Crescent: Emulating Heterogeneous Production Network at Scale5

Zhaoyu Gao and Anubhavnidhi Abhashkumar, *ByteDance;*
Zhen Sun, *Cornell University;* Weirong Jiang and Yi Wang, *ByteDance*

https://www.usenix.org/conference/nsdi24/presentation/gao-zhaoyu

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Crescent: Emulating Heterogeneous Production Network at Scale

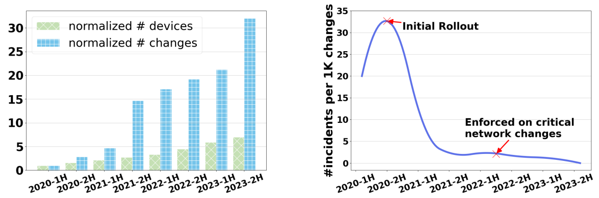
Zhaoyu Gao*, Anubhavnidhi Abhashkumar*, Zhen Sun◇, Weirong Jiang*†, Yi Wang*
ByteDance, Cornell University*◇

Abstract

This paper presents the design, implementation, evaluation, and deployment of *Crescent*, ByteDance's network emulation platform, for preventing change-induced network incidents. Inspired by prior art such as CrystalNet, *Crescent* achieves high fidelity by running switch vendor images inside containers. But, we explore a different route to scaling up the emulator with unique challenges. First, we analyze our past network incidents to reveal the difficulty in identifying a safe emulation boundary. Instead of emulating the entire network, we exploit the inherent symmetry and modularity of data center network architectures to strike a balance between coverage and resource cost. Second, we study the node-to-host assignment by formulating it as a graph partitioning problem. Evaluation results show that our partitioning algorithm reduces the testbed bootup time by up to 20× compared with random partitioning. Third, we developed an incremental approach to modify the emulated network on the fly. This approach can be 30× faster than creating a new testbed of the same scale. *Crescent* has been actively used for three and a half years, which led to a significant reduction in change-induced network incidents. We also share *Crescent*'s success in many other use cases and the critical lessons learned from its deployment.

1 Introduction

As one of the largest and fastest-growing global online service providers, ByteDance's physical network infrastructure has expanded rapidly in recent years to meet the explosive business demand [5, 7]. Such a wild expansion has created a hyperscale and heterogeneous global network that consists of medium- and large-scale data centers (DCs), regional and global wide area networks (WANs), points of presence (PoPs) of all sizes, and virtual DCs (vDCs) from multiple public cloud providers. We have network switches from nearly all the major switch vendors and multiple generations of network architectures co-existing across different DCs/PoPs. As



(a) # network devices and changes (normalized using 2020-1H). (b) Trend of # network incidents per thousand changes.

Figure 1: Growth of network size, network changes, and network incidents in last 3.5 years.

shown in Figure 1a, the total number of switches in our networks has grown by 7.5× in the past three and a half years. Meanwhile, the number of changes made on these networks has increased at an even higher rate. The changes, mainly on topology and switch configurations, include modifications to our network structures, routing policies, device software, peering with cloud and Internet service providers (ISPs), etc. Many such changes require network operators' careful designs, detailed Methods of Procedure (MOPs), and cautious executions. But even with those efforts, we still ran into a series of outages due to missed or unforeseen issues. In the second half of 2020, network changes accounted for approximately one-third of our incidents. Since the networks kept growing bigger and increasingly complex, it became a daunting challenge for our network operators to think through all scenarios and identify all possible issues before performing changes to the production networks.

To reverse the alarming trend of the change-induced incidents, we developed *Crescent*, a high-fidelity emulation platform to provide a production-like environment for network operators to test and verify their changes. As shown in Figure 1b, we rolled out *Crescent* during the second half of 2020. Since then, we have witnessed a steady decrease in network incidents along with increased deployment and usage of *Crescent*. Around the mid of 2022, we enforced *Crescent* as a mandatory requirement to conduct critical network changes, which has further bent down the curve.

†corresponding author

We are aware that a plethora of solutions have been proposed in recent years to prevent network incidents caused by configuration changes. One popular alternative to emulation is control plane verifiers (CPV) [12,29,55,57,59] which employ either simulation or formal modeling to verify the impact of configuration changes on specific reachability properties of the network. But most of them cannot detect vendor-specific behaviors (VSBs), which have caused many of our incidents (§2.2). Hoyan [55] attempts to model VSB by actively tracking the differences between its analysis output and the real production behavior. But, it can only catch VSBs that are present in production and requires extra efforts to update its VSB models. Many VSBs that caused our incidents (§2.2) were neither documented nor seen in production beforehand. Other limitations of CPV have been discussed in [19,41].

Network emulators [3,4,8,10,23,40,41,53] overcome the aforementioned CPV limitations by running switch vendor software images in virtual machines (VMs) or containers. Nevertheless, very few emulators can scale to emulate large networks, and most require manual efforts to adapt production configurations to the format supported by the vendor images (§4.2). The closest work to *Crescent* is CrystalNet [41], which leveraged cloud VMs to scale up and proposed reducing the size of the emulated network by finding a *safe static boundary*. But as admitted in [41], the boundary found by CrystalNet is based on certain assumptions from their routing policies that do not apply to everyone. Analysis of our past incidents confirms that such a boundary is much harder to find in reality (§2.2). This led us to a different route from CrystalNet in scaling up the emulator. The design and implementation of *Crescent* aim to answer the following questions, which are also the main contributions of this paper.

First, if it is hard to identify a safe emulation boundary, *does it mean we have to emulate the entire network?* Based on the analysis of our past incidents as well as the experience from other hyperscalers [13, 15, 32, 41, 42, 48, 51], we see that the changes on different devices have different impacts. High-level core devices (e.g., WAN devices) tend to have more complicated configurations, and the incidents due to the changes in these devices tend to be more severe. On the other hand, the devices inside a DC are normally in modular and symmetric architecture, and incidents caused by the changes in these devices have a limited blast radius. Inspired by these observations, we proposed emulating a baseline topology that includes all the core devices while sampling the lower-level non-core devices.

Second, the above baseline topology is still quite large. Because of the high resource requirement to run switch images, we need many hosts to emulate a large network. Then *will it matter how we map the network nodes to different hosts?* CrystalNet mapped nodes to many VM hosts but did not answer this question. We conducted thorough experiments to show that the performance was heavily impacted by the node-to-host assignment. We formulated it as a graph partitioning

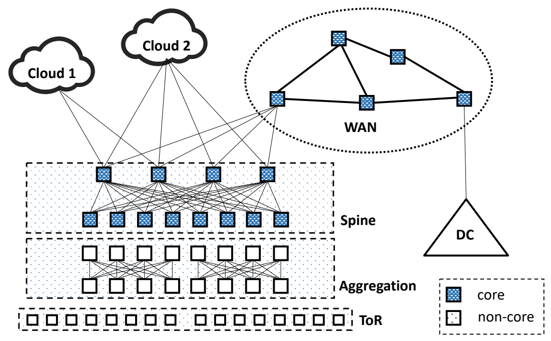


Figure 2: Simplified view of ByteDance's network.

problem and proposed a variant of the community detection algorithm to solve it. Evaluation results show that our algorithm resulted in 20× improvement in testbed bootup time compared with random partitioning.

Third, *how to test a network change plan involving devices (i.e., devices under test, namely DUT) that are not part of the above baseline topology?* We connect DUT to the baseline topology through controlled expansion to form a connected graph. And instead of rebuilding the testbed for the expanded network, we create the expanded nodes and wire them dynamically to the baseline topology which has been emulated in a pre-built testbed called *canary testbed*. Such an incremental emulation scheme improved the testbed ready time for testing a network change plan by 30×.

Last, emulation is only the first step; it reproduces the impact of a configuration change but still requires someone to analyze the updates' impact. Instead of asking users to regularly monitor and interact with the emulated network, *Crescent* takes a more proactive approach. It tightly integrates with a diverse range of efficient verification tools to proactively detect errors and unexpected behaviors that arise due to changes in real-time.

The rest of the paper is structured as follows: §2 outlines the motivation and describes the challenges in our network. The design of *Crescent* to address these challenges is proposed in §3, while scalability is tackled with a novel graph partitioning algorithm in §4. The incremental emulation approach is detailed in §5, along with various verification and monitoring tools integrated into *Crescent* to automatically detect issues in an emulated network in §6. Evaluation results for *Crescent* are presented in §7, with other use cases described in §8. Lessons learned from building and running *Crescent* for 3.5 years are shared in §9, with related works discussed in §10. Finally, §11 concludes the work.

2 Background and Motivation

2.1 ByteDance's Network

Our network mainly consists of three components: (1) a set of data center networks (DCNs) at different scales, (2) a global

WAN and several regional WANs that connect all DCs and PoPs, and (3) vDCs from various third-party cloud providers that are connected to our major DCs. Figure 2 gives a simplified overview of our network. All these components together form one of the world's largest networks, serving as one of the largest online services [7].

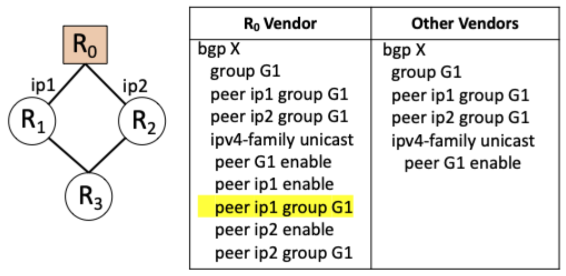
ByteDance's DCN is a variant of traditional multi-stage Clos network that can be described in the following simplified model. The bottom layer consists of ToR (Top of Rack) switches, each of which directly connects a rack of servers. These ToRs also connect to the middle layer aggregating connections from the bottom layer. These middle layer switches connect to the top layer, that comprises the core switches that aggregate middle layer fabrics and connect to other DCNs, WANs, and clouds. To be consistent with terminologies used in other works, e.g., [32], we call the middle layer "aggregation layer", and the top layer "spine layer". We define the *core devices* in our network as the WAN and spine layer devices, and the other devices as non-core devices [13, 42].

For the third-party clouds, we have no direct control over their physical devices. But we can configure certain routing policies in their management portals to control the peering and the traffic between our DC's border devices and the clouds. These policies are applied based on different demands and thus sometimes are not standardized, i.e., different regions may have different policies. Even in the same region, there can be different numbers of spine devices connected to a same cloud. As a result, the configurations on core devices are complicated and non-standardized.

Different from those core devices, the configuration of non-core devices in DCN is usually highly standardized. It entails that the devices on the same level tend to exhibit the same routing behaviors (e.g., receive the same routes from higher level devices, and announce the same granularity of routes to higher level devices). The high standardization for the non-core device configurations is ensured by (1) generating configuration for all non-core devices using automatic script and config templates; and (2) checking the running configurations periodically to detect violations against DCN design rules [26, 38]. Many other hyperscalers [13, 15, 32, 41, 42, 48, 51] also adopt the practice of standardizing and simplifying the configurations of non-core devices in DCN. Leveraging symmetry and synergy created by such standardization and simplification to streamline network analysis has been studied by many works in network verification [15, 16, 30, 43]. To the best of our knowledge, we are the first to extend this idea to the area of network emulation.

2.2 Incidents Analysis

In this section, we show two examples of our past network incidents. Then, we give a statistical analysis of all change-induced incidents in our network over the past years. These incidents motivate our features, design choices, and use cases.



R_0 Vendor	Other Vendors
bgp X	bgp X
group G1	group G1
peer ip1 group G1	peer ip1 group G1
peer ip2 group G1	peer ip2 group G1
ipv4-family unicast	ipv4-family unicast
peer G1 enable	peer G1 enable
peer ip1 enable	
peer ip1 group G1	
peer ip2 enable	
peer ip2 group G1	

Figure 3: An illustration of incident B.

2.2.1 Incidents Examples

Incident A. In one incident, a config change was made on a global WAN device to prepend an AS number to the AS-PATH of a DC route. However, the same AS number of global WAN was used for prepending, which triggered an unexpected VSB: while some vendors prevent AS loop only for eBGP, others consider iBGP too. Each of our global WAN devices peers with a route reflector (RR) through iBGP. The vendor of RR did not consider the route with prepended AS number forming an AS loop and propagated the route to the rest of global WAN. But the vendor of the rest of global WAN treated this as an AS loop and dropped the route. To prevent such an incident, we need to emulate at least the global WAN device under change, the RR and another global WAN device. To detect this issue via end-to-end connectivity test, we need to emulate more devices, i.e., from the origin DC of the affected route and from the DC connected to the other global WAN.

Incident B. In another incident, a network operator drained the traffic on a switch by disabling the peers in a BGP peer group and then re-enabling the peers to undrain the traffic. However, for this vendor, disabling the BGP peer group operation will automatically delete all statements that associate peers to a peer group under IPv4-family section (highlighted in Figure 3). Later re-enabling the peers will not automatically add those statements back. Figure 3 illustrates this incident: when doing this operation on R_0, the highlighted statement disappeared after the traffic undrain operation. The BGP peer group $G1$ has a route policy to extend `AS-PATH` length to 2 from R_1 or R_2 to R_0. However, the newly enabled peer $ip1$ is not associated with $G1$, thus has a shorter `AS-PATH` compared to the other peers, e.g., $ip2$, leading to all traffic going through the link from R_1 to R_0, which caused severe link congestion. To capture this issue, emulation must include all the core devices shown in the Figure 3, emulate R_0 with its respective vendor image, and compare the routes before and after the change.

We can draw 2 quick takeaways from the above incidents:

1. VSBs are notoriously hard to prevent because people are often unaware of VSBs, most of which are not well documented. These VSBs could lead to all kinds of incidents, also observed in prior works [41, 55]. High-fidelity emu-

lation using switch vendor images is the effective option to capture the unknown VSBs.

2. Emulating only the devices under test is insufficient to catch the impact of a change. It is difficult to identify the minimum scope of the network to be emulated, especially when the blast radius of the impact of a network change is hard to predict [32, 42]. More discussions are in §2.3.

2.2.2 Statistical Analysis

We conduct a statistical analysis over the O(100) incidents that happened in our networks in the last 3 years. Out of them, about 1/3 network incidents were caused by network changes and thus can be potentially captured by emulation. The others are caused by issues such as traffic bursts, capacity losses, device failures, etc. Among these change-induced incidents, **30% of them involved VSBs**. The impact of these incidents spans all 5 severity levels, among which level 1 means the most severe incident, while level 5 means the least. Approximately 80% of change-induced incidents are identified as level 3 or higher, resulting in substantial business losses. By further examining the change-induced network incidents, we found that **over 90% of these incidents happened on our core devices.** This is likely because the configurations on core devices are more complex than non-core devices, i.e., more routing protocols besides BGP (e.g., ISIS, OSPF, SRTE, etc) are used, more route policies are applied, and more complex topologies are formed. Such complexity is more likely to trigger various unknown VSBs.

2.3 Cost v.s. Coverage

To prevent the above incidents, an intuitive solution is to emulate the entire network, but it comes with a high cost (with respect to resource usage). As the number of devices to be emulated increases, so does the cost. We list all the vendor's images, for one or more versions, and the corresponding resource requirements in Table 1. Note that different images have different resource requirements , e.g., memory usage, the primary resource bottleneck, ranges from 1 to 16 GB.

An alternative is to find a tighter safe boundary as defined in CrystalNet [41]. After analyzing the incidents that happened in our network from the past 3 years, we find that their strategy to find a safe boundary does not apply to other networks like ours. First, it assumes that AS-PATH removal and rewrite are rare, leading to shortened boundaries. However, in our network, these actions are frequently utilized, particularly on core devices, and have caused **15%** of our incidents. Second, we find the expansion algorithm proposed by CrystalNet is primarily about expanding the network to higher layers. This is not enough for us to find a safe boundary to prevent **55%** inter-DC incidents in our network. For instance, to prevent Incident A, we need to include our global WAN and two DCNs to detect the issue. But CrystalNet can only expand until the global WAN that contains all DUTs of this change. On the other hand, it's crucial to expand the network downwards and include both DCs. Similarly, in Incident B, it's important to emulate all lower-level devices and observe that after the change, R3 will only forward traffic to R1 and not both R1 and R2. Overall, CrystalNet's expansion algorithm can potentially miss approximately **60%** of the incidents, among which 10% involve both the limitations. Moreover, we find that in some scenarios, even when their assumption holds, e.g., only `AS-PATH` prepend policy exists, an incident may still happen due to other issues, e.g., ECMP reduction in Incident B.

Vendor	Image type	vCPU	RAM(GB)	Size (GB)	Port limit
v1	container	1	2	1.60	unlimited
	VM	2	4	2.08	64
v2	VM	2	8	1.47	64
v3	VM	2	8	4.31	128
v4	VM	2	4	1.57	10
	VM	8	16	4.14	100
v5	container	1	2	0.43	64
v6	VM	2	4	1.34	10
v7	VM	4	5	1.48	96

Table 1: Specifications of various vendor images (*v1-7*).

Based on our analysis result, it is essential to always emulate the core devices in our network to avoid those 90% incidents that are caused by network changes on our core devices. Besides, incidents that happen on core devices tend to cause a larger blast radius than non-core ones. Meanwhile, we are not trading off safety for cost. By including the sampled non-core devices into the canary testbed (§3), the rest (less than 10%) of network incidents caused by changes on non-core devices can also be captured by *Crescent*, assuming their configurations are standardized. By doing this (i.e. including all the core devices and sampled non-core devices), we only need to emulate less than 2% of our entire network fleet.

2.4 Scalability over Multiple Hosts

Even though we choose to emulate a small fraction of our entire network, it still contains thousands of devices that we cannot emulate on a single host due to the high resource requirement of emulating each device (Table 1). A natural idea is to employ multiple hosts, as CrystalNet [41] did. But, it is unclear how CrystalNet maps thousands of nodes onto multiple hosts. In other words, how to partition a given network to let each host emulate a subset of the network? While similar problems have been studied in other contexts e.g. [52], none of them applies to the emulators like CrystalNet and *Crescent*. We observe that a multihost testbed using different node-to-host assignment schemes can yield significant performance differences. For example, a simple random partitioning may not scale at all (§7.2). We study this challenge by uncovering the system bottleneck, formulating it as a graph partitioning problem, and solving it with a variant of the community

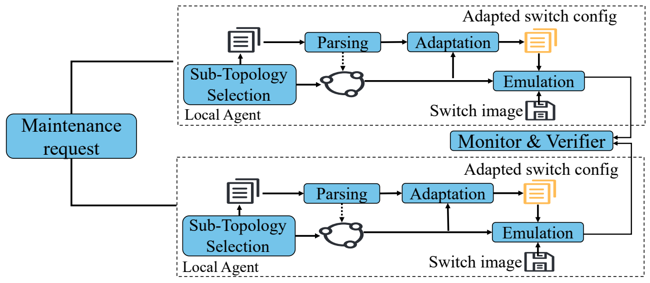


Figure 4: An illustration of *Crescent* workflow to create a new multihost canary testbed.

detection algorithm (§4.3).

3 *Crescent* Design Overview

We build a canary testbed to emulate the baseline topology that includes all the core devices and sampled non-core devices. We sample the non-core devices by selecting one plane out of multiple planes, and selecting one device per vendor on each level of a plane. To keep it simple and consistent, our strategy is to always pick devices from the 1st plane on every level, ensuring that aggregate routes can be activated, the same as in our production network. Also we integrate each testbed with proactive monitoring and verification tools to automate the detection of impact of changes.

3.1 Multi-Host Canary Testbed

We run the large-scale canary testbed by distributing thousands of emulated nodes across a cluster of baremetal servers using a multi-host setup (§4.1). We employ a novel partitioning algorithm to minimize the number of cross-host links, which significantly reduces the canary testbed bootup time (§4.3) as well as the time to connect DUTs to a canary testbed.

Figure 4 shows the workflow in a multi-host setup. Each host runs a local agent that manages emulation on that host. When the agent receives input for a maintenance request, it first identifies which new nodes and links need to be emulated on that host. Next, it fetches the configurations of new nodes from production networks and parses and adapts them to be compatible with vendor images. After this, the agent launches and connects each node's containers with the appropriate configurations and vendor images. Finally, when required, it sends the emulation data to the monitoring and verification tools for proactive analysis.

3.2 Connect DUTs to Canary Testbed

Instead of creating a large-scale testbed from scratch for every network change request, we run multiple instances of nonstop canary testbeds. For each request, we connect its DUTs to one idle canary testbed.

If DUTs are not in the baseline topology, *Crescent* needs to emulate these DUTs in new containers and connect them to the canary testbed, along with the intermediate nodes between them. *Crescent* achieves this using a novel expansion algorithm that ensures the heterogeneity of the devices along the paths from DUTs to canary testbed (§5.1). The expansion algorithm considers different factors (e.g., vendors, planes, levels) when exploring the paths from DUTs to the running canary testbed. To allow DUTs and the intermediate devices to connect to and disconnect from the canary testbed on the fly, *Crescent* supports dynamic link addition/removal (§5.2).

It is more cost-effective to run a nonstop canary testbed than rebuilding a new one. Also based on our historical data, over 50% of our network changes happened on core devices. It suggests that there is no need to create any new containers for most of the requests, as we already include all core devices in canary testbed. Besides, the time to connect DUTs to canary testbed is also much lower than the time to rebuild a new canary testbed (§7.1).

3.3 Proactive Verification and Monitoring

Emulation alone can not detect network issues (e.g., loop and blackhole) at scale. It must be combined with various verification and monitoring tools (§6) to automatically analyze the impact of change on an emulated network. After automatically applying each command of the MOP to DUTs, we take a snapshot of the converged network and proactively run the verification and monitoring tasks to detect potential issues caused by each command.

4 Building Canary Testbed

A large-scale high-fidelity canary testbed containing all core devices in our network is the key to the design of *Crescent* as described in §3. In this section, we discuss our approach to address the scalability challenge in running a large-scale canary testbed. We formalize the scalability challenge as a graph partitioning problem and then propose a heuristic algorithm to solve the problem.

4.1 Network Emulation

Crescent's network emulation platform shares similarities with most of the other container-based network emulators [3, 8–10, 41, 53]. Each node is wrapped in an individual container with a switch OS running inside. A link on the same host is emulated by a veth pair, while a link across hosts is emulated by OVS bridge. Figure 5 shows an example of such a container network implemented in *Crescent*. We defer the implementation details to §A.

We run the canary testbed on a cluster of baremetal hosts. On each host, *Crescent* runs a local agent that interacts with OS for node and link creation. Running a multi-host canary

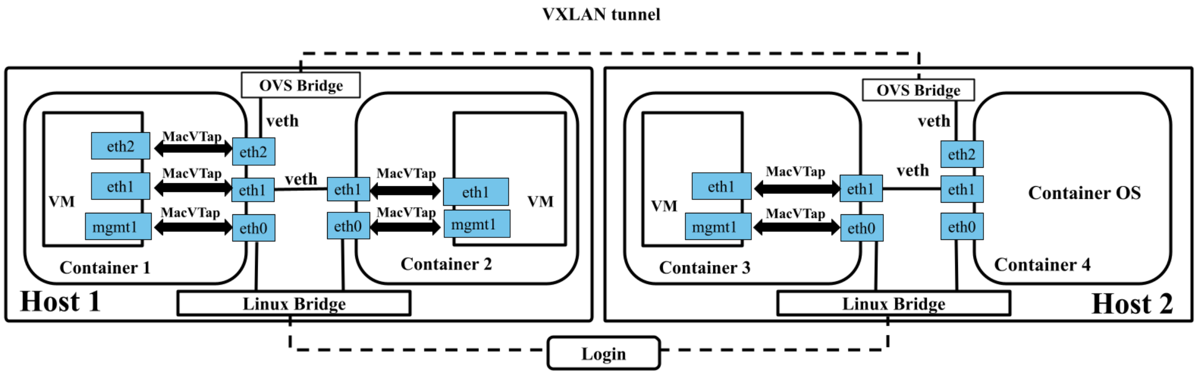


Figure 5: An illustration of *Crescent* implementation with virtual nodes (container-based and VM-based) and links.

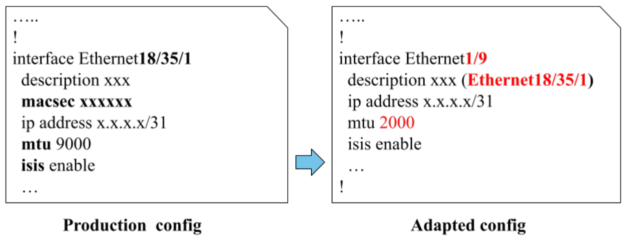
testbed is straightforward with this setup. When booting up a testbed, we send the same topology input to all the host agents. The input topology specifies the host each node is running on. Then, each local agent boots up the local portion of the testbed based on the input topology, including the nodes and links. The agent creates local veth pairs between nodes on the same host and OVS bridges for cross-host links.

4.2 Multi-Vendor Config Adaptation

Upon the testbed bootup, *Crescent* immediately loads the configuration file into the emulated nodes. However, we find the production configuration cannot be directly applied to the emulated nodes due to the limitations of the vendor's software switch images. To that end, *Crescent* has to adapt the production configuration into an image-compatible form before loading it into the emulated nodes. Figure 6 shows an example where *Crescent* adapts an interface's configuration to ensure ISIS [1] is activated on this interface. First, it is important to note that the interface names in emulation may not precisely match those used in production. Therefore, it is necessary to rename these interfaces and maintain a mapping between the emulated interface names and their counterparts in the production environment. Second, certain security-enhanced commands, like "macsec," can prevent ISIS from being activated and thus must be filtered. Last, the MTU (Maximum Transmission Unit) value in the production environment is typically greater than the default MTU value supported by our host. Thus, we must reduce the MTU value in the adapted config to a value lower than the default MTU value on our host. Config adaptation, which involves all vendors and image types, is critical to automating large-scale testbed creation. Whenever a new, previously unseen configuration template is introduced, extension of *Crescent* is necessary to support it.

4.3 Multi-Host Partitioning

A large-scale canary testbed with thousands of nodes must run in a distributed manner. Nonetheless, prior research, such



```
.....
!
interface Ethernet18/35/1
  description xxx
  macsec xxxxxx
  ip address x.x.x.x/31
  mtu 9000
  isis enable
  ...
```

Production config

```
.....
!
interface Ethernet1/9
  description xxx (Ethernet18/35/1)
  ip address x.x.x.x/31
  mtu 2000
  isis enable
  ...
!
```

Adapted config

Figure 6: An example of configuration adaptation to a simple interface config to activate ISIS status on this interface.

as Crystalnet [41], does not delve into the detail on how to partition a large-scale testbed across a set of hosts (or VMs on cloud). Other works [56,60] focuses on minimizing cross-host communication overhead by reducing the number of cross-host links. However, we find that cross-host communication overhead is negligible (§7.3). *Crescent* also tries to reduce the number of cross-host links, but for a different reason, i.e., to reduce testbed creation and DUT connection time.

Cross-host Link Creation Overhead. A simple strawman approach is to partition all the emulated nodes across hosts randomly, following a uniform distribution. However, we find that it creates thousands of cross-host links, which incurs a significant overhead on creating a testbed and connecting DUTs. This phenomenon is because the overhead to create cross-host links increases linearly in the Linux kernel. It is not specific to the OVS bridge, but also applicable to the Linux bridge (used by CrystalNet [41]). As shown in Figure 7, both Linux bridge and OVS bridge creation incur a linear overhead (the x-axis is in log scale), i.e., the more links there are on the host, the longer it takes to create a new link[1]. This overhead is inevitable even if all bridges are created in parallel.

Consequently, we must minimize the number of cross-host links. The benefit has two folds. First, connecting DUTs usually needs to create O(100) new links (§5.2), thus, by reducing

[1]Besides merely creating the bridge, the bridge creation operation we consider here also includes the operations of creating veth pair, binding VXLAN port to the bridge, and bringing up all these virtual devices

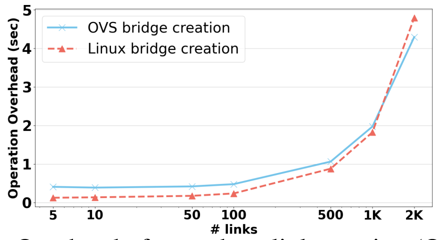


Figure 7: Overhead of cross-host link creation (OVS bridge and Linux bridge).

the number of cross-host links, we can substantially expedite the process of connecting DUTs. Second, when booting up the multihost testbed, with fewer cross-host links to create, the overall bootup time can also be reduced.

Problem Model. We model the cross-host link minimization problem as follows. Given a graph $G = (V, E, W_V, W_E)$, where V is the set of vertices, E is the set of edges, $w_v \in W_V$ is the weight of node v and $w_e \in W_E$ is the number of links between the vertices connected by e. In practice, we find that memory usage is usually the bottleneck. Thus the node's weight is the memory a node needs, which is constrained by the server's total amount of available memory C. Given the server capacity C, the goal is to find a partition P for V with n disjoint sets $V_1, V_2, \ldots, V_n$.

$$\begin{aligned}
\min \quad & \sum_{i,j} \sum_{e \in E_{ij}} w_e \\
\text{s.t.} \quad & 1 \le i, j \le n \\
& E_{ij} = E \cap V_i \times V_j \\
& V = V_1 \cup V_2 \cup \ldots V_n \\
& V_i \cap V_j = \emptyset \\
& \sum_{v \in V_k} w_v \le C, 1 \le k \le n
\end{aligned} \tag{1}$$

Partitioning Algorithm. This is a graph partitioning problem, which is NP-complete [56]. It has also been widely studied in previous literatures [21, 56, 60]. In the field of social networks, the problem is called community detection [18, 22, 31]. Namely, detecting closely connected vertices in a graph merely based on the links rather than other attributes, which is different from clustering.

To solve the problem, we explore the traditional community detection algorithm [31]. The traditional approach begins with a set of all n vertices in the graph, with no edges between them. Then, it adds edges between pairs one by one in order of their weights. As edges are added, the resulting graph shows a nested set of increasingly large components (connected subsets of vertices), eventually forming a partitioning scheme.

We made a few modifications to the traditional community detection algorithm to solve the optimization problem (1). First, we define the edge weight when merging two components with newly added edges as the geometric mean of the two components' weights. Second, we add a capacity constraint to the traditional algorithm to ensure the size of each partition does not exceed a single host memory capacity. Last, to avoid being trapped in a local optimum partitioning scheme, we follow an exponential stochastic process to randomly choose the next edge to add to the graph. These modifications ensure that the algorithm generates a scheme with well-balanced partitions, all within the server's capacity. We defer the detailed algorithm pseudocode to §B.

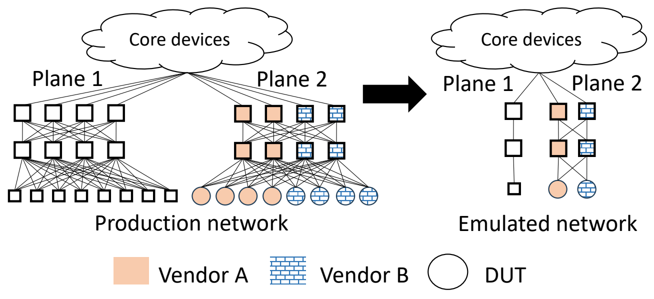


Figure 8: DUT sampling and topology expansion.

5 Connecting DUTs to Canary

For a maintenance request, if DUTs are not in the canary testbed, *Crescent* needs to connect these DUTs to the canary testbed, along with the intermediate nodes between them.

5.1 Topology Expansion

As mentioned in §2.1, the configuration on non-core devices is auto-generated, and DCN topology is highly standardized. We take advantage of this simplicity and standardization to do a more controlled expansion. While finding a generic algorithm that works for all networks may be challenging, we have successfully tested this strategy in hundreds of maintenance requests without causing any issues in production.

When expanding from DUTs, the algorithm ensures that each node has at least one neighbor for each vendor, level, and plane. The number of devices included in the expansion process grows exponentially. However, given that there are at most 3 to 4 levels from the canary testbed to the lowest non-core device (i.e., ToR), at most 3 different vendors on the same level in our DCN, and tens of planes, each DUT will only expand to tens of devices to establish paths up to the canary testbed at worst.

If DUTs are on the lower level, they also exhibit similar routing behavior per level. And we can apply the same expansion rules to select DUTs for emulation during maintenance.

Figure 8 presents an example of applying the algorithm on a change with 8 DUTs in a plane. It only shows the result for two planes. In this example, all 8 ToR DUTs have a similar configuration, and the devices belong to two vendors. After sampling, we reduce the emulated DUTs to 2. And besides the 2 DUTs, we only emulate 7 instead of 24 additional nodes. We explain the expansion algorithm in §C.

5.2 Dynamic Connection

To connect the DUTs to a canary, *Crescent* sends a new topology input to all the host agents of the canary testbed. Each agent compares its local topology information with the newly received topology to find the new links it needs to create. Similarly, when removing links, the local agent must determine the existing nodes as well as the links it needs to destroy based on the topology information.

CrystalNet [41] does not support dynamic link addition and removal; thus, we extend its techniques to enable this functionality. Link addition/removal requires rebooting nodes on both ends of the link because no vendor image supports dynamic interface addition/removal, i.e., software switch OS can only detect interfaces during bootup. To solve this issue, we decouple the network stack from the actual DUT container's runtime. For each actual DUT container, we create a container to hold the network stack for the DUT container. This way, links can be added after stopping the DUT container, and we can restart the DUT after adding all the new links.

6 Proactive Verification and Monitoring

We run various verification and monitoring tasks to test, analyze, and troubleshoot the changes made in the emulated network before pushing it to production. Motivated by our incidents (§2.2), our tools are tailored to catch the following behaviors: routing loop, blackholes, unexpected ECMP reduction, and unexpected route withdrawal/change/churn. We run four major tasks to cover these issues.

Config Checker. A static configuration analysis tool similar to RCC [26] analyzes the routing-related segments of the parsed switch configurations to identify basic syntactical and semantical errors. We first convert vendor-specific configurations into a vendor-independent format. Then our config checker analyses them to identify common errors like undefined variables [26] (referring variables without defining them), configuration dissimilarity (configs deviating from the template [38]), etc. This tool's primary goal is to detect non-standard configurations on non-core devices and to identify any new configuration errors on DUTs.

Route Differ. Our route-differ aims to detect local forwarding changes at each node. It periodically captures the emulated nodes' latest FIB (Forwarding Information Base) and compares it to the previous snapshot to detect route changes. A centralized full-scale implementation may not perform well in emulation due to resource scarcity. As a runtime optimization, we developed a lightweight distributed version to run the diff locally within each container and only send the result to the central process for aggregation. The route differ plays a crucial role in identifying incidents that lead to reduced ECMP next hop count, resulting in congestion (e.g., Incident B as discussed in §2.2). Additionally, this helps in detecting unexpected additions or removal of routes.

Pingmesh. We run end-to-end ping connectivity tests between all the emulated servers of a network. It is much simpler than the original Pingmesh [33]. We run a script inside each emulated server to ping the other IPs and to only send the ping failure messages to the central process for aggregation. The main objective of this task is to identify connectivity-related incidents, e.g., Incident A in §2.2

Data Plane Verifier. Data plane verifiers (DPV) model how packets will be sent in the network across a set of forwarding tables and verify if they satisfy specific policies, like detecting routing loops, blackholes, waypoint violations, etc. We built our DPV based on APKeep [58] with customized optimizations. The occurrences of blackhole and routing loop incidents are primarily detected by DPV.

7 Evaluation

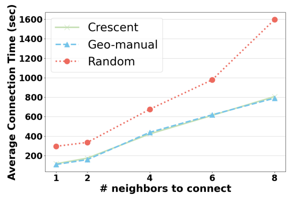
In this section, we evaluate *Crescent* performance for (1) the time to connect different numbers of DUTs to the canary testbed, (2) canary testbed bootup time at different scales, (3) the time to run the most commonly used MOP commands for a canary testbed, and (4) the time to run various verification and monitoring tasks.

We show the impact of three different partitioning schemes on the performance metrics. The first scheme is random partitioning that assigns all n emulated nodes to k hosts randomly, so that each host has $\frac{n}{k}$ nodes to run. In the experiment, we find that the random partitioning scheme over the original testbed creates too many cross-host links, which prevents the testbed from booting up successfully. The second scheme is done by our proposed algorithm (§4.3). We call it *Crescent* scheme. The last scheme is done by manually partitioning the canary testbed into k parts, with each part containing nodes within a specific geographic affinity region This scheme is referred to as the *Geo-manual*. The *Geo-manual* scheme is provided by one of our network experts who is familiar with our network architecture.

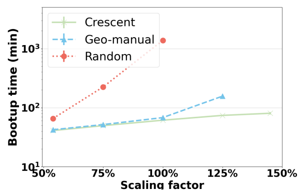
We do all the experiments on a dedicated cluster of bare-metal servers, each equipped with a 2.10GHz 128-core CPU and 500GB DDR4 RAM. Currently, a canary testbed without any connected DUTs takes $k = 4$ servers to run. Unless otherwise specified, all results shown in this section are the median over 10 runs.

7.1 Connection Time

The *Crescent* partitioning scheme (§4.3) minimizes the number of cross-host links. Thus it is expected to outperform random partitioned testbeds when connecting DUTs. The connection time is measured as the time to establish cross-



(a) Connection time.



(b) Bootup time.

Figure 9: Connection time and testbed bootup time.

host links between two pre-existing testbeds (i.e., canary and a small testbed with DUTs).

Figure 9a shows the testbed connection time for different numbers of neighbors (i.e., aggregation level devices) to connect to the canary testbed. The geo-manual scheme almost yields the same connection time as the *Crescent* scheme, while it takes 2-3 times longer to connect the same testbed to the canary for the random scheme. This is because the random scheme has the most cross-host links. Note that connecting DUTs to the canary testbed yields a much better performance than creating a new large-scale canary testbed. For example, it only takes about 2 minutes to connect a DUT to the canary, while it takes about an hour on our platform to create a new canary testbed, resulting in a 30× performance improvement.

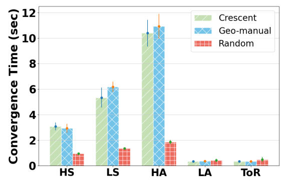
7.2 Bootup Time

We show *Crescent*'s scalability by adjusting the size (i.e. the number of nodes) of the canary testbed. We define *scaling factor* as the ratio of the size of a scaled canary testbed over the size of the original canary testbed. For example, if the original canary testbed contains N nodes, and the scaled canary testbed contains $0.75N$ nodes, then the corresponding scaling factor is 75%. When scaling down the canary testbed, we sample core devices on the spine level to ensure that the canary remains completely interconnected. When scaling up, we add more non-core devices to the canary testbed. In this experiment, we set the number of hosts proportional to the size of the scaled testbed. For example, given that each host runs about 25% portion of the original canary testbed, we use 5 hosts to run a 125% scaled testbed.

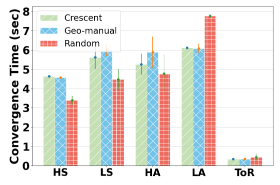
Figure 9b shows the system bootup time with different scaling factors for the three partitioning schemes. It shows only *Crescent* scheme is scaling, while both the other two stop scaling at a certain point, i.e., 100% for Random and 125% for Geo-manual. A scheme is considered not scalable when it takes too much time to boot up a new testbed. In this experiment, we set the limit to 2 hours. The reason that the Geo-manual strategy fails to scale is because there are a few DCNs whose size are much larger than the others, thus it ends up with an imbalanced partitioning scheme when adding non-core devices. *Crescent* partitioning scheme also outperforms the other two in terms of bootup time. For example, it is 20× faster to boot up the original canary testbed (i.e., 100%) with *Crescent* partitioning scheme than the random partitioning.

We need to reboot canary testbeds occasionally. For example, when there is a significant change to our production network (e.g., a new DC or a new backbone dataplane is added), we need to reshard the testbed to rebalance all the partitions. To that end, we use the proposed algorithm (§4.3) to generate a new partitioning scheme, then reboot canary testbeds in production with the new partitioning scheme one by one. During the service reboot, we want to introduce minimum impact to the online service, i.e., DUTs of planned maintenances can still connect to the running canary testbeds. In that case, *Crescent* scheme exhibits a better bootup time, about 10% faster than Geo-manual scheme. Moreover, the Geo-manual strategy stops scaling when running 125% size of the original canary testbed. This is because our global network is not geo-graphically balanced, i.e., certain regions may have more devices than others. Consequently, we may need to manually divide these regions to rebalance the network. Nevertheless, as our network expands, manually rebalancing the graph becomes increasingly challenging, even infeasible.

After booting up a canary, it's necessary to periodically update its configuration to align with the latest changes in our production network. Although more than 50% network changes occur on core devices (3), typically only a small fraction (less than 10%) of the network undergoes changes in canary. In the worst-case scenario, where the entire canary testbed needs updating, it takes an average of 208.4s to complete the process, which includes fetching, parsing, uploading, and reloading configuration in the emulated node. Given that we update the canary on a daily basis, the 3min update time is almost negligible for our platform in practice.



(a) Traffic drain.



(b) Traffic undrain.

Figure 10: Convergence time with different operations.

7.3 Network Convergence Time

In this experiment, we show that the emulated network can converge quickly after applying MOP commands to different levels of devices. We test the traffic drain operation, which is one of the most commonly used operations in network maintenance. We test the operation in the canary testbed across various levels of devices, including ToR, lower level aggregation (LA), higher level aggregation (HA), lower level spine (LS), and higher level spine (HS) devices. We use a BGP listener to collect all BGP update messages from all the emulated nodes in the canary testbed. The convergence

	Partitioning		
Task Runtime (s)	Random	Geo-manual	*Crescent*
route-diff	2.31±0.2	1.58±0.11	1.57±0.19
pingmesh	2±0.5	1±0.3	1±0.5
DPV loop	52±6		
DPV reach-diff	55±4		
config check	21±1.2		

Table 2: Evaluation of proactive task performance.

time is measured as the duration from the time an operation is issued to the time the last BGP update message is received.

Figure 10a shows the convergence time for traffic drain operations on different levels of devices. Surprisingly, the random partitioning scheme converges faster than the other two schemes. The reason is that the overall convergence time largely depends on the route update processing time instead of the route propagation time. Since both *Crescent* and geo-manual scheme tend to put the geographically close nodes on the same host, and most route updates happen within the same DCN, the host that runs the nodes of the DCN becomes the bottleneck for route update processing. Whereas, in the random scheme, the nodes that get affected by this operation are spread across all hosts, thus can process the route updates much faster with more resources. Figure 10b also shows the convergence time for undraining traffic after maintenance is done on the device. Overall, as depicted in both figures, the emulated network can converge very quickly, i.e., within 10s. Compared to other overhead, such as testbed connection time, the network convergence overhead is negligible.

7.4 Verification Time

We conducted four experiments, each with eight DUTs. After connecting the DUTs and network convergence, we execute multiple verification and monitoring tasks. Table 2 displays the average runtime for all experiments across all partitioning schemes. For DPV, we list the result of loop check and reachability difference (reach-diff) tasks. The reach-diff tasks compute end-to-end reachability differences for all network traffic classes. Since DPV tasks model and analyze O(100K) routes for O(1K) devices, they have a longer runtime than other tasks. The config checker has the next longest runtime. These centralized tasks fetch and send route/configs to a verification service and remain agnostic to the partitioning scheme. Route diff and pingmesh are lightweight, distributed tasks executed inside the containers, resulting in a faster runtime. Both the *Crescent* and geo-manual schemes exhibit similar runtime results for these tasks. And the random scheme shows a slight increase in runtime. Overall, it takes approximately a minute to verify the network for all tests.

8 Beyond Network Change

Crescent's primary mission was to prevent change-induced network incidents. Meanwhile, we have found *Crescent* helpful in many other use cases.

8.1 Catching Regression Under Failures

Some network changes may not trigger immediate impact until certain failures (e.g., link down) events occur. This has led to a few outages in the past. To expose such *deferred* impact from a network change, we run periodic workflows on *Crescent* to catch regressions in the network for various failure scenarios. We import the latest configurations from production networks and inject failure events such as a node or link down for critical parts of the network. Some significant errors we have identified so far include:

1. A border device missing a critical configuration that could lead to routing loops under specific link failures.
2. A global WAN device missing routes for specific regions due to misconfiguration. This could cause blackholes in case of another WAN device failure.
3. A newly-added configuration having incorrect community commands that could redirect a large volume of traffic to the management device, leading to congestion.

8.2 Self-Service Platform

Crescent provides rich user interfaces, including a web portal and chatbot, to assist users in creating emulation environments for any purpose.

Network Design Evaluation. When building a new network or evolving an existing one, our planners need to evaluate multiple design choices and conduct feasibility studies before coming up with the final MOPs. They leverage *Crescent* to create large testbeds, which would cost much higher if using physical devices. In one case, they found tens of bugs in their configuration templates.

Device Behavior Comparison. Cautious about VSB, our operators use *Crescent* to test the vendor software for non-standardized behaviors. One of these tests helped us discover VSB regarding BGP route aggregation. We observed that the inherited attributes for the aggregate route differed among the vendors. For example, some did not inherit the AS-PATH at all, while some inherited the common AS numbers.

Incident Reproduction. *Crescent* serves as a learning platform to reproduce past incidents, which is used to refresh our memory and educate new hires. Such training helps us avoid repeating similar mistakes from the past.

8.3 SDN Testing

SDN-based traffic engineering (TE) [36,39,45,54] has played an important role in our networks. A high-fidelity testbed is needed to test the correctness of the TE software, including the controllers and their dependent components. Small-scale physical testbeds are helpful but cannot scale to the same size as the production network. Besides, multiple TE projects may compete for limited physical testbeds, with high overhead to reset the environment during handover. Instead of waiting for days to use physical testbeds, it takes minutes for TE developers to create a production-like environment using *Crescent* for both component and integration tests of TE software.

9 Discussion

Crescent has been deployed in ByteDance since 2020. It has been serving tens of network change requests per week. This section shares our lessons learnt from its deployment and our thoughts for future work.

9.1 Lessons Learned

Road To Enforcement. *Crescent* has been integrated with our production network change workflow since its early days. But as shown in Figure 1b, it took almost two years from the initial rollout of *Crescent* to enforce it as a mandatory step for changing critical devices in production networks. Other than resource constraints, the major blockers were the lack of vendor images and the unfriendliness of user interfaces. We should partner with switch vendors earlier to develop and qualify the images as a collaborative effort. And we should involve our users more closely when designing user interfaces.
Unknown Image Limitations. Switch vendor images are blackboxes to us. Many of their limitations are unknown or undocumented. Some of these limitations may even manifest in a surprising way. For example, when we created a certain amount of ports in a container running a vendor image, its LLDP process started crashing while its layer 3 and above protocols, including BGP, worked fine. We did not notice the issue until the host ran out of memory. It turned out that the vendor used memory to store all the logs, and the huge-sized LLDP crash logs exhausted the host memory.
Fidelity Can Backfire. *Crescent*'s mission is to prevent network incidents. But as it is deployed in production hosts, failure to isolate from the production network could trigger unexpected consequences. For example, each emulated device connects to the host via a management or CPN interface through the docker bridge. Docker would add a default route automatically for such interfaces. In one case, we did not remove such a default route, and the emulated device sent syslog messages to the production collector, which could not distinguish it from the production device. This caused false alarms as the syslog reported many down links. It cost our network operators tremendous efforts to figure it out. We added the fix to remove the default route from each container and disable the syslog configuration during adaptation.
Kernel Tuning Can Help. In addition to employing multiple hosts, we explored increasing the number of containers for each host to support emulating larger networks. But, we failed to create a large number of containers while both CPU and memory usage were well below the host limits. After digging into the container logs, we spotted an error message and realized certain kernel settings for *inotify* should be relaxed.

9.2 Incidents Missed

The deployment of *Crescent* has significantly reduced the number of incidents, as demonstrated in Figure 1b. Emulation and verification using *Crescent* has helped detect and prevent various errors from occurring in production, e.g., configuration syntax errors and routing loops. However, there are still some network incidents that cannot be captured adequately with *Crescent*. For instance, we cannot identify issues related to traffic dependencies. One such incident was caused by hash polarization or imbalance, emerging from an unevenly balanced traffic load after being hashed twice or more. Additionally, high-level Quality of Service issues such as congestion, latency, and jitter can cause traffic-related complications. Emulation also lacks the potential to capture hardware-related issues. These include common problems like device failures, link flapping, etc, and rare issues like reduced Optical Transport Network (OTN) capacity leading to packet losses, etc. Moreover, in some cases, incidents may result from other components interacting with the network and its configuration, such as malfunctioning management software like a Netconf controller. While we have some regression tests with failures, these still need to be exhaustive, as certain failures could lead to never-emulated scenarios. Nevertheless, we constantly look for potential problems and enhance our emulation testing pipeline.

9.3 Future Work

Crescent does not attempt to search the tight safe boundary, as we think the correct boundary, if it exists, depends on certain assumptions of the routing policies. One idea to explore in future is to derive the routing policies from production network configurations, confirm them with network operators, and then add invariant checks for these policies. The derived routing policies can help guide the search of the boundary. Selecting the appropriate boundary is one method of reducing emulation resources without impacting fidelity. Another option is to investigate reducing the size of the emulation image by customizing it to include only relevant routing features.

Crescent presents network emulation as an alternative to CPV, but both can be used in combination to achieve high fidelity and performance. CPV can model specific aspects of the network, while emulation can emulate the remaining components to incur minimal performance overhead.

The baseline topology may evolve, e.g., adding more core devices to the production network. Today *Crescent* has to recreate the canary testbed with re-partitioning. We are exploring the possibility of updating the canary testbed dynamically with minimum overhead.

We plan to quantify *Crescent*'s fidelity by comparing the data plane of the emulated network with the one in production. The key challenge is that the data plane in production is more dynamic due to link flaps, controller programming, etc.

Finally, we anticipate that the number of emulation jobs will increase over time on multiple canaries. The scheduler plays a critical role in determining which job to schedule first. We are exploring using different scheduling strategies to handle incoming requests so that we can handle those requests not merely based on the request order, but also based on other factors, e.g., request emergency level.

10 Related Work

Network Emulation. Network emulation allows network operators to evaluate and design networking solutions. In today's landscape, commercial and open source network emulators, such as EVE-NG [4], GNS3 [8], DockerTopo [3], Mininet [40], Maxinet [53], vrnetlab [10], and SEED [23], are widely adopted by network operators for small scale network tests. Nevertheless, these emulators prove inadequate when it comes to large-scale network emulation involving thousands of nodes, primarily due to their lack of features such as dynamic lab formation (e.g., automatic node and link creation) and automatic configuration adaptation. CrystalNet [41] is the closest work to ours, which enables a large scale emulation for DC networks. Our work is inspired by CrystalNet, but with several enhancements on top. First, we find the algorithm to identify a safe boundary proposed in [41] is not safe for us. As a result, we choose to always include core devices in emulation. Second, we propose a greedy algorithm to address the graph partitioning problem when running a large-scale testbed using a multihost setup. Note that running a large-scale testbed with a better partitioning scheme can benefit any multi-host setup, no matter with VMs on a cloud infrastructure or with baremetal servers. In another word, the deployment strategy (where to run) addressed by CrystalNet is orthogonal to the partitioning scheme (how to run). Last, we made other efforts to enhance system automation for *Crescent*, e.g., combining emulation with automatic verification, config adaptation, etc. In summary, all of these emulators lack the features required for automatic timely verification through high-fidelity large-scale emulation, such as cross-vendor adaptation, dynamic link/node alterations for connecting DUT, etc.

Network Verification. Over the years, researchers have developed many DPVs and CPVs. DPVs usually analyze devices' forwarding rules and detect reachability policy violations. Recently researchers have added optimizations like making them incremental based on data plane updates [14,34,58], partitioning packets into equivalence classes to reduce the input search space [35,58], parallelizing verification by dividing global properties to local checks [37], etc. CPVs usually model the network control plane to detect similar reachability violations. Similarly, researchers have added optimizations like making them incremental based on configuration updates [57], modeling policies as graph properties [12], using Packet and Failure Equivalence Class [59] and abstract interpretation [15,16,30] to reduce the input search space, etc. Hoyan [55] is a verifier that aims to model vendor software behavior. It uses black-box testing to detect modeling deficiency but requires manual intervention to correct the model. We can integrate *Crescent* with any of these verifiers.

Configuration Management. Operators have made many attempts to automate configuration management. Network synthesizers use high-level policy intents to produce policy-compliant configurations. Some [17,24] create brand-new configurations from scratch. Whereas others [11,25,46] incrementally update the existing configurations. Some are very platform-specific, like Facebook's configuration management and syntax generation tool called Robotron [50]. NAssim [20] creates configuration models from device user manuals using NLP and deep-learning techniques. However, these are still limited in terms of feature coverage. *Crescent* can operate orthogonally to assist these tools' development for troubleshooting, debugging, etc.

11 Conclusion

Network changes are a major source of incidents. As a solution, we have developed *Crescent*, a large-scale high-fidelity emulation platform coupled with timely verification and monitoring tools. Instead of emulating the entire network, we exploit the symmetry and modularity of data center networks and build canary testbeds that emulate all core devices and sampled non-core devices. To achieve high scalability, we use a multi-host setup and propose a graph partitioning algorithm for a scalable node-to-host assignment to reduce the overhead due to cross-host links. We support dynamic link addition/removal to allow expansion/modification to the canary testbed on the fly. Our experience running *Crescent* has demonstrated its effectiveness in detecting problematic network changes, especially those resulting from undocumented vendor-specific behaviors.

Acknowledgements

We would like to thank many ByteDance colleagues who have contributed to this work. These include Jianghang Ning, Puyi Dang, Xiang Zhao, Yanlin Qiao, Feiyu Yue, Yuchen Shi, Hui Liu, Yifeng Zhou, Beixin Huang, Anjian Chen, Changqing Zhu, Henry Han, Weibin Wang, Amandeep Singh, Jianfei Cheng, Yanwen Chen, Jianwei Hu, Dong Zhao, Weibin Shen, Huang Zhang, Jintao Liu, Juntao Zhong, Gen Li, Bo Cao, Rohit Khairnar, Navneet Nagori, Wenhao Wang, Sisi Wen, Rui Du, Haotian Deng, Yiyang Chang, Yantong Meng, Heng Qiu, Yihan Zou, Yibo Zhu, Chuanxiong Guo, and many others. We would also like to thank Peng Zhang at XJTU, our reviewers, and our shepherd Guyue Liu, for their helpful comments.

References

[1] Intermediate System to Intermediate System (IS-IS) Extensions for Traffic Engineering (TE). https://datatracker.ietf.org/doc/html/rfc3784, June 2004.

[2] QEMU Documentation/Networking. https://wiki.qemu.org/Documentation/Networking, 2019. Last accessed on February 13, 2023.

[3] docker-topo. https://github.com/networkop/docker-topo, 2020. Last accessed on February 13, 2023.

[4] EVE-NG. https://www.eve-ng.net/, 2021. Last accessed on Sep 13, 2023.

[5] Hyperscale Data Center Count Grows to 659 – ByteDance Joins the Leading Group. https://www.srgresearch.com/articles/hyperscale-data-center-count-grows-to-659-bytedance-joins-the-leading-group, 2021. Last accessed on Sep 14, 2023.

[6] KVM guest virtual network configuration using MacVTap. https://www.ibm.com/docs/en/linux-on-systems?topic=configurations-kvm-guest-virtual-network-configuration-using-macvtap, 2021. Last accessed on Sep 13, 2023.

[7] Cloudflare Radar. https://radar.cloudflare.com, 2023. Last accessed on Sep 14, 2023.

[8] GNS3. https://www.gns3.com/, 2023. Last accessed on Sep 13, 2023.

[9] MaxiNet. https://maxinet.github.io/, 2023. Last accessed on Sep 4, 2023.

[10] vrnetlab. https://github.com/vrnetlab/vrnetlab, 2023. Last accessed on Sep 13, 2023.

[11] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 482–495, 2020.

[12] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, 2020.

[13] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running bgp in data centers at scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 65–81, 2021.

[14] Ryan Beckett and Aarti Gupta. Katra: Realtime verification for multilayer networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 617–634, 2022.

[15] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 476–489, 2018.

[16] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

[17] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, pages 328–341, 2016.

[18] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[19] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proceedings of the 2023 Conference of the ACM Special Interest Group on Data Communication*, pages 122–135, 2023.

[20] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. Software-defined network assimilation: bridging the last mile towards centralized network configuration management with nassim. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*, pages 281–297, 2022.

[21] Patrick Ciarlet Jr and Françoise Lamour. On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. *Numerical Algorithms*, 12(1):193–214, 1996.

[22] Marek Ciglan and Kjetil Nørvåg. Fast detection of size-constrained communities in large networks. In *International Conference on Web Information Systems Engineering*, pages 91–104. Springer, 2010.

[23] Wenliang Du, Honghao Zeng, and Kyungrok Won. Seed emulator: an internet emulator for research and education. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 101–107, 2022.

[24] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In *International Conference on Computer Aided Verification*, pages 261–281. Springer, 2017.

[25] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, 2018.

[26] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 05)*, pages 43–56, 2005.

[27] Andrew D Ferguson, Steve D Gribble, Chi-Yao Hong, Charles Edwin Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98, 2021.

[28] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture. Technical report, 2018.

[29] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, May 2015.

[30] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification*, pages 305–323. Springer, 2019.

[31] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[32] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, pages 58–72, 2016.

[33] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication*, pages 139–152, 2015.

[34] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y Richard Yang. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*, pages 314–335, 2022.

[35] Alex Horn, Ali Kheradmand, and Mukul Prasad. Deltanet: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, 2017.

[36] 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. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[37] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, et al. Validating datacenters at scale. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, pages 200–213. 2019.

[38] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 999–1013, 2020.

[39] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with blastshield. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 325–338, 2022.

[40] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[41] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613, 2017.

[42] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018.

[43] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. *ACM SIGPLAN Notices*, 51(1):69–83, 2016.

[44] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[45] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, pages 418–431, 2017.

[46] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 Conference of the ACM Special Interest Group on Data Communication*, pages 33–49, 2021.

[47] John Scudder, Rex Fernando, and Stephen Stuart. Bgp monitoring protocol (bmp). Technical report, 2016.

[48] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[49] Kotikalapudi Sriram, Doug Montgomery, D McPherson, Eric Osterweil, and Brian Dickson. Problem definition and classification of bgp route leaks. Technical report, 2016.

[50] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, pages 426–439, 2016.

[51] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the 2023 Conference of the ACM Special Interest Group on Data Communication*, pages 94–107, 2023.

[52] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, dec 2003.

[53] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pages 1–9. IEEE, 2014.

[54] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, pages 432–445, 2017.

[55] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng 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.

[56] Ken Yocum, Ethan Eade, Julius Degesys, David Becker, Jeff Chase, and Amin Vahdat. Toward scaling network emulation using topology partitioning. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, pages 242–245. IEEE, 2003.

[57] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential network analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 601–615, 2022.

[58] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. Apkeep: Realtime verification for real networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 241–255, 2020.

[59] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic router execution. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*, pages 336–349, 2022.

[60] Huaiyi Zhao, Xinyi Zhang, Yang Wang, Zulong Diao, Yanbiao Li, and Gaogang Xie. Improving the scalability of distributed network emulations: an algorithmic perspective. *IEEE Transactions on Network and Service Management*, 2023.

A Network Emulation

In this part, we describe the detials about how we emulate network devices, how we wire these emulated devices via various types of virtual interfaces (§A.2), and how we emulate edges (§A.3) and control plane network (§A.4) with *Crescent*.

A.1 Emulating Network Devices

Most of our switch vendors only provide VM images, while a couple also provide container images. Table 1 outlines all the vendors' images supported by *Crescent*, along with the least resources (e.g., CPU, memory) required to boot up. For vendors that support both VM and container images, we use container images as default because they are more lightweight than VM ones. However, if a device uses functions not supported by container images but supported by VM images, e.g., Segment Routing [28], we will emulate it with the VM image. Some vendors provide multiple VM images of different specs. For example, *v4* provides two VM images: one supporting more ports while consuming more resources. Thus before emulating a device of *v4*, we count the number of ports the device needs to emulate. If it is below the limit, we use the image with a lower resource requirement.

Native Vendor Image Support. *Crescent* uses containers as the basic units to mock up network devices [3,4,41]. *Crescent* treats VM images the same as container images by wrapping the VM image, a KVM hypervisor, and other libraries into a container image. We host a centralized image registry for hosts to pull the images from. By making this implementation choice, we are able to manage all emulated devices using a single underlying runtime system. Specifically, *Crescent* uses the Docker engine to manage all images and containers.

A virtual network interface attached to a container with a VM running inside are mapped to an interface of the VM via a MacVTap interface [6], which forwards ingress traffic to or egress traffic from the guest OS's corresponding virtual interface without traversing guest OS's kernel (e.g., container 1,2, and 3 in Figure 5). Inside the container, we use virtio [44] as the default network interface adapter attached to the guest VM to achieve a better performance. However, due to compatibility issue, some vendor's image only supports obsolete network adapters such as e1000 [2].

While there may be some additional overhead in wrapping a VM image into a container and routing traffic through virtual interfaces compared to running guest VMs directly on the host, this overhead is negligible in comparison to the time it takes for network convergence. For example, our evaluation using vendor *v3* shows that the additional overhead resulted in about 2×10^{-4}s of end-to-end ping latency, while network convergence could take several seconds.

A.2 Emulating Network Links

We use different virtual network interfaces to wire the emulated devices. We emulate a link on the same host by a veth pair directly connecting two virtual interfaces attached to two emulated devices. We must put a virtual network interface attached to a container running with a container image into the container's namespace before its containerized switch OS boots up [41]. Otherwise, the switch OS cannot recognize the virtual network interfaces. To deal with this issue, *Crescent* pauses a container immediately after the container starts running so that its network namespace is ready to attach virtual interfaces before starting the OS.

We setup a virtual link across two hosts with a VXLAN tunnel via the OVS bridge. One end of a veth pair is attached to the emulated device container network namespace, and the other end is attached to the OVS bridge. OVS is responsible for encapsulating and decapsulating packets across hosts. We always reserve the first virtual interface (i.e., eth0 in Figure 5) for the management port of the emulated device. The management port's virtual interface is then attached to the host's default Docker bridge, which is exposed for users to log into the emulated device via SSH or Web UI.

Figure 5 illustrates how we wire devices on the same host and across hosts. Containers running a guest switch VM inside (e.g., containers 1-3) use MacVTap to pipe traffic from the virtual interfaces attached to the VM to the virtual interface attached to the container or vice versa. For a container booted up from a container-based image, e.g., container 4, an end of the veth pair is attached to the container directly.

A.3 Emulating Edges

We consider three types of network elements at the edge: ISP, cloud, and servers under ToR, over which we have no, partial, and complete control in reality, respectively.

ISPs and clouds peer with our border devices via BGP. For the minimum requirements, *Crescent* must be able to inject the routes received from the edges into our network to ensure some properties, e.g., the traffic to certain IP prefixes must be routed through the gateway between our network and a public cloud rather than the gateway to an ISP's network. We have a partial control over the cloud peers (such as the ability to specify route policies), while no control over the ISP peers. As mentioned in [41], one way to replay the routes received from cloud peers is to inject these routes directly into our border devices using BGP speakers. However, the simplicity of this approach comes at a cost, as it can only be used to replay the routes announced from the edge to our network, but not vice versa. For example, if we want to observe the routes announced from our network to cloud peers (e.g., to avoid route leaks [49] from us), we would have to employ another component to collect the routes announced to the cloud peers, then analyze the collected routes later.

Crescent emulates these edge peers using a lightweight switch container image, as opposed to the proposed approach in [41]. *Crescent* adds loopback and static prefixes to an emulated device (which need not be identical to the real-world cloud peers) to establish BGP connections with our border devices and then configures these devices to announce the routes. By doing this, we are also emulating the cloud peer's behavior in addition to merely injecting routes to our border devices, which allows us to capture not only the changes in FIB of our border devices but also those of the emulated cloud peers with our proactive monitoring tools (§6). Besides, this approach allows us to emulate and validate route policies over which we have control on some public cloud peers in our emulation environment.

In a production DC, there are tens of servers under each ToR, where we have full control of their network stack. For ToR level devices, *Crescent* connects at least one virtual server to it to run proactive monitoring such as pingmesh (§6). These virtual servers are emulated using lightweight Linux images by default. We also support running switch images in these virtual servers, to help emulate the scenarios where advanced users establish BGP sessions between a server and the ToR.

A.4 Emulating Control Plane Network

Our networks are impacted not only by device configuration changes, but also by traffic engineering (TE) controllers that install BGP routes to override the default routing behaviors. It is critical to ensure the reliability of the software implemented for the TE controllers and their dependent components, e.g., BGP speakers for route announcement and BGP Monitoring Protocol (BMP [47]) collector for route collection. Instead of competing for limited physical testbeds, we utilize *Crescent* to provide a testing environment for these SDN software. One way is to containerize these SDN software and then let them establish connections to emulated devices through the emulated network. Nonetheless, this in-band approach requires TE developers to wrap their software into container images, which takes extra efforts if the software was not originally developed for a containerized deployment. Also it is not resource-efficient to emulate the in-band network while the controller manages only a small portion of the network. *Crescent* provides a more flexible option by using a dedicated Linux bridge to emulate Control Plane Network (CPN) [27]. CPN can be used to establish out-of-band connections between emulated devices and software components in a way that is transparent to the software under test. This method eliminates the need for the SDN software to be enclosed in containers, thereby reducing the amount of time required for our software developers to set up the test.

B Multi-Host Partitioning Algorithm

Algorithm 1 shows the pseudocode of the heuristic multi-host partitioning algorithm proposed in §4.3.

C Topology Expansion

Algorithm 2 presents our topology expansion algorithm. This algorithm primarily takes the devices under change and *expansion_width* (w) as input and returns the emulated nodes as output. It uses a variable N to track all traversed nodes and map them to their traversed neighbors.

The algorithm first expands the topology upwards (line 5) and downwards (line 6) to add upstream and downstream neighbors. This expansion uses a modified depth-first search (DFS) algorithm called *MV_DFS* (line 11). *MV_DFS* stands for multi-vendor DFS. It is a directional DFS to expand the network by w and add all required vendors per level. For each node, *MV_DFS* does not explore all the node's neighbors. It terminates when it has added at least w neighbors and ensures they represent all the unique vendors attached to this node (line 28). The algorithm customizes the for-loop traversal in line 21 in two ways. It prioritizes nodes already selected for emulation. This strategy ensures we reduce the number of expanded nodes. And it also prioritizes nodes belonging to vendors that it has yet to explore in that loop. This strategy aims to satisfy the condition in line 28 as soon as possible. During *MV_DFS*, it adds nodes (line 22) representing either the leaf or top node or those representing nodes with newly explored neighbors (line 26).

Next, the algorithm expands the network horizontally (line 7). It ensures that for each device under change, we add other nodes from all vendors that belong to the same level (lines 33-34). Again, the algorithm customizes this to prioritize nodes already selected for emulation.

Algorithm 1: Randomized Multi-Host Partitioning

```
Preprocess: Generate an undirected weighted graph $(V,E)$. For
   every partition $p$, we have a corresponding node $p'$ in $V$. And for
   every pair of partitions $p_1, p_2$, we have $e = edge(p'_1, p'_2)$ in $E$ and
   $e.weight$ = # of links between $p_1$ and $p_2$. We also define the cost
   of a node $p'$: $p'.cost$ = sum(memory_cost of each node in $p$).
Input: $V, E, partition\_memory\_limit$
Output: map($v \rightarrow n$) for each $v$ in $V$ // map each node to a
   partition number

$uf \leftarrow$ union_find($V$); // assign each $v$ in $V$ to a new set
$non\_zero\_edges \leftarrow$ [$e$ for $e$ in $E$ if $e.weight \neq 0$] ;
$zero\_edges \leftarrow$ [$e$ for $e$ in $E$ if $e.weight = 0$];
// Instead of always choosing the edge with the
   largest weight,
// we sample(without replacement) from a distribution
   based on the edge weight.
$non\_zero\_edge\_distribution \leftarrow$ distribution over $non\_zero\_edges$,
   $p(e_i) = e_i.weight/sum$;
$zero\_edge\_distribution \leftarrow$ distribution over $zero\_edges$,
   $p(e_i) = \sqrt{e_i.src.cost \times e_i.dst.cost}/sum$;

while $non\_zero\_edge\_distribution$ is not empty or
   $zero\_edge\_distribution$ is not empty do
    // Merge the edges with non-zero weight first. If
       all remaining edges has zero weight,
    // use the geometric mean of $src.cost$ and $dst.cost$
       as weight.
    // $sample\_without\_replacement$: with replacement, a
       value can be selected multiple times.
    // For each edge we only visit it once, so we use
       $without\_replacement$ here.
    if $non\_zero\_edge\_distribution$ is not empty then
        $e \leftarrow$
          $non\_zero\_edge\_distribution$.sample_without_replacement();
    else
        $e \leftarrow$
          $zero\_edge\_distribution$.sample_without_replacement();
    $src\_root \leftarrow uf$.get_root($e.src$);
    $dst\_root \leftarrow uf$.get_root($e.dst$);
    if $src\_root = dst\_root$ then
        continue;
    if $src\_root.cost + dst\_root.cost > partition\_memory\_limit$
      then
        continue;
    $new\_root \leftarrow uf$.union($e.src$, $e.dst$);
    $new\_root.cost \leftarrow src\_root.cost + dst\_root.cost$

$m \leftarrow$map();
for $v$ in $V$ do
    $m[v] \leftarrow$ indexof($uf$.get_root($v$));
return $m$
```

Algorithm 2: Topology Expansion

```
Global inputs: $S$: devices under change, $w$: expansion width;
Global output: $T$: emulated devices, initialized to $S$;
Global variables: $N$: map node to a set of its neighbors;
procedure TOPO_Expansion()
    EXPAND_VERTICAL(up);
    EXPAND_VERTICAL(down);
    EXPAND_HORIZONTAL();
Input: $dir$: direction (up or down);
procedure EXPAND_VERTICAL(dir)
    for $s \in S$ do
        MV_DFS(s, $dir$);
        if $N$ has changed then
            for $n$ in new nodes in $N$ do
                $T$.add($n$);
Input: $s$: visiting node;
procedure MV_DFS(s, dir)
    if $s \in N$ then
        return
    $newN \leftarrow \{\}$;
    $D \leftarrow s$.neighbors($dir$);
    for $n \in D$ do
        // The for loop will prioritize nodes already
           in $T$ and belonging to untraversed vendors.
        if $n$ is top or leaf node then
            $newN$.add($n$);
        else
            MV_DFS($n$, $dir$);
            if $N$ has changed then
                $newN$.add($n$);
        if $size(newN) \geq w$ and $vendors(newN) = vendors(D)$
          then
            break;
    $N[s] \leftarrow newN$;
procedure EXPAND_HORIZONTAL()
    for $s \in S$ do
        // Prefer selecting $n$ already in $T - D$
        if $L(s) \neq \emptyset$ s.t. $L$ represents other nodes at same level then
            $\forall v \in vendors, \exists n \in L(s,v)$ s.t. $T$.add($n$);
```