Secure, Fast, Location-Independent Routing for the Global Data Plane

Rahul Arya Praveen Batra Thanakul Wattanawong

December 2021

Abstract

This paper describes the implementation of a fast, flexible location-independent routing system for the Global Data Plane. The system contains many features of a complete GDP switching network including location-independent routing, certificate chaining, and a routing information base. Other features were also implemented for security and performance including packet-level encryption that can be used to build full dTLS support at each hop, and almost lock-free caching of the forwarding table. Benchmarks indicate that the system has superior performance both in general and compared to previous implementations, and we propose that our implementation can be naturally extended to support other novel features such as arbitrary trust domains for routing.

1 Introduction

The Global Data Plane (GDP) [1] is a system for secure distributed computing. An essential aspect of the GDP is its ability to utilize fog and edge devices to distribute computation. While this allows the GDP to use a wider variety of compute and data sources than traditional cloud or distributed compute systems, it also means that encrypting data, verifying computation and hiding information are essential to prevent attacks on the data plane. Existing components for this include DataCapsules, which are cryptographically verified bundles of data, and secure code execution on untrusted nodes within secure enclaves such as Intel SGX.

Location-independent routing ties all these components of the GDP together. Each node in the network is addressed by its GDPName, a hash of its metadata, rather than its IP address. This allows nodes to be relocated physically while retaining the same address. For instance, a DataCapsule addressed by a particular GDPName could move to a different storage provider without requiring any action on the part of its clients. Because a single computation might use data from multiple edge devices and run compute across multiple devices, it is essential for all of these devices to be able to communicate through a routing network. This motivates the need for an overlay network on top of IP that allows nodes in the GDP to communicate using their GDPNames.

An important property of this system is to ensure that packets are correctly routed to their destinations, even in the presence of adversarial nodes in the system. For instance, it should not be possible for a malicious node to claim a particular GDPName and intercept or drop all traffic intended for that target. To enforce this, the basic idea is to use GDPNames as a root of trust, by including the public key of each node as part of the metadata whose hash yields the GDPName. Then, for a server to advertise its ownership of a GDPName, it must present a certificate signed by this public key proving that it is authorized to do so, in a process known as *secure delegation*. By chaining such certificates, and verifying them at each router in the network, honest participants can communicate even in the presence of malicious participants.

A secondary property is the reduction of metadata leakage. For instance, it should be difficult for an adversary monitoring the incoming and outgoing traffic of an honest router, but with no access to the Routing Information Base (a distributed system that communicates with GDP routers to determine the path of packets in the system), to determine which GDPNames it is communicating with, or how they map to IP addresses.

The routing system must be highly performant and minimize overhead due to the amount of data being transferred and the complexity of modern network paths. Internet of Things (IoT) and edge or fog computing demands mean the system should maintain its efficiency at a potential scale of up to 10^{12} nodes.

Our goal was to build a prototype of software-based switches and a Routing Information Base (RIB) with faster performance than existing implementations that can naturally be extended to support advanced features needed for GDP routing. This switch should be able to run on commodity hardware in order to facilitate the growth of the GDP network. To that end, we used the Rust programming language [2] and the Capsule framework [3]. Rust is a highly performant compiled language that also includes strong type annotations and compiler guarantees, helping make our code more robust and secure than a C implementation would be while still achieving faster performance than interpreted languages like Python. Capsule is an open-source networking framework built on the Data Plane Development Kit (DPDK) [4], which is a kernel-bypass networking solution that can be used to implement highly performant packet processing routines.

2 Related Work

2.1 Existing GDP implementations

2.1.1 The Python-based GDP research prototype

An existing GDP research prototype implementation exists in Python, which implements a large number of features including GDPNames derived from the hash of metadata, the *AdCert* and *RtCert* certificate delegations, an RIB implementation which they call the GLookupService, and so forth [1, 5]. The author of this research prototype have done extensive performance evaluations and comparisons, and a large part of our implementation relies on the models they have proposed. However, the switch provided in this implementation has very poor throughput especially at low packet sizes, making it unsuitable for production level use. Our goal is to provide a performant design and implementation of a subset of this prototype's features, and add additional features of our own contribution.

2.1.2 Click-based production prototype

The Click-based [6] production prototype is written in C/C++, and supports a subset of the features in the Python implementation [7]. Critically, though, this system focuses on location-independent routing, and does not support certificate validation, meaning that it does not verify resource ownership before routing packets, a key security requirement of our system. The original authors benchmarked the system primarily on latency, although throughput results are also available in a separate paper [8]. This system is production quality and, at the time of writing, is the currently deployed version. We have used this implementation as a benchmark for some of our performance comparisons, and we generally exceed their results by a significant margin.

2.2 Other routing networks

2.2.1 Dam and Palmskog

An existing system for location independent routing is presented in [9]. Unlike our approach, this system does not use a Routing Information Base (RIB) and uses decentralized communication between individual switches. In addition, while their system can be built on IP, it is designed to also be possible to implement at a lower level, bypassing IP altogether.

While this approach provides some advantages, such as having fewer possible sources of failure, having to gradually propagate network topology changes to different nodes introduces a possible inefficiency compared to our approach, where the RIB serves as the most up-to-date source of truth. In addition, this paper focuses on describing the theory of the system without an implementation, meaning that it is unclear how practical the routing system described would actually be to implement. In contrast, our approach is simple and can achieve high performance, as we demonstrate with our Rust implementation.

2.2.2 The Tor network

The Tor routing system [10] is a encrypted network system that hides node IPs (in the case of "hidden services") and uses onion routing. In onion routing, clients choose the path to communicate across, relying on the obfuscation of their traffic amongst that of other network participants to reduce metadata leakage. In general, participants in Tor accept that they may be communicating across malicious routers, and rely on there being at least one "honest" participant in their chain (or multiple non-cooperating malicious participants) to ensure anonymity.

In contrast, our approach is designed to work with trusted routers. However, we also ensure that packets only pass through routers specifically delegated from the source or destination addresses, so any malicious routers cannot see traffic that they are not authorized to view, reducing information leakage.

2.2.3 Lira

The location independent system Lira [11] focuses on optimizing content delivery, so it is fairly different from our system: names change rather than being hashes of metadata, and the content has at least one provider whose IP is known. Lira does not seek to hide information to the same degree, which removes some performance constraints on our system, but is also more limited in where it can be deployed (content delivery).

The GDP is more flexible and supports more use cases because of its encryption and security, but this comes at performance costs. We evaluate these costs in our Results section when we compare throughput and forwarding rate with and without encryption.

2.2.4 TARP

The TARP system [12] does not deal with securing communications the way our system or Tor do, but it does attempt to create a scalable system of routers. Like [9], it does not use a central Routing Information Base; instead, it uses a distributed system where individual routers store paths of a certain length or connect to other routers to get larger paths.

Our system can theoretically support this kind of scale in a more centralized fashion with hierarchical trust domains, each of which is managed by an RIB. The advantages of a centralized approach come in simplicity and information hiding from individual routers, while the downsides include more vulnerability in the case of the RIB failing or being compromised by an attacker.

3 Design

Our system is heavily based on the design presented in Chapters 5 and 6 of [1] - we first provide a brief summary of that design, and then elaborate on the extensions we made to it.

3.1 Base Design

We illustrate the existing design through the trace of a hypothetical packet through the network. [1] provides a formal description of the protocol, along with a rough security proof.

Consider a source server A connected to a GDP switch R_A, and a destination B connected to a switch R_B. These four machines all have private/public key pairs, and are each addressed by their GDPName, a hash of their public keys (and potentially other metadata).

For A to communicate to B, it sends a packet to its switch R_A indicating that B is its final destination. Along with the data payload, this packet contains a *routing certificate* (`RtCert`) authorizing R_A to send packets on behalf of A, signed by the private key of A. This is known as *advertising* the GDPName A to R_A.

R_A validates this certificate, then contacts the RIB to search for the switch associated with the server B. The RIB replies with R_B, along with a `RtCert`$(B \to R_B)$. The RIB also returns a `RtCert`$(R_B \to$ `IP`$)$, where `IP` is the current IP address of the switch R_B.

R_A validates these certificates using the known public keys of B and R_B, then generates a new certificate `RtCert`$(R_A \to R_B)$ authorizing R_B to forward packets on behalf of R_A for the purpose of relaying packets from A to B. We call this the *delegated advertisement* of address A by R_A to R_B. This certificate is appended to the packet, which is then forwarded to R_B.

R_B verifies the certificate chain `RtCert`$(A \to R_A)$ and `RtCert`$(R_A \to R_B)$ bundled with the packet. Since it is connected to B directly, it can directly forward the packet to the known IP address of B.

3.2 Obtaining Metadata

Each node needs to verify the certificates associated with the packet being transmitted. For instance, R_A needs to verify certificates signed by A, B, and R_B. To do so, they need to know the public keys corresponding to these nodes.

One solution, used in the Python GDP implementation in [1], would be to include the public keys and other metadata as part of the certificate. This is not self-referential, because the GDPName of each node is a hash of its metadata, including its public key. So while knowing a GDPName is not sufficient for us to know its public key, we can *verify* that a public key corresponds to a given GDPName by comparing hashes.

Instead, our implementation stores the metadata for each node in the RIB, then caches it on each router. When a router needs to validate a certificate owned by some GDPName X, it checks its local cache for the metadata associated with X. If this metadata is present, it continues as normal. Otherwise, it sends a request to the RIB for the metadata associated with X. This gives us a reduced packet size (since the certificates are smaller) at the expense of increased latency on initial connections, since we need to wait for the RIB to populate the router cache.

3.3 Packet Rejection

It is often the case that a router does not have the information it needs to verify or forward a packet, as it may be waiting on a response from the RIB. To handle this, one solution (used by the Python GDP router) would place the packet on a queue and forward it once the RIB response arrives. However, this approach has issues under high load, since these queues can grow rapidly, possibly exceeding the memory limit of the router. It also adds significant complexity to the router, since upon receiving a RIB response, it would have to check all waiting queues (across all cores) and determine which packets it is now capable of forwarding.

Instead, our implementation offloads this responsibility to the client. If the router receives a packet that it cannot immediately forward, the packet is dropped and a `NACK` is sent back to the origin requesting a retry after a short period of time. Simultaneously, a request is sent to the RIB for the missing data, so that when the origin retransmits the packet, the router has the data it needs to forward it.

One consequence of requiring clients to handle the retransmission logic is that they must receive the `NACK` packets to know to stop transmitting. These packets may need to be forwarded through intermediate GDP routers to get to the client. For instance, if we are sending a packet along $A \to R_A \to R_B \to B$, and B rejects an incoming packet, the `NACK` needs to be forwarded by R_B back to R_A, and finally back to A. To do this, each switch maintains a `nack_cache` storing all recently active connections and the IP address of their "previous hop". For instance, here R_B would record that packets with source GDPName A were received from node R_A (as a tuple (A, R_A)), so `NACK` packets with a source A should be sent to R_A. A tuple (X, R_Y) is only inserted into the `nack_cache` of a router R after R_Y has successfully advertised X to R, so it is not possible to use NACK packets to bypass the guarantees offered by the certificate system.

NACKs are also used to indicate an overlarge message payload size. Even if the PDU is small enough at the first hop, it may become too large after certificates are appended in subsequent hops, so the client must be instructed to reduce the payload size. This design is spiritually similar to the Path MTU Discovery technique used at the IP layer, but takes place at the level of the GDP overlay network, with NACK packets used as the analogue of ICMP packets.

3.4 Hop-to-hop Encryption

One risk with the system as presently described is metadata leakage across hops. An external observer of the network could trace the path of a packet through the network, and so recover the source IP address of some GDPName A even if all its packets went through a router R_A. To address this, each pair of directly communicating routers first participate in a handshake to establish a shared temporary key, then decrypt and re-encrypt using the appropriate keys the GDP headers and data payload before forwarding the packet, following the dTLS protocol. This also allows us to avoid the GDP handshake described in [1], since dTLS protects us against replay attacks. We discuss the performance implications of this in the next section.

3.5 Certificate Caching

A key optimization made is to cache valid certificate chains, so that we do not have to validate them and generate new routing certificates for every packet that passes through the switch. Consider a stream of packets passing through $A \to R_A \to R_B \to R_C \to C$. At R_B, when the first packet arrives, we verify that the included certificate chain authorizes R_A to advertise on behalf of A (at least for packets destined for C, and for a limited timespan). We then cache the tuple (A, R_B), along with the generated RtCert authorizing R_C to advertise on behalf of R_B for this purpose. When subsequent packets pass through R_C along the same path, we can then skip certificate validation, and just append the new certificate to the packet before re-encryption and transmission. This is because dTLS ensures that packets claiming to have previously passed through R_B actually have, and the existing certificate chain (and the new certificate) is therefore still valid. As a consequence, we remove many expensive cryptographic operations from the "hot path" of the router - unlike the decryption and encryption in dTLS, which uses AES and so is fast on modern hardware, our certificates use public-key cryptography, which is much slower and so should be avoided when possible.

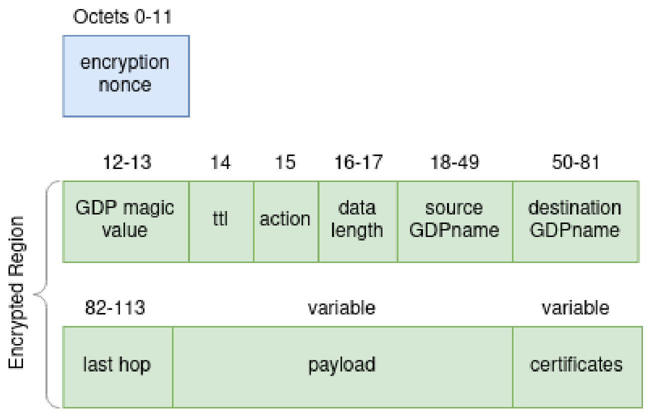

Figure 1: GDP PDU

3.6 Node Discovery

When a client node A connects to the system, it first broadcasts a packet within its local network to search for available GDP routers. After some router R responds, they perform a dTLS handshake to establish a secure connection and share their GDPNames. Then, the node generates a certificate RtCert($A \to R$) delegating its name to its router for a specific duration (say, 10 minutes). The router stores this certificate and publishes it to the RIB, so that other routers wishing to communicate with A know to go through R. If A is unable to find any routers on its local network, it is also capable of contacting a "root" GDP router at a hardcoded IP address and delegating its name to that router.

4 Implementation

4.1 GDPinUDP Protocol Data Unit

The fields in our GDP PDU, which is transported within a UDP datagram, are as follows:

- **Encryption nonce:** A 96-bit randomly generated nonce for the AES-GCM encryption process.
- **GDP magic value:** A 16-bit magic value used to distinguish GDP packets from regular UDP packets in order to avoid the overhead of decryption failure.
- **TTL:** GDP level time-to-live that limits how many router hops a packet can traverse.
- **Action:** Denotes a specific GDP action such as RIB querying, payload delivery, Nack, etc.
- **Data length:** A field describing the length of the payload, used to enable appending certificates along hops without parsing the entire payload.
- **Source GDPName:** GDPName of the source

- **Destination GDPName:** GDPName of the intended destination
- **Last hop GDPName:** GDPName of the previous hop, used to look up the proper verification data at each router.
- **Payload:** The payload being transmitted.
- **Certificates:** A binary-serialized list of certificates that are required to verify the provenance and transit route of this packet.

See Figure 1 for the layout of these fields.

The GDP packet including headers is fully encrypted using AES-GCM, with the randomly generated nonce prepended to the encrypted portion. Currently, our software switch utilizes pre-shared keys, but we plan to build on top of this system to implement full dTLS. The current implementation is sufficient to demonstrate the impact of encryption on switching performance.

4.2 Capsule/DPDK Runtime

We chose to use the Capsule runtime [3], which is built on top of DPDK [4] mainly for performance and ease of development.

DPDK is a collection of data plane libraries and poll-mode NIC drivers that allow high throughput user-space packet processing. In particular, the runtime uses light-weight Mbufs, which are fixed-size cache-aligned buffers that store packet data. Poll mode drivers generally perform well as they avoid the interrupt processing overhead as in the case of the Linux kernel networking stack, at the cost of saturating the entire CPU core. DPDK drivers also generally implement advanced features if they are supported by the NIC, such as Receive-Side Scaling, which allows packets to be distributed between receive queues, allowing us to scale our implementation to multiple cores. Additionally, we have also modified an existing DPDK-based project [13] to build a baseline software switch to benchmark our implementation against and found it to be very performant.

Capsule is a framework for network function development using abstractions proposed by the NetBricks paper [14]. The NetBricks model provides intuitive high-level abstractions for packet-processing and control flow of packets using user-defined functions. The advantage is superior ease of development while retaining fast packet processing performance. We hope that the intuitive model will facilitate additional development as there are still many features needed to fully realize the vision of the GDP. One limitation of Capsule is that it currently only supports single-segment DPDK Mbufs per packet, which imposes a limitation on our payload size. We plan to address this in the future by exploring alternative avenues such as supporting multi-segment Mbufs or implementing fragmentation at the GDP layer.

One general limitation of this runtime is that kernel bypass requires exclusive access to the NIC, making our implementation difficult to both run and be managed remotely on systems with only one interface as is the case on most consumer motherboards. Possible avenues of further exploration in this area include bifurcated drivers which requires hardware support, virtual adapters plus the use of eBPF and XDP or other kernel packet processing mechanisms, or implementing remote management capabilities directly in the software switch.

4.3 Rust

We chose to implement our switch in Rust for the memory- and thread-safety guarantees without compromising on performance. GDP switches have a nontrivial amount of certificate validation and route lookup logic, which are difficult to correctly implement in languages like C without risking the introduction of memory-safety/security issues. Anecdotally, the existing C implementation of the GDP router (the "production" system) was considered extremely difficult to work with, despite supporting very few features compared to the research prototype, written in Python (for instance, certificate generation/validation was entirely unsupported). Our hope is that our choice of language will make it easy to extend this switch directly, rather than having to first experiment with a separate research prototype.

4.4 Multi-Core Caching

As discussed in previous sections, the GDP router must cache data from the RIB, as well as keep track of the state of recently active connections (for the `nack_cache`). High-throughput routers are typically multi-core, with incoming packets distributed across cores based on a hash of the flow parameters (source and destination IP and ports), known as Receive-Side Scaling. To minimize lock contention, the ideal scenario would be for each core to operate entirely independently with its own caches for certificate, metadata, and connection data.

However, this solution would lead to redundant RIB queries, since each core would have to separately request data even if it were already available to a neighboring core. A naive shared cache, however, would hurt throughput through severe lock contention. For instance, while processing a response from the RIB, we would have to block all cores from processing packets while the shared cache was being written to.

One solution would be to use a lock-free data structure for these caches (typically hashmaps from a GDPName to the appropriate data structure), but we found that this led to significant implementation complexity. Instead, we elected to use a two-level cache. Each core would have its own cache, which it could access without contention from other cores. In addition, there would be a global cache that cores would query upon a local cache miss, before ultimately falling back to an RIB lookup. If data is found in the global cache, it will be mirrored in the local cache for future accesses. Due to RSS, local cache misses that are handled by the global cache will not be common in the first place since most flows will be pinned to a single core, meaning that the majority of cache accesses will ultimately be lock-free.

This design was also easily extendable to support cache expiration, needed since routing certificates have a limited lifespan, and we do not need to cache stale connections indefinitely. Each local core cache is an LRU cache with a fixed capacity, preventing them from growing without bound at the cost of occasionally evicting valid entries. The global cache is a hashmap with an unbounded (or extremely high) capacity. For both the local and global caches, we evict entries lazily - specifically, if we access a key and find an expired value, we delete it from the cache and report a miss. In addition, to prevent the global cache from growing without bound, a single core periodically samples the global cache and deletes expired entries until sampling shows them to have dropped to an appropriately low fraction of entries.

Although this sampling process holds a write lock on the global cache, in practice we found it to not affect throughput significantly, since most of the time each core has the data it needs in its local cache. This is discussed further in the next section.

5 Experimental Results

In this section, we evaluate the performance characteristics of the GDP router implementation on AWS EC2 instances. Our primary success metrics are forwarding rate measured in packets per second, and throughput in megabits/gigabits per second. We further derive synthetic metrics including efficiency of handling RIB queries. We also compare against existing implementations of the protocol where possible, and several software switch baselines. For all experiments, we take the average of the metric over ten seconds to reduce variance.

Our experimental setup consists of four GDP nodes on AWS EC2 in the topology and instance types shown in Figure 2.

The instances were each configured with an Amazon ENA (Elastic Network Adapter) and in a cluster placement group. In this configuration, AWS guarantees up to 25Gbps of multi-flow traffic between any two instances based on the 5-tuple of the UDP header. Since we randomize the source and destination ports at the UDP layer, inter-node GDP traffic is considered multi-flow for these purposes. For DPDK, we use a memory pool size of $2^{18} - 1$ Mbufs, with a cache of 512 elements per core. We configure the port ring buffer queue sizes at 8192 for receive and 1024 for transmit (the maximum supported by the ENA NIC) respectively.

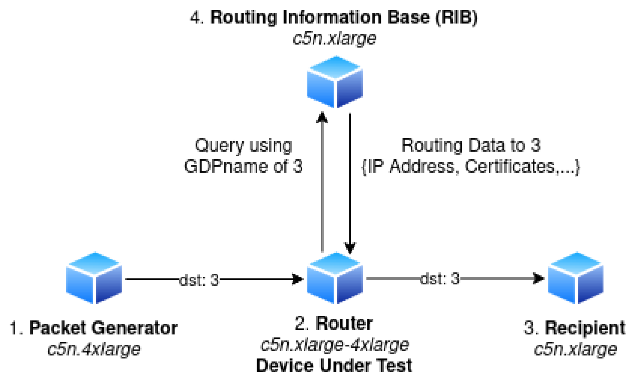

Figure 2: EC2 Instances Layout

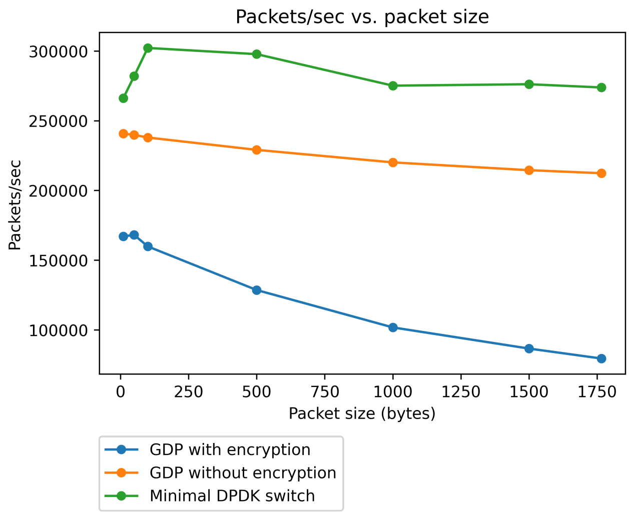

Figure 3: Single-Core Forwarding Performance

5.1 Performance with packet size and encryption

For this experiment, we used a single 4-core c5.xlarge instance for the router node, but only used one core for the switch.

Figures 3 and 4 shows the throughput and forwarding rate of the the GDP router as we vary the packet size and toggle encryption. We also plot the results from a minimal DPDK-based software switch that forwards raw TCP/IP packets to a hardcoded destination without any extra processing. For this experiment we configured the router node as one c5n.large instance with one core pinned to the switch.

As previously mentioned, the Capsule runtime only supports

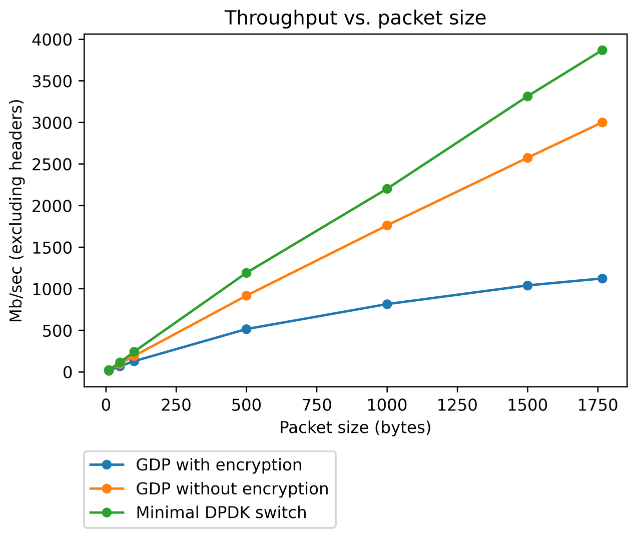

Figure 4: Single-Core Throughput

single-segment DPDK Mbufs. Our implementation uses 2048 byte Mbufs, and we have measured our maximum payload size under this topology at 1766 bytes.

Our implementation achieves a peak throughput of 1.12Gbps with encryption, and 3.00Gbps without by using such 1766 byte payloads. Our unencrypted throughput lags slightly behind that of the minimal DPDK switch, which could be attributed to the extra verification and packet processing overhead in our code. Without encryption, our throughput scales linearly with packet size, and the forwarding rate exceeds 200kpps even at 1776 byte payloads. It can be seen that the performance penalty of encryption increases with packet size, leading to smaller increases in throughput as a result.

To compare with existing implementations, the Click-based production GDP router achieved 500Mbps using 1000 byte PDUs using four cores on the c5.xlarge instance [7]. At 1000 byte payloads, using only a single core, we achieved 814.27Mbps, which is around a 6.5x increase in total throughput normalized by cores. We do note that the c5.xlarge has two physical cores and two hyperthreads, and so the core-to-core comparison is not exactly equal. We suspect the performance difference is due to several reasons. The authors of the Click-based implementation claim their implementation has not yet undergone optimization. Our kernel-bypass implementation uses a highly optimized driver for the ENA interface, which in itself is significantly more powerful than most commodity NICs, and we have configured the runtime specifically for performance. Finally, their implementation seems to saturate at 1Gbps even in the cloud, suggesting some type of limit in the routing path that the authors do not elaborate upon.

In general, the performance results of our implementation are extremely promising for a production use case. We hope to test on a wider variety of systems in the future to establish the feasibility of running GDP routers at the edge and so forth.

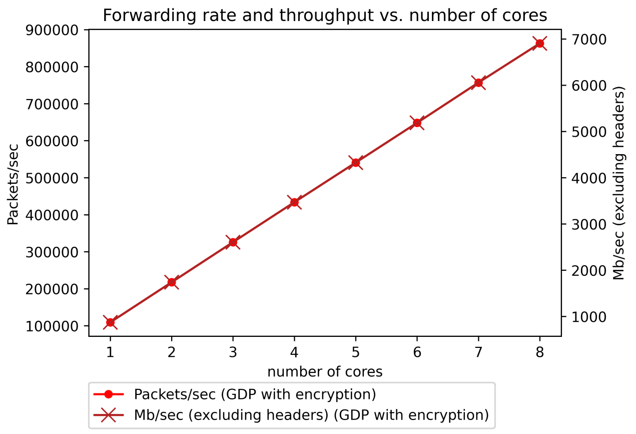

Figure 5: Core Scaling

5.2 Multi-core performance scaling

This experiment used a c5.4xlarge instance, which has 8 physical cores. We scaled the switch to pin to multiple cores in order to measure how packet processing performance scales. We configured the packet generator to send all packets to node 3, and so that route should remain cached in the forwarding table for the duration of the experiment.

Figure 5 shows perfect linear scaling of throughput up to eight physical cores, the maximum on this instance type. We achieved slightly more than 100kpps per core, demonstrating our implementation's ability to scale in switching a single flow with a route that is already in the forwarding table.

We experimented with including the other 8 hyperthread siblings, but observed extremely erratic performance and no notable performance increase. Thus we conclude that hyperthreading is generally of little benefit in our implementation. This is generally concordant with recommendations for Network Function Virtualization (NFV) systems from vendors such as Red Hat [15].

5.3 Multi-core performance with forwarding table misses

This experiment also used the same c5.4xlarge instance for the switch. Forwarding table misses, or RIB misses, refer to a case when the packet's destination address is not in the local forwarding table. Such packets generate two packets, a Nack back to the sender, and a RIB query, which in itself will generate an RIB response that must be added to the forwarding table at some point in the future. This modification of the forwarding table acquires an exclusive write lock, which can

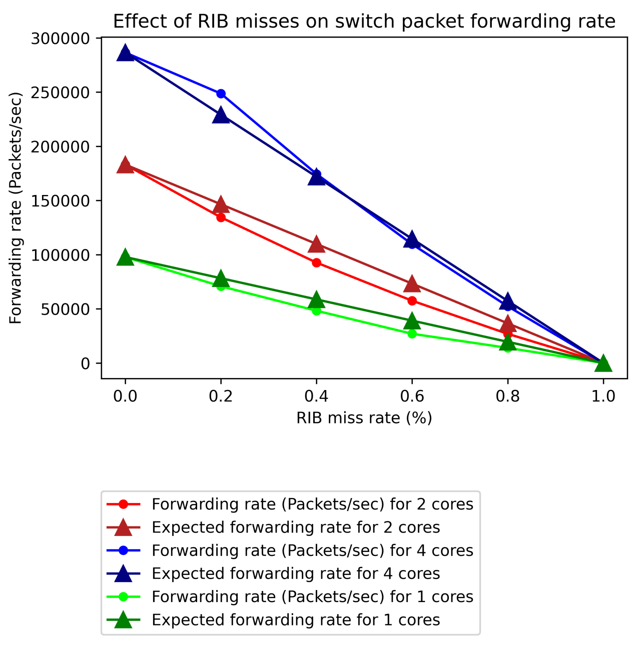

Figure 6: Forwarding Rate with RIB Miss

have adverse impacts on read throughput. Forwarding table misses will be common in the GDP network as routes are mapped into the network lazily, and so a performant architecture is critical.

In order to compute how efficient we are at handling forwarding table misses and operations on the local forwarding table, we generated a workload to flood the switch with packets destined for node 3, but with a proportion destined for random GDP addresses, which causes forwarding table misses. The RIB was specially configured in this case to return a null route as to trigger writes to the forwarding table. For this experiment, we measure the forwarding rate at the receive side of node 3 in order to observe how many packets were forwarded by the switch.

Figure 6 illustrates how our implementation's forwarding rate scales with forwarding table miss rate. The expected forwarding rate for each case is derived by multiplying the hit rate with the forwarding rate at zero miss rate. As per the graph, under most cases the forwarding rate to known destinations only differs minimally from the expected value, and by no more than 10% even at very high miss rates. In general, by using a per-core cache we expect next-hop lookups to be resolved without locking, and so this benchmark measures the impact of write lock contention on the forwarding table. We limited the switch to four cores as we were unable to accurately measure receive rate at node 3 above this number, and we surmise this is due to the comparatively smaller instance type. In general we expect our results to scale for a reasonable number of cores, indicating that our implementation can

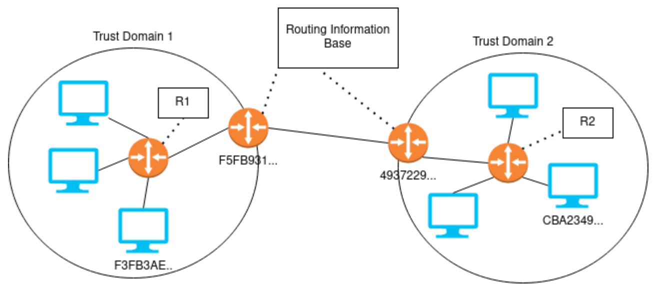

Figure 7: Trust Domains

handle a high rate of forwarding table misses with minimal loss in switching performance.

6 Future Work

6.1 Hierarchical Trust Domains

One important area of future work is supporting hierarchical trust domains (see Figure 7). The basic idea behind hierarchical trust domains is that instead of having a single set of routers all managed by an RIB, there are multiple domains, each of which has border and interior routers and its own RIB. Interior routers can only communicate with each other, and communication between domains requires additional information bases that coordinate links between border routers.

The main benefit of these trust domains is that the path of packets within a domain is hidden from other domains. If a node A in domain X communicates with node B in domain Y, none of the routers in X know the exact routing path through Y, reducing the information leakage throughout the system in case some domains are monitored or compromised in some way.

Our system is designed to be able to maintain its performance in trust domain systems: because our certificate verification scheme works with multi-hop paths and only validates when establishing connections, it could naturally be extended to paths that cross multiple trust domains. The main innovation required in implementing trust domains would be setting up multiple information bases, distinguishing border routers from interior routers, and adding domain-wide certificates indicating access to a resource.

To make trust domains hierarchical – that is, to allow trust domains to contain other trust domains – would also be a straightforward extension of our system. That is because from a router's perspective, even if a chain of communication goes into a trust domain, then into another trust domain, the validation is still just a certificate chain, so our code would still work. Again, the main difficulty in this future work would be coordinating the information bases themselves and adding additional certificates for nested trust domains.

6.2 Integration with existing GDP infrastructure

Currently, the routing system uses abstractions representing GDP structures. For instance, we use a custom metadata structure to store the public key and compute the GDPName. While these abstractions are effective for evaluating performance and functionality, integrating with the existing GDP stack and implementations of DataCapsules etc. would allow for this routing system to be used for real GDP applications.

6.3 Deployment to the edge

Finally, we hope to deploy this system to a real network of communicating devices on the edge, supporting the DataCapsule and secure computation abstractions as part of the GDP. This would allow us to obtain more accurate benchmarks on real traffic, rather than the synthetic workloads discussed earlier. Similar benefits could be obtained by running on real edge devices (rather than AWS cloud instances and virtualized NICs).

7 Conclusion

We present an implementation of a routing system for the Global Data Plane built on top of Capsule/DPDK in Rust that achieves faster performance than previous implementations. We have demonstrated the performance on this system under various synthetic workloads, and shown that it scales well across multiple cores with performance comparable to raw IP switches. Our system supports features such certificate issuance and verification, that lay the foundation for supporting hierarchical trust domains in the future as part of a complete implementation of the GDP.

References

[1] Nitesh Mor. *Global Data Plane: A Widely Distributed Storage and Communication Infrastructure*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2020.

[2] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[3] Capsule. Capsule-rs/capsule: A framework for network function development. written in rust, inspired by netbricks and built on dpdk. https://github.com/capsule-rs/capsule.

[4] DPDK Project. Home - dpdk. https://www.dpdk.org/.

[5] Overview - gdp - global data plane. https://gdp.cs.berkeley.edu/redmine/projects/gdp.

[6] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, page 217–231, New York, NY, USA, 1999. Association for Computing Machinery.

[7] Nikhil Goyal, John Wawrzynek, and John D. Kubiatowicz. Global data plane router on click. Master's thesis, EECS Department, University of California, Berkeley, Dec 2015.

[8] Nitesh Mor, Richard Pratt, Eric Allman, Kenneth Lutz, and John Kubiatowicz. Global data plane: A federated vision for secure data in edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1652–1663, 2019.

[9] Mads Dam and Karl Palmskog. Location independent routing in process network overlays. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 715–724, 2014.

[10] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13 th Usenix Security Symposium*, 2004.

[11] Ioannis Psaras, Konstantinos V. Katsaros, Lorenzo Saino, and George Pavlou. Lira: A location independent routing layer based on source-provided ephemeral names. *CoRR*, abs/1509.05589, 2015.

[12] Robert Gilbert, Kerby Johnson, Shaomei Wu, Ben Y. Zhao, and Haitao Zheng. Location independent compact routing for wireless networks. In *Proceedings of the 1st International Workshop on Decentralized Resource Sharing in Mobile Computing and Networking*, MobiShare '06, page 57–59, New York, NY, USA, 2006. Association for Computing Machinery.

[13] NEOAdvancedTechnology. Neoadvancedtechnology/minimaldpdkexamples: Minimal examples of dpdk. https://github.com/NEOAdvancedTechnology/MinimalDPDKExamples.

[14] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, Savannah, GA, November 2016. USENIX Association.

[15] Chapter 8. nfv performance tuning with ovs-dpdk reference architectures 2017. https:

//access.redhat.com/documentation/en-us/
reference_architectures/2017/html/
deploying_mobile_networks_using_red_hat_
openstack_platform_10/nfv_performance_
tuning_with_ovs_dpdk#picking_cpu_cores_on_
system_enabled_for_hyperthreading, 2017.