Debuglet: Programmable and Verifiable Inter-domain Network Telemetry

Seyedali Tabaeiaghdaei
ETH Zürich

Filippo Costa
Switch

Jonghoon Kwon
ETH Zürich

Patrick Bamert
Zürcher Kantonalbank

Yih-Chun Hu
University of Illinois Urbana-Champaign

Adrian Perrig
ETH Zürich

***Abstract*—On today's Internet, end-user debugging is largely limited to simple tools such as `ping` and `traceroute`, supplemented by purpose-built services such as bandwidth measurement, and website uptime monitors. Unfortunately, these tools do not provide sufficient data to isolate specific network faults, nor do they give the user results that can be validated by external entities. Furthermore, since networks disparately treat measurement packets, as our empirical results confirm, measurement packets need to be indistinguishable from data packets. In this paper, we argue for a distributed network debugging infrastructure and describe Debuglet, a *deployable* and *incentivized* architecture that allows inter-domain network debugging using real data packets and user-defined code, which facilitates accurate and flexible measurements of the network performance experienced by data packets. We implement the *Debuglet* system, and demonstrate its feasibility by deploying it on a network testbed, evaluating its measurement accuracy, and analyzing its deployment costs.**

***Index Terms*—network debugging, distributed system, remote code execution, blockchain**

I. Introduction

Internet applications have become mission-critical for many businesses. As teleconferencing increasingly replaces business travel, and as collaboration between multiple worksites increasingly relies on Internet applications, the Internet has become critical infrastructure for everyday life. When encountering Internet outages or reduced performance, users and technicians alike want to know: which link or system is responsible? Who can help to resolve the issue?

On today's Internet, end-user debugging is largely limited to rudimentary tools such as `ping` and `traceroute`, supplemented by purpose-built services such as bandwidth measurement, website failure detection, and uptime monitors. Unfortunately, these tools suffer from three major shortcomings. First, they are either end-to-end [11], [14], [17], complicating fault localization, or they are not based on real data traffic [18], [32], [34], allowing networks to preferentially treat measurement traffic over regular traffic, such that their measurements do not accurately reflect the fate of real data traffic. Second, techniques that require participation by forwarding routers [27], [28], such as `traceroute`, do not always get the desired support, or may get replies from non-public IP addresses, making fault localization difficult. Finally, the results of a measurement are not verifiable by external entities.

In this paper, we propose a distributed network debugging infrastructure, *Debuglet*, in which Autonomous Systems (ASes) deploy services intended for network-measurement applications. The *Debuglet* executor, distributed at the edge of ASes, provides a policy-constrained remote code execution environment. By distributing and executing applications tailored to the intended network measurement to the executors, endpoints perform real-world data-plane operations at different vantage points, allowing them to localize the origins of network failures with high accuracy. Endpoints pay to run their network debugging applications in these environments using some form of micro-payments, using traditional currencies or cryptocurrencies brokered through their Internet Service Providers (ISP). The output of these applications can then be certified by the deploying AS, allowing third parties to verify the measurement results.

Our contributions in this paper are:

- Demonstrating that real-world network performance depends on the type of network traffic being sent, thus to achieve accurate measurement the packet type of the probes should match the packet type of the traffic whose performance is being debugged,
- Designing the distributed *Debuglet* architecture, allowing programmable and verifiable network measurements in inter-domain networks, enabling fine-grained network fault localization, and
- Introducing a marketplace model for purchasing measurement service from remote ISPs, incentivizing the deployment of the architecture and paving the path to form an ecosystem for Internet debugging.

II. Motivation

The inadequacy of today's ICMP-based network measurement tools to reproduce and accurately locate network faults in federated networks motivates us to design a new system addressing these problems. This inadequacy stems from the differential forwarding treatment by routers, i.e, routers' forwarding decisions do not only rely on the destination of

Work has been completed during his presence at ETH Zürich

TABLE I: RTT and drop rate between the specified location and London. Each measurement consists of 86400 packets, one per second over a day. Data is expressed in milliseconds. Loss rate is given in per-thousandths (‰).

Location	UDP		TCP		ICMP		Raw IP	
	mean	std	mean	std	mean	std	mean	std
Bangalore	146.01	7.01	158.05	5.27	145.44	3.89	151.44	2.87
	0.23 ‰ lost		1.72 ‰ lost		0.57 ‰ lost		0.41 ‰ lost	
Frankfurt	14.75	1.78	14.72	1.22	11.95	0.51	15.36	0.55
	0.00 ‰ lost		1.09 ‰ lost		0.01 ‰ lost		0.00 ‰ lost	
New York	73.94	6.64	71.58	6.12	76.08	3.98	76.47	4.02
	5.59 ‰ lost		16.19 ‰ lost		0.24 ‰ lost		0.27 ‰ lost	
San Francisco	134.79	1.00	134.42	0.70	134.62	0.66	135.09	1.71
	0.00 ‰ lost		1.56 ‰ lost		0.02 ‰ lost		0.03 ‰ lost	
Singapore	176.14	10.04	176.95	4.33	181.74	3.00	178.98	4.61
	0.09 ‰ lost		1.74 ‰ lost		0.06 ‰ lost		0.03 ‰ lost	
Sydney	274.01	7.79	278.60	5.19	277.99	5.15	278.44	5.18
	0.50 ‰ lost		1.09 ‰ lost		0.96 ‰ lost		1.01 ‰ lost	

packets but also on type, size, and values in the header fields. We expose this problem through real-world measurements. We design and carry out experiments demonstrating that the protocol of data packets can affect the network performance.

Protocols. We conduct experiments on the following protocols:

- UDP packets,
- TCP packets without any special flag, with random sequence numbers,
- ICMP echo requests,
- Custom IP packets with an unassigned protocol number (201)

Performance Metrics. We consider two network performance metrics: round-trip time (RTT) and drop rate.

Experiment Setup. We set up identical virtual machines (VM) hosted by Digital Ocean in seven locations around the globe: Bangalore, Frankfurt, London, New York, San Francisco, Singapore, and Sydney. We measure RTT and drop rate of considered protocols between the VM in London and other VMs in the other six locations. To achieve this, we deploy a client Go application in those six locations, each of which sends probes to the VM in London in a round-robin manner, rotating between the considered protocols with a period of one second and sending one probe packet of that protocol's type. The server Go application in London replies to these probes, and the client collects experienced RTTs and drop rates based on the received replies. To ensure that any performance differences are only due to the difference in protocols, the applications ensure that the total length of each packet, from the beginning of the layer-3 header, is the same for all four protocols. We conduct measurements during 24 hours.

Based on our investigation using traceroute, traffic between these datacenters traverses the public Internet as the cloud provider does not own a global network connecting all its data centers; thus, our results represent the performance of inter-domain networks.

Empirical Results. Table I summarizes the measurement results. The results suggest that the network's forwarding

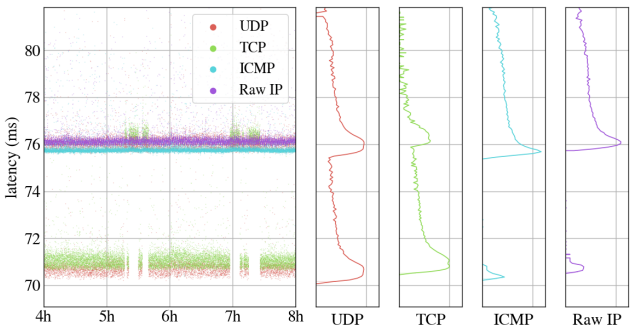


Fig. 1: New York - London RTT, over 4 hours. The leftmost plot shows latency variations over time, while the four vertical line plots represent the density function of latency distribution for each protocol type, logarithmic scale.

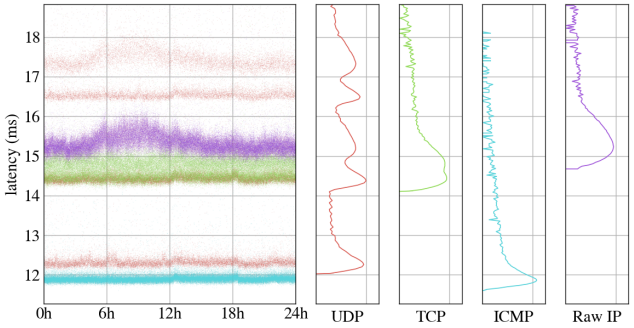


Fig. 2: Frankfurt - London RTT, 24 hours.

behavior differs for different transport protocols. ICMP's and raw IP's RTT demonstrate greater stability compared to UDP and TCP. For ICMP packets, this phenomenon could be explained by routers treating them specially, as they are mainly used for network debugging.

Conversely, UDP demonstrates the highest variation in measurements. Because it is generally assumed that applications using UDP can tolerate some packet re-ordering, the variation may be caused by route diversity, from routers load-balancing UDP on a more fine-grained basis than per-flow or per-flowlet, as is typical for TCP flows.

TCP experiences the highest drop rate. One explanation is that routers deprioritize TCP packets on congested links to make senders reduce their transmission rate, since UDP may not respond to loss as significantly.

Figure 1 shows a 4-hour window of the 24-hour RTT measurement between London and New York. UDP and TCP consistently experience lower RTT than ICMP and raw IP. However, in some periods, a sudden increase of about 5 ms is observable, signaling a change in forwarding behavior, e.g., route changes. Figure 2 plots 24-hour results between London and Frankfurt. UDP results show four clearly visible clusters, which we hypothesize as representing four different

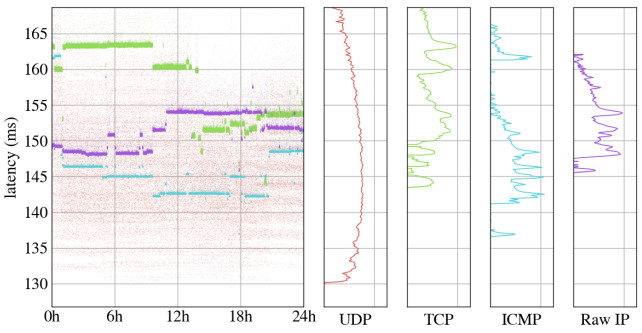


Fig. 3: Bangalore - London RTT, 24 hours.

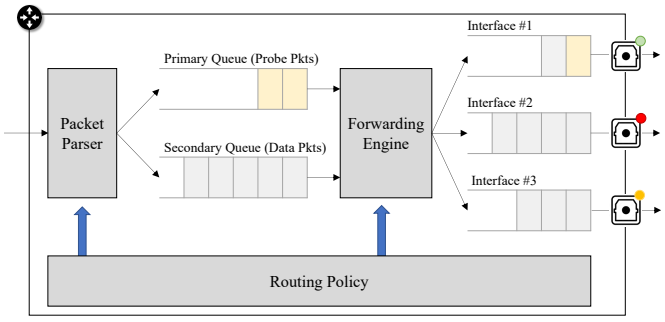


Fig. 4: Forwarding devices may handle packets differently based on their type, resulting in different forwarding decisions.

routes on which UDP traffic is forwarded. Another interesting observation is that for several hours the RTT of UDP and raw IP packets show a noticeable increase that is not reflected in ICMP nor TCP. Figure 3 shows that UDP's RTT between Bangalore and London is distributed over a 30 ms range, almost randomly. Other measures, even though being consistent for relatively short periods of time, vary several times during a day, without any clear correlation.

These results indicate that measurement packets and data packets of different types can receive differing treatment by networks. This differing treatment means that performance issues for a TCP application are best diagnosed using TCP packets, rather than using ICMP or UDP packets. Although this conclusion is driven from ping measurements, it also holds for traceroute, which also uses ICMP, making traceroute packets distinguishable from data packets. However, it is possible to perform traceroute with TCP or UDP packets, i.e., encapsulated in IP packets with increasing TTL. However, regardless of the protocol used, there are two important limitations with traceroute: (1) responding with ICMP TTL exceeded message is disabled or rate-limited on many routers, and (2) routers responding with ICMP TTL exceeded message process such messages on the slow path, while data packets are processed in the fast path, resulting in different performance experienced by data and measurement packets.

III. Design Principles

The *Debuglet* architecture aims to enable network administrators to locate network failures in an inter-domain network. Inspired by the motivation presented in Section II, we lay down the principles we pursue in the design of *Debuglet*.

Segment-by-Segment Network Measurements. To accurately locate network failures, it is necessary to perform debugging per network segment. For instance, if a network failure occurs on an inter-domain link, the optimal source and destination locations for measurement packets for network debugging would be the edges of consecutive ASes. Through this approach, network inspection is performed individually for specific inter-domain or intra-AS links, isolating each measurement to identify network failures independently.

Reproducibility. As demonstrated earlier, the application data packet type affects the forwarding decision of on-path routers. For example, as shown in Figure 4, probing packets may be assigned to a priority queue over regular data packets and can be transmitted through more stable links. Thus, forwarding delays can vary based on the protocol, size, and header field values. Additionally, changes in the destination address—and sometimes also the source address—can influence the decision of forwarding interfaces. Therefore, to ensure a consistent network experience, replicating the same type of application data packets and transmission through the same path (i.e., the same interface port) is required.

Enabling Unidirectional Measurements. It is necessary to allow unidirectional fault localization on the Internet because of two reasons: (1) Internet paths may not be symmetric, and (2) load distribution on different directions of each link on a path can be different, resulting in different performance in forward and backward directions, e.g., one link can be congested in the forward direction and uncongested on the backward direction, resulting in more queuing delay and drop rate on the forward direction. To distinguish faults on the forward path from the ones on the backward path, *Debuglet* should provide the ability to measure the performance of each direction of the path.

Transferability and Verifiability of Measurement Results. It might be beneficial for networks to reuse the measurement results so that no other domains need to perform similar measurements to locate the same network fault, scaling down the total number of probes sent and ultimately saving resources and costs for network debugging. Therefore, the *Debuglet* architecture should support publishing the results of measurements if desired. The main challenge in enabling this capability is the verifiability of measurement results: how can a domain trust that the results published by other end domains provide a realistic picture of the measured segment in the network? In other words, how can the system provide a guarantee that no domain can deliberately provide an ungrounded worse or better picture of an ISP's performance by publishing fake measurement results or by cherry-picking

the worst or best real measurements and just publishing those results? Such behavior can be motivated by domains trying to damage the reputation of other domains, or claiming a refund in case of a service level agreement.

A. Requirements

We describe the prerequisites we need to follow the principles of designing *Debuglet*.

Remote Code Execution (RCE). To be able to reproduce data packets close to faults in an inter-domain network, where faults can occur out of end domain's administration, it is necessary that end domains execute code at remote domains near network faults. Furthermore, RCE is necessary for enabling unidirectional network measurements as it facilitates the programmability of debugging applications, providing control over both sender and receiver sides of measurements.

Path Awareness. Control over forwarding paths is one of the design principles of *Debuglet* which can only be achieved through path-aware networking [30]. Path-aware network architectures provide endpoints with information about network paths to any destination and allow them to select paths based on their needs. Therefore, to achieve control over forwarding paths, *Debuglet* relies on path-aware architectures such as SCION [12] and segment routing [13]. The main benefit of leveraging these two specific architectures is that they provide path control at a fine granularity – in the case of SCION the ingress and the egress links of each AS on the path to the destination.

IV. The Debuglet Architecture

A. Overview

The *Debuglet* architecture consists of a *data plane* that enables running custom network measurements at precise locations in a public inter-domain network and a *control plane* that enables the scheduling of measurements and verifiable publishing of the results. Every measurement performed using *Debuglet* requires two distinct executors at two points of a path and two pieces of application: the *Debuglet* client that starts measurements and *Debuglet* server that receives measurement packets, which are deployed by the initiator as shown Figure 5.

Network measurement in *Debuglet* architecture is a five-step process:

(1) Upon experiencing network anomalies, network endpoints request debugging from their ISP.
(2) The ISP generates *Debuglet* applications (or just *Debuglet*s) based on the reported issue, and requests the ASes with executor instances close to the points of interest on the forwarding path to run the *Debuglet*s.
(3) The ASes who accept the request deploy the *Debuglet* on the agreed executor instances.
(4) Executors run the *Debuglet*s, and collect measurements.
(5) Executors report the results back to the requesting AS.

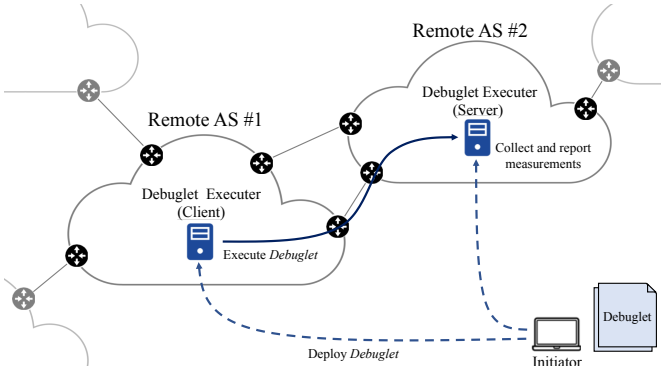


Fig. 5: High-level overview of *Debuglet*.

B. Data Plane

The data plane of the *Debuglet* architecture has two building blocks: (1) *Debuglet*s, which are pieces of application that produce and send data packets to measure network performance at selected locations, and (2) *Debuglet* executors, which are machines deployed at ASes to execute *Debuglet*s on behalf of end domains, collect the results, and submit them to the control plane.

***Debuglet* Executor.** In the *Debuglet*'s distributed network-debugging infrastructure, *Debuglet* executors are small-scale cloud services intended for network measurement applications. To allow the safe execution of unverified code from other ASes, the received application must be executed in a sandboxed environment that provides memory safety, and finishes in a bounded number of instructions. Although virtual machines (VMs) can achieve these properties, significant setup time and effort of VMs inhibit agile network debugging. To overcome this, we build our architecture using WebAssembly (WA) [9], [16] to run *Debuglet*s[1].

By default, a WA application operates within an isolated environment and lacks direct communication capabilities with the external world, except for input parameters and return values. The executor therefore provides these bytecodes with (1) buffers from/to which WA applications can read or write raw application data, and (2) an API through which WA applications can request sending or receiving packets with a variety of transport, network and link layer protocols (if applicable). Therefore, WA applications have the freedom to choose the application layer protocol, content, and data rate. To facilitate network transmissions, dedicated memory regions are designated to serve as buffers. These buffers are assigned to specific namespaces, such as `udp_send_buffer` or `tcp_receive_buffer`, making them easily accessible as global variables. Another buffer is also allocated to store

[1]Some alternatives are also available, e.g., *Extended Berkely Packet Filter* (eBPF) [1], enabling the execution of customized code in the kernel, or uBPF [7], a user-space implementation of the same concept, or the Dandelion system [19].

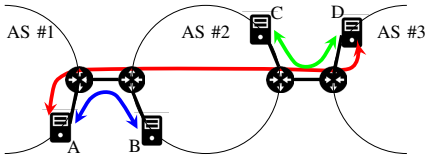


Fig. 6: Executors are co-located with all border routers of ASes, achieving network debugging in an inter-domain link granularity.

the result of the execution. The `output` buffer serves as the location where a WA bytecode stores the results of network measurements. The output can then be certified by the deploying AS of the executor, allowing third parties to verify the measurement results.

Debuglets. A *Debuglet* is compiled as WA bytecode that provides the `run_debuglet` function as the entry point, serving as a *main* function in a typical program. The bytecode can perform arbitrary computations until its execution concludes, after which it must return a result.

In order to successfully execute a *Debuglet*, it is accompanied by a manifest upon submission of the application. The manifest is evaluated by the remote AS prior to execution. This manifest contains the following information:

- Resource requirements, including CPU time, execution duration, peak memory usage, and the number of packets sent and received,
- The list of addresses desired to be contacted, and
- Specific capabilities required for execution.

Location of Executors. The location of executors in each AS is a crucial part of the design of *Debuglet* architecture. Network operators need to know where they should locate *Debuglet* executors for two reasons: (1) resource provisioning, and (2) ensuring that *Debuglet* measurements do not disclose secret internal network topology. Furthermore, the locations of executors in each AS determine the achievable accuracy of fault localization. Thus, identifying the location of executors requires the identification of the desired fault-localization accuracy.

We consider co-locating executors with border routers—a router connecting an AS to a neighboring AS—meaning that an AS deploys executors preferentially at its borders (other alternative deployment scenarios are discussed in Section VI-G). This is because in inter-domain fault localization, end domains are interested in finding which ASes or inter-domain links on a path are responsible for a network failure.

Figure 6 illustrates how such a deployment model enables distinguishing faults within an AS and on inter-domain links. Consider if an end domain has suspected that the segment between the egress border router of AS #1 and the ingress border router of AS #3 is responsible for the performance degradation, suggesting that the fault can occur (1) between AS #1 and #2, (2) within AS #2, or (3) between AS #2 and #3. Assuming that all ASes have deployed one executor co-located with each of their border routers, a remote domain can validate the mentioned three hypotheses by pursuing the following procedure:

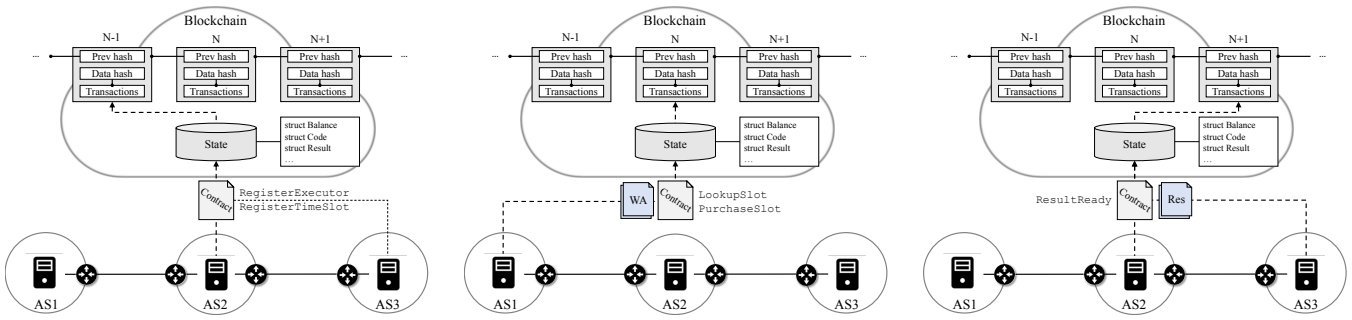
(1) Deploying a pair of *Debuglet*s at executors A and D that run measurements to validate that the segment (red) between the egress border router of AS #1 and the ingress border router of AS #3 is in fact faulty, and measuring the performance on this segment,
(2) Deploying a pair of *Debuglet*s at executors A and B that run measurements to evaluate the performance of the link between ASes #1 and #2 (blue),
(3) Deploying a pair of *Debuglet*s at executors C and D that run measurements to evaluate the performance of the link between ASes #2 and #3 (green),
(4) Deriving the performance of the part of path *within* AS #2 based on the measurements collected in the last three steps for the whole segment and each of the inter-domain links connecting to AS #2.

Note that we intentionally do not measure the performance of AS #2 by deploying a pair of *Debuglet*s at executors B and C directly inside the target AS, because in that case the traffic would be *intra-domain* traffic within AS #2, which might be treated differently from the original *inter-domain* traffic coming from AS #1 and going to AS #3, hindering the accurate reproduction of the original forwarding behavior.

C. Control Plane

The responsibility of the *Debuglet* control plane is to (1) provide a public list of *Debuglet* executors, (2) schedule *Debuglet* executions such that it enables the synchronized execution of two applications in two independent ISPs in a Internet-scale environment where only a limited number of requests can be accommodated at each executor due to finite resources, and (3) allow ISPs to publish the results of *Debuglet* executions publicly in a verifiable manner, so that other entities cannot alter or create fake results. Furthermore, the control plane *can* provide an interface between the initiators of *Debuglet* applications and the executors, through which they can exchange application and results. In this section, we present a centralized design, while we sketch a possible decentralized alternative design in Section VI-A.

Blockchain-based Marketplace. As an instantiation of such a control plane, we propose a marketplace model for *Debuglet* measurements through which domains trade time slots for measurements, exchange *Debuglet* applications, and *can* publish measurement results. This model allows for simple scheduling of measurements without requiring a complex algorithm: an initiator can purchase two measurement slots for the same timespan at the two executors of its interest, and publish the bytecode for each purchased slot. The use of blockchains for implementing such a marketplace ensures verifiable publishing of results: All transaction history for the results published by the *executors* is traced in the blockchain, i.e., none of the *Debuglet* participants can modify the results without others noticing. Using blockchains also allows for integrating a payment method with the marketplace.



(a) ASes register their executors along with available time slots to the blockchain network through a global smart contract.

(b) An initiator looks up available slots and, upon finding them, submits *Debuglet* applications and the tokens to be paid.

(c) After running measurements, executors report results and retrieve the payment from the smart contract.

Fig. 7: Basic *Debuglet* execution model with a smart contract.

Integrating with Smart Contract. We design such a marketplace as a smart contract. Thus, all executors and initiators join a blockchain as nodes.

The smart contract defines application and results as objects: An application object contains an object ID, a string containing the actual WA bytecode, and the tokens that the initiator embeds to be transferred to the executor upon completion of the *Debuglet* execution. A result object contains its object ID and a string containing the actual results.

The smart contract defines the following data structures as its state to accomplish its task as a measurement marketplace:

- ExecutorAddressMap: A map indexed by the concatenation of the AS number and inter-domain interface[2] identifier ($< AS, intf >$) associated with executors to their node's address. The map enables deploying *Debuglet* applications to the intended executors. If multiple physical executors exist for a $< AS, intf >$, the smart contract does not distinguish between them. Instead, it is the task of their administrating domain to coordinate them.
- ExecutionSlotsMap: A map indexed by $< AS, intf >$ to the sorted list of available non-overlapping time slots for the corresponding executor. Similar to the cloud's IaaS model, ASes provide the executors' time slots from which initiators can choose. Each time slot is associated with the following five tuples: (1) the number of available CPU cores, (2) available memory capacity, (3) assigned network bandwidth, (4) start and end time in Unix UTC time, and (5) the price for the slot.
- ApplicationsMap: A map indexed by the concatenation of the *Debuglet* client's AS number and interface ID, the *Debuglet* server's AS number and interface ID, and the time slot ($< AS_c, intf_c, AS_s, intf_s, t_{start}, t_{end} >$) to the list of vector of *Debuglet* applications associated with the tuple, pointing to the applications stored in the blockchain.
- ResultsMap: A map indexed by the object IDs of *Debuglet* applications to the results struct.

[2]An inter-domain interface specifies either end of an inter-domain link connecting two neighboring ASes.

Figure 7 shows the life cycle of *Debuglet* for a measurement. The smart contract defines the following functions that are called by different parties.

Bootstrapping. An executor calls `RegisterExecutor()` to register itself at the smart contract. It provides its AS number and interface ID as arguments to the function. This function adds the mapping between the AS and interface ID of the caller and its address to ExecutorAddressMap.

Later, via `RegisterTimeSlot()`, executors register their available time slots by provide their AS number, interface ID, and the list of time slots (a time slot is defined by a 5-tuple explained in the ExecutionSlotsMap in this section). The function call first checks that the provided AS number and interface ID are, in fact, associated with the calling executor based on ExecutorAddressMap and updates the ExecutionSlotsMap.

Initiating Network Measurements. An initiator calls `LookupSlot()` to first look for available slots for its desired measurements. It provides as an argument the AS number and interface ID of the executor running the client application and the same information for the server side, the required number of cores, amount of memory, and bandwidth for both client and server applications. The function checks by looking up the ExecutionSlotsMap, when the first available time slot that both to-be-involved executors can accommodate the measurement would be, and how many execution slots need to be purchased at each executor. The function returns the price that needs to be paid and the first possible time slot to the initiator.

To buy a slot, the initiator calls `PurchaseSlot()`. The initiator provides the following information as the arguments

to the function: the client's and server's AS and interface IDs, the time slot's 5-tuple, the number of execution slots it desires to purchase from each of the executors, the string representations of the WA bytecodes of the client and server, and the cryptocurrency tokens required for running each of the client and the server application on each executor. The function first verifies that the embedded tokens suffice for the specified execution slots. Then, it creates two application objects, embeds the tokes in them and adds them to ApplicationsMap, and emits events to notify the executors. The executors, which must have subscribed to the event with arguments containing their AS number and interface ID, retrieve the applications and schedule them for the requested time slot.

Reporting Results. An executor calls `ResultReady()` after it has executed the *Debuglet*. The executor provides the object ID of the executed *Debuglet* and the string representation of the results as the arguments. The function transfers the embedded tokens in the application object to the executor's address. It also creates a result object, in which it embeds the result string, and inserts it into the ResultsMap. Finally, it emits an event to notify the initiator, which should have subscribed to notifications with the object ID of its application, of the readiness of the result.

`LookupResult()` allows any node to search for the result of any measurement. The function looks up ResultsMap for measurements and returns the list of object IDs of results together with the time they were executed and the object ID of their associated applications. Note that an initiator may want to keep the results private by encrypting the results in the client and server applications using a cryptographic key embedded in the applications. In that case, the results are not readable by third parties.

V. Implementation and Evaluation

A. Implementation

We implement a proof-of-concept of all components of the *Debuglet* architecture, i.e., executors, sample *Debuglet* server and client application, and the smart contract to control all *Debuglet* procedures. We implement each executor as a Go application using Wasmer [8] WA [9] runtime to run *Debuglet* WA bytecodes. We write *Debuglet* client and servers as Rust applications and compile them to WA bytecodes. We write the smart contract in Move language [2] and deploy it on a local instance of the Sui blockchain, a modern blockchain capable of thousands of transactions per second, sub-second finality, and low transaction cost. We deploy our implementation in the SCIONLab [4], [20] testbed, a global testbed for the SCION Internet architecture.

B. Evaluation Results

We evaluate the feasibility and practicality of the *Debuglet* architecture by evaluating three important aspects of its design, i.e., (1) the impact on the accuracy of running WA bytecode for measurements, (2) the delay-to-measurement introduced by the control plane design, and (3) the cost of using blockchains as the control plane of the system.

The Impact of Running WA on Measurement Accuracy. To this end, we developed:

- Two native Go applications, one acting as a UDP client, sending packets at a steady rate, and the other as a UDP echo server, and
- Two WA Debuglets with exactly the same functionalities as the native Go applications mentioned above.

We then performed four experiments to quantify the delay added by using WA.

(1) Debuglet to Debuglet (D2D), where both the client and server are Debuglets,
(2) Application to Debuglet (A2D), where the client is a native application, and the server is a Debuglet.
(3) Debuglet to Application (D2A), where the client is a Debuglet, and the server is a native application.
(4) Application to Application (A2A), where both client and server are native Go applications.

We ran these experiments simultaneously for one day, between virtual machines located in London and New York, sending one packet per second for each experiment. In total, we collected $4 \cdot 86400$ data points.

Figure 8 illustrates the results of these measurements. According to this figure, D2D measurements have a mean latency of 75.12 ms while A2A measurements have a mean latency of 74.81 ms. Therefore, WA execution adds a delay of approximately 300 microseconds, which is attributable to the additional operations for switching between Go and WA. D2A and A2D measurements fall in between, with latencies of 75.01 ms and 74.88 ms, respectively. Importantly, the figure suggests that WA execution does introduce some noise to the measurements, but an almost constant delay, which can be offset from the results if the execution environment is known, thus enabling extraction of the ground truth measurement results.

In terms of packet loss, our measurements show negligible difference between measurements: D2D, 1.68% of the packets are lost, while A2D experiences 1.38% loss, D2A 1.66%, and A2A 1.71%.

The Delay-to-measurement. The delay-to-measurement is an important aspect of the design and implementation as it is a determining factor in how fast the system can locate faults. Importantly, if faults are short and transient, long delay-to-measurement can cause missing the fault. Delay-to-measurement of *Debuglet* comprises three delays: (1) the blockchain operations delay, (2) the waiting time until the scheduled time slot, and (3) the WA environment's setup time. Regarding the first delay, modern blockchains like Sui achieve very high transaction throughput (more than 100,000 per second) and very low finality latency (less than half a second) per transaction [5]. In our implementation, two transactions are on the critical path of running a measurement, i.e., calling `LookupSlot()` and `PurchaseSlot()`, which are negligible. The second delay, which contributes to

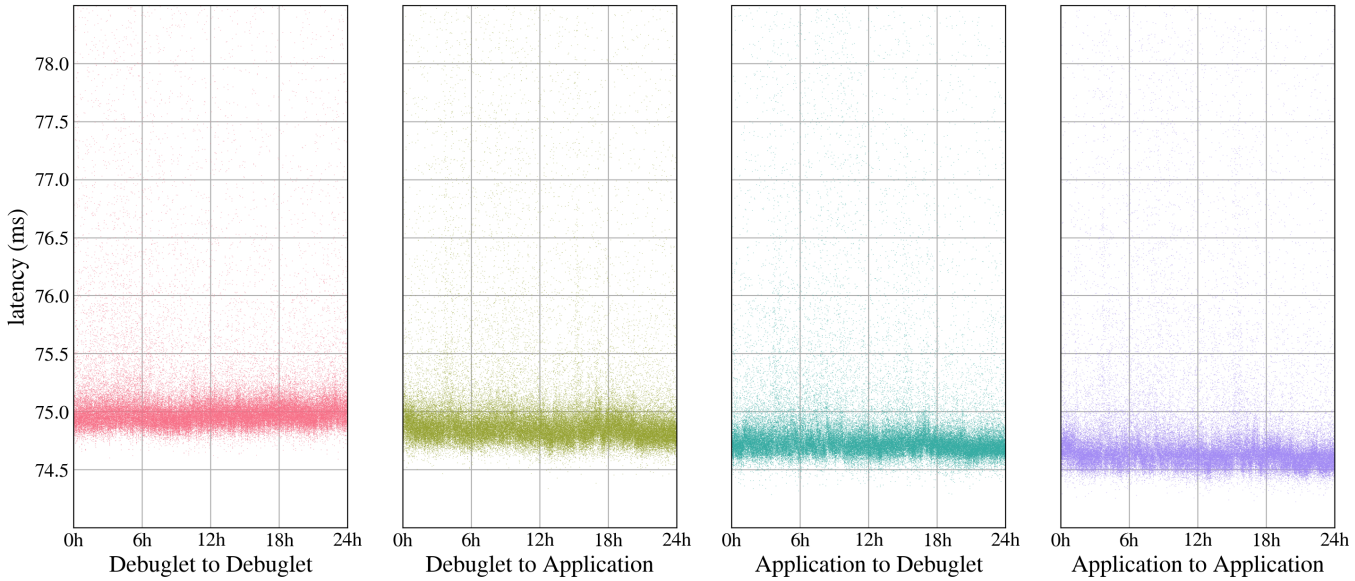


Fig. 8: Latency measurements using different Debuglet and application combinations. Each dot represents a measurement.

the majority of delay-to-measurement, can only be evaluated in an operation as it depends on the demand for measurements, the available resources for executors in ISPs, and how often faults occur in the network. Therefore, despite its determining effect, this delay is neither a matter of design nor the implementation of *Debuglet*. However, because of the marketplace-based IaaS model we propose in this work for executing *Debuglet* application, we expect ISPs to dedicate resources to executors proportionally to the demand for running application, resulting in enough resources available for running measurements, and thus, limited wait time for measurements. We evaluate the third type of delay, i.e., execution environment setup time, for different WA bytecodes, and observe an almost constant setup time of around 10 ms across all executions. To conclude, the design and implementation of *Debuglet* data and control planes allow for a *sub-second* reaction to an experienced fault.

Blockchain Costs. Blockchains charge nodes for transactions they perform and the data they store on the blockchain. This means that deploying the *Debuglet* infrastructure introduces additional costs both to initiators and executors. We evaluate these costs for submitting and storing a *Debuglet* application using the Sui blockchain. The cost for results has a similar pattern, i.e., a constant cost and a storage cost proportional to required storage. Table II shows the price (in SUI) for submitting and storing applications with different sizes on the Sui blockchain (the main net). The cost can be significantly lowered by storing applications or results off-chain and only storing a link to the stored data and a hash of data on the chain, so that the data can be verified against the on-chain hash. Consequently, keeping only the hashes of the data and code on-chain, the Sui transaction fees amount to about 1 cent, which is sufficiently low for most use cases. If desired, a reduction in transaction costs can be achieved by using optimizations to perform micropayments on blockchains [24].

TABLE II: Cost of submitting *Debuglet* application to the Sui blockchain in SUI for different application sizes. Each SUI is equivalent to $0.94 as of May 14th, 2024 [6]. The storage rebate is refunded after the stored data is freed up.

application size	Total Cost (in SUI)	Storage Rebate (in SUI)
0 B	0.01369	0.00430
100 B	0.01585	0.00632
1 kB	0.03527	0.02456
5 kB	0.12160	0.10562
10 kB	0.22953	0.20696

VI. Discussion

A. Alternative Channel for Discovering Executors

A logically centralized control plane for the *Debuglet* system, such as the marketplace model proposed in Section IV-C, integrates executor discovery with code and result exchange, fine-grained scheduling of executions, verifiable result publication, and payment for the provided service. However, such a model entails a single point of failure, i.e., the marketplace, and assumes that all ISPs agree on what entity should operate the marketplace. To address these issues, we present an alternative decentralized approach to discover executors without relying on a centralized controller. In the decentralized approach, ISPs advertise the address and location of their executors as route metadata encoded in routing messages they originate. Thus, other domains learn about the executors through the inter-domain routing

protocol, which is a distributed protocol executed by all domains. An initiator of *Debuglet* applications uses these addresses to directly contact the executors. They need to bilaterally negotiate the scheduling and the price of the execution and the method of payment. The initiator sends the applications directly to the executors, and upon the end of the execution, the executors return the results directly back to the initiator. As the results are not publicized by the trusted marketplace, they would not be publicly verifiable. Both the centralized and the decentralized approach can coexist, and we anticipate that both would be used catering to different application scenarios.

B. Incremental Deployment

Though full deployment provides the best picture of how each AS impacts path performance of real network packets, partial deployments allow direct measurement between any two deploying ASes, and between a deploying AS and either endpoint. For applications such as service-level agreement (SLA) enforcement, a small deployment of *Debuglet* services at the ISP, or, ideally, at neighboring ASes, will allow for verifiable measurements of the performance between a subscriber (as measured at the Customer Premises Equipment, for example) and their ISP and its providers. Further deployment of *Debuglet* services would help ISPs prove their innocence.

An AS that knows it contributes to poor performance may be disincentivized from deploying *Debuglet* services as it tries to conceal its contribution to poor network performance; however, increased deployment of *Debuglet* services at other ASes will make it increasingly clear where the bottleneck lies. For example, a Tier 3 ISP with poor performance will be exposed because its customers suffer poor performance that cannot be isolated to its providers. Also, a Tier 1 or Tier 2 ISP, despite not deploying *Debuglet* services, will still be on the routing path for many *Debuglet*-to-*Debuglet* or *Debuglet*-to-client paths, and its contribution to network bottlenecks will become more clear as its neighbors (both upstream and downstream) deploy *Debuglet* services. Therefore, though an AS can initially hide its poor performance by not deploying *Debuglet* services, as the system gains adoption, that AS will be increasingly exposed over time.

At the same time, well-performing ASes will seek to provide *Debuglet* services, as they will provide three important benefits: (i) they will help validate the performance of that AS' network, (ii) they will help the AS' customers verify the actual bottlenecks and improve customer support, and (iii) they will obtain additional revenue from running *Debuglet*s.

C. Economic Impact

Debuglet applications, which are generally short-lived and require relatively little resources, should be fairly inexpensive to run. Considering cloud computing prices as a baseline, running a *Debuglet* can cost less than 100 millicents. When *Debuglet* application demand is low, deploying ISPs can either deploy virtual machines on computers already running for other purposes, such as network management, or by deploying a low-cost and low-power Single Board Computer, such as a Raspberry Pi 4, which draws between 4–6 W of power, or less than 53 kWh each year.

The low cost, both of deploying *Debuglet* services, and of executing *Debuglet* applications, can generate significant value for ISPs. For example, a Customer Premises Equipment (CPE) that can interact with *Debuglet*s in firmware can help technical support isolate a customer's connectivity or performance issues. For example, the system may determine that the issues are entirely contained within the home network, where support personnel can suggest resetting the WiFi router, or plugging directly into the CPE. In other cases, the ISP may be able to determine that the network issues arise outside of the ISP's network, for example on the downstream path toward a particular service. Because technical support staffing is a significant cost to ISPs, even minor productivity improvements to their workflow can provide significant Return on Investment for amounts spent on deploying *Debuglet* services or for executing *Debuglet* applications.

D. Wise Selection of Debuglet Executions by Initiators

Fast and efficient fault localization in *Debuglet* demands a strategic selection of executors. This choice profoundly impacts time-to-locate, executor load, and measurement costs. For example, consider a path over 10 consecutive ASes with a fault in the last inter-domain link. Running a series of consecutive measurements from the first to the last AS can impose a long time-to-locate and high cost. Simultaneously examining all the links may not address cost concerns.

In a general case without prior knowledge regarding the fault, a binary search offers a cost- and time-effective solution. Initial measurements from the path's midpoint to each end domain may reveal the faulty half, repeating until all faults are located. Nevertheless, the responsibility lies with initiators to choose the selection strategy, relying on educated initial guesses, historical data, and potentially leveraging machine learning for informed decisions. We envision interesting research to devise smart approaches for different circumstances, and leave them for future work.

E. ISPs' Effort to Hide their Network Faults

ISPs may aim to hide faults from *Debuglet* measurements to protect their reputation and revenue, e.g., refund claims for SLA violations. They may attempt this by prioritizing packets to/from *Debuglet* executors or manipulating measurement results pre-submission to the blockchain. We argue that both cases are either impractical or costly. In the first case, tracking all the executors' addresses and prioritizing their packets in forwarding devices is challenging. A feasible attack is to prioritize packets from the prefixes of interest, but can be easily cross-validated by running measurements from diverse network vantage points. Similarly, the second case can also be easily detected by running multiple measurements and cross-checking the results.

F. Age of Information

For immediate network diagnostics, results that are more than a few seconds old may no longer be of use. However, historical information can still be useful in some cases. For example, given multiple measurements a common network diagnostic (e.g., latency and loss rate of TCP packets) over a fixed path, the trend in measured results over time might help identify the time at which the path started experiencing performance degradation, as well as the possible location of that degradation. Thus, it would be beneficial for several applications if network measurements were retained for a period between a week and several months. However, such archiving does not necessarily need to be kept on-chain; instead, blockchain explorers or network information monitoring sites could retain measurements that are still potentially useful, and the hash of measurements would be stored on the chain for verifiability purposes.

G. Alternative Executor Locations

We now review other potential scenarios of executor deployment that ASes can consider.

Arbitrary Locations in each AS. Deploying *Debuglet* executors at arbitrary locations within each AS introduces challenges in fault localization accuracy. In scenarios where executors are not necessarily on the original data forwarding path, false positives and negatives can occur. Furthermore, arbitrarily located executors struggle to provide fine-grained fault localization between consecutive ASes, unable to distinguish network failures in the preceding AS, the inter-domain link, or the succeeding AS. This lack of precision may render measurements inconclusive. Additionally, deploying executors within a network poses a security risk, potentially using *Debuglet* infrastructure for unveiling internal topology, routing policy, and traffic engineering rules.

Co-location with Every Router in the AS. Deploying at least one executor on each router within an AS offers potential advantages in minimizing false positives and negatives compared to the arbitrary model. Here, end domains can strategically select an executor on the forwarding path for measurements, enhancing the accuracy of fault localization. However, this model comes with drawbacks. First, it entails a substantial violation of ASes privacy, demanding the explicit disclosure of internal topology and routing policies. Additionally, it provides fault information at a granularity finer than necessary, pinpointing vulnerable points and bottlenecks with high accuracy, thereby exposing the AS to targeted attacks. Moreover, it demands extensive resource requirement, rendering it impractical for real-world implementation.

VII. Related Work

In-band network telemetry (INT) [29] is a promising and innovative network telemetry approach to monitor network performance and localize faults with high accuracy and in real-time. INT accomplishes this goal by using packets of the original data flow to collect information about network performance. This is done by every router on the forwarding path, adding telemetry information to the header of every data packet. The added information can include but is not limited to timestamps of when the packet arrives at a router, when it leaves a router, and what the queue length and queue waiting times are. The last router on the path collects this information and sends it to a centralized controller for further analysis. Nonetheless, INT is typically applied for *intra-domain* fault localization. Applying INT to public inter-domain scenarios would face challenges: (1) dependency on the destination domain's support for collecting and forwarding measurements to a (remote) central controller, (2) potential packet prioritization by domains with INT headers to hide their faults, and (3) lack of control by on-path ASes over the rate of INT which leads to increased load during network faults.

RIPE Atlas [3] is a vast Internet measurement infrastructure with probes distributed across the globe, offering real-time insights into Internet dynamics. Users earn points by deploying and running probes, and use the points to conduct network measurements. Besides a default set of pre-programmed measurements, users can also schedule customized measurements. Although *Debuglet* shares the foundational principle of running customized measurements from various vantage points, it seeks different goals. While RIPE Atlas aims for a comprehensive understanding of the Internet, *Debuglet* focuses on providing domains an efficient tool for pinpointing inter-domain network faults. *Debuglet* deploys executors at domain borders, facilitating fine-grained fault localization, diverging from (often) home-based deployment of RIPE Atlas probes. Nonetheless, through running measurements between the RIPE Atlas probes and the *Debuglet* executors, both architectures can be mutually beneficial.

WTF [26] is a status-reporting and fault-localization mechanism that collects health bits and uses them to localize faults using heuristics or machine learning. WTF can combine with a tracing mechanism that gathers relevant health-bit histories for fault-localization. *Debuglet* differs in that they do not rely on other applications (or other instances of the same application) to reliably report health status; instead, they use *active measurement* over sub-paths to establish ground-truth about network conditions.

NetQuery [25] takes an AS' knowledge and measurements and makes them available to applications through a *sanitizer* that removes sensitive topology information. NetQuery's measurements are taken using Trusted Computing, which prevents an AS from sending false measurements to applications. *Debuglet* can adopt NetQuery's sanitizer, but differs in that they allow remote applications to conduct their own measurements, constructing arbitrary probing packets, while incentivizing deployment through micropayments.

iPlane [22] uses an overlay network built on top of PlanetLab to perform periodic distributed measurements of loss rate, capacity, and available bandwidth, and from these results predict network performance. By contrast, *Debuglet* is deployed inside the network, allowing for direct measurements of path components rather than inferring them from

overlay measurements. *Debuglet* also provides a different, externally-verifiable mechanism for distributing the results of these measurements.

Network monitoring and debugging has been explored in numerous previous papers, including systems that use information from end-hosts [10], [15], [33], and information from the infrastructure [21], [23], [31]. Many of these systems include localization schemes that would work in a synergistic manner with *Debuglet*.

VIII. Conclusion

Because different types of traffic experience the network differently, we argue for a distributed network debugging infrastructure, *Debuglet*, to address the shortcomings of purely end-to-end debugging systems. By compensating ASes for hosting *Debuglet* executors in their network, we remove many usage-based attack vectors, and by retaining results within a blockchain, we allow users to prove their performance to third parties and allow users to explore performance trends. Because our design is based on smart contracts, both payment and result logging can be enforced through code rather than trust. *Debuglet* does not aim to replace existing network debugging infrastructure; rather, build on previously proposed mechanisms and provide data points that were so far difficult to obtain. These data points can be combined through the use of fault localization algorithms to provide a more fine-grained and robust view of the network.

Acknowledgments

We would like to thank François Wirz for his technical support of this study; and Karl Wüst for providing us with insightful knowledge of modern blockchains. We gratefully acknowledge support from ETH Zürich, and from the Zürich Information Security and Privacy Center (ZISC).

References

[1] ebpf-dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io.

[2] Move concepts. https://docs.sui.io/concepts/sui-move-concepts archived at https://perma.cc/F5YB-74PF.

[3] Ripe atlas. https://atlas.ripe.net archived at https://perma.cc/PDG2-H67LE.

[4] Scionlab. https://www.scionlab.org archived at https://perma.cc/7YXB-UG2E.

[5] Sui Lutris: The distributed system protocol at the heart of Sui. https://mystenlabs.com/blog/sui-lutris-the-distributed-system-protocol-at-the-heart-of-sui# archived at https://perma.cc/287P-LLCM.

[6] SUI Price. archived at https://perma.cc/VG2T-C3Y2.

[7] Userspace ebpf vm - github.com. https://github.com/iovisor/ubpf.

[8] Wasmer. https://wasmer.io archived at https://perma.cc/6DLH-8J6N.

[9] Webassembly. https://webassembly.org archived at https://perma.cc/DWT5-EMJW.

[10] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *Proceedings of the ACM SIGCOMM*, 2016.

[11] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-End Performance Analysis of Large-scale Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[12] L. Chuat, M. Legner, D. Basin, D. Hausheer, S. Hitz, P. Müller, and A. Perrig. *The Complete Guide to SCION*. Springer, 2022.

[13] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. Segment Routing Architecture. *RFC 8402*, 2018.

[14] K. Gao, C. Sun, S. Wang, D. Li, Y. Zhou, H. H. Liu, L. Zhu, and M. Zhang. Buffer-based End-to-End Request Event Monitoring in the Cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[15] C. Guo. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the ACM SIGCOMM*, 2015.

[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[17] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, et al. Canopy: An End-to-end Performance Tracing and Analysis System. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2017.

[18] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed Diagnosis in Enterprise Networks. In *Proceedings of the ACM SIGCOMM*, 2009.

[19] T. Kuchler, M. Giardino, T. Roscoe, and A. Klimovic. Function as a Function. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2023.

[20] J. Kwon, J. A. García-Pardo, M. Legner, F. Wirz, M. Frei, D. Hausheer, and A. Perrig. Scionlab: A next-generation internet testbed. In *Proceedings of the International Conference on Network Protocols (ICNP)*, 2020.

[21] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the ACM SIGCOMM*, 2016.

[22] H. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. IPlane: An Information Plane for Distributed Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[23] V. Nathan, S. Narayana, A. Sivaraman, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Demonstration of the Marple System for Network Performance Monitoring. In *Proceedings of the SIGCOMM Posters and Demos*, 2017.

[24] R. Pass and A. Shelat. Micropayments for decentralized currencies. https://eprint.iacr.org/2016/332.pdf, 2016.

[25] A. Shieh, E. G. Sirer, and F. B. Schneider. NetQuery: A Knowledge Plane for Reasoning about Network Properties. *ACM SIGCOMM Computer Communication Review*, 2011.

[26] W. Sussman, E. Marx, V. Arun, A. Narayan, M. Alizadeh, H. Balakrishnan, A. Panda, and S. Shenker. The Case for an Internet Primitive for Fault Localization. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. ACM.

[27] P. Tammana, R. Agarwal, and M. Lee. Simplifying Datacenter Network Debugging with PathDump. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[28] P. Tammana, R. Agarwal, and M. Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[29] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li. In-band Network Telemetry: A Survey. *Computer Networks*, 2021.

[30] B. Trammell, J.-P. Smith, and A. Perrig. Adding Path Awareness to the Internet Architecture. *IEEE Internet Computing*, 22(2):96–102, 2018.

[31] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[32] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[33] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the Characteristics and Origins of Internet Flow Rates. *ACM SIGCOMM Computer Communication Review*, 2002.

[34] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.