Automated Signature Extraction for High Volume Attacks*

Yehuda Afek
Blavatnik School of Computer Sciences, Tel Aviv University
Tel Aviv, Israel
afek@cs.tau.ac.il

Anat Bremler-Barr
Computer Science Dept., Interdisciplinary Center
Herzliya
Israel
bremler@idc.ac.il

Shir Landau Feibish
Blavatnik School of Computer Sciences, Tel Aviv University
Tel Aviv, Israel
shirl11@post.tau.ac.il

ABSTRACT

We present a basic tool for zero day attack signature extraction. Given two large sets of messages, P of messages captured in the network at peacetime (i.e., mostly legitimate traffic) and A captured during attack time (i.e., contains many attack messages), we present a tool for extracting a set S of strings, that are frequently found in A and not in P. Therefore, a packet containing one of the strings from S is likely to be an attack packet.

This is an important tool in protecting sites on the Internet from Worm attacks, and Distributed Denial of Service (DDoS) attacks. It may also be useful for other problems, including command and control identification, DNA-sequences analysis, etc. which are beyond the scope of this work.

Two contributions of this paper are the system we developed to extract the required signatures together with the problem definition and the string-heavy hitters algorithm. This algorithm finds popular strings of variable length in a set of messages, using, in a tricky way, the classic heavy-hitter algorithm as a building block. This algorithm is then used by our system to extract the desired signatures. Using our system a yet unknown attack can be detected and stopped within minutes from attack start time.

1. INTRODUCTION

Signature extraction is an important tool in several network security problems. In Distributed Denial of Service (DDoS) mitigation, for example, there has recently been a growing demand for zero day attack signature extraction solutions.

Two basic techniques are traditionally used to identify DDoS attacks, flow authentication based on challenge response and flow behavioral analysis based on statistics and learning. Recent attacks with millions of zombies generating seemingly legitimate flows go under the behavioral radar screen therefore leaving a loophole in the defense mechanisms and creating the demand for a DDoS zero day attack signature extraction solution.

Identifying signatures for unknown DDoS attacks is extremely difficult due to the seemingly legitimate content found in the packets which comprise the attack. Leading industry experts confirm, that the signatures found in recent zero-day DDoS attacks are usually a bi-product of the attack tools which the attackers use. These tools, often leave some footprint caused unintentionally by the program, such as a short string or some anomaly in the packet content structure. These subtle signatures are not identified by the current automated defense mechanisms, but rather by a manual process which may take hours or days. Clearly, in order to stop such unknown attacks while they are occurring, such signatures must be extracted quickly, hence automatically.

*This work is part of the Kabarnit-Cyber Consortium (2012-2014) under Magnet program, funded by the chief scientist in the Israeli ministry of Industry, Trade and Labor.
This research was also partly supported by European Research Council (ERC) Starting Grant no. 259085.

1.1 Zero-day Attack Signature Detection System

Generally speaking, leading security companies provide systems which offer several layers of defense against high-volume attacks. When all layers of defense fail, the attacked customer contacts the security company's support team to alert them and get their assistance in stopping the attack. Since the automated mechanisms were not able to identify or stop the attack, the attacked customer is in need of manual assistance. This manual assistance may include several stages, one of which is identifying attack signatures by a human expert. The attack mitigation process is therefore long and may take hours to days, in addition it is labor intensive. Moreover, in many cases the human eye misses the identifying string which could be an extra space, line-feed etc.

We present a system for automatic extraction of signatures for high volume attacks, using a constant amount of space and a single pass over the input.

Our system takes as input two samples of traffic collected during an attack and during peacetime. The peacetime traffic sample may be collected as a routine scheduled procedure. The attack traffic sample can be collected once the attack has been identified. We note that for DDoS attacks there are existing mechanisms for identifying when an attack has started and for differentiating between Flash events and DDoS attacks, for instance that of Park et.al. [18]. The system then analyzes both traffic samples to identify content that is frequent in the attack traffic sample yet appears rarely or not at all in the peacetime traffic.

Our system makes no assumptions on traffic characteristics such as client behaviour, address dispersion, URL statistics and so forth. Therefore, it is generic in that it can be easily adapted to solving other network problems with similar characteristics.

The following are the basic requirements of our system:

1. *Don't include signatures found in legitimate traffic.*
2. *Allow signatures of varying lengths.*
3. *Find a minimal set of signatures.*
4. *Minimize space and time usage.*

More specifically, given some constant k we wish to find all strings $s_1, ..., s_m$, s.t. $\forall i, 1 \leq i \leq m$:

1. $|s_i| \geq k$

2. s_i appears frequently enough in the attack traffic.

3. Either one of the following holds:

   (a) The frequency of s_i in peacetime is very low.

   (b) The frequency of s_i in peacetime is moderate, yet in the attack traffic its frequency is significantly higher.

4. In order to have a minimal set, no string s_i is contained in another string s_j.

These requirements are formally explained in Section 5.2.

In Section 6, we test our system on real life traffic logs of attacks and peacetime that from real attacks that have occurred recently.We show that our solution has good performance in real life, with a recall rate average of 99.95% and an average precision rate of 98%.

Additionally, our system makes use of an algorithm we have devised for finding heavy hitters in textual data which is described in Section 4 and is of independent interest.

2. BACKGROUND: HEAVY HITTERS

Our work deals with a variant of the heavy hitters problem which we call the *string heavy hitters*, which we will soon define.

The problem of finding the heavy hitters or frequent items in a stream of data is defined as follows: given a sequence of N values $\alpha = \langle \alpha_1, ...\alpha_N \rangle$, using a constant amount of space, find n_v values, each having a frequency (the number of times it appears in α) which is greater than θN. Many solutions have been proposed for the classical heavy hitters problem, for example, the solutions suggested in [14, 1, 4, 6, 12]. A description of a few counter-based algorithms as well as other significant results regarding the heavy hitters problem can be found in [2]. For our evaluation, we chose to implement the algorithm of Metwally et. al. [13], since it is simple yet it provides quite accurate counter estimations for values seen early in the stream [2]. The pseudo code of the algorithm in [13] is shown in Procedure *Metwally et.al. Heavy Hitters*.

Procedure Metwally et.al. Heavy Hitters

Data: $\langle \alpha_1, ...\alpha_N \rangle$, constant $n_v << N$
Result: n_v heavy hitters
initialization;
// Maintain n_v heavy hitter candidates.
$Frequent[1...n_v]$ = {$item = NULL$ and $count = 0$};
main;
for $i = 1 \rightarrow N$ **do**
  // If in Frequent, increment count.
  if $\exists j$ *s.t.* $Frequent[j].item == \alpha_i$ **then**
    $Frequent[j].count++$;
  else
    // Look for item with smallest count, and replace it.
    find j s.t. $\forall h$
    $Frequent[j].count \leq Frequent[h].count$;
    $Frequent[j].item := \alpha_i$;
    $Frequent[j].count++$;
  end
end
return $Items$;

The error rate ϵ of this algorithm is $\epsilon = \frac{N}{n_v}$ [13], meaning that each counter in the output of the algorithm is at most ϵ higher than the actual number of times that the value appeared in the stream. The algorithm performs in $O(N)$ time, it makes only a single pass over the input, and requires constant space.

Heavy hitters algorithms are usually performed on numeric data, whereas here the focus is on textual values. The *String Heavy Hitters* problem is defined as follows: given a sequence $S = \langle S_1, ...S_N \rangle$ of N strings and a constant k, using a constant amount of space, find n_v substrings of length at least k, each having a frequency[1] which is over some threshold θN, and no output string is contained in another output string.

Notice that although the Hierarchical Heavy Hitters algorithms (see for example [3]), may seem suited for textual data, they work well on data which forms a well defined hierarchical structure such as a sequence of IP addresses. Since our algorithm searches for recurring strings in the traffic, and the context of the strings is not relevant for our purposes, identical strings need to be grouped together regardless of what comes before them or after them in the content. Another related problem is that of compressed sensing. Many interesting works have been done in this field such as [5, 17, 19]. It has yet to be seen if the solutions presented for the compressed sensing problem can be adapted to outperform the above heavy hitters algorithms for the frequent items problem.

3. RELATED WORK

In the past, automated signature extraction has been mostly used as a tool for identifying computer malware such as worms and viruses. As such, most algorithms presented for this problem generally consist of two stages:

1) Identifying suspicious traffic which contains malware with high probability. This is done using methods such as honeypots [10], behavioural traffic analysis [20], etc.

2) Generating signatures for the suspicious content.

Therefore, the signature generation process of the previous works [8, 10, 9, 20, 7] done on malware identification, was based on the use of traffic that is known to be malicious. In our work we deal with the scenario in which the suspicious traffic can not be detected a-priori, but rather, the suspicious traffic contains some unique prevalent content which needs to be identified. Meaning, attack-time traffic is analyzed, parts of it may be malicious and others may be legitimate. Therefore it is crucial to identify which prevalent content is found only in malicious content and create signatures for that content alone. Furthermore, our methods allow us to identify not only seemingly legitimate malicious content, but it can in fact, be legitimate in other traffic. For example, in HTTP level attacks, an attacker can make use of a legitimate yet not commonly used HTTP header field. Use of a this field can, in this case, be an identifier of malicious traffic, yet in a different case be completely legitimate.

In addition, most of the previous works were done for signatures of a fixed length [8, 20, 7]. Finding varying-length signatures poses inherent difficulties. We note two works which have been done which generate varying length strings. The first is Honeycomb [10] which was presented by Kreibich and Crawcroft. There, signatures are created for suspicious traffic using pattern matching techniques. Specifically using searches for longest common substrings within packet payloads, using suffix trees. While this method allows creating varying-length signatures, and the suffix tree can be created in linear time using Ukkonen's online suffix tree construction algorithm [22], the space complexity of the suffix tree is at least linear

[1]Frequency of a substring s can be defined as the total number of times s appears in S, or as the number of strings in S in which s appears. For our purposes we will be using the latter.

in the size of the input, and therefore not scalable when dealing with large amounts of data.This is perhaps the most substantial difference from our solution which uses a constant amount of space while still maintaining a time complexity which is linear in the size of the input.

Another work in which varying-length signatures are generated, is Autograph [9], presented by Kim et. al. To generate varying length signatures, the payload of suspicious traffic is divided into variable-length content blocks based on the Content based Payload Partitioning method first presented in [15]. Content blocks are chosen as signatures based on their prevalence in the traffic flows. While the signatures produced are indeed of varying length, the Content based Payload Partitioning performed is done using a predetermined *breakmark* which is used to partition the payload into blocks whose size is no more and no less than some predefined values. Additionally, the average block size is also predetermined. Evaluation done in [9], shows that a larger minimum content block, such as 32 or 64 bytes is needed to avoid a high false positive rate. Signature structure is therefore based on predefined parameters which determine the breakmark and the signature length. The system presented in our work allows shorter signatures to be generated, and more importantly, does not use a predefined breakmark for content partition so that signatures can vary significantly from one another.

An interesting variation of the above problem is that of signature extraction solutions with the ability to support morphisms in malware. This problem was addressed in various works [21, 11, 16, 9], where different algorithms for automatic signature generation for polymorphic worms are presented. We are currently in the process of expanding our solution so that it may deal with such variations as well.

Another related problem is that of traffic behavioral analysis. Many different solutions for this problem have been proposed using techniques from the fields of machine learning, streaming algorithms, pattern matching and others. While the solution we propose may have interesting applications for this problem as well, we do not treat them in this work.

4. STRING HEAVY HITTERS

The String Heavy Hitters problem for packets is to find strings that are heavy hitters in a given sequence of packets[2]. Solutions which perform an exact count of the strings would use at least a linear amount of space [2], therefore a more efficient solution must be found.

This problem is closely related to the heavy hitters problem, defined in section 2. However, in order to find heavy hitters in textual data, we first need to define the input strings to the heavy hitters algorithm.

Define a $k\text{-}gram$ to be a string of chars of length exactly k. Our approach is to start with all $k\text{-}grams$ ($k = 8$ in our case), and then we need to resolve the following problems:

1. *The substring pollution problem:* If a string s, $|s| > k$ appears many times in the input text, then all the $k\text{-}grams$ which are substrings of s show up as heavy hitters and are output by the heavy hitters algorithm. We name this problem the *substring pollution* problem. The following is an example of the problem: suppose the signature is $abcabc$ and $k = 4$, than all the $4\text{-}grams$ which make up the signature, i.e., $abca$, $bcab$ and $cabc$ will be heavy hitters and will therefore *pollute* the data structure. We deal with this problem by combining $k\text{-}grams$ that have repeatedly appeared in sequence. Therefore, allowing the use of varying length grams, as described in section 4.1.

2. *The frequency estimation problem:* Another problem which arises when creating values from textual data is that heavy hitters may be substrings of one another. This can occur, for example, if both the strings $ABCDEF$ and $BCDE$ recur frequently in *separate* locations in the text. The counter of $BCDE$ provided by the algorithm would not reflect the times that $BCDE$ appeared as part of $ABCDEF$. In order to provide a better estimation of the frequency of each string, the algorithm must be modified to support this. We treat this issue using an additional procedure, which we describe in section 4.1.3.

4.1 The Double Heavy Hitters Algorithm

We propose the Double Heavy Hitters algorithm. The purpose of this algorithm is to identify frequent substrings of varying length in the given packets.

4.1.1 Overview

The *Double Heavy Hitters algorithm*, denoted DHH, makes use of two independent heavy hitters components, HH_1 and HH_2, as follows:

1. HH_1 finds $k\text{-}grams$ that appear frequently, i.e., that are heavy hitters.

2. HH_2 finds varying length strings that occur frequently in the input (which are combinations of the strings found in step 1).

The input to the DHH algorithm is a sequence of n_p packets, a constant k which will determine the size of the $k\text{-}grams$ used and a constant r which is a ratio that we will soon explain. Conceptually, the process works as follows: the algorithm traverses the packets one by one. For each index in the packet, a $k\text{-}gram$ is formed by taking the k characters starting from that index. These $k\text{-}grams$ are given as an input to HH_1. To form the varying length strings which are the input to HH_2, while HH_1 processes the $k\text{-}grams$, the algorithm seeks to find the longest run of consecutive $k\text{-}grams$ such that: *1*) they are all already in HH_1 (i.e., at this stage they are heavy hitters), *2*) they have similar counters. The objective is that combining two $k\text{-}grams$ should occur only if they should be part of the same signature. Without this ratio, if some $k\text{-}gram$ appears very frequently, but the character that usually follows this $k\text{-}gram$ is inconsistent, then the preferred signature should not combine this $k\text{-}gram$ with the one that follows it. Specifically, counters of two consecutive $k\text{-}grams$ maintain a ratio of r. In our experiments we tested r from 0 to 1. Since for our purposes a longer signature was preferable we use a ratio of 0.1 in our testing. Testing with a ratio of 0.5 or higher produced significantly shorter signatures. An example of this process can be seen in Figure 1. Once the entire input has been traversed, the algorithm outputs the items found in HH_2.

This process of creating a string from consecutive $k\text{-}grams$, is a key factor in substantially reducing the substring pollution in the output. For each such consecutive sequence, the process creates a single input to HH_2, which is a varying length sequence of values, that has been naturally filtered by a preceding heavy hitters procedure, HH_1.

4.1.2 The detailed Double Heavy Hitters Algorithm

[2]The problem can be generalized as follows: given a sequence of strings, we wish to find which substrings appear most frequently in the strings.

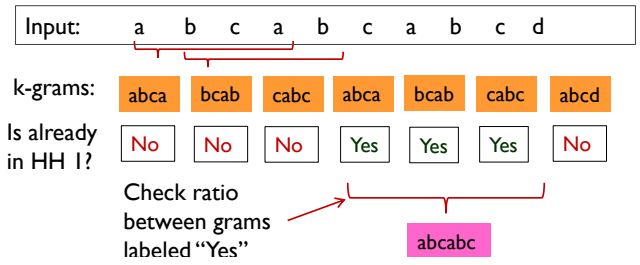

Figure 1: An example of the process of creating varying length strings from consecutive k-$grams$.

The pseudo code of the DHH algorithm is found in Procedure $DoubleHeavyHitters$, and makes use of the support functions $Init(n_v)$ and $Update(\alpha)$ (See Appendix A) which are derived from the algorithm in [13], and the sub-procedure *InputToHH2*. The output of the $DoubleHeavyHitters$ procedure is the list of heavy hitter values found in HH_2 at the end of the procedure.

The input provided to the algorithm is a sequence of n_p packets, and constant integers: k and r as explained above, and n_{HH_1} and n_{HH_2} which indicate the number of items HH_1 and HH_2 will be configured to hold respectively.

The algorithm works as follows: the packets are traversed one by one. For each index in the packet, a k-$gram$ is formed by taking the k characters starting from that index. The k-$gram$ is given as an input to HH_1, which in return provides the k-$gram$'s counter in HH_1 (a return value of zero indicates that this is a new k-$gram$).

In order to account for varying length strings, while performing the above traversal, an additional string s_{temp} is maintained. For any location in the packet, s_{temp} is the last longest heavy hitter string found until that location. s_{temp} is maintained in the following manner: At the beginning of each packet, the string s_{temp} is empty. For each k-$gram$ that is inserted to HH_1, we check its returned value:

1. If s_{temp} is empty and the returned value is greater than zero, s_{temp} is set to be this k-$gram$.
2. Otherwise, if s_{temp} is not empty, one of the following two occur:
   (a) If the returned value is equal to zero, s_{temp} which is the longest "heavy" string we found until here, is given as an input to HH_2, and s_{temp} is reset to empty.
   (b) Otherwise, the returned value is greater than zero. In this case, this value is compared with the counter value of the previous k-$gram$. If the ratio between the two values is over some predefined ratio r, s_{temp} is concatenated with the last character in the current k-$gram$.

The algorithm then proceeds to treat the next index. When all of the packets have been traversed, the algorithm outputs the output of HH_2.

We note, that the algorithm also maintains a set of all the treated strings in each packet so that each string is counted only once. This allows us to find strings that appeared frequently in different packets rather than strings that have a high overall frequency.

4.1.3 *Improving the frequency estimation*

Due to the frequency estimation problem, as explained in Section 4, it is possible that a string t in HH_2 may contain a substring

Procedure DoubleHeavyHitters

Data: sequence of n_p packets, constants k, n_{HH_1}, n_{HH_2}, and ratio r
Result: the n_{v_2} candidates for being the heavy hitters
initialization;
$s_{temp} = empty$;
$temp\_counter = 0$;
$HH_1.Init(n_{HH_1})$;
$HH_2.Init(n_{HH_2})$;
main;
for $i = 1 \rightarrow n_p$ **do**
  Denote $\alpha_1, ..., \alpha_h$ the bytes of packet p_i **for** $j = 1 \rightarrow h - k + 1$ **do**
    $counter = HH_1.Update(\alpha_i ... \alpha_{i+k-1})$;
    if $counter > 0$ **then**
      if $string == empty$ **then**
        $s_{temp} = (\alpha_i ... \alpha_{i+k-1})$;
        $temp\_counter = counter$;
      else
        if $counter/temp\_counter > r$ **then**
          $s_{temp} = s_{temp} || \alpha_{i+k-1}$;
          $temp\_counter = counter$;
        else
          Procedure $InputToHH2$;
        end
      end
    else
      Procedure $InputToHH2$;
    end
  end
end
for $i = 1 \rightarrow n_{HH_2}$ **do**
  for $j = 1 \rightarrow n_{HH_2}$ **do**
    if $i! = j$ *and* $item[i].ID$ *is a substring of* $item[j].ID$ **then**
      $item[i].count+ = item[j].count$;
    end
  end
end

Procedure InputToHH2

$temp\_counter = 0$;
if $s_{temp}! = empty$ **then**
  $HH_2.Update(s_{temp})$;
  $s_{temp} = empty$;
end

t' which is also a string in HH_2. However, when processing t in HH_2, the counter of t' is not incremented. The goal of our algorithm is to provide an estimate of the actual number of times that a string was encountered. In order to achieve a better estimation, we perform an additional procedure on the strings found in HH_2 at the end of the above algorithm, to find which items in HH_2 are substrings of other items in HH_2. The counter of the contained item is incremented by all of the counters of the items that contain it. In this manner, our final counters provide a better estimation of the number of packets in which each string was encountered.

4.2 Error Rate Analysis

The heavy-hitters algorithm that we use is an approximation algorithm, and therefore the DHH algorithm is also an approximation. As can be seen in the below analysis, the error rate of our algorithm is only a factor of 3 higher than that of the heavy-hitters algorithm that we use as a building block. In fact, as can be seen in the experimental results in Section 6, the error rate of our algorithm is significantly smaller in practice.

THEOREM 1. ***Bounds of the Double Heavy Hitters Algorithm:*** *The final counters provided by the algorithm may incur an error of at most* $3\frac{n_k}{n_{HH}}$ *where* $n_{HH} = min\{n_{HH_1}, n_{HH_2}\}$ *and* n_k *denotes the total number of* $k\text{-}grams$ *processed by the algorithm.*

PROOF. In order to analyze the error rate of our algorithm, we must first analyze the error rate of each of its components. As described in Section 2, the error rate of each of the HH items is $\epsilon = \frac{N}{n_{HH}}$, where n_{HH} is the number of items maintained by the HH, and N is the number of values in the input. We have defined the number of items maintained by HH_1 and HH_2 to be n_{HH_1} and n_{HH_2} respectively. Given an input sequence of packets, the size of the input is calculated as follows:

1. For HH_1: Define the total number of $k\text{-}grams$ in all the packets in the sequence to be n_k which is the bound on the size of the input to HH_1.

2. For HH_2: The input to HH_2 is made up of the strings which are a sequence of consecutive $k\text{-}grams$. Denote n_c the number of such strings. n_c is maximized when the inputs to HH_2 are all a single $k\text{-}gram$. To understand how these strings can be formed lets look at the example in Fig. 2. Suppose the $k\text{-}gram$ $abcd$ is a heavy hitter. In order for the string beginning with this occurrence of $abcd$ to be made up of a single $k\text{-}gram$, the following character e must be of a high variability in this context throughout the input. Otherwise, the $k\text{-}gram$ $bcde$ would also be a heavy hitter, and therefore $abcd$ would be merged with $bcde$, meaning the string would be longer than a single $k\text{-}gram$. One can see that this would be true for all the following $k\text{-}grams$ which contain the character e, and therefore they too can not be heavy hitters. The closest following $k\text{-}gram$ that can be a valid candidate for being a heavy hitter is the $k\text{-}gram$ following the character e. It follows that $n_c \leq \frac{n_k}{k+1}$.

It follows from the above calculation that the error rate of HH_1 is $\frac{n_k}{n_{HH_1}}$, and the error rate of HH_2 is $\frac{n_c}{n_{HH_2}} \leq \frac{n_k}{n_{HH_2}(k+1)}$.

In order to complete the analysis, it remains to account for occurrences of strings that are not produced as part of the input to HH_2. Generally, a string s is produced as an input to HH_2, if the $k\text{-}grams$ that comprise it are already found in HH_1. Lets take a look at the sequence of $k\text{-}grams$ processed by HH_1. For some index j, the j^{th} $k\text{-}gram$ will be found in HH_1 only if its frequency is over $\frac{j}{n_{HH_1}}$. Since this must be true for all $k\text{-}grams$ that comprise s, it follows that there can be at most $\frac{n_k}{n_{HH_1}}$ appearances of S that are not produced as part of in the input to HH_2.

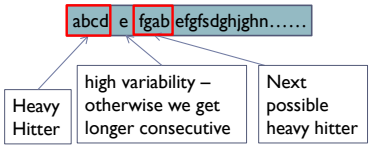

Figure 2: Non-consecutive heavy hitters

It follows that the overall error rate of our algorithm is $2\frac{n_k}{n_{HH_1}} + \frac{n_k}{n_{HH_2}(k+1)}$. Taking $n_{HH} = min\{n_{HH_1}, n_{HH_2}\}$, we get that the error rate of the algorithm is bound by $3\frac{n_k}{n_{HH}}$.

□

5. THE ZERO-DAY HIGH-VOLUME ATTACK DETECTION SYSTEM

The main purpose of our system is to efficiently extract a minimal set of signatures that distinguish malicious packets from good legitimate packets. Therefore, a major factor in producing signatures which achieve both a low false negative rate (i.e., a high detection rate) and a low false positive rate (i.e., a low rate of legitimate traffic that is wrongly identified as malicious), is the algorithm's ability to identify strings which appear very frequently in malicious traffic and which are hardly found in legitimate traffic.

5.1 System Overview

Given a sample of peacetime traffic and a sample of the attack traffic, the following three stages are performed:

1. Analyzing peacetime traffic

2. Analyzing attack traffic: the attack traffic is analyzed to identify strings that are very frequent in the attack traffic yet seldom or not found at all during peacetime.

3. Filtering the signature candidates: the strings found in the above step are filtered according to predefined frequency and containment requirements as will be explained in the following sections.

Note that for DDoS mitigation for example, the traffic that will be analyzed by our system can either be captured in the DDoS mitigation apparatus or in the cloud by sampling the traffic from several collectors. The signatures produced by our algorithm can be used by the anti-DDoS devices and firewalls to stop the attack. Using our algorithm, mitigation can be achieved in minutes, allowing proper defense against such attacks. Also, since DDoS attacks are usually high-volume attacks, a sample of the traffic is sufficient.

5.2 System Requirements

The system generates a *white-list* and a *maybe-white-list* using the following thresholds:

1. $Attack\text{-}high$: a string s can only be an attack signature if its frequency in the attack traffic is greater than $attack\text{-}high$.

2. $Peace\text{-}high$: a string s with a peacetime frequency over $peace\text{-}high$ can't be a signature for the malicious traffic, and will enter the *white-list*.

3. $Peace\text{-}low$: a peacetime frequency below $peace\text{-}low$ is deemed irrelevant for the attack signature selection process, and the string will be placed in the *not-white-list*.

4. $Delta$: a string s with a peacetime frequency between $peace\text{-}low$ and $peace\text{-}high$ can be considered as a possible signature for the malicious traffic only if its frequency in the attack traffic is at least $delta$ higher than its peacetime frequency, in this case it will enter the *maybe-white-list*.

Given a sequence of packets P of traffic captured during peace time and a sequence of packets A of traffic captured during an attack, and given the thresholds: $peace\text{-}high$, $peace\text{-}low$, $delta$ and $attack\text{-}high$, and some constant gram size k the problem is formally defined as follows: Find all strings $s_1, ..., s_m$, s.t. $\forall i, 1 \leq i \leq m$:

1. $|s_i| \geq k$

2. The frequency of s_i in the attack traffic is at least $attack\text{-}high$.

3. One of the following holds:

   (a) The frequency of s_i in peace time is less than $peace\text{-}low$.

   (b) Both of the following hold: 1) The frequency of s_i in peace time is between $peace\text{-}low$ and $peace\text{-}high$. 2) The difference between the frequencies of s_i in the attack traffic and in the peacetime traffic is at least $delta$.

4. To avoid redundancy, no string is contained in another (i.e., $\nexists j : s_j \subseteq s_i$ or $s_i \subseteq s_j$).

5.3 System Details

Our zero-day high-volume attack detection system makes use of our DHH algorithm, to analyze both the peace-time traffic and the attack traffic.

5.3.1 Analyzing peacetime traffic

Here we run the DHH on the peacetime traffic and categorize the strings in the output to three lists of strings, white-list, maybe-white-list and not-white-list, as explained above.

Note that to speed up mitigation, the peacetime traffic can be analyzed in advance to produce these lists. Additionally we note, that in some cases it is difficult to get a capture of peacetime traffic in advance since the mitigation device only receives attack time traffic. As can be seen in our evaluation (Section 6), those cases can be handled by other means.

5.3.2 Analyzing attack traffic

Here we run DHH on the attack traffic, with the modification that the algorithm omits potential output strings if they are equal to or contained in a string in the white-list, to reduce false-positives. The other way around is allowed (i.e., $www.facebook.com$ may appear frequently in the legitimate traffic, yet the string $www.facebook.com/BadPerson$ could appear frequently in the malicious traffic). We name this property the *one-way containment* property. Due to this problem, we can not filter out strings which appear frequently in legitimate traffic a-priori, but rather a more intricate solution is needed. Intuitively, the algorithm performs as follows: it receives as an input the sequence of packets captured during an attack, and a list of white-list strings. In order to avoid creating a signature for the attack traffic which appears as a string or a substring of a string in the white-list, the algorithm will only add a string to the input of HH_2 if it is not contained in a white-list string.

The main difference, therefore, between the DHH algorithm and the $Attack\text{-}DHH$ algorithm, is that the $Attack\text{-}DHH$ is provided with the white-list. Therefore, HH_2 is now updated with an s_{temp} only if s_{temp} is not found (as a whole white-list string or as part of one) in the white-list (see Fig.3). The only change therefore is in the sub-procedure $InputToHH2$. The pseudo-code of the modified sub-procedure can be seen in Procedure $Modified\text{-}InputToHH2$.

Procedure ModifiedInputToHH2

```
temp_counter = 0;
if s_temp != empty then
    if s_temp is not a string or part of a string in the white-list
    then
        HH_2.Update(s_temp);
    end
    s_temp = empty;
end
```

The strings output by the attack traffic analysis will be referred to as the signature candidates. A graphical depiction of the attack traffic analysis process and the filtering process described in the following step can be seen in Fig.3.

5.3.3 Filtering the signature candidates

Notice that all signature candidates in the output of the attack traffic analysis have a frequency below *peace-high* in the peacetime traffic. The strings in the output of the above step are narrowed down as follows:

1. Strings with a frequency in the attack traffic that is below the threshold *attack-high* are discarded

2. Check if any of the strings are equal to or contained in a string in the maybe-white-list. For such strings, calculate the difference between the frequency of the string during the attack and the frequency during peacetime of the relevant string in the maybe-white-list. If this difference is greater than the threshold *delta*, the string is kept, otherwise, it is discarded.We note that strings not found in the maybe-white-list must have a frequency below *peace-low* in the peacetime traffic.

3. Once the final signature candidates are acquired by the above process, they are checked for containment. If a signature candidate is contained in another signature candidate, the algorithm will only choose one signature based on user policy (i.e., the longest, the shortest, the one that produces the smaller number of false positives). Furthermore, the algorithm may further reduce the number of signatures by finding which signatures usually appear together in the same packets, therefore removing the redundant signatures. Selecting which signatures to discard can also be done based on user policy as described above.

6. EVALUATIONS

In our evaluation, we focus on high volume DDoS attacks, and specifically on unknown application layer attacks in HTTP requests, commonly known as HTTP-GET flooding attacks.

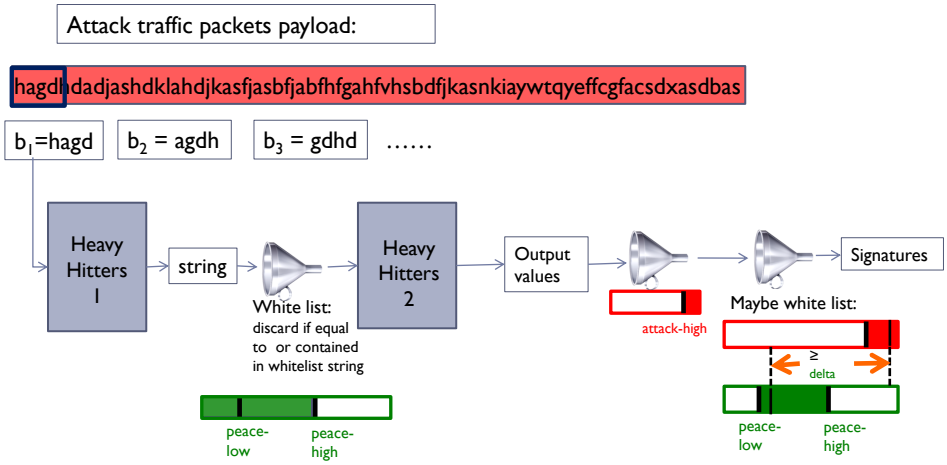

Figure 3: The process of extracting attack content signatures.

6.1 Test Setup

In our evaluations we used real captures from a top security company. Each test included a real HTTP-GET flooding attack time capture and a peacetime capture that included either real traffic or synthetically generated traffic. In some cases peacetime traffic was not available, and synthetically generated peace-time was used, such as a result of crawling through the victim site. If no such capture is available, we synthetically generate a peacetime capture by sending requests to the attacked server and capturing the traffic we create (i.e., a synthetic peacetime traffic capture). Our evaluation included 11 different attacks as follows:

1. We tested 3 attacks for which both the peace time capture and the attack time capture were recorded on the same server during a time of normal functioning and then later during an actual DDoS attack. We name these tests *real-real*.

2. We tested 6 attacks for which the attack time capture was recorded during an actual DDoS attack, and the peace time capture was created after the attack by recording traffic created by crawling the victim's site. We name these tests *real-synthetic*.

3. We tested a single attack which included textual log files of the HTTP GET requests during an actual DDoS attack, and a log file of HTTP GET requests which were identified as being legitimate during the time of the attack, which was used as the peace time traffic. We name these tests *log*.

4. We tested a single synthetic attack which was made up of peace time traffic which was captured by us and then a synthetic attack was merged into the peacetime traffic. We name these tests *synthetic-synthetic*.

For each of the above tests, the zero-day high-volume attack detection system was used to extract attack signatures. In order to evaluate the system's results, for each of the above scenarios, we preformed three tests:

1. System quality testing: Performed by evaluating both the recall and precision rates of the signatures extracted by the system. Recall and precision, which we will soon define, are standard measures of relevance in fields such as pattern recognition and information retrieval.

2. Frequency estimation accuracy test of the DHH algorithm: Performed by counting the number of packets in the *attack* traffic in which each of the attack signatures appears, and comparing the counters with the counters of the DHH algorithm.

3. Threshold testing: Several threshold value sets were tested.

A summary of the test statistics can be found in Table 1 which is explained in the next section.

6.2 System Quality Test Results

A summary of the test statistics is presented in Table 1. All of the attacks analyzed, are attacks that were not detected by any automated defense mechanism, and these attack samples were therefore analyzed manually by a human expert. The columns in the results section of the table are as follows:

1. *Manual attack rate estimation*: the estimated percent of the packets in the attack traffic capture, that were identified as attack packets by the manual analysis.

2. *System attack rate estimation*: the percent of the packets in the attack traffic capture, that contain one or more of the signatures extracted by the system.

3. *Recall rate estimation*: the percent of packets identified as attack packets by the manual analysis which were identified by the signatures extracted by our system. The aim is to have a recall of 100%, since the recall is an indication of how many of the relevant results were identified.

4. *Precision rate estimation*: we estimate the precision rate of our system by two methods, for both of which the aim is to have a precision of 100%, as precision is an indication of how many relevant results were returned as opposed to non-relevant results.:

   (a) *Peacetime based precision*: the percent of peacetime traffic packets that *were not* identified by the signatures extracted by our system either.

   (b) *Attack based precision*: the percent of attack traffic packets which were not identified by the manual analysis

that *were not* identified by the signatures extracted by our system either.

We note several comments and conclusions regarding the results: *1*) For each test, the system identified the signatures that were found by the human expert in addition to other signatures which were not identified by the expert.

2) For all of the attacks tested, one or more signature was found that creates a false positive of 0%, meaning they do not appear in the peacetime traffic at all. As explained in section 5.3.3, item 3, the final signature candidates may be filtered according to user policy. We chose to select the candidates with the lowest frequency in peacetime traffic, meaning the lowest false-positive rate. The final filtering process of the signature candidates, selected these signatures alone to achieve the results shown in the table. This filtering process was done by searching the peacetime traffic for the final signatures candidates to select those with the lowest false positive rate.

3) If both the attack and the peacetime captures are real, the system's attack detection rate is most likely to be very close or equal to the estimated detection rate of the manual analysis. On the other hand, as can be seen in tests 4 and 8 for example, a synthetic peacetime capture may cause a system detection rate which is higher than the manual estimation. The difference between them could indicate the false positive rate caused by the system's signatures.

4) All tests were performed with thresholds: $attack-high = 50\%$, $peace-high = 3\%$ $peace-low = 2\%$, $delta = 90\%$. Except for test 10 which was done with: $attack-high = 10\%$, $peace-high = 3\%$ $peace-low = 2\%$, $delta = 90\%$. The value of $attack-high$ was selected based on the characteristics of the attacks themselves and can be selected based on, for example performance variations in the attacked site and so forth. The rest of the thresholds were selected based on testing done, which is presented in section 6.4. There it is shown that a $peace-high$ value of 3 should be selected and determining the other two thresholds follows from setting this value.

Our testing included a preliminary phase for determining the settings and parameters of the DHH algorithm. These include the values of k, n_{HH_1}, n_{HH_2}, r, $attack\text{-}high$, $peace\text{-}high$, $peace\text{-}low$ and $delta$. The value k indicates the length of the $k\text{-}grams$, and n_{HH_1} and n_{HH_2} indicate the number of items each of the HH modules is configured to hold. The value of k was set to 8, since testing showed that longer signatures are likely to increase the rate of false negatives, and shorter signatures are often not substantial enough therefore increasing the possibility of false positives. The values of n_{HH_1} and n_{HH_2} were both set to be 3000. Our tests included values raging from 1000 to 10000, and it was found that 3000 is sufficient. The above values of these parameters were kept unchanged throughout the testing of the detection system. An additional parameter used by the DHH algorithm is the ratio r explained in Section 4.1.1. This value was tested within the detection system with values ranging from 0 to 1. It was found that values closer to 1 yielded the extraction of shorter signatures. This value should therefore be chosen based on the desired characteristics of the output. The thresholds which are used to determine the whitelists and the chosen signatures are configurable in the system and we discuss some tested values of these thresholds in Section 6.4.

The space required by our system is constant and dependant on n_{HH_1} and n_{HH_2}. The running times of the algorithm were measured for the samples, for which the system performance was tens of Mbps. The code that we used was not optimized for time efficiency therefore a more significant evaluation of the running times was not performed. Analyzing the time complexity of the system, it can be seen that had the code been optimized, the system should exhibit performance similar to that of the classic $SpaceSavingHeavyHitters$ algorithm [13], which was evaluated in [2].

6.3 Frequency estimation

Recall that to test the accuracy of the frequency estimation provided by the algorithm, the estimated frequency of each signature was compared to an actual count of the signature in the attack traffic. Figure 4 shows this comparison for the signatures of a single test. We also note that the average difference exhibited in this test between the estimated frequency and the actual frequency was under 1% over all of the 3000 signature candidates that were produced. This is much better than the analytical error bounds of the algorithm, which is probably due to the fact that the number of strings in the input to HH_2 is significantly smaller than the worst-case bound provided in the analysis in Section 4.2. The results of the comparison in the other tests were similar.

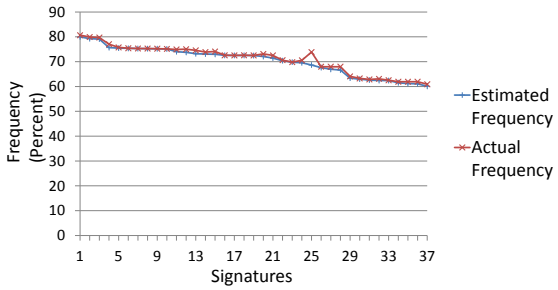

Figure 4: Signature frequency: algorithm estimation vs. the actual frequency

6.4 Threshold Testing

Both the false positive and false negative rate achieved by our system are influenced by the values of the thresholds discussed in Section 5.2. As part of our testing, a range of thresholds were tested. While intuitively, it may seem reasonable to take a $peace-high$ threshold that is relatively high (i.e., at least 50%), testing showed that this would lead to a very high false positive rate. An example of this can be seen in Fig. 5. This graph shows testing of different $peace\text{-}high$ values, on a single set of files. The graph shows the false positive rates caused by the different $peace\text{-}high$ values when all other values remain unchanged. The false positive rate shown in the dotted line measures the percent of peacetime packets identified by the generated signatures. The false positive rate shown in the whole line measures the percent of attack traffic packets identified by the generated signatures which are not malicious. As can be seen, a $peace\text{-}high$ value of 3 is the highest values that minimizes both false positive rates, therefore this is the value that was chosen for our tests.

6.5 Signature Examples

An interesting aspect of testing real attacks is to see the actual signatures for these attacks. Some examples of signatures include: An extra carriage-return (i.e., newline) somewhere in the packet payload where it was not usually found; Use of upper-case characters in a field which is normally found in legitimate traffic with lower-case characters; Use of an HTTP field that is rarely used; Use of a rare user agent. These signatures are a clear indication of the importance of analyzing the peacetime traffic.

Test Statistics										
Test Capture Files Data						Test Results				
Test	Target Category	Attack Time	Test type attack-peace	Number of Packets in Sample		Manual attack rate estimation	System attack rate estimation	Recall rate estimation	Precision rate estimation	
				Attack time	Peace time				Peacetime based	Attack based
1	Telephony	Nov 2011	Real-Real	407	2347	59%	59%	100%	100%	100%
2	eGaming	Jul 2012	Real-Real	157560	2468	98%	98%	99.8%	100%	100%
3	eGaming	May 2012	Real-Real	191192	47168	75%	75%	99.8%	100%	100%
4	National bank	Jan 2012	Real-Syn.	7050	369	78%	99%	100%	100%	79%
5	News	Mar 2012	Real-Syn.	47569	216	99.9%	100%	100%	100%	99.9%
6	eCommerece	Jan 2013	Real-Syn.	35014	253	NA	98%	NA	100%	NA
7	Mobile	May 2013	Real-Syn.	608	497	93%	94%	100%	100%	99%
8	Government	Mar 2012	Real-Syn.	6875	318	69.5%	90%	100%	100%	79.5%
9	Government	Mar 2012	Real-Syn.	5867	77	NA	92%	NA	100%	NA
10	News	May 2013	Log	34721	70322	47%	47%	100%	100%	100%
11	Synthetic	NA	Syn-Syn	57112	9016	84%	84%	100%	100%	100%

Table 1: Summary of the statistics of the tests performed. Note that the captures are *samples* of the traffic.

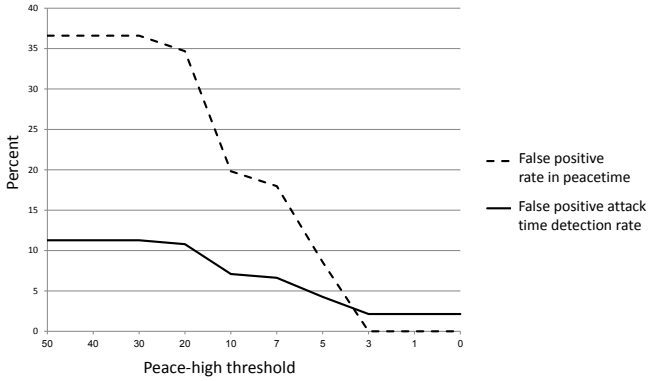

Figure 5: Comparing peace-high values.

7. CONCLUSIONS AND FUTURE WORK

We presented a system for zero-day attacks signature extraction. Our system made use of the DHH algorithm which we devised to solve the string heavy hitters problem. Testing our system on captures of real life attacks have shown that the signatures extracted by our algorithm detect high volume attacks with very high recall and precision rates.

This research opens many further directions which we would like to explore. Our main future goal is to expand the variability of the signatures that we are able to extract, to include, for example, signatures which include regular expressions, or signatures that contain "Don't-Care"s, and mismatches. We feel that this expansion of the problem may yield a result which is both of theoretical interest and will be of great use to the networking community.

Additionally, we would like to improve the robustness of our algorithm by identifying a generic white-list which would extenuate the need for acquiring peacetime traffic. We are also performing tests and adaptations for other attacks and anomalies that could be identified by this mechanism, for example, we are using our algorithm for identifying new malicious command and control servers.

Finally, as part of our ongoing work, we are developing a mechanism for identifying which of the signatures usually appear together in the same packets. In many cases, the system produces numerous signatures, which we would like to further reduce. To minimize the amount of signatures used, we are currently developing an additional process which finds the frequent sets of signatures that are often grouped together in the same packets. This information can allow us to group together signatures using relations of OR and AND. Signatures which frequently appear together in the same packets can be grouped using the AND relation to give a stronger indication of an existing malicious content, and therefore further reduce the false positive rates of the system. On the other hand, if signatures rarely appear together in the same packets, they can be grouped together with an OR relation to minimize the false negative rate of the system.

8. ACKNOWLEDGMENTS

We thank Matan Atad, Sharon Shitrit and Ziv Gadot from Radware Ltd. for helpful discussions.

9. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[2] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *PVLDB*, 1(2):1530–1541, 2008.

[3] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, pages 464–475, 2003.

[4] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In M. Farach-Colton, editor, *LATIN*, volume 2976 of *Lecture Notes in Computer Science*, pages 29–38. Springer, 2004.

[5] A. C. Gilbert, H. Q. Ngo, E. Porat, A. Rudra, and M. J. Strauss. ;2/;2-foreach sparse recovery with low risk. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *ICALP (1)*, volume 7965 of *Lecture Notes in Computer Science*, pages 461–472. Springer, 2013.

[6] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In S. Mehrotra and T. K. Sellis, editors, *SIGMOD Conference*, pages 58–66. ACM, 2001.

[7] K. Griffin, S. Schneider, X. Hu, and T. cker Chiueh. Automatic generation of string signatures for malware detection. In E. Kirda, S. Jha, and D. Balzarotti, editors, *RAID*, volume 5758 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2009.

[8] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *4th International Virus Bulletin Conference*, Sept. 1994.

[9] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286. USENIX, 2004.

[10] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *Computer Communication Review*, 34(1):51–56, 2004.

[11] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, pages 32–47. IEEE Computer Society, 2006.

[12] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. *PVLDB*, 5(12):1699, 2012.

[13] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In T. Eiter and L. Libkin, editors, *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[14] J. Misra and D. Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[15] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, pages 174–187, 2001.

[16] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241. IEEE Computer Society, 2005.

[17] H. Q. Ngo, E. Porat, and A. Rudra. Efficiently decodable compressed sensing by list-recoverable codes and recursion. In C. Dürr and T. Wilke, editors, *STACS*, volume 14 of *LIPIcs*, pages 230–241. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[18] H. Park, P. Li, D. Gao, H. Lee, and R. H. Deng. Distinguishing between fe and ddos using randomness check. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *ISC*, volume 5222 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2008.

[19] E. Porat and M. J. Strauss. Sublinear time, measurement-optimal, sparse recovery for all. In Y. Rabani, editor, *SODA*, pages 1215–1227. SIAM, 2012.

[20] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60. USENIX Association, 2004.

[21] Y. Tang and S. Chen. Defending against internet worms: a signature-based approach. In *INFOCOM*, pages 1384–1394. IEEE, 2005.

[22] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

APPENDIX

A. FUNCTIONS FOR THE DHH ALGORITHM

Function Init(V)

Items[V];
for $i = 1 \rightarrow V$ **do**
  $Items[i].count = 0$;
  $Items[i].ID = null$;
end

Function Update(α)

if $\exists j Items[j].ID == \alpha$ **then**
  $Items[j].count++$;
  $output = Items[j].count$;
else
  find j s.t. $\forall h\ Items[j].count \leq Items[h].count$;
  $Items[j].ID = \alpha$;
  $Items[j].count++$;
  $output = 0$;
end
return $output$;