X-Pro: Distributed XDP Proxies against DDoS

Syafiq Al Atiiq
syafiq_al.atiiq@eit.lth.se
LTH, Lund University
Lund, Sweden

Christian Gehrmann
christian.gehrmann@eit.lth.se
LTH, Lund University
Lund, Sweden

Abstract

The concept of Internet of Things promised to connect billions of devices around the globe to the internet and capable to communicate each other. However, the device itself is especially vulnerable to be used as a bot to launch a massive Distributed-Denial-of-Service attack. The main problem with the IoT devices are usually resource-constrained, such that implementing a strong security protection is a non-trivial problem. Hence, they are easily turned from a usable embedded device into a bot for launching DDoS attack.

A source-based DDoS detection could help reducing the impact of DDoS attack by blocking traffic from the source address of adversary. However, determining the legitimacy of the traffic close to the source is a hard problem, because the volume might not be big enough to be deemed as an attack. Also, from the economic standpoint, placing a classifier node close to the source is not incentivising the service provider. In this paper, we present a reverse-firewall proxy on the cloud working together as a distributed synchronous proxies to mitigate DDoS attack. We evaluate the performance of our solution from several point of view. In conclusion, our solution offers a better decision making in terms of an attack traffic, as well as preserving the resource of the victims from a severe DDoS attack.

Keywords reverse-proxy, internet of things, denial of service, security

ACM Reference Format:
Syafiq Al Atiiq and Christian Gehrmann. 2020. X-Pro: Distributed XDP Proxies against DDoS. In *Proceedings of SEC@SAC '21: The Security Track at the ACM Symposium on Applied Computing (Gwangju '21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Gwangju '21, March 22–26, 2021, Gwangju, Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

From September 2016, a tremendous distributed denial of service attack launched against several high-profile website on the internet, namely OVH [8], Dyn [5], and Krebs on Security [9]. Surprisingly, the source of the traffic came from a huge amount of the embedded devices turned into bots. These bots controlled by a master process, which later known as Mirai botnets [1]. The master process scanned the whole internet and infects embedded devices that still running with insecure default password.

On the other hand, the emerging of Software Defined Networking (SDN) and Network Function Virtualization (NFV) bolster the shifting perspective of networking gears from a hardware oriented into a more "software based" product. This way, a system designer has more freedom on how they develop their networks, since most of the Network Functions (NF) can be installed into a white label hardware, i.e. standard commodity servers. Also, as the demand of the network capacity grows, the complexity of this "software based" network is also growing rapidly.

A software based network function also needs a fast packet processing function to achieve the same set of standard the hardware oriented networking gears already did. A normal packet processing procedure that rely on the user-space application in linux would be barely acceptable in terms of the performance. Therefore, a special purpose tool, namely Data Plane Development Kit (DPDK) [7], solves this exact problem by entirely by-passing the operating system and let an application govern the network hardware directly. This comes at a price of dedicating a single (or more) CPU in the systems solely for packet processing function. Also, as the OS is by-passed, the application does not have the advantages of running many useful linux kernel module in the system.

An alternative to the previous solution is eXpress Data Path (XDP) [6], where the fast packet processing is still achieved while keeping the operating system networking stack at the same time. The idea is to have an execution environment at the earliest possible point where packet can be tapped and apply a specific logic safely in the kernel context.

In this paper, we introduce X-Pro, a well coordinated XDP proxies running together as a distributed systems to counteract DDoS at the earliest possible point. The detection happens even before the kernel using a standard linux stack inside a white label hardware. The distributed proxies are sharing information between each other to assist in the decision making of packet filtering. Different from prior-art

source based DDoS mitigation techniques, our solution does not happen close to the source, i.e. embedded device, but rather at the cloud. This allows sharing of a real-time traffic situation for a very large number of devices at the proxy cloud side. This in turn gives much better possibilities to detect network based DDoS attempts as traffic information about DDoS attack destination targets is available to all proxies.

We have made a proof-of-concept implementation with the following details. On the cloud side, we leverage eXpress data path [6] running on a baremetal hardware. It is possible to run our software on the higher stack in the operating system. However, we try to get a better performance by having only the packet processor down in the kernel-space. On the embedded device side, we have made an implementation on top of pycom platform [11]. Fipy [10], connected to an NB-IoT network, acting as the virtualized network interface. This comes without having to modify the firmware or software architecture in the main MCU. Also, as the protocol itself is general, it can be implemented on any platform, without the necessity to be run on the same hardware. As the proxy allows real-time information sharing in regards to the traffic condition, our protocol offers better ability to detect a possible DDoS attempt from a broader perspective. Our results show that the impact on the victim can be reduced significantly, even during a massive DDoS attack.

The rest of the paper is organized as follows. We discuss related works in Section II and background concepts in Section III. We provide the application scenario in Section IV. Section V presents scalable reverse firewall proxy, while in Section VI we provide a performance evaluation. Section VII draws our conclusions and anticipate future works.

2 Related Work

A major security problem in the current IoT networks is Distributed Denial-of-Service (DDoS) attacks where the IoT units are used as bots. In particular, we consider the problem of source based DDoS in the context of Internet of Things. It is considered as a non-trivial problem, since the source of the attacks can be distributed in different domains making it difficult for each of the source to detect and filter attack flows accurately. Also, it is difficult to differentiate between legitimate and attack traffic near the sources, since the volume of the traffic may not be big enough as the traffic typically aggregates at points closer to the destinations.

To counteract DDoS harnessing IoT as bots, there have been several approach proposed. In this section we take a closer look on what is already available solution and how they work. The first approach is machine learning based solution, where the idea comprises a machine learning in a way or another. Based on [13], the ML based solutions are classified into the following techniques : (i). Using IoT network behavior, and feature extraction [4]. The author claims that using IoT-specific network behaviors, i.e. finite number of endpoints and assuming that there is a well-specified interval between pakcets, it is possible to do feature selection with high veracity to detect DDoS in the network. The way the feature selection is performed is by implementing a low-cost machine learning algorithm in the middleboxes, i.e. home-router, local-gateway, switch, etc. However, we argue that if the underlying assumption does not met, we might end-up getting false positive (or negative) in the feature selection.

(ii). Harnessing SDN architecture to perform DDoS detection [3]. The authors present how SDN, combined with statistical approach can detect DDoS attempt in the network. The authors emphasizes that the work is not only to classify the bogus messages, but also more importantly, to misclassify the legitimate traffic. While the result shows that it is possible to employ statistical classification along with SDN to detect DDoS, the authors explicitly show that a meticulous consideration needs to be performed to select classifiers with lowest possible overhead. Otherwise, the computing resource to perform the classification itself will occupy the middlebox and worsen the traffic condition.

Another new approach is by detecting malicious traffic based on the heartbeat association [14]. The heartbeat association then used to create heartbeat network, the heartbeat associated graph and the attribute propagation algorithm based on the graph. While the result shows that the method can effectively detect the DDoS malignant host, the computation needs to be performed at the border of the router. Even though the computing resource might not be a problem, but we argue that a much larger resource would be needed when the DDoS attack is getting bigger.

Our solution fills the gap by effectively send all the traffic from the IoT device to our proxy while at the same time maintaining all the proxy well-informed on the traffic condition from a wider perspective. Also, as the informations are shared between proxies, proxy in different locations can catch-up the whole situation faster once DDoS happen in the network. This lead to a better DDoS mitigation in case the master of the botnets decide to launch the attack simultaniously using different bots from different locations.

3 Background

3.1 Express Data Path (XDP)

Express Data Path (XDP) [6] is a novel programmable packet processing, living inside the kernel-space. XDP has been part of the mainland linux kernel since 4.20. One would argue that a kernel by-pass solution (i.e. DPDK [7]) would be more suitable for a fast packet processing. However, by-passing the kernel would neglect its rich features that might be useful for an intermediate node processing, i.e. firewall, router, proxy, etc.

When the packet arrives, the device driver runs and eBPF program within XDP hook. This process happens even before

touching the packet data. During this early process, the eBPF program is able to determine the fate of the packet, in which one of the following would happen:

1. Drop the packet
2. Send the packet back out to the same interface
3. Redirect the packet, either to the same or different interface
4. Forward the packet to the userspace program, or
5. Allow the packet to be processed through a normal networking stack

In accordance with that, our aim in this paper is to block the bogus packets at the earliest possible point, even before processed in the kernel. The process of selecting which packet is legitimate and/or which packet is bogus is communicated through multiple BPF maps. A more detailed description about how we utilize XDP is explained in the next chapter.

The logic of eBPF program running inside XDP hook is written in high level language, i.e. C, and compiled into a bytecode. Kernel has the job to safeguard the eBPF program by verifying them. This verification happens during a load time of the program to the interface. If the code is safe, then the bytecode will be translated into a native machine instructions in order to achieve high performance.

4 Reverse-Firewall-Proxy

4.1 Application Scenario

4.2 Proxy system architecture

The proxy system consists of a set of proxies interconnected through an internal IP network. Each proxy is also assumed to have direct network connectivity. The proxy nodes shares filtering and also load information using a shared DB in the system. The solution is agnostic to a particular DB sharing method, but in our case, we use an in-memory database called Redis [12]. Our proxy architecture can be seen in the fig 1.

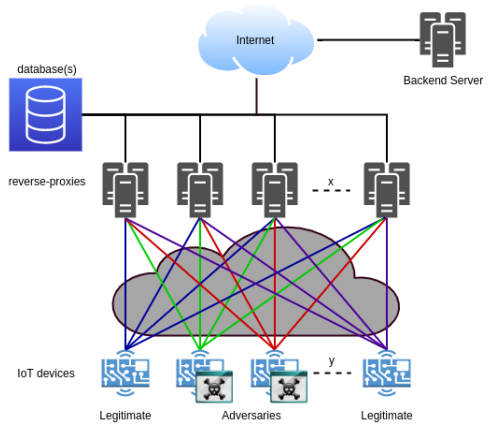

Figure 1. Mesh Network between IoT Devices and Reverse-Proxies

Each IoT unit must have the connectivity towards all available proxy, meaning that mesh network will be created between IoT units and the respective proxy. Also, as already mentioned previously, all proxies must have the same visibility in regards to which message needs to be blocked. Therefore, there has to be a mechanism to share information about this between proxies one way or another.

As we would like to block the bogus messages at the earliest possible point, i.e. in the XDP hook, those information needs to be shared between kernel-space and user-space (and vice versa). This is where the BPF maps comes into place. As shown in figure 2, each proxy has a running redis instance, which syncronize each other through a process on each proxy, namely *syncdb*. This process performs a synchronization protocol explained in subchapter 5.2. The syncronized data then communicated to the eBPF program using another process called *sync_maps_and_ldb*. This process synchronize an already synched local redis instance with the BPF maps, which can be accessed directly by the eBPF program in the kernel.

This way, we are able to pass the information between proxies without the need to send the invalid packet to the userspace. Hence, reducing the CPU utilization in the user-land and allocate the CPU to a more important task, i.e. packet filtering in the kernel. Our argument is that the less tasks performed in the userspace, the more CPU can be utilized by the XDP to block the invalid messages, hence we get more packet filtering capacity in the kernel. Figure 2 represent the connection between our packet filter mechanism with the syncronization function between databases.

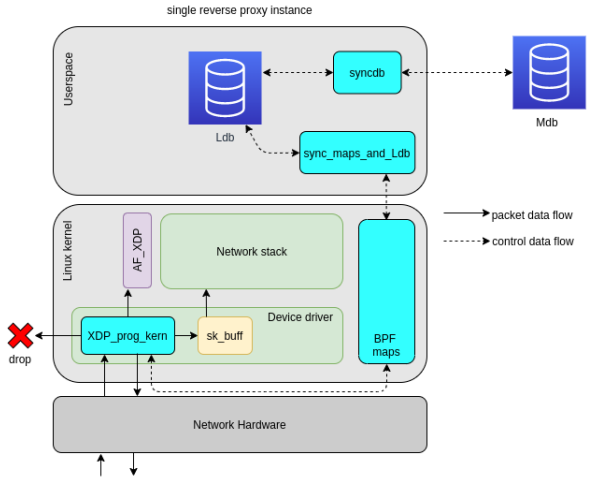

Figure 2. Syncronization between proxy and master DB

5 Algorithm

5.1 Notations

- Set of client devices in the system: U

- An client device: $u \in U$
- Set of proxies devices in the system: P
- Proxy: $p \in P$
- A unique index given to a client: i
- An client device with index i: u_i
- A proxy unique network address associated with a proxy: j
- A proxy with address j: p_j
- Work load of a proxy unit: W
- Work load threshold of a proxy unit: T_W
- Time stamp indicating the "oldest time" packet time for a particular (i, D_{addr}) pair: ts_1
- Time stamp indicating the "most recent" packet time for a particular (i, D_{addr}) pair: ts_2
- Packet counter or a particular (i, D_{addr}) pair: c
- Delta packet counter (internally within a proxy) counter for a particular (i, D_{addr}) pair: dc
- First filtering reset threshold used by a proxy: T_{T1}
- Filtering minimum measure time threshold used by a proxy: T_{T2}
- Second filtering minimum measure time threshold used by a proxy: T_{T3}
- Fourth filtering minimum measure time threshold used by a proxy: T_{T4}
- First packet maximum allowed frequency threshold used by a proxy: T_{F1}
- Second packet maximum allowed frequency threshold used by a proxy: T_{F2}
- Frequency division factor: r^2
- Destination IP address of a packet: D_{addr}

5.2 Proxy Synchronization Protocol

In order to have the same visibility on every proxy, one needs to have a synchronization function between them. This section explain in more detail those function, which has been implemented in our proof of concept inside the *syncdb* process.

For every time period t, each proxy p_j performs the synchronization procedure with the master database, described in the algorithm 1. The initial process started with p_j updates its current workload to the M_{DB}, which then replied by M_{DB} with the information containing the workload for all other proxies except p_j. p_j iterate through every (i, D_{addr}) inside the M_{DB}, and for each pair of (i, D_{addr}), p_j looks up at the local database L_{DB}. If L_{DB} contains information about (i, D_{addr}), it checks whether *mark* is equal to 1, together with if ts_1' minus ts_1 is greater than T_{T1}. If both conditions hold, both local value of $mark'$ and dc' is set to 0, and the value of ts_1, ts_2 and c is updated from the local value in p_j.

On the other hand, if both conditions do not hold, $mark'$ is set to 0. If local value ts_2' is less than M_{DB}'s value ts_2, the local value is updated with the remote master database value. Furthermore, if ts_2' minus ts_1 is greater than the fourth filtering minimum measure time threshold used by a proxy, T_{T4}, then the local value of ts_1' is updated through the following equation : $ts_2' - (ts_2' - ts_1)/r$. The rest of the process is described in the algorithm 1.

Algorithm 1 Proxy Synchronization Protocol

```
1:  p_j send the W value to M_DB
2:  p_j reads the current workload for all other proxies in the
    system, {W_k}, p_k ∈ P, k ≠ i
3:  p_j looks all the pair (i, D_addr), for u_i ∈ U
4:  for each (i, D_addr) in M_DB do
5:      if L_DB ∋ i, D_addr then
6:          if (mark' = 1 and ts_1' − ts_1 > T_T1) then
7:              < mark' = 0, dc' = 0 >
8:              < ts_1 = ts_1', ts_2 = ts_2', c = c' >
9:          else
10:             < mark' = 0 >
11:             if ts_2' < ts_2 then
12:                 < ts_2' = ts_2 >
13:             else if ts_2' − ts_1 > T_T4 then
14:                 < ts_1' = ts_2' − (ts_2' − ts_1)/r >
15:                 < c = ⌊c/r⌋, ts_1 = ts_1', ts_2 = ts_2' >
16:             else
17:                 < ts_2 = ts_2' >
18:             end if
19:             if ts_1' > ts_1 then
20:                 < ts_1' = ts_1 >
21:             else if ts_2' − ts_1' > T_T4 then
22:                 < ts_1 = ts_1' >
23:             end if
24:             < c' = c + dc', c = c', dc' = 0 >
25:         end if
26:     else
27:         < ts_1' = ts_1, ts_2' = ts_2 >
28:         < c' = c, dc' = 0, mark' = 0 >
29:     end if
30: end for
31: for each (i, D_addr) in L_DB do
32:     if M_DB ∌ i, D_addr then
33:         < ts_1 = ts_1', ts_2 = ts_2', c = c' >
34:     end if
35: end for
```

5.3 Packet Filtering Procedures

When the packet arrives at the proxy, the procedures described in the algorithm 2 takes place. All the process in this part, happens inside the kernel space, within the *xdp_prog_kern* mentioned in the figure 2. Therefore, the local database L_{DB} we mention in this section means a bpf map, not a redis instance.

Initially, p_j looks up and read the time stamps ts_1' and ts_2' for the record (i, D_{addr}) from the local database L_{DB}. If the record is found, p_j compares the difference between the

current time and ts'_2 with the first filtering reset threshold T_{T1}. If the difference value is bigger than T_{T1}, then ts'_1 is set to be the current time, c' and dc are reset to be 0 and $mark$ equal to 1.

On the other hand, if L_{DB} does not contain (i, D_{addr}), both ts'_1 and ts'_2 will be set to the current time, as well as resetting c', dc and $mark$ to 0. The next step is to increase both the counter c' and the counter difference dc values with 1, and set ts'_2 with the current time.

If the difference between ts'_2 and ts'_1 is larger than the second filtering minimum measure time threshold T_{T2}, then p_j compares the value of c' divided by the difference of ts'_1 and ts'_2 with the first packet maximum allowed frequency threshold, T_{F1}. If its greater than T_{F1}, the respective packet is dropped by the XDP, followed by sending an overload warning to the responsible administrator.

If the packet passed through the previous checks, further packet processing involving statistical measurement is performed. First, p_j calculates the minimum value of ts_1 corresponding to the same D_{addr} from all the set of U, namely ts_1*. Second, it calculates the maximum value of ts_2 and the sum of $c + dc$ from the same set U, namely ts_2* and $c*$ respectively. If $c*$ divided by ts_2* minus ts_1* is greater than T_{F2}, the respective packet (i, D_{addr}) is dropped, otherwise the packet is forwarded to D_{addr}.

Algorithm 2 Packet Filtering Procedures

```
<Lookup ts'_1, ts'_2, c for record (i, D_addr) in L_DB >
if record found then
    if t − ts'_2 > T_T1 then
        ts'_1 = t, c' = 0, dc = 0, mark = 1
    else
        <do nothing>
    end if
else
    ts'_1 = ts'_2 = t, c' = 0, dc = 0, mark = 0
end if
c' = c' + 1, dc = dc + 1, ts'_2 = t
if ts'_2 − ts'_1 > T_T2 then
    if c'/(ts'_2 − ts'_1) > T_F1 then
        <Drop packet>
        <Send an overload warning>
    end if
end if
ts_1* = min_{u_i ∈ U_{D_addr}} ts'_1_i
ts_2* = max_{u_i ∈ U_{D_addr}} ts'_2_i
c* = Σ_{u_i ∈ U_{D_addr}} (c_i + dc_i)
if ts_2* − ts_1* > T_T3 then
    if c* /(ts_2* − ts_1*) > T_F2 then
        <Drop packet>
    end if
end if
<forward packet>
```

6 Experimental Evaluation

To evaluate X-Pro, we have developed a proof of concept of the implementation. Both the proxies and the centralized database are implemented as a virtual machine on Fedora 30 operating system, running kernel version 5.6. Our implementation is as an open-source software at [2]. This section discusses the results from our experimental evaluation, following different scenarios in the proxy side. We show that, by placing the logic of distributed proxy at the earliest possible point, even before reaching the kernel, we can have a decent performance of dropping undesired packets.

Our experiment follows the scenario in figure 1, deployed over a local ethernet network. All of the VMs of the proxies and centralized database are running with one vCPU and 1024 MB of memory.

6.1 Results

7 Conclusion and Future Work

References

[1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[2] Syafiq Al Atiiq and Christian Gehrmann. 2020. X-Pro: Distributed XDP Proxies. (2020). https://github.com/syafiq/xpro

[3] J. N. Bakker, B. Ng, and W. K. G. Seah. 2018. Can Machine Learning Techniques Be Effectively Used in Real Networks against DDoS Attacks?. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. 1–6. https://doi.org/10.1109/ICCCN.2018.8487445

[4] R. Doshi, N. Apthorpe, and N. Feamster. 2018. Machine Learning DDoS Detection for Consumer Internet of Things Devices. In *2018 IEEE Security and Privacy Workshops (SPW)*. 29–35. https://doi.org/10.1109/SPW.2018.00013

[5] Scott Hilton. 2016. Dyn Analysis Summary Of Friday October 21 Attack. (2016). https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/

[6] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[7] Intel. 2020. Data Plane Development Kit (DPDK). (2020). https://dpdk.org

[8] Octave Klaba. 2016. Octave Klaba Twitter. (2016). https://twitter.com/olesovhcom/status/778830571677978624

[9] Brian Krebs. 2016. KrebsOnSecurity Hit With Record DDoS. (2016). https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/

[10] Pycom. 2020. Fipy. (2020). https://pycom.io/product/fipy/

[11] Pycom. 2020. Pycom Platform. (2020). https://pycom.io/solutions/hardware/

[12] Salvatore Sanfilippo. 2020. Redis. (2020). https://redis.io/

[13] K. Wehbi, L. Hong, T. Al-salah, and A. A. Bhutta. 2019. A Survey on Machine Learning Based Detection on DDoS Attacks for IoT Systems. In *2019 SoutheastCon*. 1–6. https://doi.org/10.1109/SoutheastCon42311.2019.9020468

[14] Ding Wei, Hua Zidong, Li Panhui, Gong Qiushi, and Cheng Yuxi. 2020. Botnet Host Detection Based on Heartbeat Association. In *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy (ICCSP 2020)*. Association for Computing Machinery, New York, NY, USA, 42âĂŞ46. https://doi.org/10.1145/3377644.3377653