The 4th IEEE LCN Workshop on Network Measurements (WNM 2009)
Zürich, Switzerland; 20-23 October 2009

Hash Tables for Efficient Flow Monitoring: Vulnerabilities and Countermeasures

David Eckhoff, Tobias Limmer and Falko Dressler
Computer Networks and Communication Systems, University of Erlangen, Germany
{eckhoff, limmer, dressler}@cs.fau.de

***Abstract*—Aggregation modules within flow-based network monitoring tools make use of fast lookup methods to be able to quickly assign received packets to their corresponding flows. In software-based aggregators, hash tables are usually used for this task, as these offer constant lookup times under optimal conditions. The hash functions used for mapping flow keys to hash values need to be chosen carefully to ensure optimal utilization of the hash table. If attackers would be able to create collisions, the hash table degenerates to linked lists with worst-case lookup times of $O(n)$ and greatly reduces the performance of the aggregation modules. Thus, independent of the available computational power of the monitor, an attacker would easily be able to overload the system. In this report, we analyze the aggregation modules of the software-based flow meters Vermont and nProbe. We evaluate the resilience strength of used hash functions by theoretical analysis and confirm the results by performing real attacks. These attacks show how easily flow monitors can be overloaded if the hash algorithm has not been chosen carefully. Based on our observations, we finally present a hash function which we believe has none of the weaknesses we have discovered.**

***Index Terms*—targeted attacks, flow monitoring, denial of service, hash collision**

I. Introduction

A crucial function for a high-speed flow-based network monitors is to use efficient data structures for packet processing [1], [2]. Therefore, flow monitoring tools like Vermont [3] and nProbe [4] use hash tables for internal flow processing. A flow record is the aggregated representation of all packets that share common properties. These attributes are typically the source and destination IP addresses, the source and destination ports, and the used protocol. In the IP Flow Information Export (IPFIX) standard [5], [6], which supersedes the Netflow.v9 architecture [7], these attributes are referred to as flow keys.

The advantage of using hash tables is the constant lookup time $O(1)$, i.e. for each received packet, it takes constant time to determine whether the packet belongs to an already existing flow, or if a new flow needs to be created. Hash tables usually have a constant size h_{size} and consist of so-called hash buckets. A hash function $k = h(x)$ maps values to discrete integer values in the range $k \in [0, h_{\text{size}})$. However, due to the limited size of such a hash table, it is inevitable that two different flows are mapped to the same hash bucket sooner or later. One possible way to handle such a collision is that each bucket of the hash table is able to store multiple flows in form of a linked list. This list maintains all the concurrent flows that match the same hash key. The disadvantage of this method is that it takes an additional linear lookup time $O(n)$ to find an entry in this unsorted list. Therefore, it is essential to keep the length of these lists as small as possible. Otherwise, the system might become overloaded and it will not be able to process all the received packets. Dropped packets imply dropped information and, thus, attacks or anomalies may remain undetected [8].

If the hash function used for calculating the hash keys offers a uniform distribution, the length of all bucket lists would be almost equal. Thus, a reasonably sized hash data structure should perform well, i.e. flow lookup is fast because of minimal list lengths.

A typical Distributed Denial of Service (DDoS) attack sends huge amounts of data packets, e.g. TCP packets with the SYN flag set, to a victim host or subnet in order to deplete resources at the receiving end. Assuming that the source address is used as a flow key, the use of randomized spoofed source IP addresses would cause a network monitor observing those packets to create a new flow for each attack packet. If the spoofed address is completely random and if no countermeasures are taken, such an attack will uniformly fill the bucket lists in the hash table until the lists become too long to be efficiently scanned and the flow monitor will begin to drop packets. The bigger the hash table is, the more packets are needed to create such long collision lists and it is more likely that the hash table does not represent a bottleneck.

However, if an attacker is able to systematically create packets that lead to collisions in the monitor's hash table, it would require much less effort, i.e. less packets, to bring the monitor into such an overloaded state. The attacker will try to create new flow records that would be inserted in very few, or even a single bucket inside the hash table. Then the monitor will spend most of the time scanning these extraordinarily long bucket lists, and so its performance will decrease and a considerable amount of packets will be dropped.

A hash function should therefore have the following characteristics:

- The computed hash keys should be uniformly distributed
- It should be impossible for an attacker to create directed collisions
- The hash function must be fast so that it does not become a bottleneck

In this report, we study the hash functions used in the two frequently used flow monitors Vermont and nProbe (Section II). We identify possible vulnerabilities that can be exploited for directed attacks against the flow monitor. Vermont

978-1-4244-4487-8/09/$25.00 ©2009 IEEE

uses CRC to hash flow keys, so we specifically analyze this hash function for vulnerabilities (Section III). Furthermore, we demonstrate the feasibility of such attacks and present selected measurement results (Section IV). According to our findings, we present a hash function, which meets the three requirements and hence offers a possible countermeasure (Section V). Finally, we give some conclusions (Section VI).

II. Hash functions used in Vermont and nProbe

Vermont and nProbe use different functions to calculate the hash keys for incoming packets. The input for the hash function is given by the specification of the flow template. This template includes so-called flow keys that define the fields used as grouping criteria. A typical 5-tuple based flow aggregation would use the following flow keys: <$srcIP, dstIP, srcP, dstP, proto$>.

Whereas nProbe uses simple binary addition of all flow keys for its hash function, Vermont uses a nested call hierarchy of the CRC32 algorithm. In the following, we analyze both techniques and determine their attack vectors.

A. Hash index calculation in nProbe

Due to the simplicity of nProbe's hash function, it is easy to see how collisions can be created. Assuming packets with the same values in source IP address ($srcIP$), destination IP address ($dstIP$), source port ($srcP$), destination port ($dstP$), and protocol ($proto$) are aggregated to a single flow, nProbe would calculate the hash key k of size h_{size} as follows:

$$k = (srcIP + dstIP + srcP + dstP + proto) \mod h_{\text{size}} \quad (1)$$

Every packet producing the same result from Equation 1 will be stored in the same hash bucket and will therefore increase the length of the corresponding bucket list unless it belongs to an already saved flow. If an attacker wants to create a packet p_{i+1}, which collides with packet p_i, he simply sets $srcP_{p_{i+1}} = srcP_{p_i} - 1$, $srcIP_{p_{i+1}} = srcIP_{p_i} + 1$, and copies all other flow keys. Both packets then produce the same hash key. Attackers usually avoid modifying $dstIP$, as the packet may not reach the targeted monitoring system anymore – unless the attacker has more information about the network structure.

Since the goal is to deplete resources of the network monitor, it is not necessary that the packet is actually accepted by a receiver. It is therefore possible to modify all the remaining fields used in Equation 1 – resulting in four freely changeable fields: $srcIP$ (32 Bit), $srcP$ (16 Bit), $dstP$ (16 Bit), and $proto$ (1 Bit for TCP and UDP, assuming no other protocol will be used). Thus, it is theoretically possible to create $(2^{32} \times 2^{16} \times 2^{16} \times 2^{1})/h_{size}$ different packets matching the same hash bucket. For a hash table size of 16 Bit, 2^{49} flows can be created that are added to the same hash bucket. Even if the destination port cannot be varied, e.g. due to installed firewalls, still 2^{33} different flows are matching a single hash bucket.

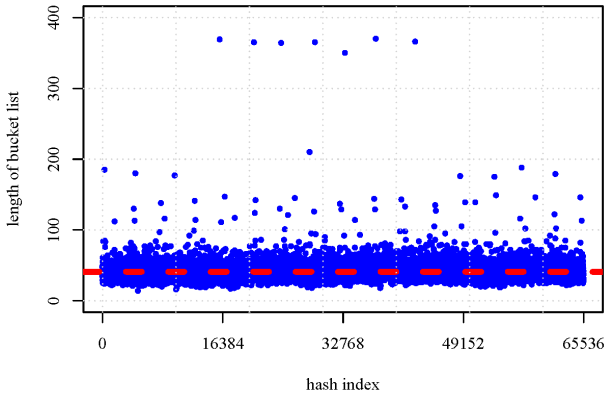

Figure 1. Distribution of nProbe's hash function

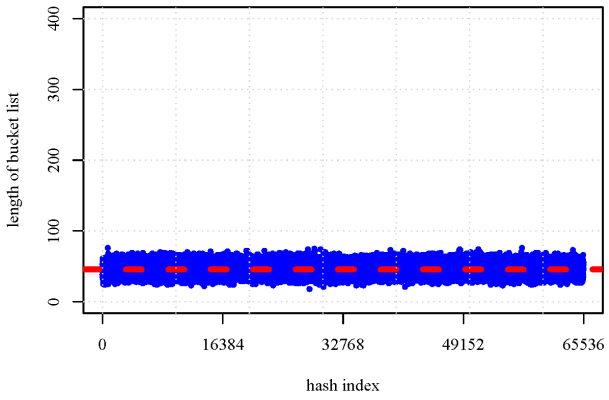

Figure 2. Distribution of Vermont's hash function

We captured a packet trace from a network and used nProbe to aggregate it to flows. h_{size} was set to 2^{16} and nProbe was configured not to remove any buckets from its hash table during the test. Figure 1 shows the utilization of hash buckets after 3 million packets. Whereas the distribution over all buckets seems to work appropriately, there is a noticeable amount of buckets that contain a considerably larger number of entries. Thus, the distribution is not uniform – however, in the general case, the number of overused buckets might be small enough to ensure a proper operation of the flow aggregation.

B. CRC32 based hashing in Vermont

Instead of summing up all flow keys, Vermont calculates hash keys via repeated execution of the CRC32 checksum algorithm. Listing 1 shows Vermont's index calculation for the hash table. The initial seed value can either be constant or random. Flow aggregation with five flow keys results in calling the CRC32 function five times. Furthermore, each result from the preceding calculation is used as the seed for the next calculation.

We used the identical test setting as in the hash bucket test of nProbe: Vermont was configured not to remove any entries from its hash table with size 2^{16} and used the same

packet trace to monitor 3 million packets. Figure 2 shows that this method offers a roughly uniform distribution over the hash table and, thus, fulfills one of the three requirements. Furthermore the CRC32 algorithm performs quite fast, so its usage as a hash function seems suitable.

Listing 1. Calculate hash bucket

```
hash = seed;
flowkey = flowkeys.first();
while (flowkey != null)
{
  hash = crc32(hash, flowkey);
  flowkey = flowkey->next;
}
return hash;
```

III. Limitations of the CRC Hash

In order to show that the CRC32 algorithm does not provide appropriate collision resistance, we first explain the operation of CRC32. The most frequently used method in performance critical systems is the direct lookup table method [9], which is depicted in Figure 3. This algorithm, which is listed in detail in Algorithm 1, works on a per byte basis. The creation of the lookup table is depicted in Listing 2.

Listing 2. Lookup table creation

```
for(int i = 0; i < 256; i++)
{
  CRC = i;
  for(int j = 8; j > 0; j--)
  {
    if(CRC & 1)
      CRC = (CRC >> 1) ^ GeneratorPoly;
    else
      CRC >>= 1;
  }
  CRCTable[i] = CRC;
}
```

It has been shown that it requires the change of n Bit in the byte stream to create a collision for a CRC-n function [10]. Please note that these collisions are calculated and not the result of a plain brute force attack. Collisions for a known seed can be calculated straight forward.

The IEEE recommended CRC32, which has been first proposed in 1975 [11], uses 0xFFFFFFFF as the initial seed. This seed is placed in the register before the algorithm is executed. Before returning the final result, i.e. the value of the register after processing all bytes, the register is XORed with a final value. This final XOR is 0xFFFFFFFF for the CRC32, i.e. the bits of the register are simply flipped. The CRC16 uses 0x0000 for both initial seed and final XOR value.

Algorithm 1 CRC calculation

1: move seed s into register r
2: **while** there are input bytes i to process **do**
3: put lowest byte of r in idx
4: Shift r right by one byte
5: XOR idx with new byte from i to yield index into lookup table
6: XOR lookup table value into register r
7: **end while**
8: XOR final value f into register r

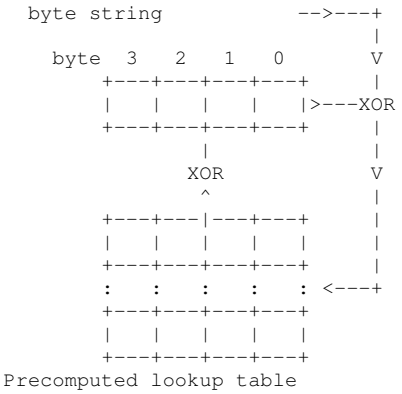

Figure 3. Working principle of the CRC32 lookup table method [10]

In the following, we refer to the computation of the CRC of byte vector a for seed s as $\text{crc}(s, a)$. For the CRC32, four bytes are needed to create a collision. Those bytes do not have to be added to the end, but can be placed anywhere in the byte array as shown in [12]. Furthermore, it is also possible to change four bytes that are already existing based on two given byte arrays a, b of arbitrary length, a byte vector x with a length of $4\,\text{Byte}$, a seed value s, and the final XOR value f.

Assuming we want to find $\text{crc}(s, axb) = Y$, we simply calculate a seed s' so that $\text{crc}(s', b) = Y$, then an x can be found so that $\text{crc}(s, ax) = f \oplus s'$. This will result in $\text{crc}(s, axb) = Y$.

Even nested CRC32 calls do not provide additional security against collision attacks. Let s be again the initial seed value, a, a', b, and c random strings, and f be the final XOR value, then

$$\begin{aligned}&\text{crc}(\text{crc}(s, a), b) = \text{crc}(\text{crc}(s, a'), b)\\ &\Leftrightarrow \text{crc}(s, a') = \text{crc}(s, a).\end{aligned}$$

Furthermore, it holds that

$$\begin{aligned}&\text{crc}(\text{crc}(s, b), a) = \text{crc}(s, ca)\\ &\Leftrightarrow \text{crc}(s, c) \oplus f = \text{crc}(s, b).\end{aligned}$$

For CRC16, the initial seed and final XOR value are set to 0x0000, so

$$\begin{aligned}\text{crc}(s, ab) &= \text{crc}(f \oplus \text{crc}(s, a), b)\\ &= \text{crc}(\text{crc}(s, a), b).\end{aligned}$$

Collisions found for one seed value will not necessarily collide for another seed value. However, it is possible to use already found collisions and add a prefix so they will collide for another seed. Let the byte vectors a and b collide for seed s, i.e. $\text{crc}(s, a) = \text{crc}(s, b)$. We can now let these byte arrays collide for seed s' by adding a prefix p, which will fill the register with value s, i.e. $f \oplus \text{crc}(s', p) = s$. If such a p is

found, then

$$\begin{aligned}\text{crc}(s', pa) &= \text{crc}(s', pb)\\ &= \text{crc}(s, a)\\ &= \text{crc}(s, b).\end{aligned}$$

A possible countermeasure against attacks based on the described hash collision procedure would be to use a random seed value, i.e. to make it impossible for the attacker to guess the seed to use for the collision calculations. Unfortunately, we found out that two strings a and b with $|a| = |b|$ will collide independently of the seed. For simplicity reasons, we only show this for the CRC16.

Collisions are seed independent if the bytes used to create collisions are independent of the seed. If a collision for a three byte long vector $\vec{k} = k_0k_1k_2$ should be found, we need to calculate x_1x_2 to a given x_0 so that $\text{crc}(s, \vec{k}) = \text{crc}(s, \vec{x})$.

Using the table lookup function lkp (see Listing 2), the calculation of the CRC $\text{crc}(s, \vec{k}) = \text{crc}(s, k_0k_1k_2)$ works as follows:

$\text{crc}(s, \vec{k})$:

iter.	$hi(r)$	$lo(r)$	table lookup
0	$hi(s)$	$lo(s)$	$a_0 = lkp(lo(r) \oplus k_0)$
1	$hi(a_0)$	$hi(s) \oplus lo(a_0)$	$a_1 = lkp(lo(r) \oplus k_1)$
2	$hi(a_1)$	$hi(a_0) \oplus lo(a_1)$	$a_2 = lkp(lo(r) \oplus k_2)$
3	$hi(a_2)$	$hi(a_1) \oplus lo(a_2)$	$r \oplus f = CRC$

$\vec{r}$ denotes the register, and $hi()$ and $lo()$ refer respectively to the higher and lower byte of the given variable.

Similarly, $\text{crc}(s, \vec{x}) = \text{crc}(s, x_0x_1x_2)$ can be calculated:

$\text{crc}(s, \vec{x})$:

iter.	$hi(r)$	$lo(r)$	table lookup
0	$hi(s)$	$lo(s)$	$a'_0 = lkp(lo(r) \oplus x_0)$
1	$hi(a'_0)$	$hi(s) \oplus lo(a'_0)$	$a'_1 = lkp(lo(r) \oplus x_1)$
2	$hi(a'_1)$	$hi(a'_0) \oplus lo(a'_1)$	$a'_2 = lkp(lo(r) \oplus x_2)$
3	$hi(a'_2)$	$hi(a'_1) \oplus lo(a'_2)$	$r \oplus f = CRC$

The high byte of each entry in the lookup table is unique [10]. Thus, from $\text{crc}(s, \vec{k}) = \text{crc}(s, \vec{x})$ it can be concluded that $a_2 = a'_2$ and $a_1 = a'_1$. This means that

$$hi(s) \oplus lo(a'_0) \oplus x_1 = hi(s) \oplus lo(a_0) \oplus k_1$$

$$\begin{aligned}\Rightarrow x_1 &= lo(a'_0) \oplus lo(a_0) \oplus k_1\\ &= lo(lkp(lo(s) \oplus x_0)) \oplus\\ &\quad\ lo(lkp(lo(s) \oplus k_0)) \oplus k_1\end{aligned}$$

and furthermore

$$hi(a'_0) \oplus lo(a'_1) \oplus x_2 = hi(a_0) \oplus lo(a_1) \oplus k_2$$

$$\begin{aligned}\Rightarrow x_2 &= hi(a'_0) \oplus hi(a_0) \oplus k_2\\ &= hi(lkp(lo(s) \oplus x_0)) \oplus\\ &\quad\ hi(lkp(lo(s) \oplus k_0)) \oplus k_2.\end{aligned}$$

To prove that x_1 and x_2 are independent of s, we have to show that $lkp(lo(s) \oplus x_0) \oplus lkp(lo(s) \oplus k_0))$ is independent of s. This can be done by studying the $lkp()$ function, which is described in Listing 2. This function creates the lookup table used for the CRC calculation.

A mathematical representation of the $lkp()$ function is given below (we assume that $lkp(x)$ will create the lookup table entry for index x, with G_i being the i'th bit of the generator polynomial and $c_i(t)$ referring to the i'th bit after the t'th iteration):

$$\begin{aligned}\vec{c}(0, x) &= x\\ c_i(t, x) &= \begin{cases} 0 & i > 15\\ (G_i \& c_0(t-1, x)) & \\ \quad \oplus c_{i+1}(t-1, x) & i \leq 15\end{cases}\\ lkp(x) &= \vec{c}(7, x)\end{aligned}$$

Looking into the calculation of the i'th bit of $lkp(lo(s) \oplus x_0) \oplus lkp(lo(s) \oplus k_0))$, we first show that the calculation only depends on the first iteration of the $lkp()$ function:

$$\begin{aligned}&c_i(t, x) \oplus c_i(t, k) =\\ &= ((G_i \& c_0(t-1, x)) \oplus c_{i+1}(t-1, x)) \oplus\\ &\quad ((G_i \& c_0(t-1, k)) \oplus c_{i+1}(t-1, k))\\ &= (G_i \& c_0(t-1, x)) \oplus (G_i \& c_0(t-1, k)) \oplus\\ &\quad c_{i+1}(t-1, x) \oplus c_{i+1}(t-1, k)\\ &= (G_i \& (c_0(t-1, x) \oplus c_0(t-1, k))) \oplus\\ &\quad c_{i+1}(t-1, x) \oplus c_{i+1}(t-1, k)\end{aligned}$$

Now that we have reduced the problem of $\vec{c}(t, x) \oplus \vec{c}(t, k)$ to $\vec{c}(t-1, x) \oplus \vec{c}(t-1, k)$, it is sufficient to show that all seed bits cancel each other out for $\vec{c}(0, x) \oplus \vec{c}(0, k)$:

$$\begin{aligned}\vec{c}(0, x) \oplus \vec{c}(0, k) &= lo(s) \oplus x_0 \oplus lo(s) \oplus k_0\\ &= x_0 \oplus k_0\end{aligned}$$

If it can further be shown that this is valid not only for vectors of length three but for byte arrays of arbitrary length, we can prove that collisions for the CRC16 algorithm are independent of the seed if both input vectors have equal length. The calculation of $\text{crc}(s, \vec{k})$ and $\text{crc}(s, \vec{x})$ with $|\vec{x}| = |\vec{k}| = n - 1$ is similar to the three-byte operation.

$\text{crc}(s, \vec{k})$:

iter.	$hi(r)$	$lo(r)$	table lookup
0	$hi(s)$	$lo(s)$	$a_0 = lkp(lo(r) \oplus k_0)$
1	$hi(a_0)$	$hi(s) \oplus lo(a_0)$	$a_1 = lkp(lo(r) \oplus k_1)$
.	.	.	.
.	.	.	.
n-1	$hi(a_{n-2})$	$hi(a_{n-3}) \oplus lo(a_{n-2})$	$a_{n-1} = lkp(lo(r) \oplus k_{n-1})$
n	$hi(a_{n-1})$	$hi(a_{n-2}) \oplus lo(a_{n-1})$	$a_n = lkp(lo(r) \oplus k_n)$
n+1	$hi(a_n)$	$hi(a_{n-1}) \oplus lo(a_n)$	$r \oplus f = CRC$

crc$(s, \vec{x})$:

iter.	$hi(r)$	$lo(r)$	table lookup
0	$hi(s)$	$lo(s)$	$a'_0 = lkp(lo(r) \oplus x_0)$
1	$hi(a'_0)$	$hi(s) \oplus lo(a'_0)$	$a'_1 = lkp(lo(r) \oplus x_1)$
.	.	.	.
.	.	.	.
n-1	$hi(a'_{n-2})$	$hi(a'_{n-3}) \oplus lo(a'_{n-2})$	$a'_{n-1} = lkp(lo(r) \oplus x_{n-1})$
n	$hi(a'_{n-1})$	$hi(a'_{n-2}) \oplus lo(a'_{n-1})$	$a'_n = lkp(lo(r) \oplus x_n)$
n+1	$hi(a'_n)$	$hi(a'_{n-1}) \oplus lo(a'_n)$	$r \oplus f = CRC$

For the computed bytes x_{n-1} and x_n, the bytes that create the collision, we now have:

$$\begin{aligned} x_{n-1} &= hi(a'_{n-3}) \oplus hi(a_{n-3}) \oplus \\ &\quad lo(a'_{n-2}) \oplus lo(a_{n-2}) \oplus k_{n-1} \\ x_n &= hi(a'_{n-2}) \oplus hi(a_{n-2}) \oplus k_n \end{aligned}$$

We already demonstrated that all seed bits are canceled for $n = 2$. In order to show that this holds for any n, we have to prove that $a_i \oplus a'_i$ is independent of s. The CRC algorithm shows that values for a_i (and a'_i likewise) can be calculated as follows:

$$\begin{aligned} a_0 &= lkp(lo(s) \oplus k_0) \\ a_1 &= lkp(hi(s) \oplus lo(a_0) \oplus k_1) \\ a_i &= lkp(hi(a_{i-2}) \oplus lo(a_{i-1}) \oplus k_1) \end{aligned}$$

With this knowledge, first, it can be seen that increasing n by one results into one additional nested call to $lkp()$ in a_{n-2} and a'_{n-2}. If $|\vec{x}| = |\vec{k}|$, the number of nested calls are equal. Secondly, as shown in Listing 2, the $lkp()$ function performs bit-wise operations and each input bit will always affect the output bits at the same position. This means the input bits of the innermost nested call to $lkp()$ will affect the same bits of the final outcome.

Due to its recursive nature, the results of the two innermost calls will be a_0 and a_1, or a'_0 and a'_1, respectively. This results in s affecting the same bits in a_i and a'_i. Hence $a_i \oplus a'_i$ will cancel out all bits of s, thus, x_n and x_{n-1} are independent of s.

N.B., if $|\vec{x}| \neq |\vec{k}|$, the number of nested calls on both sides differ and the seed bits will affect different bits of the final outcome, thus will not cancel each other out.

Exploiting the knowledge that random CRC seeds do not protect against hash collision attacks, an attacker can construct packets that produce hash collisions, i.e. the corresponding flows will be stored in the same hash bucket in Vermont. For random destination and source ports, it is possible to create any CRC value by manipulating the source IP field, which provides sufficient space for the needed bytes.

Say an attacker wants to create collisions for the hash key Y. Using the IP-5-tuple as flow keys, Vermont calculates the hash key as follows:

$$Y = \text{crc}(\ldots(\text{crc}(s, srcIP), dstIP), prot), srcP), dstP).$$

The attacker then does the following:

- Choose random values for $srcP'$ and $dstP'$
- Calculates seed s' so that $Y = \text{crc}(\text{crc}(s', srcP'), dstP')$. This is possible using the reverse CRC operation described in [10].
- If the destination IP should not be changed and the protocol is either TCP or UDP, then calculate seed s'' so that $\text{crc}(\text{crc}(s'', dstIP), prot) = s'$.
- In a final step, the necessary source IP is calculated so that $\text{crc}(s, srcIP) = s''$.

With this method, it is possible to create $2^{16} \times 2^{16} \times 2^1 = 2^{33}$ different flows for which Vermont's CRC32-based hashing algorithm produces the same hash value. Because all of the above fields have the same length for each packet, they will collide independently of the seed.

IV. Attacking the network monitor

We evaluated the identified threats for producing attacks against the hash based data structures of typical flow monitors. In particular, we explored this for nProbe and Vermont. For all our experiments, we prepared a test network consisting of two server PCs (Dual Xeon 2 GHz) directly connected over a 100 MBit/s network link to evaluate the particular attack model. The network monitoring software was installed on one PC while the attack was launched from the other one. In a first test, we simulated a typical SYN flood based DoS attack [13] by randomly creating TCP SYN packets and sending these to the monitor. The attack packets were sent using the `tcpreplay` toolkit [14].

For our experiments, we configured the hash table size to 2^{16} buckets. Both the passive and active timeout were set to values larger than 60 s, thus, no packets were exported during the test. Figure 4 shows the results of our experimental attacks against nProbe and Vermont. Each value represents the average percentage dropped packets over a 60 s interval.

While the regular SYN-based DoS attack reached a maximum of about 6 % of dropped packets, the hash collision attack only needed a data rate of about 7500 packets/s to exceed 90 % of dropped packets. Note that the hash table size has no influence on the hash collision attack, but only on the SYN DoS attack because the former just targets one bucket, while the latter spreads over the whole data structure. As can be seen, the hash collision attack is similarly effective for nProbe and Vermont. The SYN DoS attack is only shown for Vermont. The results for nProbe were similar.

In order to measure the effect of flow exporting on the collision attack, we configured Vermont's export interval to 30 s with a passive timeout of 30 s. In this set of experiments, we used three different packet rates for the hash collision attack. The results are depicted in Figure 5.

It took a few seconds for the hash bucket list to exceed some maximum length (which is dependent on the available processing performance of the used CPU) for the lower attack rates. As can be seen, 10 000 packets/s were enough to exceed a rate of 50 % dropped packets after one second and 90 % of dropped packets after three seconds. The longer the bucket list

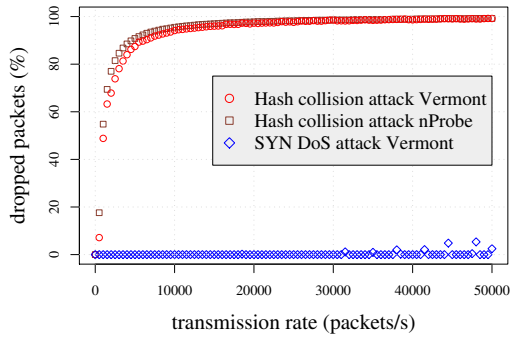

Figure 4. DoS vs. hash collision attacks on Vermont and nProbe

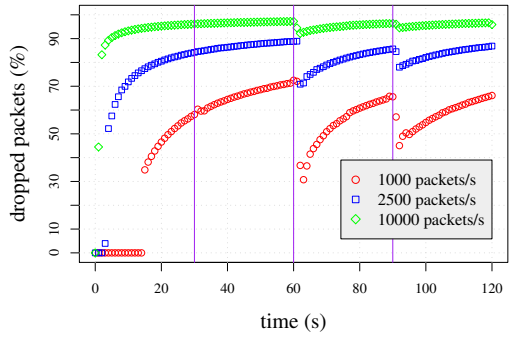

Figure 5. Hash collision attack with different speeds

is, the more packets are dropped because it takes longer time to traverse the list.

There are no flows exported in the first export because there are no flows older than 30 s. The second export at 60 s will shorten the bucket list by the amount of flows received and progressed longer than 30 s ago. Although this should be a considerable percentage of all saved flows it only has temporary effects because the bucket list will instantly be filled again by the attacker. With an attack speed of 10 000 packets/s even the temporary effect is only marginal. N.B., an attack speed of 10 000 packets/s results in a transmission rate of about 4 MBit/s due to the small packet size of 54 byte.

V. Finding an optimized hash function

The obvious countermeasure against hash collision based DoS attacks is a hash function for which collisions cannot that easily be created. Cryptographic hash functions such as SHA-1 [15] would provide such a feature, however, they are computationally too expensive for efficient use in a flow monitor. Since computation time is obviously important, the first idea was to keep the algorithm as simple as possible, whereas not exploitable. Unfortunately, we were not able to design a simple function based only on two simple operators, which provided a uniform distribution. Our attempts showed a very similar distribution compared to nProbe (see Figure 1). So, in order to sustain the uniform distribution offered by Vermont's method, we decided that the CRC constitutes a good basis for a hash function.

To ensure that an attacker cannot exploit the used hash function to create collisions, we designed a slightly modified CRC-based version. Instead of computing the CRC over a flow key, we first add a secret (random) value, which is initialized upon starting the flow monitor. Every flow key i has its corresponding random value $w(i)$. The resulting formula for calculating the hash index for a packet is then:

$$\begin{aligned} index \quad = \quad & crc(...crc(s, srcIP + w(srcIP)), \\ & dstIP + w(srcIP)), \\ & proto + w(proto)), \\ & srcP + w(srcP)), \\ & dstP + w(dstP)) \bmod h_{\text{size}} \end{aligned}$$

The addition of value $w(i)$ to a flow key of length n can be seen as a shift of the flow key in its respective residue class ring $\mathbb{Z}_{2^n}$ and hence does not affect the index distribution, meaning the new method offers the same distribution of hash keys as Vermont. This is also shown in Figure 6. Because addition is not distributive over exlusive or and vice versa we believe that this makes it significantly harder to create collisions. The crc calculation, with $c(x+y)$ denoting the carry over bit from $x+y$ and w being the corresponding random value for k and x, can then be described as follows:

crc(s,$\vec{k}+\vec{w}$)

iter.	$hi(r)$	$lo(r)$	table lookup
0	$hi(s)$	$lo(s)$	$a_0 = lkp(lo(r) \oplus (k_0 + w_0))$
1	$hi(a_0)$	$hi(s) \oplus lo(a_0)$	$a_1 = lkp(lo(r) \oplus (k_1 + w_1 + c(k_0 + w_0)))$
2	$hi(a_1)$	$hi(a_0) \oplus lo(a_1)$	$a_2 = lkp(lo(r) \oplus (k_2 + w_2 + c(k_1 + w_1)))$
3	$hi(a_2)$	$hi(a_1) \oplus lo(a_2)$	$\vec{r} \oplus f = CRC$

If we want to create a collision, meaning input values $\vec{k}$ and $\vec{x}$ produce the same hash, we depict the calculation for $\vec{x}$ as follows:

crc(s,$\vec{x}+\vec{w}$)

iter.	$hi(r)$	$lo(r)$	table lookup
0	$hi(s)$	$lo(s)$	$a'_0 = lkp(lo(r) \oplus (x_0 + w_0))$
1	$hi(a'_0)$	$hi(s) \oplus lo(a'_0)$	$a'_1 = lkp(lo(r) \oplus (x_1 + w_1 + c(x_0 + w_0)))$
2	$hi(a'_1)$	$hi(a'_0) \oplus lo(a'_1)$	$a'_2 = lkp(lo(r) \oplus (x_2 + w_2 + c(x_1 + w_1)))$
3	$hi(a'_2)$	$hi(a'_1) \oplus lo(a'_2)$	$\vec{r} \oplus f = CRC$

This results in:

$$\begin{aligned} x_1 \quad = \quad & (lo(a'_0) \oplus lo(a_0) \oplus (k_1 + w_1 + c(k_0 + w_0))) \\ & -(w_1 + c(x_0 + w_0)) \\ = \quad & (lo(lkp(lo(r) \oplus (x_0 + w_0))) \\ & \oplus lo(lkp(lo(r) \oplus (k_0 + w_0))) \\ & \oplus (k_1 + w_1 + c(k_0 + w_0))) - (w_1 + c(x_0 + w_0)) \end{aligned}$$

We believe that x_1 can not be calculated without knowledge of w. This only holds as long as an attacker does not know a pair x, y with $y = crc(\vec{x} + \vec{w})$.

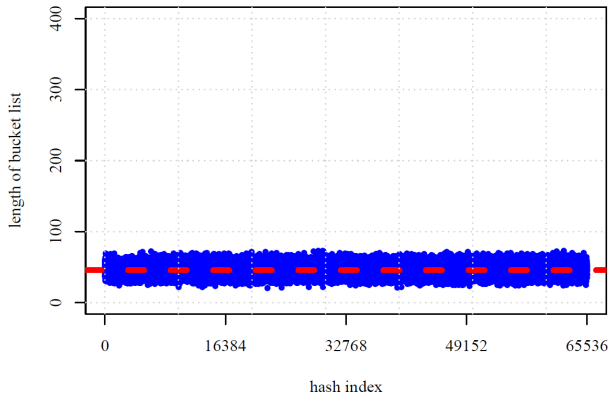

Figure 6. Distribution of the improved crc hashing method

Although it is slightly more effort to compute the hash values in comparison to Vermont's or nProbe's method, it still computes fast enough as can be seen in Table I). All functions were benchmarked on an Athlon™ 64 X2 Dual Core Processor 5200+ using only one core.

Algorithm	avg time/iteration	iterations per second
nProbe	152 ns	6 578 947
Vermont	192 ns	5 208 333
new method	203 ns	4 926 108

Table I
COMPARISON OF HASH CALCULATION TIMES

Alternatively, there is a class of hash functions called universal hash functions, first introduced in 1977 [16], [17]. These functions provide the feature that the probability of $h(M') = h(M)$ for any two input vectors M, M' and $M' \neq M$ is at most $1/N$, with N being the cardinality of the set of possible outputs. These functions make use of a pre-computed random vector, based on a pseudo random function, and a large prime number. The length of the random vector must fit the maximum input length. Due to the limited and known input lengths of flow keys these functions seem suitable. There are several optimized versions which should run fast on today's processors [18], [19].

The third and probably best solution would be the use of a very fast implementation of the MD5 algorithm [20]. The Crypto++ library provides a good enough performance to efficiently deploy MD5 hashing in a flow monitor. Our tests confirmed the results shown on the Crypto++ website.[1] The implementation was only marginally slower than our modified CRC version. The monitor would then calculate the hash table index as follows, with $\circ$ being concatenation.

$$\text{index} = \text{md5}(srcIP \circ srcP \circ dstIP \circ dstP \circ proto) \bmod h_{\text{size}}$$

[1]http://www.cryptopp.com

VI. CONCLUSION

In this paper, we investigated hash functions used for the two typical software based flow monitors Vermont and nProbe. We clearly identified several vulnerabilities of these hash functions w.r.t. possible attacks against the flow monitors, i.e. against the network security environment itself. Such attacks cannot easily be prevented by simply adding further computational resources as the attack can be successfully performed even with low-rate attack streams. Therefore, such an attack cannot be compared to simple distributed DoS attacks that focus on sending huge amounts of packet data in order to degrade the available quality of service in the network (or even to bring down selected networking entities).

Choosing the right hash function for use in a flow monitor is therefore a critical task. The pure computational effort seems to be the most important property at first sight, i.e. a function, which primarily fulfills this requirement, could result into a non-uniform distribution of the hash keys or even create a security problem similar to the analyzed hash collision attack. The proposed hashing method seems to offer protection against hash collision attacks and computes fast enough to be deployed in high speed flow meters. There might even be a simpler solution using randomized permutation or substitution tables.

REFERENCES

[1] R. Sommer and A. Feldmann, "NetFlow: information loss or win?" in *2nd ACM SIGCOMM Internet Measurement Workshop (IMW 2002)*. Marseille, France: ACM, November 2002, pp. 173–174.

[2] T. Limmer and F. Dressler, "Seamless Dynamic Reconfiguration of Flow Meters: Requirements and Solutions," in *16. GI/ITG Fachtagung Kommunikation in Verteilten Systemen (KiVS 2009)*. Kassel, Germany: Springer, March 2009, pp. 179–190.

[3] R. T. Lampert, C. Sommer, G. Münz, and F. Dressler, "Vermont - A Versatile Monitoring Toolkit Using IPFIX/PSAMP," in *IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation (MonAM 2006)*. Tübingen, Germany: IEEE, September 2006, pp. 62–65.

[4] L. Deri, "nProbe: an Open Source NetFlow Probe for Gigabit Networks," in *TERENA Networking Conference (TNC 2003)*, Zagreb, Croatia, May 2003.

[5] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," IETF, RFC 5101, January 2008.

[6] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer, "Information Model for IP Flow Information Export," IETF, RFC 5102, January 2008.

[7] B. Claise, "Cisco Systems NetFlow Services Export Version 9," IETF, Tech. Rep. RFC 3954, October 2004.

[8] G. Carle, F. Dressler, R. A. Kemmerer, H. König, C. Kruegel, and P. Laskov, "Manifesto - Perspectives Workshop: Network Attack Detection and Defense," in *Dagstuhl Perspectives Workshop 08102 - Network Attack Detection and Defense 2008*, Schloss Dagstuhl, Wadern, Germany, March 2008.

[9] D. V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Look-Up," *Communications of the ACM*, vol. 31, no. 8, pp. 1008–1013, 1988.

[10] Anarchriz/DREAD, "CRC and how to Reverse it," April 1999. [Online]. Available: http://www.woodmann.com/RCE-CD-SITES/Anachriz/programming/crc.htm

[11] K. Brayer and J. L. Hammond Jr, "Evaluation of error detection polynomial performance on the AUTOVON channel," in *National Telecommunications Conference*, vol. 1. New Orleans, LA: IEEE, December 1975, pp. 8–21 to 8–25.

[12] B. Maxwell, D. R. Thompson, G. Amerson, and L. Johnson, "Analysis of CRC methods and potential data integrity exploits," in *International Conference on Emerging Technologies*, Minneapolis, MI, August 2003, pp. 25–26.

[13] H. Wang, D. Zhang, and K. G. Shin, "Detecting SYN Flooding Attacks," in *21st IEEE Conference on Computer Communications (IEEE INFOCOM 2002)*, New York, NY, June 2002.
[14] A. Turner, "tcpreplay." [Online]. Available: http://tcpreplay.synfin.net/trac/wiki/tcpreplay
[15] D. Eastlake and P. Jones, "US secure hash algorithm 1 (SHA1)," IETF, RFC 3174, September 2001.
[16] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *9th ACM Symposium on Theory of Computing*. Boulder, CO: ACM, May 1977, pp. 106–112.
[17] S. Crosby and D. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *12th USENIX Security Symposium*, Washington, D.C., August 2003, pp. 3–3.
[18] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "UMAC: Fast and secure message authentication," in *19th International Cryptology Conference (Advances in Cryptology, CRYPTO '99)*, vol. LNCS 1666, Santa Barbara, CA, August 1999, pp. 216–233.
[19] D. Bernstein, "Floating-point arithmetic and message authentication," March 2000. [Online]. Available: http://cr.yp.to/hash127.html
[20] R. Rivest, "The MD5 Message-Digest Algorithm," IETF, RFC 1321, April 1992.