# Mitigating DNS Random Subdomain DDoS Attacks by Distinct Heavy Hitters Sketches

Shir Landau Feibish[1], Yehuda Afek[1], Anat Bremler-Barr[2], Edith Cohen[1], and Michal Shagam[1]

[1]Blavatnik School of Computer Sciences, Tel-Aviv University, Israel

[2]Computer Science Dept., Interdisciplinary Center, Herzliya, Israel

*lfshir@gmail.com, afek@cs.tau.ac.il, bremler@idc.ac.il, edith@cohenwang.com, michalshagam@mail.tau.ac.il*

## ABSTRACT

Random Subdomain DDoS attacks on the Domain Name System (DNS) infrastructure are becoming a popular vector in recent attacks (e.g., recent Mirai attack on Dyn). In these attacks, many queries are sent for a single or a few victim domains, yet they include highly varying non-existent subdomains generated randomly.

Motivated by these attacks we designed and implemented novel and efficient algorithms for distinct heavy hitters (dHH). A (classic) heavy hitter (HH) in a stream of elements is a key (e.g., the domain of a query) which appears in many elements (e.g., requests). When stream elements consist of <key, subkey> pairs, (<domain, subdomain>) a distinct heavy hitter (dhh) is a key that is paired with a large number of different subkeys. Our algorithms dominate previous designs in both the asymptotic (theoretical) sense and practicality. Specifically the new fixed-size algorithms are simple to code and with asymptotically optimal space accuracy tradeoffs.

Based on these algorithms, we build and implement a system for detection and mitigation of Random Subdomain DDoS attacks. We perform experimental evaluation, demonstrating the effectiveness of our algorithms.

## 1. INTRODUCTION

The Domain Name System (DNS) service is a critical element in the internet functionality. Distributed Denial of Service (DDoS) attacks on the DNS service typically consist of many queries coming from a large botnet and sent to the root name server or an authoritative name server along the domain chain. According to Akamai's state of the internet report [2] nearly $20\%$ of DDoS attacks in $Q1$ of 2016 involved the DNS service, some of them on the *root* name servers [19].

This research was supported by a grant from the Blavatnik Cyber Security Councile, by a grant of the European Research Council (ERC) Starting Grant no. 259085, and by the Ministry of Science and Technology, Israel.

### 1.1 Random Sundomain Attacks

One type of particularly hard to mitigate DDoS attack is the randomized attack on the DNS service called Random Subdomain Attack [16] (also known as Authoritative Exhaustion Attack [4]; Nonsense Name Attack [14]; Pseudo-random Subdomain Attack [3]).

In this attack, queries for many different pseudorandom non-existent subdomains (subkeys) of the same primary domain (key) are issued [3]. Since the response to a query for a new subdomain is not cached at the DNS resolver, these queries are propagated to the domain authoritative server. Initially, the authoritative server is able to respond and typically answers with an "NXDOMAIN" response, indicating that the domain can't be found. Once the authoritative server becomes overwhelmed, it will either crash or implement a response rate limiting mechanism. Either way, no response will be received from the authoritative server and it will appear unresponsive to the ISP.

Shortly after, the ISP resolvers, that store each recursive request until a response is received, exhausts all available storage space and also becomes debilitated, causing legitimate clients to experience an increase in "Server Failure" responses from the ISP resolvers [3] While these attacks were first witnessed in 2009, in October 2016 they made headlines, when hundreds of top websites were drastically affected by the Mirai IoT Botnet attack on domains delegated to the Dyn DNS resolvers [4]. Mitigation of these attacks took hours.

Mitigation of Random Subdomain attacks is difficult since the packets in the attack are correctly formed DNS requests. Furthermore, the queries are normally received from legitimate ISP clients and therefore source based filtering can not be used. The solution of internet providers so far has been to identify the targeted zone by analyzing query logs for anomalies, and to temporarily prevent the name server from handling queries for this zone [3, 14], or alternatively to reduce the number of queries handled using rate limiting.

### 1.2 Our Contributions

The two major contributions of this paper are novel practi-

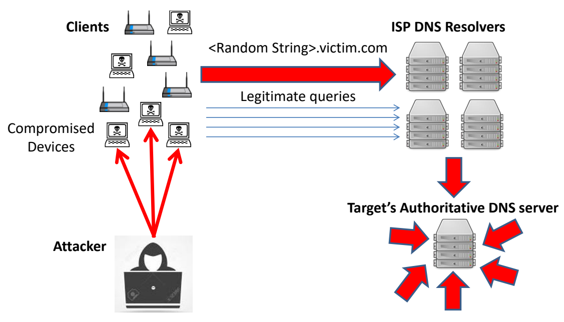


Figure 1: DNS Random Subdomain attack overview

cal sampling-based structures for *distinct heavy hitter (dHH)* detection, and the system we propose for detection and mitigation of Random Subdomain attacks.

### 1.2.1 Attack Mitigation System

In this paper we present a system for the mitigation of this attack. Our system is based on the observation that the number of distinct subdomains in queries for targeted domains significantly increases during this attack due to the pseudo-random part of the query. Our system detects this sudden rise in the number of distinct subdomains, and therefore identifies the targeted domain automatically. Depending on the rate of the attack, our system can detect an attack within seconds of attack start time.

During normal network operation, the number of distinct subdomains for each domain is usually relatively constant and typically a small number. One exception to this is the increasing usage of disposable domains. These are large volumes of automatically generated domains, legitimately created by top sites and services (e.g., social networks and search engines), to give some signal to their server [6]. By analyzing traffic during normal server load (i.e. "peacetime"), our system creates a baseline of the normal number of distinct subdomains of such domains so that it can detect the abnormal rise during the attack. Using this baseline, our system can identify attacks while significantly reducing false positives. Furthermore, we are able to automatically identify most of the legitimate requests for the targeted domain.

### 1.2.2 Algorithms for Distinct Heavy Hitters

Our system is based on our algorithms for finding distinct heavy hitters. Consider a stream of DNS queries, with the second-level domain, for example, serving as the key. A key that appears many times in the query stream constitutes a *"classic" heavy hitter* (e.g., google.com, cnn.com, etc,). If each query's subdomain serves as the subkey (e.g., mail., home., game1., etc,), a key with many different subkeys is then a *distinct heavy hitter (dHH)* or *superspreader* [18].

Generally, approximate distinct heavy hitters algorithms exhibit a tradeoff between detection accuracy and the amount of space they require. Cardinality estimate accuracy is even more difficult to achieve with a fixed-size structure since a key may be evicted from the cache and then re-enter the cache, resulting in cardinality inaccuracies. Our proposed fixed-size dHH algorithm, named *Distinct Weighted Sampling (dwsHH)*, uses a fix-size structure and outperforms known solutions both in terms of cardinality accuracy and practicality.

We demonstrate the effectiveness of our algorithms via experimental evaluations on real Internet traces and attacks.

## 2. DISTINCT HEAVY HITTER ALGORITHMS

## 2.1 Preliminaries and Background

### 2.1.1 Problem Definitions

Formally, our input is modeled as a stream of elements, where each element has a primary key $x$ from a domain $\mathcal{X}$ and a subkey $y$ from domain $D_x$. For each key, the *(classic) weight* $h_x$ is the number of elements with key $x$, and the *distinct weight* $w_x$ is the number of different subkeys in elements with key $x$.

A key $x$ with weight that is at least an $\epsilon$ fraction of the (respective) total is referred to as a heavy hitter. When $h_x \geq \epsilon \sum_y h_y$, $x$ is a (classic) *heavy hitter (HH)*, and when $w_x \geq \epsilon \sum_y w_y$, $x$ is a *distinct heavy hitter (dHH).*

### 2.1.2 Background

*Sample and Hold:* The Sample and Hold (S&H) algorithm [13, 10] is applied to a stream of elements, where each element has a key $x$ and weight $h_x$ as described above.

The *fixed threshold* design is specified for a threshold $\tau$. The algorithm maintains a cache $S$ of keys, which is initially empty, and a counter $c_x$ for each cached key $x$. A new element with key $x$ is processed as follows: If $x \in S$ is in the cache, the counter $c_x$ is incremented. Otherwise, a counter $c_x \leftarrow 1$ is initialized with probability $\tau$. The *fixed-size* design is specified for a fixed sample (cache) size $k$ and works by effectively lowering the threshold $\tau$ to the value that would have resulted in $k$ cached keys.

An important property of S&H is that the set of sampled keys is a probability proportional to size without replacement (ppswor) sample of keys according to weights $h_x$ [17].

*Approximate Distinct Counters:* A distinct counter is an algorithm that maintains the number of different elements in a stream of elements. An exact distinct counter requires state that is proportional to the number of different elements in the stream. Fortunately, there are many existing designs and implementations of *approximate* distinct counters that have a small relative error but use state size that is only logarithmic or double logarithmic in the number of distinct elements [12, 7, 5, 11, 8]. The basic idea is elegant and simple: We apply a random hash function to each element, and retain the smallest hash value. This value, in expectation, is becoming smaller as there are more distinct elements, and thus can be used to estimate the number of distinct elements. The different proposed structures have different ways of enhancing this approach to control the error. The tradeoff between structure size and error are controlled by a parameter $\ell$: A structure of size proportional to $\ell$ has normalized root mean square error (NRMSE) of $1/\sqrt{\ell}$. In Section 2.2.1 we use distinct counters as a black box in our dHH structures, abstracted as a class of objects that support the following operations: `Init`: Initializes a sketch of an empty set; `Merge(`$x$`)`: merge the item $x$ into the set ($x$ could already be a member of the set or a new item); `CardEst`: return an estimate on the cardinality of the set (with a confidence interval)

## 2.2 Distinct Weighted Sampling

### *2.2.1 Distinct Heavy Hitters Algorithms Overview*

Our distinct weighted sampling schemes take as input a stream of elements that are key and subkey pairs. We build on the fixed-size classic S&H schemes but make some critical adjustments: First, we apply hashing so that we can sample the distinct stream instead of the classic stream. Second, instead of using simple counters for cached keys as in classic S&H, we use approximate distinct counters applied to subkeys. Third, we maintain state per key that is suitable for estimating the weight of heavy cached keys (whereas classic S&H was designed for unbiased domain queries).

### *2.2.2 Fixed-size distinct weighted sampling*

The fixed-size Distinct Weighted Sampling (dwsHH) algorithm is specified for a cache size $k$. Compared with the fixed-threshold algorithm, we keep some additional state for each cached key:

- The threshold $\tau_x$ when $x$ entered the cache (represented in the pseudocode as dCounters[$x$].$\tau$). $\tau_x$ is important in deriving confidence intervals on $w_x$. Intuitively, $\tau_x$ captures a prefix of elements with key $x$ which were seen before the distinct structure for $x$ was initialized, and is used to estimate the number of distinct subkeys in this prefix.
- A value `seed(`$x$`)` $\equiv \min_{(x,y)\text{in stream}}$ `Hash(`$x,y$`)` which is the minimum `Hash(`$x,y$`)` of all elements with key $x$. (in the pseudocode, dCounters[$x$].`seed` represents `seed(`$x$`)`). Note that it suffices to track `seed(`$x$`)` only after the key $x$ is inserted into the cache, since all elements that occurred before the key entered the cache necessarily had `Hash(`$x,y$`)` $> \tau_x$, as the entry threshold $\tau$ can only decrease over time.

The fixed-size dwsHH algorithm retains in the cache only the $k$ keys with lowest seeds. The effective threshold value $\tau$ that we work with, is the seed of the most recently evicted key. The effective threshold has the same role as the fixed threshold since it determines the (conditional) probability on inclusion in the sample for a key with certain $w_x$. A pseudo code is provided as Algorithm 1.

### *2.2.3 Analysis and estimates*

**Algorithm 1:** Fixed-size streaming Distinct Weighted Sampling (dwsHH)

```
Data: cache size $k$, stream of elements of the form (key,subkey),
    where keys are from domain $\mathcal{X}$
Output: set of $(x, c_x, \tau_x)$ where $x \in \mathcal{X}$
dCounters $\leftarrow \emptyset$; $\tau \leftarrow 1$ // Initialize a cache of
  distinct counters
foreach stream element with key $x$ and subkey $y$ do // Process a
  stream element
  if $x$ is in dCounters then
    dCounters[$x$].Merge($x,y$)
    dCounters[$x$].seed $\leftarrow$
      $\min\{$dCounters[$x$].seed, Hash($x,y$)$\}$
  else
    if Hash($x,y$) $< \tau$ then // Create dCounters[$x$]
      dCounters[$x$].Init
      dCounters[$x$].Merge($x,y$)
      dCounters[$x$].seed $\leftarrow$ Hash($x,y$)
      dCounters[$x$].$\tau \leftarrow \tau$
      if |dCounters| $> k$ then
        $x \leftarrow \arg\max_{y \in \text{dCounters}}$ dCounters[$y$].seed
        $\tau \leftarrow$ dCounters[$x$].seed
        Delete dCounters[$x$]

return (For $x$ in dCounters,
($x$, dCounters[$x$].CardEst, dCounters[$x$].$\tau$))
```

We first consider the sample distribution of dwsHH. As we mentioned, it is known that classic S&H applied with weights $h_x$ has the property that the set of sampled keys is a ppswor sample according to $h_x$ [9]. Surprisingly, the sample distribution properties of S&H carries over from being with respect to $h_x$ (classic S&H) to being with respect to $w_x$ (dwsHH). Therefore, when working with a fixed $k$, a key with weight $w_x$ is selected with probability $\geq 1-(1-w_x/m)^k$, where $m = \sum_x w_x$ is the sum of weights of all keys. If the threshold is $\tau$, a key with weight $w_x$ is selected with probability $1-\exp(-\tau w_x)$. A detailed proof is found in the technical report [1]. We therefore obtain that key $x$ is very likely to be sampled when $w_x \gg \max_{i\in[0,k-1]}(m - \sum_{x\in top_i} w_x)/(k-i)$ where $top_i$ is the set of $i$ heaviest keys. A detailed explanation of this bound with relevant proofs can be found in [1].

In terms of time complexity, given a stream of ¡key, subkey¿ pairs, for each pair the dwsHH algorithm makes at most $O(log(n))$ accesses to memory for keys not in the cache and 2 accesses for keys in the cache. The first is an item 'read' for checking if the key is in the structure. If the key is in the structure, only an additional 'write' is needed. Otherwise, the algorithm searches for the item with the maximal seed. The number of accesses for this depends on the implementation. We assume an efficient structure is used for max search which, assuming there are $n$ items in the structure, makes $O(log(n))$ reads for the search, followed by an additional write operation.

*Estimate quality and confidence interval.*

With the fixed-size (dwsHH) scheme, we expect the cache to include keys with $w_x \gg \sum_y w_y/k$ but it may also include some keys with small weight.

For many applications, an estimate on the weight $w_x$ of the heavy hitters is needed. We compute an estimate with a confidence interval on $w_x$ for each cached key $x$, using the entry threshold dCounters[$x$].$\tau$ and the approximate distinct count dCounters[$x$].`CardEst`.

We obtain the confidence interval [dCounters[$x$].`CardEst`$- a_\delta\sigma_2$, dCounters[$x$].`CardEst` $- 1 + 1/\tau + a_\delta\sqrt{\sigma_1^2 + \sigma_2^2}$]

where $a_\delta$ is the coefficient for confidence $1-\delta$ according to the normal approximation. E.g., for 95% confidence we can use $a_\delta = 2$. We note the confidence intervals are tighter for keys that are presented earlier and thus have $\tau_x \ll \tau$.

### 2.2.4 Integrated dwsHH design

We propose a seamless design (Integrated dwsHH) integrating the hashing performed for the weighted sampling component with the hashing performed for the approximate distinct counters. We use a type of distinct counters based on *stochastic averaging* ($\ell$-partition) [12, 11] (see [8] for an overview). This design hashes strings to $\ell$ buckets and maintains the minimum hash in each bucket. We estimate the distinct counts using the tighter HIP estimators [8].

For a sampled $x$, we can obtain a confidence interval on $w_x$ using the lower end point dCounters[$x$].`CardEst` $+ 1$, with error controlled by the distinct counter and the upper end point dCounters[$x$].`CardEst`$+1/$dCounters[$x$].$\tau$, with error controlled by both the distinct counter and the entry threshold. The errors are combined as explained in Section 2.2.3 using the HIP error of
$\sigma_2 \approx (2\ell)^{-0.5}$dCounters[$x$].`CardEst` .

The size of our structure is $O(k\ell\log m)$ plus the representation of the $k$ cached keys. Note that the parameter $\ell$ can be a constant for DDoS applications: A choice of $\ell = 50$ gives NRMSE of 10%. Additional details can be found in [1].

## 2.3 Evaluation

### 2.3.1 Theoretical Comparison

In Table 1 we show a theoretical memory usage comparison of our distinct weighted sampling algorithms, Superspreaders and Locher [15], assuming all algorithms use the same distinct count primitive. We are using the notations: $\delta$ as the probability that a given source becomes a false negative or a false positive, $N$ as the number of distinct pairs, $r$ as the number of estimates, $s$ as the number of pairs of distinct counting primitives used to compute each estimate, and c (for a $c$-superspreader (i.e. we want to find keys with more than $c$ distinct elements) choosing $c = \tau^{-1}$.Note that the Superspreaders algorithm does not provide an estimate on the distinct weight of the keys, but rather only reports which keys have high enough weights. Locher's algorithm provides an estimate error which is incomparable theoretically and significantly higher than ours in practice

| Algorithm | Memory usage | Keys' distinct weight estimation error |
|---|---|---|
| Fixed-threshold distinct WS | $O(\tau\sum_y w_y \cdot \ell\log m)$ (Exp.) | $\tau^{-1} + w_y/\sqrt{2\ell}$ |
| Fixed-size dwsHH | $O(k\ell\log m)$ | $(1/k)\sum_y w_y + w_y/\sqrt{2\ell}$ |
| Superspreaders 1-Level Filtering [18] | $O(\frac{N}{c})$ | $NA$ |
| Superspreaders 2-Level Filtering [18] | $O(\frac{N}{c} ln\frac{1}{\delta})$ | $NA$ |
| Locher [15] | $O(rs \cdot 2\ell + \lvert k\rvert)$ | $NA$ |

Table 1: Theoretic Comparison between methods

### 2.3.2 Practical Evaluation

*Accuracy and Parameters.*

The following tests were done using a $4GB$ trace of $40M$ DNS queries captured at our campus network. For each DNS query $q = ...p_6.p_5.p_4.p_3.p_2.p_1$, we sliced the query at most 5 times to produce the $< key, subkey >$ pairs, $< p_1, ...p_6.....p_2 >, < p_2.p_1, ...p_6.....p_3 > ... < p_5.....p_1, ....p_6 >$. This process gave us a total of over $120M$ pairs composed of a total of nearly $1M$ distinct pairs.

We compare the affect of different cache sizes ($k$) on the output of our dwsHH algorithm. As shown in Fig 2a we set the number of buckets to be 32 and use cache sizes of 100, 500, 1000, 10000. Using a cache size of 100, our algorithm reports keys with cardinality at least 0.005 of the total number of distinct items with a false negative rate under 5%. A false negative rate of under 5% is also achieved with cache size of 500 for cardinality over 0.0008. Using a cache of 1000 and 10000, our algorithm reports keys with cardinality at least 0.0004 of the total, with false negative rates of 2% and 0% respectively.

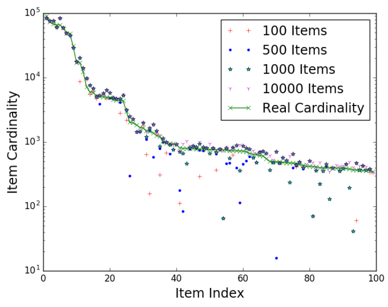
Additionally, we compare the affect of the different number of buckets ($\ell$) on the output of our dwsHH algorithm. As shown in Fig 2b we set cache size to be 1000 and use $4, 8, 16, 32$ and 64 buckets. For the reported keys, using 4 buckets gave a median distinct weight estimated error of 49% over all reported keys and 8, 16, 32 and 64 buckets gave a median error of 33%, 18%, 13% and 9% accordingly.

To report, for example, all keys which have a weight of at least 0.001% of the total number of distinct pairs, using the dwsHH algorithm, we use cache size of 1000, achieving a false negative rate of 0% and a false positive rate of 0%. Using 32 buckets, the weight estimates provided have a median error of less than 10% of the item cardinality for the reported keys. This test is shown in Figure 2c.

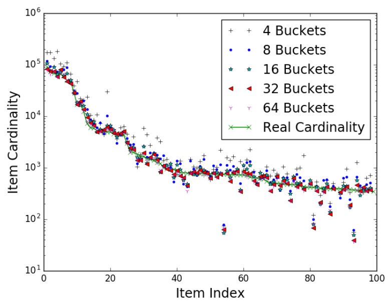
## 3. RANDOMIZED SUB-DOMAIN ATTACKS MITIGATION SYSTEM

## 3.1 Notations

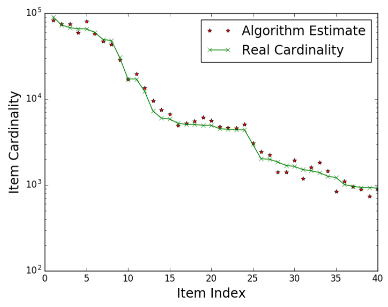
We denote a domain, subdomain and subpart in the following manner: Given a query $q = ...d_6.d_5.d_4.d_3.d_2.d_1$,



(a) Modified cache size



(b) Modified Number of Buckets



(c) 32 Buckets, 1000 Items

Figure 2: Distinct Weighted Sampling (dWS) parameter comparison results

a subpart of a domain is any individual part $d_i$. (i.e., $d_1$, $d_2$ etc.). A domain-suffix of the query is any suffix of $q$ composed of whole subparts. The subdomain-prefix of a domain-suffix is the prefix of $q$ up to and not including the domain-suffix. For example, for domain-suffix $d_1$ the subdomain-prefix is $...d_6....d_2$. For brevity, we refer to a domain-suffix as a domain and a subdomain-prefix as a subdomain. Note that we refer to the length of a domain to be the number of domain subparts and not the number of characters.

## 3.2 Overview

*Attack Detection:* As depicted in Figure 3, attack detection is done in two stages. The first stage is a preprocessing of traffic captured when there is a normal DNS query load (this is considered to be peacetime). Using our system, a baseline is created which identifies domains which have many different subdomains on a regular basis (for example, domains of sites that use disposable domains). Additionally, a whitelist of common domain subparts (i.e., mail, maps etc.) is also identified and used during mitigation to allow the legitimate queries of targeted domains. The second stage is an analysis of traffic during an attack. The system identifies domains which are potential attack targets. If the number of distinct queries for these domains is significantly higher than the peacetime baseline, these domains are set as attack signatures.

The main component of our system is the Distinct Heavy Domain Hierarchy Extractor (HDDH) (Section 3.3), which is used for both the baseline creation as well as the attack signature extraction.

*Attack Mitigation:* Once signatures have been extracted, consequent queries are matched against the attack signatures. Queries which match an attack signature and do not qualify as whitelisted are dropped before reaching the ISP resolvers. For example, in an attack on 'victim.com', our system would generate the signature '*.victim.com'. Using the whitelist of common domain subparts, our system identifies that 'mail.victim.com' is not an attack query and it is allowed. Other queries for 'victim.com' are dropped. The whitelist can be fine-tuned for each signature during the attack to further reduce false positives.

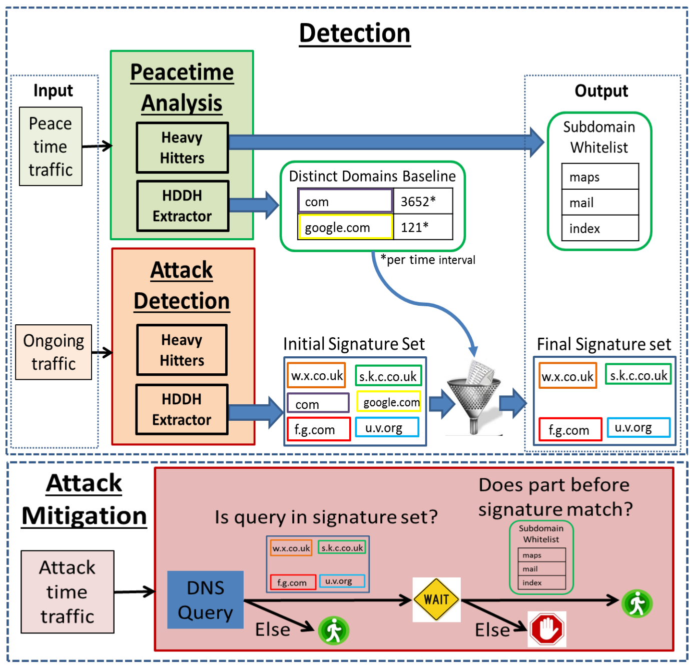


Figure 3: DNS Random Subdomain mitigation system

Our system makes no assumptions on the resource consumption or behaviour of the resolvers making it more robust in terms of detection.

## 3.3 Heavy Distinct Domain Hierarchy (HDDH) Extractor

The HDDH can be better visualized using a trie. As can be seen in Figure 4, each edge of the trie is labled with a domain subpart. Each node represents a domain (e.g. the domain $*.site.org$ is represented by the right-most leaf in the trie). Each node is labeled with the number of distinct subdomains seen for that domain. For example, there were 500 different queries for domain $*.com$, of which 420 were for domain $*.google.com$, 60 for $*.cnn.com$ and the remaining 20 were for domains that had a cardinality below $min\_heavy$. Note that the remaining cardinality of each node is the number indicated on the node minus the sum of its child nodes in the next level of the tree. We would like to find a minimal set of nodes in the trie with a cardinality above $min\_heavy$ that cover the leaves of the trie.

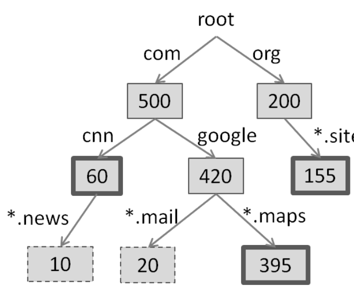


Figure 4: Hierarchy of heavy distinct domains. Bold edged nodes are in the cover, dashed edge nodes do not surpass the minimum cardinality.

In order to extract the hierarchy of heavy distinct domains, we need to efficiently compute how many of the distinct subdomains are contributed by each branch of the hierarchy. For each heavily distinct domain we would like to identify which, if any, of its subdomains is also heavily distinct. Furthermore, we would like to calculate the accumulative cardinality of all of its heavily distinct subdomains. Since extracting the entire hierarchy of queried domains would consume way too many resources, we provide an approximate solution, that allows extracting the desired information mainly for the heavily distinct domains.

The HDDH Extractor is composed of our Fixed-size streaming Distinct Weighted Sampling (and specifically Integrated dwsHH) structures for Distinct Heavy Hitters detection.

Our structure maintains 5 Integrated dwsHHstructures which we denote $DHH_1$-$DHH_5$. Denote as $k_i$ the size of each $DHH_i$. The keys in each $DHH_i$ are domains of length $i$ (i.e. domains of the form $*.d_i.d_{i-1}.....d_1$).

Given a stream of traffic (or a traffic capture), for each query $q = ...d_6.d_5.d_4.d_3.d_2.d_1$ received, the key $*.d_1$ (of length 1) is inserted to $DHH_1$ with subkey $...d_6....d_2$, key $*.d_2.d_1$ (of length 2) is inserted to $DHH_2$ with subkey $...d_6....d_3$ and so on. However, a insertion is made to $DHH_2$ only if $*.d_1$ was already found in $DHH_1$. Similarly, a insertion is made to $DHH_3$ only if $*.d_2.d_1$ was already found in $DHH_2$ and so on. Meaning, that a longer domain is only inserted into the structure if a shorter domain of that URL was already sufficiently heavy to be an item in the structure. In this manner, only domains which are somewhat likely to become signatures are inserted into the structure.

To find the distinct heavy domain cover, once the traffic capture has been analyzed, or after every fixed time interval, a heavy domain cover must be extracted from the structure. To identify the heavy domain cover, we build a trie as shown in Figure 4, using only the items in our HDDH Extractor. Intuitively, each domain found in our HDDH extractor can be placed on a branch of the trie and the value of each node needs to be calculated. This is done by traversing each key $d$ in each $DHH_i$ (for $1 \leq i \leq 4$). For each key, we calculate the sum of the cardinalities of its children in the next level of the trie, i.e. in $DHH_{i+1}$, defined as: $SumChildren_d = \sum\{$CardEst of all items in $DHH_{i+1}$ s.t. $d$ is their suffix$\}$. Once this sum has been calculated for all keys in $DHH_1$ through $DHH_4$, based on predefined parameters, our algorithm identifies, for every branch, the deepest node that has enough distinct subdomains and its child nodes do not. These nodes will compose the heavy domain cover from which the signature set will be selected.

## 4. REFERENCES

[1] Tech report: https://www.dropbox.com/s/r9kgatk4htfyu1w/techreport.pdf?dl=0.
[2] akamais [state of the internet] / security – q1 2016 report. www.akamai.com/StateOfTheInternet, 2016.
[3] Cathy Almond. Recent authoritative exhaustion attacks, 2016. https://www.arbornetworks.com/threats/.
[4] Chris Baker. Recent authoritative exhaustion attacks, October 2016. Talk given on behalf of Dyn Inc.
[5] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*. ACM, 2002.
[6] Yizheng Chen, Manos Antonakakis, Roberto Perdisci, Yacin Nadji, David Dagon, and Wenke Lee. DNS noise: Measuring the pervasiveness of disposable domains in modern DNS traffic. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 598–609. IEEE, 2014.
[7] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. System Sci.*, 55:441–453, 1997.
[8] E. Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. *TKDE*, 2015.
[9] E. Cohen, N. Duffield, H. Kaplan, C. Lund, and M. Thorup. Algorithms and estimators for accurate summarization of unaggregated sata streams. *J. Comput. System Sci.*, 80, 2014.
[10] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM'02 Conference*. ACM, 2002.
[11] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms (AOFA)*, 2007.
[12] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. System Sci.*, 31:182–209, 1985.
[13] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*. ACM, 1998.
[14] C. Liu. A new kind of ddos threat: The nonsense name attack. *Network World*, 2015. [Online; posted 27-January-2015].
[15] T. Locher. Finding heavy distinct hitters in data streams. In *SPAA*. ACM, 2011.
[16] Latest internet plague: Random subdomainattacks. https://nominum.com/wp-content/uploads/2014/10/Nominum-Whitepaper-Latest-Internet-Plague-Random-Subdomain-Attacks.pdf, 2014.
[17] B. Rosén. Asymptotic theory for successive sampling with varying probabilities without replacement, I. *The Annals of Mathematical Statistics*, 43(2):373–397, 1972.
[18] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2005.
[19] Verisign distributed denial of service trends report q4 2015. https://www.verisign.com/assets/report-ddos-trends-Q42015.pdf, 2015.