Carbide: Highly Reliable Networks Through Real-Time Multiple Control Plane Composition

Paper ID: 226; Number of Pages: 12 + References + Appendices

ABSTRACT

Achieving highly reliable networks is essential for network operators to ensure proper packet delivery in the event of software errors or hardware failures. Networks must ensure reachability as well as additional routing correctness, such as subnet isolation and waypoint routing. Existing work in network verification relies on centralized computation at the cost of fault tolerance, while other approaches to compose multiple control planes do not provide any guarantees of correctness. In this paper, we present *Carbide*, a novel system to achieve high reliability in networks by composing multiple control planes in real-time. *Carbide* introduces an online composition layer that uses a novel *real-time distributed verification algorithm* to compose control plane forwarding rules in accordance with provided correctness requirements. We provide several extensions to *Carbide* to address challenges in route stability and resource management. We implement *Carbide* and show that *Carbide* reduces downtime by 43% over the most reliable individual underlying control plane, while enforcing correctness requirements on all traffic.

1 INTRODUCTION

The expectation for network availability is increasingly demanding. For example, Google increased their service level objectives (SLOs) from 99% availability in 2013 to 99.99% in 2018 [9, 12]. This is because business-critical-applications are increasingly reliant on networks, and the cost of infrastructure downtime can now reach $7 million per hour [21].

Given its importance, substantial efforts have been devoted to increase network reliability [4, 5, 6, 13, 14, 15, 28, 18, 26, 27]. Recently, the approach of network verification has received much attention from researchers, vendors, and operators. Many verification methods [13, 14, 15] utilize a network-wide data structure to store all possible forwarding behaviors and compute possible violations of network requirements. Other approaches in verification [4, 5] convert network state and requirements into systems of boolean constraints and utilize SAT or SMT solvers to compute correctness. Some designs [6, 28] involve detecting packet behavior experimentally by using test packets or packet traces. In addition, previous work has been done in composing multiple control plane layers to provide reliability [18, 26, 27], generally by running a distributed control plane as a robust backup underneath a preferred centralized control plane.

Despite considerable progress, existing approaches still suffer major limitations. For example, most verification tools rely on a centralized entity to collect snapshots of the network states or configuration files from all of the network devices; but in the event of network partitions, real-time verification across the network is no longer feasible, and correctness enforcement is lost. In reality, however, two nodes in the same partition may still be able to communicate with each other in accordance with network requirements. In addition, existing work in multiple control plane composition fails to provide correctness guarantees. Such systems usually use distributed control planes to provide reach-ability without consideration of other desirable network requirements.

In this paper, we present *Carbide*, a novel system to compose multiple control planes in real-time to achieve high reliability in networks. Specifically, *Carbide* allows a network to run multiple control planes (CP) simultaneously. For example, CP_1 is a new SDN controller release (beta), still under testing for potential software bugs; CP_2 is a stable SDN controller version; and CP_3 is a distributed control plane running OSPF. By running multiple control planes concurrently and composing them correctly in real-time, *Carbide* can dynamically recover from various network errors, such as configuration errors, physical failures, and network partitions, by reverting affected traffic to a safe CP (e.g., the stable SDN release) while still ensuring correctness.

However, dynamically composing control planes presents three major challenges. (1) How can a device locally detect in real-time whether a control plane can be used? (2) Since *Carbide* relies on running multiple control planes concurrently, how can *Carbide* manage resource contention and usage among the various competing processes at each device? (3) Inconsistency and stability are major problems in such a distributed composition system.

To address the three challenges, *Carbide* introduces a novel online composition layer in each device containing three key components: (1) *CPCheck*, a light-weight distributed event-driven verification module to verify forwarding information of each control plane in real time, (2) an adaptive resource control mechanism to adjust and allocate resources to different processes, and (3) a BGP-inspired verification damping process and light-weight packet tag to alleviate route oscillation and prevent forwarding loops.

We develop *Multijet*, a switch OS software suite, to deploy *Carbide* on real white-box switches. We conduct extensive

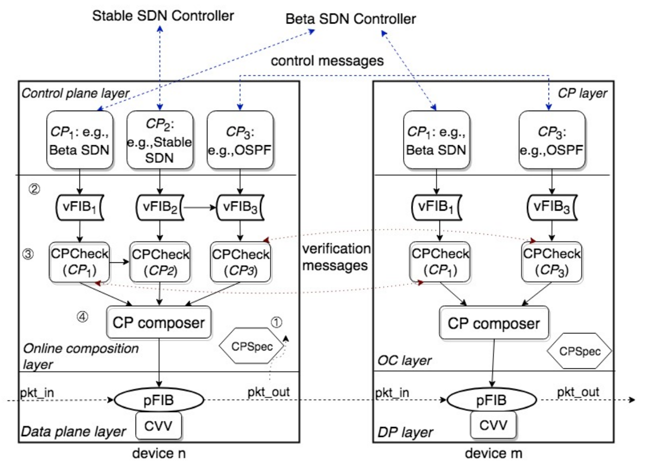


Figure 1: *Carbide* architecture.

experiments on a large Rocketfuel network topology, with white-box switches included. The micro-benchmarking results of *CPCheck* show that the local lightweight distributed real time verification process can be completed locally at each device in less than 2.5 ms. In fact, 95% of the local verifications can even be completed in under 1 ms. The system experiment results demonstrate that *Carbide* reduces downtime by over an order of magnitude compared to SDN, and by up to 43% when compared to OSPF. Even in the presence of network partitions, *Carbide* correctly enforces network requirements (e.g., on security, waypoint) on all packets. This work does not raise any ethical issues.

2 *CARBIDE* OVERVIEW

In this section, we present a high-level overview of *Carbide*. We introduce its components, describe its workflow, and illustrate its capabilities. The key components of *Carbide* are shown in Figure 1. Each network device runs (1) a control plane layer, (2) a novel online composition layer, and (3) an enhanced data plane layer.

2.1 Control Plane Layer

The control plane layer consists of a set of control plane (CP) instances $\mathcal{CP} = \{CP_1, ..., CP_k\}$ running in parallel. A control plane instance may be centralized (e.g., SDN) or distributed (e.g., OSPF). For example, a device may run three control plane instances: CP_1 as an SDN control plane receiving OpenFlow messages from a new release of an SDN controller, CP_2 as an SDN control plane receiving OpenFlow messages from a stable release of an SDN controller, and CP_3 as a traditional link-state routing protocol such as OSPF. *Carbide* treats every control plane instance as a black-box and only depends on the output (i.e., forwarding information) of each control plane instance. This design decision allows *Carbide* to use any existing implementation (open source or commercial) for each control plane instance.

2.2 Online Composition Layer

This novel layer dynamically composes the information from the control plane instances to satisfy the requirements specified by an administrator. The online composition layer consists of four components.

- ***CPSpec***: This component enables administrators to specify correctness requirements (e.g., waypoint traversal, ACL, etc.) for each CP of different traffic types, and the overall preference order between control plane instances. For the rest of this paper, we consider $\mathcal{CP} = \{CP_1, ..., CP_k\}$ as an ordered set of control planes from most to least preferred. Because different CPs may have different desired behavior, *CPSpec* allows operators to specify *different* requirements for each CP.
- **vFIB**: Each CP is associated with a virtual forwarding information base that collects and stores the forwarding information from that control plane instance.
- ***CPCheck***: Each CP is also associated with *CPCheck*, a light-weight event-driven verification module. Upon detecting any change in a vFIB or local forwarding state (e.g., port/link failures), *CPCheck* verifies the correctness requirements for that CP in real time. For example, an administrator may specify a new release of the SDN controller (CP_1) to enforce specific waypoint traversal. A software bug may result in incorrect Openflow rules and fail to enforce the waypoint requirement. The goal of *CPCheck* is to detect which packets have the desired correctness requirements satisfied. One key challenge for *CPCheck* is that it cannot rely on a centralized entity; in the case of network partitions, a centralized entity may not be reachable. Details of the verification module are described in Section 3.
- **CP composer:** The CP composer takes as inputs the forwarding rules from the vFIBs and the verification results from the *CPCheck* modules and computes the configuration to enforce at the data plane layer.

2.3 Data Plane Layer

This layer provides two main functions:

- Process data packets at line rate based on rules installed by the CP composer in the device's physical FIB (pFIB).
- Prevent forwarding loops for data packets by adding a control plane visiting vector (CVV) tag in each packet to record the previously visited control planes. In particular, devices may choose to forward a given packet using different CPs during periods of convergence. For example, a device *i* may use CP_1 to send a packet to device *j*. However, device *j* may in turn send the packet back to *i* based on CP_2, resulting in a forwarding loop. CVV addresses such issues by tagging the CPs and setting sufficient conditions

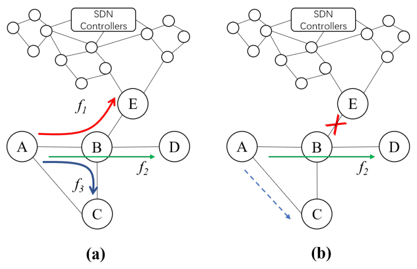


Figure 2: Example network. The failure of link *BE* causes a network partition with a set of nodes no longer able to communicate with the SDN controllers.

	Reachability	Traffic blocking	Traverse a firewall (B)
$\{CP_1, CP_2, CP_3\}$	All traffic that $dstPort! = 23$	All traffic that $dstPort = 23$	Flow that $srcIP = A$, $dstIP = C\vert D\vert E$, $dstPort = 80$

Table 1: Illustration of correctness requirements for three CPs. Each row represents a CP_i. Each column represents a correctness requirement j. Each cell (i, j) represents the set of packets for which the correctness requirements j must be satisfied through CP_i.

to prevent such loops. The details of CVV tagging and inconsistency issues are discussed in Section 4.

2.4 Resource Control

Carbide consists of a large number of running processes (e.g., control plane instances, *CPCheck* modules, CP composer) that compete for shared resources (e.g., CPU, memory, bandwidth). Resource control, therefore, plays a critical role. Without resource control, a control plane process may use all of the device's resources, resulting in resource starvation for other processes, and ultimately causing poor overall performance. To address this issue, we develop an adaptive resource control mechanism to adjust and allocate resources to different processes. Intuitively, when several of the preferred control plane instances already achieve the desired objectives and performance, the resources allocated to other control plane instances (e.g., CPU and control channel bandwidth) can be reduced. The details of the resource control will be described in Section 4.2.

2.5 *Carbide* Workflow

We now present the workflow to illustrate how *Carbide* works. We consider the example in Figure 2. Assume each device in the network runs three CPs, with CP_1 being an SDN control plane receiving OpenFlow message from a recent release of an SDN controller, CP_2 being an SDN control plane receiving OpenFlow message from a stable release of an SDN controller, and CP_3 being OSPF. We also assume the requirements specified in Table 1. The three flows f_1, f_2 and f_3 are required to traverse a firewall located at B. The following steps are also labeled in Figure 1.

- **Step ①**: Through the *CPSpec* component, administrators specify the requirements for each CP, and the preference order of the CPs. For example, an administrator may want to implement the requirements in Table 1, and prefer CP_1, to CP_2, to CP_3. The requirements and preferences are then distributed across all devices.
- **Step ②**: As the CPs run, they gradually populate their vFIBs with forwarding rules.
- **Step ③**: As the vFIB of each CP_i evolves, the associated *CPCheck* module verifies whether each of the requirements specified by the administrator in Step 1 can be satisfied.
- **Step ④**: The CP composer takes results from the different *CPCheck*, and the vFIB contents as inputs to configure the data plane. Specifically, the CP composer creates rules for packets to be assigned to the most preferred CP whose verification result returns true. For example, after the network converges, packets with $srcIP = A, dstIP = D, dstPort! = 23$ will be processed by CP_1 along the path ABD.

To illustrate how *Carbide* responds to failure events, we now consider what happens when link BE fails, causing the network to become partitioned into two parts. *Carbide* allows devices in the lower partition to discover their loss of connectivity from the SDN controllers. Specifically, device B first detects the link failures, and triggers the *CPCheck* module to update its verification results and notify the *CPCheck* modules at its peers (e.g., A) of the changes. Then, the *CPCheck* modules at different devices (e.g., in A and B) detect that destinations in the upper partition are no longer reachable. As a result, the CP composers update the data plane configuration to remove those corresponding rules (e.g., f_1). In contrast, the rules corresponding to destinations in the same partition still remain. Assuming the connectivity with the SDN controllers is not resumed, the OpenFlow rules will timeout and get deleted. Consequently, the network will gradually switch to OSPF. To comply with the correctness requirements in Table 1 specified by the administrator, the CP composer inserts additional rules in the data plane configuration. For example, when switching to OSPF, it inserts a selection rule match $port! = 23$ to comply with the reachability and traffic blocking requirements. For the firewall traversal requirement, the local verification process at A can infer that the route to C (i.e., f_3) calculated by OSPF does not satisfy the requirement. As a result, the CP composer insert additional rules to drop packets that match $dstPort = 80, dstIP = C$. Traffic from A to D with dst port 80 (i.e., f_2) can still satisfy the firewall traversal requirement. This example demonstrates how *Carbide* can locally recover from failures while ensuring correctness.

```
requirement →  (property, DHS)
             | (requirement & requirement)
   property →  (property | property)
             | (property & property)
             | (¬ property)
             | list(hop)
        hop →  [d = D] (device specifier)
             | . (any device)
             | * (zero or more of previous hop)
             | ^ (beginning of path)
             | $ (end of path)
             | < (local device)
             | > (packet destination)
        DHS →  list(wildcard)
   wildcard →  p ∈ {0, 1,*}^L
```

Figure 3: *ReqLang* specification

3 VERIFYING CONTROL PLANE CORRECTNESS

Verifying correctness of each control plane is crucial to *Carbide* as it allows *Carbide* to perform its composition systematically to ensure desired correctness properties for each packet in the network. In this section, we will first present abstractions to allow for specification of general network requirements. Then we will describe *CPCheck*, a novel real-time distributed verification engine.

3.1 *ReqLang*: Network Requirement Specification

Requirement and property abstractions. First, we provide a formal framework for requirement specification in *Carbide*.

Definition 1 (Property). *A property is a boolean predicate on the set of possible paths in a network. A control plane CP is said to satisfy a property for a given packet if the path specified by CP for the packet satisfies the property.*

Definition 2 (Requirement). *A requirement is a sequence of* $(prop_i, DHS_i)$ *pairs, where each* $prop_i$ *is a property and each* DHS_i *is the desired header subspace for* $prop_i$.

Examples of policies may be *Ensure reachability for all traffic except on port 23* or *All traffic with destination d must pass through a load balancer at device m.*

Requirement specification grammar. To allow network operators to specify custom correctness requirements, we provide a policy specification grammar to specify a broad range of requirements, shown in Figure 3, inspired by FlowExp [13]. Each property is expressed as a sequence of matches on path hops, and each requirement is expressed as a sequence of `(property, headerSpace)` pairs.

Localization of requirements. We now describe a method of converting this requirement specification into local requirements at each device. The purpose of this conversion is twofold: (1) *Carbide*'s CPSpec can reduce each property in *ReqLang* into a subset of regular expressions, allowing us to use generic regular expression parsers for local verification, and (2) *Carbide* can refine the quantity and specification of requirements by utilizing knowledge on which flows must pass through which device. *Carbide* performs this by localizing each `(property, headerSpace)` pair.

To achieve (1), *Carbide* must convert the local device match $<$ and packet destination match $>$. For a given destination D and device N, it can expand each $<$ and $>$ into the device specifies $[d = N]$ and $[d = D]$, respectively. To achieve (2), we consider the user-provided device specifiers $[d = N]$. For each such specifier, device N need not be aware of matches before the first $[d = N]$ identifier. Furthermore, we observe that only *ingress* nodes need to be aware of the full path constraint, as they have the full view of the path to be taken by a packet, and ingress nodes need only enforce each property on packets for which it is the source. Section 3.2.3 demonstrates localization and correctness computation of several common requirements.

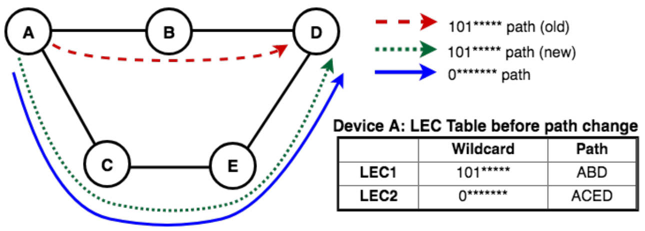
3.2 Correctness Verification Using *CPCheck*

Now we will describe the design and operation of *CPCheck*, a novel distributed verification algorithm. The benefits of *CPCheck* are twofold: (1) by allowing for *purely distributed* verification, *CPCheck* allows *Carbide* to enforce control plane requirements in spite of network partitions and failures, and (2) real-time verification allows *Carbide* to quickly assign control planes to incoming packets in response to network events. To achieve (1), we design lightweight data structures and novel algorithms to initialize and update them. To achieve (2), we utilize two key observations: (a) most data plane events only affect small subsets of the packet space, and (b) most data plane events only affect small subsets of network devices.

CPCheck consists of three major components:

- A *Local Equivalence Class Table* (LEC table) at each device to store relevant forwarding information for downstream packets (§3.2.1).
- A distributed update algorithm to verify correctness of network events in real time (§3.2.2).
- A correctness computation engine to generate control plane forwarding assignments for incoming packets based on the forwarding behavior stored in the LEC table and operator-specified correctness requirements (§3.2.3).

3.2.1 LEC Tables. At each device, *CPCheck* consists of local data structures known as Local Equivalence Class Tables



Device A: LEC Table before path change

	Wildcard	Path
LEC1	101*****	ABD
LEC2	0*******	ACED

Figure 4: An example with two flows using *ABD* and *ACED* at beginning, and later one flow's path will be updated from *ABD* to *ACED*. The table shows the initial LEC table at *A*.

(LEC tables) which store the relevant forwarding behavior for outgoing flows at each device. To motivate this design, we first consider a naive distributed implementation of existing centralized verification systems (e.g., VeriFlow, NetPlumber) by replicating the global data structures on each device. Devices would maintain the data structures by broadcasting any local flow rule updates across the network, and each device would individually compute correctness as specified by the centralized verification system. However, this approach has two major limitations: (1) the memory requirements of these global data structures generally scale with the *total* forwarding rule space of the *entire* network, which does not scale well in larger networks, and (2) network devices often have much less computing power than a typical controller, which may hurt verification time.

To avoid these limitations, we observe that each device only needs to be aware of forwarding rules at subsequent devices that affect its outgoing flows. We can then trim the stored data by only considering the subset of network devices reachable by downstream flows. To illustrate these insights, consider Figure 4. Any forwarding rules at device C or device E will not affect outgoing flows from device B, so device B need not be aware of such forwarding rules. To take advantage of this, we will introduce the notion of Local Equivalence Classes (LECs).

Definition 3 (Local Equivalence Class). *Let CP_i be a control plane, and n be a device in a network. For two packets p_1 and p_2, let P_1^n and P_2^n be the paths traversed by p_1 and p_2, respectively, when processed by CP_i's forwarding rules beginning at n. We say that p_1 and p_2 are in the same* ***local equivalence class*** *if $P_1^n = P_2^n$.*

At each device, *CPCheck* partitions the packet header space into LECs, and by definition, each LEC will have a unique downstream forwarding path. This associative map between LECs and forwarding paths will form LEC tables, the core data structures at each device, and this data will provide a complete local context on which *CPCheck* can fully determine a given packet's forwarding behavior and thus allow us to query for general correctness requirements. Figure 4 shows the initial LEC table stored at device A.

To initialize the data structures, we utilize a vector-based algorithm, as shown in Algorithm 1. Each device initially only has access to local forwarding information, which may only consist of next-hops (e.g., EIGRP). The initialization algorithm allows *CPCheck* to aggregate next-hop information across devices in order to generate the LEC paths at each device.

3.2.2 *Distributed Real-time Updates.*

In this section we will describe a novel distributed algorithm to verify correctness in response to real-time data plane updates. Algorithm 2 shows the general update algorithm. We will first introduce the types of message utilized in *CPCheck*, then demonstrate an example workflow in response to a series of SDN flow rule updates.

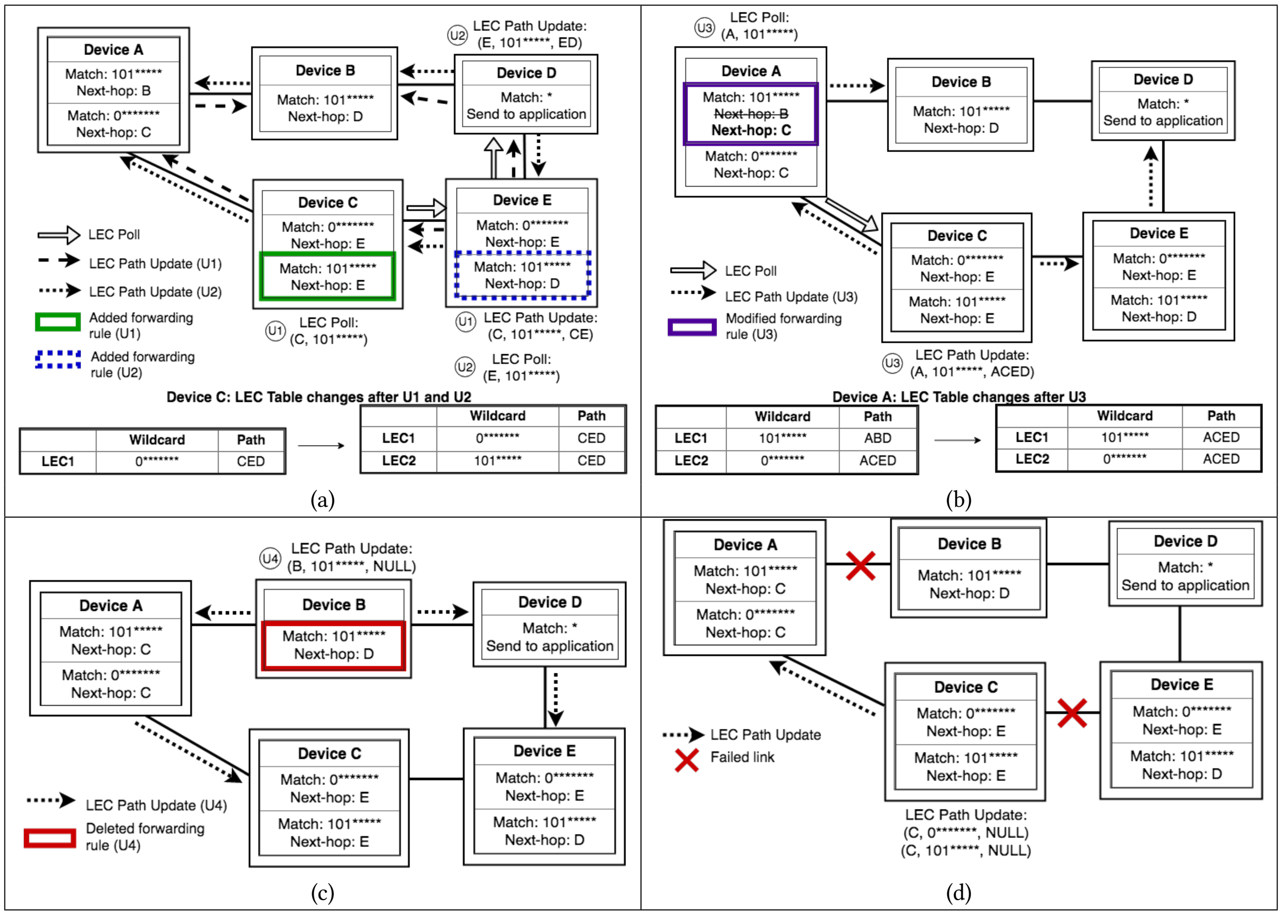
LEC path update messages. *CPCheck* detects network events (e.g., forwarding rule update, link failure, etc.), locally and generates an *LEC path update message*, (n, HS_{aff}, P), where n is the device at which the network event was detected, HS_{aff} is the header subspace whose behavior is affected by the update, and P is the new path taken by HS_{aff} starting at device n. This announcement is broadcast to all devices in the network. Upon receiving this announcement, each device then performs a local search to find affected LECs and update their path information. First, the device checks for all LECs with a *path dependency* on n, or in other words, all LECs whose forwarding paths contain n. In addition, the device checks for LECs with a *header space dependency* on HS_{aff}, or in other words, LECs that overlap with HS_{aff}. For each LEC that has both path and header space dependencies on the incoming announcement, the device then updates its path according to the new segment P.

LEC poll messages. In some instances, such as is shown in Figure 5, when a forwarding rule is added or modified, the local device may not have the needed downstream path P stored in its LEC table. In this case, the device sends an *LEC poll message* to the next-hop n' of the new rule to request the relevant path. Device n' has the complete information to construct the correct LEC path update, so it floods the update to the network directly.

We will now describe a realistic example to illustrate the behavior of *CPCheck* under various network events. Consider the topology shown in Figure 4, where SDN-specified routes are shown. In this example, the controller issues a series of updates to change the path taken by all packets matching 101^{*****}:

U1: Add $C - E$ forwarding rule
U2: Add $E - D$ forwarding rule
U3: Change $A - B$ forwarding rule to $A - C$
U4: Delete $B - D$ forwarding rule

U1-U2: Adding a forwarding rule. As shown in Figure 5(a), when device C receives its new forwarding rule <



	Wildcard	Path
LEC1	0*******	CED

	Wildcard	Path
LEC1	0*******	CED
LEC2	101*****	CED

	Wildcard	Path
LEC1	101*****	ABD
LEC2	0*******	ACED

	Wildcard	Path
LEC1	101*****	ACED
LEC2	0*******	ACED

Figure 5: *CPCheck* behavior in response to (a) new forwarding rules at C (update $U1$) and E (update $U2$), (b) a modified forwarding rule at A (update $U3$), (c) a deleted forwarding rule at B (update $U4$), and (d) a network partition.

101^{*****}, Next-hop: E >, it does not have the downstream routing information for 101^{*****} so it first sends an LEC poll to the rule's specified next-hop, E. Device E then calculates the new path segment for 101^{*****} and broadcasts the LEC path update $(C, 101^{*****}, CE)$ to all devices in the network. Upon receiving this update, device C notes that the device specified in the incoming updat message is itself and updates its LEC table with the entry $(101^{*****}, CE)$. Likewise, when device E detects its new forwarding rule, it sends a poll message to D, who in turn broadcasts the LEC path update $(E, 101^{*****}, CE)$. Device E updates its LEC table accordingly. At device C, the entry $(101^{*****}, CE)$ has both path and header space dependencies on the incoming update, so C updates its LEC table as well.

U3: Modifying an existing forwarding rule. Figure 5(b) shows the workflow in response to U2. Again, because A does not have the needed downstream forwarding information, it sends an LEC poll to C, who constructs and broadcasts the LEC path update. The resulting changes to the LEC table at A are shown as well.

U4: Deleting a forwarding rule. Figure 5(c) shows the deletion of the flow rule at device B. On detecting event U4, B generates the LEC path update $(B, 101^{*****}, NULL)$ and broadcasts it to the network. At this point, because device A has already changed its downstream path to $ACED$, it does not need to make any changes to its LEC table.

Network partition. On the event of a network partition, two portions of the network may be completely separated from each other. Furthermore, in the case of an SDN control plane, some devices may be unable to communicate with the SDN controller and therefore be unable to respond to the link failures. *CPCheck* allows these devices to quickly detect which devices are still reachable even in the absence of communication with the control plane. In Figure 5(d), device C detects the link failure of CE and notes that this affects both of its forwarding rules. C broadcasts an LEC path update message for the match sets of each forwarding rule, as depicted, and device A updates its LEC table accordingly. Device A also detects the AB link failure, but none of its forwarding rules depend on that link, so it sends no updates.

Thus device A can quickly detect its partition from D and E while recognizing reachability to C.

3.2.3 Correctness Computation & Output. In this section we will demonstrate several common network correctness requirements and enumerate the output of *CPCheck*.

Reachability. Consider the reachability requirement for CP_1 as shown in Table 1. This may be expressed in *ReqLang* as (`.*>$`, $\{p \mid p.dstPort \neq 23\}$), which will be localized to each device v as (`.*>$`, $\{p \mid p.dstPort \neq 23 \& p.srcIp = v\}$). *CPCheck* can then match the localized requirement with each path in the LEC table.

Traffic blocking. For the traffic blocking requirement of CP_1, we can utilize the boolean composition of properties in *ReqLang*, namely the negation of a property, by specifying the requirement as ($\neg$`.*>$`, $\{p \mid p.dstPort \neq 23\}$)

Reachability via a waypoint. Consider the firewall property for CP_1 in Table 1, which can be specified as (`.*[d=F].*[d=L].*$`, HS), where F represents the firewall, L represents the load balancer, and HS is the subset of packets where $srcIP = X, dstIP = Y, dstPort = 80$. *CPCheck* localizes this at devices X, F, and L as follows:

- Device X has an identical local requirement to the global requirement
- Device F has local requirement (`[d=F].*[d=L].*$`, HS)
- Device L has local requirement (`[d=L].*$`, HS)

Loop-freedom. It is often desired to ensure no routing loops for all traffic in the network. *ReqLang* can express this requirement as ($\neg$`.*<.*`, $*$), which gets localized to each device V as ($\neg$`.*[d=V].*`, $*$). Note that due to the distributed nature of *CPCheck* and the limitations of regular expressions, a device can only detect routing loops of which it is a part, not downstream routing loops.

***CPCheck* output:** (HS_i^T, HS_i^F). For a given control plane CP_i, the output from *CPCheck* is the set of packets HS_i^T for which CP_i provides correctness, as well as its complement, HS_i^F. To calculate HS_i^T, *CPCheck* performs two steps at each device using locally available information: (1) for each property $prop_j$, calculate S_j, the set of packets for which CP_i satisfies the localization of $prop_j$, and (2) aggregate the S_j's into the desired result. To perform (1), *CPCheck* can simply take the union of all LECs whose paths (as given in the device's LEC table) successfully match the localization of $prop_j$. To perform (2), let $HS_i^F = \cup_j(DHS_j \setminus S_j)$, the set of all packets for which CP_i fails to satisfy the requirements, and let $HS_i^T = (HS_i^F)^c$.

3.3 Verification Result Composer

Based on the output of *CPCheck*, *Carbide* provides a selection mechanism to determine a correct, usable control plane for each incoming packet. Composing control planes independently for different packet sets allows for greater flexibility in the usage of more preferred control planes. For example, in the case of network partition shown in Figure 5(b), while reachability is violated for $A-D$ flows, the control plane still offers reachability between A and B. A finer-grained composition at the level of packet spaces allows us to continue using the control plane for $A-B$ flows, even though it may not satisfy the requirements for other flows.

To perform this composition, we introduce the *Verification Result Composer*, which takes as an input the (HS_i^T, HS_i^F) results from *CPCheck* and outputs a CP assignment table, which maps sets of packets to the best usable control plane.

Algorithm 1: LEC Table Initialization

```
Data: Local forwarding rules at each device
Initialize associative map $T$
/* First we initiate messages for LECs for which
   we are the end of the route */
foreach flow rule $f$ do
    if $f.nexthop == f.dst$ then
        send LEC Path announcement $(n, f.match, n)$ to all neighbors
        $T$.insert($f.match, n$)
on event incoming LEC path announcement $n_i, HS_i, P_i$
    foreach local flow rule $f$ do
        if $f.nexthop == n_i$ then
            $EC_i \leftarrow f.match \cap HS_i$
            $T$.insert($EC_i, P_i$)
            send LEC Path announcement $(n, HS_{end}, n \cup P_i)$ to all neighbors
```

Algorithm 2: LEC Update Algorithm

```
Data: Associate map $T$ and Flow Graph $G$ as computed in initialization algorithm
Input: Modified flow rule $f$
on event modified local flow rule f
    $HS_{out} \leftarrow f.match$
    send LEC poll $(n, HS_{out})$ to device $f$.nexthop
/* on receiving LEC poll */
on event incoming poll message $n_i, HS_i$
    $HS_{all} \leftarrow T$.retrieveOverlapping($HS_i$)
    foreach $HS_i \in HS_{all}$ do
        broadcast LEC Path update $(n_i, HS_i, T$.getPath($HS_i$))
on event incoming LEC Path update $n_{in}, HS_{in}, P_{in}$
    foreach $LEC_i$ whose path $P_i$ contains $n_{in}$ do
        if $LEC_i \cap HS_{in} \neq \emptyset$ then
            $P_{new} \leftarrow P_i$.slice($0, n_{in}$) $\cup P_{in}$
            $T$.update($LEC_i, P_{new}$)
```

Implementation details of the assignment table can be found in §5.

4 *CARBIDE* IN PRACTICE

Due to the distributed nature of *CPCheck*, several complications may arise in practical settings. From a stability perspective, (1) a CP may oscillate between various routes, causing *CPCheck* to continuously recompute correctness results for the same updates, and (2) different devices may repeatedly choose different control plane assignments for the same packet, causing forwarding loops. From a performance perspective, (1) we may wish to opportunistically utilized centralized computing resources to improve efficiency, and (2) desired requirements may depend on finer-grained forwarding information not contained in routes. In this section we will provide extensions to *CPCheck* to address these issues.

4.1 Stability

Alleviating CP oscillation with damping. In some instances, a CP may oscillate between different routes due to hardware, software or configuration errors, leading to continuous repeated updates to the vFIB. Because *CPCheck* eagerly recomputes correctness results in response to any change in a control plane's vFIB, this may lead to excessive message and computation overhead. Furthermore, due to possible oscillations in *CPCheck* output, a packet's control plane assignment may oscillate as well.

Inspired by route flap damping in in BGP, we can deploy a CP oscillation damping mechanism to address this issue. Each (LEC, CP) is assigned a penalty value which is increased by a fixed parameter (e.g., 1000) every time a verification update is triggered. If the penalty value exceeds a *suppress limit* (e.g., 3000), any subsequent vFIB updates will not trigger *CPCheck*. The penalty value also decays exponentially according to a *half-life time* (e.g., 15 minutes) as long as no *CPCheck* updates are triggered. If the penalty for a suppressed (LEC, CP) pair falls below a *reuse limit*, *CPCheck* will resume responding to subsequent vFIB updates. The suppress limit, half-life time, and reuse limit may all be configurable. This mechanism reduces the impact of oscillating vFIB updates on *CPCheck* without affecting the reaction time and availability of stable routes.

Avoiding cross-CP loops. Using the requirements specified in §3.2.3, *CPCheck* can detect forwarding loops within a single CP. However, because *Carbide* allows different devices to process a given packet with different control planes, infinite loops may still be possible, as shown in Figure 6(a). However, we observe that this example may only occur when devices A and B obtain different results from *CPCheck* for the same control plane, causing different CP assignments for the same packet. To understand the problem, we introduce the

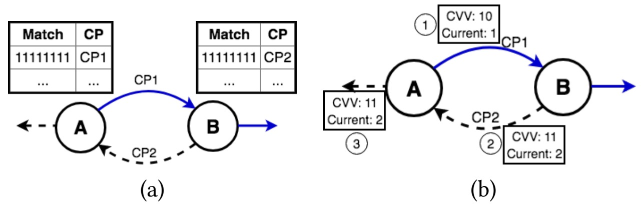


Figure 6: (a) Example of cross-CP forwarding loop. (b) Demonstration of cross-CP loop avoidance using CVV. ① An incoming packet at device A is forwarded using CP_1 and is tagged accordingly. ② At device B, CP_2 is usable because the second bit of the incoming CVV vector is unset, so the packet is forwarded using CP_2 back to A. ③ At A, CP_1, the CVV tag makes CP_1 unusable, forcing A to forward using CP_2 and breaking the cross-CP loop.

notion of *transitive requirements*. If a network consists only of transitive requirements, then no cross-CP loops can occur. Examples of transitive requirements include reachability, traffic blocking, and device avoidance.

Definition 4 (Transitive requirement). *For a given CP_i, consider a requirement of the form $(prop_i, DHS_i)$, and let $pr_v(prop_i)$ denote the localization of $prop_i$ to device v. We say that this requirement is transitive if for every device n and packet p such that CP_i satisfies $pr_v(prop_i)$, CP_i also satisfies $pr_w(prop_i)$, where w is the next-hop for p as specified by CP_i. Otherwise, CP_i is intransitive*

Proposition 1. *If all requirements in a network are transitive, then no cross-CP loops can occur.*

However, in the presence of intransitive properties, cross-CP loops may occur. For example, single-CP loop-freedom, as defined in §3.2.3, is intransitive. To remedy this issue, we introduce a lightweight tagging mechanism on each packet consisting of two fields: (1) an 8-bit **Control Plane Visiting Vector (CVV)** field where the ith bit is set if and only if CP_i was used at any point to forward the packet, and (2) the **current CP** denoting the CP most recently used to forward the packet. We enforce the restriction that each device may use CP_i for an incoming packet if (1) the current CP tag is set to i or (2) the ith bit of the CVV field is not set. This guarantees that infinite switching between control planes will not occur, which will prevent the cross-CP loops discussed.

Because the tags are attached to each incoming packet, we can implement this selection process by refining the CP assignment table.

4.2 Control Plane Resource Management

Carbide consists of a large number of processes competing for different resources (e.g., CPU, memory and I/O) within a device, and resource allocation among these processes is therefore critical. For example, if a single CP verification process consumes all of a device's execution resources, other processes would starve and not be able to progress.

In particular, when an event happens (e.g., link failure), and multiple *CPCheck* processes are launched, and competing for resources, how should the resources be allocated? Intuitively, important processes should be assigned more resources. However, how should importance be defined in this specific context? In addition, uncertainty can further exacerbate the problem. For example, if more resources are assigned to a verification process which after running for a period of time returns False, would such resources have been better assigned to a different process?

To address these issues, we developed an adaptive resource allocation algorithm that takes into account both importance and uncertainty. Importance is defined based on three observations. First, we note that a verification process starts when a CP detects a change in a vFIB. The verification process goal consists in verifying whether that CP can satisfy the correctness requirement for a LEC. Each verification process therefore is associated with a CP, and the first factor to consider is the global preference order between the CPs as specified by the administrator. Second, the size of the LEC for which the verification process is launched represents the size of the packet header space that the rule directly affects. The large the size, the more important the corresponding verification process is. Finally, the third observation is that if other CPs (e.g., SDN) can already satisfy the correctness requirement for a LEC, the need for a CP (e.g., OSPF) to satisfy it becomes less urgent. This can be captured by a weight that is inversely proportional to the number of other CPs that can satisfy the correctness requirement for that LEC. Finally, uncertainty is captured by the probability that the verification process launched by a specific CP for a particular LEC returns True. The product of those four factors becomes the utility functions of a verification process; and we then simply allocate the resources to the different verification processes by maximizing the sum of their utilities subject to the total amount of resources available. Details of adaptive resource allocation is described in Appendix.

4.3 Optimization Extensions

Opportunistic offloading to a centralized process. In the 3-CP example described in Section 2, two out of the three layers involve a centralized controller from which most control plane updates originate. Thus, we can extend *CPCheck* to greedily take advantage of the controller's global network view and greater computation power, when available, for three main steps: (1) initialization, (2) LEC path update broadcast, and (3) correctness computation. First, the initialization process extends well to centralization; when the SDN controller initializes flow rules for each device, it can also pre-compute the relevant local ECs, local data structures, and initial verification results and install this information at each device. Second, for LEC path updates, *CPCheck* can leverage the fact that in SDN control planes, most network updates are initiated through, or processed by, the controller. The controller can then maintain a copy of each device's LEC tables and compute LEC path updates for all controller-handled events and send them to each device. Third, we note that with large numbers of properties, the correctness computation time may become a burden for switches. We can extend the *CPCheck* so that each device uses the controller as a querying engine and only computes correctness locally if disconnected from the controller.

Generalization of the datapath. *ReqLang* expresses each LEC's behavior in terms of paths through the network. One could extend this data by storing the sequence of actions applied to each LEC to provided more fine-grained requirement specification. For example, an operator may want to require that packets from a certain source are not modified. An extension of *CPCheck* which stores sequences of actions, along with an extension of *ReqLang* which allows for regular expression matches on types of actions, would allow for both specification and enforcement of such a requirement. Furthermore, in instances when a packet is modified along the route, *CPCheck* can use the sequence of actions of an LEC to calculate header space dependencies by applying the actions specified on each LEC and comparing against incoming LEC path updates.

5 IMPLEMENTATION

We develop *MultiJet*, a switch OS level verification software suite and an evaluation framework which supports hybrid control planes, to implement the system described in Section 2. *MultiJet* is mostly written in Python 2.7. *MultiJet* is designed in a data driven model by realizing a single in-memory data store to decouple modules. This architecture makes *Carbide* highly extendable. *MultiJet* is mainly composed of 4 components, as shown in Figure 7:

- **In-memory datastore.** A key-value datastore that stores *local forwarding rules* and *verification results* for all control planes, and exposes *Get/Put/Delete/Listen* APIs for manipulating data and listening to data change events. We implement the datastore in Redis for performance optimization.
- **Verification thread pool.** A thread pool that executes *CPCheck* for each control plane. It listens to forwarding rule change events from the *in-memory datastore* to trigger verification updates. All running threads in the pool share a single datastore but are labeled by control plane index. We use a thread pool here for 2 main reasons: (1) there may be multiple messages received from neighbors simultaneously due to LEC path update flooding. We must handle them concurrently and reduce the overhead for starting a new thread, (2) the thread pool size can be adjusted at

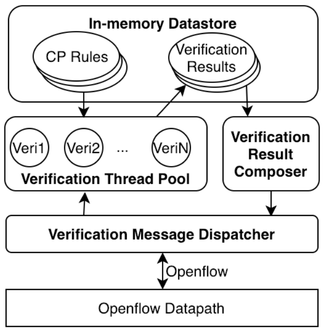


Figure 7: The Architecture of MultiJet.

any time to control the performance of verification (e.g., one could reduce the thread pool size to limit the resource usage). Finally, the verification result will be send to the *in-memory datastore*.

- **Verification result composer.** A component that listens and reads the verification results in *in-memory datastore* and does composition of verification results, then generate the CP assignment table which will be sent to data plane eventually.
- **Verification message dispatcher.** A local Openflow controller application built on top of the Ryu framework[24] that collects and dispatches messages for the *Verification Thread Pool*. It uses Openflow v1.3 to communicate with the Openflow datapath. Verification messages are encapsulated into IPv4 packets with IP protocol number 143.

We deploy *MultiJet* on two real Openflow white-box switches to validate the feasibility. Their models are Pica8 P-5401 and Dell Z9100-ON. These two white-boxes are included in our testbed with virtual switches for evaluation. The testbed is built based on Docker [1]. We run Quagga [22] and OpenvSwitch [20] daemons in each docker container which mounts the *MultiJet* verification software suite directory.

6 EVALUATION

This section demonstrates the benefits of *Carbide*. We conduct experiments on a large real network topology with white-box switches included. First, we perform extensive micro-benchmarking measurements and show that the local lightweight distributed real time verification process can be completed locally at each device in less than 2.5 ms. In fact, 95% of the local verifications can even be completed in under 1ms. Second, we evaluate the effectiveness of *Carbide*. We show that on average, *Carbide* can recover from failures even 43% faster than the control plane with the lowest downtime. Also, in the presence of network partitions, *Carbide* not only guarantees correctness (e.g., no flow violates security requirements), but also allows larger amount of traffic to satisfy the requirements specified by the administrator (e.g., HTTP traffic must traverse a firewall). In other words, *Carbide* significantly increases reliability, and correctness.

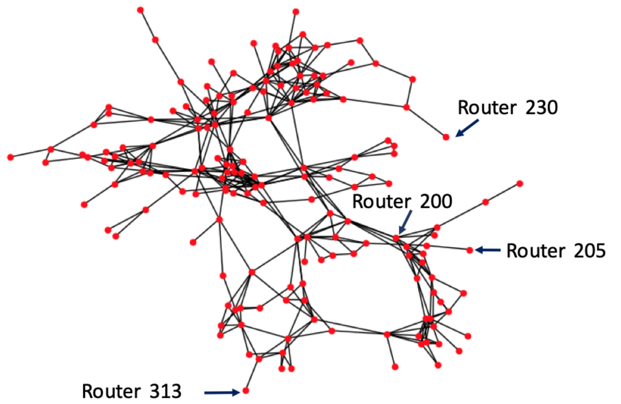


Figure 8: The Rocketfuel topology.

6.1 Methodology

We run all experiments in a virtualized environment + real white-box switches. The host server is a dedicated server with 2 Intel Xeon 8168 CPU (2.70GHz) which has 192 cores in total and 384GB memory. We conduct the experiments on a Rocketfuel topology [3] (AS 1755) consisting of 172 routers (per-port buffer size 208k) and 381 links as shown in Figure 8. Two of the routers are replaced with the white-box switches. Each of the rest network device runs as a separate Docker container, with two CPs: a Quagga OSPF, and an SDN. The SDN CP receives OpenFlow messages from a Ryu 4.24 controller. OSPF can locally compute paths which are then synchronized to containerized OpenvSwitch. *MultiJet* software suit runs in each container as a daemon to conduct real-time verification. The bandwidth of the links is set to 500Mbps.

6.2 Micro-Benchmarking *CPCheck*

The goal of this first set of experiments is to evaluate the efficiency, and computation time of *CPCheck*. We initialize *CPCheck* using the distributed LEC Table Initialization algorithm for the selected Rocketfuel topology. Because the performance of *CPCheck* depends greatly on the number of LECs each device has to search through, we provide data in Figure 9(a) to show the number of LECs calculated at each device during the initialization process. After initialization, we can see that most devices have between 380 and 420 LECs, with 396 on average. The distributed LEC Table Initialization algorithm was benchmarked on the Rocketfuel topology and took a total of 69.876 seconds to complete, generating a total of 67907 LEC table entries across the network.

Figure 9(b) shows the computation time required from each device to calculate and update LECs during *CPCheck*'s initialization process. Our evaluation log showed 232,832 total verification messages flooded in the entire network during initialization over 70 seconds. As the CDF shown in

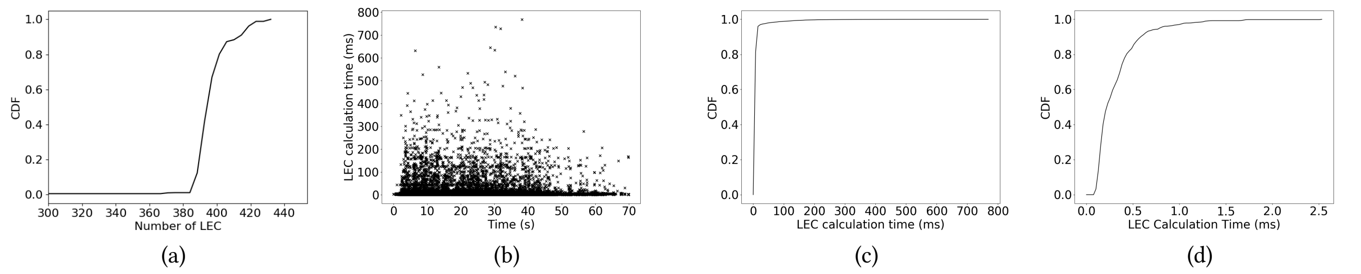


Figure 9: Micro-benchmarking *CPCheck*. (a) The CDF of LEC in network in initialization. (b) The computation time of LEC during *CPCheck*'s initialization process. (c) The CDF of computation time of LEC in network when initializing. (d) The CDF of local calculation time when in real-time updates.

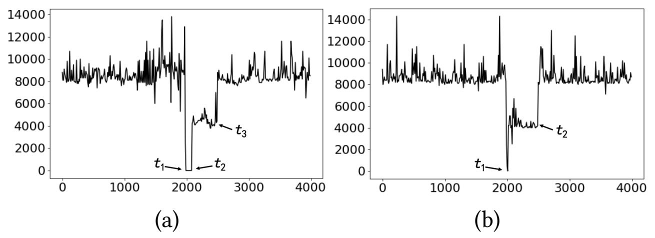


Figure 10: Packet receiving rate for the fast recovery experiments. The failure in (a) affects both the SDN and OSPF CPs, while that in (b) affects only the current SDN.

Figure 9(c), over 95% of LEC calculations completed in 40ms, with an average calculation time of 6.525ms.

In response to real-time updates, *CPCheck* computed LEC table updates much more quickly. Figure 9(d) shows the time between a given device receiving an LEC path update to when it finishes updating its LEC table. Over 95% of such calculations including LEC table updates and correctness computations completed in under 1ms.

6.3 Failure Recovery

The goal of this set of experiments consists in evaluating the effectiveness of *Carbide* in recovering from link failures.

Methodology. We randomly select a pair of nodes (e.g., router 313 to router 230) and generate UDP traffic between them by running iperf [2] at a rate of 100Mbps. For the SDN CPs, we implement the algorithm in BwE [17] to compute the paths. We then randomly select links to fail on one of the paths used by the UDP traffic. For the SDN CPs to recover from the failures, we implement a reactive approach. That is, when a failure happens, the device that detects it sends a control message to the controller to recalculate alternative forwarding paths and update the vFIB of the affected devices accordingly [25]. For each run, we measure the downtime defined as the amount of time between the moment the destination stops receiving packets because of the failure to the moment the receiver starts receiving packets again. We repeat this process for 10 runs.

	SDN	*Carbide*	OSPF
Average downtime	112.861ms	8.492ms	22.572ms

Table 2: Average downtime of using different systems.

Results. Figure 10 illustrates some of the patterns we observed: First, in Figure 10(a), the failure affected both SDN and OSPF CPs. In other words, both CPs used the failed link for the next hop. As such, at t_1, no valid route can be used, leading to a throughput of 0. Then, after OSPF discovers an alternate path, *Carbide* uses OSPF to forward traffic, and the throughput starts increasing at t_2. Finally, after the SDN recovers the route, the device uses the OpenFlow rules to forward traffic, and the rate increases further starting from t_3. In Figure 10(b), the link failure affects only the SDN CP. The device can immediately switch to OSPF to continue forward the packets.

Table 2 summarizes the average downtimes of the two CPs, and that of *Carbide* across the runs. While the average downtime of the SDN CP is 112.861ms, the average downtime of OSPF is 22.572ms, and that of *Carbide* is even lower than both of them with 8.492ms. The results show that *Carbide* can reduce the downtime of SDN by more than an order of magnitude, and that of OSPF by 43%. While one may be surprised that on average, *Carbide* can recover faster than OSPF, the main reason is that in some runs, the SDN CP recovers faster, whereas in other runs, the OSPF CP recovers faster; and in every case, *Carbide* switches to the one that recovers the fastest.

6.4 Enforcing Network Requirements

The goal of this set of experiments consists in evaluating the effectiveness of *Carbide* in ensuring correctness requirements.

Methodology. We assume that the administrator wants to enforce waypoint routing. Specifically, all traffic to a specific destination (router 205) must traverse a firewall (at router 200). For the SDN CP to generate paths that are compliant with such requirement, we implement the algorithm in [19].

	SDN	*Carbide*	OSPF
Before partition	171/171 (100%)	171/171 (100%)	105/171 (61%)
After partition	72/122 (59%)	115/122 (94%)	69/122 (56%)

Table 3: Fraction of paths that can reach router 205 after traversing waypoint at router 200.

We then gradually fail random links in the middle of the topology, until a subset of nodes get partitioned and lose connectivity with the SDN controller. We focus on nodes in that partition, and considering the paths from every node in that partition to the destination router 205, we compare the fraction of paths that can traverse the firewall at router 200 and still complies with the requirement. For the SDN CP, although the controller is no longer reachable, packets are forwarded along the existing rules. For OSPF, this CP cannot enforce waypoint routing. Instead, when the computed path from a source to the target destination traverses the desired waypoint, the path is considered to satisfy the requirement. In contrast, when the computed path from a source to the target destination does not traverse the desired waypoint, the path is considered to violate correctness. Finally, with *Carbide*, given a source, if the path provided by the SDN CP satisfies the correctness requirement (i.e., traverses the waypoint), the path is selected. Else, if the path provided by the OSPF CP satisfies the correctness requirement (i.e., traverses the waypoint), *Carbide* selects that path. Otherwise, the packets are dropped.

Results. Table 3 shows the fraction of paths that satisfy the requirement. Before the network partition, the network consists of 171 source nodes. Using SDN, all of the paths satisfy the requirement. Similarly, with *Carbide*, the requirement is satisfied for all paths. In contrast, with OSPF, only 61% of paths to destination 205 traverse the waypoint at router 200.

After network partition, the component disconnected from the SDN controller consists of 122 nodes. Using SDN, only 59% of the paths satisfy the requirement. This is because the OpenFlow rules may be obsolete, and may forward packets along invalid paths and into blackholes. Using OSPF, the fraction of paths that satisfy the requirement drops to 56%, and the remaining 44% would still allow packets to reach destination 205 but without traversing the waypoint 200, and would thus violate correctness. In contrast, with *Carbide*, 94% of the paths satisfy the requirement, and for the remaining 6%, their traffic would be dropped. As such, *Carbide* guarantees correctness for all the paths. In summary, both *Carbide* and the SDN CP guarantee correctness whereas the OSPF CP may violate it. And, between *Carbide* and SDN, *Carbide* allows a larger number of paths to satisfy the requirement given that it can also use the paths not only from the SDN CP but also from other CPs when they satisfy the requirements.

7 RELATED WORK

Recent work on network verification, and SDN reliability, focuses on the following directions:

Centralized verification. A number of methods have been proposed to verify network behaviors. HSA [14] provides a general framework to statically check network specifications based on a snapshot of network state. NetPlumber [13], Veriflow [15], and Delta-net [10] can detect violations of network-wide invariants in the data plane of SDN networks in real-time. Minesweeper [4] and ARC [5] verify the configuration files of distributed control planes in an offline manner, by encoding the problem into logic formulas that can be checked for satisfiability by constructing a Binary Decision Diagram or calling an SAT/SMT solver. ATPG [28] and NetSight [6] verify and diagnose the run-time data plane by debugging the traces and histories of probing packets. This line of work relies on a centralized entity to collect network states or configuration files, making the system less resilient to device or link failures. *Carbide* provides greater robustness by utilizing a distributed verification algorithm while opportunistically utilizing centralized resources when available.

Hybrid SDN. Tilmans, et al. [26] proposed an architecture, "IGP-as-a-Backup to SDN", in which each device contains a local software agent that runs an IGP as a backup to traditional SDN routes. However, the approach only guarantees reachability. It does not support requirements such as isolation, and waypoint routing. Vissicchio, et al. [27] provide a classification of routing protocols based on their interactions with the Routing Information Base (RIB) and Forwarding Information Base (FIB) of network devices, and identify sufficient conditions to prevent routing loops and black holes. However, their analysis does not provide guidance or mechanisms for guaranteeing other requirements such as waypoint routing or route symmetry either. In contrast, *Carbide* introduces a light-weight distributed verification module in the online composition layer to supporting a broad range of correctness properties for each packet in the network.

Multi-controller deployment. To avoid single-point-of-failure, recent SDN architectures propose multi-controller deployment where devices disconnected from one controller can get served by other ones [16, 7, 8, 23, 11]. Deployment of multiple controllers in SDN architectures increases network reliability, but can still be vulnerable in the event of network partition. Our work is complimentary to the multi-controller deployment , where the backup controllers can be considered as one of the CPs as well in *Carbide*.

REFERENCES

[1] Docker container platform. https://www.docker.com.
[2] Iperf, a tool for active measurements of the maximum achievable bandwidth on ip networks. https://iperf.fr/.

[3] Rocketfuel: An isp topology mapping engine. http://www.cs.washington.edu/research/networking/rocketfuel.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168. ACM, 2017.

[5] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 359–373. ACM, 2017.

[6] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, volume 14, pages 71–85, 2014.

[7] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.

[8] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

[9] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87. ACM, 2018.

[10] Alex Horn, Ali Kheradmand, and Mukul R Prasad. Delta-net: Real-time network verification using atoms. In *NSDI*, pages 735–749, 2017.

[11] Yannan Hu, Wang Wendong, Xiangyang Gong, Xirong Que, and Cheng Shiduan. Reliability-aware controller placement for software-defined networks. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 672–675. IEEE, 2013.

[12] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[13] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[14] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, volume 12, pages 113–126, 2012.

[15] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 15–27, 2013.

[16] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[17] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 1–14. ACM, 2015.

[18] Amund Kvalbein, Audun Fosselie Hansen, Stein Gjessing, and Olav Lysne. Fast ip network recovery using multiple routing configurations. In *in INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings.* Citeseer, 2006.

[19] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 15. ACM, 2014.

[20] OpenvSwitch. https://www.openvswitch.org.

[21] 2016 Cost of Data Center Outages Report. https://datacenterfrontier.com/cost-of-data-center-outages/.

[22] Quagga. https://www.quagga.net.

[23] Francisco J Ros and Pedro M Ruiz. On reliable controller placements in software-defined networks. *Computer Communications*, 77:41–51, 2016.

[24] Ryu. Component-based software defined networking framework. https://osrg.github.io/ryu/.

[25] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. A demonstration of fast failure recovery in software defined networking. In *International Conference on Testbeds and Research Infrastructures*, pages 411–414. Springer, 2012.

[26] Olivier Tilmans and Stefano Vissicchio. Igp-as-a-backup for robust sdn networks. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 127–135. IEEE, 2014.

[27] Stefano Vissicchio, Luca Cittadini, Olivier Bonaventure, Geoffrey G Xie, and Laurent Vanbever. On the co-existence of distributed and centralized routing control-planes. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 469–477. IEEE, 2015.

[28] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.

APPENDIX

A CONTROL PLANE RESOURCE MANAGEMENT

Carbide consists of a large number of processes competing for different resources, and resource allocation among these processes is therefore critical. For example, if a single CP verification process consumes all of a device's execution resources, other processes would starve and not be able to progress. To address this problem, we propose an *adaptive resource allocation* which ensures that important processes receive enough resources, and *Carbide* can operate properly, and efficiently. First, We present the control model for the resource allocation. Then, we describe the resource optimization problem, and present the adaptive resource allocation algorithm.

A.1 Control Model

To describe the control model, we consider a device running *Carbide*, and we assume it to run n control planes. As a result, that device actually runs three types of processes: (1) a CP process type for each of the n control planes, (2) a CP composer process type for the CP composer, and (3) a Veri process type which is launched by a CP to verify the correctness requirement for a LEC upon detecting changes in the vFIB.

LEC	Wildcard Exp.
LEC_1	11******
LEC_2	101*****
LEC_3	100*****
LEC_4	0*******

(a)

Packets	CP
LEC_3	CP_1
LEC_4	CP_1
others	drop

(b)

s_{ij}	CP_1	CP_2
LEC_1	U	U
LEC_2	U	U
LEC_3	T	U
LEC_4	T	F

(c)

Figure 11: A device with two CPs and four LECs. (a) shows the wildcard matches for each LEC. (b) shows the CP assignment table (c) shows the state of system after an event affecting LEC_1, LEC_2, and LEC_3.

The CP processes, and the CP composer process run continuously, and are assigned fixed amounts of resources. This is because the CP composer is crucial and at the core of *Carbide*. For example, all communication between the control plane and data plane is controlled by the CP composer as shown in Figure 1. The CP processes are also assigned a fixed amount of resources because our evaluation showed that CP processes consume very limited amount of resources, especially compared to those consumed by the Veri processes.

In contrast, the Veri processes have shorter lifetimes, and are assigned dynamic amounts of resources determined by the adaptive resource allocation. A Veri process starts when a CP detects a change in a vFIB. The Veri process goal consists in verifying a correctness requirement for a LEC. It terminates by returning a False or True value. Veri processes consumes three types of resources: CPU, memory, and network bandwidth. The adaptive resource allocation focuses on determining the resources allocated to each of the Veri processes, based on their importance.

A.2 Adaptive Resource Allocation

When an event happens (e.g., link failure), and multiple Veri processes are launched, and competing for resources, how should the resources be allocated? Intuitively, important processes should be assigned more resources. However, how should importance be defined in this specific context? In addition, uncertainty can further exacerbate the problem. For example, if more resources are assigned to a Veri process which after running for a period of time returns False, would such resources have been better assigned to a different process?

Example: To illustrate the problem, we consider the example in Figure 11. It represents the state at a device after a link failure. Figure 11(a) illustrates the four LECs present in the device. Figure 11(b) represents the current outcome of the CP composer. Figure 11(c) shows the correctness requirement status for the different LECs. For example, for LEC_4, CP_1 can satisfy the correctness requirement, but CP_2 cannot.

At that device, five Veri processes are currently competing for resources. Each process verifies a LEC for a CP, corresponding to a cell marked as "U" for Unknown status in Figure 11(c). For example, a Veri process is verifying whether CP_1 can satisfy the correctness requirements for LEC_1.

After the event, we have the verification state table shown in Figure 11c. First, we do not run any verification for LEC_4 since it is not affected. We observe that LEC_1 matches with 2^6 packets, LEC_2 matches with 2^5 packets, and LEC_3 matches with 2^5 packets. Thus, we deduce that verifying LEC_1 is important than verifying LEC_2 and LEC_3 since it has larger subspace (i.e., it matches with more packets). Moreover, LEC_2 should have priority over LEC_3 as CP_1 has already been verified and it can processes packets matching of LEC_3. Thus, we conclude that it is essential to allocate resources to the most important processes given the resource limitation. However, this raises two issues. First, how we can decide the importance of processes. Second, when we allocate resources to a verification process, we cannot guarantee that the configuration will be verified. That is, the result of verification can only be known after execution.

R	Total amount of available resource
n	Number of local equivalance classes
j	Number of control planes
LEC_i	Local equivalence class i
CP_j	Control plane j
$CPCheck_{ij}$	Verification process of LEC_i on CP_j
s_{ij}	Quaternary $(T, F, U$ or $E)$ state of $CPCheck_{ij}$
d_{ij}	Resource demand of $CPCheck_{ij}$
x_{ij}	1 if d_{ij} allocated to $CPCheck_{ij}$
p_{ij}	Probability of s_{ij} turns T from U
v_{ij}	Value of $CPCheck_{ij}$
r_{ij}	Reward of $CPCheck_{ij}$
b_j	Weight of a CP_j given by the user
$U_{LEC_i}(\{s_{ij}\})$	Utility function of LEC_i
$\mathcal{S} = \{s_{ij}\}$	State of the system
$\mathcal{X} = \{x_{ij}\}$	State of resource allocation
$U(\mathcal{S})$	Utility function of state $\mathcal{S}$

Table 4: Summary of notations used in resource allocation.

Problem Definition. We start with introducing the maximization problem for resource allocation. Table 4 provides a reference to the notations used in the formulation. Our evaluations show that the CP processes and CP-composer do not consume significant resources comparing to verification processes. Thus, our resource allocation model is based on verification processes. A verification process of a LEC on a CP can be in four states: Verified (T), Not verified (F), Undecided (U) (i.e., there is a new update on CP, and verification is not started yet), and executing (E). The set of all verification processes on a device forms the system state matrix $\mathcal{S}$.

Parameters. We define three parameters for a verification process $CPCheck_{ij}$: the resource demand d_{ij}, the probability of successful verification p_{ij}, and the user defined weight of the related control plane b_j. The first parameter, resource demand of a verification process, can be computed by profiling the previous executions. We use exponentially weighted moving average (EWMA) chart to estimate success probability of the verification processes. Moreover, the users assign an integer weight b_j to each CP_j for their importance. we define the value v_{ij} of a verification application $CPCheck_{ij}$ based on both the size of LEC_i and the weight of CP_j, that is, $v_{ij} = |LEC_i|b_j$. Finally, we define the reward r_{ij} of $CPCheck_{ij}$ as

$$r_{ij} = \begin{cases} p_{ij}v_{ij} & \text{if } s_{ij} = U \\ 0 & \text{otherwise} \end{cases}$$

If we consider the example in Figure 11 assuming the user assign weights for CPs as $b_1 = 2$, and $b_2 = 1$, the reward set will be $\{r_{1,1} = 128p_{1,1}, r_{1,2} = 64p_{1,2}, r_{2,1} = 64p_{2,1}, r_{2,2} = 32p_{2,2}, r_{3,1} = 0, r_{3,2} = 32p_{3,2}, r_{4,1} = 0, r_{4,2} = 0\}$.

Formulation. Finally, we state the resource allocation problem for R amount of resource as:

$$\begin{aligned} \text{maximize} \quad & U(\mathcal{S}) = \sum_{i=1}^{n}\sum_{j=1}^{k} r_{ij} \\ \text{subject to} \quad & \sum_{i=1}^{n}\sum_{j=1}^{k} x_{ij}d_{ij} \le R \end{aligned} \tag{1}$$

The maximization problem states that the system aims to have the maximum possible utilization for the given state $\mathcal{S}$ such that the total resource allocation does not exceed the available resource. Moreover, we can compute the utilization function of the system as the sum of utilization functions of LECs since all LECs are pairwise disjoint by definition.

The main purpose of adaptive resource allocation is to solve the maximization problem on an event-driven manner. We update the parameters after an event to compute real-time demand of each verification process. For example, if the size of a LEC decreases, its reward will also decrease.

Algorithm. *Carbide* requires to run adaptive resource allocation in every time slot to respond real-time resource demands. Thus, solving the maximization problem using integer programming is not possible due to its high time complexity. We introduce two intuition of adaptive resource allocation: First, we pick the CP with highest reward / demand ratio for each LEC. Second, if a verification processes of CP with higher weight already verified a LEC, it is unnecessary to run lower weight verification processes for that LEC.

s_{ij}, d_{ij}, r_{ij}	CP_1	CP_2
LEC_1	U, $3d$, 44.80(0.70 × 64)	U, $2d$, 25.6 (0.80 × 32)
LEC_2	E, $2d$, 0	F, 0, 0
LEC_3	F, 0, 0	U, d, 15.20 (0.95 × 16)
LEC_4	U, $3d$, 115.20(0.80 × 128)	T, 0, 0

Figure 12: State of the device in Figure 11 at the beginning of a timeslot t. Utility r_{ij} for the *CPCheck*$_{ij}$ having $s_{ij} = T$ or $s_{ij} = F$ are not calculated as they will not be given any resource. The values given in parentheses are probability p_{ij} and value v_{ij} of a process v_{ij}, respectively.

We give a greedy algorithm to allocation problem in Algorithm 3 based on the two main ideas of the adaptive resource allocation described above. The algorithm consists of two parts. First, it calculates the reward value r_{ij} for each verification process $CPCheck_{ij}$. Second, it allocates available resources to the first verification processes in U state for each LEC using knapsack algorithm. The items of knapsack have the weights d_{ij} and values $\frac{r_{ij}}{d_{ij}}$. The capacity of knapsack is the available resources, $R - allocatedR$. We use integer resource values to implement knapsack problem using dynamic programming.

Consider the example in Figure 11 with the state given in Figure 12. Suppose the total available resource is $8d$. The first iteration of the Algorithm 3 will compute r_{ij} values given in Figure 12. Since $CPCheck_{2,1}$ is running, the algorithm will allocate only $6d$ of the resources. In the first iteration of the second loop, $\mathcal{R}$ will have $CPCheck_{1,1}$ and $CPCheck_{3,2}$ and $CPCheck_{4,1}$ since $CPCheck_{3,2}$ and $CPCheck_{4,1}$ are the only verification processes in the state U having no preceding verified (T) verification process, $CPCheck_{1,1}$ has the highest reward demand ratio for LEC_1. When the knapsack algorithm is executed with values $\{\frac{44.80}{3d}, \frac{15.20}{d}, \frac{115.20}{3d}\}$, it will pick $CPCheck_{3,2}$ and $CPCheck_{4,1}$ because it is the combination giving the maximum value. In the second iteration, the remaining resource will be $2d$, and the set $\mathcal{R}$ will only consist $CPCheck_{1,2}$. The knapsack algorithm will allocate resources to $CPCheck_{1,2}$ as it demands $2d$. The algorithm will finish execution since all available resources are allocated.

Complexity. The time complexity of first iteration is $O(|\mathcal{S}|)$. Dynamic programming implementation of knapsack problem has the complexity $O(nR)$, where R is the available resource. Then, we can compute the worst case complexity of second iteration as $O(knR)$ assuming only one verification process is assigned resources per iteration. Notice that, the time complexity of integer programming of this problem scales with powers of nk as the system state has nk entries.

Algorithm 3: Adaptive Resource Allocation verification processes

```
Data: Updated state of resource allocation $\mathcal{X}$
Input: State of the system $\mathcal{S}$, the set of probabilities
       $\{p_{ij}\}$, the set of values $\{v_{ij}\}$, and total amount of
       available resource $R$
begin
    $allocatedR \leftarrow 0$
    /* Calculate the reward of incomplete
       verification processes. If a process is
       already running do not preempt.          */
    foreach $s_{ij} \in \mathcal{S}$ do
        if $s_{ij} == U$ then
            $r_{ij} \leftarrow p_{ij} * v_{ij}$
        else
            $r_{ij} \leftarrow 0$
            if $s_{ij} == E$ then
                $allocatedR \leftarrow allocatedR + d_{ij}$
    while $allocatedR \leq R$ do
        $\mathcal{R} \leftarrow \emptyset$
        for $i \leftarrow 1...n$ do
            if $\exists s_{ij} == U, \forall j' < j, s_{ij'} \neq T$ then
                $\mathcal{R} \leftarrow \mathcal{R} \cup \{CPCheck_{ij} | \max_{j'=j}^{k} \{r'_{ij}\}\}$
        if $\mathcal{R} == \emptyset$ then
            break
        /* Run knapsack algorithm to allocate
           resources and update $allocatedR$.     */
        $knapsack(\mathcal{S}, \mathcal{R}, \{d_{ij}\}, \{\frac{r_{ij}}{d_{ij}}\}, R, allocatedR)$
        foreach CPCheck$_{ij} \in \mathcal{R}$ do
            if CPCheck$_{ij}$ is in knapsack solution then
                $s_{ij} \leftarrow E$
            else
                $s_{ij} \leftarrow F$
```