Highly Efficient Techniques for Network Forensics

Miroslav Ponec
mip@cis.poly.edu

Paul Giura
pgiura@cis.poly.edu

Hervé Brönnimann
hbr@poly.edu

Joel Wein
wein@poly.edu

Department of Computer and Information Science
Polytechnic University, Brooklyn, New York

ABSTRACT

Given a history of packet transmissions and an excerpt of a possible packet payload, the *payload attribution problem* requires the identification of sources, destinations and the times of appearance on a network of all the packets that contained such payload. A module to solve this problem has recently been proposed as the core component in a network forensics system. Network forensics provides useful tools for investigating cybercrimes on the Internet, by, for example, tracing the spread of worms and viruses, identifying who has received a phishing email in an enterprise, or discovering which insider allowed an unauthorized disclosure of sensitive information.

In general it is infeasible to store and query the actual packets, therefore we focus on extremely compressed digests of the packet activity. We propose several new methods for payload attribution which utilize Rabin fingerprinting, shingling, and winnowing. Our best methods allow data reduction ratios greater than 100:1 while supporting queries with very low false positive rates, and provide efficient querying capabilities given reasonably small excerpts of a payload. Our results outperform current state-of-the-art methods both in terms of false positive rates and data reduction ratio. Finally, these approaches directly allow the collected data to be stored and queried by an untrusted party without disclosing any payload information nor the contents of queries.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Algorithms, Performance, Security

Keywords

Payload Attribution, Bloom Filter, Network Forensics

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

1. INTRODUCTION

Cybercrime today is alive and well on the Internet, and growing [11]. Given the trends of increasing Internet usage by individuals and companies alike, and the numerous opportunities for anonymity and non-accountability of Internet use, this trend should continue for some time. While there is much excellent work going on targeted at preventing cybercrime, unfortunately there is the parallel need to develop good tools to aid law-enforcement or corporate security professionals in investigating committed crimes. Identifying the sources and destinations of all packets that appeared on a network and contained a certain excerpt of a payload, a process called *payload attribution*, can be an extremely valuable tool in helping to determine the perpetrators or the victims of a network event and to analyze security incidents in general [18, 20, 8, 10].

It is possible to collect full packet traces even with commodity hardware [1] but the storage and analysis of terabytes of such data from today's high-speed networks is extremely cumbersome. Supporting network forensics by capturing and logging raw network traffic, however, is infeasible for anything but short periods of history. First, storage requirements limit the time over which the data can be archived (e.g., a 100Mbit/s WAN can fill up 1TB in just one day) and it is common practice to overwrite old data when that limit is reached. Second, string matching over such massive amounts of data is very time-consuming.

Recently Shanmugasundaram *et al.* [18] have presented an architecture for network forensics in which payload attribution is a key component [17]. They introduced the idea of using Bloom filters to achieve a reduced size digest of the packet history which would support queries about whether any packet containing a certain payload excerpt has been seen; the reduction in data representation comes at the price of a manageable false positive rate in the queries. Subsequently a different group has offered a variant technique for the same problem [7].

Our contribution in this paper is to present new methods for payload attribution which substantially improve over these state-of-the-art payload attribution systems. Our approach to payload attribution, which constitutes a crucial component of a network forensic system, can be easily integrated into any existing network monitoring system. The best of our methods allow data reduction ratios greater than 100:1 and achieve very low overall false positive rates. With a data reduction ratio 100:1 our best method gives no false positive answers for query excerpt sizes 250 bytes; in contrast, the prior best techniques had 100% false positive rate

at that data reduction ratio and excerpt size. The reduction in storage requirements makes it feasible to archive data taken over an extended time period and query for events in a substantially distant past. Our methods are capable of effectively querying for small excerpts of a payload but can also be extended to handle excerpts that span several packets. The accuracy of attribution increases with the length of the excerpt and the specificity of the query. Further, the collected payload digests can be stored and queries performed by an untrusted party without disclosing any payload information nor the query details.

This paper is organized as follows. In the next section we review related prior work. In Section 3 we provide a detailed design description of our payload attribution techniques, with a particular focus on payload processing and querying. In Section 4 we discuss several issues related to the implementation of these techniques in a full payload attribution system. In Section 5 we present a performance comparison of the proposed methods and quantitatively measure their effectiveness for multiple workloads. Finally, we summarize our conclusions in Section 6.

2. RELATED WORK

When processing a packet payload by the methods described in Section 3, the overall approach is to partition the payload into blocks and store them in a Bloom filter. In this section we first give a short description of Bloom filters and introduce Rabin fingerprinting and winnowing, which are techniques for block boundary selection. Thereafter we review the work related to payload attribution systems (PAS).

2.1 Bloom Filters

Bloom filters [3] are space-efficient probabilistic data structures supporting membership queries and are used in many network and other applications [6]. An empty Bloom filter is a bit vector of m bits, all set to 0, that uses k different hash functions, each of which maps a key value to one of the m positions in the vector. To insert an element into the Bloom filter, we compute the k hash function values and set the bits at the corresponding k positions to 1. To test whether an element was inserted, we hash the element with these k hash functions and check if all corresponding bits are set to 1, in which case we say the element is in the filter. The space savings of a Bloom filter is achieved at the cost of introducing false positives; the greater the savings, the greater the probability of a query returning a false positive. While we use Bloom filters, our payload attribution techniques can be easily modified to use any data structure which allows insertion and querying for strings with no changes to the structural design and implementation of the attribution methods.

2.2 Rabin Fingerprinting

Fingerprints are short checksums of strings with the property that the probability of two different objects having the same fingerprint is very small. Rabin [14] defined a fingerprinting scheme for binary strings based on polynomials. This scheme has found several applications [4], for example, to defining block boundaries for identifying similar files [12] and for web caching [15]. We derive a fingerprinting scheme for payload content based on Rabin's scheme in Section 3.3 and use it to pick content-dependent boundaries for a priori unknown substrings of a payload.

2.3 Winnowing

Winnowing [16] is an efficient fingerprinting algorithm enabling accurate detection of full and partial copies between documents. It works as follows: for each sequence of k consecutive characters in a document, we compute its hash value and store it in an array. Thus, the first item in the array is a hash of $c_1c_2 \ldots c_k$, the second item is a hash of $c_2c_3 \ldots c_{k+1}$, etc., where c_i are the characters in the document, for $i = 1, \ldots, n$. We then slide a window of size w through the array of hashes and select the minimum hash within each window. If there are more hashes with the minimum value, we choose the rightmost one. These selected hashes form the fingerprint of the document. They show that fingerprints selected by winnowing are better for document fingerprinting than the subset of Rabin fingerprints which contains hashes equal to 0 mod p, for some fixed p, because winnowing guarantees that in any window of size w there is at least one hash selected. We will use this idea to select boundaries for blocks in packet payloads in Section 3.5.

2.4 Attribution Systems

There has been a major research effort over the last several years to design and implement feasible network traffic traceback systems, which identify the machines that directly generated certain malicious traffic and the network path this traffic subsequently followed. However, these approaches restrict the queries to network floods, connection chains, or the entire payload of a single packet in the best case.

The *Source Path Isolation Engine* (SPIE) [19] is a hash-based technique for IP traceback that generates audit trails for traffic within a network. SPIE creates hash-digests of packets based on the packet header and a payload fragment and stores them in a Bloom filter in routers. SPIE uses these audit trails to trace the origin of any single packet delivered by the network in the recent past. The router creates a packet digest for every forwarded packet using the packet's non-mutable header fields and a short prefix of the payload, and stores it in a Bloom filter for a predefined time. Upon detection of a malicious attack by an intrusion detection system, SPIE can be used to trace the packet's attack path back to the source by querying SPIE devices along the path.

In many cases, an investigator may not have any header information about a packet of interest but may know some excerpt of the payload of the packets she wishes to see. Designing techniques for this problem that achieve significant data reduction is a much greater challenge; the entire packet payload is much larger than the information hashed by SPIE; in addition we need to store information about numerous substrings of the payload to support queries about excerpts. Shanmugasundaram *et al.* [17] introduced the *Hierarchical Bloom Filter* (HBF), a compact hash-based payload digesting data structure, which we describe in Section 3.1. A payload attribution system based on HBF is a key module for their distributed forensics network [18]. The system has both low memory footprint and achieves a reasonable processing speed at a low false positive rate. It monitors network traffic, creates hash-based digests of payload, and archives them periodically. A user-friendly query mechanism based on XML provides an interface to answer postmortem questions about network traffic. SPIE and HBF are both digesting schemes, but while SPIE is a packet digesting scheme, HBF is a payload digesting scheme. With

an HBF, one can query for substrings of the payload (called excerpts throughout this paper).

Recently, another group suggested an alternative approach to the payload attribution problem, the *Rolling Bloom Filter* (RBF) [7], which uses packet content fingerprints based on a generalization of the Rabin-Karp string-matching algorithm. Instead of aggregating queries in a hierarchy as an HBF, they aggregate query results linearly from multiple Bloom filters. They report performance similar to the best case performance of the HBF.

The design of an HBF is well documented in the literature and currently used in practice. We created our implementation of the HBF as an example of a current payload attribution method and include it in our comparisons in Section 5. The RBF's performance is comparable to that of the HBF and experimental results presented in [7] show that RBF achieves low false positive rates only for small data reduction ratios (about 32:1).

3. METHODS FOR PAYLOAD ATTRIBUTION

In this section we introduce various data structures for payload attribution. We first describe the basics of block based payload attribution and the Hierarchical Bloom Filter [17] as the current state-of-the-art method. We then propose several new methods which solve multiple problems in the design of the former methods. In Section 5 we give an evaluation of the properties of these attribution methods and an experimental performance comparison.

As noted earlier, all of these methods follow the general program of dividing packet payloads into blocks and inserting them into a Bloom filter. They differ in how the blocks are chosen, what methods we use to determine which blocks belong to which payload in which order ("consecutiveness resolution"), and miscellaneous other techniques used to improve the number of necessary queries and to reduce the probability of false positives.

A naive method to design a simple payload attribution system is to store the payload of all packets. In order to decrease the demand for storage capacity and to provide some privacy guarantees, we can store hashes of payloads instead of the actual payloads. This approach reduces the amount of data per packet to about 20 bytes (by using SHA-1, for example) at the cost of false positives due to hash collisions. By storing payloads in a Bloom filter (described in Section 2.1), we can further reduce the required space. The false positive rate of a Bloom filter depends on the data reduction ratio it provides. A Bloom filter preserves privacy because we can only ask whether a particular element was inserted into it, but it cannot be coerced into revealing the list of elements stored; even if we try to query for all possible elements, the result will be useless due to false positives. Compared to storing hashes directly, the advantage of using Bloom filters is not only the space savings but also the speed of querying. It takes only a constant time to query the Bloom filter for any packet.

The approaches above allow only querying for the whole payload. Instead of inserting the entire payload into the Bloom filter we can partition it into blocks and insert them individually. This simple modification can allow queries for excerpts of the payload by checking if all the blocks of an excerpt are in the Bloom filter. However, we need to determine whether two blocks appeared consecutively in the same payload, or if their presence is just an artifact of the blocking scheme. The methods presented in this chapter deal with this problem by using offset numbers or block overlaps. The simplest data structure that uses a Bloom filter and partitions payloads into blocks with offsets is a *Block-based Bloom Filter* (BBF) [17]. Note that, assuming that we do one decomposition of the payload into blocks, starting at the beginning of the packet, we will need to query this data structure for multiple starting positions of our excerpt in the payload, as it need not start at the beginning of a block.

We need to set two parameters which determine the time precision of our answers and the smallest query excerpt size. First, we want to be able to attribute to each excerpt for which we query the time when the packet containing it appeared on the network. We solve that by having multiple Bloom filters, one for each time interval. The duration of each interval depends on the number of blocks inserted into the Bloom filter. In order to guarantee an upper bound on the false positive rate, we replace the Bloom filter by a new one and store the previous Bloom filter in a permanent storage after a certain number of elements are inserted into it. There is also an upper bound on the maximum length of one interval to limit the roughness of time determination. Second, we specify the size of blocks. If the chosen block size is too small, we get too many collisions as there are not enough unique patterns and the Bloom filter gets filled quickly. If the block size is too large, there isn't enough granularity to answer queries for smaller excerpts.

We need to distinguish blocks from different packets to be able to answer who has sent/received the packet. The BBF as briefly described above isn't able to recognize whether two blocks occurred in two different packets. In order to work properly as an attribution system over multiple packets a unique packet identifier (*packetID*) must be associated with each block before inserting into the Bloom filter. A packet identifier can be the concatenation of source and destination IP addresses, optionally with source/destination port numbers. We maintain a list (or a more efficient data structure) of packetIDs for each Bloom filter and our data reduction estimates include the storage required for this list. This significantly increases the number of queries required for an attribution as the packetID of the packet that contained the query excerpt is not known a priori and we have to run queries for all packetIDs, which leads to higher false positive rate and decreases query performance. Therefore, we may either maintain two separate Bloom filters to answer queries, one with blocks only and one with blocks concatenated with corresponding packetIDs, or insert both into one larger Bloom filter. The former allows *data aging*, i.e. for very old data we can delete the first Bloom filter and store only the one with packetIDs at the cost of higher false positive rate and slower querying. Another method to save storage space is to take a Bloom filter of size $2b$ and replace it by a new Bloom filter of size b by computing the logical or operation of the two halves of the original Bloom filter. This halves the amount of data but the false positive rate increases significantly. We do not take packetIDs into account in method descriptions throughout this section for clarity.

We have identified several data structure properties of methods presented in this paper and a summary can be found in Table 1. These properties are thoroughly explained within the description of a method in which they appear first. Their impact on performance is discussed in Section 5.

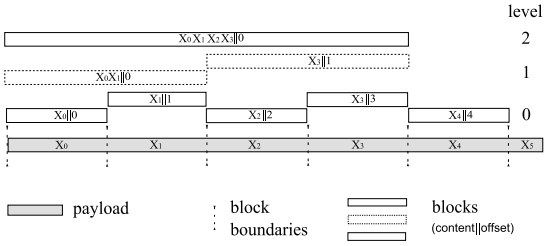


Figure 1: Processing of a payload consisting of blocks $X_0X_1X_2X_3X_4X_5$ in a Hierarchical Bloom Filter.

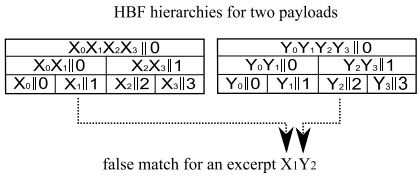


Figure 2: The hierarchy in the HBF doesn't cover double-blocks at odd offset numbers. In this example, we assume that two payloads $X_0X_1X_2X_3$ and $Y_0Y_1Y_2Y_3$ were processed by the HBF. If we query for an excerpt X_1Y_2, we would get a positive answer which represents an offset collision, because there were two blocks $(X_1||1)$ and $(Y_2||2)$ inserted from different payloads.

There are many possible combinations of the techniques presented in this paper and therefore the following list of methods is not complete due to space constraints. However, the selected subset provides enough details to construct the other methods that we have tested, for example, a method which builds a hierarchy of blocks with winnowing as a boundary selection technique, and also includes the best method, Winnowing Multi-Hashing.

In all the methods in this section we can extend the answer from yes/no to give details about which parts of the excerpt were found and return, for instance, the longest continuous part of the excerpt that was found.

3.1 Hierarchical Bloom Filter (HBF)

An HBF [17] supports queries for excerpts of a payload by dividing the payload of each packet into a set of blocks of size s bytes. The blocks of a payload of a single packet form a hierarchy (see Figure 1) which is inserted into a Bloom filter with appropriate offset numbers. Thus, besides inserting all blocks of a payload as in the BBF, we insert several super-blocks, i.e. blocks created by the concatenation of 2, 4, 8, etc., subsequent blocks into the HBF. This produces the same result as having multiple BBFs with block sizes multiplied by powers of two. A BBF can be looked upon as the base level of the hierarchy in an HBF.

When processing a payload, we start at the level 0 of the hierarchy by inserting all blocks of size s bytes. In the next level we double the size of a block and insert all blocks of size $2s$. In the n-th level we insert blocks of size $2^n s$ bytes. We continue until the block size exceeds the payload size. The total number of blocks inserted into an HBF for a payload of size p bytes is $\sum_l \lfloor p/(2^l s) \rfloor$, where l is the level index s.t. $0 \leq l \leq \lfloor log_2(p/s) \rfloor$. Therefore, an HBF needs about twice as much storage space compared to a BBF to achieve the same theoretical false positive rate of a Bloom filter, because the number of elements inserted into the Bloom filter is twice higher. However, for longer excerpts the hierarchy improves the confidence of the query results because they are assembled from the results for multiple levels.

We use one Bloom filter to store blocks from all levels of the hierarchy to improve space utilization because the number of blocks inserted into Bloom filters at different levels depends on the distribution of payload sizes and is therefore dynamic. The utilization of this single Bloom filter is easy to control by limiting the number of inserted elements, thus we can limit the (theoretical) false positive rate.

Offset numbers are the sequence numbers of blocks within the payload. Offsets are appended to block contents before insertion into an HBF: (content||offset), where $0 \leq$ offset $\leq \lfloor p/(2^l s) \rfloor - 1$, p is the size of the entire payload and l is the level of hierarchy. Offset numbers are unique within one level of the hierarchy. See the example given in Fig. 1. We first insert all blocks of size s with the appropriate offsets: $(X_0||0)$, $(X_1||1)$, $(X_2||2)$, $(X_3||3)$, $(X_4||4)$. Then we insert blocks at level 1 of the hierarchy: $(X_0X_1||0)$, $(X_2X_3||1)$. And finally the second level: $(X_0X_1X_2X_3||0)$.

Note that in Figure 1 blocks X_0 to X_4 have size s bytes, but since block X_5 has size smaller than s it doesn't form a block and its content is not being processed. We analyze the percentage of discarded payload content for each method in Section 5.

Offsets don't provide a reliable solution to the problem of detecting whether two blocks appeared in the same packet consecutively. For example, in a BBF if we process two packets made up of blocks $X_0X_1X_2X_3$ and $Y_0Y_1Y_2Y_3Y_4$ and query for an excerpt X_2Y_3, the BBF will answer it had seen a packet with a payload containing such an excerpt. We call this event an *offset collision*. This happens because of inserting a block X_2 with an offset 2 from the first packet and a block Y_3 with an offset 3 from the second packet into the BBF. When blocks from different packets are inserted at the appropriate offsets, a BBF can answer as if they occurred inside a single packet. An HBF reduces the false positive rate due to offset collisions and due to the inherent false positives of a Bloom filter by adding supplementary checks when querying for an excerpt composed of multiple blocks. In this example, an HBF would answer correctly because the check for X_2Y_3 in the next level of the hierarchy fails. However, if we query for an excerpt X_1Y_2, both HBF and BBF fail to answer correctly. We discuss offset collisions in an HBF again in Section 3.6. Because in the actual payload attribution system we insert blocks along with their packetIDs, collisions are less common, but they can still occur for payload inside one stream of packets within one time interval as these blocks have the same packetID.

Querying an HBF for an excerpt x starts with the same procedure as querying a BBF. First, we have to try all possible offsets, where x could have occurred inside one packet. We also have to try s possible starting positions of the first block inside x since the excerpt may not start exactly on a block boundary of the original payload. To do this, we slide a window of size s through the first s positions of x and query the HBF for this window (with all possible starting offsets). After a match is found for this first block, the query proceeds to try the next block at the next offset until all blocks of an excerpt at level 0 are matched. An HBF

	boundary selection	consecutiveness resolution	block size bounds	specials	possible false negatives*	possible N/A answers*
HBF	fixed	offsets	fixed block size	hierarchy	yes	no
FBS	fixed	shingling	fixed block size	-	yes	no
VBS	Rabin fingerprinting	shingling	no	-	no	yes
EVBS	enhanced Rabin fingerprinting	shingling	lower bound	-	yes	yes
WBS	winnowing	shingling	upper bound	-	no	yes**
VD	Rabin fingerprinting	doubles	no	doubles	no	yes
WMH	winnowing	shingling, multiple	upper bound	multi-hashing	no	yes**

** For query excerpts greater than the block size*
*** Only for query excerpts of size less than twice the block size*

Table 1: Summary of properties of methods from Section 3. It shows how each method selects boundaries of blocks when processing a payload and how it affects the block size, how each method resolves the consecutiveness of blocks, its special characteristics, and finally, whether each method allows false negative and N/A answers to excerpt queries.

continues by querying the next level for super-blocks of size twice the size of blocks in the previous level. Super-blocks start only at blocks from the previous level which have even offset numbers. We go up in the hierarchy until all queries for all levels succeed. The answer to an excerpt query is positive only if all answers from all levels of the hierarchy were positive. The maximum number of queries to a Bloom filter in an HBF in the worst case is roughly twice the number for a BBF.

3.2 Fixed Block Shingling (FBS)

In a BBF and an HBF we use offsets to determine whether blocks appeared consecutively inside one packet's payload. This causes a problem when querying for an excerpt because we don't know where the excerpt starts inside the payload (the starting offset is unknown). We have to try all possible starting offsets, which not only slows down the query process, but also increases the false positive rate because a false positive result may occur for any of these queries.

An alternative to using offsets can be block overlapping, which we call *shingling*. In this scheme, the payload of a packet is divided into blocks of size s bytes as in a BBF, but instead of inserting these blocks we insert strings of size $s + o$ bytes (the block plus a part of the next block) into the Bloom filter. Blocks overlap as do shingles on the roof (see Figure 3) and the overlapping part assures that it is likely that two blocks appeared consecutively if they share a common part and both of them are in the Bloom filter. For a payload of size p bytes, the number of elements inserted into the Bloom filter is $\lfloor p/s \rfloor$ for both the BBF and the FBS. The maximum number of queries to a Bloom filter in the worst case is about v times smaller than in a BBF, where v is the number of possible starting offsets.

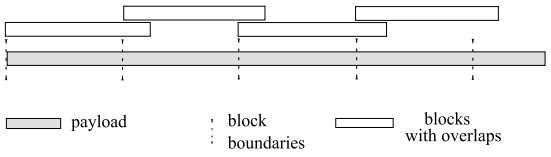


Figure 3: Processing of a payload with a Fixed Block Shingling (FBS) method.

The goal of the FBS scheme (of using an overlapping part) is to avoid trying all possible offsets during query processing in a HBF to solve the consecutiveness problem. However, both these techniques (shingling and offsets) are not guaranteed to answer correctly, for HBF because of offset collisions and for FBS because multiple blocks can start with the same string and FBS then confuses their position inside the payload (see Figure 4). Thus both can increase the number of false positive answers. For example, the FBS will incorrectly answer that it has seen a string of blocks $X_0X_1Y_1Y_2$ after processing two packets X and Y made of blocks $X_0X_1X_2X_3X_4$ and $Y_0Y_1Y_2$, respectively, where X_2 has the same prefix (of size at least o bytes) as Y_1.

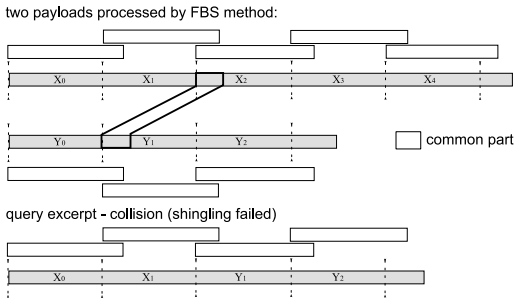


Figure 4: An example of a collision due to a shingling failure. The same prefix prevented a FBS method from determining whether two blocks appeared consecutively within one payload. The FBS method incorrectly treats the string of blocks $X_0X_1Y_1Y_2$ as if it was processed inside one payload.

Querying a Bloom filter in the FBS scheme is similar to querying a BBF except that we do not use any offsets and therefore don't have to try all possible offset positions of the first block of an excerpt. When querying for an excerpt x we slide a window of size $s + o$ bytes through the first s positions of x and query the Bloom filter for this window. When a match is found for this first block, the query can proceed with the next block (including the overlap) until all blocks of an excerpt are matched. Since these blocks overlap we assume that they occurred consecutively inside one single payload. The answer to an excerpt query is considered to be positive only if there exists an alignment (i.e. a position of the first block's boundary) for which all tested blocks were found in the Bloom filter.

3.3 Variable Block Shingling (VBS)

The use of shingling instead of offsets in a FBS method lets us avoid testing all possible offset numbers of the first block during querying, but we still have to test all possible

alignments of the first block inside an excerpt. A *Variable Block Shingling* (VBS) solves this problem by setting block boundaries based on the payload itself.

We slide a window of size k bytes through the whole payload and for each position of the window we compute a value of function $H(c_1, \ldots, c_k)$ on the byte values of the payload. When $H(c_1, \ldots, c_k) \bmod m$ is equal to zero, we insert a block boundary immediately after the current position of byte c_k. Note that we can choose to put a block boundary before or after any of the bytes c_i, $1 \leq i \leq k$, but this selection has to be fixed. For a function H random and uniform the parameter m sets the expected size of a block. For random payloads we will get a distribution of block sizes with an average size of m bytes. This variable block size technique's drawback is that we can get many very small blocks, which can flood the Bloom filter, or some large blocks, which prevent us from querying for smaller excerpts. Therefore, we introduce an enhanced version of this scheme, EVBS, in the next section.

In order to save computational resources it is convenient to use a function that can use the computation done for the previous position of the window to calculate a new value as we move from bytes $c_1, \ldots, c_k$ to $c_2, \ldots, c_{k+1}$. Rabin fingerprints (see Section 2.2) have such iterative property and we define a fingerprint F of a substring $c_1 c_2 \ldots c_k$, where c_i is the value of the i-th byte of the substring of a payload, as:

$$F(c_1, \ldots, c_k) = (c_1 p^{k-1} + c_2 p^{k-2} + \cdots + c_k) \bmod M, \quad (1)$$

where p is a fixed prime number and M is a constant. To compute the fingerprint of substring $c_2 \ldots c_{k+1}$, we need only to add the last element and remove the first one:

$$F(c_2, \ldots, c_{k+1}) = (p\ F(c_1, \ldots, c_k) + c_{k+1} - c_1 p^k) \bmod M. \quad (2)$$

Because p and k are fixed we can precompute the values for p^{k-1}. It is also possible to use Rabin fingerprints as hash functions in the Bloom filter. In our implementation we use a modified scheme [5] to increase randomness without any additional computational costs:

$$F(c_2, \ldots, c_{k+1}) = (p(F(c_1, \ldots, c_k) + c_{k+1} - c_1 p^k)) \bmod M. \quad (3)$$

The advantage of picking block boundaries using Rabin functions is that when we get an excerpt of a payload and divide it into blocks using the same Rabin function that we used for splitting during the processing of the payload, we will get exactly the same blocks. Thus, we don't have to try all possible alignments of the first block as in previous methods.

The rest of this method is similar to the FBS scheme where instead of using fixed-size blocks we have variable-size blocks depending on the payload. To process a payload we slide a window of size k bytes through the whole payload. For each its position we check whether the value of F modulo m is zero and if yes we set a new block boundary. All blocks are inserted with the overlap of o bytes as shown in Figure 5.

Querying in a VBS method is the simplest of all methods in this section because there are no offsets and no alignment problems. Therefore, this method involves much fewer tests for membership in a Bloom filter. Querying for an excerpt is done in the same way as processing the payload in previous paragraph but instead of the insertion we query the Bloom filter. When all blocks are found in the Bloom filter the answer is positive. The maximum number of queries to a Bloom filter in the worst case is about $v \cdot s$ times smaller than in a BBF, where v is the number of possible starting offsets and s is the number of possible alignments of the first block in a BBF, while assuming the average block size in a VBS method to be s.

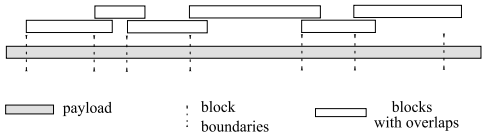


Figure 5: Processing of a payload with a Variable Block Shingling (VBS) method.

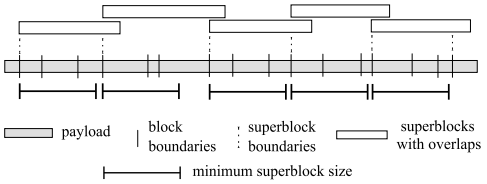


Figure 6: Processing of a payload with an Enhanced Variable Block Shingling (EVBS) method.

3.4 Enhanced Variable Block Shingling (EVBS)

The enhanced version of the variable block shingling method tries to solve a problem with block sizes. A VBS can create many small blocks, which can flood the Bloom filter and don't provide enough discriminability, or some large blocks, which can prevent querying for smaller excerpts. In EVBS we form superblocks composed from blocks found by a VBS method to achieve better control over the size of blocks.

To be precise, when processing a payload we slide a window of size k bytes through the entire payload and for each position of the window we compute the value of the fingerprinting function $H(c_1, \ldots, c_k)$ on the byte values of the payload as in the VBS method. When $H(c_1, \ldots, c_k) \bmod m$ is equal to zero, we insert a block boundary after the current position of byte c_k. We take the resulting blocks of an expected size m bytes, one by one from the start of the payload, and form superblocks, i.e. new non-overlapping blocks made of multiple original blocks, with the size at least m' bytes, where $m' \geq m$. We do this by selecting some of the original block boundaries to be the boundaries of new superblocks. Every boundary that creates a superblock of size greater or equal to m' is selected (Figure 6). Finally, superblocks with an overlap to the next superblock of size o bytes are inserted into the Bloom filter. The maximum number of queries to a Bloom filter in the worst case is about the same as for a VBS, assuming the average block sizes for the two methods are the same.

However, this leads to a problem when querying for an excerpt. If we use the same fingerprinting function H and parameter m we get the same block boundaries in the excerpt as in the original payload, but the starting boundary of the first superblock inside the excerpt is unknown. Therefore, we have to try all boundaries in the first m' bytes of an excerpt (or the first that follows if there was none) to form the first boundary of the first superblock. The number of possible boundaries (approximately m'/m) in a EVBS method is much smaller than the number of possible alignments (i.e. the block size s) in an HBF, for usual parameter values.

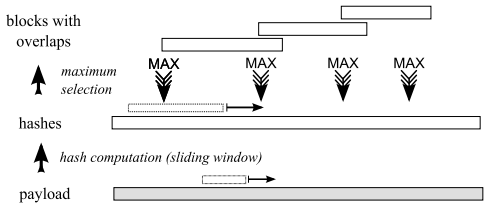


Figure 7: Processing of a payload with a Winnowing Block Shingling (WBS) method. First, we compute hash values for each payload byte position. Subsequently boundaries are selected to be at the positions of the rightmost maximum hash value inside the winnowing window which we slide through the array of hashes. Bytes between consecutive pairs of boundaries form blocks (plus the overlap).

3.5 Winnowing Block Shingling (WBS)

In a *Winnowing Block Shingling* method we use the idea of winnowing, described in Section 2.3, to select boundaries of blocks and shingling to resolve the consecutiveness of blocks. We select the winnowing window size instead of a block size and we are guaranteed to have at least one boundary in any window of this size inside the payload. This also sets an upper bound on the block size.

We start by computing hash values for each payload byte position. In our implementation this is done by sliding a window of size k bytes through the whole payload and for each position of the window we compute the value of a fingerprinting function $H(c_1, \ldots, c_k)$ on the byte values of the payload as in the VBS method. In this way we get an array of hashes, where the i-th element is the hash of bytes $c_i, \ldots, c_{i+k-1}$, where c_i is the i-th byte of the payload of size p, for $i = 1, \ldots, (p-k+1)$. Then we slide a winnowing window of size w through this array and for each position of the winnowing window we put a boundary immediately before the position of the maximum hash value within this window. If there are more hashes with maximum value we choose the rightmost one. Bytes between consecutive pairs of boundaries form blocks (plus the beginning of size o of the next block, the overlap) and they are inserted into a Bloom filter. See Figure 7.

When querying for an excerpt we do the same process except that we query the Bloom filter for blocks instead of inserting them. If all were found in the Bloom filter the answer to the query is positive. The maximum number of queries to a Bloom filter in the worst case is about the same as for a VBS, assuming the average block sizes for the two methods are the same.

There is at least one block boundary in any window of size w. Therefore the longest possible block size is $w+1+s$ bytes. This also guarantees that there are always at least two boundaries to form a block in an excerpt of size at least $2w+s$ bytes.

3.6 Variable Doubles (VD)

This method is a response to a problem with a hierarchy in an HBF: the hierarchy is not complete in a sense that we do not insert all double-blocks (blocks of size $2s$) and all quadruple-blocks ($4s$), and so on, into the Bloom filter. For example, when inserting a packet consisting of blocks $S_0S_1S_2S_3S_4$ into an HBF, we insert blocks S_0,..., S_4, S_0S_1, S_2S_3, and $S_0S_1S_2S_3$. And if we query for an excerpt of size

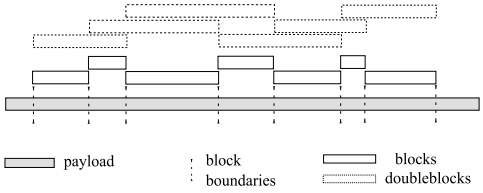


Figure 8: Processing of a payload with a Variable Doubles (VD) method.

$2s$ (or up to size $4s-2$ bytes), for example, S_1S_2, this block of size $2s$ is not found in the HBF, as is true of all other double-blocks at odd offsets. The same is true for other levels of the hierarchy and the probability that this occurs rises exponentially with the level. As an alternative approach to the hierarchy we insert all double-blocks as shown in Figure 8, but do not continue to the next level to not increase the storage requirements. Thus, in a VD method we insert all single blocks and all double-blocks composed of all adjacent single blocks.

The block boundaries are determined by a fingerprinting function as in a VBS (Section 3.3). The number of blocks inserted into the Bloom filter as well as the maximum number of queries in the worst case is approximately twice the number for a VBS scheme. An example of payload processing in a VD method is given in Figure 8.

Note that in this method we neither use shingling nor offsets, because the consecutiveness problem is solved by the level of all double-blocks which overlap with each other.

The query mechanism is as follows. We divide the excerpt into blocks by a fingerprinting method and query the Bloom filter for all blocks and for all double-blocks. Finally, if all answers are positive we claim that the excerpt is found.

3.7 Winnowing Multi-Hashing (WMH)

The WMH method uses multiple instances of WBS (Section 3.5) to reduce the probability of false positives for excerpt queries. The WMH gives not only excellent control over the block sizes due to winnowing (see Figure 10(c)) but also provides much greater confidence about the consecutiveness of the blocks inside the query excerpt because of overlaps both inside each instance of WBS and among the blocks of multiple instances. Both querying and payload processing are done for all t WBS instances and the final answer to an excerpt query is positive only if all t answers are positive.

In our experiments in Section 5 we use two instances of WBS with identical winnowing window size and store data from both methods in one Bloom filter. By storing data of each instance in a separate Bloom filter we can allow data aging to save space by keeping only some of the Bloom filters for very old data at the cost of higher false positive rates.

4. PAYLOAD ATTRIBUTION SYSTEMS (PAS)

A payload attribution system performs two separate tasks: payload processing and query processing. In payload processing, the payload of all traffic that passed through the network where the PAS is deployed is examined and some information is saved into permanent storage. This has to be done at line speed and the underlying raw packet capture component can also perform some filtering of the packets, for example, choosing to process only HTTP traffic.

Data is stored in archive units, each of which having two timestamps (start and end of the time interval in which we collected the data). For each time interval we also need to save all packetIDs (e.g., pairs of source and destination IP addresses) to allow querying later on. This information can be alternatively obtained from connection records collected by firewalls, intrusion detection systems or other log files.

During query processing given the excerpt and a time interval we first have to retrieve all corresponding archive units. We query each unit for the excerpt and if we get positive answer we try to query successively for each of the packetIDs appended to the blocks of the excerpt and report all matches.

4.1 Attacks on PAS

As with any security system, there are ways an adversary can evade proper attribution. We identify the following types of attacks on a PAS (mostly similar to those in [17]):

(a) Fragmentation. An attacker can transform the stream of data into a sequence of packets with payload sizes much smaller than the (average) block size we use in the PAS. Methods with variable block sizes where block boundaries depend on the payload are harder to beat, but, for very small fragments, e.g. 6 bytes each, the system won't be able to do the attribution correctly. A solution is to make the PAS stateful so that it concatenates payloads of one data stream prior to processing. However, such a solution would impose additional memory and computational costs and there are known attacks on stateful IDS systems [9], such as incorrect fragmentation and timing attacks.

(b) Boundary Selection Hacks. For methods with block boundaries depending on the payload an attacker can try to send special packets containing payload that can contain too many or no boundaries. The PAS can use different parameters for boundary selection algorithm for each archive unit so that it would be impossible for an attacker to fool the system. Moreover, winnowing guarantees at least one boundary in each winnowing window.

(c) Hash Collisions. Hash collisions are very unpredictable and therefore hard to use by an attacker because we use different salt for the hash computation in each Bloom filter.

(d) Stuffing. An attacker can inject some characters into the payload which are ignored by applications but in the network layer they change the payload structure. Our methods are robust against stuffing because the attacker has to modify most of the payload to avoid correct attribution as we can match even very small excerpts of payload.

(e) Resource Exhaustion. Flooding attacks can impair a PAS. However, our methods are more robust to these attacks than raw packet loggers due to the data reduction they provide. Moreover, processing identical payloads repeatedly doesn't impact the precision of attribution because the insertion into a Bloom filter is an idempotent operation. On the other hand, the list of packetIDs is vulnerable to flooding, for example, when a worm tries to propagate out of the network by trying many random destination addresses.

(f) Spoofing. Source IP addresses can be spoofed and a PAS is primarily concerned with attributing payload according to what packets have been delivered by the network. Local source attribution can be accurate up to the local subnet that a PAS is deployed in. Moreover, for connection oriented sessions it is difficult to spoof source IP addresses.

(g) Compression&Encryption. If the payload is compressed or encrypted, a PAS can allow to query only for the exact compressed or encrypted form.

4.2 Multi-packet queries

The methods described in Section 3 describe how to query for excerpts inside one packet's payload. Nevertheless, we can extend the queries to strings that span multiple packets. Methods which use offsets have to continue querying for the next block which was not found with its sequential offset number with a zero offset instead. Methods that use shingling can be extended without any further changes if we return as an answer to the query the full sequence of blocks found.

4.3 Privacy

Processing and archiving payload information must comply with the privacy and security policies of the network where they are performed. Furthermore, authorization to use the payload attribution system should be granted only to properly authorized parties and all necessary precautions must be taken to minimize the possibility of a system compromise.

Our methods allow the collected data to be stored and queried by an untrusted party without disclosing any payload information nor giving the query engine any knowledge of the contents of queries. We achieve this by adding a secret salt when computing hashes for insertion and querying the Bloom filter. A different salt is used for each Bloom filter and serves the purpose of a secret key. Without this key the Bloom filter cannot be queried and the key doesn't have to be made available to the querier (only the indices of bits for which we want to query are disclosed). Without knowing the key a third party cannot query the Bloom filter. However, additional measures must be taken to enforce the third party to provide correct answers and to not alter the archived data. A detailed analysis of privacy achieved by using Bloom filters can be found in [2].

4.4 Compression

In addition to the inherent data reduction provided by our attribution methods due to the use of Bloom filters, our experiments show that we can achieve another about 20 percent storage savings by compressing the archived data (after careful optimization of parameters [13]), for example by gzip. The results presented in the next section don't include this additional compression.

5. EXPERIMENTAL RESULTS

In this section we show performance measurements of payload attribution methods described in Section 3 and discuss the results from various perspectives. For this purpose we collected a network trace of 4 GB of HTTP traffic from our campus network. For performance evaluation throughout this section we consider processing 3.1 MB segment (about 5000 packets) of the trace as one unit trace collected during one time interval. The results presented do not depend on the size of the unit because we use a data reduction ratio to set the Bloom filter size (e.g., 100:1 means a Bloom filter of size 31kB). Each compared method uses one Bloom filter of an equal size to store all data. Our results didn't show any deviations depending on the selection of the segment within

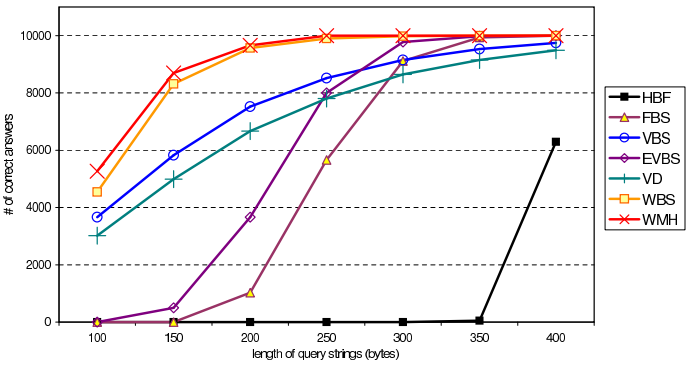


Figure 9: The graph shows the number of correct answers to 10000 excerpt queries for varying length of a query excerpt for each method (with block size or winnowing window size 64 bytes) and data reduction ratio 100:1. This reduction ratio can be improved to 120:1 by compressing the archived data by gzip.

the trace. All methods were tested to select the best combination of parameters for each of them. Results are grouped into subsections by different points of interest.

5.1 Block Size Distribution

The graphs in Figure 10 show the distributions of block sizes for three different methods of block boundary selection. We use a block (or winnowing window) size parameter of 32 bytes, a small block size for an EVBS of 8 bytes, and an overlap of 4 bytes. Both VBS and EVBS show a distribution with an exponential decrease in the number of blocks with an increasing block size, shifted by the overlap size for a VBS or the block size plus the overlap size for an EVBS. Long tails were cropped for clarity and the longest block was 1029 bytes long.

On the other hand, a winnowing method results in quite uniform distribution where the block sizes are bounded by the winnowing window size plus the overlap. The apparent peaks for the smallest block size in graphs 10(a) and 10(c) are caused by low-entropy payloads, such as long blocks of zeros. The distributions of block sizes obtained by processing random payloads generated with the same payload sizes as in the original trace show the same distributions just without these peaks. Nevertheless, the huge amount of small blocks doesn't significantly affect the attribution because inserting a block into the Bloom filter is an idempotent operation.

5.2 Unprocessed Payload

Some fraction of each packet's payload is not processed by the attribution mechanisms presented in Section 3. Figure 10(d) shows how each boundary selection method affects the percentage of unprocessed payload. For methods with a fixed block size the part of a payload between the last block's boundary and the end of the payload is ignored by the payload attribution system. With (enhanced) Rabin fingerprinting, and winnowing methods the part starting at the beginning of the payload and ending at the first block boundary, and the part between the last block boundary and the end of the payload are not processed. The enhanced version of Rabin fingerprinting achieves much better results because the small block size, which was four times smaller that the superblock size in our test, applies when selecting the first block boundary in a payload.

Winnowing performs better than other methods with a variable block size in terms of unprocessed payload. Note also that a WMH method, even though it uses winnowing for block boundary selection as well as a WBS does, has about t times smaller percentage of unprocessed payload than a WBS because each of the t instances of a WBS within the WMH covers a different part of the payload independently. Moreover, the "inner" part of the payload is covered t times which makes the method more resistant to collisions because t collisions have to occur at the same time to produce a false positive answer.

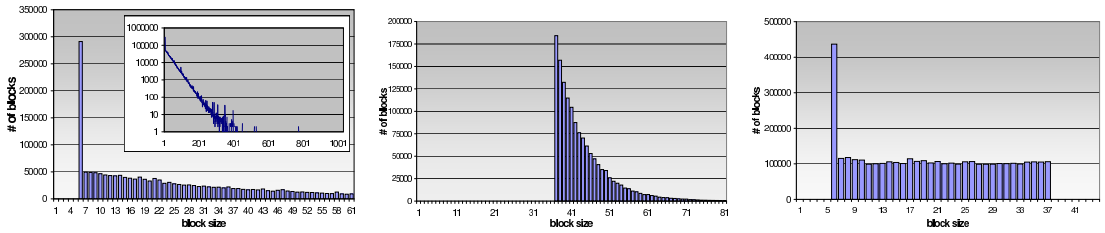
For large packets the small percentage of unprocessed payload doesn't pose a problem, however, for very small packets, e.g., only 6 bytes long, it means that they are possibly not processed at all. Therefore we can optionally insert the entire payload of a packet in addition to inserting all blocks and add a special query type to the system to support queries for exact packets. This will increase the storage requirements only slightly because we would insert one additional element per packet into the Bloom filter.

5.3 Query Answers

To measure and compare the performance of attribution methods, and in particular to analyze the false positive rate, we processed the trace by each method and queried for random strings which included a small excerpt of size 8 bytes in the middle that did not occur in the trace. In this way we made sure that these query strings did not represent payloads inserted into the Bloom filters. Every method has to answer either YES, NO or answer not available (N/A) to each query. A YES answer means a match was found for the entire query string for at least one of the packetIDs and represents a false positive. A NO answer is the correct answer for the query, and an N/A answer is returned if the blocking mechanism specific to the method did not select even one block inside the query excerpt. The N/A answer can occur, for example, when the query excerpt is smaller than the block size in an HBF, or when there was one or no boundary selected in the excerpt by a VBS method and so there was no block to query for.

In Table 1 we summarize the possibility of getting N/A and false negative answers for each method. A false negative answer can occur when we query for an excerpt which has size greater than the block size but none of the alignments of blocks inside the excerpt fit the alignment that has been used to process the payload which contained the excerpt. For example, if we processed a payload $ABCDEFGHI$ by an HBF with block size 4 bytes, we would have blocks $ABCD$, $EFGH$, $ABCDEFGH$, and if we queried for an excerpt $BCDEFG$, the HBF would answer NO. Note that false negatives can occur only for excerpts smaller than twice the block size and only for methods which involve testing the alignment of blocks.

Table 2(b) provides detailed results of 10000 excerpt queries for all methods with the same storage capacity and data reduction ratio 50:1. The WMH method achieves the best results among the listed methods for all excerpt sizes and for excerpts longer than 200 bytes has no false positives. The WMH also excels in the number of N/A answers among the methods with a variable block size because it guarantees at



(a) (b) (c)

boundary selection method	unprocessed payload
fixed	1.52%
Rabin fingerprinting	8.04%
enhanced Rabin f.	4.49%
winnowing	4.26%

(d)

Figure 10: The distributions of block sizes for three different methods of block boundary selection after processing 100000 packets of HTTP traffic are shown: (a) VBS (the inner graph shows an exponential decrease in log scale), (b) EVBS, (c) WBS. Table (d) shows the percentage of unprocessed payload of 50000 packets depending on the block boundary selection method used. Details are provided in Sections 5.1 and 5.2.

(a)

query string length	150	200	250	300	350	400	450	500
# of answers								
YES	2379	861	340	127	50	18	5	0
NO	7621	9139	9660	9873	9950	9982	9995	10000
N/A	0	0	0	0	0	0	0	0
false positive rate	0.2379	0.0861	0.034	0.0127	0.005	0.0018	0.0005	0

(b)

length	70 Bytes			100 Bytes			120 Bytes			150 Bytes			200 Bytes			250 Bytes		
answer	YES	NO	N/A	YES	NO	N/A	YES	NO	N/A	YES	NO	N/A	YES	NO	N/A	YES	NO	N/A
HBF	10000	0	0	10000	0	0	10000	0	0	10000	0	0	3384	6616	0	117	9883	0
FBS	10000	0	0	9794	206	0	8874	1126	0	4906	5094	0	338	9662	0	20	9980	0
VBS	473	4973	4554	412	7233	2355	370	8156	1474	260	9046	694	88	9733	179	37	9920	43
EVBS	9210	0	790	6063	3924	13	3036	6962	2	676	9324	0	32	9968	0	1	9999	0
WBS	2118	7683	199	488	9512	0	137	9863	0	24	9976	0	0	10000	0	0	10000	0
VD	1508	4291	4201	1445	6416	2139	1181	7484	1335	834	8539	627	413	9431	156	146	9815	39
WMH	1974	8022	0	377	9623	0	130	9870	0	22	9978	0	0	10000	0	0	10000	0

Table 2: (a) False positive rates for data reduction ratio 130:1 for a WMH method with a winnowing window size 64 bytes and therefore an average block size about 32 bytes. The table summarizes answers to 10000 queries for each query excerpt size. All 10000 answers should be NO. YES answers are due to false positives inherent to a Bloom filter. WMH guarantees no N/A answers for these excerpt sizes. (b) Measurements of false positive rate for data reduction ratio 50:1. The table summarizes answers to 10000 excerpt queries using all methods (with block size 32 bytes) described in Section 3 for various query excerpt lengths (top row). These queries were performed after processing a real packet trace and all methods use the same size of a Bloom filter (50 times smaller than the trace size). *All 10000 answers should be NO* since these excerpts were not present in the trace. YES answers are due to false positives in Bloom filter and N/A answers mean that there were no boundaries selected inside the excerpt to form a block for which we can query.

least one block boundary in each winnowing window. The results also show that methods with a variable block size are in general better than methods with a fixed block size because there are no problems with finding the right alignment. The enhanced version of Rabin fingerprinting for the block boundary selection doesn't perform better than the original version. This is mostly because we need to try all alignments of blocks inside superblocks when querying which increases the false positive rate.

Graph 9 shows the number of correct answers to 10000 excerpt queries as a function of the length of a query excerpt for each method with block size parameter set to 64 bytes and data reduction ratio 100:1. The WMH method outperforms all other methods for all excerpt lengths. On the other hand, the HBF's results are the worst because it can fully utilize the hierarchy only for long excerpts and its has very high false positive rate for high data reduction ratios due to the use of offsets, problems with block alignments, and the large number of elements inserted into the Bloom filter. For very long excerpts all the methods provide highly reliable results.

The Winnowing Multi-Hashing achieves the best overall performance in all our tests and allows us to query for very small excerpts because the average block size is approximately half of the winnowing window size plus the overlap size. The average block size was 18.9 bytes for a winnowing window size 32 bytes and an overlap 4 bytes. Table 2(a) shows false positive rates for WMH for data reduction ratio 130:1. The data reduction ratio means that the total size of a processed payload was 130-times the size of the Bloom filter which is archived to allow querying. The Bloom filter could be additionally compressed by gzip to achieve final compression of 158:1.

6. CONCLUSION

In this paper, we presented several new methods for payload attribution. When incorporated into a network forensics system they provide an efficient probabilistic query mechanism to answer queries for excerpts of a payload that passed through the network. Our methods allow data reduction ratios greater than 100:1 while having a very low false positive rate. They allow queries for very small excerpts of a payload and also for excerpts that span multiple packets. The experimental results show that our methods represent a significant improvement in performance compared to previous attribution techniques. The results also testify that the accuracy of attribution increases with the length and the specificity of a query. Moreover, privacy is achieved by one-way hashing with a secret key in a Bloom filter. Thus, even if the system is compromised no raw traffic data is ever exposed and querying the system is possible only with the knowledge of the secret key.

7. ACKNOWLEDGEMENTS

We thank Torsten Suel, Kulesh Shanmugasundaram and the anonymous reviewers for their helpful suggestions. This research is supported by NSF CyberTrust Grant 0430444.

8. REFERENCES

[1] E. Anderson and M. Arlitt. Full packet capture and offline analysis on 1 and 10 gb networks. Technical Report HPL-2006-156, 2006.

[2] S. Bellovin and W. Cheswick. Privacy-enhanced searches using encrypted Bloom filters. Cryptology ePrint Archive, Report 2004/022, 2004. Available at http://eprint.iacr.org/.

[3] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. In *CACM*, pages 422–426, 1970.

[4] A. Broder. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[5] A. Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[6] A. Broder and M. Mitzenmatcher. Network Applications of Bloom Filters: A Survey. In *Annual Allerton Conference on Communication, Control, and Computing*, Urbana-Champaign, Illinois, USA, October 2002.

[7] C. Y. Cho, S. Y. Lee, C. P. Tan, and Y. T. Tan. Network forensics on packet fingerprints. In *21st IFIP Information Security Conference (SEC 2006)*, Karlstad, Sweden, 2006.

[8] S. Garfinkel. Network forensics: Tapping the internet. O'Reilly Network, 2002. Available at http://www.oreillynet.com/pub/a/network/2002/04/26/nettap.html.

[9] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the USENIX Security Symposium*, Washington, USA, 2001.

[10] N. King and E. Weiss. Network Forensics Analysis Tools (NFATs) reveal insecurities, turn sysadmins into system detectives. Information Security, Feb. 2002. Available at www.infosecuritymag.com/2002/feb/cover.shtml.

[11] A. Locke. Red tape chronicles: Phishing gets much more profitable. MSNBC.com, November 9, 2006. Available at http://redtape.msnbc.com/2006/11/phishing_gets_m.html.

[12] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 1994.

[13] M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604 – 612, 2002.

[14] M. O. Rabin. Fingerprinting by random polynomials. Technical report 15-81, Harvard University, 1981.

[15] S. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proceedings of the Twelfth International World Wide Web Conference*, May 2003.

[16] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, New York, NY, USA, 2003. ACM Press.

[17] K. Shanmugasundaram, H. Brönnimann, and N. Memon. Payload Attribution via Hierarchical Bloom Filters. In *Proc. of ACM CCS*, 2004.

[18] K. Shanmugasundaram, N. Memon, A. Savant, and H. Brönnimann. ForNet: A Distributed Forensics Network. In *Proc. of MMM-ACNS Workshop*, pages 1–16, 2003.

[19] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP traceback. In *ACM SIGCOMM*, San Diego, California, USA, August 2001.

[20] S. Staniford-Chen and L.T. Heberlein. Holding intruders accountable on the internet. Oakland, 1995. Proceedings of the 1995 IEEE Symposium on Security and Privacy.