Warrens: Decentralized Connectionless Tunnels for Edge Container Networks

Tom Goethals[1], *Member, IEEE*, Mays Al-Naday[2], Bruno Volckaert[1], *Member, IEEE*, Filip De Turck[1], *Fellow, IEEE*

***Abstract*—In recent years, workload containerisation has been extended to the edge, bringing with it the need for flexible overlay networking. However, current container networking solutions are generally designed for the cloud, aimed at relatively static clusters with centralized generation of container subnet addresses and assigning them to nodes. Added to that existing tunneling solutions, such as Virtual Private Networks (VPN), also have centralized components. Conversely, the network edge is geo-dispersed and has a volatile topology,with edge nodes typically hidden behind routers, in private networks. To enable large-scale networking at the edge, there is need for decentralized self-management of container network addresses and overlay tunnels. This manuscript presents Warrens, a framework for fully decentralized and self-organizing cloud-edge container networks. Warrens enables communication between edge nodes in different private networks by enabling connectionless tunnels, supported by decentralized self-assignment of container IP addresses, with the assignment scheme minimizing address conflict to a negligible level. Warrens has been implemented in two variants using kernel-level *eBPF* for processing speed, and user-level *Golang* for wider compatibility. Warrens is shown to be highly scalable compared to a typical VPN solution, and performance evaluations demonstrate it can handle a full network load on both x64 devices and a Raspberry Pi with $\approx 0.5\%$ to 5% total CPU load, depending on traffic direction and protocols used.**

***Index Terms*—edge computing, container networking, decentralization,**

I. Introduction

In recent years, containerized cloud applications have been extended to the edge by various frameworks, including different flavours of Google's widely adopted Kubernetes [1].

Each edge node maintains a limited number of autonomous deployments of containers, communicating with each other within the boundaries of the edge gateway, but not directly with deployments outside it [2]. To enable edge container networks at scale, there is a need for novel solutions that *transcends* the boundaries of edge gateways and *scales* to a large number of containers. Existing container networking solutions are typically limited by the number of manageable containers and compute nodes, in the case of Kubernetes this is approximately 5.000 nodes and 150.000 containers [3]. As a result, these solutions have generally been managed using IPv4 addressing schemes, either exclusively or by default, despite Kubernetes IPv6 support. This approach implicitly reinforces node limits with a limited address size, although exceptions exist such as Cilium [4] and cloud provider solutions [5]. Such limits stand in contrast to edge computing, where there can be tens of thousands of nodes in a logical cluster (e.g. containerizing smart residential areas), with each node only able to run a small number of containers, and typically small subsets of nodes communicating with each other at any given time. Furthermore, cloud container networking solutions are highly centralized; the control plane needs to generate subnet addresses and assign them to worker nodes, while various higher-level network functionalities are handled by plugins in the control plane (e.g. service/ingress routing, DNS).

As edge networks steadily grow larger, integrating new types of devices suitable for containerized edge computing, such centralized networking approaches no longer suffice to organize all edge devices in a cluster, and the containers they run [6].

First, a centralised solution - e.g. based on a Virtual Private Network (VPN) [7] - needs to maintain a large state of address assignments that correspond to active nodes and containers in the edge. Additionally, there is the added challenge of tracking and updating state as containers join or leave networks. Second, having a central solution implies that all traffic of a container network must pass through a central server. Considering the potential scale of the edge, this will likely induce a high number of concurrent flows at the server, requiring an unrealistic service rate to avoid congestion. Third, a VPN-link solution can lead to routing scalability and security issues, because it enables full connectivity between all endpoints in the overlay at any point in time. In reality, however, only a smaller subset of containers (i.e. those in the same service architecture) need to communicate with each other at any given time.

Hence, there is a growing need for a decentralized container networking solution that offers a globally uniform network structure across private network boundaries, which allows communication between any pair of devices, while primarily ensuring optimal communication with nearby nodes and services.

To that end, this manuscript presents Warrens as a novel decentralized solution to container networking in the edge. Warrens establishes lightweight, connectionless tunnels between edge nodes, created as required. The name is derived from rabbit warrens, which have a multitude of tunnels between locations rather than a single central hub. To discover nodes and map their containers to network addresses, Warrens requires an external discovery algorithm; one option is to use the node discovery algorithm of SoSwirly [8], which discovers available

[1] Ghent University - imec, IDLab,Gent, Belgium, Email: tom.goethals@UGent.be
[2] University of Essex, School of Computer Science and Electronic Engineering, Colchester, UK

nearby edge nodes and their containerized deployments in near real-time. Warrens has two implementations: one using an eBPF program in kernel space for high performance; and another using Golang in user space for broad compatibility with a variety of edge devices. Furthermore, a custom IPv6 addressing scheme is proposed that allows each node to self-assign an IP subnet for containers deployed on the node with a negligible chance of collisions. As such, the solution is entirely decentralized and shown to be highly scalable. We evaluate the performance of both Warrens implementations for TCP and UDP flows, under a range of edge scenarios and show both to achieve superior scalability compared to VPN.

Concretely, the contributions of this manuscript are:

- **C1 (Design):** A decentralized and self-organising container networking solution, consisting of an agent on each node that uses a novel addressing scheme to generate container addresses, and sets up communication with remote containers.
- **C2 (Implementation):** A Proof-of-Concept (PoC) implementation in two variants: using eBPF for performance and Golang for broad-compatibility. Both variants allow devices behind NAT or firewalls to join a cluster, while presenting options for near real-time node and service discovery.
- **C3 (Evaluation):** Experimental evaluation to characterize the feasibility and performance of the eBPF and Golang implementations compared to a VPN-based solution.

The rest of this manuscript is organized as follows: Section II presents related work, while Section III introduces all the high level architecture aspects. Section IV discusses all relevant implementation details. In Section V, the evaluation setup, scenarios and methodology are detailed, while the results are presented in Section VI and discussed in Section VII. Finally, Section VIII discusses ideas for future work, and Section IX draws high level conclusions from the manuscript.

II. Related Work

Various studies examine properties of Kubernetes CNI plugins in edge scenarios, for example low latency edge services [9] or a comparative analysis of CNI plugin performance and network degradation [10].

Extended Berkeley Packet Filters (eBPF) can be used to develop various networking functions which operate in kernel space [11]. eBPF functionality is used extensively to improve (container) network security [12], or for Kubernetes networking [13], notably by Cilium [14] which provides a wide array of eBPF functionality.

EdgeVPN [15] is a solution for self-organizing virtual Ethernet-level edge networks, providing an overlay infrastructure that allows higher-level applications such as Kubernetes to run in the edge without modifying CNI plugins. EdgeVPN.io [16], on the other hand, is a Kubernetes CNI plugin with tunneling capabilities which allows integrating devices behind NAT into a container network, although working from a Kubernetes perspective it lacks self-organizational elements. Warrens on the other hand aims to combine these elements into self-organization at the container network level. KubeEdge [2] approaches edge networking by using Kubernetes in the cloud, but implementing KubeBus as an overlay network to allow edge devices to communicate with the cloud. Edge to edge communication is enabled by setting up tunnels between nodes through KubeBus, routing traffic via the cloud.

A non-Kubernetes approach on interservice communication in the edge [17] uses service-based fog Management And Network Orchestrator (sbMANO) nodes, forming a decentralized network which enables peer-to-peer communication and allows the decentralized collection of service metadata to optimize service call routing. Evaluation of this approach was shown to work for a variety of parameters (e.g. inter-node latency, resources) and nodes, ranging the entire edge-cloud spectrum.

A review on the observability of distributed, container-based edge systems [18] illustrates the difficulty of using classical tools to analyze issues in the network edge, naming issues such as a decentralized nature and heterogeneity.

Segment Routing (SR) [19] is a technique that manages a middle ground between centralized and decentralized network control for overlay networks by dividing a packet route into segments and including the information for each segment along the path, allowing for detailed traffic engineering, load balancing and latency management. However, the focus of Warrens is merely that individual endpoints can communicate across private network borders, and that each node organizes its own containers. Unlike SR, it is not aimed at traffic optimization and thus needs no knowledge of intermediate nodes, relying on the regular functioning of network routing to manage traffic between nodes.

While IPv6 Stateless Address Autoconfiguration (SLAAC) [20] can be used to generate unique link local IPv6 addresses for individual devices (e.g. in IoT scenarios), the addressing scheme used in Warrens is aimed at the coherent generation of container subnets for each node in a cluster, minimizing collisions only from a decentralized point of view.

Fowler-Noll-Vo based systems have previous been used for key management in IoT devices [21] and IPv6 autoconfiguration schemes for IoT devices [22]. However, this work aims to use a similar Fowler-Noll-Vo approach for both IPv4 and IPv6, in the context of container subnets per device rather than individual IP addresses or keys.

Finally, a scalability analysis of various VPN solutions [23] indicates that, excepting WireGuard, VPN software is only suitable for connecting hundreds of containers (and by extension devices) in private networks to a cloud cluster, far below the scale of many edge computing use cases.

III. Warrens Architecture

This section describes the architecture of Warrens as per contribution **C1**, and the frameworks used to enable its operation such as a suitable edge orchestration agent. Warrens is a two-stage networking solution, supported by peer-to-peer service discovery and self-assignment of IP subnet addresses to enable decentralized overlay construction. The two-stage networking involves container-node packet forwarding inter-node flow tunneling, with either L3-over-L4 or L2-over-L4

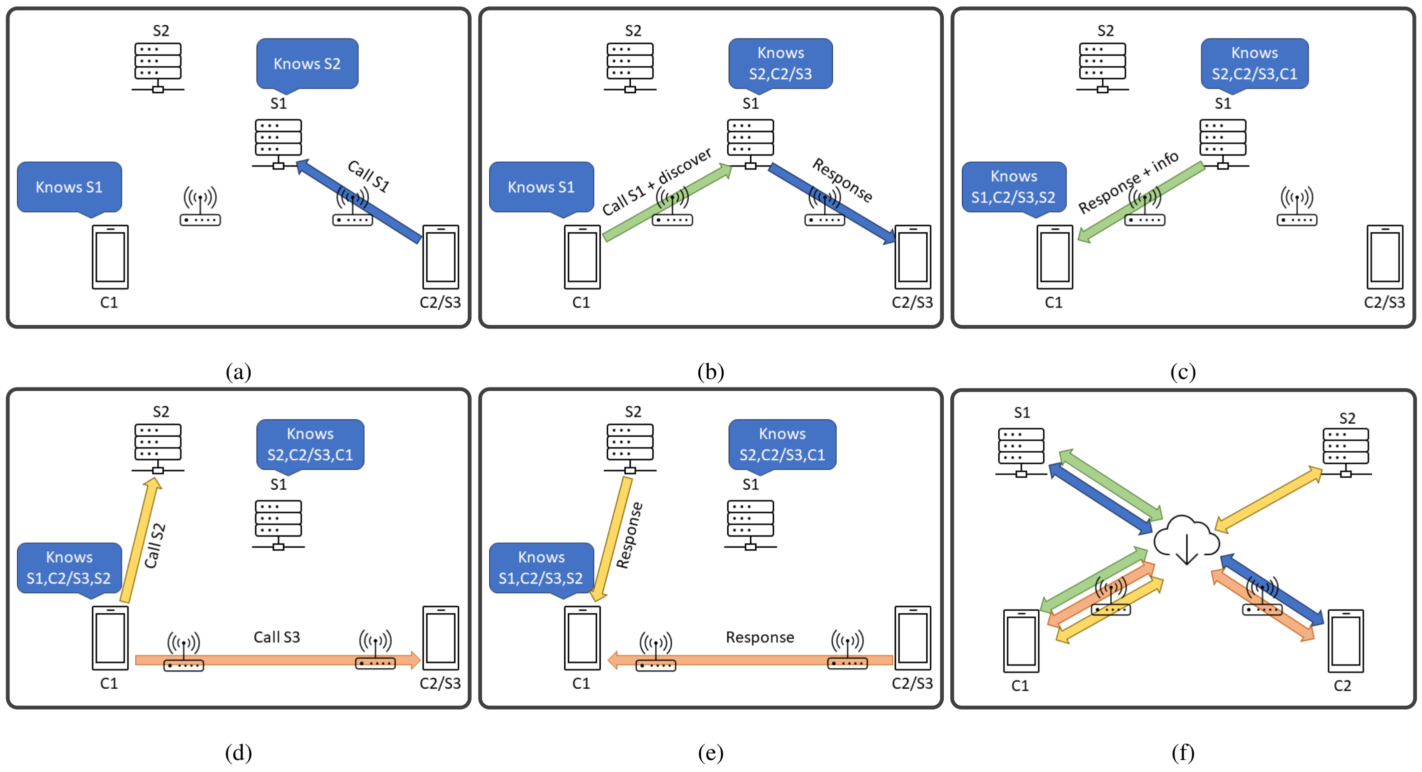

Fig. 1: Illustration of Warrens using connectionless "tunnels" between **C**lients and **S**ervices as required, exchanging service information in the process Fig.1a - Fig.1e. Note that tunnels do not require explicit setup or teardown, and exist only as an abstract concept allowing container traffic to cross private network borders. This method can use the full network throughput between each pair of devices. Fig.1f shows the same communication channels going through a centralized component, eliminating the need for information exchange between nodes, but causing a communication bottleneck in case of too many clients.

tunnels. Tunnel establishment follows a decentralized approach, involving collaborative service discovery, address self-assignment and local container orchestration.

A. *Container Network Tunneling*

Fig. 1 shows the high-level concept of inter-node/inter-service communication using Warrens. Communication consists of on-demand, point-to-point connectionless tunnels, established between containers-hosting nodes and connected to form an overlay network. Warrens bootstrapping is a decentralized process, realised through peer-to-peer communication among interested entities. Subfigures 1a - 1e show how Warrens exchanges metadata to enable communication using "tunnels" between services S1-S3 and clients C1-C2. Here, both C1 and C2 are devices that require service S1, while C2 also runs S3. The device hosting S1 knows the location of S2, while C1 already knows S1. C2 has discovered the device hosting S1 (Fig. 1a) with a suitable discovery solution such as that summarised in Section III-E. Upon discovery, both S1 and C2 obtain information of each other and Warrens communication can be established, allowing C2 to call S1. C1 calls S1 and performs a discovery update (Fig. 1b), resulting in transitive knowledge of S2, and C2/S3 through S1. C1 can now reach S2 as well as C2/S3 and establish Warrens tunnels to call their respective services (Fig. 1c-1e). Subfigure Fig.1f shows how this compares to the same service calls facilitated by a centralised tunneling solution with a single point of management, such as a VPN server node or KubeEdge networking control plane.

To clarify, Warrens "tunnels" are only temporarily defined in the sense that container traffic is routed between a pair of nodes; no connection is established, and without container traffic tunnels effectively seize to exist without the need for teardown. To enable traffic flow, nodes merely need to exchange their public IP addresses and self-assigned container subnet address (as detailed in Section III-C), and traffic is routed to the correct containers as required. This connectionless approach allows for a lightweight implementation with minimal memory use. However, connections may have to be explicitly maintained in some cases, for example by keeping a NAT mapping entry active with a periodic heartbeat mechanism. This is particularly expected for nodes in private networks, and can be integrated as part of node and service discovery algorithms as explained in Sections III-D and III-E.

Fig. 2 shows two forms of tunneling on two separate nodes. The tunneling approach depends on the implementation of components required to set up tunnels on a single node, further elaborated in Sections III-B through III-E. The container orchestrator agent, container addressing and discovery algorithms are identical on both nodes. In both cases, all

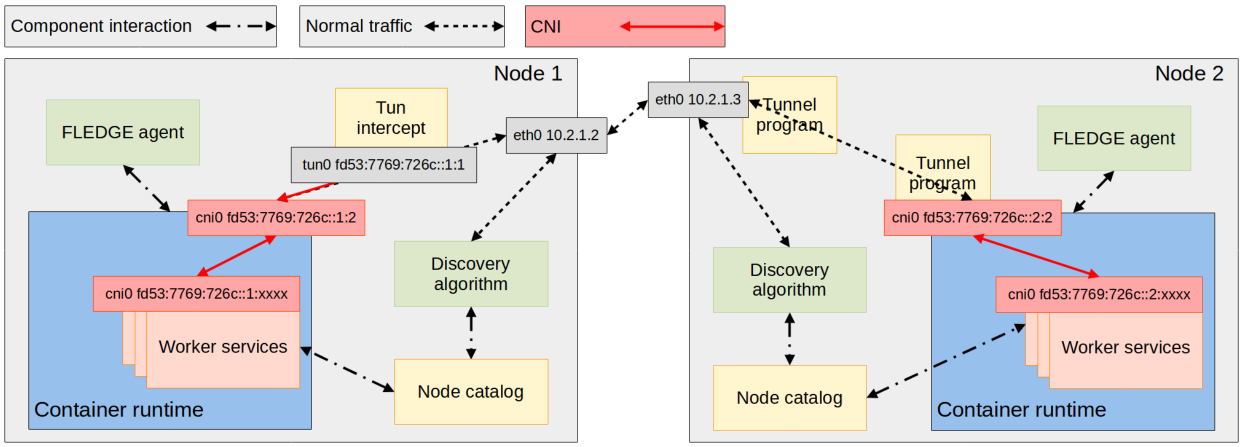

Fig. 2: Components and networking aspects of two different possible Warrens implementations, using either a TUN device (Node 1) or kernel-level intercepts (Node 2).

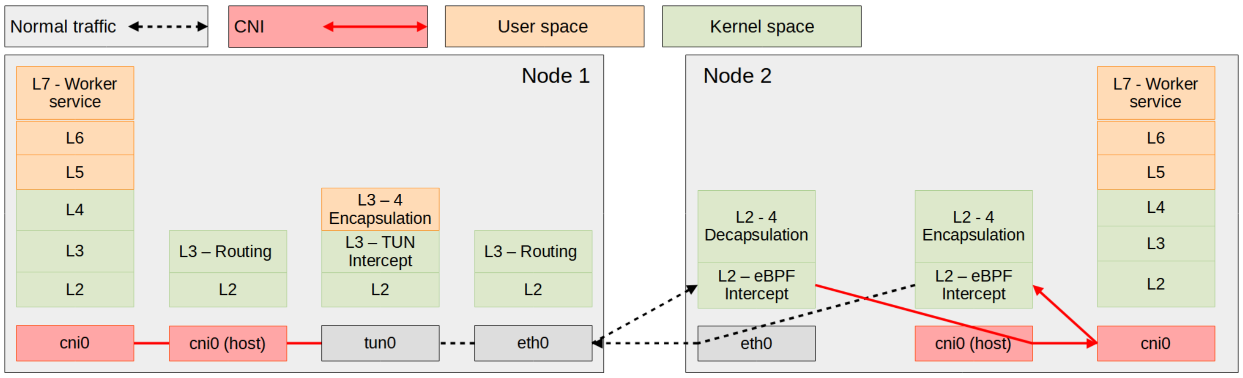

Fig. 3: Network interfaces and stacks used in Warrens. Only the highest used OSI model level for each stack is shown. Note that the eBPF intercepts “cross” each other; they bypass parts of the network stack of eth0 and cni0, emitting directly from the outbound interface of the other.

container interfaces consist of virtual Ethernet interface pairs between the host and container namespaces, with the host ends connected to a bridge interface, *cni0*, specifically set up to connect all local containers. At this point, the implementation at Node 1 uses Linux routing rules to forward packets targeted at other container subnets to an IP-level tunnel device, (TUN, for example tun0), where a userspace process intercepts them, encapsulates them in UDP packets targeted at port 31337 of the public IP address of the remote node, and forwards them to the public-facing interface, for example eth0, essentially forming an L3-over-L4 overlay network. Incoming traffic at port 31337 is similarly routed to tun0, decapsulated by the same userspace process and forwarded to cni0, where Linux routing takes over to forward packets to the destination container(s). This L3-over-L4 approach is deliberately chosen for its comparability to a VPN; Warrens traffic is easily distinguished from other incoming traffic at the public-facing interface, with only port 31337 traffic needing further processing. Moreover, the L3-over-L4 approach with IP-in-UDP was chosen over an IP-in-IP counterpart because many types of (home) routers can only effectively route traffic for either UDP or TCP, by using NAT or explicit routing rules.

In the alternative approach, implemented at Node 2, a tunneling program is attached directly to cni0 at the kernel level, which examines all container traffic to determine which packets have a remote destination, operating as L2-over-L4 encapsulation. In case of a remote destination, the packet is encapsulated in a UDP packet targeted at the remote node. This version bypasses several layers of the networking stack, removing packets from the inbound stack of cni0, and emitting them from the outbound stack of eth0, with no processing or possible intercepts in between. A different kernel-level program is attached to eth0 to examine incoming traffic, which cuts off the UDP header of Warrens traffic at port 31337 and emits it directly from cni0, bypassing the rest of the networking stack. At cni0, Linux routing takes over to send the packet to the correct container interface.

Fig. 3 shows the same architecture on both nodes, from the point of view of network interfaces and network stacks. At Node 1, network traffic goes through the cni0 bridge at the IP level, rerouted to tun0 as required. At tun0, a TUN device picks up IP packets and the user space process encapsulates them, forwarding them to a remote destination. At Node 2, the process is slightly different for incoming and outgoing traffic, but symmetric. In either case, traffic is processed by a kernel space filter at the Ethernet level before it enters the network stack of either eth0 or the cni0 bridge, encapsulated or decapsulated as required, and then emitted from the outbound stack of the other interface.

Although the kernel-level version is by far preferable in terms of performance, it is useful to co-implement the first as a universal option, for compatibility reasons detailed in Section IV. Considering that neither implementation encrypts network traffic for performance reasons, applications should secure their network traffic whether they communicate over Warrens or any other network, and the current approach does not risk exposing device information other than Warrens container addresses.

While the operational stability of Warrens on any individual node depends on implementation, the decentralized networking approach described results in a highly resilient container network overall; if any single node fails, the rest of the network keeps functioning, while a suitable node discovery algorithm ensures stability when either the physical or logical network topology changes.

B. Container Management

For establishing Warrens between containers in the edge, a resource efficient orchestrator agent is needed to manage containers on edge nodes. For the purposes of this manuscript, Warrens is used in combination with FLEDGE [24], a Kubernetes-compatible edge orchestration agent. The role of FLEDGE in this architecture is that of a high-level container runtime, receiving Kubernetes pod deployments and managing pod containers through containerd. Its default container networking solution is substituted with Warrens, enabling any containers deployed by FLEDGE to communicate with other nodes through Warrens tunnels. Although FLEDGE is a Kubernetes-oriented agent, Warrens itself is orchestrator agnostic and could be implemented as a solution for other edge computing architectures.

C. Container Network Addressing

Warrens is a decentralized networking solution in which every node constructs its own container IP subnet address, using only local information. Both IPv4 and IPv6 solutions are devised, although the larger range of IPv6 naturally lends itself to considerably lower IP range collisions across self-assigned addresses. As Fig. 4 shows, in both cases the address consists of a prefix, a global identifier to indicate a Warrens address, a node identifier and a container address range. The Warrens global identifier is derived from the hex representation of the string literal "Swirl" for easy identification, while the node identifier is derived from /etc/machine-id through a 64-bit Fowler–Noll–Vo hash.

1) IPv6: IPv6 addresses are based on, but do not adhere strictly to the unique local address structure [25]. The necessary prefix and 40-bit random data components are kept, the latter as the Warrens global identifier. As Warrens aims to create a container network between large numbers of edge devices, the sizes of the subnet identifier and interface identifier fields are swapped, allowing a maximum of 2^{64} devices and up to 65.520 containers per device, both sufficient by far

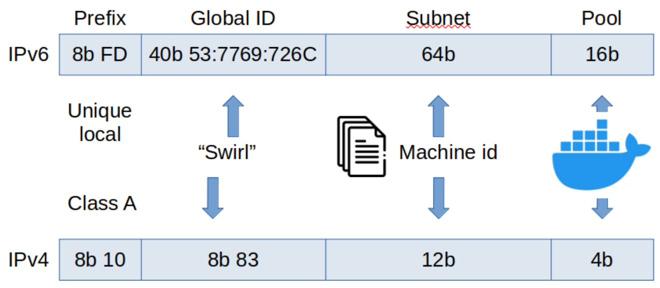

Fig. 4: Components and content of IPv6 and IPv4 Warrens container addresses.

for container networks in current edge networks. Although hashing a 128-bit machine id into 64 bits introduces an increased risk of address collisions, the birthday problem [26] shows that the risk is minimal even for millions of mutually known devices, and mitigation measures are introduced in network discovery algorithms.

2) IPv4: IPv4 options are severely restricted; assuming private address types offering 24 freely assignable bits, the global identifier is reduced to a single byte, while the remaining bits offer room to assign 4096 nodes a subnet identifier, each with 8 container addresses. The limited subnet identifier range greatly increases the risk of collisions at only tens of mutually known devices, which must be taken into account by network discovery algorithms.

D. Node Discovery

Warrens holds a remote node registry in memory, mapping self-assigned container subnets of remote nodes to their public IP address. While this registry could technically be loaded from a static configuration, an alternative approach would be to incorporate a discovery protocol based on locality, with fast reaction times to topology changes.

A suitable example of such an algorithm is a modified version of the node discovery algorithm from SoSwirly [8], which scales independently of the total number of nodes in an edge cluster. Instead, scalability depends on local node density and discovery distance. This algorithm is designed to find all nodes within a certain logical distance (e.g. latency) that are capable of providing specific servicesby crawling an edge cluster transitively until it finds suitable nodes sufficiently close to itself. To support Warrens operation, the algorithm can be extended to provide node metadata. Using this algorithm, two mutually unreachable nodes (e.g. in private home networks) can still exchange metadata and set up a Warrens tunnel, given the existence of a single publicly available node known to both nodes as shown in Figs. 1a through 1c. The main limitation of this approach is that it cannot have global knowledge of all services, as that would result in too much network traffic for real-time discovery, and only asymptotically approaches an ideal situation. However, the SoSwirly algorithm ensures that a service with an acceptable Quality of Experience (QoE) is always nearby, essentially letting services self-organize around user requirements.

Note that the SoSwirly algorithm is explicitly built around volatility in edge networks; the algorithm periodically polls all its neighbours to update their positions and services, and to discover new nodes. As such, this algorithm implicitly keeps NAT mappings active for nodes in private networks once it is established.

E. Service Discovery

Recall in Figs. 1a-1e clients discover services around them. This requires a service exposure and resolution solution, with DNS being the intuitive one. While DNS is not explicitly built into the architecture at this point, the SoSwirly algorithm polls for any services running on a node in addition to node information itself, as it tries to find the closest provider(s) for a specific service name. As such, it is relatively straightforward to compile service IP addresses and names into a node and service catalog. Such a catalog can be implemented on the node itself by maintaining a dedicated Warrens hosts file to be mounted in containers, or by having some nodes run CoreDNS-based servers [27].

F. Security

While this article focuses purely on a solution architecture to enable decentralized container networks in the edge, there are several important security aspects to consider:

- Network traffic should be suitably encrypted to hide both unencrypted payload and any information about container network structure from potential attackers. A suitable encryption scheme and its likely performance impact are discussed in Sections V-C and VI-A.
- Decentralized tunneling and trusted connections ideally require decentralized methods to validate the integrity of nodes, as well as (self-)generating trusted encryption keys and certificates with minimal to no backing from central authorities. While these topics are beyond the scope of the article, they are highly relevant future work.
- At the tunnel level, Warrens implementations should be as secure as possible. This aspect is explored in Section IV-D.

IV. Implementation Details

This section describes two Proof-of-Concept (PoC) implementations of Warrens; GoLang-based for compatibility and eBPF-based for performance. Important implementation details are elaborated as required by contribution **C2**, especially as they differ between the eBPF and Golang versions.

A. Addressing

The concrete addressing scheme is identical for both implementations, although some devices may not appear in either. Table I summarizes static devices and ranges for IPv6 as partially illustrated by Fig. 2.

TABLE I: Dedicated IP addresses and available IP ranges for devices in the IPv6 version of Warrens.

Type	Range	Description
tun0	::1	A TUN device, if required
cni0	::2	The bridge connecting all local containers
reserved	::3 - ::f	Future system devices
containers	::10 - ::ffff	Assignable container IPs

B. Golang Implementation

The Golang implementation is based on the Node 1 setup in Fig. 2. During startup, FLEDGE generates a container subnet as described in Section III-C, sets up a cni0 bridge device, and a tun0 device. Traffic routed from cni0 to tun0 is picked up by a Goroutine in FLEDGE, encapsulating OSI level 3 packets (IP) into UDP packets and resending them from cni0 to the public interface. A secondary Goroutine listens on the same port for incoming packets, cutting off the Ethernet and UDP headers and resending them from cni0 towards container interfaces. As this implementation is meant for compatibility with all devices, it is highly straightforward but relatively slow; the code does not need to consider Ethernet-level details such as MAC addresses or checksums.

Container-side, every interface is attached to the cni0 bridge through a virtual ethernet pair. However, as IPv6 is used, both veth ends must be explicitly set as neighbours to avoid ICMP discovery messages, which are not correctly answered by default Linux mechanisms, and can not be intercepted and faked by either Warrens implementation. Additionally, IPv6 forwarding must be explicitly enabled, and MTU is lowered to 1420 to make sure packets fit a standard 1500 MTU once encapsulated.

C. eBPF Implementation

Extended Berkeley Packet Filters (eBPF) enable the creation of highly specific programs to act as kernel hooks for a variety of functions. By acting at the kernel level, eBPF programs can outperform similar (userspace) programs, although strict security and memory requirements are imposed by the compiler and interpreter. In the context of Warrens, the eBPF eXpress Data Path (XDP) allows high-speed processing of packets, which are intercepted before they enter the network stack of the interface an eBPF program is attached to.

The eBPF implementation is based on Node 2 in Fig. 2, and is meant for maximum performance if supported. While most modern Linux kernel versions (> 4.16) support eBPF to some degree, support libraries may have to be installed for optimal functionality. eBPF programs operate directly on memory addresses at the kernel level, but cannot allocate any other memory or persist variables between executions; in this case the entire programs are executed per packet.

Unlike the Golang implementation, the eBPF version only creates cni0, loading two eBPF programs into the kernel and attaching them to the eth0 and cni0 inbound stacks, respectively. Both programs share a memory map with the Golang program in which information about discovered nodes can be added. However, as eBPF is highly restrictive in operations and data types, this information is limited to primitive types.

The cni0 program examines all packets, comparing them to known nodes in the memory map. If tunnel information for a specific packet is found, it is encapsulated in new Ethernet, IP, and UDP headers, with suitable IP and MAC addresses; the latter requires extra metadata to be gathered by the SoSwirly discovery algorithm. Local traffic packets are simply passed, minimizing performance impact. Next, the packet checksum must be recalculated, making this a relatively expensive operation. The packet is then sent directly from the eth0 outbound stack. Importantly, this operation skips the entire network stack of both cni0 and eth0, so packets move almost directly from a container interface to the public interface. Similarly, the eth0 program examines all incoming packets, relinquishing any that are not explicitly Warrens traffic as soon as possible. For Warrens packets, the Ethernet, IP and UDP headers are removed, MAC addresses are modified to match local source and target devices, and the packets are resent from the cni0 outbound stack.

D. *Security Implications*

As Warrens is designed to function in the edge across networks, packet-based security issues warrant significant attention compared to container networks operating entirely in private networks. Both implementations hide the Layer 2 infrastructure of the container network on a node from the view of potential attackers by removing local MAC addresses from encapsulated packets. While the current state of the implementations allows the interception of Warrens container addresses, the implementation of an encryption scheme would eliminate this option as well by encrypting the internal IPv6 header with container addresses, making it impossible for attackers to map the structure of the container network and inject packets targeted at specific services. Additionally, as the eBPF implementation uses XDP, any malformed Warrens packets are intercepted by the eBPF program before they can exploit application or network stack security issues.

V. Evaluation

This section describes the evaluation setup, evaluation scenarios, methodology and any tools used. The code for FLEDGE, Warrens, and the evaluation scenarios is made available on GitHub[1]. Note that the evaluation will focus exclusively on the performance of Warrens tunnels; the theoretical and practical performance characteristics of the decentralized discovery algorithm SoSwirly are extensively covered in prior work [8], showing both high scalability and over 95% effectiveness in discovering edge node/service topologies.

A. *Evaluation Setup*

Most evaluations are performed on the IDlab Virtual Wall [28], using baremetal machines, each with 2 Intel Xeon E5520 CPUs and 12GiB of RAM. Additional evaluations to gauge performance on edge devices are run using a Raspberry Pi 3B+ (RPI), explicitly chosen over newer models to more realistically represent limited-resource edge hardware. All evaluations are performed using 1Gbps LAN connections, although the RPI is hardware limited to 100Mbps. Due to kernel and library compatibilities, Ubuntu 22.04 is required for eBPF evaluations, while Ubuntu 20.04 is used for the rest. On the RPI, Ubuntu Server 23.04 ARM 64bit is used.

[1]https://github.com/togoetha/cniwarrens

B. *Scenarios*

Three basic topologies are used for the evaluations:

- Point-to-point (*P2P*): provides a performance baseline for client-server traffic between two nodes. This topology is evaluated for both x64 and ARM.
- Four-to-one *Star*: simulates a star topology with 4 client nodes sending data to a single server. The scenario is included to evaluate multi-client performance.
- Five-node *Ring*: simulates a ring topology in which each node both sends and receives data, attempting maximum throughput in both directions. This scenario evaluates scalability and the ability to handle mixed traffic.

C. *WireGuard: a VPN Alternative*

A solution similar to Warrens can be achieved by organizing the proposed container addressing and networking on top of a VPN. While there are many VPN solutions, existing work shows [23] that WireGuard provides excellent performance for container network traffic. WireGuard is a VPN solution which is integrated into the Linux kernel, and like Warrens uses eBPF functionality, scaling to hundreds of VPN clients with no noticeable increase in latency or packet loss.

To compare Warrens and WireGuard, each scenario is also run with a WireGuard VPN set up across the nodes with a single node acting as VPN server. In FLEDGE, a configuration flag enables using an existing VPN interface for inter-node container traffic rather than Warrens. Note that traffic encryption cannot be disabled in WireGuard; although this results in an incomplete comparison, relative differences between scenarios will illustrate the scalability of Warrens compared to a classical VPN. Additionally, the performance impact of adding similar encryption in Warrens can be estimated. WireGuard uses ChaCha20 [29] for encryption, specifically for its relatively small key size and high performance. Considering a 20 round encryption over the entire payload of a packet, and that the eBPF implementation of Warrens must calculate a checksum over the same payload when sending packets, Warrens performance would likely drop by a factor of 20 when implementing ChaCha20.

D. *Methodology*

All evaluations use iperf3 to generate traffic between nodes. Base throughput is set at 1Gbps, although for UDP lower rates are used which asymptotically approach actual performance starting from 1Gbps. CPU load, memory overhead and network throughput are measured as the main metrics for each alternative. These metrics are measured for both TCP and UDP traffic, although UDP traffic was too unreliable to measure for the star topology as some nodes would never start sending traffic. Additionally, CPU-relative performance is

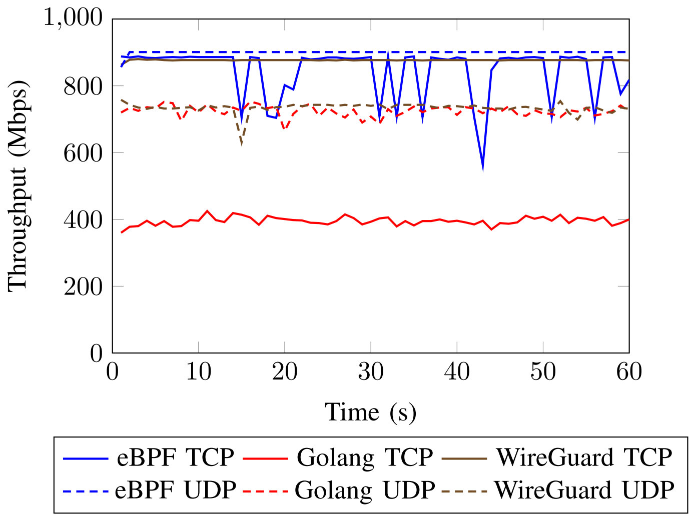

Fig. 5: Point-to-point bulk TCP/UDP throughput.

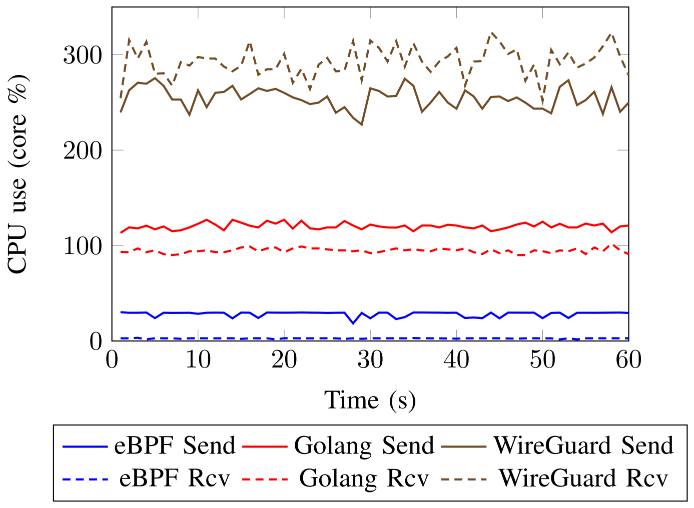

Fig. 6: Point-to-point bulk TCP traffic CPU load.

compared between nodes acting as senders (i.e. iperf3 clients) and receivers (i.e. iperf3 servers).

Network throughput is taken directly from iperf3 output. Memory overhead is determined by taking the memory use of Warrens as reported by the "top" command, and adding any memory used by kernel processes spawned by the evaluated alternative. However, eBPF program memory use is reported as a static 4096 bytes by "bpftool" (i.e. 8192 bytes for both eBPF programs combined), while "top" reports 0 bytes of memory for all kernel processes spawned by WireGuard, making the latter factor negligible.

CPU use is determined differently for each evaluated alternative. For eBPF evaluations, BPF stats are enabled in the kernel and measured every second through "bpftool". For the Golang implementation, CPU use is taken directly from the FLEDGE/Warrens process(es) as reported by "top". WireGuard spawns too many processes to track using "top", so CPU use is calculated based on total idle CPU reported by "top", and adding the CPU use of any iperf3 processes.

Each metric is measured for a continuous period of 65 seconds, ensuring at least a 60 second overlap between all metrics of all nodes after starting an evaluation run. However, the bash "sleep" command is found to run slow on the RPI, resulting in only around 53 measurements per minute. As such, those evaluations are cut off "earlier" despite running a full minute.

VI. Results

This section presents the results of the evaluations as per contribution **C3**. Each step examines raw throughput, CPU load, and CPU-relative throughput in order to discover all performance nuances.

A. *Point-to-point*

Fig. 5 shows TCP and UDP bulk throughput speed for a straightforward point-to-point connection between containers on two nodes. Whereas for UDP, the eBPF program manages to reliably saturate the gigabit connection, TCP traffic shows some performance drops. This behavior is more pronounced in the Golang tunnel due to being CPU limited. WireGuard shows the inverse behavior, with UDP traffic being slower, although official WireGuard benchmarks do not go into sufficient detail to explain the difference[2].

However, as Fig. 6 illustrates, the CPU load generated by the various options varies by orders of magnitude. The eBPF program can process a gigabit of traffic with only around 3% of a single CPU core, as it only needs to modify a limited amount of packet header information. Sending traffic requires around 30%, caused by recalculation of packet checksums. Golang shows a highly similar absolute difference, but because of its userspace implementation generates 90% more CPU load for only 45% of the throughput. Due to its encryption, WireGuard generates 10 to 100 times more CPU load than the Warrens eBPF program, showing that a full VPN is unlikely to be an ideal solution for edge devices, as they have far less powerful CPUs.

Finally, Fig. 7 shows UDP and TCP throughput relative to CPU load. In both cases, the Golang implementation performance is on par with WireGuard to send traffic, but twice to several times faster to receive traffic. However, as Warrens do not offer traffic encryption like WireGuard, this merely indicates that the Golang implementation is an acceptable fallback option from eBPF to enable up decentralized container networking. The latter is orders of magnitude faster than either the Golang implementation or WireGuard, achieving tens to hundreds of Mbps/s per percent of a single CPU core depending on traffic direction. Considering the earlier estimate that ChaCha20 encryption would slow down Warrens by a factor of 20, eBPF implementation performance would be similar to WireGuard when encryption is implemented.

B. *Star Topology*

Fig. 8 shows the received traffic throughput of a server node at the center of a star topology. As traffic depends on four client nodes, throughput is not quite constant, however both

[2]https://www.WireGuard.com/performance/

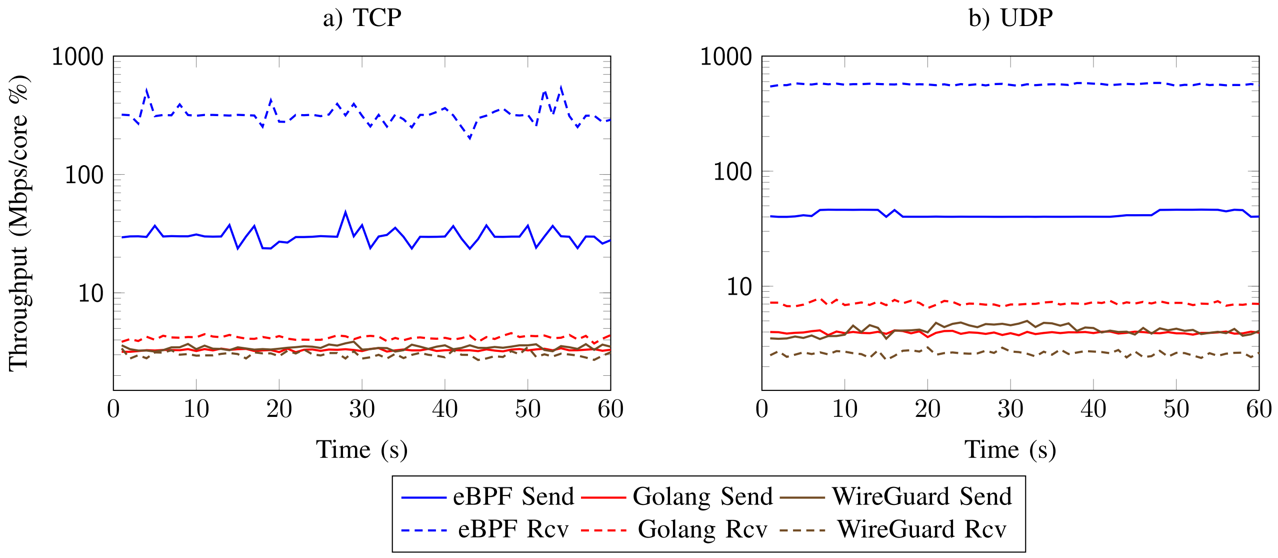

Fig. 7: Throughput per CPU core % for point-to-point bulk traffic.

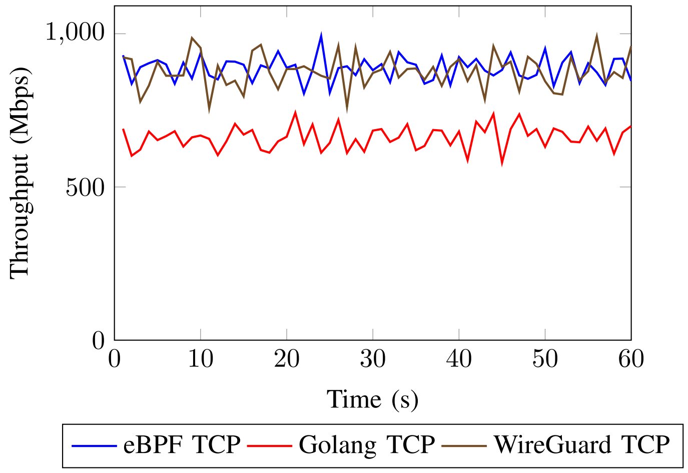

Fig. 8: Star topology server total TCP throughput. Note that each of the four clients steadily produce around 25% of this throughput.

eBPF and WireGuard manage to saturate a gigabit connection in a star topology. UDP is not examined in this scenario as the results are highly similar to the point-to-point scenario. Golang performance in this scenario is only around 25% slower than that of both alternatives. However, considering the roughly 400Mbps Golang TCP throughput from Fig. 5, the star topology results indicate this implementation is not capable of fully serving two clients, much less four. As such, the Golang implementation is a sufficient, but not excellent fallback options for star topologies.

However, there are significant differences in node behavior even within each alternative. Fig. 9 shows the CPU use of each node throughout the scenario for eBPF, Golang and WireGuard. For eBPF, client CPU use fluctuates between 4% and 12% of a core over the course of the scenario. Server CPU use, on the other hand, is quite stable at around 4.5% despite fluctuating throughput. Relative performance between clients and server reaffirms that Warrens servers with the eBPF implementation are highly efficient due to limited eBPF intervention in received packets. The performance of the Golang implementation and WireGuard is highly similar, differing only in absolute terms as WireGuard uses around twice as much CPU. While the underlying reasons are different, the explanation is simply that both options execute similar amounts of code for both incoming and outgoing packets, resulting in a server CPU use which is almost exactly the sum of client CPU use. However, as the Golang implementation uses only two threads, one for packet handling in either direction, server CPU use is at its limit while client CPU use is reduced to what the server is capable of handling.

The relative efficiency of the implementations is confirmed by Fig. 10, with the eBPF implementation again handling orders of magnitude more throughput per CPU load than the Golang implementation and WireGuard. While eBPF send performance is the same as in the point-to-point scenario, receive performance drops by 50% as a result of being overloaded by several clients. Similarly, Golang and WireGuard performance is 30% to 50% lower due to star topology bottlenecks, although their relative performance is nearly identical to the point-to-point scenario.

C. *Ring Topology*

As shown in Fig. 11, this scenario is where the performance benefits of decentralized Warrens become apparent. Using Warrens, each pair of nodes can nearly saturate a gigabit connection in both directions simultaneously when using the eBPF implementation, while a VPN generally has need of VPN server nodes that form bottlenecks. As a result, each of the five nodes in the ring only gets 200 Mbps of throughput on average. In more general terms, in a topology with n nodes and v necessary VPN server nodes, the average throughput for Warrens is $O(1)$ while for a VPN it is $O(v/n)$. Only in the edge case where each node is its own VPN master

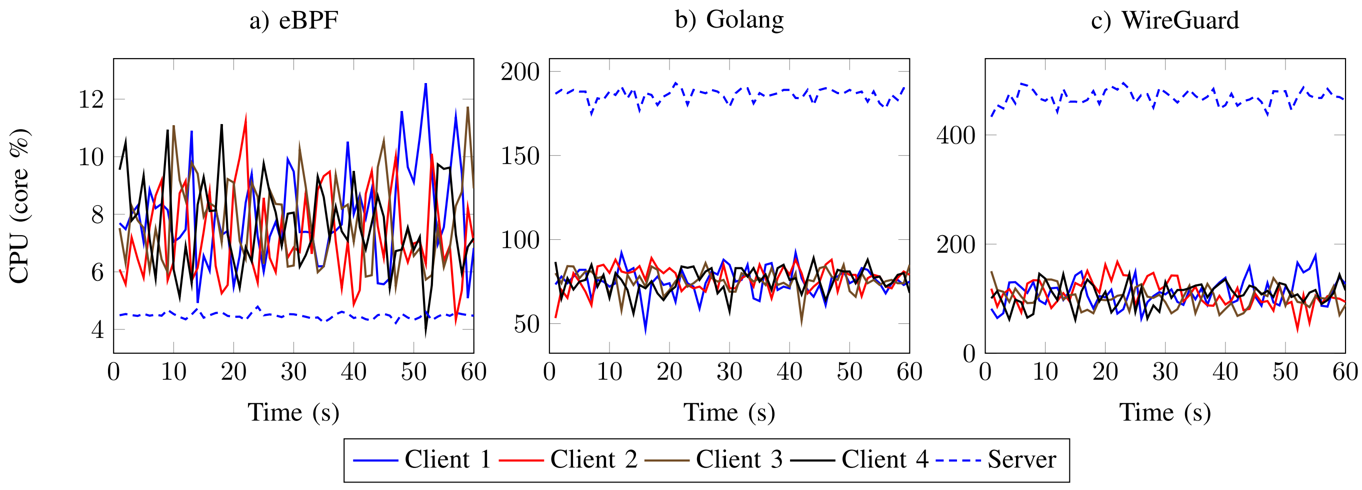

Fig. 9: Star topology CPU use of individual nodes for all evaluated alternatives. Note: each Y-axis is a different scale to emphasize client-server behavior.

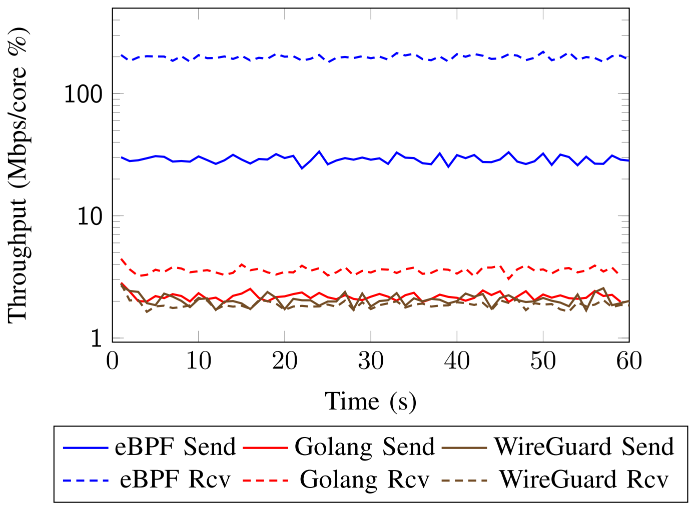

Fig. 10: Throughput per CPU core % for TCP traffic in star topology.

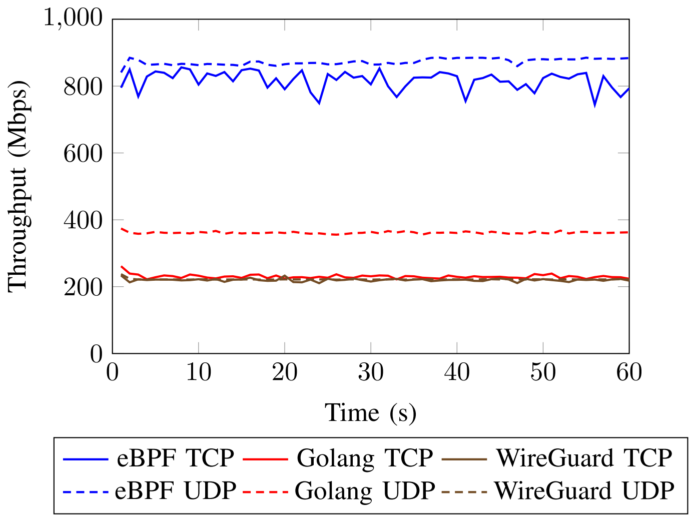

Fig. 11: 5 node ring TCP/UDP throughput.

could a VPN technically approach Warrens performance, although there would be a significant overhead factor for inter-node synchronization of VPN metadata. UDP throughput is included for illustrative purposes; both WireGuard TCP and UDP performance are capped at 200 Mbps due to the server bottleneck, unlike in the point-to-point scenario. For the same reason, the Golang implementation achieves a significantly higher throughput than WireGuard in this scenario, although evidently it remains a fallback option compared to eBPF.

Fig. 12 shows the throughput and CPU use of each node over the scenario runtime, illustrating the architectural differences in the networking solutions in more detail. Throughput is measured at the sending node, although influenced by the amount of simultaneously received traffic at any moment. Both the Warrens eBPF and Golang implementations show similar performance; each node has the same throughput (a + c) barring some spikes up to 20% for eBPF and noise up to 10% for Golang. However, CPU use on each node (b + d) follows traffic generation exactly, keeping throughput per CPU constant and showing that the noise is more likely an iperf3 artifact. Note that although the Golang has separate threads for packet handling in both directions, resulting in a slightly higher than 1 core CPU use. In relative terms, the eBPF implementation remains around 14 times faster than Golang for sending traffic. WireGuard throughput (e), on the other hand, is capped throughout the scenario for node 2 (the first client to connect), while the remaining throughput is chaotically divided among the rest of the nodes, varying by up to 50% over a few seconds. WireGuard CPU use (f) shows the disadvantage of a server node for ring (or otherwise decentralized) topologies; it requires 4 CPU cores to handle all traffic, while the other nodes in the “ring” consistently use 2 CPU cores due to network resource contention, which is significantly more than in the straightforward star topology scenario.

Finally, WireGuard UDP throughput (g) is included for illustrative purposes. Due to the nature of UDP, for all alternatives the throughput for each node is almost exactly enough

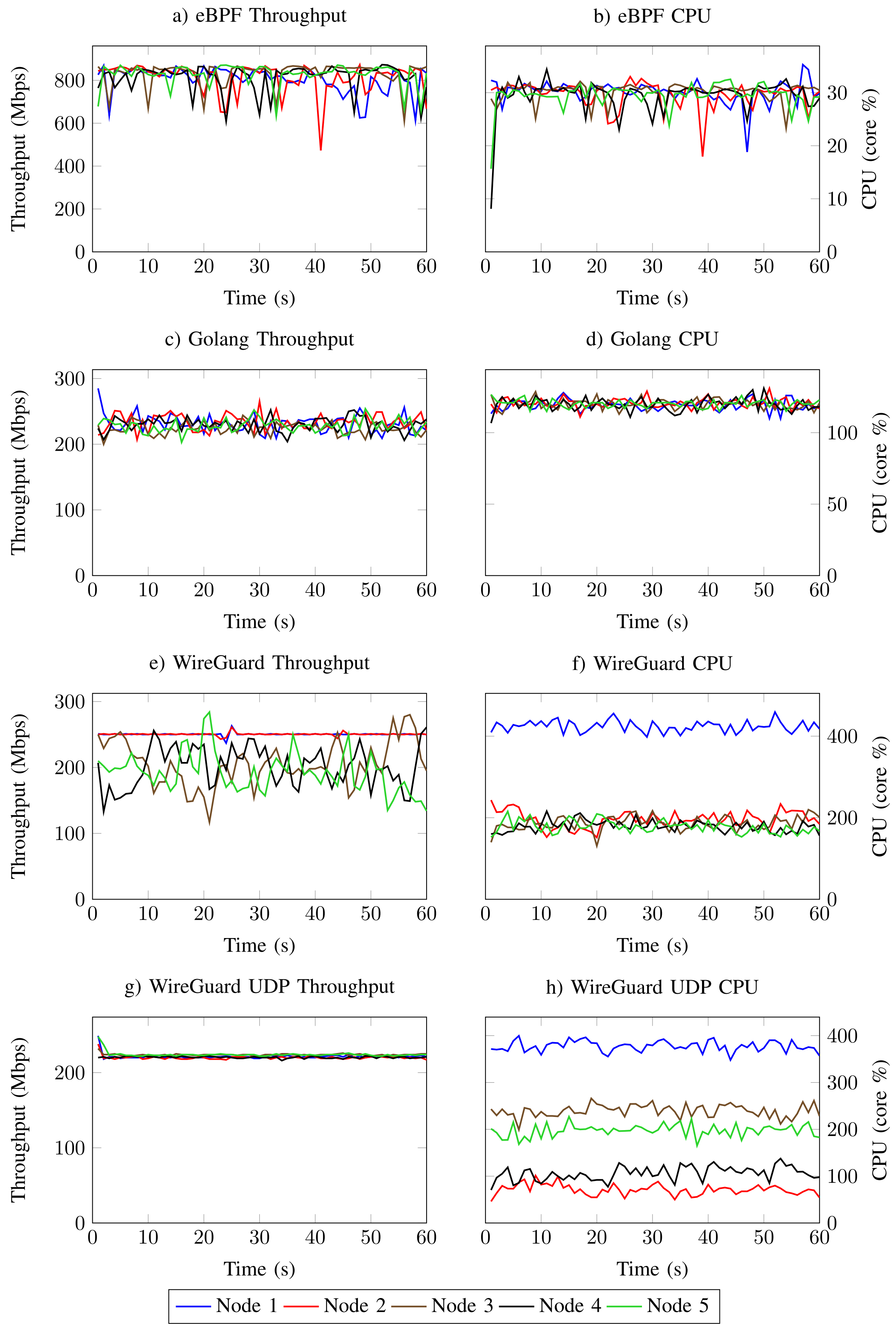

Fig. 12: Throughput and CPU use overview for all nodes in 5 node ring topology, for TCP traffic using each evaluated alternative. WireGuard UDP is included as an example of resource contention.

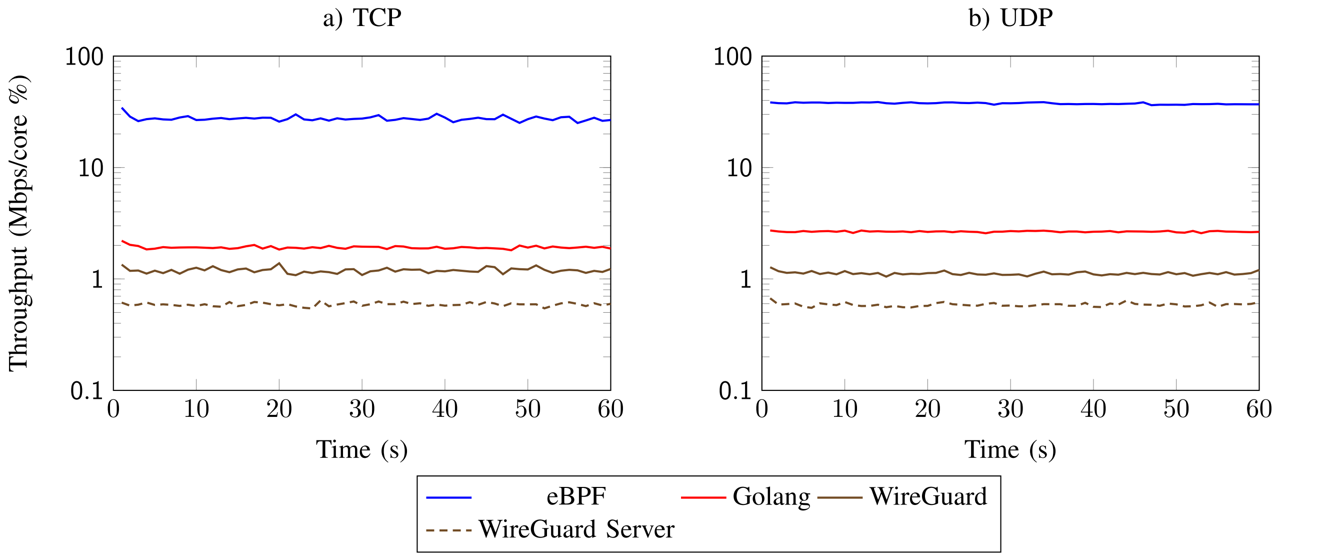

Fig. 13: Throughput per CPU core % in 5 node ring topology. Note that each node is simultaneously traffic sender and receiver, averaging out the relative performance of both roles; "WireGuard Server" indicates the VPN server node.

to saturate a gigabit connection. In the case of WireGuard however, this comes down to 220 Mbps per node, although despite almost identical throughput per node, CPU use (h) varies nearly 400% even between client nodes. This effect persists throughout reruns of the scenario, likely making this another manifestation of resource contention. UDP performance of the eBPF and Golang implementations is unremarkable and nearly identical to TCP performance, barring a slight scaling of CPU use.

The average throughput per CPU use of all nodes is shown in Fig. 13, illustrating that in decentralized topologies both Warrens implementations pull ahead of (VPN) solutions based on server nodes. Despite its userspace implementation, the Golang implementation is around 3 times faster than the WireGuard server node for TCP, and 5 times faster for UDP.

D. Cross Comparison

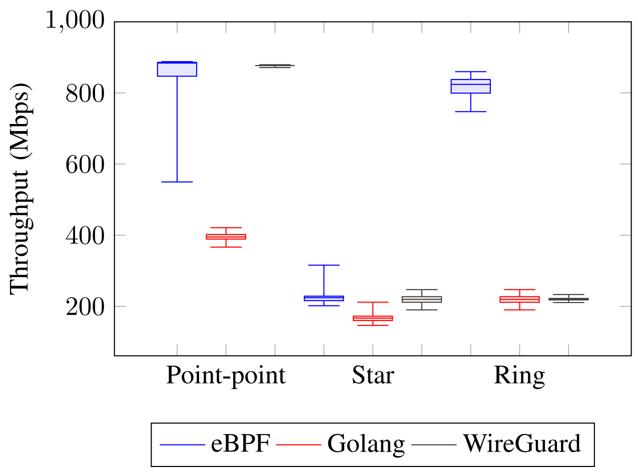

Fig. 14: Cross-scenario TCP throughput comparison.

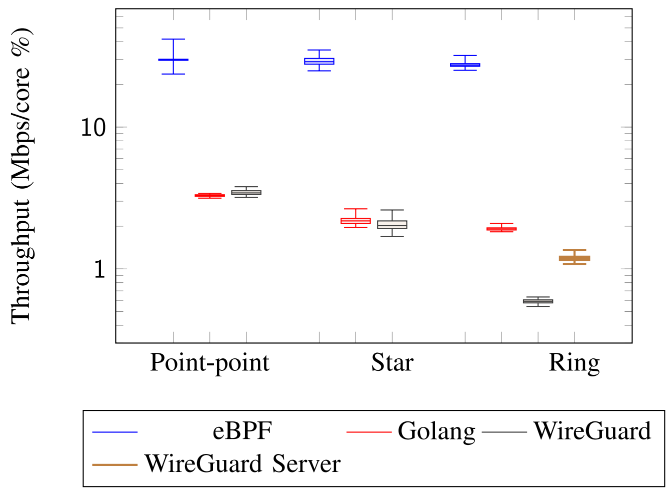

Fig. 15: Cross-scenario comparison of TCP throughput per CPU core % of traffic generators. "WireGuard Server" indicates the VPN server node in the ring topology.

Fig. 14 shows the cross-scenario TCP throughput for all alternatives. As with each individual scenario, relative differences are most important. For point-to-point, WireGuard throughput is consistently high, and more stable than the eBPF implementation. Moving to the star topology, performance differences are compressed as the server node becomes a bottleneck, but in this scenario all alternatives are equally stable. The advantage of decentralization becomes apparent in the ring scenario, where the eBPF implementation returns to nearly the same throughput as the point-to-point scenario, while the Golang implementation throughput is halved due to CPU limits. WireGuard, however, falls to the level of the Golang implementation because its server node must handle all traffic in the ring. This difference is even more pronounced in Fig. 15 when comparing throughput per CPU use for traffic

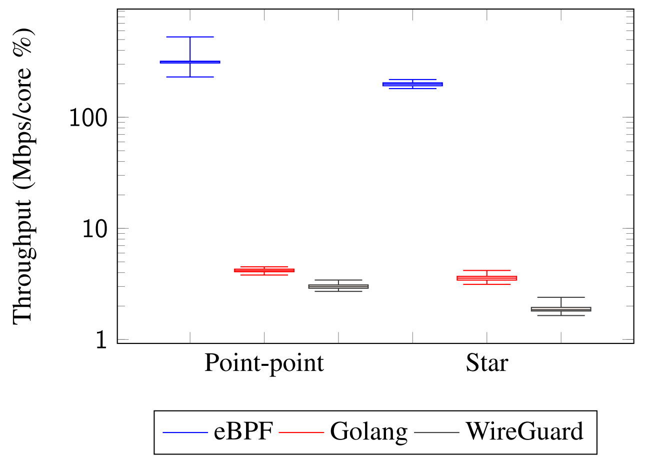

Fig. 16: Cross-scenario comparison of TCP throughput per CPU core % of traffic receivers.

generating nodes; in the point-to-point scenario WireGuard is more or less on par with the Golang implementation, whereas its performance grows progressively worse throughout the scenarios, being only 20% as fast as the Golang implementation in a ring topology. While the Golang implementation itself also slows down around 30% in the star and ring scenarios, this is solely because the evaluated version is double threaded (i.e. one thread to send traffic and one thread to receive traffic) and these scenarios on average generate more traffic per node. The eBPF implementation has nearly identical performance in all scenarios, showing no scaling effects or bottlenecks. Fig. 16 shows the CPU-relative performance for nodes which receive traffic, albeit only for the point-to-point and star topologies. However, as Golang and WireGuard performance is similar to traffic generation, eBPF performance improves an order of magnitude due to the more efficient inbound program.

E. ARM Performance

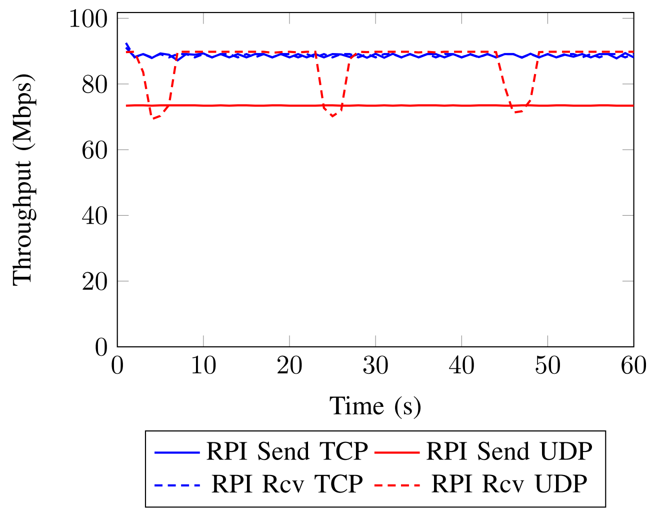

Fig. 17: ARM point-to-point bulk TCP/UDP throughput.

This subsection presents the Raspberry Pi evaluation results for the eBPF implementation, giving an indication of Warrens performance on edge devices. As Fig. 17 shows, Warrens can saturate the Model 3B+ connection using both TCP and UDP, although when sending UDP traffic there are periodic lapses of around 20%. While the results seem to indicate slower reception of UDP traffic, Fig. 18 shows this is not the case; Warrens only uses around 14.5% of a single core to send UDP traffic, and given a suitable traffic generator could saturate the 100 Mbps interface. Taking into account the throughput difference, TCP traffic uses slightly more CPU than UDP in both directions, and receiving traffic is similarly efficient as it is on x64 machines, using less than 1% for UDP and 1.9% for TCP.

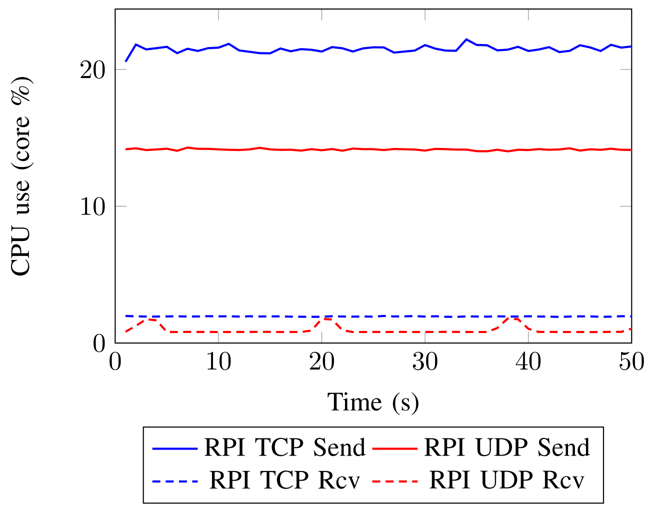

Fig. 18: ARM point-to-point bulk TCP/UDP connection CPU load.

Fig. 19, directly compares CPU-relative performance between x64 and ARM. For both TCP and UDP, receiving traffic is only around 5 times slower on the RPI than it is on the x64 server, despite a far less powerful CPU. Sending traffic, however, is slightly more CPU intensive and is 7-10 times slower. However, edge devices generally do not generate large amounts of traffic, and even when saturating the RPI interface Warrens only requires 4-5% of the total CPU power. Finally, the periodic UDP performance lapses are more pronounced relative to CPU use, although no explanation was found in the eBPF implementation.

F. Memory Use

Fig. 20 shows the memory use of each alternative on x64. Note that nearly all of the memory use is because of container management through containerd; the eBPF and WireGuard options have essentially zero overhead during the entire scenario runtime. Only the Golang implementation, running in the same process, shows an elevated memory use. However, this is limited to around 11 MiB to manage interfaces and traffic, and due to decentralized implementation effectively scales $O(1)$ with the number of globally active Warrens nodes.

VII. Discussion

The main performance characteristics of Warrens are illustrated by the point-to-point and IoT scenarios. As Warrens

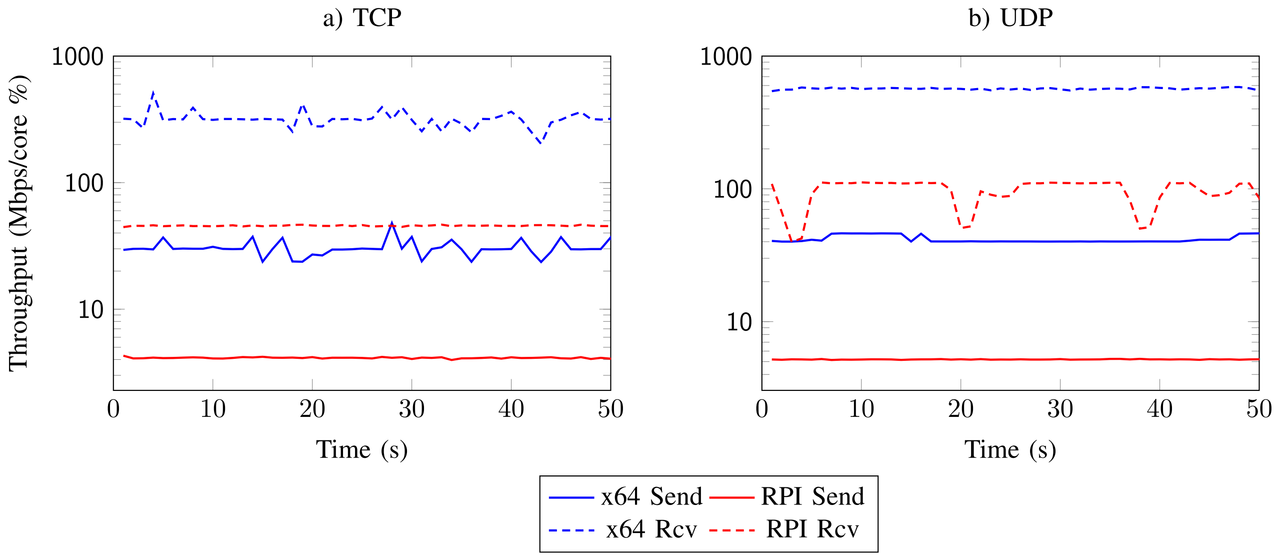

Fig. 19: ARM throughput per CPU core % for point-to-point bulk traffic.

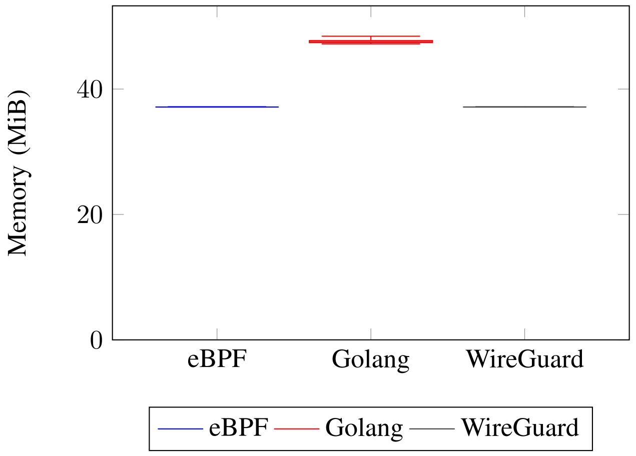

Fig. 20: Memory use during star topology evaluation. Note that because WireGuard and the eBPF solution both run in the kernel, their memory overhead is essentially zero.

does not encrypt traffic itself and is thus not directly comparable to WireGuard, the eBPF implementation is currently up to two orders of magnitude faster than WireGuard, and estimated to be on par if a comparable encryption scheme is implemented. The current implementation requires as little as 2% of total CPU power to saturate a gigabit connection. The Golang implementation is significantly slower, requiring up to 25% total CPU use for gigabit traffic. However, this implementation is only included for compatibility reasons, sacrificing performance to enable Warrens on older and limited devices.

The comparison between a star topology and ring topology illustrates the scalability of Warrens, showing that unlike a network architecture with centralized components (i.e. WireGuard server), Warrens performance does not drop with the number of independent nodes in a decentralized container network; only the local density, specifically the number of nearby nodes that actually exchange traffic with a common node, has an impact on the performance of that common node. The evaluations reflect this, as not only WireGuard server performance drops linearly with network size, but client CPU use increases as a result of resource contention. Warrens throughput, however, remains the same as in the point-to-point scenario, barring a small reduction for simultaneously sending and receiving traffic.

Evaluation results consistently show that the eBPF implementation is particularly efficient for received traffic, as it only needs to cut off the headers of received packets in-memory. As a result, the eBPF implementation server is the only one in the star topology which is more efficient than its clients. Although technically CPU load scales with $O(n)$ for all options, nodes in container networks that primarily receive traffic could benefit greatly from running this implementation.

Warrens is shown to work on a Raspberry Pi 3B+, saturating the hardware network interface with only 4-5% total CPU power when sending data to other devices. While the Golang implementation has a relatively high 11 MiB memory overhead, it remains a fallback option. Additionally, this overhead is measured on x64; it is likely to fall below 10 MiB on an ARM device due to compiler differences. The 8 KiB overhead of the eBPF implementation is negligible.

The evaluation does not cover setup times, as they are essentially zero. Rather, Warrens examines every packet independently and determines ad-hoc where it should be sent, assuming the target container is in a known subnet.

Finally, Warrens is edge-focused by design, allowing nodes behind NAT or firewalls to join a cluster with minimal impact on other traffic. By tracking public IP addresses, nodes can connect through NAT and the SoSwirly discovery mechanism can be used to keep NAT active. Both the Golang and eBPF implementations only intercept incoming traffic on port 31337, and other container traffic is limited to dedicated network interfaces. Due to eBPF restrictions, the latter implementation is technically more secure as it cannot manipulate memory outside the packet or call unrelated system functions.

VIII. Future Work

Future work includes the integration of the service discovery and DNS functionality mentioned in Sections III-D and III-E; while the SoSwirly discovery algorithm has been proven to work, the evaluations in this manuscript are performed with static node catalogs as to not let node discovery interfere with performance measurements.

A further challenge is evaluating how Warrens can be made compatible with existing container networking standards, possibly for integration with Kubernetes. This includes possibly running side-by-side with a classical CNI plugin, or allowing centralized control to impose a certain structure on Warrens while mostly maintaining the independence of node discovery and IP range assignment.

To improve Warrens security both at the network and application levels, an attestation mechanism should be devised to periodically authenticate nodes after initial discovery, as well as verifying their catalogs of hosted services. Parameters for attestation can include device safety features (e.g. encryption options, hardware security enclaves), (hardware) reliability and past node behavior. Additionally, encryption options can be added to secure Warrens traffic, which can be enabled between a pair of nodes if they support the required hardware and kernel options.

Finally, Warrens performance can be greatly improved by removing the UDP header and thus the need for a checksum. Ideally, only an IP-in-IP header would be required, although compatibility with a wide range of routers should be considered in terms of routing options (i.e. NAT and port forwarding in the current case of UDP).

IX. Conclusion

Existing container networking solutions are increasingly demonstrating rigidity in meeting the needs of edge computing environments. This manuscript presents Warrens as a solution for decentralized, scalable container networking in the edge, using connectionless tunnels. The architecture of Warrens is described in depth, with possible additions for node and service discovery. Warrens has been implemented in two variants, one using Golang for wider-compatibility and another using eBPF for better performance. Details of the Golang and eBPF implementations are provided, highlighting the trade-offs in their performance. Warrens has been evaluated experimentally over four scenarios, comparing both implementations to WireGuard, a centralized VPN alternative solution. Evaluation results show that although Warrens does not encrypt traffic like WireGuard, the relative conclusions can be drawn that the eBPF Warrens implementation is highly performant, especially for received (i.e. downstream, response) traffic, and that it would be on par with WireGuard performance when implementing encryption. The Golang implementation is shown to be a suitable, but significantly slower alternative for devices that do not sufficiently support eBPF, while both implementations are highly scalable compared to classical VPN solutions such as WireGuard. Finally,future work has been discussed, centered around DNS, standardization and reconciliation with normal CNI operation, as well as security enhancements and performance improvements.

References

[1] G. Baldoni, L. Cominardi, M. Groshev, A. De la Oliva, and A. Corsaro, “Managing the far-edge: are today’s centralized solutions a good fit,” *IEEE Consumer Electronics Magazine*, 2021.

[2] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, “Extend cloud to edge with kubeedge,” in *2018 IEEE/ACM Symposium On Edge Computing (SEC)*. IEEE, 2018, pp. 373–377.

[3] “kubernetes.io - considerations for large clusters,” Sep. 2023. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/

[4] “High performance cloud native networking (cni),” May 2024. [Online]. Available: https://cilium.io/use-cases/cni/

[5] “Ipv6 addresses for clusters, pods, and services,” May 2024. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/cni-ipv6.html

[6] L. Kong, J. Tan, J. Huang, G. Chen, S. Wang, X. Jin, P. Zeng, M. Khan, and S. K. Das, “Edge-computing-driven internet of things: A survey,” *ACM Computing Surveys*, vol. 55, no. 8, pp. 1–41, 2022.

[7] T. Goethals, D. Kerkhove, B. Volckaert, and F. D. Turck, “Scalability evaluation of vpn technologies for secure container networking,” in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–7.

[8] T. Goethals, F. D. Turck, and B. Volckaert, “Self-organizing fog support services for responsive edge computing,” *Journal of Network and Systems Management*, vol. 29, no. 2, jan 2021.

[9] J.-G. Cha and S. W. Kim, “Design and evaluation of container-based networking for low-latency edge services,” in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, 2021, pp. 1287–1289.

[10] J. Yoon, J. Li, and S. Shin, “A measurement study on evaluating container network performance for edge computing,” in *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2020, pp. 345–348.

[11] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, “Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications,” *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. [Online]. Available: https://doi.org/10.1145/3371038

[12] J. Nam, S. Lee, P. Porras, V. Yegneswaran, and S. Shin, “Secure inter-container communications using xdp/ebpf,” *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 934–947, 2023.

[13] F. Parola, L. D. Giovanna, G. Ognibene, and F. Risso, “Creating disaggregated network services with ebpf: the kubernetes network provider use case,” in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 254–258.

[14] “cilium.io - what is cilium?” Sep. 2023. [Online]. Available: https://cilium.io/get-started/

[15] K. Subratie, S. Aditya, and R. J. Figueiredo, “Edgevpn: Self-organizing layer-2 virtual edge networks,” *Future Generation Computer Systems*, vol. 140, pp. 104–116, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22003235

[16] R. Figueiredo and K. Subratie, “Demo: Edgevpn.io: Open-source virtual private network for seamless edge computing with kubernetes,” in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020, pp. 190–192.

[17] M. AL-Naday, N. Thomos, J. Hu, B. Volckaert, F. de Turck, and M. J. Reed, “Service-based, multi-provider, fog ecosystem with joint optimization of request mapping and response routing,” *IEEE Transactions on Services Computing*, pp. 1–15, 2022.

[18] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, “A survey on observability of distributed edge & container-based microservices,” *IEEE Access*, 2022.

[19] Z. N. Abdullah, I. Ahmad, and I. Hussain, “Segment routing in software defined networks: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 464–486, 2019.

[20] J. L. Shah and J. Parvez, “Optimizing security and address configuration in ipv6 slaac,” *Procedia Computer Science*, vol. 54, pp. 177–185, 2015.

[21] T. C. Priyadharshini and D. M. Geetha, “Efficient key management system based lightweight devices in IoT,” *Intelligent Automation & Soft Computing*, vol. 31, no. 3, pp. 1793–1808, 2022.

[22] W. Yao, C. Jiang, X. Luo, and Z. Bao, “A fnv based ipv6 address autoconfiguration scheme for power iot sensory device,” in *2022 2nd International Conference on Networking Systems of AI (INSAI)*, 2022, pp. 216–221.

[23] T. Goethals, D. Kerkhove, B. Volckaert, and F. D. Turck, “Scalability evaluation of VPN technologies for secure container networking,” in *2019 15th International Conference on Network and Service Management (CNSM)*. IEEE, oct 2019.

[24] T. Goethals, F. D. Turck, and B. Volckaert, “Extending kubernetes clusters to low-resource edge devices using virtual kubelets,” *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2623–2636, oct 2022.

[25] “Rfc 4193 - unique local ipv6 unicast addresses,” Oct. 2005. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4193#section-3

[26] A. DasGupta, “The matching, birthday and the strong birthday problem: a contemporary review,” *Journal of Statistical Planning and Inference*, vol. 130, no. 1-2, pp. 377–389, 2005.

[27] “Coredns: Dns and service discovery,” Sep. 2023.

[28] IDlab, “Virtual wall,” Apr. 2020. [Online]. Available: https://www.ugent.be/ea/idlab/en/research/research-infrastructure/virtual-wall.htm

[29] F. De Santis, A. Schauer, and G. Sigl, “Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 692–697.