Improving IoT security with data sketch algorithms

Gal Peretz

Abstract

The era of Internet of things(IoT) has come, according to state.com more than 26 billion IoT devices where connected to the internet as of 2019 and this number is expected to reach over 30 billion next year. The definition of IoT devices has evolved due to emerge of new technologies. typically an IoT device is a low resource hardware such as sensors, cameras, lighting fixtures and many more. While we can benefit from connecting all of our devices to the internet this also reveal severe IoT cybersecurity challenges. Since those devices have low memory and compute capabilities it is not possible to use the familiar firewall and anti-viruses (or nearly any familiar security agent) software on the device itself, thus more common paradigm to secure those devices is to aggregate billion of security logs and telemetry and then utilize the cloud distributed compute power in order to apply detection algorithms on the aggregated data. The main problem with this paradigm is that "time to detect" is very long because of the high volume and high throughput of information that comes from the devices. This paper will review two of the most common attack methods in the IoT space, port scanning and DNS random subdomain DDOS attack. In paricular, we will review in details the algorithms suggested in [1] and in [4] and show how we can utilize data sketch algorithms to improve security in the IoT space. This paper will be devided to 5 main sections. the first section will be an introduction to the IoT cybersecurity world. the second section will review in details the IoT device port scan attack and how we can use sliding HyperLogLog algorithm to mitigate these attacks. the third section will show how we can use distinct weight sampling havy hitter (dwsHH) algorithm to mitigate DNS random subdomain attack, an attack that caused huge damange in October 2016 by the Mirai malware and caused damage to big sites like AirBnB and Github. In the fourth section we will perform an experiments to evaluate the dwsHH novel algorithm and the end to end detection algorithm suggested in [4].In the last section we will conclude everything and provide our insights.

1 Introduction

IoT devices are simply an interconnected hardware that are linked by the internet, through which they can transmit and receive data. IoT devices are disrupting consumer and enterprise industries. In the consumer space there are couple of multibillion companies like IRobot, Nest, Xiaomi that send sensetive users information from devices to cloud and from the cloud back to the devices. In the enterprise space IoT devices like oil drills need to send telemetry and get commands from the cloud to operate correctly. although companies are starting to understand the security hazard in the IoT space and invest bilions in cybersecurity solutions most of those solutions try to aggregate telemetry from the devices and provide a security services that analyze this big data on the cloud. Thankfuly to cloud computing we can analyze large amount of security logs and detect potential attacks, however the volume of the data is huge thus it is hard to provide a "near real time" detection that way. This paper will focus on methods to utilize data sketching algorithms that usally been overlooked when building security solutions.

2 Port Scans and HyperLogLog

Most of the attacks directed to IoT devices often start by scanning the IoT network to search for open "doors", or open ports. ports scanning is a common way to identify a weak point in the device network, this technique consist of sending messages to devices where each message directed to a different port in order to get information about vournable applications that listen for connections on the open ports. Several port scan detection mechanisms have been propsed but most of them use the same hyristic that detect a port scan if specific source communicate with more than X ports in a given time window. The main problem with this hyristic is that the state that we need to save is too big therefore force us to look on a small time window. the scan program can easly delay the different requests and by doing this fool the detection mechanisms. Chabchoub et al [1] suggest to use sliding HyperLogLog to solve this problem. HyperLogLog(HLL) is very efficient probablistic algorithm for cardinality estimation. it can estimate the number of distinct elements in a stream of data with very small memory footprint. to understand how HLL works we need to review it's evolution progress from probablistic counting [5] to LogLog algorithm [2] and eventually HyperLogLog algorithm [6] as we know it today.

2.1 Probablistic counting

Assume that we have a vector of length n consist of 0,1 where each bit is independent to the other bits and all the bits identically distributed with equal probability to get 0 or 1. the probability that this vector will start with zero is $\frac{1}{2}$ and the probability to get 2 zeros is $\frac{1}{4}$ and in general the probability to get k leading zeros is $(\frac{1}{2})^k$. This means that if we saw that the maximum number of leading zeros is k for all vectors in a stream of random distributed vectors there is a good chance that the cardinality of the stream is 2^k. to get a randomly distributed vector to apply this intuition we can use hash function that will transofrm our input to randomly distributed vector as needed. The problem with this algorithm is that the variance is too big thus the estimation error can be big as well.

2.2 LogLog

To reduce the variance Flajolet et al [2] suggested to take the first x bits from the vectors and convert it to a bucket identifier. for example if x equal 3 then the vector 010 011000 belong to bucket 2 (because 010 = 2) and the max leading zeros for this vector is 1. the total number of buckets will be 2^x and each bucket will store the maximum leading zeros of all vectors that was mapped to it. then to estimate the cardinality we will average the buckets using harmonic mean. This method reduce the variance and as a result get a standart error of $\approx \frac{1.3}{\sqrt{m}}$ (m is number of buckets). the memory footprint for this algorithm is small because if the vector length is n then the max cardinality is log(n) and we can represent log(n) number by log(log(n)) bits thus the total memory complexity for all buckets is $O(m \cdot log(log(n)))$ hence the name loglog algorithm.

2.3 HyperLogLog

The HyperLogLog paper by Flajolet et al [6] shows mathmaticlly how we can improve the LogLog algorithm by taking only the max 70% of the buckets we can reduce the error to $\frac{1.04}{\sqrt{m}}$. we doing this by first sort the buckets values when we want to evaluate the cardinality then in the average process we take the top 70% of the buckets instead of all buckets.

```
Let m be the number of buckets
    **Input**: stream of elements
    **Initialization**: R[1]...R[m] $\leftarrow$ 0
    h $\leftarrow$ random distributed hash function
    **while** *stil e in stream* **do**
        x $\leftarrow$ hash(e)
        j $\leftarrow$ bucket label
        R[j] $\leftarrow$ max(R[j],$\rho(x)$) ($\rho$ gives the number of leading zeros)
    **end**
    Z $\leftarrow$ $(\sum_{j=1}^{m} 2^{-R[j]})^{-1}$ compute the harmonic mean of
    Return $a_m m^2 \cdot Z$ with $a_m$ equal $(m \int_0^{\inf} log((\frac{2+u}{1+u})^m du))^{-1}$
```

Algorithm 1: HyperLogLog algorithm

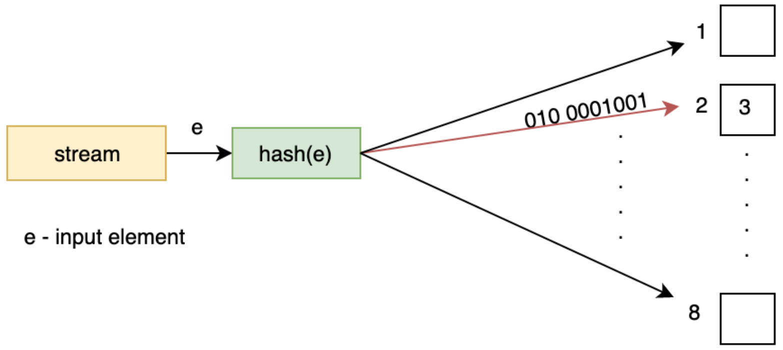


Figure 1: Ingestion of input element to the HyperLogLog algorithm

Figure 1 describe the ingestion of single element to the HLL algorithm. the hash function is randomly distributed hash function and hash the elment to sequence of bits where each bit has equal probability to be 0 or 1. after applying the hash function we take the m first bits (m equal 3 in this example) and hash it to the corresponding bucket and if the number of leading zeros in the following bit sequence is higher then the bucket value we replace the bucket value to the number of leading zeros.

2.4 Sliding HyperLogLog

The Sliding HyperLogLog (SHLL)is based on the HLL algorithm and provide a way to estimate the cardinality of elements that bounded by length of sliding time window. This algorithm saves only the necessary information to compute the cardinality of the elements in the time window. The stream element need to include timestamp that indicate the time context of the element, then after the hashing process and extracting the size of the leading zeros we eliminate any previous elements that are too old or that there number of leading zeros are less then current element.

Let LFPM ← []
let W ← window time
Let $< t_k, b_k, R_k >$ be the timestamp, bucket number and the max leading zeros correspondingly

```
Input: stream of elements
Initialization: while stil e in stream do
    $< t, b, R >$← element timestamp,bucket number, number of leading zeros
    for $< t_k, b_k, R_k >$ in LFPM delete $< t_k, b_k, R_k >$ if t - $t_k > W$
    for $< t_k, b_k, R_k >$ in LFPM delete $< t_k, b_k, R_k >$ if $R_k \leq R$
    add $< t, b, R >$ to LFPM
end
Continue same as HyperLogLog algorithm
```

Algorithm 2: Sliding HyperLogLog algorithm

When we want to compute the cardinality of the set we use the current LFPM list as an input to the HyperLogLog algorithm(we need to adjust the implmentation because the bucket number and number of leading zeros have already been extracted from the bits sequence).

2.5 Detect scan port with sliding HyperLogLog

One of the most common port scan method is vertical scan. The vertical port scan targets several destination ports on single machine. to evade the detection machinisem the scan has a delay mechanisms between each message, this way the detecor cannot keep up with the state of distinct sources that communicated with different ports in a determenistic way because this state is too big. In [1] the author suggest to embed the sliding HyperLogLog with threshold machinisem that check if distinct source send messages to more than x ports in specific time window. the small memory footprint (couple of KBs) makes the Sliding

HyperLogLog a good solution for embeded detection machinisem on the IoT device itself. Another solution for detection machinisem can be the Distinct Weighted Sampling Heavy Hitters algorithm that also use HLL and output the heavy hitters sources that communicated with the most distinct ports. this way when device is under attack the attacker will emerge as a heavy hitter in by the dwsHH algorithm. we will discuss this algorithm in the next section and provide experimental validation for it.

3 Mirai, Botnets and data sketch algorithms

Mirai is the code name of a well-known mallware that infect iot devices, turning them into a remotly controlled bots. This network of remotly controlled devices known as botnet can be used to launch distributed denial of service(DDOS) attack. Denial of service happens when too many requests are sent to server and the server can't fulfil more requests thus it starting to reject users‘ requests. In october 2016 Mirai‘s botnet has been used to launch DDoS attack that was directed to Dyn, a major Domain Name System(DNS) provider, this attack had major impact on well known companies including AirBnB, Netflix, PayPal, and GitHub.

The "Mitigating DNS random subdomain DDoS attacks by distinct heavy hitters sketches" [4] paper describe how to use noval data sketch algorithm, distinct weighted sampling heavy hitters(dwsHH), to detect DNS random subdomain DDoS attacks like the one that happened in the Mirai case. In the random subdomain attack many DNS resolve requests are sent to the DNS server for single or few victim domains. the big amount of randomly generated DNS requests coming from the botnets devices causing the DNS server to be overwhelmed and to start reject authentic resolve requests. The mitigation of this kind of attack is hard because the DNS requests come from a legitimate sources which eliminate the posability for source filtering. The detection algorithm suggested by Feibish et al[4] uses distinct counters estimation combine with variation of sample and hold [3] algorithm to find the distinct heavy hitters. The premise of the detection algorithm is that when the DNS server is under attack the number of distinct subdomain of the attacked domain will get a high value. The dwsHH algorithm takes as an input a stream of domain and subdomain (for example photos as a subdomain and google.com as a domain that represent the DNS resolve request for photos.google.com) and natural number 'k' that indicate the cache size (or how many counters to use), and by using adaptive thresholding, sample and hold algorithem and distinct counters estimation outputs the distinct heavy hitters which mean the domains that pair with the biggest number of distinct subdomains in the stream of DNS resolve requests. the algorithm uses distinct estimate counters and leave as an implemntation choice the specific probabilistic data structures to use, neverthless this data structure need to support three functions.

- Init - Initialization of new distinct counter
- Merge(subdomain) - add subdomain for the distinct count for the domain
- CardEst - return the set cardenality meaning an estimation for the number of distinct subdomains

In addition uniformly distributed Hash function will be use, $Hash \sim U[0,1]$, to hash the (domain, subdomain) pair.

```
Input: stream of elements(domain, subdomain), k
Initialization: dCounters ← $\emptyset$;$\tau \leftarrow 1$
Output: set of (domain, subdomain cardenality, certainty variable)
while (domain, subdomain) in stream do
    if domain in dCounters then
        dCounters[domain].Merge(subdomain)
        dCounters[domain].seed ← min{dCounters[domain].seed,
         Hash(domain, subdomain)}
    else
        if Hash(domain, subdomain) < $\tau$ then
            dCounters[domain].Init;
            dCounters[domain].Merge(subdomain)
            dCounters[domain].seed ← Hash(domain, subdomain)
             dCounters[domain].$\tau \leftarrow \tau$
            if $|dCounters| > k$ then
                domain ← $argmax_{y \in dCounters} dCounters[y].seed$
                $\tau \leftarrow dCounters[domain].seed$
                Delete dCounters[domain]
            end
        end
    end
end
Return (For domain in dCounters, (domain,
 dCounters[domain].CardEst, dCounters[domain].$\tau$))
```

Algorithm 3: Fixed-size streaming Distinct Weighted Sampling

the paper proves that a domain is likly to be sampled when the cardenality of it's subdomains set is among the top k. In addition the paper suggests a pregmatic architecture to detect DNS random subdomain attack by seprate the stream to peacetime and attack time. In the peacetime the system will white list domains that are among the heavy hitter according to the dwsHH algorithm because these domains probably are creating random subdomains on a regular basis, thus we cannot predict an attack on them. in addition the pipline use classic heavy hitter algorithm to white list subdomains that are common. In the attack time the system will count only domains and subdomains that are not white listed and use dwsHH to estimate the cardenality and then compute the different between the attack time and peacetime for the top heavy hitters. In this paper we will get an empirical validation by condacting an experiments and validate the dwsHH algorithm's performance Our first experiment will check the performance for the cardenality estimation of the subdomains sets for the heavy hitters domains, in addition we will estimate the memory that we need to get a good cardenality estimation. The second experiment will check the suggested end to end detecting algorithm for DNS random subdomains attacks.

4 Experiments

We will use a dataset containing close to 170 milion DNS requests. the probbalistic data structure for the distinct counters estimation will be HyperLogLog. The first experiment will evaluate the performance of the cardenality estimation for different cache sizes, 200, 500, 1000, 5000, 10000. In the evaluation phase we will compare the real number of distinct subdomains for the top 100 domains to the estimated number.
Our second experiment will simulate an attack on github.com by injecting random subdomains DNS requests in the stream of legitimate DNS requests on the attack phase, then we will compute the ratio between the number of distinct subdomains in attack phase to the baseline computed in the peacetime phase and compare it to other domain's ratio that was not under attack.

4.1 Exploring the data

Our dataset contains DNS requests for records with type CNAME and A. a CNAME DNS record is an alias to other DNS address and it serve as an intermidiate address to the resolve of the server computer IP and normaly maped to A or AAAA record. The A record is an alias to IPv4 that mark the server concrete IP address. First we will use spark to analyze our dataset in deterministic way.

```
+---------------+------------------+
|         domain|distinct_subdomain|
+---------------+------------------+
|   blogspot.com|           9534421|
|  wordpress.com|           5562462|
|      home.blog|           2526296|
|    skyrock.com|           2060336|
| deviantart.com|            946580|
|  herokuapp.com|            849595|
|        fc2.com|            823218|
|     weebly.com|            680945|
|livejournal.com|            637255|
|     lofter.com|            415571|
|  myshopify.com|            376913|
| cloudflare.net|            345711|
|filetransit.com|            328810|
|    wixsite.com|            325492|
|  amazonaws.com|            311323|
|    soft112.com|            299266|
|      jimdo.com|            283252|
|list-manage.com|            245873|
|        wix.com|            216419|
|      food.blog|            185435|
+---------------+------------------+
```

top 20 domains with distinct subdomains

We can see that the top 20 domain with the highest number of distinct subdomain requests are domains that host other people sites like blogspot.com, wordpress.com and wix.com this probably because they create big amount of subdomain for their customers so it will hard to tell the different between DDOS attack to authentic increase in traffic to those sites. using spark we can compute the real cardenality for the distinct subdomains for the top 100 domains

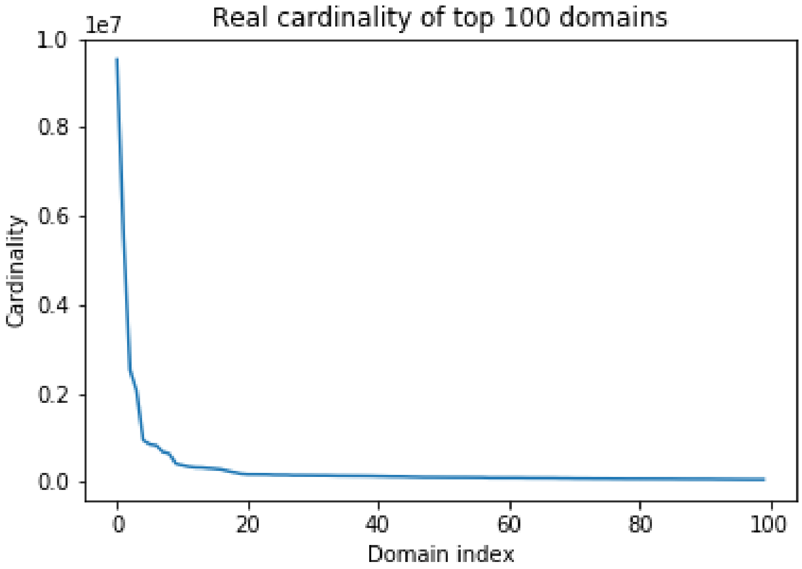
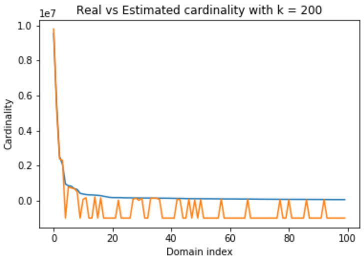
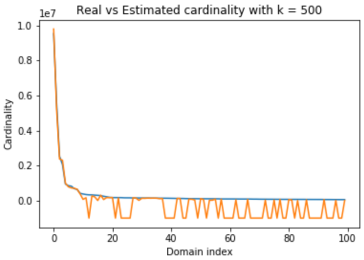


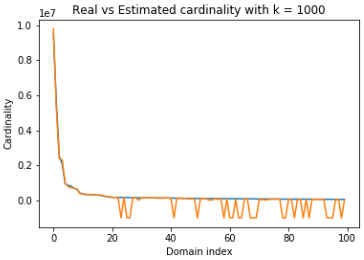
Figure 2: Top 100 real cardenality for the subdomains sets

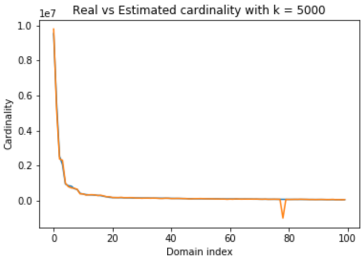
4.2 Results

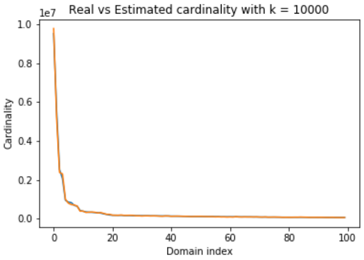
By using dwsHH algorithem we will compute the cardenality estimation of the top 100 domains for the different values of k (the cache size). if the algorithem will not output specific domain that exists in the real top heavy hitter list put -1000 as it's estimation to visualize the missing domains.


Real vs Estimated cardinality with k = 200
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Cardinality
0
20
40
60
80
100
Domain index


Real vs Estimated cardinality with k = 500
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Cardinality
0
20
40
60
80
100
Domain index


Real vs Estimated cardinality with k = 1000
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Cardinality
0
20
40
60
80
100
Domain index


Real vs Estimated cardinality with k = 5000
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Cardinality
0
20
40
60
80
100
Domain index


Real vs Estimated cardinality with k = 10000
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Cardinality
0
20
40
60
80
100
Domain index

we can see that for k equal to 10000 we get a pretty good estimation for all of the 100 top heavy hitter. to be more concrete using the dwsHH algorithem we get an average error of 0.5% for all of the top 100 domains and memory usage of 11 Mb compare to 4.2GB that spark uses to compute the deterministic cardenality of the subdomains set. In addition we can tune the parameter for each counter by increase the number of buckets for each HyperLogLog and to find the sweet spot between number of buckets to cache size, thus to reduce the memory usage even further. For the next experiment we will implement the suggested end 2 end detection mechanism by spliting the input stream to peacetime stream and attack time stream. In the peacetime phase we will learn the whitelist of domains and subdomains using dwsHH algorithem and classic heavy hitter algorithem. In the attacktime phase we will filter out any whitelist domains and subdomains and use dwsHH to detect suspected dns domains. we will inject randomly subdomains DNS requests for the github.com domain to see if we can identify the attack using this architecture.

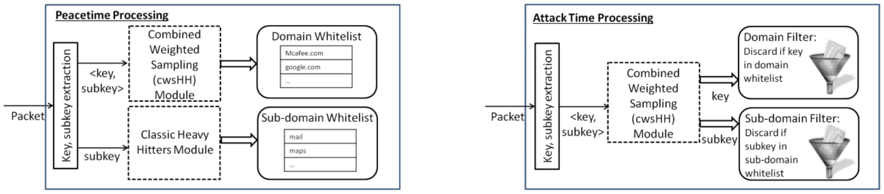

Peacetime Processing
Packet
Key, subkey extraction
<key, subkey>
subkey
Combined Weighted Sampling (cwsHH) Module
Classic Heavy Hitters Module
Domain Whitelist
Mcafee.com
google.com
...
Sub-domain Whitelist
mail
maps
...
Attack Time Processing
Packet
Key, subkey extraction
<key, subkey>
Combined Weighted Sampling (cwsHH) Module
key
subkey
Domain Filter: Discard if key in domain whitelist
Sub-domain Filter: Discard if subkey in sub-domain whitelist

Figure 4: Detection architecture

After analyzing[1] the detector estimation we can see two things, the first one is that all the heavy hitters domains that was whitelisted in the peacetime are not in the heavy hitters list for the attackedtime dwsHH algorithem output. The second thing is that the ratio between github.com cardenality estimation in attack time vs peacetime is very high (~ 700) compare to other domains, thus it easy to detect the attack.

5 Conclusion

We can see that data sketch algorithems can be very usefull to mitigate common cybersecurity issues in the IoT space. the big advantage of those algorithems is the small memory footprint that makes them ideal for IoT devices that have small memory capabilities and in the same time exchange a lot of infomation with the cloud. In addition we can use data sketch algorithems to get a "near real time" detection for destructive attacks like the DNS random subdomains DDOS attack.

[1] All the experiments results avilable on this repository : https://github.com/galprz/dns-random-subdomains-ddos-attack

References

[1] Yousra Chabchoub, Raja Chiky, and Betul Dogan. “How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?” In: *EURASIP J. on Info. Security* 2014.1 (Dec. 2014), pp. 1–11. ISSN: 1687-417X. DOI: 10.1186/1687-417X-2014-5.

[2] Marianne Durand and Philippe Flajolet. “Loglog Counting of Large Cardinalities”. In: *SpringerLink* (Sept. 2003), pp. 605–617. DOI: 10.1007/978-3-540-39658-1_55.

[3] Cristian Estan and George Varghese. “New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice”. In: *ACM Trans. Comput. Syst.* 21.3 (Aug. 2003), pp. 270–313. ISSN: 0734-2071. DOI: 10.1145/859716.859719.

[4] Shir Landau Feibish et al. “Mitigating DNS random subdomain DDoS attacks by distinct heavy hitters sketches”. In: (Oct. 2017). DOI: 10.1145/3132465.3132474.

[5] Philippe Flajolet and G. Nigel Martin. “Probabilistic counting algorithms for data base applications”. In: *J. Comput. Syst. Sci.* 31.2 (Sept. 1985), pp. 182–209. ISSN: 0022-0000. DOI: 10.1016/0022-0000(85)90041-8.

[6] *Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm.* [Online; accessed 8. Oct. 2019]. 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.9475.