Multi-pattern Signature Matching for Hardware Network Intrusion Detection Systems, by Haoyu Song and John W. Lockwood, IEEE Globecom 2005, St. Louis, MO, Nov. 28, 2005, pp. CN-02-3.

# Multi-pattern Signature Matching for Hardware Network Intrusion Detection Systems

Haoyu Song, John W. Lockwood
{hs1, lockwood}@arl.wustl.edu
Department of Computer Science and Engineering
Washington University in St. Louis, USA, 63130

***Abstract*— Network Intrusion Detection System (NIDS) performs deep inspections on the packet payload to identify, deter and contain the malicious attacks over the Internet. It needs to perform exact matching on multi-pattern signatures in real time. In this paper we introduce an efficient data structure called Extended Bloom Filter (EBF) and the corresponding algorithm to perform the multi-pattern signature matching. We also present a technique to support long signature matching so that we need only to maintain a limited number of supported signature lengths for the EBFs. We show that at reasonable hardware cost we can achieve very fast and almost time-deterministic exact matching for thousands of signatures. The architecture takes the advantages of embedded multi-port memories in FPGAs and can be used to build a full-featured hardware-based NIDS.**

## I. Introduction

Some content strings of Internet packet payload, also known as "signatures," imply network intrusion attempts. Signature-based Network Intrusion Detection System (NIDS) collects these signatures and scans the payload of the Internet packets for them in order to identify, deter and contain such malicious behaviors. A scalable and fast solution is needed to accommodate the largest signature set today and to sustain the real time processing of the high-speed network.

Bloom Filter [4] is an efficient data structure enabling fast membership query with tunable false positive rate. Dharmapurikar et al have designed a multi-pattern signature-matching scheme using Bloom Filters [6]. On the scan process, whenever the front-end Bloom Filter reports a possible match, the string is extracted and used to probe another independent hash table to decide the final match. There are two drawbacks in this scheme. Firstly, the extra lookups in the hash table might become the performance bottleneck due to the hash collisions. Secondly, there are many different signature lengths and the signature distribution on length is unbalanced, so to assign each length a Bloom Filter is inefficient in memory usage.

We find that the scheme does not effectively use the information revealed by the Bloom Filters and there is little consideration about the string load balancing among different Bloom Filters. To overcome these drawbacks, we propose an extension of the Bloom Filter data structure and a new lookup algorithm named Extended Bloom Filter (EBF). It is scalable and suitable for fast incremental updates. The hardware-based EBF is an alternative of the multi-pattern signature-matching problem and outperforms the software-based algorithms.

In this paper, we review the related work in Section II and then discuss our data structure and algorithms in Section III. A theoretical analysis and simulations follow in Section IV and V. Some improvements are presented in Section VI to further reduce the memory usage and boost the performance. The scheme to reduce the number of EBFs is introduced in VII. We briefly talk about the hardware NIDS implementation in Section VIII and conclude our contribution in Section IX.

## II. Related Work

Given a packet payload $T$ of length $n$ and a set of $m$ signatures $S[1]...S[m]$ of variable length for intrusion detection, the signature-matching problem is to determine any exact match of signature $S[i]$ and a substring of $T$. In NIDS, signature matching is a crucial component and decides the overall system performance. An analysis shows that in Snort, an open-source software-based NIDS, the signature matching alone consumes 30% to 80% of the CPU time [9]. While the network bandwidth and the size of the signature set keep growing, to perform real time detection is still far from realistic.

Boyer-Moore is the best-known algorithm for single string matching and is actually adopted for the implementation of the Snort. Fisk extended the Boyer-Moore algorithm to support set-wise string matching [8]. Coit does similar work in [5]. Aho-Corasick [2] is a finite state automaton supporting multi-pattern string matching. The major drawback is its excessive memory consuming. A modified algorithm of Aho-Corasick due to Tuck [14] reduces the amount of memory and improves its performance. Wu-Manber [15] uses a hash table plus the bad character heuristics to accelerate the searching speed. All these algorithms are developed mainly for software implementation. Analysis and experiments show no such algorithm is fast enough for real-time string matching in high-speed network. Thus, a hardware-assisted or pure hardware solution is becoming more and more attractive.

Sidhu [12] implemented Nondeterministic Finite Automaton (NFA) in hardware and later Moscola [10] implemented Deterministic Finite Automaton (DFA) in hardware to perform regular expression matching. While the match speed is fast, they both suffer the scalability problem: Too many states consume too many hardware resources. Dharmapurikar then proposed to use Bloom Filters to do the deep packet inspection [6]. Attig implemented a prototype of this scheme [3]. Our paper proposes significant improvement to this work and

0-7803-9415-1/05/$20.00 (C) 2005 IEEE

*This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE GLOBECOM 2005 proceedings.*

addresses more problems in a real hardware based NIDS implementation. Recently, Yu proposed a TCAM-based scheme for multi-pattern matching [16]. Despite the widely criticized inefficiency of TCAM device, the long string matching support involves multiple tables.

## III. EBF Data Structure and Algorithm

```
struct EBF {
  bit      hit;
  integer  counter;
  struct llist* link_list;
}

struct llist {
  struct slist* string;
  struct llist* next;
}
```

Fig. 1. EBF Data Structure

EBF eliminates the need of another hash table for match verification. Moreover, the information gained from the Bloom Filter is exploited to accelerate the lookups. As shown in Figure 1, each EBF bucket has three fields. The first field (Hit Bit) is one bit and has the same definition as in the original Bloom Filter. The second field is a counter that counts how many signatures address the bucket. Unlike the Counting Bloom Filter (CBF) [7], in which the counters are used for update purpose only, EBF uses the counters to accelerate the signature matching. The last field is a pointer. If some signature sets the Hit Bit, this pointer will eventually lead to it. All signatures hashed to a bucket are organized in a link list. In order to save memory, only one copy of each signature is stored and pointed by a pointer from the link list nodes. The algorithm can be described as two steps: EBF programming and membership query. Assuming the EBF contains $m$ buckets and $r$ hash functions: $H_1$ ... $H_r$, the algorithm to programming a signature $S$ is described in Figure 2.

```
Procedure Programming_EBF(S)
  Allocate memory space for S at address m;
  Store S at address m;
  for i in 1 to r do
    k = Hi(S);
    EBF[k].hit = 1;
    EBF[k].counter ++;
    Allocate memory space for a link list node
LN at address j;
    LN.string = m;
    LN.next = NULL;
    Follow the link till a node LLN.next = Null;
    LLN.next = j;
```

Fig. 2. Algorithm for EBF Programming

Figure 3 shows an example of EBF for which $k = 3$ and $m = 13$. Four signatures have been programmed into the EBF in sequence of $S1$, $S2$, $S3$, and $S4$.

Once the EBF is programmed, it is ready for membership query. As described in Figure 4, the algorithm retains the basic feature of Bloom Filter by filtering out the impossible match quickly. Moreover, in case all Hit Bits are set for a string,

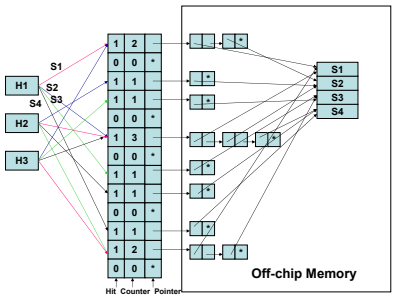


Fig. 3. Programming the Extended Bloom Filter

```
Procedure Query_EBF(S)
  min_counter = max possible number
  for i in 1 to r do
    k = Hi(S);
    if EBF[k].hit == 0
      There is no match, return;
    else
      if EBF[k].counter < min_counter
        min_counter = EBF[k].counter;
        min_index = k;
  pt = EBF[min_index].link_list;
  do while pt != Null
    if pt.string == S
      There is a match, return S;
    else
      pt = pt.next;
  There is no match, return;
```

Fig. 4. Algorithm for Signature Matching in EBF

the counter mechanism guides us to only search a shortest list to verify the membership. The algorithm is targeted at hardware implementation so we can fully exploit the hardware parallelism. All $r$ hash functions can be calculated in parallel and the multiple EBF buckets can be probed simultaneously.

It is straightforward to perform the signature update. To insert a signature, we call the procedure Programming_EBF. To delete a signature, we use the similar method described in the CBF scheme with additional work to remove the corresponding list node and recycle the memories. In all operations, memory for signatures and list nodes are dynamically allocated and de-allocated.

## IV. Analysis

Firstly, we analyze the average length of the link list in any EBF bucket. Assuming there are $n$ signatures, for a bucket, the probability $f_{set}$ that the Hit Bit is set by one signature is:

$$f_{set} = 1 - (1 - \frac{1}{m})^r \quad (1)$$

The probability $f_{i\_set}$ that a bucket's Hit Bit is set by $i$ signatures is:

$$f_{i\_set} = \left( \begin{array}{c} n \\ i \end{array} \right) [1 - (1 - \frac{1}{m})^r]^i (1 - \frac{1}{m})^{r(n-i)} \quad (2)$$

So the average length of link list $l_{avg}$ is

$$l_{avg} = E[i] = \Sigma_{i=0}^{n} i \times f_{i\_set} = n(1 - (1 - \frac{1}{m})^r) \quad (3)$$

For the Bloom Filter, when the false positive probability is minimized with respect to $r$, we get the relation

$$r = (m/n)ln2 \tag{4}$$

Based on these analysis, the approximation of link list length as:

$$l_{avg} = n(1 - (1 - \frac{1}{m})^{(m/n)ln2}) \approx ln2 \approx 0.7 \tag{5}$$

Under the same conditions, the probability for a bucket to be empty is roughly 0.5. This implies that half of the total buckets are never hit by any member signature. Using same parameters as in [3]'s Bloom Filter implementation, the false positive rate is as low as 0.00097 and the average link list length is only 0.4. This result shows that the link list is typically very short.

Theoretically, if the $r$ buckets are randomly selected from all the buckets with non-zero counter value, we would like to know the expected shortest link list. Assuming the percentage of buckets with a counter value of $i$ over all the buckets with non-zero counter value is $p_i$. $P(j)$ is the probability that the smallest counter value is $j$ among the $r$ randomly selected buckets. Therefore, we get the equation:

$$P(1) = 1 - P(> 1) = 1 - (1 - p_1)^r \tag{6}$$

and when $j > 1$

$$P(j) = 1 - P(< j) - P(> j)$$
$$= (1 - \Sigma_{k=1}^{j-1} p_k)^r - (1 - \Sigma_{k=1}^{j} p_k)^r \tag{7}$$

The expected value of $j$ is

$$E[j] = \Sigma_{j=1}^{\infty} j \times P(j) = 1 + \Sigma_{s=1}^{\infty} (1 - \Sigma_{t=1}^{s} p_t)^r \tag{8}$$

We know that in the optimal case, half of the bucket have a zero value counter, so we have the relation:

$$p_i = 2 \times f_{i\_set} \tag{9}$$

Using the optimal EBF parameters, $P(1)$ is greater than 0.99 and $E[j]$ is slightly larger than one. This shows in most cases actually, only the link lists with length one need to be checked if none of the $r$ hashed counters is zero. Note that the analysis is not very strict and only reflect the approximate results. We use simulations as backup in the following section.

## V. Simulation

Simulation results support our theoretical analysis. We use the same parameter configurations as in [3]: 1419 signatures are programmed by 10 hash functions into a 20K-bucket EBF. In the worst and extremely rare cases, at most two memory accesses are needed. In case that the signatures are all 2-byte long, we test all the 65535 possible strings, only seven of them need two memory accesses and others need only one, even though the link list lengths range from two to five. The longer signature cases exhibit even better performance.

We also did some experiments based on the different combinations of the number of hash functions and EBF buckets

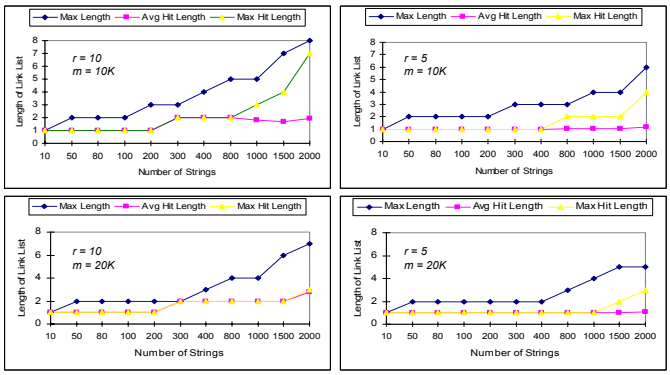


Fig. 5. Simulation I

as shown in Figure 5. We ran a million queries that cover all the programmed signatures. The "Max Length" curve indicates the length of the longest link list in EBF. As more signatures are programmed into EBF, the maximum length of link list grows. Fortunately, we do not need to traverse the longest list. The "Max Hit Length" curve shows the longest list we need to traverse in order to find the matching in any query. By decreasing the number of hash functions or increasing the number of buckets, we can effectively lower this number. However, using less hash functions may increase the false positive rate and using more EBF buckets will increase the resource consumption. Finally, the "Avg Hit Length" curve shows the average length of the list we traverse during the simulation. This number is typically less than two. For the Snort signatures, we have shown for any specific length, there are about or much fewer than 100 distinct signatures. In this case, our experiments indicates only one list node check is needed. By fine-tuning the parameters, we can support more than 1000 signatures with tolerable number of memory accesses.

To test the effect on Snort rule set, for each signature length, we only program 200 distinct signatures in EBF. This leaves spacious room for future update. By varying the number of EBF buckets and the number of hash functions, we see how the number of false positive match and average number of link list accesses change as shown in Figure 6.

In a query process, once a string is matched at some list node, there is no need to go through the following list nodes. This actually makes the real average performance better than the simulation results. In hardware, we could pipeline the memory lookups to achieve even better performance.

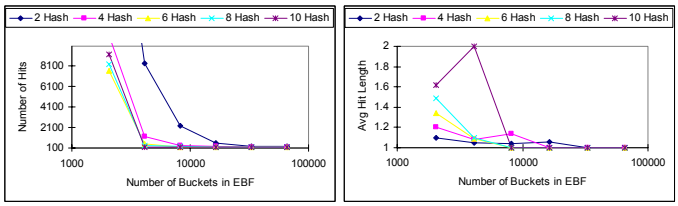


Fig. 6. Simulation II

## VI. EBF Optimizations

The EBF is stored in on-chip multi-port SRAM that enables parallel access in one clock cycle. Beside the Hit Bit, we budget 4 bits for the counter that supports link list length up to 15. We leave 11 bits to the pointer field that distinguish up to 2K signatures. Thus, each bucket is only 2- byte. With a 40K-byte on-chip SRAM, we could support 2K any length signature matching. Now we introduce several optimizations to further improve the storage efficiency and lookup performance.

*Max-Min Threshold:* After all signatures are programmed, the minimum link list length for each signature $S_i$ is $min(S_i)$. A value $B$ is maintained along with EBF, where $B = max\{min(S_i)..., \forall S_i\}$. The usage of $B$ is straightforward: whenever the Bloom Filter reports a match but the queried minimum counter value is greater than $B$, we know for sure this is a false positive so no link list access is needed. This heuristics comes form the observation that given the small false positive rate, $min(S_i)$ tends to be small. When 20K buckets, 10 hash functions and 2000 signatures are used, simulation shows $B = 1$ and for 1 million test inputs there are 114 false positive matches with shortest link list longer than 1. This simple improvement eliminates most of the expensive and unnecessary memory accesses.

*Compressed Counter:*In previous description, we separate the "Hit bit" and Counter in EBF. Actually, a non-zero counter implies some signatures hit the bucket so the "Hit bit" is redundant. A more aggressive counter scheme is used in EBF implementation. Each on-chip EBF bucket only contains $k$ bits for a saturated counter, where $k = \lfloor log(B+1) \rfloor + 1$. The pointer field is maintained in a table located in off-chip memories. In practical EBF configurations where $B$ is often less than two, two bits are enough to represent the counter and differentiate all useful scenarios, where saturated value "11" means the bucket is hit by three or more signatures. With this optimization, same design mentioned at the beginning of this section can be achieved by using only 40K-bit on-chip memories. Note that this saturated counter scheme loses the ability to track the real number of signatures in a list.

*Shared Link List Nodes:* In EBF, up to $r$ link list nodes are allocated for one signature since $r$ hash functions are used. Assuming each link list node is 4-byte and each signature is 32-byte, to support 2000 signatures we need roughly 144K-byte memory. Considering in Snort rule set there are almost 50 distinct signature lengths, we have the scalability issue to support multiple EBFs. By slightly modifying our algorithm, we can greatly eliminate the data structure redundancy. Ideally, if we always perform a deterministic lookup to the off-chip memory, we need only store a single list node for a signature. However, we want to support the fast incremental updates, and the simple and unified control operations to EBF, so we introduce a scheme that is not optimal but still significant in memory reducing.

When we program a signature in EBF, we need update $r$ EBF buckets. We always insert the signature at the head of the list. For all the buckets that have not been set by any other signature, we only allocate one list node shared by these buckets. Since we already know that most of lists in EBF only include one node, this scheme can significantly reduce the memory consumption compared with the original one. The formal description of the improved algorithm is shown in Figure 7

```
Procedure Programming_EBF_new(S)
  Allocate memory space for S at address m;
  Store S at address m;
  token = 1;
  for i in 1 to r do
    k = Hi(S);
    EBF[k].counter ++;
    if (EBF[k].hit == 1)
      Allocate memory space for a link list node
LN at address j;
      LN.string = m;
      LN.next = EBF[k].pointer;
      EBF[k].pointer = LN
    else
      EBF[k].hit = 1;
      if (token == 1)
        Allocate memory space for a link list node
LN at address j;
        token = 0;
      LN.string = m;
      LN.next = Null;
      EBF[k].pointer = LN;
```

Fig. 7. Improved EBF Programming Algorithm

In Figure 8, we use 10 hash functions and 20K EBF buckets to test our improved algorithm. Simulation shows 63% to 90% memory saving and the saving is more significant when the number of signatures is smaller. However, the absolute saving becomes bigger and bigger when the number of signatures increases.

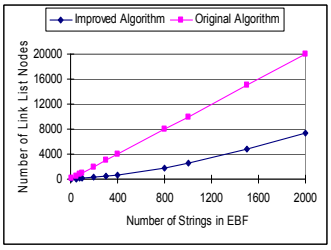


Fig. 8. Memory Saving with Improved Algorithm

Other optimization is also possible. Since we only store each signature once by using exogenous link lists, this doubles the number of memory accesses to retrieve a signature. To improve the lookup performance, we can save a short hash digest in each link list node to avoid the unnecessary comparison with the long signature itself by comparing the hash digest first.

## VII. Long Signature Matching

The signature lengths distribute in a large range from a few bytes to hundreds. In a parallel computing environment, it is infeasible to maintain an EBF for each signature length. Based on the system resource availability, we set a threshold $t$. Any signature longer than $t$ should be segmented into a set of substrings with length of $t$ except the last substring that the

length is possible shorter than $t$. Two more bits are needed to store along with each substring in off-chip memory. One bit indicates if this substring is a partial match, the other bit indicates if this partial match substring is the first segment. We also maintain two other tables. One is the Partial Match Table (PMT) with $t \times t$ bytes. One extra bit for each entry indicates if this entry is valid. The other table is the Concatenate Verification Table (CVT) that stores all the combinations of two segments that form the substrings of the signatures. One extra bit indicates if this is the last two segments. CVT could be implemented using hash table or CAM.

From the start of payload scan, a pointer $p$ is calculates as $(\#\ of\ bytes\ scanned)\%\ t$. Table I shows the algorithm for long signature matching.

TABLE I
LONG SIGNATURE MATCHING ALGORITHM

```
if a string is partial and the first segment
    register it in entry p
else if a string is partial but not the first segment
    if the PMT entry p is valid
        concatenate this string with content in entry p
        lookup the CVT
        if found in CVT
            if indicate this is the last block
                report match
            else
                replace the old entry in PMT with this string
        else
            drop it
            invalidate the entry p
    else
        drop it
```

For Snort rule set, if we set $t = 24$, then PMT is only 576 bytes and the CVT needs to store only about 200 48-byte substrings. We need only maintain less than 24 EBFs instead of more than 50.

## VIII. IMPLEMENTATION

Network Intrusion Detection Systems need to inspect the packets based on both the header and payload. A hardware-based NIDS, taking the advantages of the stat-of-art FPGA technology, is proposed to implement the Snort rule matching. Figure 9 illustrates the high-level overview of the system. The layered protocol wrappers are used to handle the TCP/IP header and maintain the flow state [11] and the module of packet header classification is used to inspect the packet header and help to match the full rules [13]. We implement our data structure and lookup algorithm in string matching module to identify the signatures.

An FPGA chip can contain more than 10-Mbits SRAMs, configured in different depth and width [1]. We implement the front-end EBF data structure on-chip by taking advantage of these flexible multi-port SRAMs.

The link lists and signatures are stored in off-chip memory. The memory management logic dynamically allocates the memory for them. The operation takes several clock cycles. This extra complexity only happens when we initialize or update the signatures in EBF. As we know, the database updating is relatively infrequent compared with the signature matching task, so this is not a serious concern.

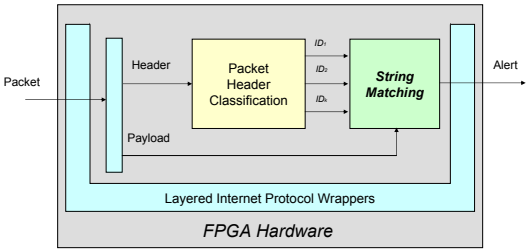


Fig. 9. FPGA-based NIDS

## IX. CONCLUSION

In this paper, we present a new data structure that extends the classical Bloom Filter. We propose the corresponding algorithms for hardware-based multi-pattern signature matching. We also present a scheme to support long signature matching. We demonstrate that our EBF algorithm is highly efficient in terms of both throughput and memory storage. We integrate this algorithm into a full-featured hardware-based NIDS. We believe this is a key step towards building a real time NIDS, which is capable of actively monitoring and filtering all pass-through traffic even in a high-speed network.

## REFERENCES


[1] Xilinx Inc. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, 2004.

[2] A. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, June 1975.

[3] M. Attig, S. Dharmapurikar, and J. Lockwood. Implementation of bloom filters for string matching. In *FCCM*, 2004.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, July 1970.

[5] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster pattern matching for intrusion detection or exceeding the speed of snort. In *DISCEX II*, 2001.

[6] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood. Deep Packet Inspection Using Parallel Bloom Filters. In *HotI*, 2003.

[7] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, Mar. 2000.

[8] M. Fisk and G. Varghese. Applying fast string matching to intrusion detection, 2004.

[9] E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis. Exclusion-based signature matching for intrusion detection. In *CCN*, 2002.

[10] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an Internet firewall. In *FCCM*, 2003.

[11] D. V. Schuehler, J. Moscola, and J. W. Lockwood. Architecture for a Hardware Based, TCP/IP Content Scanning System. In *HotI*, 2003.

[12] R. Sidhu and V. K. Prasanna. Fast regualr expression matching using fpgas. In *FCCM*, 2001.

[13] H. Song and J. Lockwood. Efficient Packet Classification for Network Intrusion Detection Using FPGA. In *ACM FPGA*, 2005.

[14] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE Infocom*, 2004.

[15] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. *Technical Report TR-94-17*, 1994.

[16] F. Yu, R. Katz, and T. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM.