Zero-Day Signature Extraction for High Volume Attacks

Yehuda Afek, *Member, IEEE,* Anat Bremler-Barr, *Member, IEEE,* and Shir Landau Feibish, *Member, IEEE,*

***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, thus allowing identification of the attack packets. This is an important tool in protecting sites on the Internet from Worm attacks, and Distributed Denial of Service (DDoS) attacks and may also be useful for other problems, including command and control identification, DNA-sequences analysis, etc. The main contributions of this paper are the system we developed to extract the required signatures together with the string-heavy hitters problem definition and the algorithm for solving this problem. 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. The algorithm runs in linear time requiring one-pass over the input. Our system makes use of this algorithm to extract the desired signatures. Furthermore, we provide an extended algorithm which is able to identify groups of signatures, often found together in the same packets, which further improves the quality of signatures generated by our system. Using our system a yet unknown attack can be detected and stopped within minutes from attack start time.**

***Index Terms*—High volume attacks, DDoS, zero-day attacks, signature extraction, heavy hitters.**

I. 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 behavioural analysis based on statistics and learning. Recent attacks with millions of zombies generating seemingly legitimate flows go under the behavioural radar screen. In these types of attacks, behavioural analysis does not succeed to detect the malicious traffic, as each zombie generates little traffic, which in itself may appear to be benign. Furthermore, the huge amount of attack sources makes it unfeasible to stop the attack at the source. Recent use of Internet-of-Things (IoT) devices in Botnets has caused further increase in the number of compromised machines which may take part in the attack [12]. This therefore leaves a loophole in the defense mechanisms and creates 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. Most traditional signatures are based on the malicious code that is expected in the attack packets, which may not be the case with modern DDoS attacks. Leading industry experts [1], [2] confirm, that the signatures found in recent zero-day application-level 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 (protocol complying) anomaly in the packet content structure. Such signatures allow fine grained identification of attack packets during an attack with minimal false positives or negatives.

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 and automatically.

A. Zero-day Attack Signature Extraction 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. This manual assistance may be composed of a number of procedures, including the identification of attack signatures. 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 single pass over the input, and space dependent only on the predetermined size of the heavy hitters data structure. Our system takes as input two streams (or stream samples): one of traffic collected during an attack and a second collected during peacetime. A 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,

This research is supported by The Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, by a grant of the European Research Council (ERC) Starting Grant no. 259085, and by the Ministry of Science and Technology, Israel. 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.

Y. Afek is with the Blavatnik School of Computer Science, Tel Aviv University.

A. Bremler-Barr is with the Efi Arazi School of Computer Science, Interdisciplinary Center Herzliya.

S. Landau Feibish is with the Department of Computer Science, Princeton University.

for instance that of Park et al. [29]. 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. That said, while our algorithm can generically work on different data types, our evaluation focuses on application-level DDoS attacks.

The following are the basic requirements of our system:

1) *Signatures should not be found frequently in legitimate traffic.* One of the main difficulties in differentiating between malicious traffic and traffic from legitimate sources, lies in the fact that malicious requests may have legitimate payloads. Identifying these malicious requests therefore becomes a significant challenge.
2) *Allow signatures of varying lengths.* The signatures produced by the algorithm must be of varying length. Setting a predefined constant length for signatures would create very problematic outcomes, as described in section IV-A.
3) *Find a minimal set of signatures.* Filtering devices such as intrusion detection systems search for the signatures in the packets. The complexity of this process is dependant on the number of signatures. Furthermore, they may have a limited capacity of signatures. The algorithm must therefore aim to produce a small number of signatures that capture the attack.
4) *Minimize space and time usage.* Our solution must maintain a high level of efficiency, such that the attack can be stopped quickly with minimal space usage.

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

1) $|s_i| \le k$
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 V-B.

In Section VI, we test our system on traffic logs of real attacks that have occurred in recent years. 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 IV-A and is of independent interest. This algorithm runs in a single pass over the input and space dependent only on some predefined parameters.

B. Open Implementation

An implementation of our solution is publicly available and may be used for signature extraction from user uploaded files. It is found on our website [4]. Users are advised to prepare a peacetime pcap file and an attack time pcap file which they may upload to the website for immediate signature extraction.

II. Preliminaries

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

A. The Heavy Hitters Problem

Definition 1. Heavy Hitter*: Given a sequence of N values $\sigma = \langle \sigma_1, \ldots, \sigma_N \rangle$ and a threshold $0 \le \phi \le 1$, using a constant amount of space, a* heavy hitter *is an item which has a frequency (the number of times it appears in σ) greater than ϕN.*

The problem of finding the heavy hitters or frequent items in a stream of data is: given a sequence of N values $\sigma = \langle \sigma_1, \ldots, \sigma_N \rangle$ find n_v heavy hitters using a constant amount of space.

Many solutions have been proposed for the classical heavy hitters problem, for example, the solutions suggested in [23], [5], [11], [14], [21]. A description of a few counter-based algorithms as well as other significant results regarding the heavy hitters problem can be found in [9]. For our evaluation, we chose to implement the algorithm of Metwally et al. [22], since it provides quite accurate counter estimations for values seen early in the stream [9]. The pseudo code of the algorithm in [22] is shown in Procedure *Metwally et al. Heavy Hitters*.

Algorithm 1: Metwally et.al. Heavy Hitters()

```
Data: $\langle \sigma_1, \ldots, \sigma_N \rangle$, constant $n_v << N$
Result: $n_v$ heavy hitters
// Maintain $n_v$ heavy hitter candidates.
$Frequent[1 \ldots n_v] = \{item = NULL$ and $count = 0\}$;
for $i = 1 \to N$ do
    // If in Frequent, increment count.
    if $\exists j$ s.t. $Frequent[j].item == \sigma_i$ then
        $Frequent[j].count++$;
    else
        // Look for item with smallest
           count, and replace it.
        find $j$ s.t. $\forall h$
          $Frequent[j].count \le Frequent[h].count$;
        $Frequent[j].item := \sigma_i$;
        $Frequent[j].count++$;
return $Items$;
```

The error rate ϵ of this algorithm is $\epsilon = \frac{N}{n_v}$ [22], 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.

B. String Heavy Hitters

Heavy hitters algorithms are usually performed on numeric data, whereas our work focuses on textual values.

Definition 2. String Heavy Hitter*: Given a sequence $S = \langle S_1; \ldots; S_N \rangle$ of N strings and a constant k, a string s is a* string heavy hitter *if it is a substring of one or more strings in S, is of length at least k, and has a frequency above the threshold N. Note that the 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.*

The *String Heavy Hitters problem* is defined as follows: given a sequence $S = \langle S_1; \ldots; S_N \rangle$ of N strings and a constant k, using a constant amount of space, find n_v string heavy hitters, such that no output string is contained in another output string.

Notice that although the Hierarchical Heavy Hitters algorithms (see for example [10]), 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 [6], [13], [27], [30], [38], [42]. 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.

III. Related Work

A. Automated Signature Extraction

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 [18], behavioural traffic analysis [32], etc.
2) Generating signatures for the suspicious content.

Therefore, the signature generation process of the previous works [16], [18], [17], [32], [15], [31] 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 [16], [32], [15]. Finding varying-length signatures poses inherent difficulties. We note two works which have been done which generate varying length strings. The first is Honeycomb [18] which was presented by Kreibich and Crowcroft. 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 [35], the space complexity of the suffix tree is at least linear 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 configurable fixed 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 [17], 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 [24]. 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 [17], 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.

In [34], the authors present an automated system for detection of new application signatures for the purpose of traffic classification. In this work, the authors present a system for automatically identifying keywords of unknown applications. The key difference between the solution presented in [34] and the solution we present here is that in [34] it is assumed that flows of the same application can be identified and therefore the analysis can look for the common strings in the specified flows. In our solution, one of the main difficulties is that we do not know which of the packets are contained in the attack and are therefore malicious and we therefore can not process these packets alone to find the attack signature.

In [40], a mechanism is presented for botnet $C\&C$ signature extraction. The mechanism identifies frequent strings in the traffic and then rank the frequent strings based on traffic clustering methods. While in [40] it is not assumed that the $C\&C$ connections can be identified a-priori, their analysis is based on characteristics of the connection and the traffic. Our solutions makes no such assumptions and is therefore more robust for dealing with specially crafted packets and attacks.

It is hard to achieve an "apples to apples" comparison of all of the prior signature extraction techniques mentioned above. This is especially true when comparing with works which use machine learning techniques such as [15], where time and space complexity are several orders of magnitude higher than the requirements of our system. Table I summarizes the main differences between our work and some of the previous works described above.

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 [33], [19], [26], [17], where different algorithms for automatic signature generation for polymorphic worms are presented. In future work, we are planning to expand our solution so that it may deal with such variations as well.

B. DDoS Defense Mechanisms

There has been a great deal of work done on mitigation of different types of DDoS attacks. Recent advances include solutions for mitigation of DDoS attacks in Software Defined Networks or cloud environments (For example [39], [36]).

In order to place our solution on the map of available DDoS solutions, we follow the classification of DDoS defense mechanisms according to place and time presented in [41]. The solutions we present are generally destination based solutions used during an attack (with a preparation stage to be performed before an attack) targeting application level attacks. Our solution is a content based packet filtering method and is not based on the packet route or parameters.

It may seem natural to compare our solution to solutions based on traffic anomaly detection. While our method does look for changes in content from peace time to attack time that exceed some predefined threshold, traffic anomaly detection methods in DDoS attacks, are usually network or destination based solutions searching for abnormal traffic patterns. Unusual traffic patterns may be detected using techniques such as machine learning [37], [20] or entropy [25], [28]. Our solution is not based on traffic behaviour and makes no assumptions on normal patterns of traffic. Solutions which use traffic behavioural analysis, may fail to detect large-scale DDoS attacks that simulate normal traffic behaviour. Since our solution makes no assumptions on the traffic behaviour it may be used to detect such attacks.

IV. The Double Heavy Hitters Algorithm

We propose the *Double Heavy Hitters* algorithm. An algorithm that identifies frequent substrings of varying lengths in the given packets. The algorithm is a crucial component in our signature extraction system. It is constructed from two heavy hitters modules, hence the name *Double Heavy Hitters*. The notations used for the algorithm are summarized in Table II.

A. Packet String Heavy Hitters

We define the *packet* variant of the String Heavy Hitters problem (see Definition 2) as follows: Given a sequence $P = \langle P_1, \ldots, P_N \rangle$ of N packets and a constant k, using a constant amount of space, find n_v heavy hitter strings, such that each has a frequency (the number of packets in which it appears) which is higher than N, and such that no output string is contained in another output string. We name this variant the Packet String Heavy Hitters problem.

B. Algorithm Overview

The *Double Heavy Hitters algorithm* (denoted DHH), uses two independent heavy hitters components, HH_1 and HH_2, as follows:

1) HH_1 finds k-$grams$ that appear frequently, i.e., that are heavy hitters.
2) HH_2 combines the k-grams of step 1 into varying length strings that occur frequently in the input.

Define a k-$gram$ to be a string of chars of length exactly k. The input to the DHH algorithm is a sequence of n_p packets, a constant k which will determine the size of the k-$grams$ used and a constant r which is the ratio between the frequencies of consecutive k-$grams$ explained shortly. Conceptually, the process works as follows: the algorithm traverses the packets one by one. For each index in the packet, a k-$gram$ is formed by taking the k characters starting from that index. These k-$grams$ 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-$grams$, the algorithm seeks to find the longest run of consecutive k-$grams$ 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-$grams$ should occur only if they should be part of the same signature. Without this ratio r, if some k-$gram$ appears very frequently, but the character that usually follows this k-$gram$ is inconsistent, then the preferred signature should not combine this k-$gram$ with the one that follows it. Specifically, counters of two consecutive k-$grams$ maintain a ratio of r. In our experiments we tested values of 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 Fig. 1. Once the entire input has been traversed, the algorithm outputs the items found in HH_2.

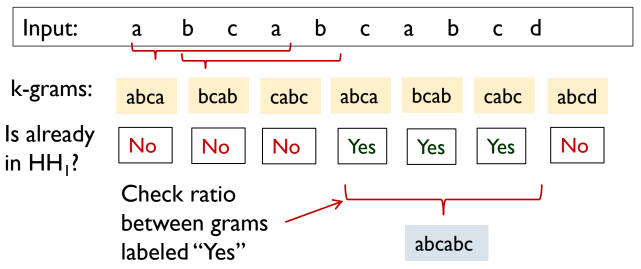


Fig. 1: Creating varying length strings from k-$grams$.

C. Challenges

The Packet String Heavy Hitters problem is closely related to the heavy hitters problem, defined in section II-A, however, applying the known heavy hitters algorithms to textual data is not at all trivial.

Specifically, the main problem is deciding how to partition the text into items. Since we are looking for varying length strings, we can not partition the text into disjoint parts as their length is not known a-priori. We must therefore consider

Solution	Signature length	Prior classification of attack traffic	Space Requirement	Passes on input
Our system	Varying length	None	Proportional to number of heavy hitters	Single pass
[18]	Varying Length	Malicious traffic known	Linear in input	Single pass
[16]	Fixed length	Malicious traffic known	Proportional to number of heavy hitters	Single pass
[15]	Fixed length	Malicious traffic known	Linear in input	Multiple
[32]	Fixed length	Malicious traffic known	Proportional to number of heavy hitters	Single pass
[17]	Varying length with limitations	Malicious traffic known a-priori	Linear in input	Multiple pass
[34]	Varying length	Content groups known a-priori	Linear in input	Single Pass
[40]	Varying length	Traffic clustered based on traffic characteristics and connections	Linear in input	Multiple passes

TABLE I: Comparison to previous techniques

k	minimal signature length (gram length)
r	ratio between the frequencies of consecutive k-grams
HH_j	Heavy hitters module j ($j \in 1, 2, 3$)
n_{HH_j}	number of items in the HH_j data structure ($j \in 1, 2, 3$)

TABLE II: Notations

a partition of the text into items which may overlap. Two main problems arise when counting the frequency of these overlapping items: the first is the *substring pollution problem*, and the second is the *frequency estimation problem.*

The substring pollution problem If a string s, $jsj > k$ appears many times in the input text, then all the k-$grams$ 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-$grams$ which make up the signature, i.e., $abca$, $bcab$ and $cabc$ will be heavy hitters and will therefore *pollute* the data structure. As explained in Section IV-B, our Double Heavy Hitters algorithm deals with this problem by combining k-$grams$ that have repeatedly appeared in sequence, therefore creating varying length grams. The process of creating a string from consecutive k-$grams$, is a key factor in substantially reducing the substring pollution in the output. For each such consecutive sequence, the process creates a single input of varying length to HH_2, that has been naturally filtered by a preceding heavy hitters procedure, HH_1.

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 as described in Section IV-B must be modified to support this. We treat this issue using an additional procedure, which we describe in section IV-D1.

D. Algorithm Details

The pseudo code of the DHH algorithm is given in Algorithm $DoubleHeavyHitters()$, which makes use of the class HH and its functions: $Init(n_v)$, $Update(\ )$ and $FixSubstringFrequency()$. The output of the $DoubleHeavyHitters()$ algorithm is a list of heavy hitter strings found in HH_2. 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-$gram$. Else, s_{temp} is given as an input to HH_2, and s_{temp} is set to be this k-$gram$.

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

The strings are checked for uniqueness before being inserted into HH_2 to ensure that each signature is only counted once per packet.

1) Improving the frequency estimation: Due to the frequency estimation problem, as explained in Section IV-A, it is possible that a string t in HH_2 may contain a substring

Algorithm 2: DoubleHeavyHitters

```
Class HH
    Members :
    Items;
    Functions :
    Procedure Init($n_v$)
        Items = allocate $n_v$ items
        for $i = 1 ! n_v$ do
            $Items[i]:count = 0$
            $Items[i]:ID = null$
    Procedure Update( )
        if $9j\ Items[j]:ID ==$ then
            $Items[j]:count++$
            $output = Items[j]:count$
        else
            find $j$ s.t. $8h\ Items[j]:count \quad Items[h]:count$
            $Items[j]:ID =$
            $Items[j]:count++$
            $output = 0$
        return $output$
    Procedure FixSubstringFrequency()
        for $i = 1 ! Items:length()$ do
            for $j = 1 ! Items:length()$ do
                if $i! = j$ and $Items[i]:ID$ is a substring of $Items[j]:ID$ then
                    $Items[i]:count+ = Items[j]:count$
Algorithm DoubleHeavyHitters
    Data: sequence of $n_p$ packets, constants $k$, $n_{HH_1}$, $n_{HH_2}$, and ratio $r$
    Result: the $n_{HH_2}$ candidates for being the heavy hitters
    $s_{temp} = empty$, $temp\_counter = 0$
    $HH_1 = new\ HH$, $HH_2 = new\ HH$
    $HH_1:Init(n_{HH_1})$, $HH_2:Init(n_{HH_2})$
    for $i = 1 ! n_p$ do
        // Denote $_1;:::;\ _h$ the bytes of packet $p_i$
        for $j = 1 ! h \quad k + 1$ do
            $counter = HH_1:Update(\ _j:::\ _{j+k\ 1})$
            if $counter > 0$ then
                if $s_{temp} == empty$ then
                    $s_{temp} = (\ _j:::\ _{j+k\ 1})$
                    $temp\_counter = counter$
                else
                    if $counter > r \quad temp\_counter$ then
                        $s_{temp} = s_{temp}jj\ _{j+k\ 1}$
                        $temp\_counter = counter$
                    else
                        if $s_{temp}! = empty$ then
                            $HH_2:Update(s_{temp})$
                        $s_{temp} = (\ _j:::\ _{j+k\ 1})$
                        $temp\_counter = counter$
            else
                if $s_{temp}! = empty$ then
                    $HH_2:Update(s_{temp})$
                $temp\_counter = 0$
                $s_{temp} = empty$
    $HH_2:FixSubstringFrequency()$
    return $HH_2:Items$
```

t^0 which is also a string in HH_2. However, when processing t in HH_2, the counter of t^0 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.

E. 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 VI, 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 $n_{HH} = minfn_{HH_1};n_{HH_2}g$ and n_k denotes the total number of k-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 II-A, the error rate of each of the HH items is $\ = \frac{N}{n_{HH}}$, 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-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-grams. Denote n_c the number of such strings. n_c is maximized when the inputs to HH_2 are all a single k-gram. To understand

how these strings can be formed lets look at the example in Fig. 2. Suppose the *k-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-gram*, the following character *e* must be of a high variability in this context throughout the input. Otherwise, the *k-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-gram*. One can see that this would be true for all the following *k-grams* which contain the character *e*, and therefore they too can not be heavy hitters. The closest following *k-gram* that can be a valid candidate for being a heavy hitter is the *k-gram* following the character *e*. It follows that $n_c \quad \frac{n_k}{k+1}$.

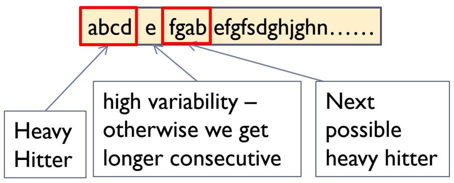

abcd
e
fgab
efgfsdghjghn……
Heavy Hitter
high variability – otherwise we get longer consecutive
Next possible heavy hitter

Fig. 2: Non-consecutive heavy hitters

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 $\frac{n_c}{n_{HH_2}} \quad \frac{n_k}{n_{HH_2}(k+1)}$.

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-grams* that comprise it are already found in HH_1. Lets take a look at the sequence of *k-grams* processed by HH_1. For some index j, the j^{th} *k-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-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.

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 $n_{HH} = minfn_{HH_1}; n_{HH_2}g$, we get that the error rate of the algorithm is bound by $3\frac{n_k}{n_{HH}}$. □

V. 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 legitimate ones. 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.

A. 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.

B. System Requirements

Four thresholds tune the operation of the system:

1) *Attack-high*: a string s is an attack signature if its frequency in the attack traffic is higher than *attack-high*.
2) *Peace-high*: a string s with peacetime frequency higher than *peace-high* cannot be a signature of malicious traffic.
3) *Peace-low*: a string with a peacetime frequency lower than *peace-low* is a signature of an attack provided its frequency during an attack is higher than *Attack-high*.
4) *Delta*: a string s whose peacetime frequency is between *peace-low* and *peace-high* is a signature of malicious traffic only if its frequency in the attack traffic is higher by at least *delta* than its peacetime frequency.

To extract the malicious signatures, the system maintains a *white-list* and a *maybe-white-list*. The *white-list* contains the list of strings whose peacetime frequency is higher than *Peace-high*. We assume that these strings can not be signatures of an attack as they can not be used to differentiate between peacetime and attack-time traffic, or otherwise would result in very high false positives. The *maybe-white-list* contains strings which are sometimes found in peacetime traffic, that is, their peace-time frequency is at least *Peace-low* and at most *Peace-high* . These strings may be used as attack signatures only if their frequency in an attack is significantly higher than there peacetime frequency. Any string with a peace-time frequency lower than *Peace-low* may be used as signatures for malicious traffic.

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-high*, *peace-low*, *delta* and *attack-high*, and some constant gram size k the problem is formally defined as follows: Find all strings $s_1; :::; s_m$, s.t. $8i; 1 \quad i \quad m$:

1) $js_ij \quad k$
2) The frequency of s_i in the attack traffic is at least *attack-high*.
3) One of the following holds:

a) The frequency of s_i in peace time is less than *peace-low*.

b) Both of the following hold: 1) The frequency of s_i in peace time is between *peace-low* and *peace-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., $\nexists j : s_j \quad s_i$ or $s_i \quad s_j$).

C. 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.

1) Analyzing peacetime traffic: Here we run the DHH on the peacetime traffic and categorize the relevant strings into the *white-list* and *maybe-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 VI), those cases can be handled by other means.

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.

Denote the modified DHH algorithm used during an attack as $Attack\text{-}DHH$. The main difference, 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).

To accommodate this change in the code of Algorithm $DoubleHeavyHitters$, we add the $AttackUpdate$ function to class HH. As can be seen below in Procedure $AttackUpdate$, the function receives both the string and an additional parameter $whiteList$ which is a pointer to a list of strings. Only if the input string is not found in the list pointed to by $whiteList$, the function will call $Update$. Furthermore, lines 41 and 46 in Algorithm $DoubleHeavyHitters$ should be replaced accordingly with $HH_2{:}AttackUpdate(s_{temp}; white \quad list)$.

We note that there can be numerous options for creating a data structure to support the search in the white-list. In our implementation we chose to maintain a hash table of all of the substrings in the white-list of length greater than k. This implementation is very good in terms of time complexity, though there is a tradeoff in that it takes a bit more space than other possible solutions.

Procedure 1: AttackUpdate

```
1 Class HH
2     Procedure AttackUpdate( , whiteList)
3         if $s_{temp}$ 2 whiteList OR $s_{temp}$ substring of
          string in whiteList then
4             return 0;
5         else
6             Update( );
```

The strings output by the attack traffic analysis will be referred to as the signature candidates. Fig.3 presents a graphical depiction of the attack traffic analysis process. The packets are first processed using the modified DHH algorithm we have just described. Then, the signatures are filtered according to the process described in Section V-C3, using the thresholds presented in Section V-B.

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) Discard strings with a frequency in the attack traffic that is below the threshold *attack-high*.
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, otherwise they would have been filtered out before entering HH_2.
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.

D. Identifying common combinations of signatures

In many cases it is interesting to identify signature combinations which are often found together in the same packet. These combinations can be of great use in attack detection mechanisms. First, they can be used to minimize the number of signatures which are needed to identify the attack (see

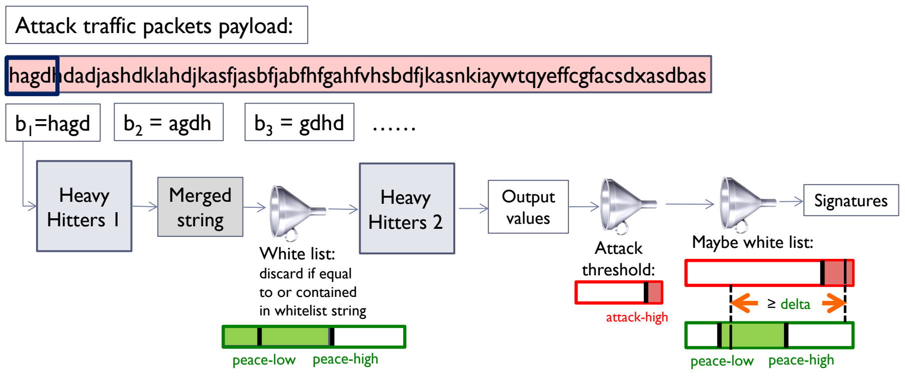


Fig. 3: The process of extracting attack content signatures.

subsection V-D2). Second, signature combinations can be used to increase the confidence level in the detection of the attack (see subsection V-D3).

1) The Triple Heavy Hitters algorithm: In order to identify the frequent signature combinations, we propose the *Triple Heavy Hitters* algorithm denoted THH. This algorithm makes use of three heavy hitter modules. Two modules will be used in a similar manner as was used in the *Double Heavy Hitters* algorithm (denoted DHH), and the third module will be used to find heavy hitters of signature combinations. While performing the DHH algorithm, for each packet treated, the THH algorithm maintains the set of strings which were identified as potential signatures, and therefore inserted into HH_2 while processing the packet. Once the entire packet has been processed, this set contains all of the signatures found in the packet and it will be inserted into the third heavy hitters calculation unit HH_3. To do so, each string in the set is concatenated with a special end-of-string delimiter and the delimited strings are concatenated together in lexicographical order to form a single string which is inserted into HH_3. Once all of the packets have been traversed, HH_3 contains the heavy hitter sets of signatures. This procedure is illustrated in Fig. 4. The pseudo code can be seen in Algorithm *TripleHeavyHitters*, which makes use of the class HH and its functions which is presented in Algorithm *Double Heavy Hitters*.

The THH algorithm has the same time complexity as the DHH algorithm, since the input to HH_3 is created as the strings are inserted into HH_2. The space complexity of the THH algorithm is dependent on the number of items in each of the HH modules.

2) Minimizing the Number of Signatures: Minimizing the number of signatures can be very significant, as some of the filtering mechanisms have a limited capacity. In addition, having less signatures can cause a reduction of the false positive rate of the signatures.

The ability to minimize the number of signatures is depicted in an example presented in Fig. 5. There, we look at a scenario in which there are six different types of attack packets. In this example, we can see, that four signatures have been extracted. However, since either the signature "bad" or "guy" appear in all of the different attack packets, they alone can be used, hence the number of signatures can be minimized without creating false negatives.

Once the signatures and the frequent signature sets are extracted by the system, we would like to check if the number of signatures can be minimized. To do so, we propose a greedy process. The pseudo code for this process is shown in Algorithm *MinimizeSignatures*. Each such set represents a packet type that had been found in the traffic sample. Intuitively, if some group of signatures appears together in some packet type, then only the signature with the highest frequency is needed to cover this packet type. To identify these signatures with the highest frequency, the process sorts all of the signatures according to decreasing frequency. The signatures are then traversed one by one, and checked to see how many "un-covered" packet types are covered by each signature. We denote a "covered" packet type as a packet type that has at least one signature from it that has been chosen as a final signature. Looking at the example shown in Fig. 5, the process would work as follows: The signatures are sorted in decreasing frequency. The most frequent signature is "bad", therefore we start with it. Since "bad" is the first signature we deal with, no packet types have been covered yet. Denote the $cover\_rate$ of a signature to be the percent of sets that it covers in this calculation. Therefore "bad" covers the packet types:$1; 2; 4; 5; 6$, and therefore we indicate its $cover\_rate$ to be 71%. The next signature we traverse is "guy". The only packet type that remains un-covered is number 3. The signature "guy" covers packet type number 3, and therefore we indicate its $cover\_rate$ to be 20%. Since all of the packet types have been covered, the $cover\_rate$ of the remaining signatures is 0%. Therefore, the only signatures needed to cover all of the packet types are "bad" and "guy".

The time complexity of this procedure in the worst case is $O(number\ of\ signatures \quad number\ of\ sets)$ which is $O(n_{HH_2} \quad n_{HH_3})$, and is therefore dependent only on the predefined size of each HH module. Since it is only done once, it only adds a constant overhead to the time complexity of the THH algorithm. The space requirements are linear in $n_{HH_1}; n_{HH_2}$ and n_{HH_3} which are configurable parameters.

3) Reducing the False Positives: Signature combinations can be used to increase the confidence level in the detection of the attack, by creating rules which are meant to identify specific attack content. Such specific rules reduce the chance

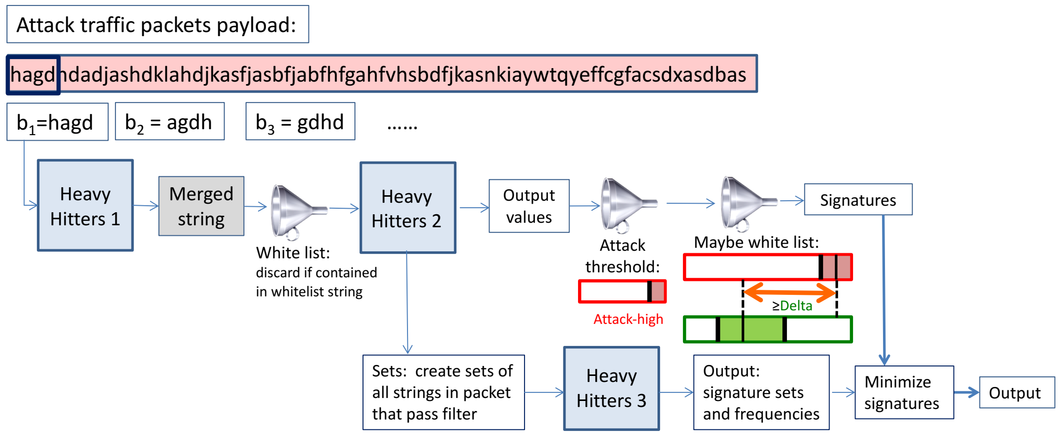
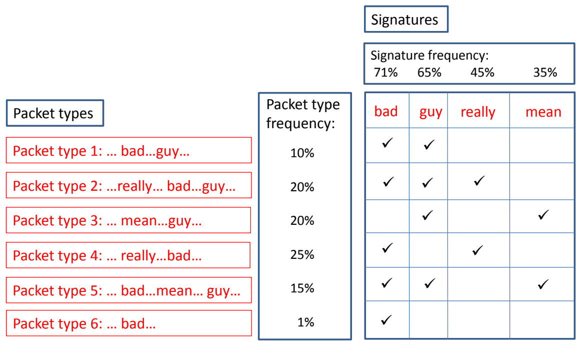

Attack traffic packets payload:
hagdhdadjashdklahdjkasfjasbfjabfhfgahfvhsbdfjkasnkiaywtqyeffcgfacsdxasdbas
b1=hagd
b2 = agdh
b3 = gdhd
……
Heavy Hitters 1
Merged string
White list: discard if contained in whitelist string
Heavy Hitters 2
Output values
Attack threshold:
Attack-high
Maybe white list:
≥Delta
Signatures
Sets: create sets of all strings in packet that pass filter
Heavy Hitters 3
Output: signature sets and frequencies
Minimize signatures
Output

Fig. 4: Extracting attack signatures with the additional minimization process.


Signatures
Signature frequency:

Packet types	Packet type frequency:	bad	guy	really	mean
Signature frequency:		71%	65%	45%	35%
Packet type 1: … bad…guy…	10%	✓	✓		
Packet type 2: …really… bad…guy…	20%	✓	✓	✓	
Packet type 3: … mean…guy…	20%		✓		✓
Packet type 4: … really…bad…	25%	✓		✓	
Packet type 5: … bad…mean… guy…	15%	✓	✓		✓
Packet type 6: … bad…	1%	✓			

Fig. 5: Different sets of signatures in different packet types.

of falsely identifying benign content as malicious, therefore making the identification of the attack traffic more certain.

In the example presented in Fig. 5, suppose packet type 6 doesn't contain malicious content. The signature ”bad” is found in the packet types:1; 2; 4; 5; 6, therefore if we create a rule which simply searches for the signature ”bad”, it will catch packets of type 6 as well, creating false positives. These false positives can be eliminated using detection rules which combine signatures with ”AND” and therefore can be used to catch specific types of attack packets. For example, we can create a rule which catches packets that contain *f*”bad” AND ”really”*g*. Such a rule will catch only packets of type 4. If we specify an additional rule that catches *f*”bad” AND ”guy”*g*, the packet types which will be caught are 1; 2; 4; 5, whereas packet type 6 will not be caught. Using such rules therefore reduces the likelihood of false positives and increases the confidence level of the detection.

We leave the process of generating the relevant rules to the system administrator's subjective preferences, and thus this process cannot be done automatically by our system. That said, the information gathered by the system can be used as the basis for creating such rules. A fine grained rule can be created for each packet type by combining all the signatures found in each packet type. In the above example, a rule searching for *f*”really” AND ”bad” AND ”guy”*g* would capture only packets of type 2. Alternatively, as shown above, rules combining less signatures can be used to detect larger groups of packets. For example, a rule searching for *f*”really” AND ”bad”*g* would capture both packets of types 2 and 4 and therefore an additional rule for type 4 would not be needed. These examples exhibit the tradeoff between how specific each rule is and the number of rules that are maintained.

VI. 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.

A. 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 and the attack time captures 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*.

Algorithm 3: TripleHeavyHitters()

```
Data: sequence of $n_p$ packets, constants $k$, $n_{HH_1}$, $n_{HH_2}$,
      $n_{HH_3}$, delimiter string $S_d$ and ratio $r$
Result: list $L_{sigs}$ of $n_{HH_2}$ candidates for being the
        heavy hitters, list $L_{sets}$ of $n_{HH_3}$ sets of
        signatures
1  $s_{temp} = empty$, $temp\_counter = 0$,
     $strings\_counter = 0$, $signature\_set = empty$;
2  $HH_1 = new\ HH$, $HH_2 = new\ HH$, $HH_3 = new$
     $HH$;
3  $HH_1{:}Init(n_{HH_1})$, $HH_2{:}Init(n_{HH_2})$,
     $HH_3{:}Init(n_{HH_3})$;
4  for $i = 1\ !\ n_p$ do
5      $signature\_set = empty$;
6      for $j = 1\ !\ h \quad k + 1$ do
           // Do the same as in the double
              heavy hitters algorithm while
              keeping the set of signatures
              for each packet
7          $counter_1 = HH_1{:}Update(\ _i{:::}\ _{i+k}\ _1)$;
8          if $counter_1 > 0$ then
9              if $s_{temp} == empty$ then
10                 $s_{temp} = (\ _i{:::}\ _{i+k}\ _1)$;
11                 $temp\_counter = counter_1$;
12             else
13                 if $counter_1 > r \quad temp\_counter$ then
14                     $s_{temp} = s_{temp}jj\ \ _{i+k}\ _1$;
15                     $temp\_counter = counter_1$;
16                 else
17                     if $s_{temp}! = empty$ then
18                         $counter_2 = HH_2{:}Update(s_{temp})$;
19                     if $counter_2 > r \quad strings\_counter$
                         then
20                         $signature\_set{:}Add(s_{temp})$;
21                         $strings\_counter = counter_2$;
22                     $s_{temp} = (\ _i{:::}\ _{i+k}\ _1)$;
23                     $temp\_counter = counter_1$;
24         else
25             if $s_{temp}! = empty$ then
26                 $counter_2 = HH_2{:}Update(s_{temp})$;
27             if $counter_2 > r \quad strings\_counter$ then
28                 $signature\_set{:}Add(s_{temp})$;
29                 $strings\_counter = counter_2$;
30             $temp\_counter = 0$;
31             $s_{temp} = empty$ ;
32     if $signature\_set{:}Size > 0$ then
           // concatenate signatures from
              the set of each packet to a
              single delimited string
33     $sigs = empty$;
34     for $i = 1\ !\ signature\_set{:}Size$ do
         $sigs = sigsjjS_djjsignature\_set[i]$ ;
35     $HH_3{:}Update(sigs)$ ;
36 $HH_2{:}FixSubstringFrequency()$;
```

Algorithm 4: MinimizeSignatures()

```
Data: list $L_{sigs}$ of $n_{HH_2}$ signatures, list $L_{sets}$ of $n_{HH_3}$
      sets of signatures
Result: the final list of signatures
// Initialize the $cover\_rate$ of all
   signatures to be zero
1  list $L_{final} = empty$;
2  for $i = 1\ !\ n_{HH_2}$ do $L_{sigs}[i]{:}cover\_rate = 0$ ;
3  Sort $L_{sigs}$ by frequency;
4  i = 0;
5  while $i < n_{HH_2}$ and $L_{sets}$ not empty do
6      for $j = 0\ !\ L_{sets}{:}size()$ do
7          if $L_{sets}[j]$ contains $L_{sigs}[i]$ then
8              $L_{sigs}[i]{:}cover\_rate+ = L_{sets}[j]{:}frequency$;
9              remove set $L_{sets}[j]$ from $L_{sets}$;
10 for $i = 0\ !\ n_{HH_2}$ do
11     if $L_{sigs}[i]{:}cover\_rate > 0$ then
         $L_{final}{:}insert(L_{sigs}[i])$ ;
12 return $L_{final}$;
```

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 on several threshold value sets.

A summary of the test statistics can be found in Table III which is explained in the next section. In addition, we performed a separate testing of the use of the *Triple Heavy Hitters* algorithm (explained in Section V-D) for identifying frequent signature combinations to minimize the number of signatures needed, as described in Section VI-F.

B. System Quality Test Results

A summary of the test statistics is presented in Table III. 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 recall is an indication of how many of the relevant results were identified.
4) *Precision rate estimation*: Precision is an indication of how many relevant results were returned as opposed to non-relevant results. We estimate the precision rate of our system in two ways:
   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 V-C3, 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. Another option would be to minimize the signatures based on frequent signature combinations as we have shown in Section VI-F, this also gives good results.

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 VI-E. 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 was sufficient for our purposes.

Note that as a rule of thumb, the size of n_{HH_1} can be determined according to the expected frequency of the signature that the system should identify and the average length of each packet. In general, in order to extract a signature which is found in fraction x of the packets ($0 \quad x \quad 1$) with average packet length being len, we would need to set n_{HH_1} to be no more than $\frac{len}{x}$. Furthermore, the sizes of n_{HH_2} and n_{HH_3} are bounded by the size of n_{HH_1}.

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 IV-B. 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 white-lists and the chosen signatures are configurable in the system and we discuss some tested values of these thresholds in Section VI-E.

C. Performance

Our implementation was done using C++, and made use of the implementation provided in [9] of the heavy hitters algorithm presented in [22]. The code was compiled using $g++$. We ran experiments on a 4-Core Intel(R) Core i7(R) $2{:}7$ GHz with 16 GB of RAM running Mac OS X 10.9 (Mavericks). We ran our algorithm on a variety of real attack captures, described in Table III. We also ran tests on synthetically generated traffic as well as a large capture of peacetime traffic. Using a single core, our algorithm was able to process between 144 and 232 Mbps when running on the real traffic traces presented in Table III. When running our algorithm on synthetically generated traffic which has skewed data frequencies the algorithm performance reached approximately $1{:}1$ Gbps.

As we show in our accuracy results, our algorithm is able to correctly identify the attack signatures even with only a small traffic sample. That said, we wanted to evaluate our algorithm's performance on a larger traffic capture. We captured over $1GB$ of real traffic from a web content providing server, and ran our system for peacetime analysis on this capture, achieving an average throughput of 206 Mbps.

The string manipulation performed by our algorithm is an elaborate task and therefore the throughput achieved by our

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 sites	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 III: Summary of the statistics of the tests performed. Note that the captures are *samples* of the traffic.

system for such a task is quite high. Furthermore, as our system only requires a small traffic sample to extract the attack signatures, the time required for the system to process the attack sample and output signatures is extremely short and provides the ability to promptly mitigate the attack.

We note that most of the running time of our system is spent updating the heavy hitters data structures. It is possible that these running times may be improved at the cost of accuracy by implementing a structure based on static memory allocation such as that presented in [7]. The running time may be further improved by using a randomized algorithm as presented in [8]. Since our running times are sufficient for our purpose we would rather not compromise accuracy for improving them. The space required by our system is linear in n_{HH_1}, n_{HH_2} and n_{HH_3}, which were set to $n_{HH_1} = 3000$, $n_{HH_2} = 200$ and $n_{HH_3} = 100$.

D. 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. Fig. 6 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 IV-E. The results of the comparison in the other tests were similar.

E. 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 V-B. We tested a range of thresholds. While intuitively, it may seem reasonable to take a $peace\quad high$ 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. 7. This graph shows the false positive rates caused by the different $peace$-$high$ values, on a single set of files, when all other values

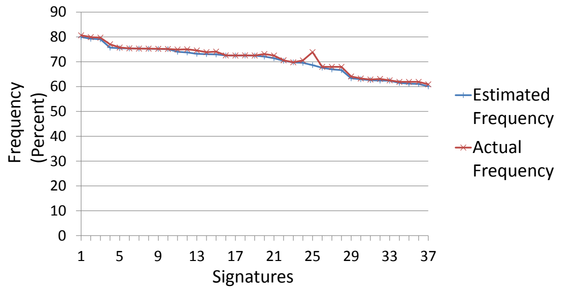


Fig. 6: Accuracy of algorithm signature frequency estimation

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 $peace$-$high$ value of 3 is the highest values that minimizes both false positive rates, therefore this is the value that was chosen for our tests.

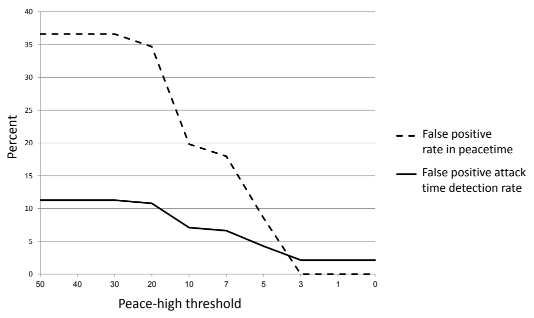


Fig. 7: Comparing peace-high values.

F. Testing Frequent Signature Combinations

We have tested our enhanced system which makes use of the *Triple Heavy Hitters* algorithm (explained in Section V-D) for identifying frequent signature combinations. The graphs in Fig. 8a and Fig. 8b respectively depict the results of tests 2 and 3 detailed in Table III. As shown in the table, for both of these tests, we have both real attack traffic captures and real peacetime traffic captures. The system identified the frequent signature combinations and then performed the algorithm for minimizing the number of signatures presented in Section V-D2. The results presented show the tradeoff between the precision and recall rates when selecting an increasing number of signatures. The results shown indicate that for the tested samples the number of signatures can be decreased substantially, thereby increasing precision significantly with almost no reduction of the recall rates.

G. 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.

VII. Conclusions and Future Work

We present a system for zero-day attacks signature extraction, extending our previous work presented in [3]. Our system makes use of the DHH and THH algorithms 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.

VIII. Acknowledgments

We thank Ziv Gadot from Red Button, and Matan Atad and Sharon Shitrit from Radware Ltd., for helpful discussions. We thank Golan Parashi for his help with code implementation.

References

[1] Radware website. *https://www.radware.com/documents/media-coverage/ddos-attacks-grow-in-sophistication*, 2011.

[2] Red button website. *https://www.red-button.net/ddos-glossary/signatures/*, 2018.

[3] Yehuda Afek, Anat Bremler-Barr, and Shir Landau Feibish. Automated signature extraction for high volume attacks. In *Symposium on Architecture for Networking and Communications Systems, ANCS '13, San Jose, CA, USA, October 21-22, 2013*, pages 147–156. IEEE Computer Society, 2013.

[4] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Golan Parashi. Cloud-based implementation of the signature extraction system. *https://www.autosigen.com/*, 2016.

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

[6] M. Amarlingam, Pradeep Kumar Mishra, K. V. V. Durga Prasad, and P. Rajalakshmi. Compressed sensing for different sensors: A real scenario for WSN and iot. In *3rd IEEE World Forum on Internet of Things, WF-IoT 2016, Reston, VA, USA, December 12-14, 2016*, pages 289–294. IEEE Computer Society, 2016.

[7] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9. IEEE, 2016.

[8] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9. IEEE, 2017.

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

[10] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, pages 464–475, 2003.

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

[12] Roland Dobbins. Mirai iot botnet description and ddos attack mitigation. *https://www.arbornetworks.com/blog/asert/mirai-iot-botnet-description-ddos-attack-mitigation/*, 2016.

[13] Anna C. Gilbert, Hung Q. Ngo, Ely Porat, Atri Rudra, and Martin J. Strauss. ;2/;2-foreach sparse recovery with low risk. In *ICALP (1)*, pages 461–472, 2013.

[14] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In Sharad Mehrotra and Timos K. Sellis, editors, *SIGMOD Conference*, pages 58–66. ACM, 2001.

[15] Kent Griffin, Scott Schneider, Xin Hu, and Tzi cker Chiueh. Automatic generation of string signatures for malware detection. In *RAID*, pages 101–120, 2009.

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

[17] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286. USENIX, 2004.

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

[19] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic-worms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, pages 32–47. IEEE Computer Society, 2006.

[20] Matthew V. Mahoney. Network traffic anomaly detection based on packet bytes. In *SAC*, pages 346–350. ACM, 2003.

[21] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. *PVLDB*, 5(12):1699, 2012.

[22] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In Thomas Eiter and Leonid Libkin, editors, *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[23] Jayadev Misra and David Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[24] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP*, pages 174–187, 2001.

[25] AS Navaz, V Sangeetha, and C Prabhadevi. Entropy based anomaly detection system to prevent ddos attacks in cloud. *arXiv preprint arXiv:1308.6745*, 2013.

[26] James Newsome, Brad Karp, and Dawn Xiaodong Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241. IEEE Computer Society, 2005.