Narrow - Distributed DDOS Prevention on EI

Ziyao Zhang
UC Berkeley
Berkeley, CA
ziyaoz@berkeley.edu

Yichi Zhang
UC Berkeley
Berkeley, CA
billyz@berkeley.edu

ABSTRACT

Internet nowadays is still prone to DDoS attacks, and a defense to which requires an equally powered system as the attack. We present Narrow, a protocol built upon the Extensible Internet [3] architecture, as a way to deny DDoS in a distributed fashion. Narrow works by limiting bandwidth of an attacking host near the source from its service node, as described in EI, at the request of the victim. Unlike previous attempts on the same topics, we do not need any trusted component on the compromised attacker host. We implemented a system locally using Mininet and Open vSwitch. We evaluated the performance of the system, and found that with reasonable amount of memory, a service node running Narrow can stop an attack at its root without significant impact on the network's latency or bandwidth.

1 INTRODUCTION

1.1 Motivation

DDoS is still a problem on the Internet [4]. In 2020, AWS recorded an attack of 2.3Tbps, the largest they have seen at the time [11]. While not indefensible, massive DDoS attacks require a similarly massive distributed infrastructure to defend against. Not only does this infrastructure take a lot of engineering time, it is also very expensive to build, forcing small and medium content providers to purchase DDoS protection from a large provider.

With the Extensible Internet (EI) proposal [3], we see a new opportunity for the Internet infrastructure itself to offer DDoS protection to all participants in the network at a low cost. Specifically, the Extensible Internet proposal introduces 1) a new network component called the service node (SN) that we can leverage to filter attack traffic and 2) a powerful trust model that we can leverage to simplify our protocol design.

1.2 Background on Extensible Internet (EI)

Before we dive into the design of Narrow, we'll first reiterate parts of the EI architecture [3] that play a crucial role in our protocol.

L3.5, the service layer, is a layer built on top of packet delivery to provide the Internet service. The Internet service model as defined by EI not only includes connectivity (e.g. current & future versions of IP, NDN [6]) but also other in-network processing capabilities like caching and flow termination. The Narrow protocol is an L3.5 protocol that provides DDoS protection to other L3.5 protocols.

Service Nodes, or SNs, provide the Internet service to end hosts. When a new host connects to EI, it runs a discovery protocol to find an SN and receives Internet service via the SN. Since SNs provide Internet service to hosts, they implement all layers in the stack, including L3.5. On the other hand, because switches and routers only provide packet delivery, they implement L3 and below.

2 PROTOCOL DESIGN

2.1 Trust Model

Although not entirely explicitly outlined in the original EI paper [3], the EI architecture is conducive to a trust model that can greatly simplify the design of a DDoS protection protocol. The trust model can be summarized as:

- Hosts and SNs are typically in a customer / provider relationship bound by contracts, though sometimes hosts and SNs could be owned by the same entity (e.g. cloud providers).
- SNs disallow address spoofing.
- Service node providers (SNPs) all directly peer with each other.
- Both SN-Host and SN-SN connections guarantee integrity and authenticity of their messages.

2.2 An Important Assumption

For ease of explanation, we assume each SNP only owns one SN starting from the next subsection. In section 7, we lift this limitation and discuss how to scale Narrow to Internet scale.

2.3 Narrow Protocol

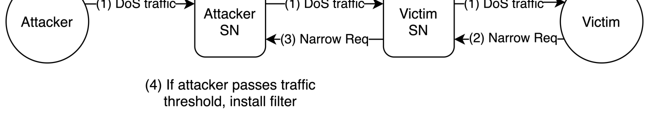

Figure 1: Protocol Overview

We now describe the normal operation of our protocol. We start with a simple topology (Figure 1) of two service nodes and two hosts, but it can be easily generalized to multiple attackers and multiple victims. We designate host H_A as the attacker and host H_V as the victim. Similarly, service node SN_A serves the attacker and SN_V serves the victim. When the attacker starts sending traffic towards the victim (1), the victim identifies that it is under attack and sends a Narrow packet destined to the attacker (2).

A Narrow packet contains the following fields:

$$\mathbf{id_{L3.5}, addr_A, addr_V, t_{end}}$$

where $id_{L3.5}$ is the identifier for the L3.5 through which the victim no longer wants to receive the attacker's traffic, $addr_A$ and $addr_V$ are the addresses of the attacker and the victim under the schema of the specific L3.5 protocol, and t_{end} is the time at which the Narrow request expires, a 64 bit integer that represents milliseconds since epoch. We deem that t_{end} should be fewer than 5 seconds from present time on SN_A to avoid availability concerns.

Upon receiving a Narrow packet, SN_V directly forwards it to SN_A, who, after verifying that the attacker has been sending enough traffic to the victim, will install a filter that expires at t_{end}.

3 PROTOCOL IMPLEMENTATION

After contacting the original authors of EI [3], we learned that the EI reference implementation is still in the works. Since finishing the EI implementation is out of scope for this paper, we start by specifying the components in an EI implementation that we utilize in our Narrow implementation. While major changes in the EI implementation will probably force the Narrow implementation to change as well, we hope that our work provides a good foundation for other possible implementations of both Narrow and EI.

3.1 EI Implementation

Pipe Terminus: When a packet first arrives at an SN or a host, we assume they first go through a centralized component that sorts the packets based on their L3.5 identifiers. Borrowing a terminology from McCauley et al. [9], we call this component the pipe terminus.

L3.5 Services: We assume each L3.5 service are logically separated from each other and get their own execution resources. We don't make any assumptions on the execution environment of an L3.5 service. Conceivably, an L3.5 service could be running on a thread, a process, a container, a VM, or even a cluster of machines.

Inter-service RPC: We assume each L3.5 service can define a set of remote procedure calls (RPCs) for other L3.5 services to invoke.

SN Metadata: We assume the EI software provides a mechanism for an L3.5 service to query what other L3.5 services are running on the SN / host and what RPCs do these L3.5 services support.

For all L3.5 services that want to support Narrow, they should implement the following RPCs:

- `prev_SNs(addr)`: returns a list of SNs that a packet from `addr` could have come from. The list does not include the current SN, and the list could be empty.
- `next_SNs(addr)`: returns a SN that is closer to `addr` than the current SN.
- `get_filter(src_addr, dst_addr)`: returns a filter that blocks traffic from `src_addr` to `dst_addr`. The format of the filter depends on the EI implementation. For example, if the EI implementation is built on top of Open vSwitch [10], the filter returned could be an OpenFlow rule. Another example is returning an eBPF program.

3.2 Filter Installation

Since the pipe terminus is a component that every packet goes through, we think it is a good candidate on which to implement a generalized filter mechanism. When a Narrow packet arrives at an SN, the pipe terminus forwards it to the Narrow module. Then, the Narrow module determines which L3.5 service does this Narrow request pertain to using $id_{L3.5}$ and queries the SN software to ensure that 1) this L3.5 service exist on this SN and 2) this L3.5 supports the required RPCs. For performance, this query could be cached.

Next, the Narrow module verifies whether the origin of the Narrow packet is legitimate. Specifically, if the Narrow packet came from a host (i.e. the current SN is SN_V), then the Narrow module will assume that the pipe terminus has handled user authentication and dropped spoofed packets, and the packet is automatically trusted. Otherwise, (i.e. the current SN is SN_A), the Narrow module calls `prev_SNs(addr)` with $addr_V$ and checks if the originating SN is in the returned list of SNs. If not, then the Narrow request is ignored.

Then, the Narrow module attempts to verify whether the attacker has sent enough traffic to the victim to warrant a filter to be installed. A specific threshold is derived in section 5.4. We propose that the SN keep a history of all previous flow bandwidth in a 10 second sliding window for this purpose, though conceivably the specific implementation of flow traffic history could vary. Furthermore, we argue that keeping all previous flows for 10s is doable because SN_A is located on the edge. Once the attacker is identified as sending too much traffic, the Narrow module calls `get_filter(src_addr, dst_addr)` on the requested L3.5 using $addr_A$ as the `src_addr` and $addr_V$ as the `dst_addr`. Finally, the Narrow module installs the filter on the pipe terminus with an expiration time t_{end}.

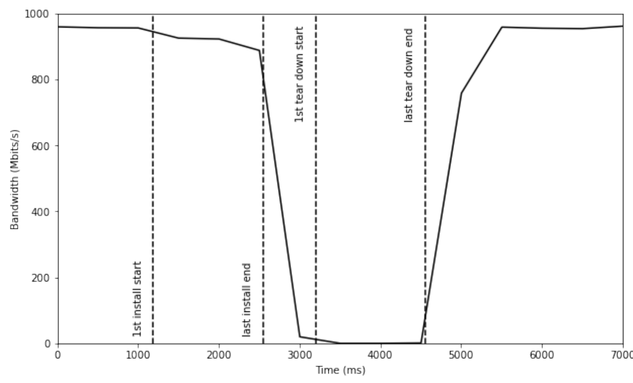

Figure 2: Bandwidth over time for a sample DDoS attack involving 100 attackers and 1 victim

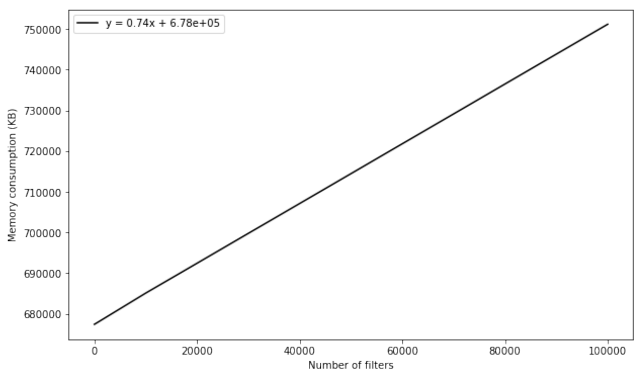

Figure 3: Number of OvS Filters vs. Memory Consumption

3.3 Packet Routing

When the Narrow module on SN_V forwards a Narrow packet to SN_A, it calls `next_SNs(addr)` with $addr_A$ to get a list of SNs. Then, the Narrow module sends a copy of the Narrow packet to each of the returned SNs.

4 EXPERIMENTAL RESULTS

There are two goals that we want to achieve through experimentation. The first goal is functional verification: we want to know if the Narrow protocol is effective in protecting against the victim in basic DDoS scenarios. The second goal is to determine bottlenecks in the Narrow protocol. As a DDoS prevention protocol, we don't want Narrow itself to be vulnerable to DDoS. Knowing the bottlenecks informs us of the weakest points in the face of an attack and additional design choices we need to make to prevent abuse. As an example, it is important to know how many filters are available at a given time per SN_A per attacker, as it positively relates how hard will certain types of attack be (see section 5.4).

4.1 Experiment Setup

We implemented a simplified local network system using Mininet [8] and Open vSwitch (OvS) [10]. The entire setup runs on an AWS EC2 c5a.2xlarge machine with 8 cores and 16GB of memory. We use Mininet to start an arbitrary number hosts as attackers and victims, all connected via an OvS Switch, which acts as an SN. We use rules on the virtual switch as filters. We install and uninstall the rules by invoking the OvS command line tools from a python script that listens on the virtual switch raw socket. Upon seeing a Narrow packet, it installs a filter and sets up a callback that uninstalls the filter at t_{end}. We use iperf on both the attacker and the victim to transmit the attack traffic.

4.2 Functional Verification

To verify that the Narrow protocol works in a basic attack, we start 100 attacker hosts and 1 victim host. The attacker hosts each have 100Mbit/s links connected to the SN, while the victim host has a 1000Mbit/s link. Without Narrow, the attackers oversubscribe the victim's network in a 10:1 ratio. After the attack starts, we manually trigger a Narrow client on the victim to send Narrow packets targeted towards the attackers. All Narrow requests had an end time 2 seconds after the time that the packets were each sent. As seen in Figure 2, the bandwidth reaches 0 after all filters are installed, and bounces back up after all filters expired. In this experiment, we did not trigger the victim to send another round of Narrow packets. In a real world deployment, the victim could have kept sending Narrow packets to keep all attackers silent, or it could have randomly chosen a subset of attackers to Narrow so that its bandwidth is not oversubscribed while decreasing the risk of categorizing a legitimate customer as an attacker. We envision a wide range of automated Narrow clients that determine what hosts count as attackers and thus need to be narrowed, but the specific strategy of such clients is left as future work.

4.3 Filter Memory Consumption

We install many IP filters, each matching a different (source, destination) tuple. We then plot the memory consumption of the OvS daemon process. As seen in Figure 3, each filter takes up a constant space of 700 bytes. This means a service node with 16G assigned filter space can fit about 24 million filters using OvS as the pipe terminus. If we assume each SN services 1000 hosts, then each host can install 240K filters on average. Since a Narrow filter is rather minimal, we think there is a lot of space to optimize the memory taken up by a single filter. However, as shown in later sections, memory is not our bottleneck, so we curb our optimization enthusiasm for now.

4.4 Filter Installation Throughput

OvS supports two modes of installing rules. For some number of rules we want to install, we can either invoke the command line tool for each one, or batch up the requests and invoke the tool once. We initially used the former method, but found it unbearably slow - 18 seconds for 10000 filter installations. Putting all the rules into a file and batch executing it significantly increases performance, lowering the run time down to 0.6 seconds. We think this performance increase comes from fewer process spawns, and maybe query optimization within OvS. In a real-world deployment, we think batching should be used to speed up filter construction at least on OvS. Furthermore, we think batching will be a dynamic process based on traffic level and Narrow request frequencies.

Despite drastic improvements over adding filters one-by-one, batching filter construction time still limits our total filter capacity more than memory. To install 240K filters (the number of filters that can fit in 16GB of memory), we need 14.4 seconds. If we assume each filter expires in 5 seconds, then the maximum number of active filters on an SN running OvS would be $5/0.6 * 10000 = 83,333$ filters - about 83 filters per host.

4.5 Filter Installation Delay

We tested how long does it take for a Narrow request takes effect. We have two hosts, h_1 is flood-pinging h_2 while h_2 makes a command line call to block the traffic. At $t = 0ms$, h_2 initiates the command line call, which finishes at $t = 3.2ms$. The last ping is at $t = 4.8ms$. During the 4.8 milliseconds, about 400 ping packets made it through to h_2. Given that OvS is only used as a prototype, we think we are not too far off by choosing the end time in Narrow in the unit of milliseconds, though we should definitely try to improve this number in future work.

4.6 Filter Effect on Datapath Delay

We tested how the number of filters installed affects the delay of the network. For every 10000 filters installed, we ping from one host to another 20 times and report the average round-trip time. As shown in Figure 4, the delay is relatively constant throughout. We think this result is logical because OvS uses hashing to match a given packet to the filters we installed, thus making the lookup time asymptotically constant with respect to the number of filters.

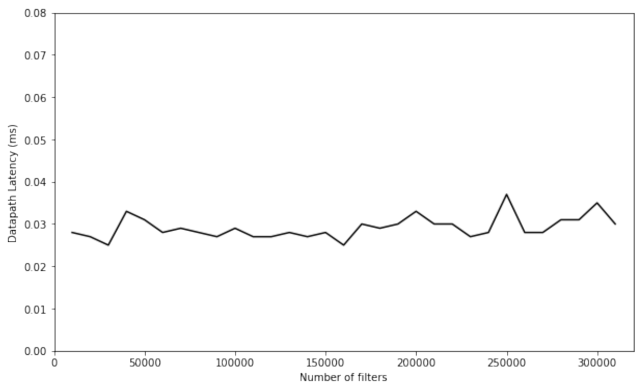

Figure 4: Number of OvS Filters vs. Datapath Latency

4.7 Experiment Summary

To briefly conclude, we have discovered the bottleneck of a given SN's filter capacity (running OvS) as about 83K per SN, and this limit is due to how quickly we can install filters. We also verified that the Narrow protocol is functional against a simulated DDoS attack.

5 ATTACK SCENARIOS

5.1 Initial Response

Knowing how fast can we Narrow attackers is important. A Narrow packet takes up about 202 bytes including L2 and L3 headers: we imagine the L3.5 header to contain a 8 byte L3.5 identifier and two 32 byte IPv6 addresses, and we imagine the payload to be a 8 byte L3.5 identifier, two 32 byte IPv6 addresses, and a 4 byte denoting the lower bits of t_{end}. Assuming a victim has an upstream bandwidth of 10Mbps, and the victim is not bottlenecked by attacker classification, the victim can send out about 6,500 Narrow packets per second. According to the report on the 2018 GitHub DDoS attack [7], "tens of thousands" of end points were involved. This means the victim can send out a Narrow request to each one of them within about 6 seconds if we assume 40,000 attackers.

5.2 Replay Attack

Replay attacks are initiated by an on-path adversary by sending a series of packets it has seen before as an attempt to impersonate the original sender. This is not a problem within Narrow because Narrow packets are idempotent (i.e. the effect is the same no matter how many Narrow packets are received by an SN) due to t_{end} being absolute. Furthermore, given that the connections between Host-SN and SN-SN are backed by tunnels that guarantee integrity and authenticity of the messages being transmitted (see section 2.1), any modifications to those messages will discovered. Therefore, we believe that Narrow is not prone to replay attacks.

5.3 Spoofing (Using Narrow to DDoS)

5.3.1 Host Address Spoofing. Host address spoofing refers to a malicious host attempting to send a Narrow packet using an $addr_V$ that it does not own. Given that all hosts need to authenticate with their SN before receiving EI service, the SNs directly serving a host have the authoritative information on whether a Narrow packet is spoofed. As long as they don't allow host address spoofing (which we outlined in section 2.1), Narrow is not prone to this problem.

5.3.2 SN Route Spoofing. SN route spoofing refers to a misconfigured or malicious SN claiming to service a set of host addresses (in an L3.5 address space) that it does not have the authority of servicing. This could lead Narrow into forwarding a Narrow request to a wrong SN or trusting a Narrow packet when it shouldn't. We argue that since Narrow have no control over how the address space of a given L3.5 service is partitioned between SNs or SNPs, the best that Narrow can do is to ask the protected L3.5 of its address space partitions, hence the requirement of `prev_SNs(addr)` to be implemented by the protected L3.5s. Furthermore, since SNPs all directly peer with each other, we argue that for an address space partitioned to SNPs and have partitions that change slow enough (much like IP addresses partitioned to ASes), `prev_SNs(addr)` could simply be a static lookup while providing good protection to SN route spoofing.

5.4 Filling up Filter Space of Attacker SN (DDoS against Narrow)

According to our experimental results in section 4, the number of filters available on SN_A is about 83 per host. This means that in the worst case where all the hosts on an SN are compromised, each host need to annoy 83 victims so that they all send a Narrow packet to SN_A before launching the attack against the actual target. Since 83 is not a huge number, this is would make Narrow useless if it worked. To defend against this attack, the Narrow protocol stipulates that SNs should verify whether an attacker has actually been sending enough traffic to the victim in the past. We now strive to answer the question of to choose this bandwidth threshold.

Let b bits/s be the average bandwidth of a single compromised attacker host without Narrow being available on EI, b_N bits/s be the amount of bandwidth available to that attacker with Narrow being available, f be the number of filters available per host on SN_A, and x be the proportion of bandwidth that the attacker needs to spend on the victim for SN_A to grant the victim Narrow permission on the attacker. This means if the attacker sends more than xb bits/s to the victim, the attacker will be prone to Narrow, where $0 \leq x \leq 1$.

If the attacker decides to simply send below the rate of xb bit/s to avoid being Narrowed, then we obtain an lower bound on b_N (it's an lower bound here because we know the attacker can at least use this much bandwidth by exploiting Narrow):

$$b_N \geq xb$$

If the attacker decides to annoy some victims and attempt to fill up the Narrow filter capacity at SN_A, then the attacker needs to send to f victims using xb bits/s each. This leaves the attacker $b - fxb = (1 - fx)b$ bits/second to perform the actual attack. Thus, we obtain another lower bound on b_N:

$$b_N \geq (1 - fx)b$$

We want to choose x so that the maximum bandwidth available to an attacker through either method is minimized:

$$\underset{x}{\operatorname{argmin}} \max(xb, (1 - fx)b)$$

$$\hat{x} = \frac{1}{f + 1}$$

Although this is not an exhaustive analysis, we have thus proven that in a particular attack scenario the number of filters inversely corresponds to the bandwidth available to an attacker. There is probably some leeway to set x significantly lower if the probability of all the hosts on an SN being compromised is low.

6 ECONOMIC ANALYSIS

6.1 Deployment Incentives

For datacenters and cloud providers, deploying Narrow on their SN allows both tenants and themselves to Narrow attack traffic, thus saving both money. Given the great network visibility brought by the large scale of some cloud providers, they can even provide better service to differentiate themselves from the competition by automatically Narrowing an attack so that their clients won't have to. Last but not least, cloud providers can avoid some of the liability and limit the damage in case some of their hosts get compromised and start sending DDoS traffic.

For ISPs, they benefit from saving traffic cost. When a downstream service node forwards a Narrow request, the ISP service node can block the traffic at its first hop, thus reducing the load of their backbones and potentially providing better services to their customers.

For content providers and end users, the Narrow protocol is strictly beneficial. For both, they benefit from a free DDoS protection mechanism built into the Internet. Furthermore, an end-user's computer becomes less valuable to attackers, which could decrease the incentive for attackers to compromise devices.

Globally, the benefit Narrow brings is straightforward: the duty of reducing garbage traffic is laid out to the very corners of the entire Internet. This effect is beneficial to everyone except the attackers: service providers no longer need to seek DDoS protection from a third party, users can worry less about their host sending out traffic leading to an increase in their Internet bills, ISPs will have less traffic travelling across their backbones, and datacenter servers should have less garbage pounding on their ports to process. Every party have the incentive to deploy Narrow.

6.2 Incremental Deployability

Although we certainly hope that Narrow get to be one of the pre-installed protocol when EI rolls out, it might need to start from a few and expand, and thus we designed the Narrow protocol with incremental deployability in mind. First, unlike previous proposals, EI does not require a trusted component on the attacker host. Second, hosts have an easy way to figure out which networks have Narrow enabled - they can send a Narrow packet to see if attack traffic from the target source has stopped. Third, the effectiveness of Narrow scales linearly with the number of attackers served by an SN supporting Narrow. As long as the SNs supporting the protocol keeps increasing, end users will enjoy better DDoS protection without explicitly taking action, which should propel more and more SNPs to adopt Narrow.

7 SCALING NARROW

In previous sections, we considered the simplified case where each SNP only has one SN. In this section, we consider how to scale Narrow to Internet scale. We use a similar convention as before, where we denote the service node provider that provides service (unknowingly) to the attacker as SNP_A and the SNP that provides service to the victim as SNP_V.

7.1 Address Wildcards

While we described Narrow from an end-to-end perspective up till this point, we do not intend Narrow to be a pure end-to-end protocol. In fact, we believe that SNPs should have the ability to Narrow unwanted traffic for their clients without the client initiating the process. Given that SNPs have a complete view of their network, they have the potential to make better decisions on which hosts could be categorized as attackers. However, the large scale of some of the SNPs now becomes a problem. If an SNP wants to protect all of its clients, it would be really cumbersome to send a Narrow packet on behalf of each client. Therefore, we propose that Narrow should support wildcards in $addr_V$. However, wildcards in $addr_V$ are reserved for service node providers (SNPs) and large customers with special arrangements with SNPs because a host should not be able to Narrow traffic on behalf of other unrelated hosts under the same SNP.

7.2 Routing & Route Verification

Given that more than one paths are available from SN_V to SN_A, there is a question on which path should SN_V choose to forward the Narrow packet. Furthermore, the attacker host may not be connected to just one SN_A, but could be connected to more than one SNs or even SNPs. Although there are no transit SNPs in the EI architecture, with more than one SN in one SNP, SN_A is no longer directly connected to SN_V. As a consequence, SN_V may not know the addresses of SN_A, only where to send a packet that gets it one hop closer. We propose that for each attacker-connected SNP_A, SN_V could choose any edge SN that belongs to SNP_A and forward the Narrow packet to that SN. This edge SN can then broadcast the Narrow packet to all SN_As within its own SNP. This does imply that SNPs typically shouldn't allow a client to use any SN and will instead assign a set of SNs that a client is allowed to use, so that SNP_As could avoid having to broadcast every Narrow request to every SN.

Implementation-wise, instead of a list of possible SNs that a packet could have been transmitted from, the return value of `prev_SN(addr)` can also include SNP specifiers or SN address wildcards so that verifying a Narrow packet is quicker. We leave a precise specification for both of the SNP specifier or SN address wildcards to the EI reference implementation. `next_SN(addr)` also needs to be slightly modified. Instead of returning just one next-hop SN, it should return a list of next-hop SNs, with each SN being from a different SNP_A.

8 RELATED WORK

There have been a lot of efforts to defend against DDoS. The first paper that proposed the idea to send explicit packets to request a bandwidth limit is **Active Internet Traffic Filtering** (AITF) [2]. AITF operates between the gateways of both the traffic recipient and the traffic sender. When a recipient wants to shut up a sender, it tells this intent to its gateway. Then, the recipient's gateway initiates a three-way handshake with the sender's gateway to block the sender's traffic. The problems of AITF as we see include 1) AITF only offers to block the traffic and suggests a minute-scale block, increasing the risk of shutting up legitimate hosts, and 2) AITF requires both end-hosts to support AITF: without a timely reaction from the host being blocked, the gateway will disconnect the host. We alleviate the first problem by suggesting the effective time to be second-scale. For the second one, we are basing our protocol over EI, therefore we do not need any support from the Narrowed host.

ShutUp: End to End Containment of Unwanted Traffic [5] is another attempt that requires support from both end-hosts - one of which is supposedly sending malicious traffic.

Accountable Internet Protocol (AIP) [1] is a more general approach towards the malicious activities. AIP thinks the reason for these attacks is the un-accountability of the Internet, and it proposes an accountable Internet by combining public-key cryptography and a two-layered addressing scheme so that it is harder to spoof addresses - the source of a large part of malicious activities. We think that AIP introduces a lot of overhead in terms of both header size and processing time: AIP header contains a standard IP header plus more than 120 bytes, and it is well known that public-key cryptography is not a fast process. AIP also requires both end-hosts to have a smart NIC and be cooperative, which is not always the case within this context.

9 ACKNOWLEDGEMENTS

We want to thank James McCauley, Sylvia Ratnasamy, Aurojit Panda, and Scott Shenker for providing feedback on an early version of Narrow. We are also grateful for Sylvia and CS268 in providing us both an opportunity and the AWS credits for working on this project.

REFERENCES

[1] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. 2008. Accountable Internet Protocol (Aip). In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/1402958.1402997

[2] Katerina Argyraki and David R. Cheriton. 2005. Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, USA, 10.

[3] Hari Balakrishnan, Sujata Banerjee, Israel Cidon, David Culler, Deborah Estrin, Ethan Katz-Bassett, Arvind Krishnamurthy, Murphy McCauley, Nick McKeown, Aurojit Panda, Sylvia Ratnasamy, Jennifer Rexford, Michael Schapira, Scott Shenker, Ion Stoica, David Tennenhouse, Amin Vahdat, and Ellen Zegura. 2021. Revitalizing the Public Internet by Making It Extensible. *SIGCOMM Comput. Commun. Rev.* 51, 2 (May 2021), 18–24. https://doi.org/10.1145/3464994.3464998

[4] Vivek Ganti and Omer Yoachimik. 2021. DDoS attack trends for 2021 Q1. (04 2021). https://blog.cloudflare.com/ddos-attack-trends-for-2021-q1/

[5] Saikat Guha, Paul Francis, and Nina Taft. 2008. *ShutUp: End-to-End Containment of Unwanted Traffic.* Technical Report http://hdl.handle.net/1813/11101. https://www.microsoft.com/en-us/research/publication/shutup-end-to-end-containment-of-unwanted-traffic/

[6] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. 2009. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1658939.1658941

[7] Sam Kottler. 2018. February 28th DDoS Incident Report. (Mar 2018). https://github.blog/2018-03-01-ddos-incident-report/

[8] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. Association for Computing Machinery, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/1868447.1868466

[9] James McCauley, Yotam Harchol, Aurojit Panda, Barath Raghavan, and Scott Shenker. 2019. Enabling a Permanent Revolution in Internet Architecture. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3341302.3342075

[10] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open VSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, USA, 117–130.

[11] Jon Porter. 2020. Amazon says it mitigated the largest DDoS attack ever recorded. (06 2020). https://www.theverge.com/2020/6/18/21295337/amazon-aws-biggest-ddos-attack-ever-2-3-tbps-shield-github-netscout-arbor