USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# A Composable Emulation Framework for Whitebox Switches

Congcong Miao, *Tencent;* Xianneng Zou, *Tencent and Tsinghua Shenzhen International Graduate School;* Chuwen Zhang, *Tsinghua University;* Shiping Yang, Qihang Liu, Zhijie Yan, and Yanke Zhang, *Tencent;* Yong Jiang, *Tsinghua Shenzhen International Graduate School;* Qiao Xiang, *Xiamen University;* Xin Jin, *Peking University;* Zili Meng, *Hong Kong University of Science and Technology;* Ang Chen, *University of Michigan*

https://www.usenix.org/conference/nsdi26/presentation/miao-whitebox

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

# A Composable Emulation Framework for Whitebox Switches

Congcong Miao[1], Xianneng Zou[1,2], Chuwen Zhang[3], Shiping Yang[1], Qihang Liu[1],
Zhijie Yan[1], Yanke Zhang[1], Yong Jiang[2], Qiao Xiang[4], Xin Jin[5], Zili Meng[6], Ang Chen[7]
[1]*Tencent,* [2]*Tsinghua Shenzhen International Graduate School,* [3]*Tsinghua University,*
[4]*Xiamen University,* [5]*Peking University,* [6]*HKUST,* [7]*University of Michigan,*

## Abstract

High-fidelity network emulation is indispensable for reliable operation at scale. While existing emulators are built for monolithic, blackbox switches, emerging switch architectures are chipping away at this assumption. Cloud providers routinely source device modules from different vendors and compose them together to construct disaggregated, whitebox switches. This raises novel challenges in network emulation, as we must move from building bespoke firmware-based emulators to a *composable emulation framework* where diverse emulators can be stitched together for high-fidelity emulation: whether for data plane, control plane, or peripheral modules (e.g., optical components). We have designed the first such framework, MirSwitch, addressing key challenges in reconciling interface contract differences across emulator modules using an adaptor-centric approach. MirSwitch can faithfully emulate a wide range of functionalities of whitebox switches, while achieving 2.2× forwarding performance improvements which are critical for forwarding emulation. We have deployed MirSwitch in production network, which operators have used to troubleshoot 97.5% of a total of 203 issues.

## 1 Introduction

Network emulation [26, 31] is key to reliability, since cloud providers (e.g., Google, Microsoft) routinely use hundreds of thousands of devices [16, 17, 27, 34, 35, 43] to back their services. At this scale, many things can go wrong: hardware failures, software bugs, configuration errors [28, 42] all risk the stability of production networks, and even a small problem can lead to an outsized outage. While network verification [18, 19, 28, 42] can eliminate classes of problems, real devices often exhibit uncaptured behaviors deviating from the models. To capture and debug these granular behaviors, high-fidelity network emulators remain indispensable.

An important goal of emulators, therefore, is to faithfully reflect the underlying device characteristics. While in the past, emulators [26, 31] have been built for *monolithic, blackbox* switches, industry is trending toward *disaggregated, whitebox* switches that chip away at today's design. Up until recently, cloud providers (e.g., Google) purchase out-of-the-box devices from switch vendors (e.g., Cisco) with little or no customization. Their emulation, therefore, usually just requires hosting vendor-provided firmware in containers or VMs [26, 31]. Little insight is needed, or can be gained, into the device internals—whether the switching ASICs, control plane software, or peripheral modules.

Recent years have seen a drastic shift, however, where cloud providers desire deeper control over their networking substrate. Modules of a switching device are increasingly sourced from different vendors and *composed* together—not only to avoid vendor lock in, to reduce costs, but also to customize new features to pace up to fast-changing application demands. The resulting devices, therefore, are a composition of disaggregated modules: the switching ASIC may be sourced from one vendor (e.g., Broadcom), but it could be coupled with customized network operating systems (NOS) as the control plane (e.g., SONiC [44]), and further with power, optical, and heat dissipation modules all purchased from disparate sources. The specific composition further varies across devices even in the same fleet, depending on the generation of the network, topological locations, and supported services. This leads to unprecedented heterogeneity that challenges state-of-the-art emulators.

Existing emulators [26, 31] directly use off-the-shelf switch vendor images, which are one-size-fits-all blackbox binary files that cannot be customized. While these have been sufficient for previous-generation networks, they are a poor reflection of emerging switch architectures that emphasize composability. In our deployment, we have found many emulation needs that cannot be addressed unless each customizable feature is captured in sufficient detail. Broadly speaking, these emulation needs span three categories, for emulating *(i)* control plane features specific to each NOS version, *(ii)* data plane features not only ASIC functionality but also sufficient levels of performance fidelity, and *(iii)* peripheral modules to capture physical faults (§2.2).

To faithfully emulate the underlying architecture, we must transform bespoke emulation of each vendor firmware into

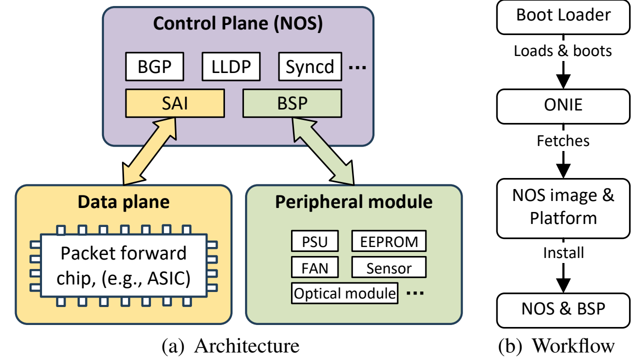
a novel, *composable emulation framework*, i.e., *MirSwitch*, that supports high customizability. The key challenge of this framework is to stitch together disparate emulated modules largely as-is, to minimize extra engineering effort needed for each scenario. We design generalizable principles that reconcile interface differences across modules, and *adapters* to realize these interface contracts, to support high heterogeneity. This translates to three technical challenges.

**Performant data plane emulation.** Traditionally, the forwarding performance of the data plane is regarded as out of scope—indeed, no emulation engine can faithfully capture Tbps-level throughput of switching ASICs. However, our experience shows that *sufficient* forwarding performance is key to emulating a range of useful properties. As an example, BFD (bidirectional forwarding detection) [2] behaviors will not be triggered unless the forwarding performance exceeds a certain threshold. Our design decision is to leverage advances from the past decade in network function virtualization (NFV): leading solutions such as VPP [11] use kernel-bypassing mechanisms and vectorized packet processing to achieve high performance. However, we need to rearchitect NFV designs to separate its data and control planes, exposing interface contracts, i.e., VPP adaptor, for the high-speed software processor to integrate with NOS control planes (§ 3.2).

**Emulating real-world NOS.** Popular NOSs like SONiC [7] use the switch abstraction interface (SAI [37]) to interact with the switching ASIC. However, since SAI in the production control plane is originally provided for physical chips, its interface contracts are different when composed with NFV solutions. We design another semantic adaptor to bridge this semantic gap, translating requests between the control plane and data plane with one-to-many mapping handling the translation of a single SAI request into multiple data plane (e.g., VPP) requests and many-to-one mapping converting several SAI requests to a single data plane request (§ 3.3).

**Peripheral module mockup.** Peripheral modules in the switch are likewise purchased from different vendors, exhibiting even higher heterogeneity. Each peripheral module is booted by its corresponding software driver in Board Support Package (BSP) [12] provided by specific vendor, so off-the-shelf and one-size-fits-all emulation images cannot capture such heterogeneity. We need to mock up these peripheral modules using their software drivers, but *without* the corresponding hardware modules. To reconcile this gap, we introduce an *environment* adaptor that creates a new mockup environment. It will generate a pseudo file to mock up the physical environment and virtualize software drivers from all vendors, thereby supporting composable emulation of diverse modules regardless of vendor sources (§ 3.4).

We conduct testbed experiments to evaluate the functionality and performance of MirSwitch. Compared to existing baselines, MirSwitch supports 89.5% of total functionalities of a standard whitebox switch [8, 9], which is 19.3% higher than existing approaches; it also improves the forwarding



(a) Architecture (b) Workflow

**Figure 1:** (a) Overview of the whitebox switch architecture and (b) workflow of booting a whitebox switch.

throughput by 2.2× to up to 256 Kpps, enabling new performance emulation capabilities. We then deploy MirSwitch to emulate our production network, where it can accurately reproduce the production network topology and packet forwarding paths, only with a small deviation of link utilization. Our experience of daily uses of MirSwitch shows it enables operators to troubleshoot 97.5% of 203 issues in a recent year (§ 4). We also report operational lessons throughout years of running MirSwitch in production (§ 5). To the best of our knowledge, we are the first to introduce a high-fidelity composable emulation framework customized for emerging, disaggregated whitebox switches.

This work **does not** raise any ethical issues.

## 2 Background and Motivation

We first provide an overview of a typical composition of whitebox switches (§ 2.1), then discuss the limitations of existing approaches to emulation (§ 2.2), and finally, identify several challenges to the design of a novel, composable emulation framework that supports high customizability (§ 2.3).

### 2.1 Disaggregated, Whitebox Switches

In recent years, cloud providers (e.g., Google [16], Microsoft [43]) are gradually deploying whitebox switches in their production networks. In contrast to the out-of-the-box devices from switch vendors (e.g., Cisco) where the switch firmware is entirely inaccessible for users, whitebox switches comprise disaggregated modules sourced from different vendors and composed together. This emerging architecture is gaining traction, because cloud providers can avoid vendor lock in, reduce costs, and introduce new features easily to their fleet. Specifically, as shown in Figure 1(a), the whitebox switches are mainly composed of three classes of modules: *(i)* a switching ASIC for packet forwarding in the data plane; *(ii)* network operating systems (NOS) as the control plane; and *(iii)* peripheral physical modules, such as electrically erasable programmable read-only memory (EEPROM), and sensors, to support normal operations of the switch. Custom switching ASICs and peripheral modules are purchased from different

|  | Control plane | Data plane | Peripheral modules |
|---|---|---|---|
| Whitebox switch | SONiC | ASIC | Yes |
| Off-the-shelf image | Private | Private | Unknown |
| SONiC-vs | SONiC | Kernel | No |
| SONiC-simu | SONiC | ASIC simulator | No |
| SONiC-bmv2 | SONiC | P4 | No |
| MirSwitch (Ours) | SONiC | VPP-extend | BSP |

**Table 1:** Existing software switches.

vendors, while the control plane is likewise customized in a scenario-specific manner (e.g., SONiC [44]). The underlying composition further varies across devices in the same provider, at high heterogeneity.

Figure 1(b) shows a typical workflow of a whitebox that is closely related to our work. Initially, the boot loader loads the open networking install environment (ONIE) package and boots it. The ONIE obtains the platform-related information (e.g., MAC address, vendor name, NOS version, port form) from the EEPROM and stores it in the disk, and then fetches the specific NOS image according to the obtained version and installs it as the control plane (e.g., SONiC). Then, the NOS will read the disk to get the related platform information and load the software driver in the Board Support Package (BSP) which is further used to boot peripheral modules. As for the switching ASIC, it is also controlled by the NOS through the switch abstraction interface (SAI [37]).

## 2.2 Motivation

Existing emulators [26, 31] directly use off-the-shelf switch vendor-provided images, which are one-size-fits-all blackbox binary files that cannot be customized. While these have been sufficient for previous-generation networks, they are a poor reflection of whitebox switches that emphasize composability. Meanwhile, there are also off-the-shelf software switches, such as SONiC-vs [20], SONiC-bmv2 [39]. We will provide the details of emulation needs that can not be addressed by off-the-shelf switches. These needs span three categories, for emulating *(i)* data plane features not only ASIC functionality but also sufficient levels of performance fidelity, *(ii)* control plane features specific to each NOS version, and *(iii)* peripheral modules to capture stealthy failures.

### 2.2.1 High-speed and faithful data plane

Firstly, we need a high-speed and faithful data plane to mimic the packet forwarding behavior of whitebox switch. Since SONiC is a leader in open source data center NOS deployments [23], we focused on the SONiC related data plane. Table 1 lists various data plane choices that can be used for emulation, including SONiC-vs [20], SONiC-bmv2 [39], and SONiC-simu that use the Linux kernel protocol stack, the BMv2 [1] software, and ASIC simulator from the switching chip vendors, respectively. We will provide the limitations of these approaches below in detail.

**Inconsistency.** SONiC-vs, relying on the Linux kernel protocol stack for packet forwarding, will cause the packet forwarding behavior inconsistent to the switching ASICs.

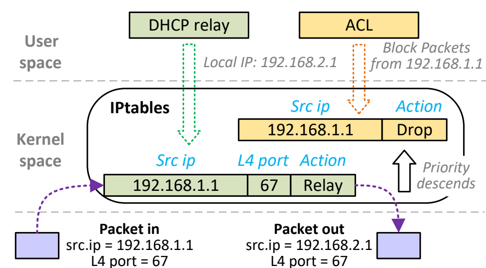



**Figure 2:** A rule conflict: DHCP-relay issues an IPtable entry with the same source IP address as an ACL entry but with a higher priority, resulting in an unexpected packet relay.

• *IPtables rule conflicts.* In the SONiC-vs software switch, some upper-layer functionales are offloaded onto the data plane through configuring the kernel IPtables. This approach can generate conflicts as the IPtable cannot guarantee priority. As shown in Figure 2, the *DHCP_relay* [7] agent relays the packet with source IP address `192.168.1.1` by adding an entry <`192.168.1.1` | *relay*> in IPtables. This relay operation will update the source IP address of the packet. Meanwhile, a higher priority ACL rule issued by users wants to drop the packet whose source IP address is `192.168.1.1` by adding an entry <`192.168.1.1` | *drop*> in IPtables. The two configurations conflict, and an unexpected DHCP packet *relay* occurs if the DHCP-relay entry is set with a higher priority.

• *ECMP hashing inconsistency.* Packets will be scheduled to different paths using the ECMP hashing algorithm. Although the traffic portions for different paths are comparable in the software and hardware switches, the exact result for a certain flow can differ. Specifically, for a certain flow, the hashing value calculated by the hardware switching ASIC can differ from that of the software switch, resulting in inconsistent forwarding paths of that flow. As a result, the network operators are not able to emulate the network in the same way as the production network. When a link along the real path fails, causing the loss of traffic, the same flow might not be lost in the emulation network.

**Low forwarding capability.** Traditionally, the forwarding performance of the data plane is regarded as out of scope—indeed, no emulation engine can faithfully capture Tbps-level throughput of switching ASICs. However, our experience shows that sufficient forwarding performance is key to emulating a large network topology. For example, the ASIC simulator is widely used to simulate the specific forwarding process of data packets inside the switching ASIC. However, we measure two types of SONiC-simu running ASIC simulator at the data plane from vendor1 and vendor2, respectively. We find that they only accommodate maximum packet forwarding rates of 320 packets per second (pps) and 480 pps for the entire switch, respectively. Given that a switch generally comprises 128 ports or more, the average packet forwarding rate per port is below 5 pps. Such limited forwarding performance hinders the emulation of a network topology that runs

various essential protocols (e.g., BFD [2], BGP [38]). For example, the BFD protocol send the packet every 0.1 seconds to ascertain if the link is down. The behaviors of BFD will not be activated unless the forwarding performance surpasses a rate of 10 pps. As for SONiC-bmv2, it provides high-fidelity emulation of the P4 pipeline, but suffers from low performance when handling large lookup tables [5]. Our design decision is to leverage advances from the past decade in network function virtualization (NFV): leading solutions such as VPP [11] use kernel-bypassing mechanisms and vectorized packet processing to achieve high performance.

#### 2.2.2 Bug-free customized control plane

The control plane is always customized by cloud providers to pace up to the fast-changing application demands. However, control plane bugs are often hidden within the software or arise from unreasonable protocol implementations in NOS. We have developed numerous features using the open source SONiC, with a new version released on average every month over the past two years. Our experiences show that when launching new features in the control plane, we always encounter several bugs. For example, we have updated the version of `teamd` function [7], which is a linux-based implementation of the LAG protocol. However, there is a bug leading to a memory leak in the new version. Once there are frequent link up and down in the physical switch, this bug will finally cause the system crash. Since the control plane of the off-the-shelf image used in CrystalNet [31] is private, it is hard to be directly used in to detect control plane bugs. To identify bugs instantly and accurately, we should use the same version of control plane in our proposed software switch as that in the physical whitebox switch deployed in production.

#### 2.2.3 Mock up peripheral modules.

Peripheral modules in the switch are likewise purchased from different vendors, exhibiting even higher heterogeneity. Existing software switches are homogeneous, leading to the emulation platform be unable to emulate the software bugs related to peripheral module events. For example, the received power of an optical transceiver is not stable. However, software bugs make it impossible to report the failure to NOS. In the homogeneous switch, network operators are not able to localize such failures that will happen in the whitebox switch. On the other hand, the homogeneous software switches cause inconsistent network characteristics compared to the physical one. We show three cases of the latter one as follows:

• *Topology inconsistency (Layer 1).* Port splitting technique [6] is widely supported in the switch to adapt to different port rates in the physical network. For example, assume that the port rates of leaf switch and ToR switches are 200Gbps and 100Gbps, respectively. The ports in the leaf switches should be split into two logical ports, each of which connects one port in the ToR switches. With the panel port information, e.g., the supported port splitting pattern, the operator configure the panel ports (e.g., number, rate, and splitting pattern) of the physical switch on demand. However, traditional software switches assume the fixed panel ports once built, and thus fail to support port splitting. Consequently, the inflexibility in the software switches makes it hard to keep consistent network topologies when the port splitting is used.

• *MAC address inconsistency (Layer 2).* The MAC addresses of the whitebox switch will also differ from the software switch though their hardware components are the same. For example, the MAC address of the existing software switch is randomly generated by SONiC at the booting step. This is different from the physical switch, which is specifically configured by the platform-related information. Such MAC address inconsistency makes network operators hard to trace L2 packets. Each time the network operators analyze layer 2 packets, they should look up the mapping table to correlate the traces between physical networks and emulation networks, which is complex and error-prone, especially when the emulated network is large-scale.

• *Forwarding path inconsistency (Layer 3).* To ensure a consistent packet forwarding path, the software switch must use the identical Hash configurations as the physical switches. The key is to obtain the Hash algorithm configured in the physical switch, which relies on the peripheral modules.

### 2.3 Challenges

To faithfully emulate the underlying architecture, we must transform bespoke emulation of each vendor's firmware into a novel, composable emulation framework that supports high customizability.A promising solution to minimize extra engineering effort to build a software switch is to use the NOS in the physical whitebox switch as the control plane, a high-performance NFV (e.g., VPP) as the data plane, and a new environment to mock up heterogeneous peripheral modules. However, given the diverse and disparate modules, stitching these modules together is non-trivial.

**Challenge 1: Existing packet processing solutions operate in a closed-loop manner.** High-performance packet processing solutions like VPP typically operate in a closed-loop manner. Specifically, VPP forwards packets according to the packet processing graph, where each node in the graph is running in user space. The data plane nodes are closely bundled with control plane nodes and do not provide interfaces for the production control plane, i.e., NOS. We should separate its data and control planes, exposing interface contracts for the high-speed software processor to integrate with realistic NOS control planes. Furthermore, these frameworks ignore some important data plane functionalities in the switching ASIC, such as L3 VLAN interface and ECMP.

**Challenge 2: There are semantic gaps between SAI and software data plane interfaces.** As switching ASIC pursues low resource and energy consumption, the size of Ternary Content Addressable Memory (TCAM) is strictly limited, whereas the software data plane is free to organize large-

capacity lookup tables. Since SAI is traditionally designed to provide interfaces for physical switching ASICs, there will be semantic gaps between SAI and the software data plane. For example, there are a series of hardware tables realizing the Forwarding Information Base (FIB) to reduce the hardware resource consumption [41]. However, in the software data plane (e.g., VPP), there is only one routing table with comprehensive routing information. Hence, the SAI APIs for configuring the hardware FIB must be different from those in the VPP. The semantic gaps between SAI and software data plane interfaces motivate us to design a new adaptor to establish a mapping between them.

**Challenge 3: Current BSP only supports being loaded in a physical environment rather than in the software switch.** The BSP is significant for adapting highly homogeneous software switches to heterogeneous whitebox switches. When installing NOS on a whitebox switch, the NOS will load the BSP only when the platform information provided by ONIE is related to hardware. However, the software switch lacks physical characteristics, and ONIE only provides software information (i.e., VM or container), resulting in the failure to load BSP. Hence, it is challenging to transparently load the BSP, regardless of the switch being software or physical.

# 3 System Design and Implementation

## 3.1 High-level Design

Our goal is to design a novel composable emulation framework called MirSwitch that supports high customizability, so that we can customize the software switch, adapting to whitebox switch and mimic its behavior in a high-fidelity way. MirSwitch inherits the basic paradigm of whitebox switches that utilizes the same production NOS control plane (i.e., SONiC) and leverages a high-performance VPP framework as the software data plane, and a new environment to mockup peripheral modules. MirSwitch addresses key challenges in reconciling interface contract differences across composed modules using an adaptor-centric approach. More concretely, MirSwitch firstly introduces an extended VPP (eVPP) to faithfully synthesize data plane functionalities like switching ASICs (§ 3.2.2), and designs a VPP adapter to expose interface contracts for the high-speed software processor to integrate with control plane (§ 3.2.1). As there are semantic gaps between SAI and the software data plane. MirSwitch design a semantic adapter that translates requests from SAI to the software data plane (§ 3.3). Meanwhile, to mock up these peripheral modules, MirSwitch introduces an environment adaptor that creates a new mockup environment (§ 3.4). Figure 3 presents MirSwitch's high-level design.

## 3.2 Software Data Plane

We use the high-performance VPP as the software data plane, instead of traditional alternatives [1, 36]. However, the original VPP framework cannot be directly used as the software

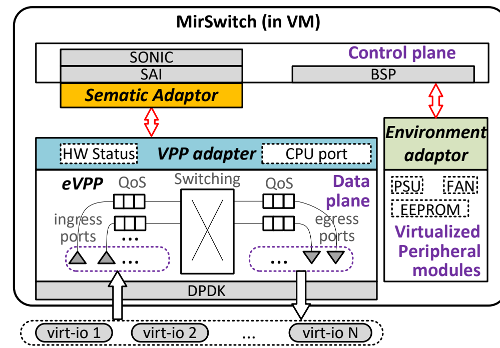



**Figure 3:** The high-level design of MirSwitch.

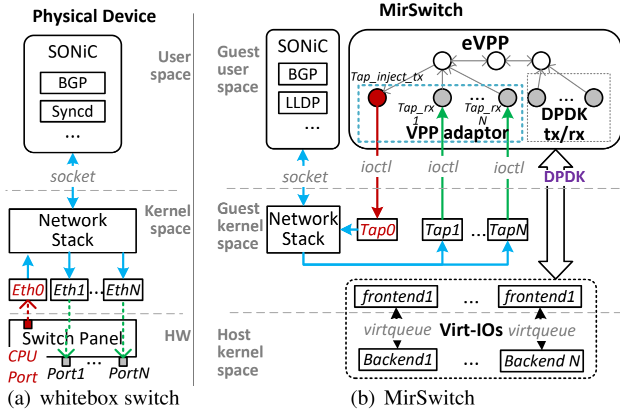
data plane because of several issues, such as closely bundled control and data plane nodes without interfaces for NOS and a lack of important functionalities. Therefore, we re-architect VPP designs to address these issues.

### 3.2.1 VPP adaptor

The original VPP forwards packets in a closed loop in user space, where the control and data plane nodes are closely bundled. This would make it challenging to interact with the NOS control plane in the same way as the whitebox switch, such as sending and receiving packets to and from the SONiC, monitoring device data, etc. To address this, we design a VPP adaptor to abstract the software data plane and provide the interfaces for SONiC.

In order to achieve the way in which the software data plane behaves like the physical data plane towards the control plane, we first study the packet forwarding workflows within a typical whitebox switch, as shown in Figure 4(a). There are three main packet forwarding workflow in the physical data plane: (1) *data plane to data plane.* The packet is received at a switch panel port and will be directly forwarded to the other panel port if the next hop of the packet in the matching table is another switch. (2) *data plane to control plane.* The packet is received at a switch panel port and forwarded to the CPU port if the destination of the packet is the switch itself. Then the packet is forwarded to the network stack through Ethernet port (*Eth0, Eth1, ..., EthN*) and finally reaches SONiC for further processing. (3) *control plane to data plane.* The SONiC generates a packet that will be forwarded to another device in the network through the switch panel port. Therefore, to faithfully mock up the data plane behavior in whitebox, the software data plane should be designed to follow the same packet forwarding workflow. Figure 4(b) depicts the workflow of MirSwitch, and we will describe it below in detail.

**Data plane to control plane.** The packets destined for the control plane will be ended in the nodes that perform control plane functionalities in the original VPP. Although many inter-process communication methods can be used in VPP adaptor, we continue to use the kernel network stack like the whitebox switch to keep transparent for SONiC, though it needs switches between user and kernel mode. As shown in Figure 4(b), the VPP node *tap_tx* and the kernel virtual L2



(a) whitebox switch (b) MirSwitch

**Figure 4:** The packet forwarding workflows in (a) a typical whitebox switch and (b) MirSwitch.

interface *Tap0* in MirSwitch, act as the CPU port and the Ethernet interface *Eth0* in the whitebox switch, respectively. Specifically, the packets destined for the control plane are directed to the *tap_tx* node in the VPP adaptor, and then forwarded to the *Tap0* by calling the `ioctl` function. The kernel network stack finally sends these packets to the SONiC. In this way, we faithfully mimic the packet forwarding process from the data plane to control plane in the whitebox switch.

**Control plane to data plane.** The packet flow from the control plane to the data plane is similar. When initiating a protocol packet, the SONiC will initiate a `socket` and then the kernel network stack routes the packet to the appropriate TAP, e.g., *Tap1*. The *tap_rx1* node in the VPP adaptor, which listens on *Tap1*, will capture the packet and start the data plane process. Finally, the packet reaches the *dpdk_tx* node in eVPP and is forwarded to the corresponding Virt-IO front-end via DPDK. VPP adaptor enables MirSwitch to behave faithfully like a whitebox switch: the CPU port is mapped to the *tap_tx* node while Ethernet interfaces (Eth) are mapped to Taps. In this way, for each interface entity in the whitebox switch, there is a unique mapping in the MirSwitch. Note, packets forwarded between the user space and kernel space introduce a little overhead, which does not influence the forwarding performance since the majority of packets are forwarded within the data plane.

**Data plane to data plane.** The VPP natively supports the packet forwarding from the data plane to another data plane according to the packet processing graph. Specifically, the VPP pulls and pushes packets from and to Virt-IO front-ends through DPDK. Here, the Virt-IO in the whitebox switch is mapped to a panel port in the whitebox switch. Although part of data plane functionalities has been supported by the VPP, some important functionalities such as VLAN and ECMP have not been faithfully supported which requires for further development (see § 3.2.2).

**Hardware status report.** In a whitebox switch, the hardware resource management process (i.e., Syncd [10]) in SONiC is responsible for monitoring the hardware status, such as port status up/down. However, there is no hardware to be monitored in MirSwitch. To support this management functionality, we enable the VPP adaptor to act as a hardware event trigger. Take the port up/down status as an example. After a new MirSwitch is launched, the VPP adaptor creates a thread to scan all the MirSwitch's panel ports (i.e., the Virt-IO front-ends) periodically. When the port status changes, such as from up to down or down to up, the thread will report this event to the Syncd module in the SONiC and then the Syncd module will notify other applications for further processing.

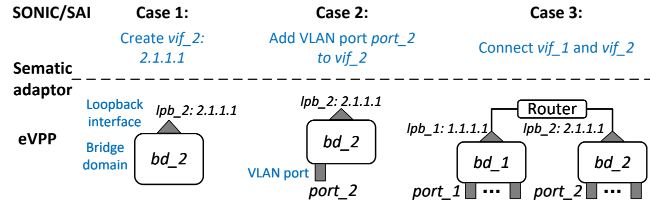


**Figure 5:** Three typical cases of configuring VLANIF.

### 3.2.2 Extended VPP

Since many functionalities are not supported in the original VPP, we introduce eVPP that extends the original VPP to extend software data plane functionalities to faithfully mock up a standard whitebox switch. Specifically, we modify dozens of data plane functionalities. Here, we present two representative functionalities in the data plane: ECMP and L3 VLAN interface (VLANIF), and their workflows can be extended into the development of other functionalities.

**Functionality1—ECMP.** Although the original VPP provides the ECMP functionality, its ECMP hashing function is private and it does not provide APIs for configuring the hashing function flexibly. In the whitebox switch, the ASIC usually provides a hashing function set to the control plane for selection. The hashing function in the VPP and ASIC must be different to a large extent, resulting in different hashing values. Therefore, the forwarding port number at a whitebox switch will be different from the Virt-IO front-end number in MirSwitch. To solve this problem, we introduce a package in the eVPP that contains all hashing function sets used in our forwarding ASICs, which are obtained from our device vendor partners. Once we obtain the object hardware platform information through vBSP (see § 3.4), eVPP can load the corresponding hashing function set. In addition, the semantic adaptor supports SAI request of configuring the hashing function set, leading to MirSwitch performing the same ECMP functionality as the switching ASIC.

**Functionality2—VLAN interface.** The original VPP is an L3 router without any L2 functionalities, including VLAN. Here, we use the most complicated L2 related functionality VLANIF to show our extension to VPP. Strictly speaking, VLANIF is a virtual L3 interface with an IP address, responsible for forwarding L3 packets to the next VLANIF, but it also supports forwarding or broadcasting L2 packets in the VLAN domain, so it has two roles.

To achieve the aforementioned functions of VLANIF, we let each VLANIF consist an L2 bridge domain and an L3 loopback interface in eVPP. Specifically, as shown in Figure 5, a VLANIF is configured in three cases: (1) When the upper layer creates a VLANIF *vif_2*, the corresponding loopback interface *lpb_2* and the bridge domain *bd_2* will be created in eVPP instantly, and then *lpb_2* will be added as a VLAN member in *bd_2*. (2) When the upper layer adds a VLAN port into *vif_2*, the port will be mounted on *bd_2* as a VLAN member. (3) When the upper layer distributes a route entry with the next hop of a VLANIF to connect two VLANIF networks, e.g., *vif_1* and *vif_2*, the next hop will be set with the IP address of *lpb_2*. Note that the above operations require the semantic adaptor to translate semantics between SONiC and the software data plane (see § 3.3). Therefore, the intra-VLAN flows (e.g., L2 packets in *bd_2*) will be forwarded to the destination port or broadcast within the bridge domain, and inter-VLAN flows (e.g., L3 packets in *bd_1* but destined for the IP address of *vif_2*) will be forwarded to the next VLANIF (e.g., *lpb_2*).

### 3.3 Semantic Adaptor

Since switch abstraction interface (SAI [37]) is originally designed to provide interfaces for physical chips, there will be semantic gaps between SAI and software data plane interfaces. To address this, we propose a semantic adaptor to bridge their semantic gaps. In summary, there are two basic types to translate requests between SAI and eVPP: (1) one-to-many mapping, denoting translating a single SAI request into multiple eVPP requests, and (2) many-to-one mapping, denoting several SAI requests determining a single eVPP request. Other types can be seen as extensions of these two types. For example, many-to-many mapping consists of multiple many-to-one mappings.

**One-to-many mapping.** An SAI request may contain various comprehensive semantics, failing to be directly translated to an eVPP request, and thus the one-to-many mapping is needed. Specially, the one-to-one mapping is a simplified situation of one-to-many mapping. Figure 5 depicts a typical example that an SAI request of configuring a VLANIF will be mapped into several eVPP requests by the semantic adaptor. Specifically, upon receiving the SAI request of creating a VLANIF (case 1 in Figure 5), the semantic translator maps it into several eVPP requests of creating the lookback interface and bridge domain, and then binding them. However, as one SAI request has been mapped into several eVPP requests, there are strict dependencies between the eVPP requests. To address this, the semantic adaptor maintains a *dependency tree* to guarantee the sequential execution of these eVPP requests. Specifically, the dependency tree confirms that the binding of the lookback interface and bridge domain is carried out after they are created.

Furthermore, the mapping may be indirect and the semantic adaptor needs to go through a series of intermediate stages. As the case2 shown in Figure 5, upon receiving a SAI request of adding a VLAN port to the VLANIF, the semantic adaptor should first extract the corresponding bridge domain by looking up the VLANIF—bridge-domain mapping table, and then composes the eVPP requests. Note that if current states do not allow initiating an eVPP request, e.g., a temporary lookup miss in the VLANIF—bridge-domain table, the adaptor will record the SAI request for future to finish it, but not blocking the subsequent independent SAI requests.

**Many-to-one mapping.** On the other hand, the eVPP may have comprehensive semantics, and several SAI requests should be aggregated into an eVPP request. However, these SAI requests always have a dependency, which makes the translation more complicated. To address this, we will take the configuration of the IP route table to illustrate how the adaptor deals with such a dependency.

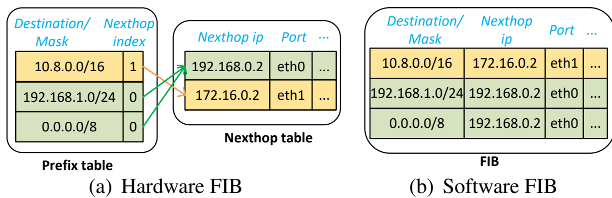
The FIB stores the route information and applies the classic Longest Prefix Match (LPM) for IP lookup. Figure 6 shows the different implementations of the hardware FIB in the switching ASIC and the software FIB in eVPP. To save the hardware resource, the hardware FIB is based on the *prefix* table and *nexthop* table [21,30,32]. The prefix table consisting of the prefixes and their next hop indexes, is stored in the TCAM and the length of table entry is limited (e.g., less than 134 bits TCAM line) to reduce hardware cost and energy consumption. The index is used to access the nexthop table stored in on-chip SRAM, which stores the rest of the route entry information, such as the next hop address and interface. Note that the nexthop entries are usually in hundreds, while the size of the prefix table can be at 10K or even 100K level, which means the hardware FIB structure reduces the total memory cost. In contrast, due to abundant memory in the software, the software FIB only needs one table to contain all route information instead of establishing multiple cascaded tables.

Our proposed semantic adaptor bridges the gaps by maintaining the three tables in Figure 6. Specifically, the configuration of the IP route table can be mainly summarized into the following four categories, and we then describe how the semantic adaptor works in detail:

(1) *Inserting a new next hop.* Upon receiving an SAI request to insert a new next hop, the semantic adaptor will directly insert an entry in the nexthop table.

(2) *Inserting a new prefix.* Upon receiving an SAI request to insert a new prefix, the semantic adaptor will insert an entry in the prefix table and look up its next hop information in the nexthop table, and then integrate them to form a route insertion request towards eVPP.

(3) *Modifying/deleting a next hop.* Upon receiving an SAI request to modify or delete a next hop, the semantic adaptor will firstly modify or delete an entry in the nexthop table, then figure out all the influenced route entries in the software FIB, and finally distribute a series of route modification/deletion requests towards eVPP.




(a) Hardware FIB (b) Software FIB

**Figure 6:** The different structure of FIB in Switching ASIC and eVPP: (a) hardware FIB uses a two-level table on TCAM and SRAM; and (b) software FIB uses a one-level software table.

(4) *Modifying/deleting a prefix.* Upon receiving an SAI request to modify or delete the prefix, the semantic adaptor will firstly modify or delete the prefix table and then distribute a route modification/deletion request towards eVPP.

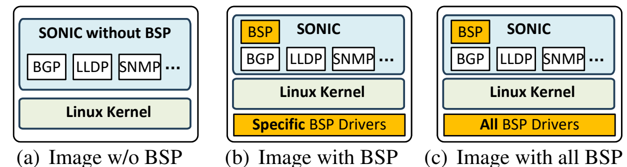
Note that the software FIB is updated synchronously in eVPP among the above IP route table configurations, to keep consistency. Besides, the semantic adaptor also limits the number of software FIB entries not exceeding that of the TCAM entries, maintaining a faithful data plane. The ECMP is a more complicated many-to-one example and the reader can refer to Appendix A for more details.

## 3.4 Environment Adaptor

The BSP is used to manage peripheral modules in the whitebox switch (§ 2.1) so that SONiC can interact with these heterogeneous peripheral modules in a vendor-agnostic way. However, existing software switch [20, 29, 39] does not support the BSP, making it highly homogeneous (see Figure 7(a)). The lack of BSP in the software switch would bring about a discrepancy between the software and physical switch, which hinders achieving high fidelity. However, our interaction with network engineers indicates that loading the BSP in the software switch is quite challenging since the BSP is loaded only when the object platform is identified as a physical switch. Meanwhile, the types of BSP diversity due to various vendors and different versions, etc. further bring new issues to achieve high fidelity. To address this, we design an environment adaptor to shield the software data plane and enable loading the same BSP as the physical switch.

**Shielding software data plane.** Recalling the workflow of booting up a whitebox switch in Figure 1(b), the ONIE obtains the platform information from the EEPROM and stores it in the disk. Then, the NOS will read the disk to get the platform information and load the corresponding BSP drivers. However, as there is no EEPROM in the software switch, the ONIE will save the VM platform information in the disk. Our proposed enviroment adaptor is to develop a new ONIE that uses a fake but properly formatted EEPROM file with the platform information. Then the new ONIE will read the platform information from the fake EEPROM file and store it in the disk for NOS to load the corresponding BSP drivers. Note that it does not introduce much overhead in this way. Therefore, the NOS can load the BSP transparently regardless of the software and hardware data plane.

**Adapting to heterogeneous hardware platforms.** Despite ONIE being able to read the switch platform information, the BSP type varies among vendors and switch versions. A naive approach is to package a specific BSP in the image for each type of whitebox switch, as shown in Figure 7(b). The number of the image is linear to the number of the BSP type. As there are tens to hundreds of whitebox switch types in production, the number of images is quite large, which requires a lot of labor on generating all types of images. To address this, the environment adaptor will package all the BSP types into one VM image, as shown in Figure 7(c). Once the platform information is read from the disk containing the specific BSP information, the NOS will load the corresponding BSP. Therefore, the BSP of the software switch is the same as the physical switch. This approach enables network operators to maintain one software switch image for all types of hardware switches in production, and the image can be transformed into different specific images by loading the specific BSP.




(a) Image w/o BSP (b) Image with BSP (c) Image with all BSP

**Figure 7:** Different images for BSP: (a) Traditional VM image does not contains any BSP; (b) Pack the specific BSP and its drivers for the object hardware platform; and (c) Pack all BSP and BSP drivers for different hardware platforms.

## 4 Evaluation

In this section, we first conducted experiments on a testbed to evaluate the performance of MirSwitch, as compared to existing software switches (§ 4.1). We deploy MirSwitch to emulate the production data center network and support our daily use in troubleshooting network issues (§ 4.2).

## 4.1 Testbed evaluation

**Experiment Setup:** Our testbed consists of 16 server nodes with AMD EPYC 7K62 CPU (48 cores, 2.6/3.3GHz) and 512GB DDR4 memory. Our MirSwitch is deployed in a virtual machine to be compatible with the container-based SONiC components. We use GNS3 [3] as the orchestrator to connect the Virt-IO back-ends to construct network topology. Linux bridge and VXLAN are used to connect our proposed software switches inside the server and between servers.

### 4.1.1 Switch-level fidelity

Table 2 presents some critical functionalities of a standard whitebox switch [8, 9] and the comparison between MirSwitch, SONiC-vs, and SONiC-bmv2. The reader can refer to the complete functionalities in Appendix B. We observe that the functionalities supported by MirSwitch are close to the physical whitebox switch and MirSwitch outperforms others significantly. In general, for a total of 57 functionalities, MirSwitch achieves the highest support ratio of 89.5%. Next, we

| | Functionality | MirSwitch | SONiC -vs | SONiC -bmv2 |
|---|---|---|---|---|
| **Ctrl plane** | BGP | ✓ | ✓ | ✓ |
| | Routing stack | ✓ | ✓ | ✓ |
| | Warm reboot | ✓ | ✗ | ✗ |
| | IPv6 linklocal | ✓ | ✓ | ✗ |
| **Mgmt plane** | SNMP | ✓ | ✓ | ✓ |
| | Platform-specific commands | ✓ | ✗ | ✗ |
| | CMIS | ✓ | ✗ | ✗ |
| | SFP utilities | ✓ | ✗ | ✗ |
| **Data plane** | Drop counter | ✓ | ✗ | ✗ |
| | Faithful ECMP | ✓ | ✗ | ✗ |
| | IPv4/IPv6 | ✓ | ✓ | ✓ |

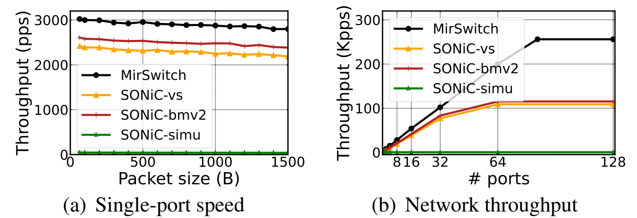
**Table 2:** Comparison between traditional solutions and MirSwitch on critical functionalities in the SONiC roadmap planning [8, 9].

categorize these functionalities into three classes: control (i.e., ctrl) plane, management (i.e., mgmt) plane, and data plane, to compare MirSwitch with other software switches in detail.

First, at the control plane, all software switches achieve high fidelity by supporting almost all of the functionalities. Since these software switches are based on the SONiC, they are natively compatible with the SONiC-based whitebox switch. MirSwitch achieves the highest degree of fidelity by supporting all 26 functionalities in the standard whitebox switch. However, the number of functionalities supported by the SONiC-vs and SONiC-bmv2 is lower since part of functionalities can not able to supported. For example, an important functionality called *warm reboot*, which is used for restarting and upgrading SONiC software without impacting the data plane, is not supported by traditional approaches. However, MirSwitch uses the semantic adaptor to store pre-reboot data plane states on the disk upon receiving the restart requests to prepare for the warm restart.

Second, MirSwitch fully supports all the 15 management plane functionalities, allowing network operators to manage MirSwitch like a physical switch. Both SONiC-vs and SONiC-bmv2 support parts of functionalities with a ratio of 73.3% (11/15 ) because they only support common functionalities, such as SNMP, and fail to support vendor-specific functionalities, such as platform-specific commands functionality. In contrast, MirSwitch introduces an environment adaptor to load BSP in the software switch to flexibly adapt to vendor-specific details. For example, the commands for getting firmware information by common management interface specification (CMIS) and configuring small form pluggable (SFP) both require the support of BSP.

Finally, for the complex data plane, none of the software switches can support all the functionalities as compared to physical switches. The main reason is that part of the data plane functionalities strongly rely on hardware characteristics. For example, the PFC function is determined by buffer management, while the QoS relies on the scheduling algorithm, which are both implemented inside the switching ASIC and not accessible. SONiC-vs exhibits the poorest performance with a ratio of 37.5% (6/16 ). This outcome can be attributed to the limited and stubborn data plane capabilities inherent



(a) Single-port speed (b) Network throughput

**Figure 8:** Packet forwarding performance of MirSwitch, SONiC-vs, SONiC-bmv2 and SONiC-simu.

in the Linux kernel. SONiC-bmv2 possesses the potential to accommodate a broader range of data plane functionalities via the P4 programming language. However, implementing these functionalities demands considerable effort, particularly due to the limited hardware pipeline stages and resources available. MirSwitch proposes an extended VPP (eVPP) by determining the detailed workflow to synthesize functionalities with a lightweight development overhead. Meanwhile, by coordinating BSP and eVPP, MirSwitch can also support some critical functionalities such as ECMP, which can not be supported by existing solutions. Therefore, MirSwitch achieves higher fidelity by supporting up to 75% (12/16) functionalities.

### 4.1.2 Network Throughput

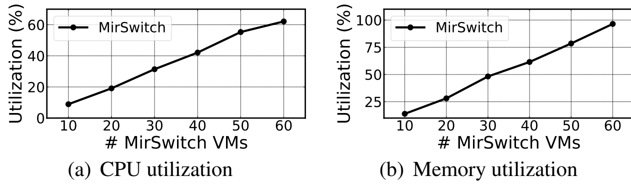
We evaluate the network throughput at the port level and switch level. Higher network throughput indicates higher forwarding performance of the software switch, enabling a more flexible usage of switch to emulate more complex networks.

**Single-port throughput.** Figure 8(a) shows the single-port network throughput of MirSwitch when the packet size increase, as compared to SONiC-vs, SONiC-bmv2, and SONiC-simu. We observe that SONiC-simu performs the worst, supporting only 3 packets per second (pps). This is attributable to its comprehensive implementation of the data plane to fully mimic the behavior of the switching ASICs. Both SONiC-bmv2 and SONiC-vs attain good throughput, but their data planes are not faithful. MirSwitch performs the best, reaching up to 3000pps while maintaining high fidelity data plane.

**Switch-level throughput.** Figure 8(b) shows the switch-level throughput of MirSwitch, SONiC-vs, SONiC-bmv2, and SONiC-simu across varying numbers of panel ports, under 128-byte packets. When the number of ports in the switch is small, an increase in the number of ports promotes a higher network throughput. However, as the number of ports becomes larger, the network throughput does not increase due to the software bottleneck inside the switch. We observe that MirSwitch supports a network throughput of 256Kpps with 128 ports, which outperforms other solutions (i.e., SONiC-bmv2 and SONiC-vs) by at least 2.2×.

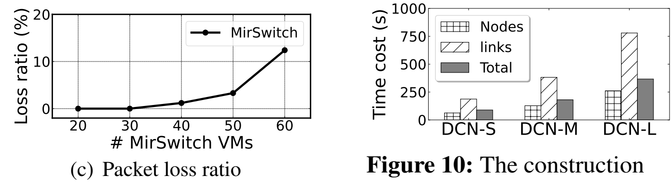
### 4.1.3 Scalability

We then study the scalability of MirSwitch to construct different scales of network topology. We use three typical network scenarios with different scales from our operational data center networks, denoted as DCN-S, DCN-M, and DCN-




(a) CPU utilization

(b) Memory utilization




(c) Packet loss ratio

**Figure 9:** CPU utilization, memory footprint and packet loss ratio as MirSwitch VMs increases

**Figure 10:** The construction time in for different networks.

L, respectively. DCN-S is a small network pod with 8 leaf switches, 96 TOR switches, and total 96*24 servers; DCN-M comprises two pods by 8 spine switches; and DCN-L, four pods, 16 spine switches.

**Resource consumption.** We conducted a pressure test on CPU and memory utilization on a server with different numbers of VM-based MirSwitch. As shown in Figure 9(a) and Figure 9(b), we observe a linear relationship as the number of MirSwitch increases, the CPU and memory utilization increases. We then use iPerf [4] to inject network traffic into MirSwitch and measure the packet loss ratio. As shown in Figure 9(c), we observe that when the number of MirSwitch is more than 30, there is part of packet loss, indicating that the server is too busy to process these packets. By taking CPU and memory utilization and packet loss ratio into consideration, we set the maximum number of MirSwitch in a server to 30. Therefore, we only use 4, 8, and 15 physical servers to construct DCN-S, DCN-M, and DCN-L, respectively.

**Construction time.** We then study the network construction as the network size increases. The total construction time is the sum time of constructing links and nodes. Figures 10 shows the time to construct links and nodes among different network sizes. We observe that the total time to construct DCN-L, DCN-M and DCN-L, is 1.5, 3.0, and 6.1 minutes, respectively. This is acceptable for emulating a production network. Note, the node and link can be further parallelly constructed among servers to reduce the total time.

## 4.2 Production Results

We use MirSwitch to study the faithful emulation of the production data center network (§ 4.2.1), and then use it to support our daily use in troubleshooting network issues (§ 4.2.2).

### 4.2.1 Emulating Production Data Center Network

To evaluate the network-level fidelity of MirSwitch for production data center networks, we test the accuracy of network configuration, network topology, packet forwarding paths, and network performance metrics on three network topologies with DCN-S, DCN-M, and DCN-L, respectively.

**Network configuration.** Figure 11 shows the configuration difference of emulating a production network with the traditional software switch (e.g., SONiC-vs, SONiC-bmv2) and MirSwitch. In the production network, two spine switches `SW0` and `SW1` split its 400Gbps ports into two 200Gbps logical ports, with each connecting to a physical port in the leaf switch. The traditional emulation network which relies on

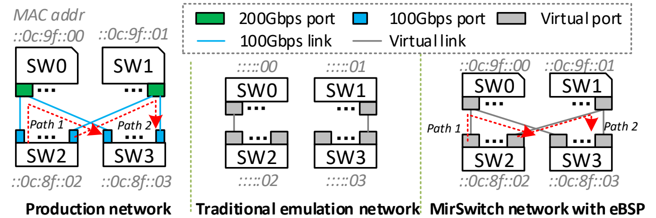



**Figure 11:** Network configuration difference: Traditional emulation network relying on off-the-shelf software switch uses random MAC addresses and does not support port splitting; With BSP obtained by the environment adaptor, MirSwitch supports the same network topology and MAC address as the production network.

the existing software switch does not support port splitting and can only establish one link for a single port, resulting in inconsistent emulation network topology as compared to the production network. The packets traversing `SW2`, `SW0`, and `SW3` cannot be replayed in the traditional emulation network, since there is no link between `SW0` and `SW3`. Meanwhile, the MAC addresses of software switches in the traditional emulation network are not the same as those in the production network. Each time the network operators analyze layer 2 packets, they should look up the mapping table to correlate the traces between physical networks and emulation networks, which is complex and error-prone, In contrast, the emulation network relying on our proposed MirSwitch accurately reproduces the production network. This shows the advantage of introducing BSP in the MirSwitch.

**Network topology similarity.** We then quantify the similarity of the emulation network with the production network. We introduce the graph edit distance (GED [25]) which is defined as the minimal number of operations (e.g., insertion/ deletions/modification for edge/vertexes) required to realign two topologies to be the same. For ease of comparison, we define the GED-based topology similarity as $1-\frac{\text{GED}}{\max(|V_p|+|E_p|,|V_e|+|E_e|)}$, where $V_p$, $E_p$, $V_e$ and $E_e$ denotes the vertexes and edges in the topologies of production and emulation networks, respectively. Figure 12 shows the average topology similarity of MirSwitch, SONiC-vs and SONiC-bmv2 networks against the real DCN_S DCN_M and DCN_L. MirSwitch achieves a similarity value of 100%, indicating that it can accurately reproduce the all production network. However, SONiC-vs and SONiC-bmv2 show a slightly lower similarity value in all three network topologies. The main reason is that the port splitting phenomenon, which may cause inconsistencies in our production network, is less than 5%.

**Packet forwarding path.** The packet forwarding path is de-

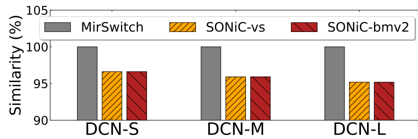


**Figure 12:** GED-based topology similarity of the emulation networks against the corresponding production networks

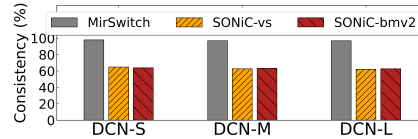


**Figure 13:** The consistency of packet forwarding paths in emulation and production networks.

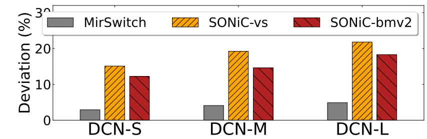


**Figure 14:** Standard deviations of the observed link's utilization in emulation and production networks.

termined by ECMP hash function. As shown in Figure 11, the packet in the production network originated from `SW2` to `SW3` has two paths by traversing `SW0` or `SW1`. For the traditional emulation network, even its network topology is the same as the production network, the path of the packet may not be the same as that in the production network due to its different ECMP hash functions. To evaluate the fidelity of packet forwarding paths, we collect one-hour production network traffic and inject it into the emulation network to monitor the consistency of packet forwarding paths. Figure 13 shows MirSwitch presents a consistency rate of 100% regarding the network topology and flow path between the production network and the emulation network, attributed to its consistent port form and ECMP hash function. However, the path consistency rates of SONiC-vs and SONiC-bmv2 are less than 65% and decrease slightly as the network scale increases.

**Network performance.** Despite we can not mock up the absolute values of network metrics such as network throughput, we can scale down the value in the emulation network. Here, we take an important metric *bandwidth utilization* of the link to study the fidelity of our MirSwitch. Figure 14 presents the standard deviation on three network topologys. We observe that MirSwitch presents the smallest standard deviation within 5%, while the standard deviations of SONiC-vs and SONiC-bmv2 reach 20% and 15% respectively in DCN-M. This phenomenon indicates our approach can reproduce these performance metrics with high fidelity.

### 4.2.2 Detect Network Failures

MirSwitch has been used in our production network on a daily basis for years. It has successfully detected many network issues summarised in Table 3, where the traditional approach is using software switches provided by our vendors. In last year, our production network suffers 203 related network issues, and MirSwitch can troubleshoot 198 (97.5%) of them.

**Control plane.** Specifically, MirSwitch helps detect nearly all SONiC bugs due to its high fidelity at the control plane [1], whereas the traditional software switch is not applicable. For instance, in the whitebox switch, when the number of neighbors exceeds the maximum size of the next hop table, SONiC will delete all neighbors and relearn them. This behavior can be accurately emulated and troubleshooted in MirSwitch. Using the same Linux kernel and drivers as the production whitebox switch, MirSwitch is able to detect many kernel and driver bugs, such as incomplete support for IPv6 protocol stack and core dump when creating an aggregation port. More importantly, by introducing BSP in the software switch, MirSwitch helps detect more stealthy bugs. For example, the received power of an optical transceiver is not stable. However, software bugs make it impossible to report to NOS. By leveraging MirSwitch, network operators are not able to localize such failures in the software that will happen in the whitebox switch.

| Field | Issues | Examples | MirSwitch | Trad | Total |
|---|---|---|---|---|---|
| Control plane | SONiC bug | bugs in SONiC components | 56 | NA | 57 |
| | Network Config bug | wrong configs for BGP and ACL | 74 | 62 | 74 |
| | BSP driver bug | bugs when light power fluctuates | 2 | NA | 2 |
| Data plane | Network optimize | load imbalance, and hash collision | 23 | 10 | 27 |
| Peripheral module | Stand-alone Config bug | wrong host name and port down | 43 | 36 | 43 |

**Table 3:** Comparison of the traditional software switch and MirSwitch to detect network issues in the last year. NA means Not Applicable, and Trad means the traditional approach.

**Data plane.** MirSwitch helps optimize the production network by analyzing the link load and packet forwarding path. For example, hash collision [15, 33] always happens within the network, resulting in some links being overloaded. By emulating the production network with high fidelity, we can precisely adjust configuration parameters to avoid congestion. Taking the hash collision in the LLM training cluster as a case in point. When network congestion occurs in the production network, we collect network topology data as well as traffic data. We then construct an emulation network and inject this data to replay network congestion. By gradually adjusting the configuration parameters, the network congestion can be resolved. Finally, the new configurations are applied to the production network. This approach greatly improves the network performance while requiring minimal intervention in the production network. MirSwitch has resolved several hash collision issues in LLM training clusters.

**Peripheral module.** For the stand-alone configuration bugs on peripheral modules, MirSwitch detects 100% of them as it has a high-fidelity data plane and supports many BSP-related configurations. For example, when an operator is trying to split a 100Gbps port into two logical 50Gbps ports, but the hardware port does not actually support port splitting, MirSwitch can detect this error with the platform information from BSP. In contrast, our traditional software switch can only detect some bugs related to the specific device.

[1] Here, the one bug that has not been detected by whitebox is related to the dhcp-relay functionalities.

## 5 Lessons learned

In this section, we present operational lessons throughout years of developing and running MirSwitch in production.

**Tradeoff between fidelity and performance.** Our MirSwitch claims high fidelity rather than full fidelity. Initially, we have tried to emphasize more on the full-fidelity. We have developed SONiC-simu software switches with the aid of ASIC simulators from switching ASIC vendors. Although the SONiC-simu vividly simulates the packet forwarding details in the switching ASIC, our measurement results show that it encounters extremely low packet forwarding rate, making it not practical to mock up physical networks. The main reason is that SONiC-simu builds complex logics in the software to achieve full fidelity. MirSwitch sacrifices part of fidelity, but can achieve high performance to mock up physical networks.

**Leveraging open-sourced frameworks with abundant functionalities.** We have learned that utilizing the framework with abundant functionalities can greatly reduce the development cost. We firstly relied on SONiC-bmv2 [39] to implement data plane functionalities. Given that the capabilities of bmv2 are rather primitive, experienced network engineers generally need around 15 man-days to develop the ECMP functionality. In contrast, we accomplished the development of this functionality in only 5 man-days. Table 4 presents the labor costs with the experienced engineers in our company for implementing some typical functionalities in MirSwitch, SONiC-vs, and SONiC-bmv2. Note that we only need a small labor cost to incorporate new components from other vendors because we have developed the environment adaptor to accommodate to heterogeneous hardware platforms (§ 3.4)

**Functionalities vs usability.** Our production experiences show that we should consider the priority during the development of different functionalities. The basic functionalities should be developed first. For example, zero touch provisioning (ZTP) [22] functionality should be prioritized to automate the deployment and configuration of network devices without manual effort. This approach significantly mitigates manual configuration errors and reduces deployment time overhead. Secondly, we develop the functionalities as needed to keep pace with the emulation requirements. Finally, to address performance bottlenecks introduced by resource-intensive protocols, e.g., BFD configured with the packet sending interval at 0.1 seconds, we can modify the configuration settings to lessen resource usage, such as 1 second. This mechanism can minimize resource consumption while maintaining the fidelity of the network emulation.

## 6 Related Work

**Network validation.** A series of works focused on network validation. Network verification systems [18, 19, 28, 42] relied on mathematical methods to answer reachability questions, but could not verify the impact of software bugs, protocol implementation differences among vendors. Discrete event simulators [13, 14, 24] simulated the packet forwarding process in discrete models, failing to detect real software bugs. In general, previous approaches presented a very different workflow to operations of production network. Network emulators such as CrystalNet [31] and Crescent [26] were closer to our work. These emulators focused on how to construct the emulation network and directly run switch vendor images inside containers, leading to a diminished fidelity in validating the production network consisting of white-box switches. For example, they are not able to emulate the bugs inside white-box switch. In contrast, MirSwitch is a composable emulation framework designed to tailor the software switch to fit the whitebox switch, thereby facilitating a high-fidelity emulation of the production network consisting of the whitebox switch.

| Functionality | MirSwitch | SONiC-vs | SONiC-bmv2 |
|---|---|---|---|
| BGP | 1 | 2 | 2 |
| TEAMD | 2 | 5 | 5 |
| VLANIF | 6 | 4 | 6 |
| ECMP | 5 | 20 | 15 |
| BSP module | 2 | NA | NA |

**Table 4:** Labor cost (unit man-day) of developing typical functionalities on MirSwitch, SONiC-vs and SONiC-bmv2.

**Software switch.** There are numerous software switches that can be used for network emulation. Some software switches such as Mininet [29] and vendor images, lost the fidelity in the control due to their non-SONIC control plane. The SONiC-based software switches including SONiC-vs [20] and SONiC-bmv2 [16, 39, 40] introduced issues such as rule conflicts and ECMP hashing inconsistency, and lost the fidelity in the data plane. Meanwhile, these approaches introduced overmuch labor for developing data plane functionalities. Although the ASIC simulator provided the high-fidelity data plane, it was not practical because it provided extremely low packet forwarding rate. In contrast, MirSwitch differs these approaches. On the one hand, MirSwitch supports most of packet forwarding functionalities and achieves high packet forwarding throughput without introducing much development cost. On the other hand, MirSwitch takes advantage of the environment adaptor to adapt to the hardware components heterogeneous among vendors. Thus, MirSwitch is able to mimic the behavior of whitebox switch in a high-fidelity way.

## 7 Conclusion

This paper presents a *composable emulation framework* called MirSwitch, where diverse emulators are stitched together for high-fidelity emulation of whitebox switches MirSwitch faithfully emulating a wide range of functionalities of whitebox switches, while achieving 2.2× forwarding performance improvements critical for forwarding emulation. We have deployed MirSwitch in the production network, which operators have used to troubleshoot 97.5% of a total of 203 issues.

## Acknowledgment

We sincerely thank our shepherd Umesh Krishnaswamy and anonymous NSDI reviewers for their insightful comments. We also thank the teams at Tencent for their advice on MirSwitch. Chuwen Zhang is the corresponding author.

## References


[1] Behavioral model (bmv2). https://github.com/p4lang/behavioral-model.

[2] Bidirectional forwarding detection (bfd) protocol. https://www.juniper.net/documentation/us/en/software/junos/high-availability/topics/topic-map/bfd.html.

[3] Gns3. https://www.gns3.com/.

[4] iperf. https://iperf.fr/.

[5] Performance of bmv2. https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md.

[6] Port splitting. https://www.h3c.com/en/Support/Resource_Center/EN/Switches/Catalog/S12500R/S12500R/Technical_Documents/Product_Literature/Hardware_Information___Specifications/H3C_12500R_SP/.

[7] Sonic architecture. https://github.com/sonic-net/SONiC/wiki/Architecture.

[8] Sonic command line interface guide. https://github.com/sonic-net/sonic-utilities/blob/master/doc/Command-Reference.md.

[9] Sonic roadmap planning. https://github.com/sonic-net/SONiC/wiki/Sonic-Roadmap-Planning.

[10] Syncd. https://github.com/dreamans/syncd/.

[11] Vector packet processing (vpp). https://github.com/FDio/vpp.

[12] Board support package (bsp), 2023. https://broadcom-switch.github.io/of-dpa/doc/html/OFDPA_PLATFORM.html.

[13] Ns-3 network simulator, 2023. https://www.nsnam.org/.

[14] Omnet++, 2023. https://omnetpp.org/.

[15] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.

[16] K. D. Albab, J. DiLorenzo, S. Heule, A. Kheradmand, S. Smolka, K. Weitz, M. Timarzi, J. Gao, and M. Yu. Switchv: automated sdn switch validation with p4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 365–379, 2022.

[17] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, et al. Empowering azure storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[18] R. Beckett, A. Gupta, R. Mahajan, and D. 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, 2017.

[19] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[20] S. community. Sonic virtual switch (vs).

[21] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 254–265, 2011.

[22] Y. Demchenko, S. Filiposka, R. Tuminauskas, A. Mishev, K. Baumann, D. Regvart, and T. Breach. Enabling automated network services provisioning for cloud based applications using zero touch provisioning. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 458–464. IEEE, 2015.

[23] L. Foundation. Software for open networking in the cloud (sonic) moves to the linux foundation.

[24] K. Gao, L. Chen, D. Li, V. Liu, X. Wang, R. Zhang, and L. Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the ACM SIGCOMM*, page 167–181, 2023.

[25] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13:113–129, 2010.

[26] Z. Gao, A. Abhashkumar, Z. Sun, W. Jiang, and Y. Wang. Crescent: Emulating heterogeneous production network at scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1045–1062. USENIX Association, Apr. 2024.

[27] D. Gibson, H. Hariharan, E. Lance, M. McLaren, B. Montazeri, A. Singh, S. Wang, H. M. Wassel, Z. Wu, S. Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, 2022.

[28] D. Guo, S. Chen, K. Gao, Q. Xiang, Y. Zhang, and Y. R. Yang. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 314–335, 2022.

[29] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264, 2012.

[30] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Rule-caching algorithms for software-defined networks. *Techical Report, Princeton University*, 2014.

[31] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613, 2017.

[32] Y. Liu, S. O. Amin, and L. Wang. Efficient fib caching using minimal non-overlapping prefixes. *ACM SIGCOMM Computer Communication Review*, 43(1):14–21, 2013.

[33] Y. Liu, Y. Xiao, X. Zhang, W. Dang, H. Liu, X. Li, Z. He, J. Wang, A. Kuzmanovic, A. Chen, et al. Unlocking {ECMP} programmability for precise traffic control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 87–106, 2025.

[34] C. Miao, Y. Xiao, M. Canini, R. Dai, S. Zheng, J. Wang, J. Bu, A. Kuzmanovic, and Y. Wang. Tensor: Lightweight bgp non-stop routing. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 108–121, 2023.

[35] C. Miao, Z. Zhong, Y. Zhang, K. He, F. Li, M. Chen, Y. Zhao, X. Li, Z. He, X. Zou, et al. Flexwan: Software hardware co-design for cost-effective and resilient optical backbones. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 319–332, 2023.

[36] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, and A. Zhou. The design and implementation of open vswitch. *;login:: the magazine of USENIX & SAGE*, 40:págs. 12–16, 2015.

[37] O. C. Project. Switch abstraction interface (sai): A reference switch abstraction interface for ocp. In *Technical Report*, 2015.

[38] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). Technical report, 2006.

[39] M. Technologiesm. Sai behavioral model. `https://github.com/Mellanox/SAI-P4-BM`.

[40] J. Xing, Y. Qiu, K.-F. Hsu, S. Sui, K. Manaa, O. Shabtai, Y. Piasetzky, M. Kadosh, A. Krishnamurthy, T. E. Ng, et al. Unleashing smartnic packet processing performance in p4. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1028–1042, 2023.

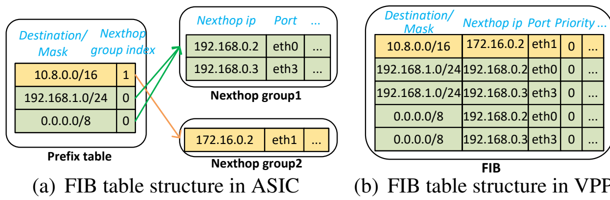
[41] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee ip lookup performance with fib explosion. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 39–50, 2014.

[42] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 599–614, 2020.

[43] Y. Yuan, O. Alama, J. Fei, J. Nelson, D. R. Ports, A. Sapio, M. Canini, and N. S. Kim. Unlocking the power of inline {Floating-Point} operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, 2022.

[44] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252, 2012.

# A Many-to-one mapping for ECMP.




(a) FIB table structure in ASIC (b) FIB table structure in VPP

**Figure 15:** ECMP table structure in ASIC and VPP: (a) ASIC ECMPreuses the IP prefix table, but the next hop index points to a next hop group; (b) Software ECMP uses a flat table

Like IP routing, ECMP is implemented differently in hardware and software, as shown in Figure 15. To save memory resource, hardware ECMP reuse the FIB table, where the next hop index refers to a next hop group instead of a single next hop entry as mentioned before. Then, the hash value is used to select one next hop in the next hop group. In comparison, eVPP uses the existing multi-path and load balancing to implement ECMP, leading to a flat route table with multiple paths towards the same destination IP address, and the hash value is also used to select one path. The solution for ECMP translation is similar to that for IP route, and the adaptor needs to maintain all of the tables locally to keep consistency.

| **Functionality** | **MirSwitch** | **SONiC -vs** | **SONiC -bmv2** |
|---|---|---|---|
| BGP | ✓ | ✓ | ✓ |
| BFD | ✓ | ✓ | ✓ |
| ARP&NDP | ✓ | ✓ | ✓ |
| DHCP relay | ✓ | ✓ | ✓ |
| Interfaces | ✓ | ✓ | ✓ |
| Kubernets | ✓ | ✓ | ✓ |
| Linux kernel dump | ✓ | ✓ | ✓ |
| Load/reload/save | ✓ | ✓ | ✓ |
| Loopback interfaces | ✓ | ✓ | ✓ |
| MGMT VRF | ✓ | ✓ | ✓ |
| NAT | ✓ | ✓ | ✓ |
| NVGRE | ✓ | ✓ | ✓ |
| PHB | ✓ | ✓ | ✓ |
| Portchannels | ✓ | ✓ | ✓ |
| Routing stack | ✓ | ✓ | ✓ |
| DHCP server | ✓ | ✓ | ✓ |
| Startup&running config | ✓ | ✓ | ✓ |
| Subinterfaces | ✓ | ✓ | ✓ |
| MACsec | ✓ | ✗ | ✗ |
| Static DNS | ✓ | ✓ | ✓ |
| NTP | ✓ | ✓ | ✓ |
| Warm restart | ✓ | ✓ | ✓ |
| Warm reboot | ✓ | ✗ | ✗ |
| Software install&manage | ✓ | ✓ | ✓ |
| IPv6 linklocal | ✓ | ✓ | ✗ |
| Troubleshooting commands | ✓ | ✓ | ✓ |

**Table 5:** Comparison of the support for the complete control plane functionalities in the standard whitebox switch.

# B Complete tables of Table 2

We list all functionalities in the standard whitebox switch and their support status.

Table 5 shows the complete control plane functionalities in the standard whitebox switch, whose grey entries are showed in Table 2. MirSwitch supports all the 26 control plane functionalities, while both SONiC-vs and SONiC-bmv2 support 24 of them. The four functionalities that SONiC-vs and SONiC-bmv2 do not support is related to BSP.

Table 6 shows the complete management plane functionalities in the standard whitebox switch. MirSwitch supports all the 15 management plane functionalities, while both SONiC-vs and SONiC-bmv2 support 11 of them.

| **Functionality** | **MirSwitch** | **SONiC -vs** | **SONiC -bmv2** |
|---|---|---|---|
| SNMP | ✓ | ✓ | ✓ |
| System state | ✓ | ✓ | ✓ |
| Radius | ✓ | ✓ | ✓ |
| Syslog | ✓ | ✓ | ✓ |
| ZTP | ✓ | ✓ | ✓ |
| Telmetry | ✓ | ✓ | ✓ |
| Rest API | ✓ | ✓ | ✓ |
| AAA | ✓ | ✓ | ✓ |
| CRM | ✓ | ✓ | ✓ |
| SSH | ✓ | ✓ | ✓ |
| Tacplus Server | ✓ | ✓ | ✓ |
| Platform component firmware | ✓ | ✗ | ✗ |
| Platform specific commands | ✓ | ✗ | ✗ |
| CMIS | ✓ | ✗ | ✗ |
| SFP utilities | ✓ | ✗ | ✗ |

**Table 6:** Comparison of the support for the complete management plane functionalities in the standard whitebox switch.

| **Functionality** | **MirSwitch** | **SONiC -vs** | **SONiC -bmv2** |
|---|---|---|---|
| Drop counter | ✓ | ✗ | ✗ |
| ECN | ✓ | ✗ | ✓ |
| Flow counter | ✓ | ✗ | ✗ |
| Gear box | ✗ | ✗ | ✗ |
| ECMP | ✓ | ✗ | ✗ |
| PFC watchdog | ✗ | ✗ | ✗ |
| QoS | ✗ | ✗ | ✗ |
| Sflow | ✓ | ✗ | ✗ |
| VLAN | ✓ | ✓ | ✓ |
| FDB | ✓ | ✓ | ✓ |
| Watermark | ✗ | ✗ | ✗ |
| ACL | ✓ | ✗ | ✗ |
| VRF | ✓ | ✓ | ✗ |
| IPv4/IPv6 | ✓ | ✓ | ✓ |
| Static routing | ✓ | ✓ | ✓ |
| VXLAN | ✓ | ✓ | ✗ |

**Table 7:** Comparison of the support for the complete data plane functionalities in the standard whitebox switch.

Table 7 shows the complete data plane functionalities in the standard whitebox switch. We can see MirSwitch realizes the highest support ratio of 75%, i.e., supporting 12 out of the 15 data plane functionalities, while SONiC-vs and SONiC-bmv2 support 6 and 5 of them, respectively.Note that we claim SONiC-bmv2 and SONiC-vs does not support ECMP as they cannot guarantee the same hashing value as the forwarding ASIC.