Gatekeeper: the design and deployment of a DDoS protection system

Michel Machado*, Cody Doucette†, Qiaobin Fu‡, John W. Byers§

*Digirati, †Raytheon BBN, ‡Google, §Boston University

ABSTRACT

It is prohibitively expensive to mitigate large DDoS attacks in the Internet today. Stakeholders lack affordable and deployable mechanisms to combat attacks, often leading them to contract defensive services from third parties. Meanwhile, waging a DDoS attack remains relatively cheap and simple, creating a cost asymmetry and power imbalance in favor of malicious actors.

To empower victims to close this cost gap, we have designed and deployed Gatekeeper, which harmonizes years of DDoS mitigation approaches from the academic community and applies them to the existing infrastructure of geographically-distributed vantage points (e.g. IXPs, clouds). Governed by software-defined policies, vantage points mitigate attacks close to the source without the coordination of other networks, enabling full deployability by a single AS.

We demonstrate that the richness enabled by eBPF policy programs is effective to mitigate DDoS attacks, and that the overhead of this flexibility is small (<1% of CPU time). We also show that Gatekeeper is production-ready, can process >11 Mpps in the worst case, and even operate at line rate with two billion flows. Moreover, an AS can greatly reduce the cost of defending against DDoS attacks over existing approaches.

1 INTRODUCTION

Despite researchers and practitioners having had decades of experience with the threat, distributed denial of service (DDoS) attacks remain one of the most vexing operational challenges facing Internet providers. According to the Worldwide Infrastructure Security Report (WISR) survey conducted by Arbor Networks [39], DDoS attacks are the top operational concern for service providers. Moreover, we have entered the terabit attack era [2], but DDoS attacks have not yet peaked, especially with the rise of amplification-based [22], IoT botnet-based [44], and cloud-based [55] DDoS attacks, each of which contribute to ever-higher attack rates. Additionally, the financial and social impacts of DDoS attacks are now widely cited in the general media, including the DDoS attack used to disrupt communications between protestors in Hong Kong [51], as well as a 500% spike in DDoS attacks during the COVID-19 pandemic, as millions are forced to work remotely via the Internet [56].

The economics of trying to defend against DDoS attacks is similarly grim. A report by Arbor Networks indicated that the average cost incurred by a victim from a major DDoS attack was over $220K [2]. In contrast, attackers can easily launch a 125 Gbps DDoS attack for only several dollars [50]. Additionally, the average cost of launching a DDoS attack is forecast to fall through at least 2023, since the attack surfaces and resources leveraged by attackers are growing fast from 2018 to 2023: the average broadband speed will more than double, and the number of IoT machine-to-machine connections will grow 2.4-fold [18].

The networking community has been aware of DDoS attacks for decades, but none of the architectural solutions proposed in academic work (§2) have been deployed. With such a compelling need, why is there such a prominent disconnect between research and realization in defending against infrastructure-layer attacks[1]

The fundamental issue is that of incentives: the proposed solutions do not provide sufficient motivation for pioneer autonomous systems (ASes) to deploy them. In particular, consider SIBRA [13]. Although SIBRA offers a deployment plan and is incrementally deployable, its effectiveness depends on the number of ASes that deploy it. Since the solution derives its effectiveness from network effects (in the economic sense), there is minimal incentive for pioneer ASes to deploy the solution initially, and SIBRA, therefore, risks never overcoming the initial activation energy needed to deploy the system widely. To our knowledge, a similar fate has befallen every proposed architectural solution to DDoS over the past 20 years.

We propose to break this deadlock by turning the question of which incentives to provide to a set of pioneer ASes into the question how to craft a system for a *single* AS. For our purpose, we have adopted the network capability and filtering designs from the literature, and adapted them to be *fully* deployable, such that the deploying entity can reap the full benefits of the system alone and on day one. We describe the design, implementation, and evaluation of such a mitigation system – Gatekeeper – and also demonstrate how to deploy it in production by crafting management policies to mitigate DDoS attacks. Moreover, we show that the overall cost of the system is moderate and within the reach of many Internet stakeholders. In fact, two providers with vastly different defensive and cost requirements – 10 Gbps and 1 Tbps protection for Digirati and Mail.ru, respectively – are already deploying Gatekeeper in their networks.

Two design decisions enable Gatekeeper to be deployable by a single AS. First, instead of trickling down a full-blown capability/filtering system into something that a single AS

[1]Infrastructure-layer DDoS attacks target all layers of a protocol stack except the application layer. According to Akamai, these attacks encompassed ~99% of the DDoS attacks in 2017 (See e.g. [1, Figure 2-1]).

can deploy, we design a system that exploits the fact that it will be deployed by a single AS. This led us to a simpler design, including the ability to centralize policy decisions, assume trust between system components, and provide for backwards compatibility of network management. Second, we employ vantage points (VPs) (e.g., Internet exchange points [IXPs] and clouds) to replace the need for joint deployments and/or collaboration with other ASes. The use of VPs reduces the capital and operating expenses of deploying an infrastructure for DDoS mitigation. It also ensures one of the most important features of a *scalable* DDoS defense system: dropping unauthorized packets close to the source.

An effective Gatekeeper deployment can only be achieved by writing and enforcing fine-grained and accurate network policies. While the basic function of such policies is to simply govern the sending ability of clients, Gatekeeper policies are capable of multiple bandwidth limits, punishing flows for misbehavior, intrusion detection, and much more. Policies are implemented as programs, providing richness and flexibility without sacrificing packet processing performance.

We propose three main contributions:

(1) The design and implementation of Gatekeeper, a fully deployable, production-quality mitigation system, which puts into practice 20 years worth of DDoS defense research in capability and filtering systems.
(2) A guide for managing a Gatekeeper deployment, realized through flexible policy programs that run on every packet and at line-rate.
(3) An evaluation of Gatekeeper that demonstrates (1) the richness that policies provide for mitigating DDoS attacks; (2) the system's state-of-the-art performance capacity, e.g., processing over 11 Mpps while managing 2 billion flows with a single server and (3) the expected operational cost of Gatekeeper is within reach of a range of potential deployers, small and large.

The remainder of this paper reviews related work (§2), presents the design of Gatekeeper and shows Gatekeeper in action (§3), describes the implementation details of Gatekeeper (§4), discusses policies in Gatekeeper (§5), evaluates Gatekeeper's performance and estimates its operational cost (§6), explores discussion topics (§7), and then concludes (§8).

2 RELATED WORK

Network capabilities and filters. Researchers have proposed many clever architectural mechanisms to defend against DDoS attacks. Thematically, our work is closest to two classes of solutions that have attracted much research attention: network capabilities [5, 58, 72, 73] and filter-based approaches [8, 29, 45, 47, 64], though many other designs exist [4, 13, 24, 25, 28]. Capabilities require that destinations deny access to their services by default, and then explicitly grant access to senders by exception. In contrast, filtering approaches require that destinations grant access by default, and halt access to senders identified as malicious. These two approaches competed [7] against each other for the better part of a decade, and led to several novel designs and research implementations. Although a few recent projects have given more attention to deployability [28, 46], architectural mechanisms have not seen broad commercial deployment. Why not?

Summarizing a long arc of work briefly, both approaches have weaknesses: filtering-based systems cannot filter attack traffic that does not reach the destination (e.g. short TTL packets), and capability systems cannot filter out attacking flows, but can only retract their capabilities. In a comparative evaluation of filtering systems with capability systems, the StopIt team concluded that both can fail in certain situations and fail-safe mechanisms are needed [45]. Additionally, both approaches lack deployment plans that enable pioneer ASes to draw value from the systems on day one.

Architecting for deployability. Without deployable mechanisms to mitigate DDoS attacks on their own, stakeholders are forced to contract commercial services. These services are often provided by content distribution networks (CDNs), which are geographically-distributed and highly-available networks that perform traffic scrubbing [24, 70]. However, such solutions can be costly, limit the ability of customers to customize their protection [46], and provide the most value for application layer attacks, which represent only a small percentage of attacks in the Internet.

To realize architectural solutions to DDoS in actual deployments, we make use of the existing ecosystem of IXPs and clouds. Placing services near the edge for reverse proxying has a lengthy history, including being the main architectural model for CDNs for content caching, but has also been used for other use cases in recent years [48, 67, 71, 75]. Using IXPs as an insertion point for DDoS mitigation has been proposed [35] and even put into production [24], but such approaches require cooperation with, and deployment by, IXPs, and are therefore not independently deployable by an AS.

Centralized management. Gatekeeper centralizes control over the operation of the data plane using programs, making it architecturally similar in spirit to Software-Defined Networking (SDN) [17, 74, 75], virtualized network functions [37], and programmable data planes [15].

3 SYSTEM COMPONENTS

This section presents *Gatekeeper*, a DDoS protection system designed for independent deployment by a single AS. To do so, Gatekeeper takes the lessons learned from over twenty years of DDoS research and adds three design contributions:

(1) **Architecture.** Gatekeeper *combines* capabilities and filters. Previous work treated the two as "duals" of each other [7], but in Gatekeeper, they are treated as complementary and neutralize each other's weaknesses.
(2) **Deployability.** Prior approaches were not deployed because in order to be successful, the deploying AS needed to control both ends of the path to do capability/filter management, or have trust arrangements with source networks. Gatekeeper circumvents these

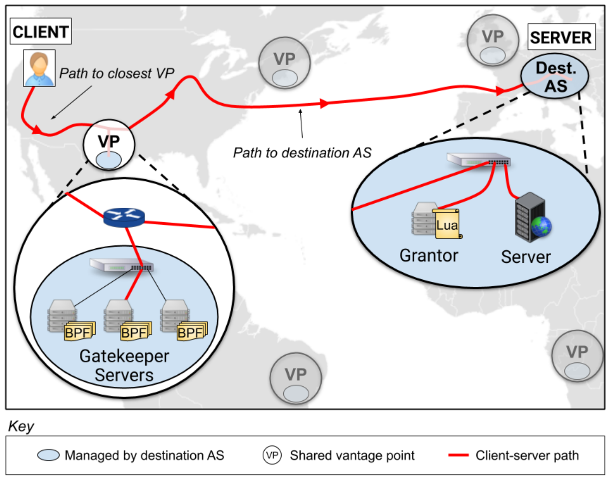

Figure 1: Overview of the components of Gatekeeper.

issues by leveraging geographically-distributed vantage points (VPs).

(3) **Management.** Gatekeeper uses similar centralized management principles as those popularized by SDN [17] to orchestrate DDoS mitigation using policies in software.

Figure 1 depicts the general topology and highlights the two main components of Gatekeeper: one or more Gatekeeper servers located in each contracted VP, and one or more Grantor servers located in the protected destination network. Each time a client tries to send traffic to a server in the destination network, the traffic is forwarded to the closest VP, which performs network capability admission control. If a capability decision for the flow has not already been made, the first packet of a flow is marked as a request and is forwarded to the Grantor server, which will decide to either grant or decline the flow based on a defined policy. If the flow has already been granted, then rate-limiting state is updated, and if the flow has enough credits then the packet is sent along the path to the destination network, where it is decapsulated by Grantor and sent to the ultimate destination. The ability to decline flows gives Gatekeeper the benefits of filter-based systems, reducing the expected connection time for legitimate clients [58, §4.2].

At a high level, Grantor provides the ability to make centralized policy decisions and Gatekeeper provides the upstream policy enforcement. Both ends of this process – decision and enforcement – can be performed using *programs* instead of static, declarative rules. Next, we provide more details about each of these components.

3.1 Vantage points

VPs are upstream insertion points for DDoS mitigation services. They are thematically in the domain of edge computing, an architectural paradigm in which the compute and storage power of the network edge is leveraged to perform reverse proxying for content caching, authentication, TCP termination, load balancing, DDoS mitigation, and other services [48, 67, 71, 75].

A prospective VP is any network location that meets four requirements: (1) compute capacity, (2) cheap ingress bandwidth, (3) BGP peering, and (4) private links to the protected AS. (1) and (2) are basic requirements to ensure that the VP is an insertion point for DDoS mitigation software, and that it can be run affordably. Some candidate VPs, such as cloud providers like Amazon AWS, Microsoft Azure, and Google Cloud, even provide customers with free inbound data transfers. Requirements (3) and (4) are motivated by the design of Gatekeeper. VPs must support the ability to redirect traffic ultimately destined for the protected network through the VP using BGP prefix announcements. This creates an anycast network where client traffic is always forwarded to the nearest VP instead of directly to the protected network. Finally, (4) is essential because the path between a VP and the deploying AS must not expose directly routable addresses to the open Internet. Otherwise, these links are vulnerable to DDoS attacks themselves. By this definition, examples of VPs include IXPs, points of presence, some cloud providers[2], and peering links/carrier hotels.

There are four main advantages to using VPs. First, they are well-provisioned, composed of multiple aggregation levels for high levels of resiliency and bisection bandwidth, which is ideal for handling DDoS-sized traffic volumes. Second, they are often topologically close to source networks, in part due to the flattening of the routing structure of the Internet [23, 34]. This proximity is key to minimize the impact on the latency of incoming flows and, in the case of an attack, to minimize the amount of downstream resources that are wasted. Third, VPs are geographically distributed. There are thousands of candidate VPs spread throughout the globe [11, 57, 59], which enables Gatekeeper to handle Internet-scale attacks. Finally, leveraging the existing infrastructure of VPs reduces the capital expenses for network operators, lowering the deployment barrier.

3.2 Gatekeeper servers

Gatekeeper servers are the main components deployed in VPs. They announce the prefixes of the protected AS via BGP, causing all flows destined for the defended services to be forwarded to the nearest VP. Ingress traffic is load balanced between the available Gatekeeper servers, which then perform the following tasks: (1) bookkeeping policy decisions and flow state, (2) enforcing policy decisions over flows, and (3) encapsulating traffic to be sent to the destination network.

When a packet arrives, the Gatekeeper server first extracts its flow information, defined as the pair (source IP, destination IP), and looks up the corresponding entry in the flow table, which can be one of four types:

(1) **Request**. Either no policy decision for the flow has yet been made by Grantor, or a previous policy decision for the flow has expired.

[2]Some clouds do not support BGP sessions, but the community maintains public lists of cloud and virtual server providers that do [61].

(2) **Granted**. The flow is permitted to send traffic, and Gatekeeper will forward its traffic at a prescribed rate. Packets from this flow beyond that rate will be dropped.
(3) **Declined**. The flow is not permitted to send traffic. All packets from this flow will be dropped.
(4) **eBPF**. The flow is to be processed by a policy enforcement program installed at Gatekeeper. The program itself is *extended Berkeley packet filter* (eBPF) bytecode, which will forward or drop packets according to the specifics of the program.

If the flow does not have an entry in the table, a new entry of type *Request* is added, and the packet is forwarded as a request to Grantor. An entry only moves from the *Request* state to another state after the Gatekeeper server receives a reply from a Grantor server, and the entry remains in that state until it expires or is revoked. Gatekeeper piggybacks capability renewal requests for granted flows that are about to expire to avoid the overhead of setting up a new capability.

Packets which are not dropped are encapsulated so that they can be transmitted to a Grantor server in the destination network. There are two reasons for the encapsulation. For request packets, a policy decision first needs to be made by Grantor before the original data packet is forwarded to the destination server. Second, the encapsulation forces non-request packets to also be processed by Grantor, which allows a central point for measurement and enables the network operator to build policies around a global view of traffic.

3.3 Grantor servers

To complement Gatekeeper servers, Grantor servers are deployed in the protected AS to make the policy decisions that are enforced in VPs. Their responsibilities include (1) decapsulating packets and sending them to their ultimate destination, and (2) running a policy decision program on request packets and informing Gatekeeper of such policy decisions. Details of the policy decision process are in §5.

Grantor has centralized authority over the operation of the data plane, making it functionally similar in design to an SDN controller [17]. Using an SDN-like architecture simplifies network management for Gatekeeper since operators only need to configure one (or a small set of) Grantor server(s) with the desired policy that is then enforced at the VPs. This naturally centralized control of flows by Grantor servers contrasts with packet sampling implemented in commercial DDoS protection systems to centralize control [48].

Finally, the presence of Grantor servers in the design makes Gatekeeper backwards compatible with the existing infrastructure of a deploying AS, since there is no need to change the deployed destinations, as was necessary in previous capability and filtering systems [8, 73].

3.4 Queue management

Our queue management scheme along the path between the VP and the protected AS is mostly a refinement of previous work [58, 73]. The path is composed of three logical channels,

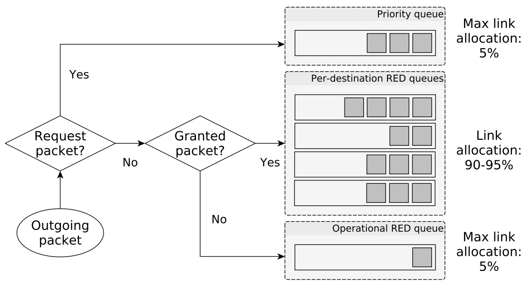

Figure 2: Queue management at Gatekeeper servers and at any capable router on the VP-AS path.

for request packets, granted packets, and operational/legacy packets. Gatekeeper servers and routers on the VP-AS path queue packets according to their type, priority, and destination address (Figure 2). For deployability, we designed the Gatekeeper packet type and priority markings to be encoded within the 6-bit Differentiated Services Code Point (DSCP) field of IP headers: 0 for operational packets; 1 for granted packets; 2 for granted packets renewing a capability; and 3 – 63 for capability requests.

Request channel. Each capability request is placed into a priority queue according to the value of its DSCP field; the higher the priority (i.e., a higher value of the DSCP field), the closer to the exit of the queue the request is placed. When the queue is full, the request with the lowest priority is dropped. The request channel receives only 5% of the bandwidth capacity of the outgoing link. Together, the priority assignment scheme and 5% channel bandwidth limit help prevent denial of capability (DoC) attacks [7] on the request channel that interfere with capability setup.

Since our system is designed to support a single AS, we can assume that the Gatekeeper servers are trusted, which allows us to employ a simpler solution to DoC than the one in Portcullis [58]. Our priority assignment algorithm simply returns the log of the packet's waiting time, i.e., $\log_2(delta\_time)$. Therefore, senders who wait longer to send a packet are assigned a higher priority, which is similar to the exponential back-off method used in Portcullis.

Granted channel. Granted packets are placed in RED queues [30]. The number of queues depends on hardware capacity of the router, and a queue is chosen by hashing the destination address. The seed of this hash should change periodically to minimize the effects of hash collisions. The queues used for granted packets command up to 95% of the bandwidth capacity of the outgoing link.

Operational channel. To avoid communication disruption between routers when the system is under attack, an operational queue reserves 5% of bandwidth for essential protocols such as ARP, ND, ICMP, and BGP.

3.5 End-to-end example

To demonstrate how the components of Gatekeeper work together, we now describe an end-to-end example of a client initiating a TCP connection (Figure 3). Since Gatekeeper supports TCP/IP clients without modification, the client first sends a TCP SYN as normal (1). In (2), the SYN packet is forwarded to the closest VP since all contracted VPs announce routes to the destination AS. Once received by the VP, the packet is transferred to the router of the destination AS, which in turn load balances the packet to one of the Gatekeeper servers.

The Gatekeeper server checks whether there is state associated with the flow of the packet, and if there is, decides what to do based on that state. In this example, we assume there is no state associated with the flow. The Gatekeeper server allocates state that contains the arrival time of the SYN packet, encapsulates the packet using the IP-in-IP protocol, and fills the new IP header as follows: (a) the source IP address is the IP address of the server, (b) the destination IP address is the address of a Grantor server, and (c) the DSCP field is a value in the range $3 - 63$ as determined by the priority assignment algorithm.

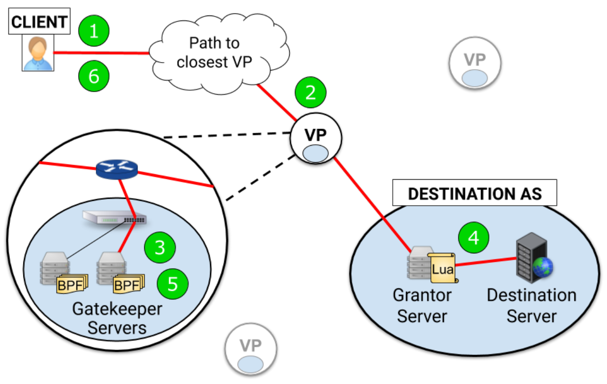

Figure 3: Connection establishment overview.

The encapsulated SYN packet of the client is then forwarded to the destination AS using a previously established tunnel. At the destination, the packet is delivered to the Grantor server (4), which then decides whether to accept the packet based on a policy defined by the deploying AS.

If the decision is to reject the connection, the Grantor server sends its *Declined* decision back to the Gatekeeper server, which includes an expiration time. If the decision is to accept, Grantor includes the rate limit for the flow (e.g., 1 Mbps) and an expiration time in its *Granted* decision that is sent back to Gatekeeper. Grantor then decapsulates the SYN packet and transmits it to the target server.

After processing the SYN packet, the destination server replies with a SYN+ACK that is sent directly to the source. The decision of the Grantor server arrives at the Gatekeeper server (5) and the SYN+ACK packet arrives at the source (6). The source continues to send packets through the Gatekeeper server to the destination as it enforces the assigned rate. Since capabilities expire, Gatekeeper servers may renew capabilities for soon-to-expire flows.

Supporting other protocols. The previous example describes the case of TCP, but what about UDP and ICMP traffic? Gatekeeper servers consider the first packet in UDP and ICMP (and any other protocol above IP) flows like a TCP SYN packet; the Gatekeeper server simply encapsulates the packet and forwards it as a request packet. The destination application does not have to be modified, because Gatekeeper will keep the capability valid as in the TCP case.

4 IMPLEMENTATION

A guiding theme throughout the design and construction of Gatekeeper was always deployability: to see the light of day, the system must be ready for production environments. More than just a research prototype, the Gatekeeper software product combines advances in packet processing technology together with hardware offloading and a suite of desirable features for network operators. The Gatekeeper codebase has around 33k lines of code, and supports both IPv4 and IPv6 deployments. To support Gatekeeper deployments along the VP-AS path, we have also merged the Gatekeeper priority queuing discipline into the Linux kernel[3].

This section describes how prioritizing deployability influenced the development of Gatekeeper, including our choice of DPDK [42] as a packet processing framework, the decomposition of Gatekeeper into functional blocks, and the hardware and software techniques leveraged to optimize performance. For space considerations, we omit the technical details of the numerous optimization techniques in group prefetching, coroutines, hash tables, etc. that greatly improve Gatekeeper's performance. All are available in [31, §4.5].

4.1 Intel DPDK

Typical OS network stacks accommodate a wide range of applications, but are not ideal for the packet processing needs of a DDoS mitigation system. Several packet processing frameworks [12, 38, 60, 63] have been proposed to allow applications to avoid the heavy overhead of OS network stacks and provide line-rate network I/O for very high-speed links (i.e., 100 Gbps). Intel's data plane development kit (DPDK) [42] best suits the needs of Gatekeeper as laid down in the following subsections. DPDK is a set of libraries to accelerate packet processing workloads by allowing applications to bypass the more general and expensive kernel processing. DPDK provides excellent performance in terms of throughput and packet processing latency [32], and its development is stable and driven by many industry collaborators.

Gatekeeper heavily relies on three key features in DPDK: (1) NUMA-aware memory management, which reduces memory access latency by allowing CPU cores to access local memory instead of remote memory; (2) burst packet I/O, which allows Gatekeeper to receive and send packets in batches, reducing the per-packet cost of accessing and updating queues; (3) lockless rings, which provide an efficient concurrency control mechanism for packet buffer allocation and inter-thread communication.

[3]Links to our open-source project and our Linux kernel update omitted to meet the double-blind requirements; included in the final version.

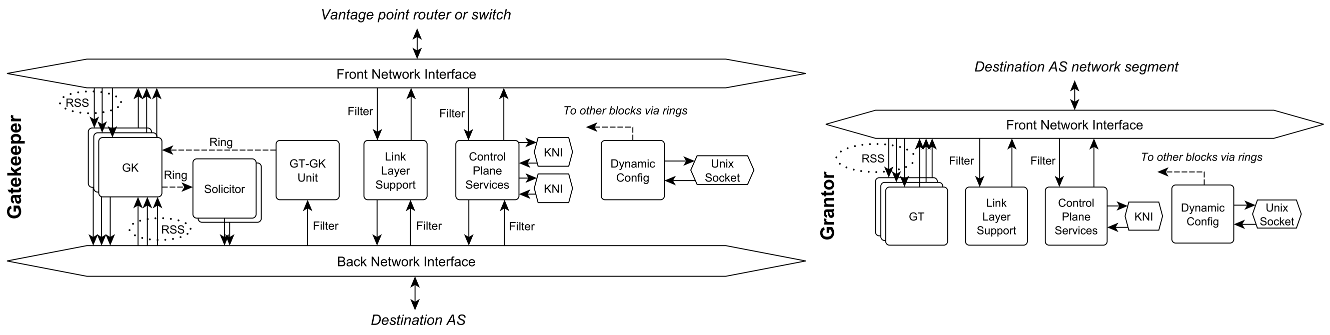

Figure 4: Functional block diagrams of Gatekeeper servers (left) and Grantor servers (right).

4.2 System overview

Figure 4 presents diagrams of the implementations of Gatekeeper and Grantor. Notice that the model of a Gatekeeper server has two physical NICs: one interface is connected to the VP router or switch, which we denote as the *front interface* of Gatekeeper, and the other interface is connected to the private link(s) that lead to the infrastructure of the AS deploying Gatekeeper, which we call the *back interface*. In contrast, Grantor servers each have a single interface that connects to the network segment of the destination network's servers. Each arrow leaving or arriving at a network interface represents a receive (RX) or transmit (TX) queue on the interface, respectively.

Gatekeeper is decomposed into precise functions, which we refer to as *functional blocks*, that represent various data plane and control plane operations. Each functional block is mapped to (at least) one DPDK *lcore*, or logical execution unit of the processor also known as a hardware thread. Each rounded-edge block in the diagram represents an lcore. To better utilize modern NUMA platforms, Gatekeeper carefully spreads functional blocks evenly among all NUMA nodes, and ensures that each functional block allocates its memory on the proper NUMA node to minimize memory access latency. Gatekeeper also creates a packet buffer pool for each functional block individually to receive packets from the NIC, which minimizes memory contention between blocks.

Other notable hardware and software features are labeled, which include (1) *RSS*, receive-side scaling, which load balances received packets across multiple queues on the same NIC; (2) lockless rings for message passing between blocks; (3) filters, or declarative traffic rules, which enable Gatekeeper to map certain protocols to functional blocks; (4) the KNI, or kernel-NIC interface, a way for DPDK to interface with the Linux kernel through a network device; and (5) a Unix socket, which provides the means for Gatekeeper to communicate with a client program for runtime configuration. More information about these components are presented alongside the hardware requirements and functional block definitions, which we describe next.

4.3 Hardware requirements

To fully leverage the potential of modern multi-/manycore CPUs, we choose NICs with multi-queue and RSS support. This gives Gatekeeper two main advantages: (1) RSS enables the NIC to distribute incoming packets to different queues, so that lcores do not need to contend with each other; (2) RSS ensures that packets belonging to the same flow will be processed by the same lcore in the same order as they arrive at Gatekeeper server, thereby avoiding out-of-order packet delivery. RSS hashes a subset of a received frame's header to assign the frame to a queue/lcore. The RSS hash function is parameterized by a secret key which Gatekeeper randomizes at startup to mitigate collision-based attacks.

Ideally, Gatekeeper NICs should also support the following features in hardware: (1) EtherType filters, which allow packets to be steered to queues on the basis of the EtherType field of the Ethernet header; (2) n-tuple filters, which allow packets to be steered to queues on the basis of fields in L3 and L4 headers; (3) checksum computation in IPv4 and UDP headers; and (4) VLAN stripping for domains where VLAN headers are required (e.g., IXPs). However, if Gatekeeper detects that any of these features are not supported in hardware, it will fall back to software equivalents.

4.4 Functional blocks

Gatekeeper and Grantor servers share a single piece of software, which is composed of modular functional blocks. Configuration settings determine whether the executable runs as Gatekeeper or Grantor. There are three main advantages to this approach: a single piece of software simplifies the engineering, configuration, and administration of the system; (2) blocks that are common to both Gatekeeper and Grantor can be re-used; and (3) operators can fine-tune the system by scaling up the blocks that implement data plane operations without wasting resources on low-bandwidth control plane operations. We now describe the roles of each block.

4.4.1 Data plane operations. **Gatekeeper (GK).** The GK block is the main component of Gatekeeper servers, as its task is to accept incoming packets, perform lookups that map flows to policy decisions, and queue requests and granted packets for transmission to Grantor. It is in the data plane and can scale across multiple GK *instances*, each with a dedicated lcore. Gatekeeper utilizes RSS to distribute incoming packets among the GK instances. Each instance processes a queue whose packets have unique pairs of source and destination addresses. Since RSS guarantees that packets belonging to the same flow will be directed to the same RX queue, and therefore the same GK instance, each GK instance maintains its own lockless flow hash table whose keys are (source IP, destination IP) pairs. Because of its position on the front lines of the data plane, meaning it will bear the brunt of attacks, the GK block uses prefetching, batching, and other memory optimization techniques to achieve high performance [31].

Grantor (GT). The GT block is the main component of Grantor servers, as its task is to accept incoming packets from Gatekeeper servers, issue policy decisions for requests, and forward granted packets to their ultimate destination. It is in the data plane and can scale across multiple lcores. §5 provides a thorough overview of the role of the policies that the GT block manages.

Solicitor (SOL). The SOL block is responsible for rate limiting and sending request packets. Packets from flows in the request state are handed to the SOL block from the GK block via lockless rings. Requests are sorted by priority and only permitted a fraction of the link capacity between Gatekeeper and Grantor, and the SOL block enforces these limits. The priority queue is implemented as a length-limited linked list of request packets, indexed by an array whose elements are references to the portion of the linked list that holds packets of each priority, providing constant time insertion, dequeuing of the highest priority request, and deletion of the lowest priority request when the queue is full. The SOL block only runs on Gatekeeper servers, and multiple SOL instances are used to spread the load coming from multiple GK instances.

GT-GK unit (GGU). The GGU processes all policy decisions that arrive from Grantor servers. Policy decisions are steered directly to the GGU block using n-tuple filters. The GGU block demultiplexes each decision to the GK instance that is responsible for the flow in question, according to its RSS hash. The GGU only operates on the back interface of Gatekeeper servers.

4.4.2 Control plane operations and configuration. **Control plane services (CPS).** Gatekeeper servers must support control plane protocols (e.g., BGP, OSPF, and IS-IS) to peer in VPs and to integrate with the destination AS. Instead of adding support for a multitude of common control plane protocols in Gatekeeper directly, we enable network operators to use existing routing daemons and management tools by leveraging the DPDK kernel-NIC interface (KNI) library. This approach is similar to the one employed in Google's Espresso [75]. Our deployments have used the popular BGP speaker BIRD [14]. Gatekeeper uses n-tuple filters to steer BGP packets to the CPS block.

Link layer support (LLS). The LLS block has the responsibility of handling all link layer protocols and address resolution services, such as ARP, ND, and LACP. Instead of resolving IP addresses to MAC addresses on demand, other functional blocks must register the IP addresses that they are interested in, and the LLS block keeps IP address-to-link layer address maps updated. Gatekeeper uses EtherType filters to steer ARP packets to the LLS block.

Dynamic configuration. The dynamic configuration block allows operators to change the parameters of Gatekeeper and Grantor servers and to diagnose runtime issues. For example, operators can update the IP ranges handled by GK blocks, list the ARP and ND tables for network diagnosis, update the enforced policy on Grantor servers, and flush all policy decisions cached at Gatekeeper servers associated with a given destination IP.

5 DESTINATION POLICIES

The correctness and precision of a policy make or break a Gatekeeper deployment. This section presents the design choices that shaped the format of policies and a deployment-tested template to write policies.

5.1 Design choices

The design of Gatekeeper policies borrows heavily from two streams of prior work: capability and filtering systems to regulate the transmission rate of flows, and SDN to centralize this decision-making process. However, the design choice of implementing destination policies as *programs* represents a breakthrough. Prior work implemented policies as sets of rules that are pattern matched to packets or flows. These rules and patterns are described under a predefined declarative language. The motivation for using declarative languages is to provide abstractions for hardware operations and filters. These languages, however, become the weakest link in a DDoS protection system when attacks target their limitations. For example, attackers can force source address filtering mechanisms in legacy routers into an untenable position: either use coarse-grained filtering rules and incur much collateral damage, or use fine-grained filtering rules and risk not mitigating the attack [66].

Running Gatekeeper policies as programs does not require giving up the hardware abstractions. However, it enables us to move away from declarative languages and toward bytecode virtual machines (VMs). We chose to adopt policies as programs in two ways:

(1) A set of policy *enforcement* programs that are run at Gatekeeper servers. On ingress, a packet's flow is extracted and mapped to the enforcement program that it has been assigned. Policy enforcement programs may keep state, e.g. for a token bucket algorithm to rate-limit traffic.
(2) A single policy *decision* program that is run at Grantor servers. The decision program maps flows to policy enforcement programs, and installs rules reflecting

those decisions at Gatekeeper servers so that the enforcement program can be run on subsequent packets in the given flow.

This approach has two advantages. First, it enables Gatekeeper to take prompt action on flows that misbehave *after* receiving a favorable policy decision, e.g. by applying secondary rate limits or by tracking negative bandwidth at policy enforcement time (§5.2). Previous capability systems have relied on capability expiration or bandwidth caps to mitigate this issue [72, 73].

Second, it allows us to separately choose the best fitting bytecode VMs for Gatekeeper and Grantor servers. Gatekeeper servers perform policy enforcement using VMs that run eBPF programs [54], whereas the policy decision program running on Grantor servers are implemented using a Lua VM [41]. The key requirements that led us to choose Lua on Grantor servers was (1) its support for dynamically editing the policy (e.g. redefining functions, modules, variables) in order to enable changing the policy with minimum impact to deployed systems, and (2) ease of integration with external libraries (typically in C). eBPF was chosen to implement policy enforcement programs due to (1) the availability of static analysis to guarantee termination, memory safety, and bounded resources [33] and (2) ease of packet inspection.

While eBPF has been used in production to implement packet redirection in load balancers [67, 71] and high-performance, ad hoc packet filters in DDoS protection systems [27], the use of eBPF programs in Gatekeeper to perform policy enforcement brings greater flexibility. The difference in these approaches is in the amount of state space under the control of eBPF programs. For example, eBPF programs in Gatekeeper can limit bandwidth per flow, while the eBPF programs running on other systems have only enough state to do so for a limited number of classes.

5.2 Writing production policies

As a guideline, we found that a policy is generally organized as follows: policy decision programs (in Lua) inspect IP addresses to map flows to policy enforcement (eBPF) programs, and eBPF programs inspect transport headers to monitor the behavior of flows and adhere to the given policy decision.

As a rule of thumb, each eBPF program reflects a network service *profile*. Consider the profile of outgoing email servers. They have no listening sockets – they only open connections to the SMTP port of remote email servers, and these connections have very small ingress traffic footprints. This profile-to-program heuristic leads to a simple breakdown of the work to write a policy: (1) identify all network profiles, (2) write an eBPF program for each of those profiles, and (3) map flows to those eBPF programs in the Lua policy.

Following this proposed breakdown of work, Lua policies are configured with a set of network prefixes, and use those prefixes to classify incoming packets based on their destination addresses. We use longest prefix matching on destination addresses by default, but source address classification could also be used, such as to approximate source location [52], identify threats [21, 68], check the purpose of the source [20, 36, 65], etc. Regardless of whether source or destination (or both) addresses are inspected, this classification is used to decide on denying or granting communication, limiting bandwidth, and differentiating service.

When translating a network profile into an eBPF program, the policy writer should classify packets into three bins: primary, secondary, and unwanted traffic. Primary traffic carries the main purpose of the service, while secondary traffic is permitted traffic that has no reason to be present at the same scale of the primary traffic. Examples of secondary traffic are TCP SYN, ICMP, and fragmented packets. Unwanted traffic, such as malformed packets, is dropped. Once the code for the classification of packets is in place, the policy writer overlays it with two bandwidth limits: one limit before the classification to control the bandwidth of the flow as a whole, and another limit for secondary traffic after the classification. These limits can be implemented as token bucket algorithms.

We use a variation of the token bucket algorithm on the primary traffic that allows the number of tokens to go negative. This negative bandwidth works as an automatic punishment for flows that consistently exceed their bandwidth limit, which is crucial for mitigating attacks where the adversary obtains a capability to send, and then abuses the capability by flooding Gatekeeper with packets. Such an abusive flow will only have its packets forwarded again once it stops sending (or reduces its rate), and enough time has passed to bring the token balance to a positive value. Therefore, the duration of the bandwidth shutoff is proportional to the offense.

Policies that follow the template explained above can easily handle the most common forms of infrastructure attacks, such as floods (e.g. SYN, UDP, ICMP), amplifications (e.g. DNS, NTP, Memcached), and arbitrary combinations of these attacks (also known as multi-vector attacks). The fraction of the attack traffic that bypasses these policies depends on how narrow the network profiles are, how precise they are implemented in the eBPF programs, and the quality of the classifications of the source addresses in the Lua policy. But even if these are all finely tuned, some attack traffic can only be identified by the protected applications or intrusion detection systems (IDSes). In this case, applications and IDSes can feed the Lua policy with source addresses and packet signatures to mitigate this sneaky traffic through ad hoc eBPF programs and assignment of lower bandwidth limits. Gatekeeper cannot help when there is no distinction between attack and legitimate traffic.

Finally, the policy template presented here leaves room for potential improvements by leveraging the fact that Grantor servers are colocated geographically. This geographical proximity provides for low latency and ample bandwidth between Grantor servers, which in turn enables the employment of a distributed database for policies to use. With the help of this database, Lua policies could potentially identify spoofed source addresses analyzing the (source address, incoming vantage point address) pair for inconsistencies, as well as make sophisticated bandwidth allocation based on the source AS and load of the links behind the Gatekeeper servers.

6 EVALUATION

We now evaluate Gatekeeper along several axes, including the effect of different policies during attacks (§6.1), the cost impact of destination policies (§6.2), performance benchmarking (§6.3), and cost estimates of deployments (§6.4).

6.1 Effect of policies

Gatekeeper policies define how flows are admitted, rate limited, and monitored, and ultimately determine which packets are transmitted or dropped at Gatekeeper servers. Therefore, they drive the ability of Gatekeeper to mitigate attacks.

To measure the effect of various policies on attack mitigation, we used an Amazon Web Services Elastic Compute Cloud (EC2) testbed. The testbed is composed of two attacking clients (instance type m5.4xlarge), which run packet generators to launch attacks from a total of 16K virtual attackers (source IPs) to a destination Web server (m4.2xlarge). The testbed also has a legitimate client (t3.medium), which uses `curl` to repeatedly upload a 20 KB file to the destination server. We chose uploading instead of downloading because competing with attackers for uploading bandwidth is more challenging for the legitimate client. For each experiment, the client uploads the file 50 times, and the average file transfer time is computed across all transfers. All client traffic is redirected through a Gatekeeper server (m5.8xlarge), which transmits requests and granted traffic through a path router to a Grantor instance (m5.8xlarge), which forwards traffic to the destination server.

We encountered several testbed limitations when using EC2, including lack of full support for hardware offloading (such as RSS), as well as packet throughput caps. This limited our ability to test Gatekeeper and Grantor servers at their full potential. Full details about these limitations and the EC2 settings we used are available in related work [26, §2.4].

6.1.1 Bandwidth floods & negative bandwidth. We first measured Gatekeeper's ability to mitigate bandwidth floods. We configured the attacker to flood the destination server using 1024B TCP (non-SYN) packets while the legitimate client repeatedly attempts to upload the file. We evaluated attack strengths from 100 Mbps up to 10 Gbps. All traffic was first directed through Gatekeeper, which applied a policy of granting *all* flows[4]. We performed different trials, in which we granted traffic at rates of 32, 64, and 128 Kbps. We compared these policies against a scenario in which Gatekeeper is not used and all client traffic is forwarded directly to the destination server (label "No Defense" in the graphs).

[4]This policy is favorable to the adversary; §5 describes how Gatekeeper policies can differentiate flows by sender behavior and identity.

The result shown in Figure 5 (left) is that Gatekeeper is able to completely mitigate the effect of the attack. Any inflation in the legitimate client's file transfer time is due to the applied policy. For example, under a 32 Kbps policy, it takes about 10 seconds (10 token bucket refill periods) for the legitimate client to transfer the file regardless of the attack strength. As the rate limit increases to 64 Kbps, the legitimate client is able to transfer the file more quickly while the attack itself is still mitigated. Without any defensive system, the file transfer time increases exponentially up through trials with 5 Gbps of attack traffic. Beyond that point, the legitimate client times out when trying to perform the transfer.

Although doubling the rate limit from 32 to 64 Kbps decreased the file transfer time, doubling it again to 128 Kbps actually drives the file transfer time up. This is due to the fact that the Gatekeeper server now admits enough attack traffic to start to effect the service of the legitimate client.

To show how policy richness can mitigate this effect, we also deployed a 128 Kbps policy enforcement program that uses *negative bandwidth*, i.e., which drops all packets from flows while they have exceeded their bandwidth allotment (§5.2). This simple enhancement still mitigates the attack while allowing the legitimate client to transfer the file within two seconds – the fastest among the policies tested.

6.1.2 SYN floods & secondary bandwidth. We also evaluated Gatekeeper against TCP SYN floods, for which we used policies that enforce two bandwidth limits for each flow. A secondary bandwidth limit (§5.2) allows Gatekeeper to impose a second, lower limit for certain types of traffic within flows, such as ICMP, UDP, or TCP SYN packets. In this evaluation, we change the traffic flood to be composed of TCP SYN packets, and change the policy decision to apply a secondary bandwidth to TCP SYNs. The primary bandwidth limit is 256 Kbps, and the secondary bandwidth is set to be 12.8 Kbps (5% of the primary limit). Figure 5 (center) shows that using such a a policy, Gatekeeper mitigates SYN floods and allows the legitimate client to transfer the file in one rate limiting period. When no defensive system is used, the file transfer time grows exponentially.

6.2 The cost of destination policies

With the use of expressive destination policies in Gatekeeper, two cost-related questions arise: (1) what is the memory cost associated with the flow tables of Gatekeeper servers? and (2) how much effort is required to write and maintain a policy? This section shows that the amount of memory needed to implement the flow tables of Gatekeeper is not excessive by current standards, and market trends make it even more cost-effective. We then summarize a policy used in production to show that policies are modest in size.

A flow entry in a Gatekeeper server consumes approximately 256 bytes[5]. Nowadays, a dual socket x86 server can easily have more than 512GB of RAM, and therefore a single

[5]Each flow entry is effectively 128 bytes long, but we assume 256 bytes per flow to account for auxiliary hash table data structures.

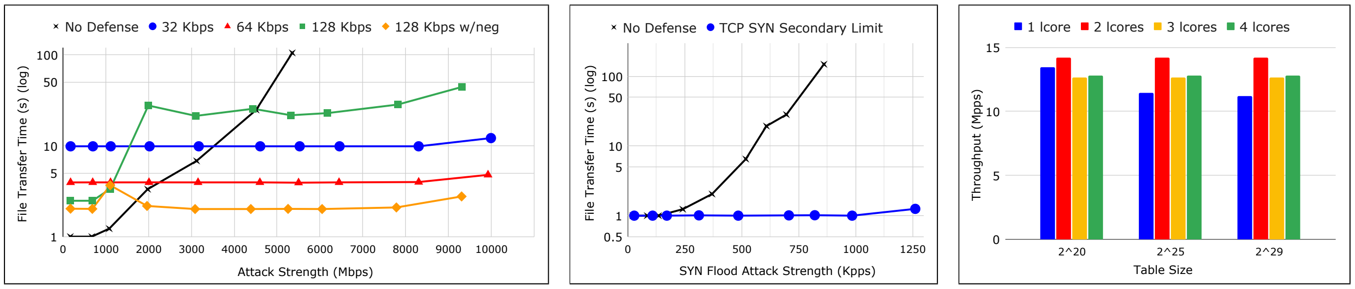

Figure 5: Average transfer time of a legitimate client under various Gatekeeper policies during bandwidth floods (left) and SYN floods (center); throughput of a Gatekeeper server during a flow state exhaustion attack (right).

Gatekeeper server can track roughly 2^{31} ($\approx$ 2 billion) flows. To put this figure in perspective, consider the large attack [19] that Cloudflare defended against in June 2020, in which over 316,000 IP addresses targeted a single Cloudflare IP address. This attack would have consumed less than 0.02% of the flow tables of a single Gatekeeper server.

On top of a reasonable memory footprint, the flow tables of Gatekeeper servers are bound to become less of a burden in the future. The cost of memory has continuously fallen more than 30% per year for over 50 years [3], and the cost of DRAM is less than \$4 per GB [53]. The Gatekeeper architecture is also horizontally scalable , as each Gatekeeper server adds memory to the system, and with more vantage points comes less memory pressure per vantage point.

The effort to write a policy following the template described in §5.2 is mostly proportional to the number of network profiles and the number of exceptions in the Lua policy in order to map flows to eBPF programs. Much of this information can come from existing system configurations. We expect that most of the effort in writing policies will come from bringing organizations up to speed with the knowledge required to write them, and not the writing process itself.

For example, consider one of the first production deployments of Gatekeeper, established by Digirati. Their policy has 17 eBPF programs with an average of 66 source lines of code (SLOC) per eBPF program, and its Lua policy has 289 SLOC. These metrics show that the amount of effort to write and maintain policies is fairly low once all of the needed information is gathered.

6.3 Microbenchmarks

Gatekeeper servers examine 100% of ingress traffic, and this examination is more intensive than simple forwarding decisions as highlighted in the previous sections. Each data plane packet is passed through an eBPF program, which may calculate and impose (potentially multiple) bandwidth constraints, apply conditional logic, perform deep packet inspection, etc. Therefore, in this section we evaluate Gatekeeper's packet processing capacity while (1) varying the multithreading level and flow table size of the GK block and (2) processing all packets through eBPF programs.

6.3.1 Flow table churn. We measured the performance of a single Gatekeeper server with overwhelmed flow tables. To do so, we fully randomized the source addresses of the attack traffic, creating a virtually indefinite stream of new flows. Attack traffic was composed of minimum-sized, 64B packets. When processing packets, Gatekeeper treated all flows as requests. We measured the capacity of Gatekeeper to process this traffic by varying the flow table size per GK instance and the number of GK instances.

The experiment was run on a Dell PowerEdge R640 server, which has 768 GB of memory and two Intel Xeon Silver 4214R 2.4 GHz processors, each equipped with 16.5 MB of cache space and 12 cores with 24 threads. For networking, the server uses Intel X550 10 Gbps adapters.

The results are shown in Figure 5 (right). Even under extreme conditions – minimum-sized packets, a full flow table, and new flows constantly arriving – Gatekeeper can process at least 11 Mpps under all flow table sizes and GK instances configurations. Note that the configuration with flow table size of 2^{29} flows per GK instance and 4 GK instances supports ~2 billion flows; a configuration whose memory cost is discussed in §6.2. For context, being able to process 10 Mpps is considered highly performant by industry standards [49]. Additionally, a rate of 11 Mpps equates to a transfer rate of 7.7 Gbps when factoring in the hardware transmission overhead imposed on 64B packets.

Gatekeeper cannot process all packets with a single GK block instance. However, with more than one GK block instance, Gatekeeper can process packets as fast as the generator sends them. Due to limitations of our testbed, the packet rate of the generator decreases as the number of GK instances increases. Therefore, under the limitations of our testbed, Gatekeeper operates at line speed in all experiments with more than one GK instance.

6.3.2 Per-packet program invocation. To measure the overhead of processing 100% of ingress traffic through arbitrary eBPF programs, we profiled Gatekeeper on the EC2 testbed using Intel VTune [62]. We ran a 3 Gbps bandwidth flooding attack with the same setup as in §6.1.1, and analyzed the CPU usage of the GK block (using one GK instance) to find the CPU time spent processing ingress packets. We compared the built-in routines for declined and granted flows,

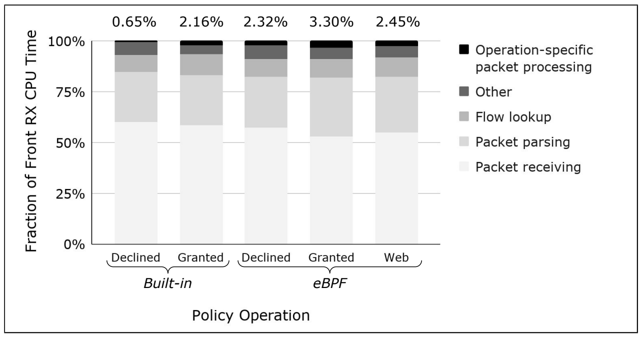

Figure 6: Profile of the GK block's CPU usage.

which consist of simple packet dropping and rate limiting respectively, with eBPF programs that perform the same operations. In addition, we ran a trial that used a "Web" eBPF program, a policy that represents a Web server and inspects transport-layer fields to decide how to handle packets.

Figure 6 shows the breakdown of the CPU usage during ingress packet processing on the front NIC. We choose to ignore the other operations that the CPU runs (ingress processing on the back NIC, packet transmission, inter-block communication, etc.) to magnify the *operation-specific packet processing* time, which is the measurement of interest. The operation-specific processing times represent the CPU times spent running eBPF programs or performing the built-in operations, and are labeled in the figure.

There is almost no difference between running eBPF programs (2-3%) and performing built-in processing of granted packets (2.16%). Declined flows can use significantly less CPU time when using the built-in routine (0.65%), but since all packets of declined flows are simply dropped, an eBPF program is unnecessary anyway. Although the Web eBPF program is a more complex routine, it is less expensive than the Granted eBPF program. This happens because the Web eBPF program drops many more packets than the Granted eBPF program due to the enforcement of negative and secondary bandwidth. When considered in the context of total CPU usage instead of only front NIC RX processing, all operation-specific processing consumes <1% of CPU time.

6.4 Cost estimates of deployments

One of the motivating factors that drove the design of Gatekeeper was to keep it affordable and within reach of smaller ISPs, enterprises, institutions, etc. In this section, we provide a back-of-the-envelope evaluation of the cost of Gatekeeper under three deployment scenarios: (1) a deployment of 2.3 Tbps of bandwidth protection, which represents one of the largest reported DDoS attacks by volume [9]; (2) a 2.3 Tbps deployment shared among 20 companies; and (3) a deployment that balances peak capacity and cost.

As a baseline, we first consider various publicly available cost estimates for DDoS protection. After the 620 Gbps DDoS attack against *KrebsOnSecurity* [44], Akamai estimated a cost of "millions of dollars" to defend against an attack of that scale [16], while a separate (anonymous) cost estimate was in the $150K to $200K per year range [44]. At a different price point (and in reach of small and medium size companies), our partner has been quoted a 20 Gbps protection contract against infrastructure-layer attacks by an established security company at a rate of $24K per year. Finally, cloud services like AWS Shield [10] also provide DDoS defense, albeit with a pay-as-you-go model, as opposed to a flat annual rate.

Now consider the cost of Gatekeeper. To shield a single AS with 2.3 Tbps of bandwidth protection, our modeled deployment for Scenario 1 would use 23 VPs, each with 100 Gbps of incoming traffic capacity. Conservatively estimating the cost of operating at each VP at $5K per month[6], the operational cost of this deployment would be $1.4M per year. This estimate includes the quoted price to contract a link in an IXP, and factors in an estimate of additional operational costs, such as those to contract layer 2 connectivity to the deploying network. In Scenario 2, we envision the Gatekeeper architecture being deployed in a cooperative model, shielding multiple customers in parallel and amortizing costs. While this would require some additional overhead to orchestrate policies of multiple ASes in one Gatekeeper instance, there would nevertheless be significant economies of scale (and ample processing power to share). We estimate that twenty companies could band together to share the 2.3 Tbps defense shield, for less than $100K per year per company.

We next consider Scenario 3. According to Arbor Networks, 99% of the DDoS attacks in 2016 peaked at less than 20Gbps [6, Figure AT3]. Thus, a small company willing to endure downtime when a 1% DDoS attack hits could deploy a 20Gbps Gatekeeper shield for about $12K per year, using the same assumptions as Scenario 1.

To make an apples-to-apples comparison to commercial solutions, we must account for additional operational costs such as labor costs, other supply costs, and administrative overhead. The simplest option would involve IXPs offering Gatekeeper servers as a service using their existing infrastructure. Interested companies would incur lower costs to add VPs and Gatekeeper servers to their deployments, keeping incremental operational cost to a minimum. IXPs would be interested in such an option because of the new revenue stream without a major infrastructure investment. Similarly, a deployment based at cloud providers rather than at IXPs can also keep operational costs low, as the elastic services provided enable companies to quickly scale the number of VPs and computing capacity up and down.

At the other extreme, one could deploy Gatekeeper for a large number of clients on top of a dedicated, managed worldwide infrastructure. While it is challenging to estimate the associated costs, one baseline can be extrapolated from Hurricane Electric [40], a profitable Internet backbone and colocation provider with estimated revenue of $38M in

[6]We could not identify public data to back this estimate, but our industry partners have verified that this value covers their market price estimate to deploy Gatekeeper at an IXP.

2020 [76]. Hurricane Electric is present in over 240 Internet exchange points, with over 100 terabits of edge capacity, terabits of bandwidth to route traffic from its edges to over 250 colocation facilities around the world, and professional management of its network. Given the similarities between what a company like Hurricane Electric does and what would be required from a company offering Gatekeeper protection at the same scale, this hypothetical 100+ terabit deployment of Gatekeeper would be at least a decade ahead of the peak capacity of DDoS attacks, and could be operated for roughly $40M annually. At this scale, one could profitably onboard hundreds of high-value customers at multi-Tbps protection ($100K/Tbps/year), and thousands of smaller ISPs and enterprises at, say, 100 Gbps protection ($10K/year).

Taken together, these rough estimates illustrate that Gatekeeper has the potential to be cost-competitive with current market solutions, both large and small, with savings comfortably in the 2 to 4x range.

7 DISCUSSION

Any production deployment carries both foreseeable and unforeseen risks. This section discusses some vulnerabilities of Gatekeeper in terms of possible attacks and failures.

Control channel co-option. Attackers could attempt to disrupt the control channel between Grantor and Gatekeeper servers. Since this channel is used to provide policy decisions, it is an especially attractive target for attackers, who could try to alter decisions to grant access for their traffic or deny service to others. Physically isolating this channel using leased, private links would mitigate this kind of attack since all links behind a Gatekeeper server are under the control of the deploying AS. The channel could also be cryptographically protected – for example, by requiring TLS authentication of Grantor servers.

Attacks against a VP. Gatekeeper does not protect against volumetric attacks attempting to overwhelm the resources of the VP itself. However, the impact of these attacks is limited in Gatekeeper deployments leveraging multiple VPs, since routes would eventually be announced to direct traffic away from any affected VP. As a last resort, Gatekeeper can blackhole traffic at a VP to geographically bound the impact.

Novel infrastructure-layer attacks. In recently proposed attacks, attackers send flows to a large number of destinations (Crossfire [43]) or to other colluding attackers (Coremelt [69]). These flows flood a small set of carefully chosen links, denying service to the targets and to other destinations as collateral damage. Although systems like SIBRA [13] can help mitigate these attacks, they would be complex to implement and deploy. Gatekeeper also has the potential to mitigate these attacks by virtue of its distributed architecture and ability to be deployed in cloud environments, the feasibility of which was analyzed in previous work [26].

Non infrastructure-layer attacks. Although non infrastructure attacks comprised ~1% of all DDoS attacks in 2017 [1, Figure 2-1]), they could become more prevalent as infrastructure attacks are neutralized. Gatekeeper can mitigate any attack that its policy programs can capture via packet header data and metadata (e.g. timing). Gatekeeper policies could also be informed of application abuses reported via APIs. However, the full effects of this have not been investigated, so we only claim that Gatekeeper is effective against infrastructure attacks. Meanwhile, Gatekeeper may be used alongside application-layer DDoS defense systems, especially those designed for encrypted application traffic.

Fault tolerance. Gatekeeper is a distributed system, and must gracefully deal with failures to minimize end user impact. Grantor servers are stateless, and therefore new Grantors can easily be provisioned on failure. However, Gatekeeper servers keep state over incoming flows, so when failures occur, flows with existing capabilities will initially be handled as requests when assigned to a new Gatekeeper.

Ethical considerations. We recognize that new methods for defending against DDoS attacks run the risk of being used for harmful ends, notably including censorship. That said, our methods are intended only as a mechanism to block *unwanted* traffic; moreover, our study does not involve human subjects, and thus does not raise ethical concerns.

8 CONCLUSION AND FUTURE WORK

We have proposed Gatekeeper, a DDoS protection system that empowers various actors, small and large, to affordably defend themselves against the attacks that cripple the Internet today. To do so, we prioritize deployability, and therefore designed Gatekeeper to fit within the existing Internet ecosystem and architecture. We do not propose new hardware, modifications to servers or clients, new wire protocols, or shared mechanisms between networks. Instead, we propose injecting a new architectural spin on old ideas, combining the benefits of previous approaches and putting them into practice.

Central to our design is the choice of vantage points, which lowers costs, obviates the need for mutual deployments, and minimizes the amount of wasted upstream resources during attacks. Additionally, to meet production demands, we utilize modern, software-defined management techniques and state-of-the-art packet processing technology.

Perhaps the most compelling evidence of the strength of our approach is the fact that there are two ongoing deployments of Gatekeeper at the time of submission. In the future, we hope to expand access to Gatekeeper by supporting *shared* deployments. We believe in an open, community-driven approach to software and security, and want to enable every stakeholder to have affordable DDoS protection on their own terms by sharing infrastructure and management. We are working on making this practical from a technical perspective, including further optimizing Gatekeeper (e.g. with SmartNICs) to achieve 100+ Gbps throughput on a single commodity server, as well as supporting cloud environments.

REFERENCES

[1] 2017. *State of the Internet / Security Report Q1 2017.* Technical Report. Akamai.

[2] 2019. *14^{th} Annual Worldwide Infrastructure Security Report.* Technical Report. Arbor Networks.

[3] AI Impacts. 2021. Trends in DRAM price per gigabyte. https://aiimpacts.org/trends-in-dram-price-per-gigabyte/. (2021).

[4] David G Andersen. 2003. Mayday: Distributed Filtering for Internet Services. In *USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Vol. 4. 20–30.

[5] Tom Anderson, Timothy Roscoe, and David Wetherall. 2004. Preventing Internet denial-of-service with capabilities. *ACM SIGCOMM Computer Communication Review (CCR)* 34, 1 (2004), 39–44.

[6] Darren Anstee, Paul Bowen, C.F. Chui, and Gary Sockrider. 2017. *12^{th} Annual Worldwide Infrastructure Security Report.* Technical Report. Arbor Networks.

[7] Katerina Argyraki and David Cheriton. 2005. Network capabilities: The good, the bad and the ugly. *ACM Hot Topics in Networks (HotNets-IV)* (2005).

[8] Katerina J. Argyraki and David R. Cheriton. 2005. Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *USENIX Annual Technical Conference (ATEC '05).* 135–148.

[9] AWS. 2020. AWS Shield Threat Landscape Report, Q1 2020. https://aws-shield-tlr.s3.amazonaws.com/2020-Q1_AWS_Shield_TLR.pdf. (2020).

[10] AWS. 2021. AWS Shield Pricing. https://aws.amazon.com/shield/pricing. (2021).

[11] AWS. 2021. Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure. (2021).

[12] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15).* 5–16.

[13] Cristina Basescu, Raphael M. Reischuk, Pawel Szalachowski, Adrian Perrig, Yao Zhang, Hsu-Chun Hsiao, Ayumu Kubota, and Jumpei Urakawa. 2016. SIBRA: Scalable Internet Bandwidth Reservation Architecture. In *Network and Distributed System Security Symposium (NDSS '16).*

[14] BIRD 2021. The BIRD Internet Routing Daemon. https://bird.network.cz/. (2021).

[15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (CCR)* 44, 3 (2014), 87–95.

[16] Hiawatha Bray. 2016. Akamai breaks ties with security expert. Boston Globe. (23 Sept. 2016).

[17] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: Taking control of the enterprise. *ACM SIGCOMM Computer Communication Review (CCR)* 37, 4 (2007), 1–12.

[18] Cisco Visual Networking Index 2020. *Cisco Visual Networking Index.* Technical Report.

[19] Cloudflare. 2020. Network-layer DDoS attack trends for Q2 2020. https://blog.cloudflare.com/network-layer-ddos-attack-trends-for-q2-2020/. (2020).

[20] Cloudflare. 2021. IP Ranges. https://www.cloudflare.com/ips/. (2021).

[21] Team Cymru. 2021. The Bogon Reference. https://team-cymru.com/community-services/bogon-reference/. (2021).

[22] Jakub Czyz, Michael Kallitsis, Manaf Gharaibeh, Christos Papadopoulos, Michael Bailey, and Manish Karir. 2014. Taming the 800 pound gorilla: The rise and decline of NTP DDoS attacks. In *ACM Internet Measurement Conference (IMC '14).* 435–448.

[23] Amogh Dhamdhere and Constantine Dovrolis. 2010. The Internet is flat: modeling the transition from a transit hierarchy to a peering mesh. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT '10).* 1–12.

[24] Christoph Dietzel, Matthias Wichtlhuber, Georgios Smaragdakis, and Anja Feldmann. 2018. Stellar: network attack mitigation using advanced blackholing. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies.* 152–164.

[25] Colin Dixon, Thomas E Anderson, and Arvind Krishnamurthy. 2008. Phalanx: Withstanding Multimillion-Node Botnets.. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '08).* 45–58.

[26] Cody Doucette. 2021. *An Architectural Approach for Mitigating Next-Generation Denial of Service Attacks.* Ph.D. Dissertation. Boston University.

[27] Arthur Fabre. 2019. L4Drop: XDP DDoS Mitigations. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/. (2019).

[28] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. 2015. Bohatei: Flexible and Elastic DDoS Defense.. In *USENIX Security Symposium (SEC '15).* 817–832.

[29] Paul Ferguson. 1998. *Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing.* RFC 2827.

[30] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM ToN* (1993), 397–413.

[31] Qiaobin Fu. 2020. *High-performance software packet processing.* Ph.D. Dissertation. Boston University.

[32] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. 2015. Comparison of frameworks for high-performance packet IO. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15).* 29–38.

[33] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted Linux kernel extensions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19).* 1069–1084.

[34] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. 2008. The Flattening Internet Topology: Natural Evolution, Unsightly Barnacles or Contrived Collapse?. In *International Conference on Passive and Active Network Measurement (PAM '08).* 1–10.

[35] Deli Gong, Muoi Tran, Shweta Shinde, Hao Jin, Vyas Sekar, Prateek Saxena, and Min Suk Kang. 2019. Practical verifiable in-network filtering for DDoS defense. In *IEEE International Conference on Distributed Computing Systems (ICDCS '19).* 1161–1174.

[36] Google. 2021. Where can I find Compute Engine IP ranges? https://cloud.google.com/compute/docs/faq#find_ip_range. (2021).

[37] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 53, 2 (2015), 90–97.

[38] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review (CCR)* 40, 4 (2010), 195–206.

[39] Richard Hummel, Carol Hildebrand, Hardik Modi, Gary Sockrider, Roland Dobbins, Steinthor Bjarnason, Jill Sopko, Suweera DeSouza, Ivan Bondar, and Oliver Daff. 2020. *NETSCOUT Threat Intelligence Report for 2H 2019.* Technical Report. Arbor Networks.

[40] Hurricane Electric Internet Services. 2021. IP Transit Service. https://www.he.net/ip_transit.html. (Dec. 2021).

[41] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. 2006. Lua 5.1 reference manual. (2006).

[42] Intel Data Plane Development Kit [n. d.]. Intel Data Plane Development Kit. https://www.dpdk.org/. ([n. d.]).

[43] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. 2013. The crossfire attack. In *IEEE Symposium on Security and Privacy (S&P '13).* 127–141.

[44] Brian Krebs. 2016. The Democratization of Censorship. https://krebsonsecurity.com/2016/09/the-democratization-of-censorship. (2016).

[45] Xin Liu, Xiaowei Yang, and Yanbin Lu. 2008. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *ACM SIGCOMM.* 195–206.

[46] Zhuotao Liu, Hao Jin, Yih-Chun Hu, and Michael Bailey. 2016. MiddlePolice: Toward enforcing destination-defined policies in the middle of the Internet. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 1268–1279.

[47] Ratul Mahajan, Steven M Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. 2002. Controlling high bandwidth aggregates in the network. *ACM SIGCOMM CCR* 32, 3 (2002), 62–73.

[48] Marek Majkowski. 2017. Meet Gatebot - a bot that allows us to sleep. https://blog.cloudflare.com/meet-gatebot-a-bot-that-allows-us-to-sleep. (2017).

[49] Marek Majkowski. 2018. How to drop 10 million packets per second. https://blog.cloudflare.com/how-to-drop-10-million-packets/. (July 2018).

[50] Denis Makrushin. 2017. The cost of launching a DDoS attack. https://securelist.com/analysis/publications/77784/the-cost-of-launching-a-ddos-attack/. (2017).

[51] Rob Marvin. 2019. Chinese DDoS Attack Hits Telegram During Hong Kong Protests. https://www.pcmag.com/news/chinese-ddos-attack-hits-telegram-during-hong-kong-protests. (2019).

[52] MaxMind, Inc. 2021. GeoIP Products. https://dev.maxmind.com/geoip/. (2021).

[53] John C. McCallum. 2021. Historical Memory Prices 1957+. https://jcmit.net/memoryprice.htm. (2021).

[54] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX Winter*, Vol. 46.

[55] Rui Miao, Rahul Potharaju, Minlan Yu, and Navendu Jain. 2015. The dark menace: characterizing network-based attacks in the cloud. In *ACM Internet Measurement Conference (IMC '15)*. 169–182.

[56] NexusGuard. 2020. NexusGuard DDoS Threat Report, 2020 Q1. https://blog.nexusguard.com/threat-report/ddos-threat-report-2020-q1. (2020).

[57] Packet Clearing House. 2021. Internet Exchange Point Growth by Country. https://www.pch.net/ixp/summary_growth_by_country. (2021).

[58] Bryan Parno, Dan Wendlandt, Elaine Shi, Adrian Perrig, Bruce Maggs, and Yih-Chun Hu. 2007. Portcullis: protecting connection setup from denial-of-capability attacks. In *ACM SIGCOMM*. 289–300.

[59] PeeringDB. 2021. The Interconnection Database. https://www.peeringdb.com/. (2021).

[60] PF_RING. 2021. PF_RING: high-speed packet capture, filtering and analysis. https://www.ntop.org/products/packet-capture/pf_ring/. (2021).

[61] Providers that offer BGP sessions 2021. Providers that offer BGP sessions. http://bgp.services/. (2021).

[62] James Reinders. 2005. VTune performance analyzer essentials. (2005).

[63] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *USENIX Security Symposium (SEC '12)*. 101–112.

[64] Dongwon Seo, Heejo Lee, and Adrian Perrig. 2013. APFS: adaptive probabilistic filter scheduling against distributed denial-of-service attacks. *Computers & Security* 39 (2013), 366–385.

[65] Amazon Web Services. 2021. AWS IP address ranges. https://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html. (2021).

[66] Lumin Shi, Devkishen Sisodia, Mingwei Zhang, Jun Li, Alberto Dainotti, and Peter Reiher. 2019. The Catch-22 Attack. In *IEEE Annual Computer Security Applications Conference (ACSAC '19)*.

[67] Nikita Shirokov and Ranjeeth Dasineni. 2018. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. (2018).

[68] The Spamhaus Project SLU. 2021. The Spamhaus Don't Route Or Peer Lists. https://www.spamhaus.org/drop/. (2021).

[69] Ahren Studer and Adrian Perrig. 2009. The coremelt attack. In *European Symposium on Research in Computer Security (ESORICS '09)*. 37–52.

[70] Thomas Vissers, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. Maneuvering around clouds: Bypassing cloud-based security providers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1530–1541.

[71] David Wragg. 2020. Unimog - Cloudflare's edge load balancer. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/. (2020).

[72] Abraham Yaar, Adrian Perrig, and Dawn Song. 2004. SIFF: A stateless internet flow filter to mitigate DDoS flooding attacks. In *IEEE Symposium on Security and Privacy (S&P '04)*. 130–143.

[73] Xiaowei Yang, David Wetherall, and Tom Anderson. 2005. A DoS-Limiting Network Architecture. In *ACM SIGCOMM*. 241–252.

[74] Zhenjie Yang, Yong Cui, Baochun Li, Yadong Liu, and Yi Xu. 2019. Software-defined wide area network (SD-WAN): Architecture, advances and opportunities. In *IEEE International Conference on Computer Communication and Networks (ICCCN '19)*. 1–9.

[75] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. 2017. Taking the edge off with Espresso: Scale, reliability and programmability for global internet peering. In *ACM SIGCOMM*. 432–445.

[76] ZoomInfo report page on Hurricane Electric 2020. ZoomInfo report page on Hurricane Electric. https://www.zoominfo.com/c/hurricane-electric-inc/56495174. (2020).