
DEEP PACKET INSPECTION USING PARALLEL BLOOM FILTERS

BECAUSE CONVENTIONAL SOFTWARE-BASED PACKET INSPECTION ALGORITHMS HAVE NOT KEPT PACE WITH HIGH-SPEED NETWORKS, INTEREST HAS TURNED TO USING HARDWARE TO PROCESS NETWORK DATA QUICKLY. STRING SCANNING WITH BLOOM FILTERS CAN SCAN ENTIRE PACKET PAYLOADS FOR PREDEFINED SIGNATURES AT MULTI-GIGABIT-PER-SECOND LINE SPEEDS.

..... There is a class of packet processing applications that inspect packets deeper than the protocol headers to analyze content. For instance, network security applications must drop packets containing certain malicious Internet worms or computer viruses carried in a packet payload. Content-based billing systems analyze media files and bill the receiver based on the material transferred over the network. Content forwarding applications look at the hypertext transport protocol headers and distribute the requests among the servers for load balancing.

Most payload scanning applications have a common requirement for string matching. For example, the presence of a string of bytes (or a *signature*) can identify the presence of a media file. Well-known Internet worms such as Nimda, Code Red, and Slammer propagate by sending malicious executable programs identifiable by certain byte sequences in packet payloads. Because the location (or *offset*) of such strings in the packet payload and their length is unknown, such applications must be able to detect strings of different lengths starting at arbitrary locations in the packet payload.

Packet inspection applications, when deployed at router ports, must operate at wire

speeds. With networking speeds doubling every year, it is becoming increasingly difficult for software-based packet monitors to keep up with the line rates. These changes have underscored the need for specialized hardware-based solutions that are portable and operate at wire speeds.

We describe a hardware-based technique using Bloom filters, which can detect strings in streaming data without degrading network throughput.¹ A Bloom filter is a data structure that stores a set of signatures compactly by computing multiple hash functions on each member of the set. This technique queries a database of strings to check for the membership of a particular string. The answer to this query can be false positive but never a false negative. An important property of this data structure is that the computation time involved in performing the query is independent of the number of strings in the database provided the memory used by the data structure scales linearly with the number of strings stored in it. Furthermore, the amount of storage required by the Bloom filter for each string is independent of its length.

Our hardware implementation groups signatures according to their length (in bytes)

**Sarang
Dharmapurikar**
**Praveen
Krishnamurthy**
Todd S. Sproull
John W. Lockwood
Washington University
in St. Louis

and stores each group of strings in a unique Bloom filter. Each Bloom filter scans the streaming data and checks the strings of corresponding length. Whenever a Bloom filter detects a suspicious string, an *analyzer* probes this string to decide whether it indeed belongs to the given set of strings or is a false positive.

Here, we present the architecture of a hardware-based Bloom filter implementation and analyze its performance. The analysis shows that an implementation with field-programmable gate arrays (FPGAs) can support multi-gigabit per second line speeds while scanning for more than 10,000 strings.

Bloom filter theory

A Bloom filter is a randomized data structure that can represent a set of strings compactly for efficient membership querying. Given string X , the Bloom filter computes k hash functions on it, producing k hash values ranging from 1 to m . The filter then sets k bits in an m -bit vector at the addresses corresponding to the k hash values. This procedure repeats for all members of the set; this entire process is called the *programming* of the filter.

The query process is similar to programming; the filter takes as an input a string for membership verification. For this new string, the Bloom filter generates k hash values, using the same hash functions used in programming. The filter then looks up the bits in the m -bit vector at the locations corresponding to the k hash values. If it finds at least one of these k bits unset, then it declares the string to be a nonmember of the set. If it finds that all the bits are set, it declares the string to belong to the set with a certain probability. This uncertainty in the membership comes from the fact that those k bits in the m -bit vector could have been set by any of the n members. Thus finding a bit set does not necessarily imply that the particular string being checked set the bit. However, finding a bit not set certainly implies that the string does not belong to the set, since if it did then all the k bits would definitely have been set during the Bloom filter's programming with that string. This explains the presence of false positives in this scheme and the absence of any false negatives. False positive probability f is

$$f = (1 - e^{-nk/m})^k \tag{1}$$

where n is the number of strings programmed into the Bloom filter.¹ We can reduce the value of f by choosing appropriate values of m and k for given size n of the member set. It is clear that the value of m must be quite large compared to the size n of the string set. Also, for a given ratio of m/n , we can reduce the false-positive probability by increasing the number of hash functions, k . In the optimal case, which minimizes false-positive probability with respect to k ,

$$k = (m/n) \ln 2 \tag{2}$$

This corresponds to a false positive probability ratio of

$$f = (1/2)^k \tag{3}$$

We can interpret ratio m/n as the average number of bits consumed by a single member of the set. This space requirement is independent of the actual size of the member. In the optimal case, the false-positive probability decreases exponentially with a linear increase in ratio m/n . This also implies that the number of hash functions, k , and hence the number of random lookups in the bit vector required to query one membership, are proportional to m/n .

Counting Bloom filters

One property of Bloom filters is that it is impossible to delete a member stored in the filter. Deleting a particular entry requires setting the corresponding k hashed bits in the bit vector to zero. This could disturb other members programmed into the filter that hash into any of these bits. To solve this problem, Fan et al. proposed the idea of *counting Bloom filters*, which maintain a vector of counters corresponding to each bit in the bit vector.² Whenever a counting Bloom filter adds or deletes a member, it increments or decrements the counters corresponding to the k hash values. When a counter changes from zero to one, it sets the corresponding bit in the bit vector. When a counter changes from one to zero, it clears the corresponding bit in the bit vector.

So, these counters change only during the addition and deletion of strings in the Bloom filter. For applications such as network intrusion detection, these updates are less frequent than for the actual query process itself. Hence,

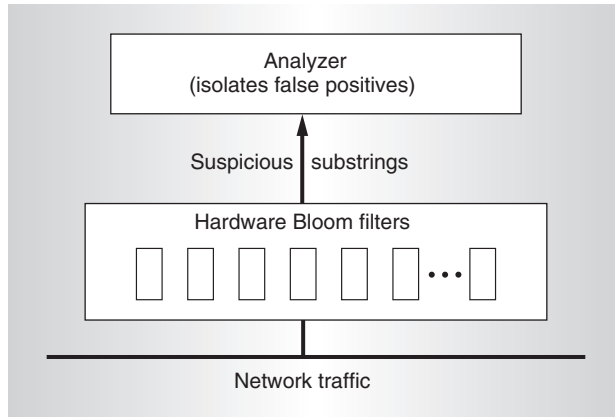


Figure 1. Bloom filters scan all traffic on a multigigabit network, looking for predefined signatures.

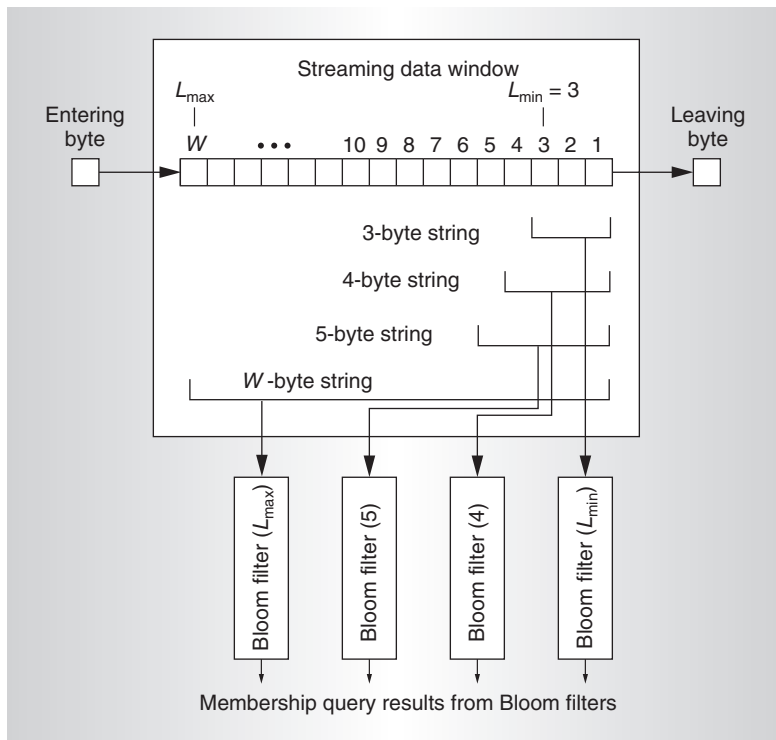


Figure 2. Window of streaming data containing strings of lengths from $L_{\min} = 3$ to $L_{\max} = W$. A Bloom filter examines each string.

an implementation can maintain counters in software, while the bit corresponding to each counter remains in hardware. Avoiding counter implementation in hardware saves memory resources.

System description

The previous section explained how the amount of computation required to detect a

string's membership in a Bloom filter is independent of the size of the set of strings once you tune the filter to a particular false positive probability. Secondly, special-purpose hardware (such as FPGAs) can perform the computation, which requires generating hash values. Hence, hardware implementation of Bloom filters is quite attractive for quickly isolating the potentially matching strings from the streaming network data.

System overview

This system relies on a predefined set of signatures grouped by length and stored in a set of parallel Bloom filters in hardware. Each Bloom filter contains signatures of a particular length. The system uses these Bloom filters to monitor network traffic and operate on strings of the corresponding length from network data, as Figure 1 shows. The system tests each string for membership in the Bloom filters. If it identifies a string to be a member of any Bloom filter, the system then declares the string as a possible matching signature. Such strings receive further probing by an *analyzer*, which determines if the string is indeed a member of the set or a false positive. Stated differently, the analyzer is a deterministic string-matching algorithm that verifies if the input string is a member of a given set or not. Based on the analyzer's determination, the system can take appropriate action (either drop, forward, or log) for the string's associated packet.

Let the signature lengths range from L_{\min} to L_{\max} . The Bloom filter engine reads as input a data stream that arrives at the rate of one byte per clock cycle. It monitors a window of L_{\max} bytes, as shown in Figure 2.

When this window is full, it contains $L_{\max} - L_{\min}$ substrings, which are potential matches for signatures. The system verifies the membership of each substring, using the appropriate Bloom filter. Each hardware Bloom filter gives one query result per clock cycle. In this way, the system can verify the memberships of all the $L_{\max} - L_{\min}$ strings in a single clock cycle. If none of the substrings match a signature, the data stream can advance by a byte. Monitoring a window in this way eventually scans all the possible strings of length from L_{\min} bytes to L_{\max} bytes in every packet. In the case of multiple substrings matching within a single window, the longest substring becomes the string of interest, a pol-

icy called longest substring first (LSF). Thus, in the case of multiple matches at the same time in the array of Bloom filters, the analyzer probes the substrings, from longest to shortest. The search stops as soon as the analyzer first confirms the match of a substring. After the search is over, the window advances by a byte, and the system repeats the same procedure.

System throughput

Bloom filters accelerate string matching by isolating most of the strings from the network data and sending just those strings with the highest probability of matching to the analyzer. A string of interest never goes unnoticed because Bloom filters never give false negatives. Now we derive an expression for the system's statistical throughput, using the following notation:

- τ , the time (in seconds) required to check the presence of a string using the analyzer;
- f , the false-positive probability of each Bloom filter;
- B , the total number of Bloom filters in the system; and
- F , the clock frequency (in Hz) at which the system operates.

Within a window, it is possible that multiple Bloom filters identify matches corresponding to their substrings. For a search that ends at the l th Bloom filter, let B_l denote the number of Bloom filters for lengths higher than l . Because each of the B_l Bloom filters can show a false match with probability f , the expected number of false matches, E_b , can be given by

$$E_i = B_l f$$

Each match requires an additional probing by the analyzer. Thus, the preceding expression represents the expected number of *additional* probes, performed by the analyzer, when the search ends at l th Bloom filter. In the worst case, $B_l = B$, hence value of E_i is upper bounded by Bf . Other calculations use this upper bound on the expected number of additional probes. Each of these probes requires time τ in the worst case, the expected additional time spent in probing is

$$T_{\text{add}} = Bf\tau$$

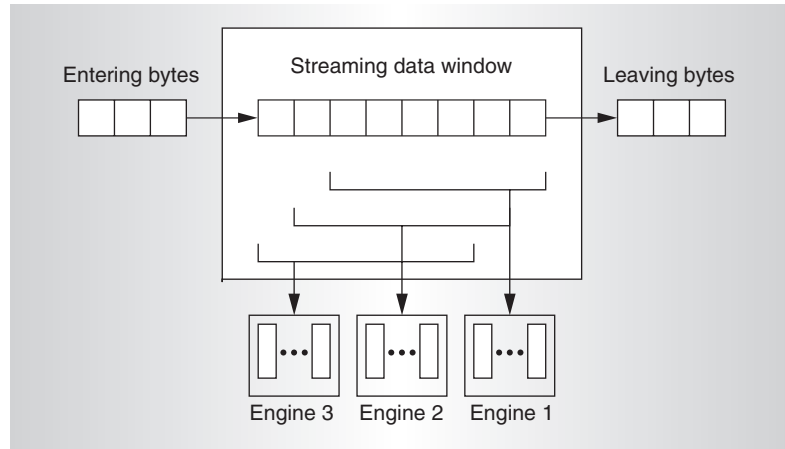


Figure 3. Using multiple parallel engines to improve throughput.

Because the search ends at Bloom filter l , if it identifies a match, it has found a *true* match; otherwise, there are no Bloom filters for string lengths less than l that have a match in the given window. In the worst case, the analyzer will spend time τ to confirm a true match. If no match is identified then no probing is needed and a time equal to the clock period, $1/F$ will be spent. If we use p to denote the frequency of occurrence for a true matching string in the data stream, on average, the time spent during the end of the search within a window is

$$T_{\text{end}} = p\tau + (p - 1)(1/F)$$

Thus, on an average, the system spends a total of $T_{\text{add}} + T_{\text{end}}$ in examining a window, after which the window advances by a byte. Hence, system throughput R is

$$R = 1/(T_{\text{add}} + T_{\text{end}}) = 1/[Bf\tau + p\tau + (p - 1)(1/F)]$$

The system shown in Figure 2 processes one byte every clock cycle. If we group the set of Bloom filters in a single scanner engine, we can instantiate multiple such engines to monitor the data stream, each starting with an offset of one byte. Thus, if we use three such engines, the byte stream can advance three bytes at a time, as shown in Figure 3.

If each of the G parallel engines has its own independent analyzer circuit, then the throughput is simply GR . If they share the same analyzer, then throughput will be dif-

ferent because of contention in accessing the analyzer. In this case, throughput becomes

$$R = G / (GT_{\text{add}} + T_{\text{end}}) = 1 / [GBf\tau + p\tau + (p-1)(1/F)] \quad (4)$$

with the assumption that only one of the G engines finds a true match in a given window.

Tuning system parameters

We can simplify Equation 4 by considering realistic values of relevant parameters. Recall that we assume the analyzer to require constant time τ to check the input string in the database. Such an analyzer is implemented as a hash table. A set of strings is inserted into a hash table with collisions resolved by chaining the colliding strings together in a linked list. Such a hash table has an average of constant search time³ and is storable in an off-chip, commodity SRAM. Although the use of ample memory can make the average search time in such a hash table independent of the number of strings, the string retrieval time from that memory depends on the string's length. Assume we have a 32-byte string probed in a hash table maintained in SRAM with a data bus width of 4 bytes. It will take 8 clock cycles to retrieve the string and compare it against the input. With L_{max} set to 32, even with an assumption of one collision and accounting for memory access latencies, a hash probe should require no more than 20 clock cycles. Hence, we use $\tau = 20/F$; that is, 20 times the system clock period. Typically, the strings of interest have a low frequency of occurrence in the streaming data, so we can assume small values of p , such as the 0.001 we use for the following example. That is, on average, the system finds one string of interest for every thousand characters scanned.

We further assume the following values: $B = 24$ (indicating a system that can scan signatures with 24 distinct lengths), $F = 100$ MHz (a typical speed for FPGAs and commodity SRAMs), and $G = 4$ (the use of four Bloom filter engines in parallel). Substituting these values in Equation 4 we obtain the following expression for the throughput:

$$R_i = 3.2 / (1,920f + 1.019)$$

To express the value of f , we use the following notation:

- f_i is the false-positive probability of the i th Bloom filter within an engine,
- m_i is the memory allocated to Bloom filter i ,
- n_i is number of strings stored in Bloom filter i ,
- M is the total amount of on-chip memory available for the Bloom filters of all G engines (each engine has M/G memory, shared by the B Bloom filters in it), and
- N is the total number of strings stored in the B Bloom filters of an engine; thus, N is the summation of n_i , from $i = 1$ to B .

Because we engineer the false-positive probability of all the Bloom filters of an engine to be the same, say f using Equation 3,

$$f_i = f = (1/2)^{(m_i/n_i)\ln 2}, \forall i \in [1 \dots B]$$

This implies that

$$m_1/n_1 = m_2/n_2 \dots m_B/n_B = \sum m_i / \sum n_i = (M/G)/N$$

Therefore,

$$f = (1/2)^{[(M/G)/N]\ln 2}$$

After substituting the value of f into the expression for R_i and plotting throughput R_G for a total of $N = 10,000$ strings, we obtain the graph in Figure 4.

As the figure shows, the effect of false positives is dominant for small values of memory, which results in a lower throughput. However, as the amount of memory increases, the throughput increases rapidly and saturates at over 3 Gbps. Thus, with merely 1 megabit of on-chip memory, this system can scan 10,000 strings at the line rate of OC-48 (2.4 Gbps). Moreover, we can increase the number of strings with a proportional increase in the memory.

Hardware design considerations

Equation 2 shows that for a fixed number of strings in a Bloom filter, the number of bits

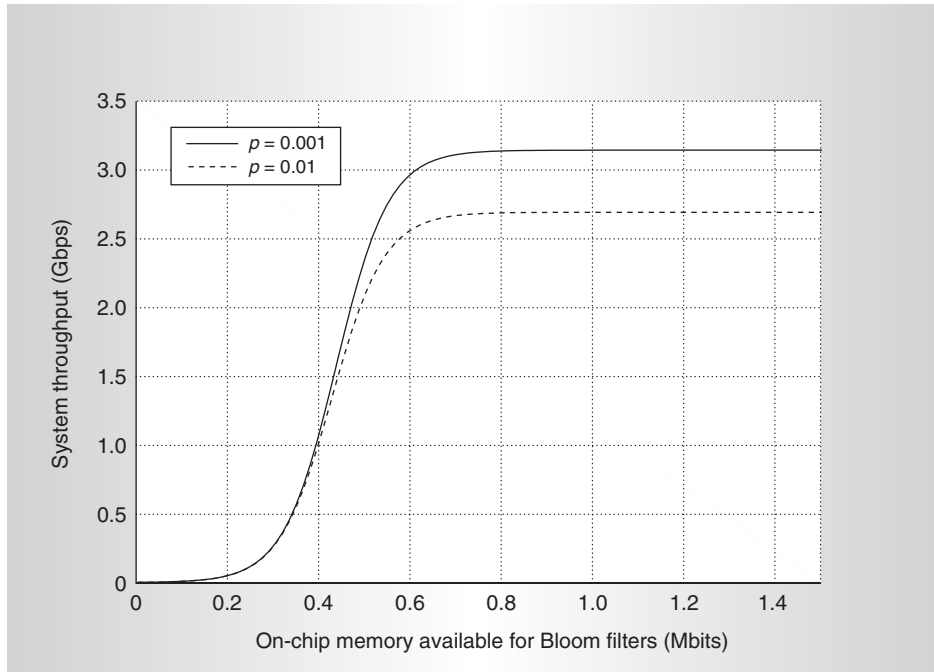


Figure 4. System throughput as a function of the available on-chip memory. The graph shows results for two values of p , the probability of a string's true occurrence of strings. We tuned the system for a total of $N = 10,000$ strings, each of $B = 24$ distinct lengths. The maximum string probing time in the analyzer is 20 times the system's clock period; clock frequency $F = 100$ MHz.

allocated to a member in a Bloom filter also decides the number of hash functions needed for that Bloom filter. For example, if we allocate 50 bits per member on an average ($m/n = 50$), then number of hash functions must be $k < 50 \times 0.7 = 35$ and the false positive probability is $(1/2)^{35} < 3 \times 10^{-11}$. We next describe how to support these hash functions in hardware and perform the corresponding random lookups in on-chip memories.

Hash functions

Ramakrishna, Fu, and Bahcekapili described a class of universal hash functions that are suitable for this hardware implementation.⁴ To generate the necessary k hash functions, we perform the following calculations. For any bit string X with b bits represented as

$$X = \langle x_1, x_2, x_3, \dots, x_b \rangle$$

we calculate the i th hash function over X , $h_i(X)$ as

$$h_i(X) = d_{i1} \cdot x_1 \oplus d_{i2} \cdot x_2 \oplus d_{i3} \cdot x_3$$

$$\oplus \dots \oplus d_{ib} \cdot x_b$$

where “ \cdot ” is a bitwise AND operator and \oplus is a bitwise XOR operator. The d_{ij} terms are predetermined random numbers in the range $[0 \dots m - 1]$. We observe that the hash function calculations are cumulative and hence the results calculated over the first i bits are reusable in calculating the hash function over the first $i + 1$ bits. This property of the hash functions results in a regular and less resource-consuming hash function matrix.

Using multiport embedded memories

Each hash function corresponds to one random lookup in the m -bit long memory array. Thus, for 35 hash functions, the Bloom filter memory should support 35 random lookups every clock cycle. Figure 5a illustrates these requirements graphically. We realize memories with such a density and lookup capacity by making use of the embedded RAMs in modern FPGA devices or VLSI chips. With today's state-of-the-art VLSI technology, it is easy to fabricate memories

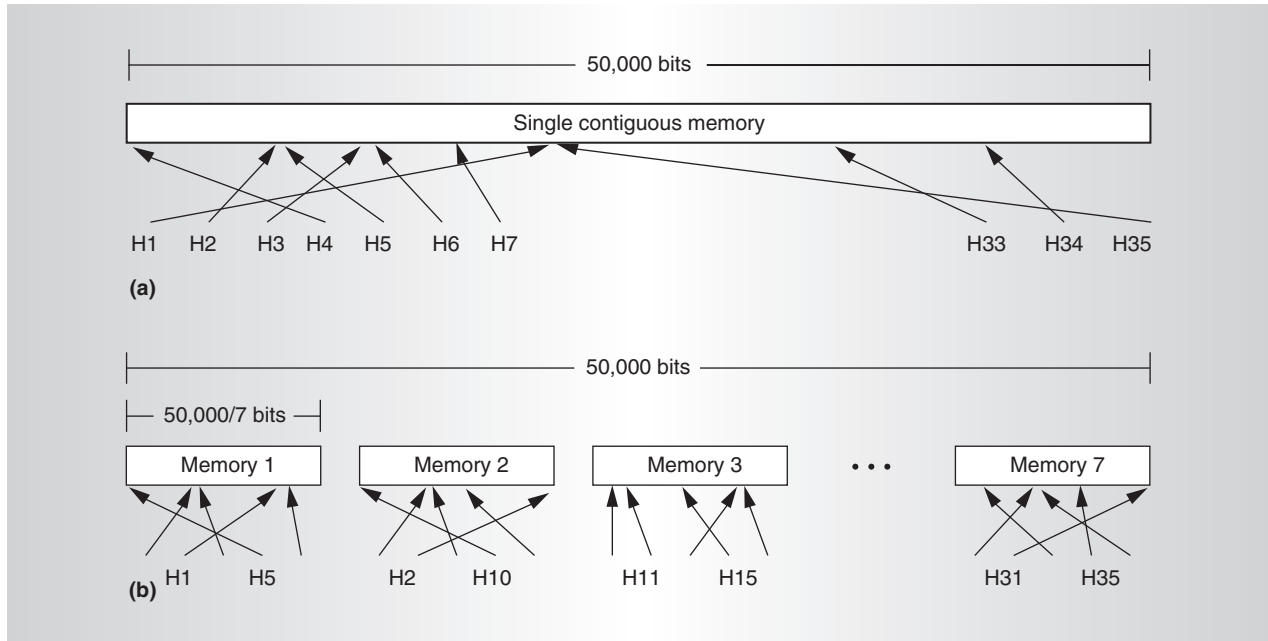


Figure 5. Bloom filter with single memory vector that allows 35 random lookups at a time (a) and an implementation using multiple smaller memories each with a smaller lookup capacity of 5 to realize the desired lookup capacity of 35 (b).

that hold a few million bits. Dipert gives a concise summary of the recent embedded memory technologies.⁵

For embedded memories with limited lookup capacity, it is possible to employ multiple memories with smaller lookup capacity. For instance, Lucent's memory core has five read-write ports.⁵ Hence, using this memory core, five random-memory locations are readable in a single clock cycle. So performing 35 concurrent memory operations requires seven parallel memory cores, each with one-seventh of the required array size, as Figure 5b illustrates. Because the basic Bloom filter allows any hash function to map to any bit in the vector, it is possible that for some member, more than five hash functions map to the same memory segment, thereby exceeding the lookup capacity of this memory core. We can solve this problem by restricting the range of each hash function to a given memory, preventing memory contention.

In general, if h is the maximum lookup capacity of a RAM as limited by the technology, then we can combine k/h such memories, each of size $m/(k/h)$, to realize the desired capacity of m bits and k hash functions. Only h hash functions can map to a single memory. We can express the false-positive probability as

$$f' = \left\{ 1 - \left[1 - \frac{1}{m/(k/h)} \right]^{hm} \right\}^{(k/h)h}$$

$$\approx \left(1 - e^{-nkl/m} \right)^k$$

Comparing the preceding equation with Equation 1, we see that restricting the number of hash functions mapping to a particular memory has negligible effect on the false-positive probability.

Prototype implementation and results

We implemented a prototype system in an Xilinx XCV2000E field-programmable gate array (FPGA), using the Field-Programmable Port Extender (FPX) platform.⁶ We implemented an application to find fixed-size signatures (hence $B = 1$) of 32 bytes to detect the transfer of high volumes of content over a network.

The XCV2000E has 160 embedded block memories, each of which is configurable as a single bit wide, 4,096-bit long array that can perform two read operations (using dual ports) in a single clock cycle. We used this memory to construct a Bloom

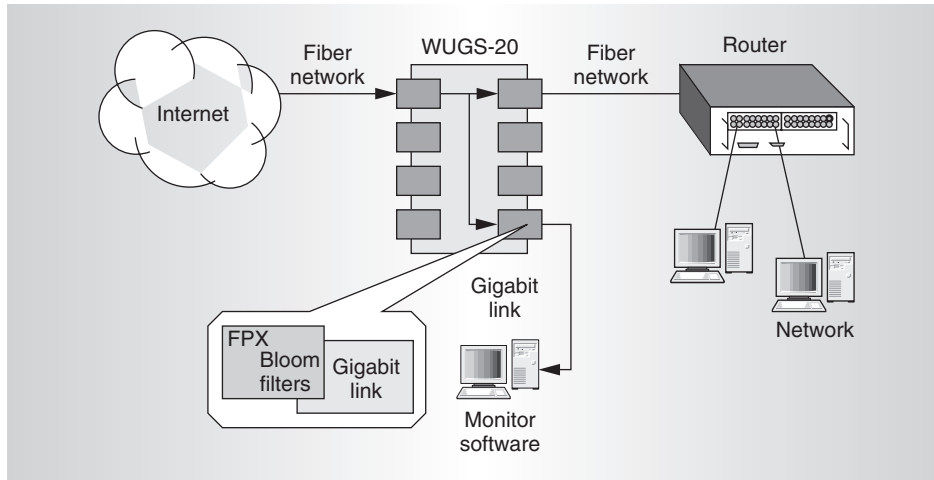


Figure 6. Deployment setup for the prototype.

filter, with $m = 4,096$ and $k = 2$. Using Equation 2 and Equation 3, we see that this block RAM can support $n = (m/2) \times \ln 2 < 1,419$ signatures with a false-positive probability of $(1/2)^2 = 0.25$. By employing five such block RAMs, we can construct a mini-Bloom filter with a 1,434 string capacity and false-positive probability of $f = (1/2)^{10}$. Using 35 block RAMs, we can construct seven such mini-Bloom filters, giving an aggregate capacity of $1,419 \times 7 = 10,038$ strings. These mini-Bloom filters constitute one engine. A system instantiating four parallel engines (which together consume $35 \times 4 = 140$ block RAMs) can push 4 bytes in a single clock cycle (hence, $G = 4$). Substituting these values into Equation 4, we see that the system can achieve a throughput of more than 2.46 Gbps (an OC-48 line rate).

Our functional prototype implements a single Bloom filter engine, consuming 35 block RAMs and only 14 percent of the FPGA's available logic resources. The system operates at 81 MHz. Figure 6 shows a deployment setup for this system. Internet traffic passes through WUGS-20, a 20-Gbps version of the Washington University gigabit switch, which multicasts the data to an FPX and to a router. The router contains a fast Ethernet switch connected to workstations. Data from the workstations pass to the router then to the Internet through the WUGS-20. Traffic coming from the Internet to the router goes to the FPX for processing. A software process replaced the analyzer in a stand-alone work-

station; the software checks all the packets marked as a possible match by the Bloom filters in the FPX.

We performed experiments to observe the practical performance of Bloom filters in terms of false-positive rate, programming the filter with different numbers of strings and counting the false positives. Figure 7 shows the result of this experiment along with the theoretical value. This plot shows that the experimental results are consistent with theoretical predictions. In our experiments, the system did not produce any false positives when using fewer than 1,400 strings to program the filters (that's approximately 200 strings in each mini-Bloom filter). This produces a dip in the curve at that point.

To determine throughput for this particular prototype configuration, we sent traffic to the WUGS-20 switch at a fixed rate and then recycled it in the switch to generate traffic at speeds of more than 1 Gbps. Using a single match engine, the circuit scanned data at the rates up to 600 Mbps.

Related work

Coit et al.⁷ explore the benefits of using the Aho-Corasick Boyer-Moore (AC_BM) algorithm to improve the performance of Snort.⁸ This algorithm is faster than the Boyer-Moore algorithm used by the current version of the Snort engine. Varghese and Fisk analyze a set-wise implementation of the Boyer-Moore algorithm;⁹ this implementation has an average-case performance that is better than that

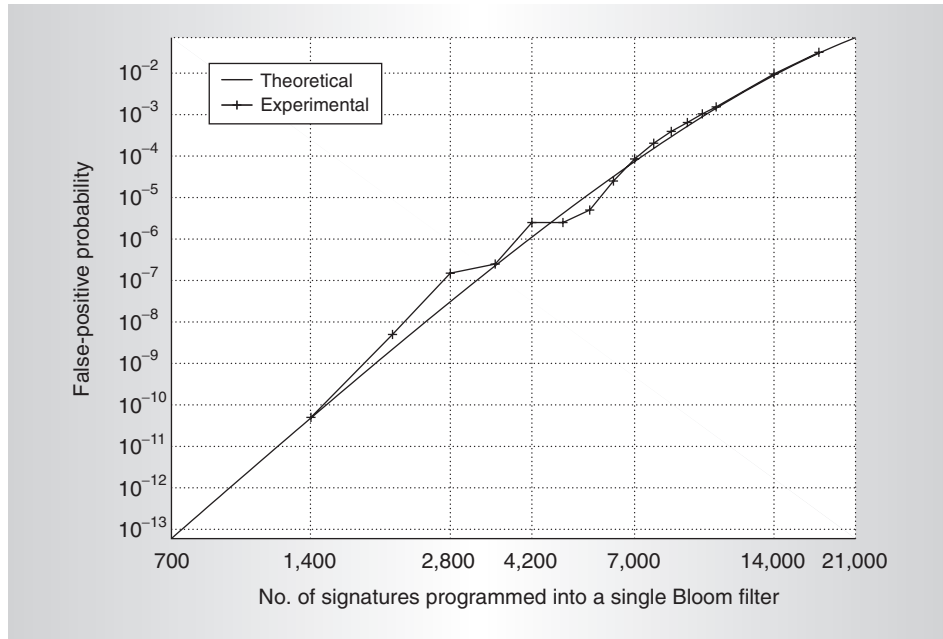


Figure 7. False-positive probability as a function of the number of signatures stored in one Bloom filter engine.

of the Aho-Corasick algorithm. These algorithms are primarily geared toward software implementation. Commercial hardware implementations of packet content inspection ICs are available; these include the Solidum Pax.port, NetScreen 5000, and the PMC-Sierra ClassiPI. Few details about these proprietary systems are available, however.

The advent of the modern reconfigurable hardware technology, particularly FPGAs, has added a new dimension to hardware-based packet inspection techniques. Literature shows that new approaches using reconfigurable hardware essentially involve building an automaton for a given search string, generating a specialized hardware circuit using gates and flip-flops for this automaton, and then instantiating multiple such automata in the reconfigurable chip to search the streaming data in parallel.¹⁰⁻¹² The common characteristic of these approaches is that the on-chip hardware resource consumption (gates and flip-flops) grows linearly with the number of search characters. Secondly, these methods require reprogramming of the FPGA to add or delete individual strings from the database. Any change in the database requires recompilation, automaton regeneration, and a repeated synthesis, placement, and routing of the circuits.

In contrast, Bloom filter-based systems can handle a larger database with reasonable resources, and support fast updates to the database. The latter is an important feature in a network intrusion detection, which requires an immediate response to attacks, such as those by malicious worms.

We have implemented a system that can detect the presence of 10,000 predefined strings in Internet packet payloads. By using parallel Bloom filters implemented on an FPGA, we scanned for content at multi-gigabit-per-second speeds. We've proven the feasibility of this idea with our experimental implementation that runs on the FPX platform.

MICRO

Acknowledgments

This research was supported by a grant from Global Velocity. John Lockwood is a cofounder and consultant for Global Velocity and an assistant professor at Washington University in St. Louis. The authors of this article have equity and can receive royalty from a license of this technology to Global Velocity.

References

1. B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, May 1970, pp. 422-426.
2. L. Fan et al., "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking*, vol. 8, no. 3, June 2000, pp. 281-293.
3. T.H. Corman et al., *Introduction to Algorithms*, Prentice Hall, 2002.
4. M. Ramakrishna, E. Fu, and E. Bahcekapili, "A Performance Study of Hashing Functions for Hardware Applications," *Proc. 6th Int'l Conf. Computing and Information*, 1994, pp. 1621-1636.
5. B. Dipert, "Special Purpose SRAMs Smooth the Ride," *EDN*, June 1999.
6. J.W. Lockwood et al., "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," *Proc. ACM Int'l Symp. Field Programmable Gate Arrays (FPGA 01)*, ACM Press, 2001, pp. 87-93.
7. J. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *Proc. 2nd DARPA Information Survivability Conference and Exposition (DIS-CEX II)*, IEEE CS Press, 2001, pp. 367-373.
8. M. Roesch, "SNORT—Lightweight Intrusion Detection for Networks," *Proc. 13th Systems Administration Conf.*, Usenix Assoc., 1999, pp. 229-238.
9. M. Fisk and G. Varghese, *Fast Content-Based Packet Handling for Intrusion Detection*, tech. report CS2001-0670, Univ. of California, San Diego, 2001.
10. J. Moscola et al., "Implementation of a Content-Scanning Module for an Internet Firewall," *Proc. 11th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 03)*, IEEE CS Press, 2003, pp. 31-38.
11. B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proc. 10th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 02)*, IEEE CS Press, 2002, pp. 111-120.
12. R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," *Proc. 9th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 01)*, IEEE CS Press, 2001.

Sarang Dharmapurikar is a PhD student in the Department of Computer Science and Engineering, Washington University in St. Louis. His research interests include various aspects of high-speed networking system design, including packet classification and forwarding; payload inspection and intrusion detection; and packet buffering and queuing. Dharmapurikar has a BS in electrical and electronics engineering from the Birla Institute of Technology and Science, Pilani, India.

Praveen Krishnamurthy is a PhD student at Washington University in St. Louis. His research interests include networking systems design, data mining, and optical interconnection networks. Krishnamurthy has a bachelor's degree in engineering from the University of Madras, India, and an MS in computer engineering from Washington University in St. Louis.

Todd S. Sproull is a PhD student in computer engineering and a research assistant at the Applied Research Laboratory, Washington University in St. Louis. His research interests include the control and configuration of reprogrammable systems, and distributed computing. Sproull has a BS in electrical engineering from Southern Illinois University, Edwardsville, and an MS in computer engineering from Washington University in St. Louis.

John W. Lockwood is an assistant professor at Washington University in St. Louis. His research interests include the design and implementation of networking systems in reconfigurable hardware. He developed the Field-Programmable Port Extender (FPX) to enable rapid prototyping of extensible network modules. Lockwood has a BS, an MS, and a PhD from the University of Illinois, Urbana-Champaign. He is a member of the IEEE, the ACM, Tau Beta Pi, and Eta Kappa Nu.

Direct questions and comments about this article to Sarang Dharmapurikar; sarang@arl.wustl.edu. Additional information about the project is online at <http://www.arl.wustl.edu/arl/projects/fpx/reconfig.htm>.