Crinkle: Tenant-Driven Debugging on Cloud Testbeds

Alexander Wolosewicz
awolosewicz@hawk.illinoistech.edu
Illinois Tech
Chicago, USA

Raffay Atiq
raffay.atiq@sri.com
SRI
Menlo Park, USA

Nishanth Shyamkumar
nshyamkumar@illinoistech.edu
Illinois Tech
Chicago, USA

Van Anderson
vanderson@hawk.illinoistech.edu
Illinois Tech
Chicago, USA

Vinod Yegneswaran
vinod.yegneswaran@sri.com
SRI
Menlo Park, USA

Ashish Gehani
ashish.gehani@sri.com
SRI
Menlo Park, USA

Nik Sultana
nsultana1@iit.edu
Illinois Tech
Chicago, USA

Abstract

Today's research testbeds are organized as multi-tenant, virtualized clouds with researchers as tenants. While the typical use of logically isolated resources provides tenants with flexibility for their experiments, isolation abstractions complicate the diagnosis of network issues. Ultimately, debugging might require researchers to obtain support from testbed operators—a scarce resource.

This paper introduces Crinkle, a network debugging approach for such virtual environments that does not require operator support or underlay modifications. Crinkle leverages two ideas: (1) controlled placement of nodes for debugging, and (2) *partial packet provenance* (3P), which we introduce to reason about observations. 3P provides an expressive foundation for a debugging API that tenants use to obtain insights into network behavior. A Crinkle prototype was developed using DPDK and evaluated on the state-of-the-art FABRIC and SPHERE testbeds. The prototype is evaluated through performance benchmarks and case studies to assess the system's usability and effectiveness. The benchmarks demonstrate that the prototype's monitors operate at up to 55 Gbps and 15.6 Mpps.

CCS Concepts

• **Networks** → **Network monitoring**; **Cloud computing**; • **Applied computing** → *Evidence collection, storage and analysis.*

Keywords

Networking, Virtual Networks, Network Observability, Debugging

ACM Reference Format:
Alexander Wolosewicz, Raffay Atiq, Nishanth Shyamkumar, Van Anderson, Vinod Yegneswaran, Ashish Gehani, and Nik Sultana. 2026. Crinkle: Tenant-Driven Debugging on Cloud Testbeds. In *ACM Symposium on Cloud Computing V.2 (SoCC '26), November 18–20, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3828161.3857892

This work is licensed under a Creative Commons Attribution 4.0 International License.
SoCC '26, Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2816-7/2026/11
https://doi.org/10.1145/3828161.3857892

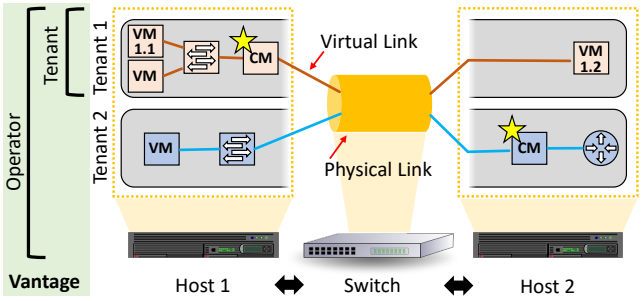

Figure 1: Crinkle mitigates the restricted vantage that tenants have when debugging their network. Crinkle works by controlling the behavior of opportunistic, ephemeral virtual resources for network debugging—here, VMs labeled as "CM" (for Crinkle Monitor).

1 Introduction

Today's state-of-the-art shared testbeds use network virtualization to accommodate the need of simultaneous researchers to flexibly define their own topology across the testbed's resources. Such testbed networks can be seen as multi-tenant cloud networks (with researchers as tenants) that support the creation of logically isolated overlays with allocations of resources to different tenants [51].

Multi-tenant research testbeds have a strong need for observability and debugging capabilities, yet current testbeds provide little to no dedicated support for debugging. A survey we conducted among FABRIC users found that researchers primarily rely on low-level diagnostic tools—such as `ping`, `traceroute`, and packet captures—to troubleshoot their experiments [77]. While these tools are useful for basic connectivity testing, they provide limited support for diagnosing complex problems that arise in multi-tenant networks.

Unlike production cloud tenants, researchers often develop prototypes that evolve over months, requiring repeated debugging as the systems mature. In addition, research testbeds are explicitly designed to support novel systems and experiments that push infrastructure beyond conventional operating conditions. As a result, researchers must often determine whether a discrepancy between *observed* and *expected* behavior stems from a bug in their prototype

or experiment, or from factors external to the experiment. For example, prior work has shown that Linux kernels lacking specific modules can incorrectly report packet drops [65], while another study identified a communication bottleneck in the testbed architecture that degraded storage performance [64]. Distinguishing between such underlay-induced effects and genuine experiment bugs is thus a recurring challenge for researchers on these testbeds.

Network problems in such testbeds can be rooted in individual tenants' virtual networks or in the *underlay* that is shared by tenants. As we know from multi-tenant cloud networks, such problems include transient congestion conditions and microbursts [11], persistent problems such as router or service misconfiguration [37, 38, 44], and dynamic service chain compositions that apply incorrect filtering, tunneling headers, or isolation, and obscure traffic provenance [30]. Regardless of the type of problem, diagnosing and debugging network behavior on such networks requires access to *vantage points* in the network to localize and observe behaviors of interest. A testbed network operator has privileged access to the network, and thus can access any vantage point—including both virtualized and direct access to hosts and switches (Figure 1). In contrast, a tenant is limited to vantage points among the virtualized resources they control—i.e., their "slice". Even with the operator's vantage, it takes time and painstaking effort to debug "gray failures" [43] that present partial and inconsistent error indications.

To underscore the similar designs of clouds and of testbeds like FABRIC, note that FABRIC relies on common, cloud-oriented building blocks such as OpenStack for VM management [27], standards-based network virtualization [26], and separate management and data networks [61]. As in commercial clouds, FABRIC tenants only control their overlay, which includes virtual machines and hardware attached to them like smart NICs, FPGAs, and GPUs. The operator controls the underlay hardware: the servers running the virtual machines, the switches connecting them, and the links between components and sites. Tenant resources are connected by virtualized network services over the underlay, and they may use overlay resources that serve as network elements within their virtual topology like routers, load balancers, or firewalls, as is possible in commercial clouds [63]. This cloud-inspired design is versatile—thousands of researchers and students have used FABRIC with many different workloads and configurations, for research on novel network architectures [14, 74], programmable networking [7, 10, 33, 49], teaching [75], distributed systems [57, 62], network management [48], distributed storage [55], network monitoring [65, 71], middleboxes [56], and reproducing research [13, 21, 73].

Given their similar designs, can we borrow debugging techniques from commercial cloud networks and adapt them for cloud testbed networks? The answer is no: we found that there is a lack of *tenant-driven* debugging techniques on cloud networks, and therefore this necessitates research into designing such techniques for testbed networks. As future work, these techniques could be ported over to commercial cloud networks, but the evaluation in this paper is scoped on research testbeds and their users.

Today's debugging and diagnosis techniques assume operator-level network access, and we lack virtualization-aware techniques for *tenants*. That is, while tenants are operators in their own virtual networks, network problems might be rooted in underlay resources, as described earlier. A virtualization-aware, tenant-usable network debugger that does not require operator-level visibility (such as knowledge of internal topologies or network controllers) would empower tenants with capabilities that conserve operator effort. **Section 2** surveys related work.

We envision an architecture where tenants in testbeds **(1)** are provided with dedicated, specialized tooling for network diagnosis and debugging; **(2)** this tooling is simple to activate and it provides convenient, programmable observability across the tenant's experiment network; **(3)** use of this tooling incurs a predictable impact on latency and throughput. This tooling is designed to give researchers more autonomy in debugging their virtual networks, and can independently gather information that helps them decide whether to escalate a network problem to the operator and to vector the operator to the cause of their problem based on collected information. This vision and research problem are analyzed in **Section 3**.

Towards realizing that vision, this paper introduces Crinkle, a network debugging approach that provides testbed tenants with resources that have restricted capabilities (specialized for debugging) and are placed in the network according to tenants' needs. On those resources, Crinkle provides an interface for flexible yet controlled diagnosis and debugging tasks. Crinkle is hardware agnostic and works with non-programmable network equipment. It is designed conservatively to work on existing systems—we validated Crinkle on two virtualized networks that we do not operate.

Intuition. Figure 1 shows how tenants can place Crinkle *monitors* (configurable observation VMs) between existing virtual resources that might be distributed across physical infrastructure. Tenants would not have direct access to the underlay, but observations from the monitors provide tenants with information about their traffic's handling. Thus Tenant 1 can determine whether traffic traveling from VM 1.1 to VM 1.2 is being affected by the underlay, and the operator could then consider if the problem lies with the physical switch. Similarly, Tenant 2 can determine whether their virtual router is misconfigured or affected by the underlay.

Example. Figure 2 demonstrates a real research topology used on a cloud testbed. When running an evaluation workload, a failure event occurs that causes traffic to be rerouted. Crinkle provides the researcher with the ability to quickly observe the path traffic was taking before and after the failure, and through this localize the failing node. They can retrieve the histories of packets that traversed the network before and after the failure with a single API call (Section 4.2). Further case studies showing exactly how Crinkle assists with debugging are shown in Section 7.2.

Foundation. To balance the operator's needs for resource management and the tenants' needs for expressive debugging, Crinkle introduces the notion of *Partial Packet Provenance* (3P) to enable rigorous and expressive network debugging. 3P builds on earlier work on packet provenance [16, 84] and related techniques [59] with an ability to select links to provide packet information, filtering according to packet properties and stateful network conditions. Crinkle, through 3P, also leverages the ability to specify VM placement with host-level granularity in current testbeds, like FABRIC [8] and SPHERE [46], to reason about underlay issues.

Debugger. Building on 3P, Crinkle provides a high-level API that exposes detailed insights into network behavior. The implementation of this API is designed to provide tenants with control over network debugging. This approach exceeds the typical link-level or

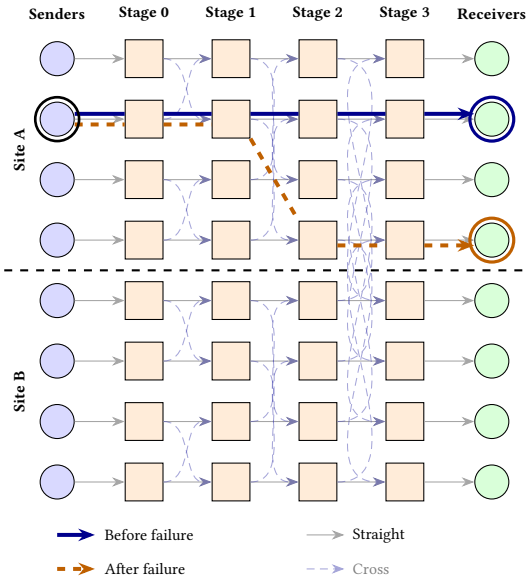

Figure 2: A butterfly topology [40] used for research on FABRIC—the largest-known single-experiment topology on the testbed. Crinkle is used to easily verify the flow of traffic before and after a failure of a switch, and identify that the stage 2 switch is the failure.

element-level scope of today's network debugging techniques, and it can be used to debug both correctness and performance. Crinkle can also collect measurements over time to observe impacts from background traffic from other tenants [72].

Prototype and Evaluation. We build an open-source, functional Crinkle prototype that is contributed to the public API for FABRIC [8], an international, state-of-the-art, network-oriented [29] research testbed that uses industry-grade infrastructure for cloud environments [18, 53]. Crinkle is evaluated on FABRIC through performance benchmarks and a user study. The Crinkle prototype uses DPDK and operates at up to 55 Gbps and 15.6 Mpps, with a median latency inflation of 8.21 µs per monitor. We further verified the functionality of Crinkle on the SPHERE testbed [46].

In summary, this paper makes the following **contributions**:

- **Section 4:** The design of Crinkle which enables *tenant-driven* network debugging by providing restricted resources for tenants to carry out network diagnosis and debugging in virtualized, multi-tenant testbed networks.
- **Section 5:** The Partial Packet Provenance (3P) specification provides our foundation for reasoning about network debugging and placement. 3P is used to characterize frequently occurring network faults. By weakening the requirement to observe *all* packets across *all* links, 3P generalizes earlier provenance techniques that were used in networking, and allows control over vantage placement.
- **Section 6:** The Crinkle prototype, which has been made available in FABRIC's public API. Crinkle-related FABRIC artifacts have been downloaded over 1,100 times. This paper describes the requirements for the network to support this prototype, for porting to other testbeds and networks.
- **Section 7:** The prototype is evaluated on the FABRIC testbed, a virtualized, multi-tenant network. The evaluation targets the performance, overhead, and applicability of Crinkle.

2 Related Work

Network debugging is a mature research area but relatively little research attention has been devoted to developing techniques that are *virtualization aware*, and even less into *tenant-driven* techniques for multi-tenant, virtualized networks.

This section discusses how Crinkle is related to other research on virtualization-aware network debugging techniques (Section 2.1). Since the source of network problems might be in a tenant's virtualized network, and not in the underlay, we discuss how Crinkle compares with broader research on network debugging (Section 2.2). Most existing techniques are intended for network *operators* because those techniques require complete access to the network and hardware. We further compare our underlying formalization, Partial Packet Provenance, to existing provenance work (Section 2.3).

2.1 Virtualization-aware Debugging

Virtualization-aware debugging uses knowledge of the network overlay and underlay to localize a network problem and understand its causes. To the best of our knowledge, VND [78] and NetHint [17] are the only research on *tenant-facing*, virtualization-aware network debugging. Virtualization platforms run by hyperscalers implement debugging interfaces that provide diagnostics, counters, tcpdump-based packet capture, and latency measures, but these are only queryable by the network operator [3, 19, 23, 31].

VND describes an extension to a virtualized network architecture to provide a shared service for network tenants to observe and analyze network behavior in their overlays. VND is embedded in the underlay, and is able to see all tenants' traffic. Tenants' queries are expressed in an SQL-like language, and the system filters away underlay information when presenting answers to tenants. In comparison, Crinkle is embedded in each tenant's overlay—this avoids the problem of potentially leaking information about the underlay or other tenants. Crinkle augments the tenant's network with bump-in-the-wire measurement nodes that are themselves VMs allocated to the tenant. The tenant is able to bypass the Crinkle API and directly run SQL queries on the database.

NetHint [17] exposes virtual topology and underlay link capacities to tenants, enabling them to diagnose performance problems that stem from underlay congestion. NetHint is mostly intended to optimize the orchestration of cloud resources for big data processing by tenants. Meanwhile, Crinkle measures runtime behavior using controlled placement of monitor nodes and packet histories [41].

2.2 Debugging Tenant-level Network Problems

We now discuss how Crinkle compares with related work for network problems rooted in the tenant's virtual network—not in the underlay. Network debugging is a research area that spans techniques that use active probing [34, 86], passive observation [6, 59, 67, 68, 80], timing and replay [34, 42], runtime checking [58], packet histories [4, 36, 41], and leverage programmable hardware [16, 35, 82, 85]. Crinkle combines all but programmable hardware to localize faults

and measure latency, and makes this functionality available to tenants. In contrast with the work cited earlier, Crinkle supports *placement* of restricted monitors to help tenants determine whether the underlay is contributing to the symptoms that tenants are observing. To avoid constraining tenant network workloads or target deployment environments, Crinkle does not use in-band telemetry [79] and does not require programmable network switches. Crinkle gathers similar metadata to that of PathDump [67] across the specialized monitoring VMs that are distributed in the tenant's network, but does not require changes to the tenant's VMs.

2.3 Provenance

Finally, we compare our Partial Packet Provenance (3P) to existing work. *Provenance* is concerned with explaining how something occurs using a history of causal relations, often through directed graphs. The most related bodies of work are from the declarative networking and distributed host-based provenance communities.

Secure Packet Provenance (SPP) [16] hails from the declarative networking community. It assumes that a set of networking events (rules being added or removed, links going up or down, and sending or receiving packets) are represented by graph vertices; causal dependencies between events are modeled as edges. While no host in the system is assumed to have the complete graph, all relevant events must be observed at some host. In contrast, 3P can operate with incomplete reporting—that is, as a packet traverses a path, an intermediate link may not have a monitor. Because the packet identifiers observed at all traversed monitors will be identical, 3P can trace the path back from the downstream monitor to the upstream one despite the absence of observations at intermediate points. (In the case of SPP, the absence of reporting at an intermediate point would break the causal tracing.) From a systems view, Crinkle does not have strong security guarantees like SPP, but these are not required for the network problems targeted by Crinkle (Section 3.3) and would add hardware constraints and runtime overhead (to support the cryptographic primitives used by SPP at line rate).

Distributed host provenance is exemplified in SPADE's [32] Audit-based Open Provenance Model (OPM) [47] representation and DARPA Transparent Computing's (TC's) [20] Common Data Model (CDM) [39]. These report network activity at the coarser granularity of flows, rather than packets. Graph vertices denote network artifacts (in SPADE's OPM) or netflow objects (in TC's CDM) and system processes. Edges denote data exchanged between processes and network flows. (Intra-host provenance also covers other relations, such as those that arise from inter-process communication and process interaction with the filesystem.) As with SPP, OPM/CDM graphs of distributed provenance require every flow edge in a path to be reported for the provenance of downstream network data to be tracked to an upstream source. The absence of a single OPM / CDM edge will create a false negative response in a corresponding distributed provenance query. As explained in the SPP description above, 3P is robust to this.

3 Tenant-driven Network Debugging

In our setting, as in commercial clouds, tenants have full control of their overlay resources and no control over or visibility into the underlay beyond what the operator can optionally provide for leasing resources, such as proximal placement (Section 4.1). Figures 1 and 3 illustrate the difference between tenant and operator visibility. An unfortunate consequence of network virtualization is that testbed tenants are exposed to the union of problems that are rooted in (1) individual tenants' (logically isolated) overlays, and (2) their common underlay. Symptoms of network problems involve unexpected observations related to network traffic. For example, packets might fail to reach their destination [87] or might be delayed [5, 59] and violate Service Level Objectives (SLOs) [15]. More mysteriously, some types of packets may arrive but not others [34], and packets may be forwarded along unexpected paths [70].

3.1 Research Problem

How can tenants of a shared testbed effectively diagnose and debug network problems without operator support, while preserving virtualization abstractions and privilege restrictions?

While tenants are empowered to diagnose and debug network problems within their virtual networks—where tenants appear as network operators—tenants today are unable to determine whether the network problems that they perceive are rooted in the underlay, nor can they localize or mitigate those problems.

3.2 Network Debugging by Tenants

Tenant-driven Network Debugging seeks to empower the tenant with more autonomy to diagnose and debug network problems that are rooted in the underlay or in the tenant's virtual network. Just as with operator-driven network debugging, it involves asking questions and making observations that lead to understanding network problems and enabling their localization [66]. For example: "Which [virtualized] paths does the traffic follow? Is the network-wide latency acceptable? Is traffic load-balanced as expected?" [70] "Did a drop happen at this link? Is application RPC latency unusually high at this [virtual] host?" [59] Moreover, is the observed behavior transient or persistent? Does it occur across the virtual network, or around specific VMs?

While tenants may never have direct visibility into underlying causes, tenants do not need full root-cause information to take meaningful action. For example, if tenants can at least *localize* the problem within their topology and *characterize* the problem (packet loss, delays, incorrect forwarding, etc) then they can narrow down the potential problem sources in their overlay—such as misconfigurations. If no problems can be found with the overlay, then tenants can escalate a more detailed error report to the network operator.

3.3 Example Problems for Debugging

By reviewing the literature, we derive a set of network debugging problem classes. These problems can be encountered in non-virtual networks, but in virtualized networks these problems are exacerbated by the virtual-to-physical network address mappings between resources [81], multi-path underlay topologies, and the layered, load-balanced design of virtual networks [45, 80].

Pr1: *Packet loss and reachability*—addressed in 007 [6], Pingmesh [34], and NetBouncer [69], which are operator-specific techniques that require underlay switch access or using specific protocols like ICMP.

Pr2: *Performance anomalies*—addressed in NetHint [17] (discussed in Section 2.1), SwitchPointer [68], DETER [42], and Pingmesh [34],

which rely on protocol-specific or switch-provided metadata that is not accessible by tenants.

Pr3: *Correct forwarding, loop detection, and path conformance*—related work includes Stroboscope [70], Trumpet [50], and PathDump [67], which rely on access to underlay switches or end-point extensions (to tenant VMs).

Pr4: *Opaque and complex causes*—the difficulty of carrying out root-cause analysis has motivated the use of *provenance* information to gather rich packet history that is used to track packets across network elements that might mutate packets. Motivation and network operator-oriented solutions to this problem are provided by dShark [80], SPP [16], and DTaP [84]. dShark distributes the effort of gathering and analyzing packets by using a setup similar to MapReduce [22] that is invoked by the network operator, and relies on port mirroring of raw traffic at switches.

3.4 Tenant-driven Debugging Requirements

Enabling tenant-driven network debugging is challenging because it should not disrupt the multi-tenant network. This section decomposes the problem into a set of requirements.

R1: *Security and Privacy.* A tenant's debugging capability must not put at risk the security and privacy of other stakeholders—including that of the operator and other tenants.

R2: *Modification-free.* The debugger should be usable on tenants' production traffic, and should *not* require tenants to modify their workloads, topologies, VM images, or software.

R3: *Scaling.* The debugging system must scale to (1) large networks of individual tenants and (2) simultaneous and independent debugging being carried out by different tenants.

R4: *Flexibility.* The debugging system must help localize the range of different problems described in Section 3.3.

R5: *Transparent Measurement.* Since the background conditions of the underlay change over time, the performance and jitter of virtualized networks can vary. To understand this variation and determine whether it is contributing to problems, the debugger must report measurements that characterize the state of the network.

4 Design of Crinkle

Crinkle provides a debugging system that enables tenants to diagnose and debug network problems in their virtualized networks without requiring operator support. It uses specialized virtual machines that uniquely tag frames and relay observations to a database that stores per-frame histories by these unique tags.

Figure 3 provides a high-level view of how Crinkle works. Tenant flows start out as shown in the top (hatched) path—traveling from VM 1.1 to VM 1.2, and where the tenant only has visibility into stages Ⓐ and Ⓔ. The placement of Crinkle monitor VMs allows the tenant to make observations of that traffic flow at different points, as shown in the bottom (solid) path. These points occur at Crinkle nodes at Ⓑ (that is placed on the same server as Ⓐ), Ⓒ (on a different server from Ⓑ but in the same rack), and Ⓓ (on the same server as Ⓔ). These are bump-in-the-wire nodes that bisect virtual links. These links might exist within a single host or span several links between the hosts containing the VMs.

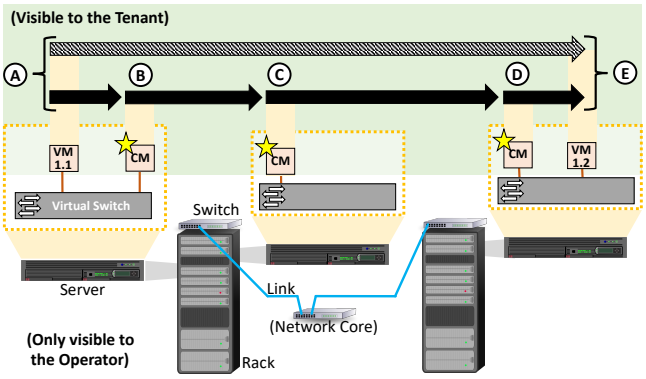

Figure 3: Crinkle nodes *decompose virtual links* to obtain granular observations about packet propagation. The lower part of the figure shows infrastructure that is only visible to the testbed operator, but whose configuration and behavior affects the testbed's users.

4.1 Network Architecture

Crinkle is designed to function on existing network architectures, without relying on programmable hardware or operator support. There are two assumptions that Crinkle relies on—these are normally provided by cloud networks: The first is that the network preserves Crinkle's **frame tagging** scheme (Section 6.1), which enables precise and durable provenance information to be tracked. Our evaluation shows that this is preserved in many cases. The second is that there is adequate **time synchronization** between a tenant's VMs; this ensures that the tenant's latency measurements are accurate. To assist tenants with localizing problems and explore the full capabilities of Crinkle, we work with the assumption that tenants can specify **proximal placement** of their resources. With it, they can place overlay resources onto hardware in relation to other overlay elements; for example, one VM not co-located with another VM. Without this feature, or if the operator cannot honor it due to load, Crinkle can still debug the overlay network. If the tenant verifies their overlay is functioning correctly, then by elimination they can reason and justify to the operator that the issue is likely underlay-induced. The lost capability is that they cannot reliably localize faults within the underlay topology.

4.2 Debugging using Crinkle's API

Table 1 shows the primary interface for Crinkle. With this API, tenants add Crinkle monitors to their topology, query them, and replay traffic to carry out additional observations. The API translates filters (on pcap expressions, timestamps, and frame IDs), formalized in Section 5, to the specific commands to build *provenance graphs*.

The resulting graphs conform to the Open Provenance Model (OPM) [47], which was designed to capture provenance in a domain-agnostic structure. The OPM specification explains how cycles can arise. These can be eliminated by versioning vertices (based on edge temporal annotations) to "unroll" the cycles. The result is a directed acyclic graph. Figure 4 uses OPM notation and shows a packet bouncing between two routers, indicating unintended forwarding—likely because of a misconfiguration. In that figure, the blue rectangle vertices represent the network interfaces of the

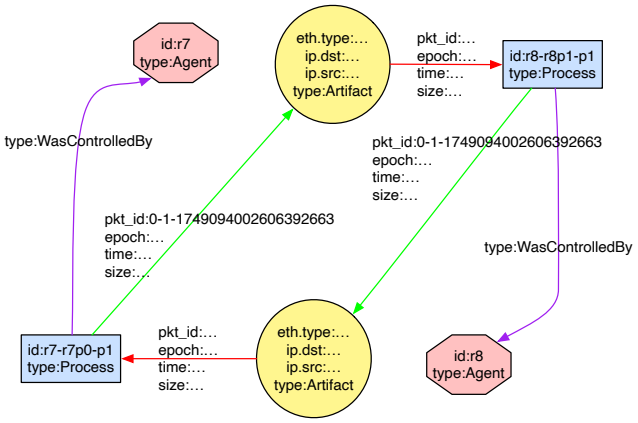

Figure 4: Crinkle-produced provenance graph with most values elided to reduce clutter. Read from top-left to bottom-right, the graph indicates that r7 is forwarding packets from r8 back towards r8. This example is referenced and explained in Section 4.2.

nodes, which are represented by red octagons. Following provenance graph convention, the edges point backwards in time. The figure can be read as packets arriving at r8's interface originate from r7, and they are the same packets that arrived at r7 from r8 since they have the same packet IDs. Using this information, the tenant can infer a forwarding problem.

4.3 Design Goals

This section describes how Crinkle satisfies the design requirements from Section 3.4. Tenants are unable to violate the security or privacy of the underlay or other tenants (**R1**) since the visibility of tenants and the added trailers remain contained within the virtualization abstractions of the testbed. To the tenant's workloads, the topology is unchanged (**R2**) but the links monitored by Crinkle allow tenants to observe their network's data plane and replay traffic selectively, to perform surgical tests without running the entire workload. Instead of relying on underlay switches or end-point extensions, Crinkle adds bump-in-the-wire VMs to monitor the network and analyze the (repeated) occurrence of packets.

The placement of Crinkle nodes takes place without modifying the applications that are running on Ⓐ and Ⓔ in Figure 3. Crinkle uses the operator's placement API and does not require special permissions or support to operate. Crinkle does not grant tenants additional access to the operator's or other tenants' resources. It is hardware-agnostic as it can treat every node in the network as a black box. Crinkle is designed to work for arbitrary tenant traffic—it is not restricted to working with ICMP or TCP traffic. Crinkle does not rely on switch-provided metadata; instead, it uses (tenant-level, virtual) links as identifiers for traffic locations.

The scalability of Crinkle reduces to that of the virtualized network (**R3**) since it uses existing resource primitives that the operator provides to tenants. Additionally, Crinkle provides an API to place those monitoring nodes relative to the tenant's existing VMs, to provide them with a range of complementary vantage points. The API is also used to control the behavior of monitoring nodes, to capture, filter, and replay traffic, thus supporting a range of different usage scenarios (**R4**) described in Section 4.2, and which will be detailed further in Section 5.

Crinkle provides high-precision, portable, record-and-replay functions to determine delays and jitter that take place along paths in the network (**R5**). Such delays and jitter might be caused by problems in the virtualized resources or in the underlay. By using Crinkle, the tenant can obtain precise measurements that can help them fix a problem in their (overlay) network, or help vector the operator to a problem in the underlay.

4.4 Filtering Network Provenance

Use of packet-level provenance tracking can yield extremely large and dense graphs. Each captured packet is mapped to a provenance vertex, and its metadata is linked through multiple edges to hosts, processes, and other artifacts. For long-running workloads or high-throughput links, the number of vertices and edges can grow rapidly, making downstream analysis expensive. The issue is compounded by the presence of routine protocol-level traffic that contributes limited semantic value in the context of experiment-specific communication patterns or application behavior.

To address the challenge of graph bloat, we use a combination of filtering strategies. Filtering is available in two phases: **(1) In-network filtering:** Applied within the packet capture pipeline, this strategy allows real-time discarding or abstraction of traffic before it is logged. **(2) Post-capture transformation:** Applied within provenance graph construction, this method performs structural or semantic filtering on logged events before final graph building.

4.5 Limitations

Crinkle focuses on tracking frames and has no visibility into control-plane changes—such as updates to flow tables. These limitations are out of scope for this project—overcoming them to provide a more integrated debugger is left for future work. For instance, Crinkle nodes could parse routing protocol messages to correlate them with updates to flow tables. More generally, Crinkle could interpose controller messages to track how the data plane responds to configuration changes. Finally, Crinkle has no visibility into VM or host-level faults—such as memory depletion or running out of storage space—which can pre-empt or explain network problems. However, Crinkle can be used to localize the fault to an unresponsive VM. Issues occurring due to Crinkle nodes can be localized as an underlay issue at the relevant point of the virtual topology.

5 Partial Packet Provenance (3P)

This section defines the *provenance relation* that gathers tenant traffic observations into a queryable database.

It defines a provenance model that is focused on the following principles: **(1)** *events* only consist of the occurrence of frames and do not include network configuration events [16, 84]; **(2)** this formalism is entirely based on the observation of frames on monitored *links*. Links are straightforward to decompose to introduce additional vantage points. This model is *partial* in terms of **(1)** visibility of the topology and **(2)** filtering of packets (limiting packet history). Compared to the introduction of Packet Histories in NetSight [36] and the Datalog-based formal model in DTaP [84], 3P provides a more abstract model to reason about packet propagation when

Table 1: An example Crinkle API, portraying operations enabled by the design (Section 4).

Function, Parameters, and Description
`monitor_link(link, locality)`: Modify the network topology to monitor a link and *place* it in a locality as the operator supports, such as on the same host as a VM, on a host on the same rack as a given VM, or on a rack that is a bounded distance from a VM's host.
`get_provenance(filters)`: Get provenance stored in the analyzer's database, filtered using a pcap expression, time, and/or frame identifier. Output as a graph, table, or dictionary.
`probe(function, iface)`: Craft a packet by calling a user-provided function, and pass the packet to a specified monitored interface.
`trace_ping(src_ip, dst_ip, iface)`: Craft an ICMP Echo Request that will directly enter a specified monitored interface. Returns a graph showing the Request's history.
`dump_counters()`: Retrieves the RX and TX packet counters for each interface in the virtual network, then returns a dictionary of the results.
`missing_frames(l1, l2)`: Given two links, list frames that arrive at one but not the other.
`reordered_frames(l1, l2)`: Given two links, list frames that reach them in a different order.
`delay_gt(l1, l2, th)`: Given two links, list frames whose recorded times to appear on l2 after l1 are greater than a threshold.
`path_of(id)`: Given a packet identifier, show the list of links it traverses. Alternatively, given a link, list the links traversed by frames that are egressed by the link.
`state_when(id, t1, t2)`: Given a packet identifier and two time thresholds, list other packets that traversed a link within that time window.

querying Crinkle. Compared to switch-based postcard generation in NetSight, Crinkle emits postcard-like information from arbitrary virtual links in the topology.

Let ϕ denote a *frame* in the network. Each frame observed by a Crinkle node is assigned a distinct identifier "$\mathsf{name}(\phi)$" by that node. This function is idempotent and deterministic—therefore different Crinkle nodes assign the same name to the same frame and its derivatives when the frame is forwarded onto other links. If a frame is forwarded by a device, then the identifier of that frame is preserved. In contrast, two frames with identical contents ($\phi = \phi'$) that are first observed at *different* times are given distinct identifiers: $\mathsf{name}(\phi) \neq \mathsf{name}(\phi')$. In its implementation, Crinkle attaches a trailer to frames to encode this identifier as described in Section 6.

Associated with each frame ϕ is a *history* $\mathsf{hist}(\phi)$ that serializes the links $\subseteq L$ (the set of the tenant's virtual links) over which ϕ flowed, times at which the frame was observed at each link, and the changes on ϕ as it moved through links. Thus $\mathsf{hist}(\phi)$ is a set of triples of frames, timestamps, and links: $\mathsf{hist}(\phi) \subseteq \Phi \times \mathsf{Time} \times L$. History information is accumulated and analyzed in response to debugging queries, as will be explained later. If $\mathsf{hist}(\phi) = \emptyset$ then ϕ was never observed in the network. The statement $\mathsf{hist}(\phi) = \{(\phi, t_1, l_1), (\phi', t_2, l_2)\}$ (where $\phi \neq \phi'$ are frames with different content and $t_1 < t_2$) means that ϕ is first observed on link l_1 at time t_1, and then appeared as ϕ' when it was observed on l_2 at t_2. It is implicit in this model that l_1 and l_2 might connect to a network element that forwarded ϕ and transformed it into ϕ'. In this case, ϕ' is a frame that is *derived* from ϕ; although $\phi \neq \phi'$, $\mathsf{name}(\phi) = \mathsf{name}(\phi')$, so ϕ' can be placed in $\mathsf{hist}(\phi)$. One example of this is a device decrementing a time-to-live (TTL) field.

Let $l \in L$ denote a (virtual) link in the network, where L is a finite set of unique link identifiers in a tenant's topology. For a finite time, a sequence of frames $\vec{\phi}$ is associated with a link. A link can be *monitored*, in which case the link extends the histories of frames that cross that link. Links are monitored by bisecting them with a monitoring node. If a link l is not monitored, then $\mathsf{mon}(l) = \emptyset$, otherwise $\mathsf{mon}(l) = (l_1, l_2)$ where l_1 and l_2 are the two "halves" of the link being monitored.

5.1 Provenance Primitives for Debugging

This section formalizes the primitives that characterize the problems described in Section 3.3.

Intended Reachability of l by ϕ: The negation of this property is commonly called a *drop*. We define (with *set comprehensions* [25]) the reachability predicate of link l by frame ϕ as a *multiset* of every appearance of l in the third ordinate among the elements of $\mathsf{hist}(\phi)$:

$$\mathsf{Reach}(\phi, l) \stackrel{def}{=} \{l' \ : \ (_, _, l') \in \mathsf{hist}(\phi);\ l' = l\}$$

Then we formalize intended reachability as: $\mathsf{Reach}(\phi, l) \neq \emptyset$.

Unintended Reachability of F by ϕ: this captures the property where a frame reaches links in the network that should not be reached. This property is formalized as the reachability for a given set of "forbidden" links F: $\forall l \in F.\ \mathsf{Reach}(\phi, l) = \emptyset$.

Loops involving ϕ in l: We approximate loop detection by formalizing multiple copies of a frame flowing through the same link:

$$\mathsf{Repeat}(\phi, l) \stackrel{def}{=} |\mathsf{Reach}(\phi, l)| > 1$$

This is a necessary, though not sufficient, condition of a forwarding loop. It is not sufficient since the multiple frame copies might result from duplication, which is formalized next.

Duplication of ϕ from l to L': Reachability and loops alone are too weak to capture correctness since they are unable to count increases in frame observations. For example, if a frame is duplicated, it is possible to have two instances of intended reachability (if both copies reach their destination), or even *both* intended and unintended reachability (if one copy reaches its destination, and the other copy is forwarded to a forbidden part of the network). For a network element whose links are represented by the set $\{l\} \cup L'$, then for ϕ arriving over l we define:

$$\mathsf{Duplicate}(\phi, l, L') \stackrel{def}{=} |\mathsf{Reach}(\phi, l)| < \Sigma_{l' \in L'} |\mathsf{Reach}(\phi, l')|$$

This formalizes the observation that *more* copies of a frame were forwarded than expected.

5.2 Debugging using Packet Provenance

To debug problems **Pr1**–**Pr4** from Section 3.3, a tenant instantiates monitors as illustrated in Figure 3, filters observations at those monitors for specific traffic of interest (Section 4.4) and applies queries to their Crinkle nodes to localize and better understand a network problem. These queries reduce to the Reach, Repeat, and Duplicate primitives introduced earlier.

Packet loss (**Pr1**) uses Reach to determine the path(s) followed by the tenant's traffic and detect where packets are going missing, as well as how frequently the loss is occurring. Crinkle does not require the tenant to inject extra traffic into their topology—the Crinkle nodes can use the tenant's existing traffic flows. Latency and jitter (**Pr2**) are detected by making accurate timing measurements

along network path(s), and identifying anomalous delays. Crinkle's implementation uses commonly available high-precision time synchronization and user-space networking primitives to achieve quasi-deterministic behavior. Loops, packet duplication, and incorrect forwarding (**Pr3**) are detected by finding patterns of unwanted traffic in the network—this involves using Reach, Repeat, and Duplicate. Crinkle uses 3P (**Pr4**) to capture debugging-related queries for **Pr1**–**Pr3**. Crinkle can track traffic that flows through (and is modified by) network elements to provide tenants with a detailed picture of network behavior. Crinkle does not require the tenant to tag or identify their traffic—that would be burdensome and error-prone for tenants. Instead, Crinkle adds custom trailers (to encode packet identifiers) to tenant frames. Even if a frame's header stack or its field values were to change, the tag would be preserved as long as the network respects a slightly lower practical MTU and avoids fragmentation. This provides a best-effort tracking of packets across different processing stages in their network. In our implementation (Section 6), tags are added as 16-byte trailers.

6 Implementation

Crinkle's implementation of the concepts from Sections 4 and 5 consists of three parts: the tenant-facing API, the code running on the monitor VMs, and the code running on an *analyzer* VM. The implementation has been contributed to FABlib [28] (FABRIC's management API). Crinkle's tenant-facing API extends that of FABlib and consists of 1,500 lines of Python. In turn, it uses FABlib's API functions to create and access virtualized resources. Similar functions were used for Crinkle's port to SPHERE.

The Crinkle prototype uses SPADE [32], a provenance tracker, to manage the provenance database. Crinkle adds a Python script to parse the data from the monitors and send it to SPADE. The Crinkle API handles transforming tenant debugging queries into SPADE queries by creating successive graph operations, one for each filter type, which reduce the database to the desired provenance graph. The provenance graph that SPADE builds conforms to the Open Provenance Model, described earlier in Section 4.2.

Crinkle nodes run a C program (1,000 lines) that uses the Data Plane Development Kit (DPDK) API [24] to handle adding frame IDs and crafting the packets that relay data for analysis. Analysis data consists of metadata about the frame—the trailer, size, location, and time. Crinkle can also be configured to clone and store some or all of a packet. The DPDK application has threads dedicated to each of the two ports that connect with tenant links—providing a bump-in-the-wire function. Adding the trailer and crafting the analyzer packets is left to a tenant-chosen number of worker threads. The number of worker threads can be scaled up for increased performance. This setup, using the DPDK "**soring**" object, guarantees that packets are transmitted in the order they are received. This ensures that Crinkle nodes cannot introduce reordering, and they behave more deterministically than if they used Receive-Side Scaling (RSS) to spread the work across cores. The latency introduced by this setup has a median of under 9 µs and is evaluated in Section 7.1.1.

6.1 Frame Identifiers

In a virtualized network, packets might undergo changes that make it difficult to understand the source of traffic or the path that it took.

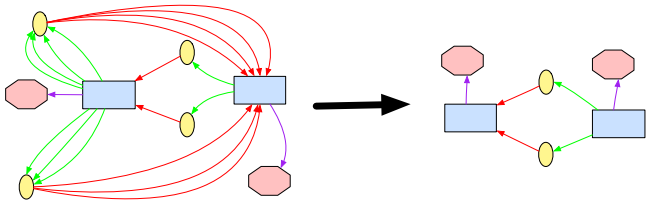

Figure 5: Demonstration of TCP flow abstraction. The filtering collapses multiple packet edges into a single flow-level edge.

For example, Section 3.3 references the network address mappings that are common in a cloud datacenter (as the *virtual* IP address of a server is translated to its real IP address in the network [9]).

Crinkle preserves the identity of packets by tagging each frame that crosses a monitored link. A tag consists of a unique 16-byte identifier that is added as a trailer without incrementing the length fields of the frame's payload. Since the tag is added as a trailer, and most network traffic consists of IP packets, these trailers are discarded by the destination host's network stack. This tag is added when a frame first crosses a monitored link, and it is observed and preserved when a tagged frame next crosses a monitored link. From empirical evaluation (Section 7.4) we found tags are preserved on non-monitored links and by virtual and physical network elements.

The 16-byte tag is structured as follows. The first 8 bytes are a 64-bit timestamp, the next 2 bytes are the unique monitor ID, the next 2 bytes are the ingress port identifier, the next 2 bytes are the 7th and 8th bytes of the timestamp, and the final 2 bytes are hard-coded as the hexadecimal value 0x6587. This tag is designed to both be both unique and recognizable, since the system needs to be able to parse the trailer both to avoid adding endless trailers, and also to notice that a packet with a certain trailer passed through.

6.2 Filtering Mechanisms

As raised in Section 4.4, capturing all network traffic quickly pollutes the provenance store with information. Thus, our implementation has filtering mechanisms for both in-network filtering (using DPDK) and post-capture transformation (using SPADE). These mechanisms target two common patterns: TCP flows, which benefit from semantic abstraction, and IPv6 Neighbor Discovery (ND) messages, which are protocol-specific control packets of limited debugging value. Figure 5 illustrates the effect of filtering mechanisms on provenance graph structure.

6.3 Research Reproducibility

Crinkle is already available to the research community in FABRIC's public API[1]. The participants of the user study described in Section 7.3 reproduced the work described in this paper and used the network debugging API described in this paper. We share artifacts that provide a short overview[2] of Crinkle and the 11-node topology[3] that is used in Section 7.2. Other researchers can also use the repo and artifacts to adapt and evaluate the Crinkle debugger in other virtualized network environments.

[1] https://github.com/fabric-testbed/fabrictestbed-extensions/tree/main/fabrictestbed_extensions/fablib/crease
[2] https://crease.cs.iit.edu/knit11/artifact
[3] https://crease.cs.iit.edu/knit11/large-artifact

7 Evaluation

This section evaluates the scalability, applicability, robustness, and measurement sensitivity of the Crinkle prototype. This evaluation was carried out on the FABRIC [8] testbed, a multi-tenant, virtualized infrastructure that spans the continental US and locations in Asia and Europe. FABlib is the API that is used by FABRIC researchers to reserve and control their virtual networks. Our evaluation used FABlib v1.9.2 and DPDK v25.07. We have also confirmed that Crinkle works on the SPHERE testbed [46].

7.1 Scalability

7.1.1 Monitor Performance. To measure the throughput and latency of a monitor, we used a dedicated Mellanox ConnectX-6 for each "in-band" port (which links to nodes in the tenant's topology), while its "out-of-band" port (which links to the Crinkle analyzer) and all other VMs used SR-IOV virtual functions on ConnectX-6s. To ensure precise measurements, we were the only tenant using the physical hardware on the testbed during this experiment. The monitor was allocated 1 GB of RAM and 6 CPU cores: 1 core is allocated to each of the monitor's two ports that connect to the tenant's network, and 4 additional cores are used as worker cores (for trailer and analyzer packet generation). As DPDK fully utilizes a core even when idle, we did not measure core utilization.

We measured the latency introduced by the code by sampling the time after a burst was transmitted and subtracting the receive time of the oldest packet in that burst. We sampled one out of every 1,024 bursts for approximately 35 seconds, producing 6,692 records. This experiment was repeated in two environments: **(1)** a quiet environment where the monitor was the only VM on the physical hardware and **(2)** a noisy environment where 30 VMs (using 2 cores, 5.2 GB RAM) were doing matrix multiplications on a 15,000 × 15,000 matrix in an endless loop. On the physical host, this represented 46.88% of total core usage and 32.63% of total memory usage. The distribution of this latency in the noisy environment is shown in Figure 6, with a mean of 8.68 µs (σ 3.87 µs) and 24 outliers beyond 22 µs. In the quiet environment, the distribution was similar but with only 6 outliers beyond 22 µs and a mean of 7.2 µs (σ 3.1 µs).

For throughput, the monitor was connected between two VMs, a generator and a receiver. The generator VM would create traffic of varying packet sizes into one port and the throughput was measured at the receiver VM, both using Pktgen-DPDK [54]. We recorded the highest throughput which had less than 0.1% packet loss. To evaluate the worst-case scenario for processing, no filtering was employed; therefore, every packet that crossed the monitor was also cloned and sent out the analyzer port. The results are shown in Figure 7. The Crinkle prototype supports 10 Gbps of *any* frame size, even in this worst-case scenario. We record peak bandwidths of 55 Gbps and 15.6 Mpps (millions of packets per second).

Contrast with status quo. To put Crinkle's performance into context, we find that leading commercial cloud providers limit instances up to 25 Gbps [2]. Further, today's cloud testbed users usually make do with tcpdump for network debugging [77]. We carried out an experiment to measure how tcpdump compares to a Crinkle monitor, using the same setup described above. We found that tcpdump supported only 3.800–3.912 Gbps with 1,280-byte frames and 0.242–0.250 Gbps with 64-byte frames.

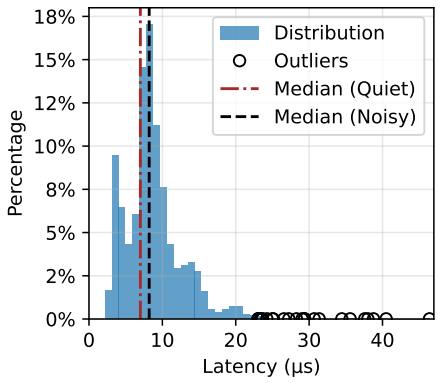

Figure 6: Latency distribution of sampled packets in a noisy environment, with a median of 8.21 µs. Shown is the median of 7 µs for the same setup in a quiet environment, which had a similar sub-22 µs distribution but fewer outliers.

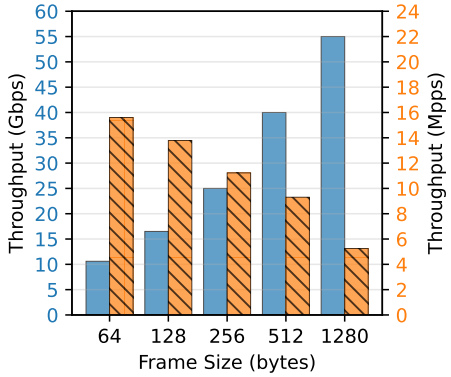

Figure 7: The throughput of the monitor at different frame sizes. The throughput is measured in Gigabits per second on the left y-axis, and in Millions-of-packets per second on the right y-axis.

7.1.2 Resource Usage. In the current prototype, monitors use a minimum of 1 GB of RAM and 4 CPU threads.[4] The cost of the analyzer(s) would depend on how much traffic is being captured by the tenant—as in NetSight [36], multiple databases can be used to handle traffic capture. The cost scales linearly with the number of monitors. Quantifying the number of monitors that are needed to accurately debug a problem depends on the topology, workload size, and the problem's frequency. To start, assume the total time to debug a problem that appears with frequency F is $D = F \times c$, with c some constant of the time taken to analyze and debug each issue after it appears. Pessimistically, for a topology with L links, monitors would be deployed over all L links. Given each analyzer node handles some A bandwidth and an average captured bandwidth C for each monitor, then the cost is $O\left(L\left(1+\frac{C}{A}\right)D\right)$ as the captured traffic is evenly balanced across the analyzers.

[4] One core for each "half" of a bisected link, and two worker threads.

7.1.3 Filtering-based Abstraction. Table 2 shows the impact of the filtering from Section 6.2 across different applications. The original query response graph size is characterized as V_{raw} vertices and E_{raw} edges, while the abstracted version contains V_{filtered} vertices and E_{filtered} edges. In all cases that we analyzed, filtering substantially reduces both the vertex and edge counts of the provenance graphs, reflecting the abstraction of TCP flows and elimination of neighbor discovery traffic. Applications such as `ssh`, `netcat`, `wget`, and `ab` exhibit similar filtered graph sizes, since each establishes a single TCP connection, leading to a consistent set of retained flow-level entities. By contrast, `iperf3` produces a noticeably larger filtered graph. This is because, in this example, `iperf3` establishes two TCP connections during execution (one for control messaging, and the other for data transfer), thereby preserving additional handshake and flow-confirmation subgraphs.

Table 2: A Crinkle provenance graph consists of two classes of elements—those induced by the underlying network topology, and those resulting from packet traversals. The graph's size is lower-bounded by the former: 39 vertices and 28 edges in our setting.

App.	$\lvert V_{\text{raw}}\rvert$	$\lvert E_{\text{raw}}\rvert$	$\lvert V_{\text{filtered}}\rvert$	$\lvert E_{\text{filtered}}\rvert$
iperf3	89	504	51	52
ssh	73	600	45	40
netcat	63	172	45	40
wget	62	152	45	40
ab	61	164	45	40

7.2 Case Studies

To evaluate applicability, we describe case studies that demonstrate how Crinkle can be used to logically debug problems belonging to the classes of network problems described in Section 3.3. These studies demonstrate the following features:

- Querying using spatial (topological) and temporal constraints over the occurrence of network traffic. This is shown in Section 7.2.1, and gives an important convenience for debugging.
- The ability of Crinkle to work with arbitrarily encapsulated data—such as packets that are tunneled across the network. This is shown in Section 7.2.2 and relies on tagging (Section 6.1) to provide resilient metadata for debugging.
- Detecting problems in the underlay or in hosts shared with "noisy neighbors" that affect the performance of a tenant's workload—demonstrated in Section 7.2.3. This allows tenants in multi-tenant, virtualized cloud networks to infer problem causes outside of their overlay.

Each case study is drawn from the literature and features a malfunctioning network element or link. Figure 8 shows the overlay topology that is used in every case study. It consists of two VM endpoints (`h1` and `h2`) separated by multiple paths across VMs running BMv2 [12] software routers (`r`X). In total, there are 6 possible 6-hop paths from `h1` to `h2`, and traffic can be forwarded across all paths.[5] A monitor was inserted in every link of this topology.

[5]Two examples of such paths are `h1` → `r1` → `r2` → `r3` → `r4` → `r5` → `h2` and `h1` → `r1` → `r2` → `r9` → `r4` → `r5` → `h2`.

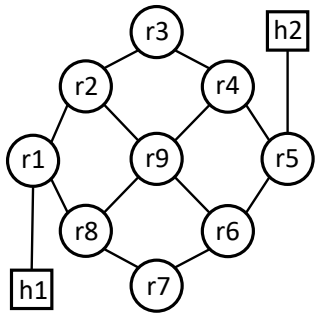

Figure 8: An abstract topology of two endpoints separated by multiple routers and paths, used for the case studies and user study. It balances complexity (6 paths across 9 routers) with size (11 nodes).

*7.2.1 Case Study 1 (**Pr1** and **Pr3**): Incorrect Forwarding [70].* In this case, router `r7` has an incorrect forwarding rule that leads to packets from `r8` being forwarded back to `r8` instead of on to `r6`, causing those packets to be dropped at `r8`. This initially appears as an intermittent reachability failure between `h1` and `h2`, as whenever the packet is forwarded along that path it is dropped but it otherwise succeeds. To troubleshoot this problem, we use monitors to record traffic until there is a reachability error. We then query the data store using the timestamp of the packet that leaves `h1` to get the packet's ID, which we can use to filter the output further—particularly if a large quantity of traffic was captured—and trace that packet through the network.

A log shows a packet was dropped at time 1760427464.6503992, which could represent any communication failure that is logged at an end-host or other monitoring system (e.g. a failed TCP handshake or corrupted packet). The next packet at time 1760427464.6512332 succeeds. We execute a single `get_provenance` command with these start and end times to get the provenance of this packet as a table by looking at the time range the missing packet could have been in the network. In this case, due to the low traffic, this is the only filtering query needed, as the returned provenance contains just a single packet record. Table 3 shows the retrieved provenance in this case, as an example of the output. If it contained too many packets, we could tighten the time range to get just the sent packet, then use the packet ID to get the full provenance.

The provenance shows the packet's path: `h1` → `r1` → `r8` → `r7` → `r8`. The provenance graph of that packet shows it leaving `h1` going to `r1`, then on to `r8`, then to `r7` and back to `r8`.

This indicates that the packet is dropped at `r8`, as all paths out of it are monitored yet the provenance ends. Our suspicion therefore turns to `r8`, and we examine its forwarding rules. Piecing together information, we see that the error actually arose from `r7` since it created a loop by forwarding back towards `r8`. Using the provenance history, we localize the problem to `r7` because of its forwarding behavior. This `r7`-to-`r8` forwarding loop in the provenance graph is akin to what is shown in Figure 4.

Indeed, this error arises from a subtle typo in `r7`'s forwarding rules that send traffic destined towards `h2` out of port 1 (towards `r8`) rather than port 0 (towards `r6`). As the tenant, we can solve this issue ourselves by patching the misconfiguration.

Packet ID	TX Interface	RX Interface	Size	Timestamp (Epoch)	Ether Type	IP Protocol
0-1-1760427464650399347	h1-eth0-p1	r1-eth2-p1	98	1760427464.6503992	0x800	1
0-1-1760427464650399347	r1-eth1-p1	r8-eth0-p1	98	1760427464.6512332	0x800	1
0-1-1760427464650399347	r8-eth1-p1	r7-eth0-p1	98	1760427464.6714187	0x800	1
0-1-1760427464650399347	r7-eth0-p1	r8-eth1-p1	98	1760427464.6725268	0x800	1

Table 3: Provenance of a dropped packet. The highlighted column shows the identifier of the receiving interface of the packet. The packet is incorrectly forwarded from `r7` to `r8` and dropped. Packet content provenance (like source and destination IP and port) is excluded for brevity. The host names appear underlined in the network interface identifiers—such as "h1" in "h1-eth0-p1".

*7.2.2 Case Study 2 (**Pr4**): Incorrect Headers [80].* In this case, packets between `r1` and `r5` are encapsulated in a GTP-U [83] tunnel, which consists of an outer IP, UDP, and 8-byte GTPv1 header. Router `r9` is faulty and does not emit the GTP header, which causes `r1` and `r5` to produce packets with corrupted IP headers as they mistakenly strip the first 8 bytes due to the missing GTPv1 header when decapsulating. As in the previous case study, this would appear as a reachability issue where traffic is intermittently lost between the endpoints. To troubleshoot this, we query the database for one of the faulty transmissions, as in the previous case study.

We probe the path between `h1` and `h2` with ping packets until one fails, then use the same timestamp technique to retrieve provenance of the failed ping. In this case, there are two packet IDs, representing the ping request which succeeds and the response which fails. We find that the response does arrive at `h1`, but the IP information is clearly corrupted with a protocol of 0 and incorrect addresses. Tracing it back towards the source, we find the response packet enters `r9` with a size of 134 bytes and exits with a size of 126 bytes. This is our evidence that `r9` is manipulating the packet. Then we investigate `r9` and find that its configuration is faulty and is not emitting an 8-byte tunneling header. When the whole set of tunnel headers is removed from the packet at `r1`, the front 8 bytes of the lower headers are removed, producing the corrupted packet. The problem is revealed once the packet's header stack is examined. As the tenant, we solve this issue by fixing the configuration on `r9`.

*7.2.3 Case Study 3 (**Pr2**): Latency Anomalies [17].* In this case study, the underlay server that hosts the router VM `r3` between `r2` and `r4` is heavily loaded. To tenants, the symptom consists of packets occasionally having unusual latency spikes. Rather than missing packets, we are detecting volatility in the latency of the packet arrivals. With pings, we see some pings take less than a second to return, while others can take over 3 seconds. To troubleshoot this, we again query the database for the transmission of interest. Inspecting the provenance table of a high-latency ping can help localize the problem. We call `get_provenance` to retrieve the provenance of the discrepant ping's lifespan.

The provenance reveals that the time taken for packets to pass through `r3` is noticeably higher than for other network elements. We can follow the flow of the ping response packet to `r3`, where it leaves over two seconds after arrival, contributing the vast majority of the latency. Inspection of the forwarding program in `r3` doesn't reveal any errors, so we can conclude that the underlay is either faulty or under load from another tenant.

At this point, as the tenant, we would either adapt our virtual network to the noisy underlay conditions—by increasing timeout values, for example—or reach out to the operator.

7.3 Pilot Deployment: User Study

We conducted a study using 22 external participants (17 professionals, 5 students) to determine if Crinkle was more effective than alternative tools that are commonly used by FABRIC tenants (generally a mix of ping and tcpdump). The full results of the study, as well as details on methodology and motivation, are published in a stand-alone article [77]. We timed how long it took the participants to localize the fault to one of the routers, with half using Crinkle and half using their free choice of method. Among participants with middling experience of FABRIC and general network debugging who completed both exercises (3 professionals each for Crinkle/non-Crinkle), we observed some gains in time to solve problems. The non-Crinkle participants (who used ping and tcpdump) took 110, 72.2, and 131 (average 104) minutes on Case Study 1 and 39.1, 1.07, and 36.9 (average 25.7) minutes on Case Study 2. Meanwhile, the Crinkle participants took 155, 21, and 56.8 (average 77.7) minutes on Case Study 1 and 40.5, 5.38, and 2.2 (average 16) minutes on Case Study 2.

Other experience bands returned less useful data; in many cases, they were unable to finish due to unfamiliarity with FABRIC or debugging or due to a lack of time. We share the results of the medium experience sample since it indicates there is potential for Crinkle to be more effective than traditional techniques, but a larger study is necessary to determine the true extent. This pilot deployment has seen Crinkle integrated into the API for the FABRIC testbed, and we have evaluated it to be compatible with the SPHERE testbed.

7.4 Robustness of Tag Preservation

For Crinkle to accurately track frames across a network, it is also necessary that the frame IDs be preserved. If a frame were to cross a link and lose its unique ID, then the provenance history would prematurely end. While all upstream provenance would be untouched, downstream provenance would be under a new ID. To evaluate the durability of the trailer tag over different types of network elements, we sent packets containing the tag across the elements that are shown in Table 4. The results confirm that the tag was durable to any layer-2 service or architecture, including bridging in Linux/BSD, Open vSwitch [1], and a Tofino2 P4-programmable switch. On FABRIC and SPHERE, the trailer was preserved across

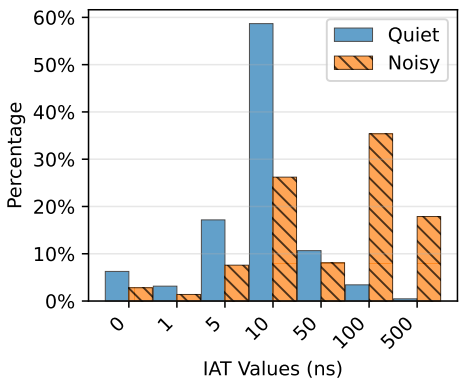

Figure 9: The run-to-run variation in packet IATs (inter-arrival times) at 40 Gbps when there is no other tenant traffic ("Quiet") and when there is a varying 35–50 Gbps of other-tenant traffic on a 100 Gbps link ("Noisy"). Measured on FABRIC using hardware timestamping of a Mellanox ConnectX-6 SR-IOV virtual function, which was synchronized via PTP through the VM to a GPS time source [52].

all virtual network services. This included FABRIC "Facility Ports" connecting FABRIC to researcher institutions [60].

Element	Pres.	Element	Pres.
Linux Bridge	●	FABRIC/SPHERE L2	●
Linux IP Fwd	○	FABRIC/SPHERE L3	●
BSD Bridge	●	FABRIC Fac. Port	●
BSD IP Fwd	○	Open vSwitch	●
Alveo-to-NIC	●	Tofino2	●

Table 4: Forwarding elements where the trailer was preserved. Tested on Linux kernel 5.15.0-143-generic (FABRIC) and 5.15.184 (local); FreeBSD using kernel 14.0-RELEASE-p6; FABRIC using FABlib v1.9.2; SPHERE using mergexp v0.3.0; Open vSwitch version 2.17.9.

7.5 Measurement Sensitivity

We evaluated the monitor's sensitivity to underlay activity by replaying a buffer of packets with carefully controlled sending and timing. Figure 9 shows how the performance of a virtual network can become more inconsistent when FABRIC's shared underlay network is noisy with other tenant traffic. This inconsistency is observed through the variation in Inter-Arrival Times (IATs) between packets. Using Crinkle, tenants can accurately replay traffic from monitors to characterize the consistency of the network over time. By being able to characterize the network's consistency, tenants can debug issues that arise from network nondeterminism by measuring whether the network has high jitter or delays, and localize virtual links that are behaving non-deterministically. This was investigated more fully in a stand-alone paper about network consistency across testbeds [76], and this section limits itself to providing the key observation on Crinkle's deployment environment.

8 Future Work

Directions for future research include porting Crinkle to other research testbeds, reporting on the evolution of the API to use features that are unevenly available across testbeds, and documenting the preservation of tags used by Crinkle in different environments and over different equipment, building on Section 7.4.

Another direction of future work consists of porting Crinkle to run on a commercial cloud network. We believe this to be almost entirely feasible, except that existing cloud providers do not (yet) provide placement primitives that could be used by Crinkle to diagnose underlay problems, as discussed in Section 4.1.

Future work could also extend Crinkle to overcome the limitations described in Section 4.5, and investigate the following: (1) extending Crinkle with primitives for debugging in multi-cloud environments, (2) automating placement decisions for monitors based on tenant-, cross-tenant, and operator-provided heuristics to quickly localize faults, and (3) aggregating monitors and load-balancing across them to support higher bandwidth. These directions rely on further research into how to coordinate across resources—including across different clouds and across monitors in the same or different clouds—to avoid inferring incorrect results or introducing bugs in the monitored networks.

9 Conclusions

This paper motivates the need for tenant-driven debugging tools in cloud testbeds, and argues that these tools could also be useful in cloud networks more generally. In cloud testbeds, each researcher is a separate tenant. This paper proposes a simple architecture for a cloud network that can provide better debugging features for tenants—including the ability to diagnose problems in the network underlay. We found that this architecture can be realized in existing cloud testbeds, and described the design and implementation of Crinkle to provide tenants with convenience and autonomy.

Crinkle provides tenants with configurably placed monitoring resources. Monitors are frugally resourced and serve to create, observe, and record traffic tags. This approach builds on ideas such as those in NetSight [36] but does not rely on programmable hardware, and targets a multi-tenant cloud network rather than bare-metal access to hardware. The resulting system was evaluated rigorously through a systems-oriented evaluation that measured the performance and overhead of Crinkle's infrastructure, and through an initial usability evaluation involving volunteers from the testbed community, including professional network engineers.

Acknowledgments

We are grateful for feedback from the anonymous reviewers and shepherd, Vaneshi Ramdhony, Komal Thareja, Mert Cevik, Hyunsuk Bang, Dale Carder, Ezra Kissel, Ashok Sunder Rajan, Henry Zhu, and the NW LIST (Lab for Internet and Security Technology) and UChicago NOISE (Network Operations and Internet Security) groups. We thank the participants of the user experiment described in Section 7.3 for their contribution. This work was supported by the National Science Foundation (NSF) under award 2346499. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

References

[1] 2016. Open vSwitch. https://www.openvswitch.org/.

[2] 2025. Amazon EC2 instance network bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html.

[3] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 233–248.

[4] Katerina Argyraki, Petros Maniatis, David Cheriton, and Scott Shenker. 2004. Providing packet obituaries. In *ACM HotNets-III*.

[5] Katerina Argyraki, Petros Maniatis, Olga Irzak, Subramanian Ashish, and Scott Shenker. 2007. Loss and Delay Accountability for the Internet. In *2007 IEEE International Conference on Network Protocols*. 194–205. doi:10.1109/ICNP.2007.4375850

[6] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically Finding the Cause of Packet Drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 419–435. https://www.usenix.org/conference/nsdi18/presentation/arzani

[7] Sandeep Bal, Zhaoyang Han, Suranga Handagala, Mert Cevik, Michael Zink, and Miriam Leeser. 2024. P4-Based In-Network Telemetry for FPGAs in the Open Cloud Testbed and FABRIC. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–6. doi:10.1109/INFOCOMWKSHPS61880.2024.10620664

[8] Ilya Baldin, Anita Nikolich, James Griffioen, Inder S Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. 2019. FABRIC: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing* 23, 6 (2019), 38–47.

[9] Tom Barbette, Erfan Wu, Dejan Kostić, Gerald Q. Maguire, Panagiotis Papadimitratos, and Marco Chiesa. 2021. Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency. *IEEE/ACM Trans. Netw.* 30, 1 (Sept. 2021), 354–367. doi:10.1109/TNET.2021.3113370

[10] Shoaib Basu and Deepak Nadig. 2024. Offloading NVMe over Fabrics (NVMe-oF) to SmartNICs on an at-scale Distributed Testbed. In *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. 316–318. doi:10.1109/NetSoft60951.2024.10588915

[11] Jeronimo Bezerra, Italo Brito, Arturo Quintana, Julio Ibarra, Vasilka Chergarova, Renata Frez, Heidi Morgan, Marc LeClerc, and Arun Paneri. 2021. Deploying per-packet telemetry in a long-haul network: the AmLight use case. In *2021 IEEE Workshop on Innovating the Network for Data-Intensive Science (INDIS)*. 44–49. doi:10.1109/INDIS54524.2021.00010

[12] BMv2 2026. Behavioral Model v2. https://github.com/p4lang/behavioral-model/tree/main.

[13] Jack Brassil. 2022. Network Traffic as a Federated Testbed Service. In *2022 IEEE Future Networks World Forum (FNWF)*. 450–455. doi:10.1109/FNWF55208.2022.00086

[14] Lloyd Brown, Emily Marx, Dev Bali, Emmanuel Amaro, Debnil Sur, Ezra Kissel, Inder Monga, Ethan Katz-Bassett, Arvind Krishnamurthy, James McCauley, Tejas Narechania, Aurojit Panda, and Scott Shenker. 2024. An Architecture For Edge Networking Services. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 645–660. doi:10.1145/3651890.3672261

[15] Prasad Calyam and Martin Swany. 2015. Research challenges in future multi-domain network performance measurement and monitoring. *SIGCOMM Comput. Commun. Rev.* 45, 3 (July 2015), 29–34. doi:10.1145/2805789.2805795

[16] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. One Primitive to Diagnose Them All: Architectural Support for Internet Diagnostics. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 374–388. doi:10.1145/3064176.3064212

[17] Jingrong Chen, Hong Zhang, Wei Zhang, Liang Luo, Jeffrey Chase, Ion Stoica, and Danyang Zhuo. 2022. NetHint: White-Box Networking for Multi-Tenant Data Centers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1327–1343. https://www.usenix.org/conference/nsdi22/presentation/chen-jingrong

[18] Cisco Systems, Inc. 2026. Cisco Network Services Orchestrator. https://nso-docs.cisco.com/.

[19] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[20] DARPA. 2020. Transparent Computing. https://github.com/darpa-i2o/Transparent-Computing.

[21] Soumyadeep Datta and Fraida Fund. 2023. Replication: "When to Use and When Not to Use BBR". In *Proceedings of the 2023 ACM on Internet Measurement Conference* (Montreal QC, Canada) *(IMC '23)*. Association for Computing Machinery, New York, NY, USA, 30–35. doi:10.1145/3618257.3624837

[22] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. doi:10.1145/1327452.1327492

[23] Rui Ding, Xunpeng Liu, Shibo Yang, Qun Huang, Baoshu Xie, Ronghua Sun, Zhi Zhang, and Bolong Cui. 2024. RD-Probe: Scalable Monitoring With Sufficient Coverage In Complex Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 258–273. doi:10.1145/3651890.3672256

[24] DPDK Project 2026. Data Plane Development Kit. https://www.dpdk.org/.

[25] Herbert B Enderton. 1977. *Elements of Set Theory*. Academic Press.

[26] FABRIC. 2025. Network Interfaces in FABRIC VMs. https://learn.fabric-testbed.net/knowledge-base/network-interfaces-in-fabric-vms/.

[27] FABRIC. 2026. How VMs are sized in FABRIC. https://learn.fabric-testbed.net/knowledge-base/how-vms-are-sized-in-fabric/.

[28] FABRIC Knowledge Base 2026. FABRIC Experiment Management APIs (FABLib). https://learn.fabric-testbed.net/article-categories/fablib-api/.

[29] FABRIC Knowledge Base 2026. Network Services in FABRIC. https://learn.fabric-testbed.net/knowledge-base/network-services-in-fabric/.

[30] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. 2014. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 533–546.

[31] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[32] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *13th ACM/IFIP/USENIX International Middleware Conference*. http://www.csl.sri.com/users/gehani/papers/MW-2012.SPADE.pdf

[33] Jose Gomez, Elie F. Kfoury, Jorge Crichigno, and Gautam Srivastava. 2024. Improving TCP Fairness in Non-Programmable Networks Using P4-Programmable Data Planes. In *2024 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*. 102–107. doi:10.1109/BlackSeaCom61746.2024.10646283

[34] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 139–152. doi:10.1145/2829988.2787496

[35] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2012. Where is the debugger for my software-defined network?. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (Helsinki, Finland) *(HotSDN '12)*. Association for Computing Machinery, New York, NY, USA, 55–60. doi:10.1145/2342441.2342453

[36] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 71–85.

[37] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 200–213. doi:10.1145/3341302.3342094

[38] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding Network Misconfigurations by Automatic Template Inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 999–1013. https://www.usenix.org/conference/nsdi20/presentation/kakarla

[39] Joud Khoury, Timothy Upthegrove, Armando Caro, Brett Benyo, and Derrick Kong. 2020. An Event-based Data Model for Granular Information Flow Tracking. *12th Theory and Practice of Provenance (TaPP)* (2020).

[40] John Kim, William J. Dally, and Dennis Abts. 2007. Flattened butterfly: a cost-efficient topology for high-radix networks. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 126–137. doi:10.1145/1273440.1250679

[41] Dexter Kozen. 2014. NetKAT: A Formal System for the Verification of Networks. In *Programming Languages and Systems*, Jacques Garrigue (Ed.). Springer International Publishing, 1–18.

[42] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. 2019. DETER: Deterministic TCP Replay for Performance Diagnosis. In *16th USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 19). USENIX Association, Boston, MA, 437–452. https://www.usenix.org/conference/nsdi19/presentation/li-yuliang

[43] Ze Li and Ryan Huang. 2024. Gray Failure: The Achilles' Heel of Cloud-Scale Systems *(SREcon '24)*. USENIX Association, San Francisco, CA.

[44] Xu Liu, Peng Zhang, Anubhavnidhi Abhashkumar, Jiawei Chen, and Weirong Jiang. 2024. Automatic Configuration Repair. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks* (Irvine, CA, USA) *(HotNets '24)*. Association for Computing Machinery, New York, NY, USA, 213–220. doi:10.1145/3696348.3696895

[45] Sarah McClure, Zeke Medley, Deepak Bansal, Karthick Jayaraman, Ashok Narayanan, Jitendra Padhye, Sylvia Ratnasamy, Anees Shaikh, and Rishabh Tewari. 2023. Invisinets: Removing Networking from Cloud Networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 479–496. https://www.usenix.org/conference/nsdi23/presentation/mcclure

[46] J Mirkovic, B Kocoloski, and D Balenson. 2024. Enabling reproducibility through the SPHERE research infrastructure. In *Usenix ;login:*.

[47] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. 2011. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems* 27, 6 (2011), 743–756. doi:10.1016/j.future.2010.07.005

[48] Rodrigo Moreira, Rodolfo S. Villaça, Moisés R.N. Ribeiro, Joberto S.B. Martins, Joao Henrique Corrêa, Tereza C. Carvalho, and Flávio de Oliveira Silva. 2025. An intelligent native network slicing security architecture empowered by federated learning. *Future Generation Computer Systems* 163 (2025), 107537. doi:10.1016/j.future.2024.107537

[49] Alicia Esquivel Morel, Durbek Gafurov, Prasad Calyam, Cong Wang, Komal Thareja, Anirban Mandal, Eric Lyons, Michael Zink, George Papadimitriou, and Ewa Deelman. 2023. Experiments on Network Services for Video Transmission using FABRIC Instrument Resources. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–6. doi:10.1109/INFOCOMWKSHPS57453.2023.10225817

[50] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and Precise Triggers in Data Centers. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 129–143. doi:10.1145/2934872.2934879

[51] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. 2011. NetLord: a scalable multi-tenant network architecture for virtualized datacenters. *SIGCOMM Comput. Commun. Rev.* 41, 4 (Aug. 2011), 62–73. doi:10.1145/2043164.2018444

[52] Hussamuddin Nasir and Pinyi Shi. 2023. FABRIC PTP. https://github.com/fabric-testbed/ptp?tab=readme-ov-file.

[53] OpenStack 2026. OpenStack. https://www.openstack.org/.

[54] Pktgen-DPDK 2024. Pktgen-DPDK: DPDK based packet generator. https://github.com/pktgen/Pktgen-DPDK.

[55] Justin Presley, Xi Wang, Xusheng Ai, Tianyuan Yu, Tym Brandel, Proyash Podder, Varun Patil, Alex Afanasyev, F. Alex Feltus, Lixia Zhang, and Susmit Shannigrahi. 2024. Hydra: A Scalable Decentralized P2P Storage Federation for Large Scientific Datasets. In *2024 International Conference on Computing, Networking and Communications (ICNC)*. 810–816. doi:10.1109/ICNC59896.2024.10556269

[56] Chengyi Qu, Joaquin Chung, Zhengchun Liu, Tekin Bicer, and Rajkumar Kettimuthu. 2022. Evaluating SciStream (Federated Scientific Data Streaming Architecture) on FABRIC. In *2022 IEEE/ACM International Workshop on Innovating the Network for Data-Intensive Science (INDIS)*. 25–31. doi:10.1109/INDIS56561.2022.00009

[57] Krishnan Raghavan, George Papadimitriou, Hongwei Jin, Anirban Mandal, Mariam Kiran, Prasanna Balaprakash, and Ewa Deelman. 2025. Advancing anomaly detection in computational workflows with active learning. *Future Generation Computer Systems* 166 (2025), 107608. doi:10.1016/j.future.2024.107608

[58] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. 2023. Hydra: Effective Runtime Network Verification. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 182–194. doi:10.1145/3603269.3604856

[59] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. 2017. Passive Realtime Datacenter Fault Detection and Localization. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 595–612. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/roy

[60] Paul Ruth. 2022. FABRIC's First FacilitlyPort: Deploying Experiments Spanning Chameleon and FABRIC. https://blog.chameleoncloud.org/posts/fabrics-first-facilitlyport-deploying-experiments-spanning-chameleon-and-fabric/.

[61] Paul Ruth, Ilya Baldin, Komal Thareja, Tom Lehman, Xi Yang, and Ezra Kissel. 2022. FABRIC Network Service Model. In *2022 IFIP Networking Conference (IFIP Networking)*. 1–6. doi:10.23919/IFIPNetworking55013.2022.9829810

[62] Khawar Shehzad, Ajay Kumar, Matthew Schutz, Chase Webb, Polycarp Nalela, Manas Jyoti Das, and Praveen Rao. 2024. A Scalable Tool for Democratizing Variant Calling on Human Genomes Using Commodity Clusters. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (Boise, ID, USA) *(CIKM '24)*. Association for Computing Machinery, New York, NY, USA, 5275–5279. doi:10.1145/3627673.3679221

[63] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: network processing as a cloud service. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Aug. 2012), 13–24. doi:10.1145/2377677.2377680

[64] Nishanth Shyamkumar, Hyunsuk Bang, Bjoern Sagstad, Prajwal Somendyapanahalli Venkateshmurthy, Sean Cummings, and Nik Sultana. 2025. Patchwork: A Traffic Capture and Analysis Platform for Network Experiments on a Federated Testbed. In *Proceedings of the 2025 ACM Internet Measurement Conference, IMC 2025, Madison, WI, USA, 28-31 October 2025*, Paul Barford, Aruna Balasubramanian, and Alberto Dainotti (Eds.). ACM, 677–691. doi:10.1145/3730567.3764462

[65] Nishanth Shyamkumar, Sean Cummings, Hyunsuk Bang, and Nik Sultana. 2024. Towards Testbed-Wide Traffic Profiling for FABRIC. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–6. doi:10.1109/INFOCOMWKSHPS61880.2024.10620844

[66] Seyedali Tabaeiaghdaei, Filippo Costa, Jonghoon Kwon, Patrick Bamert, Yih-Chun Hu, and Adrian Perrig. 2024. Debuglet: Programmable and Verifiable Inter-Domain Network Telemetry. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. 255–265. doi:10.1109/ICDCS60910.2024.00032

[67] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with PathDump. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 233–248. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/tammana

[68] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 453–456. https://www.usenix.org/conference/nsdi18/presentation/tammana

[69] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 599–614. https://www.usenix.org/conference/nsdi19/presentation/tan

[70] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. 2018. Stroboscope: Declarative Network Monitoring on a Budget. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 467–482. https://www.usenix.org/conference/nsdi18/presentation/tilmans

[71] Sankalpa Timilsina, Davide Pesavento, Junxiao Shi, Susmit Shannigrahi, and Lotfi Benmohamed. 2023. Capture and Analysis of Traffic Traces on a Wide-Area NDN Testbed. In *Proceedings of the 10th ACM Conference on Information-Centric Networking* (Reykjavik, Iceland) *(ACM ICN '23)*. Association for Computing Machinery, New York, NY, USA, 101–108. doi:10.1145/3623565.3623707

[72] Kashi Venkatesh Vishwanath and Amin Vahdat. 2008. Evaluating Distributed Systems: Does Background Traffic Matter?. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, USA, 227–240.

[73] Peter Willis, Nirmala Shenoy, Yin Pan, and Bill Stackpole. 2024. Investigating Data Center Network Protocols. In *Proceedings of the 2024 Applied Networking Research Workshop* (Vancouver, AA, Canada) *(ANRW '24)*. Association for Computing Machinery, New York, NY, USA, 91–93. doi:10.1145/3673422.3674897

[74] Peter Willis, Nirmala Shenoy, Yin Pan, Bill Stackpole, and John Hamilton. 2024. New Techniques to Route in Folded-Clos Topology Data Center Networks. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 819–828. doi:10.1109/SCW63240.2024.00116

[75] Alexander Wolosewicz, Prajwal Somendyapanahalli Venkateshmurthy, and Nik Sultana. 2025. Experience Report: Using the FABRIC Testbed to teach a Graduate Computer Networking course. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1* (Pittsburgh, PA, USA) *(SIGCSE TS 2025)*. Association for Computing Machinery, New York, NY, USA, 1246–1252. doi:10.1145/3641554.3701923

[76] Alexander Wolosewicz, Vinod Yegneswaran, Ashish Gehani, and Nik Sultana. 2025. Network Replay and Consistency Across Testbeds. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3731599.3767447

[77] Alexander Wolosewicz, Vinod Yegneswaran, Ashish Gehani, and Nik Sultana. 2025. User Study of a Network Debugger on FABRIC. In *2025 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*. 501–509. doi:10.1109/ACSACW69556.2025.00059

[78] Wenfei Wu, Guohui Wang, Aditya Akella, and Anees Shaikh. 2013. Virtual network diagnosis as a service. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 9, 15 pages. doi:10.1145/2523616.2523621

[79] Zichen Xu, Ziye Lu, and Zuqing Zhu. 2024. Information-Sensitive In-Band Network Telemetry in P4-Based Programmable Data Plane. *IEEE/ACM Trans. Netw.* 32, 6 (Aug. 2024), 5081–5096. doi:10.1109/TNET.2024.3448244

[80] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 207–220. https://www.usenix.org/conference/nsdi19/presentation/yu

[81] Lior Zeno, Ang Chen, and Mark Silberstein. 2024. In-Network Address Caching for Virtual Networks. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 735–749. doi:10.1145/3651890.3672213

[82] Delong Zhang, Chong Ye, and Fei He. 2024. P4Inv: Inferring Packet Invariants for Verification of Stateful P4 Programs. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*. 2129–2138. doi:10.1109/INFOCOM52122.2024.10621366

[83] Yiming Zhang, Tao Wan, Yaru Yang, Haixin Duan, Yichen Wang, Jianjun Chen, Zixiang Wei, and Xiang Li. 2025. Invade the Walled Garden: Evaluating GTP Security in Cellular Networks. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1159–1177. doi:10.1109/SP61157.2025.00028

[84] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. 2012. Distributed time-aware provenance. *Proc. VLDB Endow.* 6, 2 (Dec. 2012), 49–60. doi:10.14778/2535568.2448939

[85] Yu Zhou, Jun Bi, Cheng Zhang, Bingyang Liu, Zhaogeng Li, Yangyang Wang, and Mingli Yu. 2019. P4DB: On-the-Fly Debugging for Programmable Data Planes. *IEEE/ACM Transactions on Networking* 27, 4 (2019), 1714–1727. doi:10.1109/TNET.2019.2927110

[86] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 479–491. doi:10.1145/2785956.2787483

[87] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and Mitigating Packet Corruption in Data Center Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 362–375. doi:10.1145/3098822.3098849