcal{W} = \langle w_1, \ldots, w_{|\mathcal{W}|}\rangle$ and a sequence of $|\mathcal{W}| - 1$ patterns $\langle rp_1, \ldots, rp_{|\mathcal{W}|-1}\rangle$.

Route Set. A route set x_Δ is a live variable, where Δ is *conceptually* a set of *valid* routes, with the extensions below:

- A route r is *equivalently in* a route set Δ, denoted as $r \in_\sim \Delta$, if and only if $\exists r' \in \Delta$, $r \sim r'$ (as defined in §6.1).
- For a stream δ, a route r can be selected from a route set Δ, *i.e.*, $r = \psi(\Delta, \delta)$ only if $r \in \Delta$, $\check{r} = 1$ and $src_r = ingress_\delta$.

Formally, a route set x_Δ is specified as follows:

$$x_\Delta : \langle type \equiv \texttt{RouteSet}, value \equiv \Delta, deps, operator, operands, expr\rangle \quad (6)$$

where *expr* is explained below, while the rest are from (1):

- *expr*: an expression which describes the computation process of this route set live variable.

A route set x_Δ can be constructed in multiple ways:

- enumerating route objects $x_{r_1}, \ldots, x_{r_n}$, *i.e.* $\Delta = \{r_1, \ldots, r_n\}$;
- computed by a routing function $f : Y_1 \times \cdots \mapsto \texttt{RouteSet}$ with operands $\mathcal{Y}$, *i.e.* $\Delta = f(v_{y_{x,1}}, \ldots, v_{y_{x,|\mathcal{Y}|}})$;
- using an *algebraic operator* to construct a route set, where the result is as specified in Table 1.

Note that some algebraic operators are not independent and can be equally represented using other operators. However, their use can improve performance, *e.g.*, arbitrary selection can search for one valid route, which is much faster than using optimal selection with a constant cost function.

Table 2 illustrates how to express routing requirements for some network functions and scenarios using route algebra.

7 IMPLEMENTATION

We now give details of how Trident can be efficiently implemented. In particular, we introduce: *1)* how live variables are stored and managed, *2)* how to generate OpenFlow rules, and *3)* how routes are computed efficiently in Trident.

7.1 Live Variable Management

As illustrated in Figure 6, Trident organizes live variables in *tables*. Different live variable types are handled in slightly different ways, as we introduce below.

Union (∪)/Intersection (∩)/Difference (\\)

Given two route set Δ_1 and Δ_2, return the union/intersection/difference of Δ_1 and Δ_2:

$$\Delta_1 \cup \Delta_2 = \{r \mid r \in \Delta_1 \vee r \in \Delta_2\},$$
$$\Delta_1 \cap \Delta_2 = \{r \mid r \in \Delta_1 \wedge r \in \Delta_2\},$$
$$\Delta_1 \setminus \Delta_2 = \{r \mid r \in \Delta_1 \wedge r \notin \Delta_2\}.$$

Union ($\cup_\sim$)/Intersection ($\cap_\sim$)/Difference ($\setminus_\sim$) by Equivalence

Given two route set Δ_1 and Δ_2, return the union/intersection/difference of Δ_1 and Δ_2 using $\in_\sim$ instead of $\in$:

$$\Delta_1 \cup_\sim \Delta_2 = \{r \in \Delta_1 \cup \Delta_2 \mid r \in_\sim \Delta_1 \vee r \in_\sim \Delta_2\},$$
$$\Delta_1 \cap_\sim \Delta_2 = \{r \in \Delta_1 \cup \Delta_2 \mid r \in_\sim \Delta_1 \wedge r \in_\sim \Delta_2\},$$
$$\Delta_1 \setminus_\sim \Delta_2 = \{r \in \Delta_1 \cup \Delta_2 \mid r \in_\sim \Delta_1 \wedge r \notin_\sim \Delta_2\}.$$

Concatenation (+)

Given two route sets Δ_1 and Δ_2, return a new route set by concatenating all route pairs (r_1, r_2) in $\Delta_1 \times \Delta_2$ and removing the invalid ones:

$$\Delta_1 + \Delta_2 = \{r_1 + r_2 \mid r_1 \in \Delta_1, r_2 \in \Delta_2, dst_{r_1} = src_{r_2}\}.$$

Inversion (≍)

Given a route set Δ, return the inverse of $r \in \Delta$:

$$\asymp \Delta = \{\asymp r \mid r \in \Delta\}.$$

Preference (▹)

Given two route sets Δ_1 and Δ_2, return the *preferred* route. (If there is an equivalent route in Δ_1, do not use the ones in Δ_2):

$$\Delta_1 \triangleright \Delta_2 = \{r \mid r \in \Delta_1 \vee (r \in \Delta_2 \wedge \nexists r' \in \Delta_1, r \sim r')\}.$$

Selection (σ)

Given a route set Δ and an evaluation function $f : R^* \mapsto \{0, 1\}$, return all routes in Δ that are evaluated as 1:

$$\sigma_f(\Delta) = \{r \in \Delta \mid f(r) = 1\}.$$

Optimal selection (⋄)

Given one route set Δ and a routing cost function $d : R^* \mapsto \mathbb{R}$, return *any* route with the minimum value:

$$\diamond_d(\Delta) = \arg\min_{r \in \Delta} d(r).$$

Arbitrary selection (∗)

Given one route set Δ, return a route set containing exactly one route r in Δ:

$$*\Delta = \diamond_1(\Delta).$$

Table 1: Algebraic Operators.

Description	*Expression*
Bro	$I :>: DPI :-: H$
Bro (in OpenNF)	$(I :>: DPI :-: H) \triangleright (I :=: DPI :-: H)$
Shortest QoS Path	$\diamond_{QoS\text{-}f}(X + Y)$
Path Preference	$\sigma_{bw=100Gbps}(X) \triangleright \sigma_{bw=10Gbps}(X)$
"Make before break"	$:= (X)$
"Return to the same NF"	$x \leftarrow *(X), y \leftarrow *(X \cap_\sim (\asymp x))$
"Return to another NF"	$x \leftarrow *(X), y \leftarrow *(X \setminus_\sim (\asymp x))$

I - Internet, H - data center hosts, DPI - deep packet inspection, X, Y - some route sets,
x, y - temporary variables storing a route.

Table 2: Expressiveness of Route Algebra: Examples.

Managing Remote Live Variable. Remote live variables are updated by TRIDENT-compatible network functions. Each remote live variable x is mapped to a specific table cell, where the key is a tuple of (*table_key*, *cell_key*). Table key is the type name for a remote live variable, while cell key is the *uuid* of x. For example, the label of vertex "openflow:1" has a table key `"label"` and a cell key `"openflow:1"`, and the stream attribute "is_infected" for host *10.0.1.5* has a table key `"is_infected"` and a cell key `"10.0.1.5"`.

Managing Simple/Snapshot Live Variables. A simple or snapshot live variable is constructed using an operator. Even though Figure 6 uses an arrow from one source cell to one destination cell to represent an operator, it typically involves multiple source cells and multiple destination cells. These "internal" variables are managed using the dependency tracking and change propagation rules as defined in §4.2.

7.2 Binding Generation and Translation

We now introduce how TRIDENT automatically generates bindings from a unified SDN program and then translates the bindings into OpenFlow rules.

Automatic Binding Generation. TRIDENT automatically creates bindings as follows:

1) Before evaluating a unified SDN program for a given packet *pkt*, TRIDENT creates an execution *context*, which maintains a sequence of packet selectors Λ and a set of bindings $\mathcal{B}$ for this packet, which are both empty at the beginning.

2) Whenever a stream attribute $pkt.\vartheta$ or a packet header field $pkt.h$ is compared with a live variable x, TRIDENT appends a packet selector $\langle \vartheta, op, x\rangle$ or $\langle h, op, x\rangle$ to Λ.

3) Whenever a `bind` is invoked with a packet *pkt* and a route set Δ, TRIDENT adds a binding $\langle \Lambda, \Delta\rangle$ to $\mathcal{B}$.

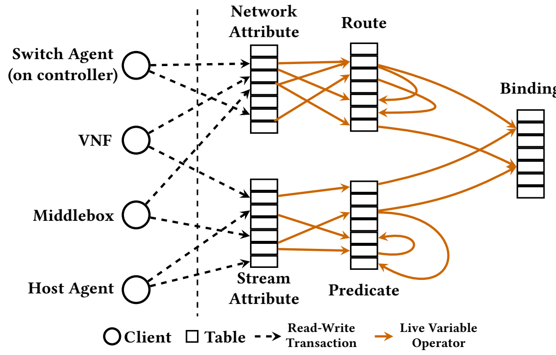


Figure 6: Tabular View of Live Variable System.

For an inverted packet (created by `inv(pkt)`), the match field in each packet selector must be inverted before the translation, *e.g.*, `pkt.sip ==="10.0.1.5"` is converted to `inv(pkt).dip ==="10.0.1.5"`.

Automatic Binding Translation. We first discuss how TRIDENT generates a match for a single packet selector and then explain how it handles the rule translation.

For a given packet *pkt*, TRIDENT uses predicates to construct a match for a packet selector λ as follows: *1)* If λ is a header field packet selector, *i.e.*, $\lambda = \langle h\ op\ x\rangle$, the predicate is $h\ op\ v_x$. *2)* If λ is a stream attribute packet selector, *i.e.*, $\lambda = \langle \vartheta\ op\ x\rangle$ where $\vartheta = \langle T_{sa}, n_{sa}, \mathcal{H}\rangle$, it is converted to a sequence of predicates: $\langle (h_1, =, pkt.h_1), \ldots, (h_{|\mathcal{H}|}, =, pkt.h_{|\mathcal{H}|})\rangle$.

For example, consider the following three packet selectors:

```
p1 = pkt.sip in "10.0.1.0/24"
p2 = pkt.http_uri === "www.xyz.com"
```

```
p3 = pkt.is_infected === true
```

If a packet with tuple $\langle$*10.0.1.5*, *10.0.2.6*, 31415, 80, *tcp*$\rangle$ satisfies all three predicates, the corresponding matches are:

Name	*sip*	*dip*	*sport*	*dport*	*ipproto*
p1	10.0.1.0/24	*	*	*	*
p2	10.0.1.5/32	10.0.2.6/32	31415	80	tcp
p3	10.0.1.5/32	*	*	*	*

The match of a stream attribute can be decomposed as the intersection of multiple header field matches. For example, p3 is translated into `pkt.sip==="10.0.1.5"`.

Trident generates OpenFlow rules using this property: it creates packet traces with only header fields by replacing each stream attribute packet selector with multiple header field packet selectors. Then, it uses Maple's trace tree [10] to generate flow rules.

7.3 Efficient Change Propagation

Simple change propagation (§4.2) is inefficient for route computation, since a route set can be the result of a complex routing function, such as a shortest path algorithm and traffic engineering. During such computation, the result becomes unknown with no valid routes available. To reduce the latency without compromising consistency guarantees in §4.1, we introduce *efficient change propagation*.

Efficient change propagation works as follows. We split a route set Δ into two subsets: a known subset $\mathcal{K}$ and an unknown subset $\mathcal{U}$[1]. We process the two subsets separately when evaluating a route algebra expression, as shown in Table 3. If the unknown subset of an expression becomes empty, usually after an arbitrary selection or a snapshot, we use the known subset as the value of this expression.

Now we prove that efficient change propagation still guarantees glitch-free consistency, by showing that the result is the same as simple change propagation.

Definition 3 (Equivalent Propagation)**.** Let x be the result of a route algebra expression f which depends on N route sets, denoted as $\Delta_1, \ldots, \Delta_N$, *i.e.*, $x = f(\Delta_1, \ldots, \Delta_N)$. For a given root cause $\kappa \prec x$, two propagation processes $\psi_1(x, \kappa)$ and $\psi_2(x, \kappa)$ are equivalent, if and only if their synchronized values $v_{x_{\psi_1}} \sim v_{x_{\psi_2}}$.

Proposition 2. *Efficient change propagation guarantees glitch-free consistency.*

Proof. **Sketch:** We prove it by showing that efficient change propagation and simple change propagation are equivalent. If the unknown subset is empty, it means the result does not depend on the unknown subsets so when they become known, the value does not change. □

[1]If a route set itself is unknown, its *known* subset is $\emptyset$.

Expr	*Known Subset*	*Unknown Subset*
$\Delta_1 \cup \Delta_2$	$\mathcal{K}_1 \cup \mathcal{K}_2$	$\mathcal{U}_1 \cup \mathcal{U}_2$
$\Delta_1 \cap \Delta_2$	$\mathcal{K}_1 \cap \mathcal{K}_2$	$(\mathcal{K}_1 \cap \mathcal{U}_2) \cup (\mathcal{U}_1 \cap \mathcal{K}_2) \cup (\mathcal{U}_1 \cap \mathcal{U}_2)$
$\Delta_1 \setminus \Delta_2$	$T_{\mathcal{U}_2=\emptyset}(\mathcal{K}_1 - \mathcal{K}_2)$	$(T_{\neg(\mathcal{U}_2=\emptyset)}(\mathcal{K}_1) \cup \mathcal{U}_1) - (\mathcal{K}_2 \cup \mathcal{U}_2)$
$\Delta_1 + \Delta_2$	$\mathcal{K}_1 + \mathcal{K}_2$	$(\mathcal{K}_1 + \mathcal{U}_2) \cup (\mathcal{U}_1 + \mathcal{K}_2) \cup (\mathcal{U}_1 + \mathcal{U}_2)$
$\asymp \Delta$	$\asymp \mathcal{K}$	$\asymp \mathcal{U}$
$\sigma_f(\Delta)$	$\sigma_f(\mathcal{K})$	$\sigma_f(\mathcal{U})$
$\diamond_d(\Delta)$	$T_{\mathcal{U}=\emptyset}(\diamond_d(\mathcal{K}))$	$\diamond_d(T_{\neg(\mathcal{U}=\emptyset)}(\diamond_d(\mathcal{K})) \cup \diamond_d(\mathcal{U}))$
$\Delta_1 \triangleright \Delta_2$	$\mathcal{K}_1 \cup T_{\mathcal{U}_1=\emptyset}(\mathcal{K}_2 - \mathcal{K}_1)$	$\mathcal{U}_1 \cup ((T_{\neg(\mathcal{U}_1=\emptyset)}(\mathcal{K}_2) \cup \mathcal{U}_2) \setminus (\mathcal{K}_1 \cup \mathcal{U}_1))$
$*\Delta$	$*\mathcal{K}$	$T_{\mathcal{K}=\emptyset}(*\mathcal{U})$

$T_\varepsilon(S)$ - the value is $S \cup \{\varepsilon\}$ if $\varepsilon = true$, and $\{\varepsilon\}$ otherwise.

Table 3: Known/Unknown Subsets of Route Algebra.

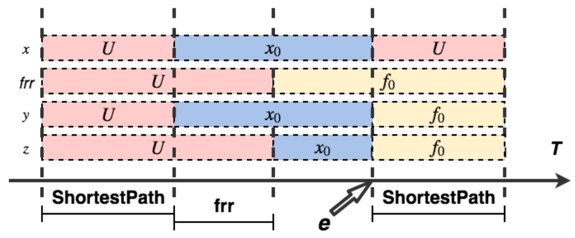


Figure 7: Results of the example in different stages. x **- the value of primary path,** *frr* **- the value of backup path,** y **- the result using efficient change propagation,** z **- the result using simple change propagation.**

Example. We use an example to demonstrate how the efficient change propagation can reduce latency. Consider the following program: use the primary path (x) if it is available, and use the backup path (*frr*) during re-computation:

```
val x = ShortestPath(G, s, t)
val xs = snapshot(x)
val y = any(xs >> frr(xs))
```

where `ShortestPath` and `frr` are two routing functions.

Figure 7 demonstrates how the *efficient change propagation* helps reduce the delay. As we can see, after the first execution of `ShortestPath`, there is one path x_0 between s and t, but the value of the backup route (*frr*) is still unknown (U). However, since only one valid path is demanded, the result of *frr* makes no difference as long as x_0 is available. Thus, the efficient change propagation is consistent and reduces the latency as it need not wait for **frr** to finish.

8 EVALUATION

In this section, we conduct extensive evaluation to answer the following key questions:

- How useful and expressive is Trident programming framework in real network management scenarios? (§8.1)
- How robust and efficient is the Trident runtime system, and what are the key factors of its performance? (§8.2)

8.1 Programming with Trident

We first demonstrate the feasibility of integrating network functions into Trident. We then show the expressiveness of Trident through real world use cases.

Name	*Attribute*	*Language*	*LoC (f)*	*LoC (a)*	*Loc (c)*
DPI	HTTP URL	Bro	40	2	2
FreeRadius	Auth status	DSL	0	12	0

LoC - Additional lines of code, **f** - LoC to implement the library in the given framework/language, **a** - LoC in a given NF, **c** - LoC for configuration.

Table 4: Integration of Different NF(V) with TRIDENT.

Ease of Integration. We measure the complexity of integrating network functions using *additional lines of code*. Table 4 shows the results for two concrete examples.

The first row is the integration of a DPI based on Bro [35] to extract HTTP URI. Our extension requires three phases: *1)* implementing a new Bro module using the `ActiveHttp` library to send inspected results to TRIDENT (40 lines), *2)* modifying a default Bro script that exposes HTTP headers (2 lines), *3)* updating the configuration (2 lines).

The second row is the integration of FreeRadius [36], an open source RADIUS server, to extract authentication status of a host. Our extension requires 12 additional lines of its domain-specific configuration language in its `rest` module and in the site configuration file.

Expressiveness of TRIDENT. We demonstrate the expressiveness of TRIDENT with two real-world programs.

The first program implements a layer-3 routing, which delivers a packet even when the topological location of its destination host is unknown. Thus, a programmer utilizes the stream attribute `dhost` (L5), which is shared by packets with the same destination IP. If the location of a packet's destination host is already known, the packet is directly forwarded to the host using the shortest path (L13); otherwise, the packet is broadcast through a spanning tree (L10). Without TRIDENT, the programmer must manually handle complexities such as host migration or topology failures.

```
1  class L3Routing extends SDNProgram {
2    val flood: RouteSet = SpanningTree(G) // G is network topology
3    val normal: RouteSet = ShortestPath(G)
4
5    val dhost = StreamAttribute[Vertex]("dhost", DST_IPADDR)
6
7    override def onPacket(pkt: Packet) = program {
8      if (pkt.dhost?) {
9        // pkt's destination host is unknown
10       bind(pkt, flood)
11     } else {
12       val x = any(normal where { dst == pkt.dhost })
13       bind(pkt, x)
14     }
15   }
16 }
```

The second program implements a SDMZ [2], which is a network access control system for high-speed scientific data traffic. The program logic is as follows: if a packet does not belong to a scientific data set, it must go through an intrusion detection system (IDS); otherwise, the packet skips the IDS and uses high-speed scientific links as long as its destination site is allowed to accept the packet's scientific data set. To achieve this goal, the programmer utilizes two stream attributes: *1)* `dataset_id` (L7), shared by packets with the same TCP 5-tuple, and *2)* `dst_site` (L8), shared by packets with the same source IP address, as data transfers are initiated by the receiver. The program groups waypoints by label (L10-12) and constructs three route sets (L13-16): `route_ids` for non-scientific traffic going through the IDS; `sdmz` and `backup` for high-speed scientific traffic bypassing the IDS, where `sdmz` uses only links with more than 40 Gbps. If a packet's dataset ID is unknown, the program forwards the packet using `route_ids` (L20); otherwise, the program extracts the packet's destination site (L22-28). If the destination site can accept the packet's data set, the program forwards the packet using `sdmz`, or if that is not feasible, using `backup`; otherwise, the program drops the packet.

Again, without TRIDENT, the programmer must explicitly handle changes of `dataset_id`, `dst_site`, `acl` to correctly forward a packet, and also topology changes to properly install the backup route.

```
1  case class ACL(val prefix: String, val allow: Boolean)
2
3  class SDMZ(val site: String) extends SDNProgram {
4    import trident.example.Net.{capacity, label} // See Section 3.2
5    val acl = SetTable[String, ACL]("acl")
6
7    val dst_site = StreamAttribute[String]("dst_site", SRC_IPADDR)
8    val dataset_id = StreamAttribute[String]("dataset_id", TCP5TUPLE)
9
10   val H = Waypoint(V where { label === "host" })
11   val IDS = Waypoint(V where { label === "ids" })
12   val GW = Waypoint(V where { label === "gateway" })
13   val route_ids = SimplePath(G, H :-: IDS :-: GW)
14                     where (capacity <= 10 Gbps)
15   val sdmz = SimplePath(G, H :-: GW) where (capacity >= 40 Gbps)
16   val backup = SimplePath(G, H :-: GW)
17
18   override def onPacket(pkt: Packet) = program {
19     if (pkt.dataset_id?) {
20       bind(pkt, any(route_ids))
21     } else {
22       val dst_site = iff (pkt.dst_site === site) {
23         // Incoming traffic, reverse the packet
24         inv(pkt).dst_site
25       } else {
26         // Outgoing traffic or unknown, use normal destination site
27         pkt.dst_site
28       }
29       iff (dst_site.acl.first(r => pkt.dataset_id == r.prefix).allow) {
30         bind(pkt, any(sdmz >> backup))
31       } else {
32         drop(pkt)
33       }
34     }
35   }
36 }
```

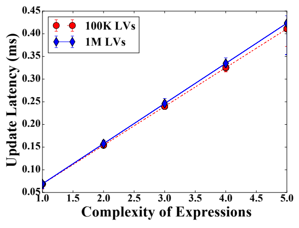
8.2 TRIDENT Runtime Performance

We evaluate TRIDENT's performance by each of our main components: live variable, stream attribute and route algebra.

Setting. We evaluate our prototype implementation on a Fedora 26 machine with an Intel Xeon CPU E5-2650 2.30GHz processor and 64G of memory, and the network is simulated with Mininet 2.3.0d1.

8.2.1 Live Variable. We conduct a microbenchmark test to evaluate how efficiently TRIDENT can handle dependency tracking, when it is scaled with the number of live variables and the number of directly dependent live variables.

Setup. We first generate up to 10 million live variables of integer type and assign a randomly generated arithmetic expression to each live variable as its dependency. We vary the


(a) Update Latency

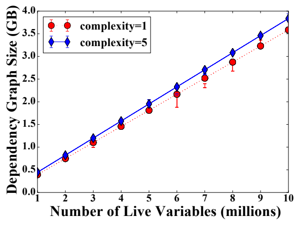
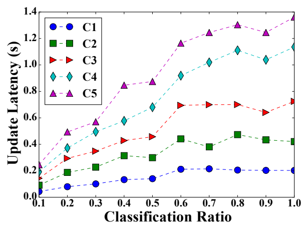

(b) Dependency Graph Size

Figure 8: Live Variable Microbenchmark


(a) Update Latency

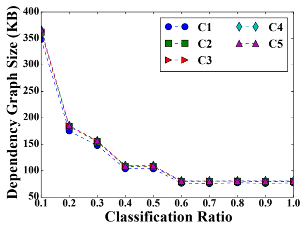

(b) Dependency Graph Size

Figure 9: Stream Attribute Benchmark

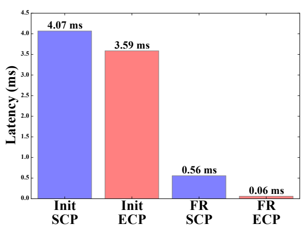
complexity of the expressions for each experiment, so that an expression of complexity n directly depends on n other live variables. Finally, We update the value of a randomly chosen live variable every 0.01 second.

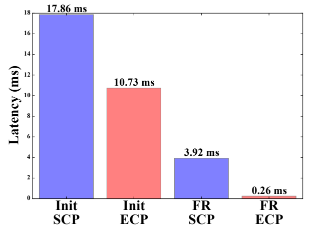
Results. Figure 8(a) shows the update latency for expressions of complexity 1 to 5, when there are 100,000 and 1,000,000 live variables. We mean the update latency by the time TRIDENT spends to correctly update all dependent variables when the value of a live variable was updated. As shown in the graph, the latency is linearly correlated with the expression complexity, while it has little relationship with the number of live variables. Figure 8(b) shows the total size of compiled dependency graphs for one to ten million live variables. Unlike the update latency, the dependency graph size is linearly correlated with the number of live variables, whereas it is relatively independent of the expression complexity. From the results, we infer that a programmer can achieve faster updates by reducing expression complexity and reduce memory usage by using fewer live variables.

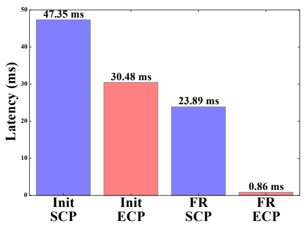
8.2.2 *Stream Attribute.*

As noted in §5.1, a programmer can enjoy reduced memory usage on the controller at runtime by specifying the proper stream type when she defines a stream attribute schema. To demonstrate this effect, we evaluate the performance of packet selection when a different fraction of packets are classified into each stream.

Setup. We first declare 100 random stream attribute schemas, all of which have source IP address as the stream type, so that we can easily control the fraction of packets classified into each stream. Based on the schemas, we randomly generate 250 packet selectors. For each experiment, we vary the complexity of packet selectors so that for complexity n,


(a) Sprint (11 nodes)


(b) Noel (19 nodes)


(c) Agis (25 nodes)

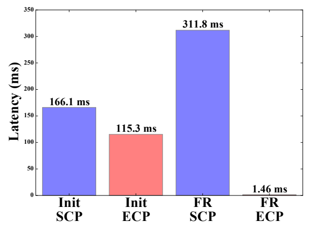

(d) Geant (40 nodes)

Figure 10: Route Algebra Benchmark

each packet selector depends on n distinct schemas. We then send 10,000 packets to the network simultaneously, where for each experiment, we vary the fraction η of all packets that share the same source IP address. We call this fraction *classification ratio*. Finally, we update a randomly chosen stream attribute every 1 *ms* and measure the performance of automatic recomputation of packet selector.

Results. Figure 9(a) and Figure 9(b) illustrate the update latency and the total dependency graph size respectively, over different classification ratios from 0.1 to 1.0. As shown in Figure 9(b), the dependency graph size decreases with a higher classification ratio, regardless of the expression complexity. This result indicates that a proper specification of the stream type for a schema can significantly reduce the memory usage on the controller at runtime. However, as shown in Figure 9(a), the update latency increases with a higher classification ratio, especially when more complex packet selectors are used. This behavior is reasonable as more number of packet selectors will need to be recomputed upon an attribute update; however, it is important for an programmer to be aware of this compromise, especially when she uses a more complex packet selector.

8.2.3 *Route Algebra.*

The key benefit of route algebra is the efficient change propagation (§7.3). We show this benefit by comparing the performance of *efficient change propagation* (ECP) and *simple change propagation* (SCP) during the initial computation (**Init**) and failure recovery (**FR**) stages.

Setup. We use the following code segment (also listed in §7.3) to compute route sets on four network topologies from Topology zoo [37]: Sprint, Noel, Agis, and Geant.

```
val x = ShortestPath(G, s, t)
val xs := x
val y = any(xs >> frr(xs))
```

Specifically, we first evaluate the total time Trident spends to compute route set `y` for all node pairs in each topology (initial computation). Then, we create link failure on an arbitrary link and measure the total time to recover valid `y` for all node pairs (failure recovery). We repeat this experiment 100 times for each topology to take the average.

Results. Figure 10 shows the results. For the initial computation, Trident achieves consistently lower latency with the ECP (second bar) than with the SCP (first bar) across all topologies. For Agis (Figure 10(d)) as an example, Trident uses 30.5 ms with the ECP, while it spends 47.4 ms with the SCP, *i.e.* 36% more efficient with the ECP. This is because, with the efficient change propagation, Trident can proceed to execute the `any` operation as long as `xs` has been computed, even if `frr(xs)` is unknown.

Trident can also recover from link failure extremely efficiently with the ECP (fourth bar), compared with the SCP (third bar), across all topologies. In particular, for Geant (Figure 10(d)), Trident spends only 1.5 ms to recover with the ECP, which is more than 200 times faster than 311.8 ms with the SCP. This is because, with the efficient change propagation, Trident can instantly update `y` as long as the already computed backup route set (`frr(xs)`) is still valid, even if `xs` became unknown due to link failure.

9 RELATED WORK

Integrating SDN and NFV. Some previous studies [7, 25, 38–42] also target the integration of SDN and NFV. However, they are leveraging the SDN technology to improve elasticity and fault tolerance of NFV systems, while Trident targets the opposite scenario.

E2 [7] and FlowTags [4] encode "stream attribute" in the data plane, and support local decisions with the attributes. However, the routing capabilities in these systems are quite limited. Trident can potentially leverage these techniques to improve the performance with a "backend" that translates bindings to FlowTags rules on FlowTag-compatible devices.

An alternative design is to use an event-driven system, such as Kinetic [6]. Trident chooses a functional reactive programming approach for the sake of simplicity.

Traceable Data and Incremental Computation. The idea of live variables comes from traceable data [43, 44], incremental computation [45–47] and reactive programming [34, 48–50]. We are the first to find novel uses of the traceable data in the field of SDN programming.

DREAM [34] and REScala [50] target fast glitch-free consistency in general reactive programming. Trident improves the performance mostly by leveraging domain knowledge.

CoVisor [51] and Wen *et. al* [52] aim to support incremental update as an add-on of SDN systems. However, they work at the level of flow rules, but cannot handle incremental update that generates these rules.

Automatic Dependency Tracking in SDN Systems. Several SDN systems [20, 53, 54] store network-related information in a distributed data store, where dependencies are managed by programmers manually. The Intent framework [22, 23] is adopted but the dependency tracking is not as flexible and efficient as Trident.

NVP [28] also uses a table-based storage system and the `nlog` language provides built-in support to identify the dependencies among different tables. However, the language is optimized for special purposes and is only used internally, it is not clear how they work in the scenario that we target.

Statesman [24] and FAST [55] encapsulate the underlying data store to track dependencies but do no leverage the domain-specific semantics in networking. Statesman also has some pre-defined semantic dependencies but they are limited to the scope of network topology.

Route Specification in Networking. The use of waypoint-based route specification is motivated by previous studies [7, 14–19]. Trident goes beyond basic network function chaining specifications in NFV systems such as E2 [7], and naturally supports QoS-based routing [56, 57]. Resource reservation (Merlin [15]) or traffic isolation (Genesis [16]) are not covered in this paper. They can be extended using customized properties in §4 and are left as future work.

With prior work such as Propane [14, 58] and DEFO [59], it is also an interesting open question that whether Trident can be used to integrate network functions with BGP.

Many route algebra operators can be overwritten by relational algebra [60], *e.g.*, concatenation is a special *join*. However, they do have domain specific meanings. Sobrinho [56] introduces basic operators (concatenation, distance function) but we extend them to a set of routes.

10 CONCLUSION AND FUTURE WORK

In this paper, we introduce Trident, a novel unified programming framework to accommodate network functions in SDN programming. Specifically, Trident introduces *live Kleene variable*, *stream attribute* and *route algebra* to achieve unified automatic dependency tracking and incremental computation, functionally complete fine-grained packet selection, and flexible route construction. We demonstrate that Trident is both expressive enough to work in various real world scenarios, and can efficiently handle changes.

Acknowledgment The authors thank Haitao Yu, Shiwei Chen, Xin Wang, Shenshen Chen (Tongji University), Alan Liu, Isabelle Carson, Yuki de Pourbaix, Lee Danilek, and Qiao Xiang (Yale University) for their help during the preparation of the paper. The authors also thank Ratul Mahajan (our shepherd) and the anonymous reviewers for their valuable comments. Kai thanks his PhD supervisor Jun Bi (Tsinghua University) for his strong support. The research was supported in part by NSFC grants NSFC #61672385 and #61472213, NSF grant CC-IIE #1440745, Google Research Award, the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001, and National Key Research and Development Plan of China (2017YFB0801701).

REFERENCES

[1] C. R. Taylor, D. C. MacFarland, D. R. Smestad, and C. A. Shue. Contextual, flow-based access control with scalable host-based SDN techniques. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, INFOCOM, pages 1–9, April 2016.

[2] Vasudevan Nagendra, Vinod Yegneswaran, and Phillip Porras. Securing Ultra-High-Bandwidth Science DMZ Networks with Coordinated Situational Awareness. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 22–28, New York, NY, USA, 2017. ACM.

[3] Sungmin Hong, Robert Baykov, Lei Xu, Srinath Nadimpalli, and Guofei Gu. Towards SDN-Defined Programmable BYOD (Bring Your Own Device) Security. NDSS'16. Internet Society, 2016.

[4] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 533–546, Berkeley, CA, USA, 2014. USENIX Association.

[5] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical Declarative Network Management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM.

[6] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable Dynamic Network Control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, 2015. USENIX Association. 00053.

[7] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[8] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM. 00547.

[9] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, Lombard, IL, 2013. USENIX Association.

[10] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98, New York, NY, USA, 2013. ACM. 00143.

[11] Ryan Beckett, Michael Greenberg, and David Walker. Temporal NetKAT. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 386–401, New York, NY, USA, 2016. ACM.

[12] Chaithan Prakash, Ying Zhang, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, and Puneet Sharma. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. SIGCOMM'15, pages 29–42. ACM Press, 2015. 00032.

[13] R Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, P Leach, and T Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999.

[14] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don'T Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 328–341, New York, NY, USA, 2016. ACM.

[15] Robert SoulÃľ, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA, 2014. ACM.

[16] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 572–585, New York, NY, USA, 2017. ACM.

[17] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM. 00163.

[18] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 109–114, New York, NY, USA, 2013. ACM.

[19] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling Path Queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 207–222, Santa Clara, CA, 2016. USENIX Association. 00029.

[20] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven SDN controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2014. 00092.

[21] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, MartÃŋn Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008. 01452.

[22] ONOS. Intent Framework, 2017. https://wiki.onosproject.org/display/ONOS/Intent+Framework.

[23] OpenDaylight. Network Intent Composition, 2017. https://wiki.opendaylight.org/view/Network_Intent_Composition:Main.

[24] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A Network-state Management Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 563–574, New York, NY, USA, 2014. ACM. 00039.

[25] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 1:1–1:15, New York, NY, USA, 2013. ACM.

[26] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing Customizable Consistency Properties in Software-Defined Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 73–85, Oakland, CA, 2015. USENIX Association.

[27] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the Brain: Races in the SDN Control Plane. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 451–468, Vancouver, BC, 2017. USENIX Association.

[28] 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. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, 2014. USENIX Association. 00191.

[29] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A Language for High-level Reactive Network Control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[30] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. pages 29–43. ACM Press, 2016. 00007.

[31] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *2015 IEEE conference on computer communications (INFOCOM)*, pages 190–198. IEEE, 2015. 00043.

[32] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM. 00433.

[33] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! page 7. ACM, 2011. 00138.

[34] Alessandro Margara and Guido Salvaneschi. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 142–153, New York, NY, USA, 2014. ACM.

[35] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-time. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 3–3, Berkeley, CA, USA, 1998. USENIX Association.

[36] freeradius.org. FreeRADIUS - the open source implementation of RADIUS, 2018.

[37] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011. 00249.

[38] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM. 00225.

[39] Jeongseok Son, Yongqiang Xiong, Kun Tan, Paul Wang, Ze Gan, and Sue Moon. Protego: Cloud-Scale Multitenant IPsec Gateway. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 473–485, Santa Clara, CA, 2017. USENIX Association.

[40] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 21:1–21:16, New York, NY, USA, 2016. ACM.

[41] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM. 00433.

[42] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 511–524, New York, NY, USA, 2016. ACM.

[43] Umut A. Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Turkoglu. Traceable Data Types for Self-adjusting Computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 483–496, New York, NY, USA, 2010. ACM. 00028.

[44] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-adjusting Computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM.

[45] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental Computation with Names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 748–766, New York, NY, USA, 2015. ACM.

[46] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, Demand-driven Incremental Computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, New York, NY, USA, 2014. ACM.

[47] Pramod Bhatotia, Alexander Wieder, \.Istemi Ekin AkkuÅ§, Rodrigo Rodrigues, and Umut A. Acar. Large-scale Incremental Data Processing with Change Propagation. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association.

[48] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[49] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[50] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[51] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015. 00037.

[52] Xitao Wen, Chunxiao Diao, Xun Zhao, Yan Chen, Li Erran Li, Bo Yang, and Kai Bu. Compiling Minimum Incremental Update for Modular SDN Languages. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 193–198, New York, NY, USA, 2014. ACM.

[53] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and others. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, volume 10, pages 1–6, 2010.

[54] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM. 00215.

[55] Kai Gao, Chen Gu, Qiao Xiang, Y Richard Yang, and Jun Bi. FAST: A Simple Programming Abstraction for Complex State-Dependent SDN Programming. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 579–580. ACM, 2016. 00000.

[56] JoÃčo LuÃŋs Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop routing in the Internet. *IEEE/ACM Transactions on Networking (TON)*, 10(4):541–550, 2002. 00328.

[57] Haijun Geng, Xingang Shi, Xia Yin, Zhiliang Wang, and Han Zhang. Algebra and algorithms for efficient and correct multipath QoS routing in link state networks. In *Quality of Service (IWQoS), 2015 IEEE 23rd International Symposium on*, pages 261–266. IEEE, 2015. 00000.

[58] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network Configuration Synthesis with Abstract Topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 437–451, New York, NY, USA, 2017. ACM.

[59] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 15–28. ACM, 2015. 00026.

[60] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970.