Efficient Byte Stream Pattern Test using Bloom Filter with Rolling Hash Functions on the FPGA

Takuma Wada, Naoki Matsumura, Koji Nakano and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The main purpose of this paper is to present an efficient FPGA implementation for the Bloom filter, in which a large set P of l-byte patterns are registered beforehand. Our Bloom filter circuit performs the byte stream pattern test such that it receives an input byte stream t and outputs the bit stream in every clock cycle. Each bit of the output bit stream is 1 if an l-byte sequence of t starting from the corresponding position is identical with one of the patterns in P. Such byte stream pattern test has a lot of applications. For example, it can be used for detecting malicious patterns in byte stream of network traffic. Our Bloom filter circuit fully utilizes 288K-bit Ultra RAMs and 18K-bit Block RAMs in the Xilinx UltraScale+ VU9P FPGA. We use Ultra RAMs to implement bit arrays to register all patterns in P and Block RAMs to compute signatures using rolling hash functions. Unlike the previously published FPGA implementations of the Bloom filter, which use XOR-based hash functions, our Bloom filter circuit using rolling hash functions can support much larger l. We have evaluated the performance of our Bloom filter circuit using Xilinx UltraScale+ FPGA VU9P, which is a popular high-end FPGA used in Amazon Web Service. The experimental results show that our Bloom filter circuit for 4800K $(= 4, 915, 200)$ patterns of length 1024 can perform the byte stream pattern test for 1.14Gbps input byte stream with false positive probability 10^{-12}. Also, we can configure our Bloom filter circuit to work for 100K $(= 102, 400)$ patterns of length 1024 and 49.5Gpbs input byte stream with the same false positive probability.**

***Index Terms*—hash function, hardware algorithm, data base, intrusion detection**

I. Introduction

A. Background

A Bloom filter [1] is a space-efficient data structure, which can be used to test if x is in P, where P is a large set of elements, and x is a query element. The Bloom filter uses a hash function f that returns an integer (or *signature*) in the range $[0, s-1]$ and a bit array B of size s initialized by zero. For every element y $(\in P)$, we write 1 in $B[f(y)]$ in advance. Clearly, for any query element x, if $B[f(x)] = 0$ then it is guaranteed that x is not in P, because $f(x) \neq f(y)$ holds for all $y \in P$. However, if $B[f(x)] = 1$ then x may or may not be in the set. Thus, this membership test of x for P using B is *false positive*. Since the membership test can be done by simply computing signature $f(x)$ and reading $B[f(x)]$, it is very efficient if the computation cost of hash function f is low. Also, the false positive probability can be any small using a larger bit array and/or multiple hash functions. A Bloom filter has a lot of applications [2], [3]. For example, it can be used to detect weak passwords [4] as follows. Let P be a set of weak passwords such as all English words. Using the Bloom filter for P, we can determine if a password x registered by a user is in P. We can quickly reject registration of x if it is in P. Note that, it is acceptable to reject x even if it is not in P. Further, the Bloom filter can be used for signature-based intrusion detection in the network traffic [5]–[7]. If we have a large set of malicious patterns such as malware, then the Bloom filter can be used to detect a malicious pattern from byte stream. It also can be used for Web cache control [8], [9].

A Field Programmable Gate Array (FPGA) is a programmable logic device designed to be configured by customers or designers after manufacturing. Since an FPGA chip maintains relative lower price and programmable features, it is widely used in those fields which need to update architecture or functions frequently such as image processing [10], [11] and education [12]. The most common architecture of recent FPGAs is an array of Configurable Logic Blocks (CLBs) [13], Block RAMs [14], DSP slices [15], and programmable routing channels connecting them [16]. In addition, latest FPGA has an Ultra RAM [14], which is 16 times larger than a Block RAM. Although the architecture of the latest FPGAs is targeted for high performance digital signal processing [15], [17], it can be used for other applications and general purpose computing [18]–[20].

B. Our contribution

The main purpose of this paper is to develop an efficient circuit for the Bloom filter as illustrated in Figure 1 to be implemented in the FPGA. *A byte stream* is input to the Bloom filter circuit such that a byte data in it is read in every clock cycle continuously. In the Bloom filter circuit, a set P of l-byte patterns are registered beforehand. It outputs a bit stream in every clock cycle continuously such that bit i is 1 if the l-byte interval starting from the i-th byte of the input byte stream matches one of patterns in P. In the figure, the 4th and the 6th bits are 1, because the byte stream has patterns $dabc$ and $bcdb$ from these positions, respectively.

The Bloom filter uses a large bit array B to store signatures of all patterns in P of length l. We use *rolling hash functions* to compute the signature $f(x)$ of a pattern x in P defined as

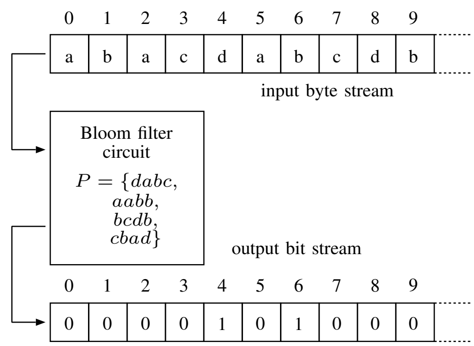

Fig. 1. A Bloom filter circuit

follows:

$$f(x) = (x_0 \cdot d^{l-1} + x_1 \cdot d^{l-2} + \cdots + x_{l-2} \cdot d^1 + x_{l-1}) \bmod q, \quad (1)$$

where q and d are integer parameters selected appropriately. We call the resulting value of rolling hash function f *signature*. For uniform distribution of signatures, q should be a prime number. Suppose that all bits in bit array B are initialized by zero and 1 is written in $B[f(y)]$ for every pattern y in P in advance. Let $t_0 t_1 \cdots$ be bytes of the input byte stream. The signature $f(t_j t_{j+1} \cdots t_{j+l-1})$ is computed for every j in turn. Clearly, if $B[f(t_j t_{j+1} \cdots t_{j+l-1})] = 0$ then $t_j t_{j+1} \cdots t_{j+l-1}$ is not in P. On the other hand, $t_j t_{j+1} \cdots t_{j+l-1}$ may not be in P, even if $B[f(t_j t_{j+1} \cdots t_{j+l-1})] = 1$. Thus, the value of $B[f(t_j t_{j+1} \cdots t_{j+l-1})]$ determines if $t_j t_{j+1} \cdots t_{j+l-1} \in P$ with some false positive probability. We can use multiple bit arrays with multiple distinct rolling hash functions so that membership is positive if all of the bit arrays return 1. Using this technique, the false positive probability can be any small.

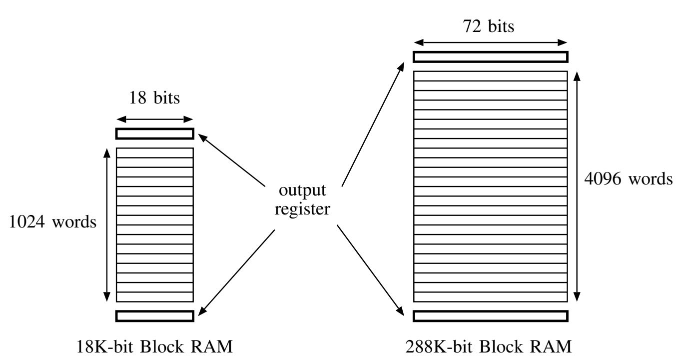

Fig. 2. Block RAM and Ultra RAM

Our target FPGA is Xilinx Ultrascale+ VU9P, a very popular high-end FPGA, which has been used by AWS (Amazon Web Service) FPGA cloud instance. This FPGA has 4,320 Block RAMs and 960 Ultra RAMs, which are on-chip memories of size 18K bits and 288K bits, respectively. Hence, it makes sense to implement bit arrays using Ultra RAMs, because larger bit arrays can support more patterns and/or can decrease the false positive probability. However, there are several difficulties of FPGA implementations for the Bloom filter using rolling hash function as follows:

- Rolling hash functions require the computation of arithmetic modulo a prime, which needs a large combinational circuit with a long critical path.
- The value of $f(t_j t_{j+1} \cdots t_{j+l-1})$ can be computed by the values of $f(t_{j-1} t_j \cdots t_{j+l-2})$, t_{j-1}, and t_{j+l-1}, but combinational circuits for this computation are quite large and have long critical paths.
- Since the capacity of an Ultra RAM is not a power of two, some non-trivial technique is necessary to fully utilize it.

To get over these difficulties, we develop the following techniques:

- We use 18K-bit block RAMs as a Look-Up-Table to compute arithmetic modulo a prime.
- We partition the input byte stream into four byte streams $t_0 t_4 t_8 \cdots$, $t_1 t_5 t_9 \cdots$, $t_2 t_6 t_{10} \cdots$, and $t_3 t_7 t_{11} \cdots$, and compute $f(t_j t_{j+4} t_{j+8} \cdots t_{j+l-4})$, $t(t_1 t_5 t_9 \cdots t_{j+l-3})$, $f(t_2 t_6 t_{10} \cdots t_{j+l-2})$, and $f(t_3 t_7 t_{11} \cdots t_{j+l-1})$. Since $f(t_j t_{j+4} t_{j+8} \cdots t_{j+l-4})$ can be computed using $f(t_{j-4} t_j t_{j+8} \cdots t_{j+l-8})$, t_{j-4}, and t_{j+l-4}, 4 clock cycles can be used for this computation. Our circuit for this computation is pipelined, and so hash functions for four byte streams can be computed using just one circuit. By combining the resulting four values for four byte streams appropriately, we have one hash function with large codomain, which can be used to determine the bit position of an Ultra RAM.
- We developed a sophisticated circuit to select one integer in 9 integers 0, 1, 2, . . ., 8 with almost equal frequency to fully utilize 288K-bit Ultra RAMs.

Table I summarizes the performance our Bloom filter circuit for various configurations, including the number of stream buffers, the number of Block RAMs, the number of Ultra RAMs as used hardware resources in the FPGA. It also shows the total number of registered patterns and the false positive probability of the Bloom filter. In the table, $\mathrm{BF}(1,1,1)$ corresponds to the minimum configuration that we call *BF engine*. It has one circuit to compute the signatures for an input byte stream using one and a half of the Block RAMs for hash function computation and a half of Ultra RAM to implement 144K-bit bit array for the Bloom filter. If we store signatures of 100K($= 102{,}400$) patterns, the false positive probability is approximately 2^{-1}. A byte stream buffer (i.e. FIFO) is attached to an BF engine, which is used to store the latest l bytes of the byte stream, because t_{j-4} and t_{j+l-4} are used to update the resulting value of the hash function. If $l <$ 2K then one 18K-bit Block RAM configured as a 2K×9 memory is sufficient to implement the byte stream buffer as a conventional ring buffer. Thus, $\mathrm{BF}(1,1,1)$ uses 1.5 Block RAMs for hash function computation and 1 Block RAM for buffering a byte stream.

If we use h BF engines, the false positive probability is decreased to 2^{-h}. This configuration corresponds to $\mathrm{BF}(1,h,1)$

TABLE I
THE PERFORMANCE OF OUR BLOOM FILTER CIRCUITS

circuit configuration	# Block RAMs out of 4320	# Ultra RAMs out of 960	# byte streams	# patterns	false positive probability
$\mathrm{BF}(w,h,p)$	$1.5hw+w$	$0.5whp$	w	100K$\times p$	2^{-h}
$\mathrm{BF}(1,1,1)$	1.5+1	0.5	1	100K	2^{-1}
$\mathrm{BF}(1,h,1)$	$1.5h+1$	$0.5h$	1	100K	2^{-h}
$\mathrm{BF}(1,h,p)$	$1.5h+1$	$0.5hp$	1	100K$\times p$	2^{-h}
$\mathrm{BF}(w,h,1)$	$1.5wh+w$	$0.5wh$	w	100K	2^{-h}
$\mathrm{BF}(1,40,p)$	61	$20p$	1	100K$\times p$	10^{-12}
$\mathrm{BF}(w,40,1)$	$61w$	$20w$	w	100K	10^{-12}

in the table. Since $\mathrm{BF}(1,h,1)$ accepts one byte stream, only one Block RAM is used for buffering latest l bytes. Clearly, the numbers of Block RAMs and Ultra RAMs are proportional to h. If we want to receive multiple byte streams, we can simply use multiple $\mathrm{BF}(1,h,1)$s, which corresponds to $\mathrm{BF}(w,h,1)$. It accepts w byte streams, and performs the byte stream pattern test for one set of 100K patterns. To perform byte stream pattern test for more patterns, we can use $\mathrm{BF}(1,h,p)$, which accept p sets of 100K patterns. This configuration has h BF engines, each of which is used to read p bit arrays for byte stream pattern test. In general, we can use configuration $\mathrm{BF}(w,h,p)$, which consists of w $\mathrm{BF}(1,h,p)$s. Thus, it performs byte stream pattern test for w byte streams and p sets of 100K patterns with false positive probability 2^{-h}. It uses $1.5wh+w$ Block RAMs and $0.5whp$ Ultra RAMs. When we use a Xilinx Ultrascale+ VU9P FPGA for implementation. $1.5wh+w \leq 4320$ and $0.5whp \leq 960$ must be satisfied not to exceed available numbers of Block RAMs and Ultra RAMs. The table also shows the used resources and the performance for $\mathrm{BF}(1,40,p)$ and $\mathrm{BF}(w,40,1)$, which fully utilizes 960 Ultra RAMs in a VU9P FPGA when $p = w = 48$. The performance of these two configurations are actually evaluated in Section IV.

C. Related work

There are several previously published works for implementing the Bloom filter in the FPGA [21]–[24]. Manoharan *et al.* [21] presented an FPGA implementation of the Bloom filter for string matching. Their implementation simply uses XOR-based hash functions [25] $f : \{0,1\}^L \rightarrow \{0,1\}^S$ such that

$$\begin{aligned} & f(x_0x_1\cdots x_{L-1}) \\ &= (d_0 \cdot x_0) \oplus (d_1 \cdot x_1) \oplus \cdots \oplus (d_{L-1} \cdot x_{L-1}), \end{aligned}$$

where $x_0x_1\cdots x_{L-1}$ are L-bit input and $d_0, d_1, \ldots, d_{L-1}$ are predetermined constant numbers with S bits. Hence, for l-byte patterns, L must be $8l$ and the circuit size for f is proportional to LS, which is quite large. Harwayne-Gidansky *et al.* [22] also uses XOR-based hash functions. Hence, their experimental results show only for $l \leq 8$. Cho *et al.* [23] presented an FPGA implementation of the Bloom filter. However, they did not describe the details of hash functions. The implementation of [24] also uses XOR-based hash functions. As far as we know, no previously published work uses rolling hash function, by which the combinational circuit size for it is fixed and independent of pattern length l.

This paper is organized as follows. In Section II, we explain the Bloom filter and rolling hash functions. Section III presents our Bloom filter circuits and FPGA implementations. In Section IV shows experimental and implementation results. Section V concludes our work.

II. BYTE STREAM PATTERN TEST USING BLOOM FILTER

The main purpose of this section is to show how byte stream pattern test is performed using Bloom filter [1] with rolling hash function.

A. Bloom filter

We first review basic ideas of the Bloom filter. Let U be a universe and $P = \{p_0, p_1, \ldots, p_{m-1}\}$ be an m-element subset of U. Let $f : U \rightarrow [0, s-1]$ be a hash function. Usually, s is much smaller than the number of elements in U and so f is a many-to-one function. We call the value of $f(x)$ *the signature* of x. We use a zero-initialized bit array B with s bits to store the signatures of all $f(p_i)$ ($0 \leq i \leq m-1$). We write 1 in $B[f(p_i)]$ for all i ($0 \leq i \leq m-1$). Clearly, $B[k] = 0$ if and only if $k \neq f(p_i)$ for all i ($0 \leq i \leq m-1$). Let x be any element in U. We can use the bit array B thus obtained to test if $x \in P$. Clearly, if $B[f(x)] = 0$ then it is guaranteed that $x \notin P$ holds. However, even if $B[f(x)] = 1$, $x \in P$ may not hold. Thus, membership $x \in P$ can be tested by the value of $B[f(x)]$, but the result is *false positive*.

Let us evaluate the false positive probability. We assume that all $f(p_i)$ ($0 \leq i \leq m-1$) take values in $[0, s-1]$ uniformly and independently at random. For any fixed value r in $[0, s-1]$ and p_i in P, we have,

$$\Pr(r \neq f(p_i)) \quad = \quad 1 - \frac{1}{s}.$$

Since $B[r] = 0$ if and only if $r \neq f(p_i)$ for all i ($0 \leq i \leq m-1$), we have

$$\Pr(B[r] = 1) \quad = \quad 1 - (1 - \frac{1}{s})^m \approx 1 - e^{-\frac{m}{s}},$$

which is equal to the false positive probability $\Pr(B[f(x)] = 1 \mid x \notin P)$.

Lemma 1: Let P be a set of m elements in universe U and x be an element in U. A Bloom filter with one hash function with a bit array of s bits can test if $x \in P$ with false positive probability $1 - e^{-\frac{m}{s}}$.

In the Bloom filter, multiple hash functions can be used to decrease the false positive probability. Let $f_0, f_1, \ldots, f_{h-1}$ be h distinct hash functions. Again, we use a zero-initialized s-bit bit array B and we write 1 in $B[f_k(p_i)]$ for all i and k ($0 \leq i \leq m-1$ and $0 \leq k \leq h-1$). Similarly, we can use the bit array B to test if $x \in P$. If $B[f_k(x)] = 0$ for some k ($0 \leq k \leq h-1$), then it is guaranteed that $x \notin P$ holds. Thus, $x \in P$ can be tested by the values of $B[f_0(x)]$, $B[f_1(x)]$, ..., $B[f_{h-1}(x)]$. The false positive probability can be computed as follows. Let r be any fixed value in $[0, s-1]$. Since $B[r] = 0$ if and only if $r \neq f_k(p_i)$ for all i and k ($0 \leq i \leq m-1$ and $0 \leq k \leq h-1$), we have

$$\Pr(B[r] = 1) \quad = \quad 1 - (1 - \frac{1}{s})^{mh} \approx 1 - e^{-\frac{mh}{s}},$$

which is equal to the false positive probability $\Pr(B[f_k(x)] = 1$ for all $k \mid x \notin P)$. Thus, we have,

Lemma 2: Let P be a set of m elements in universe U and x be an element in U. A Bloom filter with h hash functions with a bit array of s bits can test if $x \in P$ with false positive probability $(1 - e^{-\frac{mh}{s}})^h$.

If the Bloom filter with h hash functions is implemented as it is, h bits in a bit array B must be read at the same time or in turn. For efficient implementation of the Bloom filter, we should separate B into multiple bit arrays as illustrated in Figure 3, so that each array is accessed once for each test. Let $B_0, B_1, \ldots, B_{h-1}$ be h bit arrays of size $s' = \frac{s}{h}$ each and f_0, f_1, ..., f_{h-1} be hash functions that return integers in $[0, s'-1]$. For zero-initialized bit arrays $B_0, B_1, \ldots, B_{h-1}$, we write 1 in $B_k[f_k(p_i)]$ for all i and k ($0 \leq i \leq m-1$ and $0 \leq k \leq h-1$). Similarly, if $B_k[f_k(x)] = 0$ for some k ($0 \leq k \leq h-1$), then it is guaranteed that $x \notin P$ holds. Thus, $x \in P$ can be tested by the values of $B_0[f_0(x)]$, $B_1[f_1(x)]$, ..., $B_{h-1}[f_{h-1}(x)]$. The false positive probability can be computed as follows. For any fixed k and r ($0 \leq k \leq h-1$ and $0 \leq r \leq s'-1$).

$$\begin{aligned} \Pr(B_k[r] = 1) \quad &= \quad 1 - (1 - \frac{1}{s'})^m = 1 - (1 - \frac{h}{s})^m \\ &\approx \quad 1 - e^{-\frac{mh}{s}}. \end{aligned}$$

Hence, the false positive probability is

$$\Pr(B_k[f_k(x)] = 1 \text{ for all } k \mid x \notin P) \quad \approx \quad (1 - e^{-\frac{mh}{s}})^h$$

Thus, we have,

Theorem 3: Let P be a set of m elements in universe U and x be an element in U. A Bloom filter with h hash functions using h bit arrays of $\frac{s}{h}$ bits each can test if $x \in P$ with false positive probability $(1 - e^{-\frac{mh}{s}})^h$.

From Lemma 2 and Theorem 3, we can see that the false positive probability is the same. However, from $1 - (1 - \frac{1}{s})^{mh} < 1 - (1 - \frac{h}{s})^m$, we can say that the false positive probability of the single bit array Bloom filter is a little smaller (or better) than that of the multiple bit arrays. The difference of the probability is quite small for large s. Since only one bit is read in each of multiple bit arrays, the implementation of the multiple bit arrays is easier. So, we use multiple bit arrays to implement the Bloom filter in the FPGA.

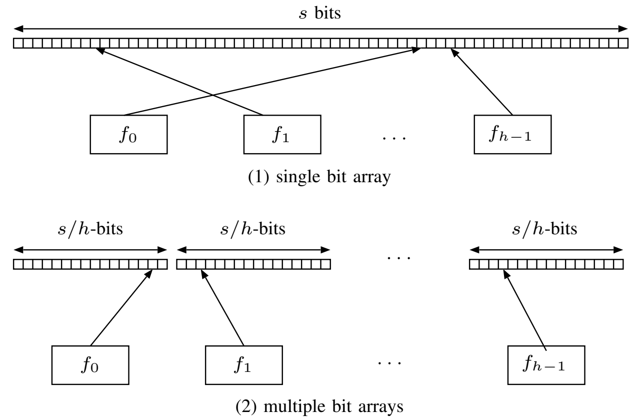

Fig. 3. Single bit array and multiple bit arrays for the Bloom filter

Suppose that the total number of bits s and the number of patterns m are fixed. We can choose the number h of hash functions (or bit arrays) to minimize the false positive probability $(1 - e^{-\frac{mh}{s}})^h$. It takes the minimum value when

$$mh = s \ln 2.$$

If this is the case, the false positive probability is

$$(1 - e^{-\ln 2})^h = 2^{-h}.$$

When $h = 1$, the false positive probability 2^{-1}. If this is the case, approximately $\frac{s}{2}$ bits in the s-bit bit array is 1 and so the information entropy of the bit array is maximized. Hence, we should design the Bloom filter so that 1 is written to a s-bit bit array $s \ln 2 = mh$ times. In the resulting s-bit array, approximately a half of the bits are 1.

Our FPGA implementation that we will show later uses a half of 288K-bit Ultra RAM to implement a bit array. Thus, we set $s' = 144$K. From $mh = s \ln 2$ and $s' = \frac{s}{h}$, we have

$$m = \frac{s}{h} \ln 2 = s' \ln 2 = 102,208. \tag{2}$$

Thus, we can perform byte stream pattern test for 100K ($= 102,400$) patterns with false positive probability approximately 2^{-h} using $\frac{h}{2}$ 288K-bit Ultra RAMs.

B. Rolling hash function

In this subsection, we assume that a universe U is a set of all sequences with l bytes and define a hash function for U.

Recall that, for efficient computation of hash functions, we employ a rolling hash function defined in formula (1) in Subsection I-B. Two parameters q and d should be selected so that $d^i \bmod q$ is non-zero for all i ($i \geq 0$). In other words, d should have prime factors that are not those of q. Clearly, f defined in formula (1) returns an integer in $[0, q-1]$. We will show that, for input byte stream $t = t_0 t_1 \cdots$, the signatures $f(t_j t_{j+1} \cdots t_{j+l-1})$ for all j (≥ 0) can be computed very efficiently. To simplify the treatment of boundary case, we assume that $t_i = 0$ for all i (< 0). These values can be computed by the following algorithm:

[Rolling Hash Function Algorithm]
$v \leftarrow 0$;
for $j \leftarrow 0$ to $n-1$
  $v \leftarrow (v \cdot d - t_{j-l} \cdot d^l + t_j) \bmod q$;

We can confirm that v stores $f(t_{j-l+1}t_{j-l+2}\cdots t_j)$ for all j (≥ 0) in turn, by induction on j. Initially, we can think that v is storing $f(t_{-l}t_{-l+1}\cdots t_{-1}) = 0$. Suppose that v is storing $f(t_{j-l}t_{j-l+1}\cdots t_{j-1}) = (t_{j-l} \cdot d^{l-1} + t_{j-l+1} \cdot d^{l-2} + \cdots + t_{j-2} \cdot d^1 + t_{j-1}) \bmod q$. By executing $v \leftarrow (v \cdot d - t_{j-l} \cdot d^l + t_j) \bmod q$, v stores $f(t_{j-l+1}t_{j-l+2}\cdots t_j) = (t_{j-l+1} \cdot d^{l-1} + t_{j-l+2} \cdot d^{l-2} + \cdots + t_{j-1} \cdot d^1 + t_j) \bmod q$. Thus, all values of $f(t_{j-l+1}t_{j-l+2}\cdots t_j)$ are computed one by one correctly.

Let us see how we select appropriate values of q and d. Let $\gamma(q, d)$ denote the minimum value of i (> 0) such that $d^i \bmod q = 1$. Note that, no such i exists if q and d are not coprime. We should not select such pair of q and d, and so we assume that $\gamma(q, d) = 0$ for such pair. Clearly, $d^0 = d^{\gamma(q,d)} = d^{2\gamma(q,d)} = d^{3\gamma(q,d)} = \cdots = 1 \pmod q$ holds for any pair of q and d. More generally, by multiplying d^j, we have $d^j = d^{\gamma(q,d)+j} = d^{2\gamma(q,d)+j} = d^{3\gamma(q,d)+j} = \cdots \pmod q$. Thus, swapping two bytes in distance of a multiple of $\gamma(q, d)$ does not change the value of hash function f. Hence, the value of $\gamma(q, d)$ should be as large as possible, because the values of $d^0, d^1, \ldots, d^{\gamma(q,d)-1} \pmod q$ are distinct and swapping two bytes in distance less than $\gamma(q, d)$ does not change the signature with high probability. Since $d^i \bmod q$ takes value in $[1, q-1]$, the maximum possible value of $\gamma(q, d)$ is $q-1$. If $\gamma(q, d) = q-1$, then we can guarantee that all values $d^0, d^1, \ldots d^{q-2} \pmod q$ are distinct.

Let $\max(\gamma(q))$ be a function such that

$$\max(\gamma(q)) \quad = \quad \max\{\gamma(q, d) \mid d > 0\}$$

Also, let $\mathrm{num}(\gamma(q))$ denote the number of ds in $[1, q-1]$ satisfying $\gamma(q, d) = \max(\gamma(q))$. Table II shows the values of $\max(\gamma(q))$, $\mathrm{num}(\gamma(q))$, and the first 8 ds satisfying $\max(\gamma(q)) = \gamma(q, d)$ for each q in $[1008, 1024]$. In the table, prime ds are boldfaced. For example, when $q = 1021$, $\gamma(q, d) = 1020$ holds for 256 numbers $d = 10, 22, 30, 31, 34, 35, 37, 40, \ldots, 1011$. From the table, we can see that, we should choose prime numbers for q, because $\max(\gamma(q)) = q-1$. In our FPGA implementation, we use four prime numbers 1009, 1013, 1019, and 1021 for q.

TABLE II
THE VALUES OF $\max(\gamma(q))$, $\mathrm{num}(\gamma(q))$, AND THE FIRST 8 qS SATISFYING $\max(\gamma(d)) = \gamma(q, d)$

q	$\max(\gamma(q))$	$\mathrm{num}(\gamma(q))$	examples of d
1008	12	128	5, 11, 13, 19, 29, 37, 43, 53
1009	1008	288	11, 17, 22, 26, 31, 33, 34, 38
1010	100	240	3, 7, 11, 13, 23, 27, 29, 33
1011	336	192	10, 19, 20, 22, 23, 29, 31, 34
1012	110	280	3, 5, 7, 13, 15, 17, 19, 27
1013	1012	440	3, 5, 7, 12, 17, 18, 20, 26
1014	156	96	7, 11, 37, 41, 59, 67, 71, 85
1015	84	288	2, 3, 11, 18, 19, 23, 26, 31
1016	126	252	3, 7, 11, 13, 15, 21, 23, 29
1017	336	192	5, 20, 23, 29, 34, 38, 43, 47
1018	508	252	3, 7, 13, 15, 19, 27, 31, 33
1019	1018	508	2, 6, 7, 8, 10, 13, 18, 21
1020	16	128	7, 11, 23, 29, 31, 37, 41, 61
1021	1020	256	10, 22, 30, 31, 34, 35, 37, 40
1022	72	144	5, 11, 13, 15, 29, 31, 33, 39
1023	30	336	5, 7, 10, 13, 14, 17, 19, 20
1024	256	256	3, 5, 11, 13, 19, 21, 27, 29

III. FPGA IMPLEMENTATIONS OF THE BLOOM FILTER

This section presents our Bloom filter circuits to be implemented in the FPGA. We first explain the details of Block RAMs and Ultra RAMs of the FPGA. After that, we show a basic circuit to compute rolling hash functions on the FPGA. We then go on to present *BF engine*, which computes bit position of the Ultra RAM to be read. Finally, we show our Bloom filter circuits using multiple BF engines.

A. Block RAM and Ultra RAM

This subsection explain a Block RAM and Ultra RAM necessary to understand our Bloom filter circuits. Xilinx Ultrascale+ family FPGA has two types of memory resources: *Block RAM* and *Ultra RAM* [14] as illustrated in Figure 2. We use Block RAMs to compute arithmetic modulo a prime number, which needs a large circuit with long delay if we use a combinational circuit. Using a Block RAM, modulo can be computed in one clock cycle. Also, Ultra RAMs are used to implement bit arrays.

A Block RAM is a 18K-bit dual port memory, which can be configured as 16K×1, 8K×2, 4K×4, 2K×9, or 1K×18. Figure 2 illustrates a 1K×18 Block RAM with 10-bit address with 18-bit word. It has two pairs of address input port with 10 bits each and data output ports with 18 bits each. Using these ports, two 18-bit words stored in two addresses can be accessed at the same time. A $1K \times 18$ Block RAM supports synchronous read and a rising clock edge is necessary to read a 18-bit word specified by the address port. More specifically, it has two 18-bit output register, a word specified by a 10-bit address input port is read and stored in a 18-bit output register, from which a stored word is continuously output to the 18-bit data output port. Note that, it is not possible to bypass the output register.

An Ultra RAM is a $4\mathrm{K} \times 72$ dual-port memory with 288K-bit capacity as illustrated in Figure 2. Unlike the Block RAM, a word size is fixed to 72. It has two pairs of address input port with 12 bits each and data output ports with 72 bits each. Using these ports, words stored in two addresses can be accessed at the same time. Thus, we can use an Ultra RAM as two 2K×72 single-port memory with 144K-bit capacity. Similarly, an Ultra RAM supports synchronous read and a rising clock edge is necessary to read a 72-bit word.

B. A basic circuit to compute rolling hash functions

We show how $(v \cdot d - t_{j-l} \cdot d^l + t_j) \bmod q$ is computed to evaluate $f(t_{j-l+1}t_{j-l+2}\cdots t_j)$ for every j one by one. Recall that, we use $q = 1009, 1019, 1013$, and 1021. Here, we use $q = 1021$ as an example for the detailed explanation. Let $\alpha : [0, 1020] \rightarrow [0, 1020]$ and $\beta : [0, 255] \rightarrow [0, 1020]$ be functions such that

- $\alpha(x) = (x \cdot d) \bmod q$, and
- $\beta(y) = (q - (y \cdot d^l \bmod q)) \bmod q$.

From $(v \cdot d - t_{j-l} \cdot d^l + t_j) \bmod q = (\alpha(v) + \beta(t_{j-l}) + t_j) \bmod q$, it is sufficient to show the computation of $(\alpha(v) + \beta(t_{j-l}) + t_j) \bmod q$. For the computation of α and β, we use one block RAM each. More specifically, each address x ($0 \le i \le 1020$) of a block RAM for α stores the value of $\alpha(x)$. By reading address x, the value of $\alpha(x)$ can be computed in one clock cycle. Similarly, each address y ($0 \le i \le 255$) of a block RAM for β stores the value of $\beta(y)$ to compute it in one clock cycle.

Let $z = \alpha(v) + \beta(t_{j-l}) + t_j$. We simply use an adder to compute the value of z. Since $z \le (q-1) + (q-1) + 255 = 2q + 253 < 3q$, exactly one of z, $z - q$, and $z - 2q$ is in $[0, q-1]$. By selecting one of them appropriately, we can obtain the value of $(\alpha(v) + \beta(t_{j-l}) + t_j) \bmod q$.

Figure 4 illustrates a circuit to compute $f(t_{j-l+1} t_{j-l+2} \cdots t_j)$. It has a FIFO of size l, which stores $t_{j-l}, t_{j-l-1}, \ldots, t_{j+1}$. Thus, the total capacity of FIFO is $8l$ bits. A 10-bit register is used to store the value of v. Two block RAMs compute $\alpha(v)$ and $\beta(t_{j-l})$. An adder computes the sum z of $\alpha(v)$, $\beta(t_{j-l})$, and t_j. Two subtractors are used to compute $z - q$ and $z - 2q$. A 3-to-1 selector chooses one of z, $z - q$ and $z - 2q$. For this purpose, the sign bits of $z - q$ and $z - 2q$ are used and the output of the selector can be determined by the following logic:

if($z - q < 0$) output z;
else if($z - 2q < 0$) output $z - q$;
else output $z - 2q$;

Since the resulting value is in $[0, q-1]$, the selector outputs $z \bmod q$ correctly.

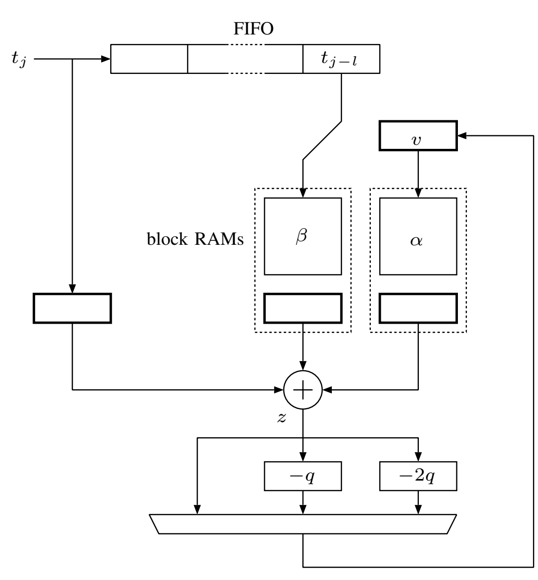

Fig. 4. A circuit to compute $v = (\beta(v) + \alpha(t_{j-l}) + t_j) \bmod q$

Let us analyze the timing of this circuit in Figure 4. We should focus on the path from the output of register v to the input of v to determine the correct timing. In this circuit, $\alpha(v)$ is computed using a block RAM. Since a block RAM works synchronous read mode, the value of $g(v)$ is read and stored in the output register in a block RAM at the rising edge of the clock input. Thus, one clock cycle is necessary to compute $\alpha(v)$. Therefore, two clock cycles are necessary to update the value of v. Also, the circuit has a long critical path. The path from the output of the block RAM to compute α to the input of v involves an adder, a subtractor, and a 3-to-1 selector. This long critical path degenerates the clock performance.

C. Hash function for efficient FPGA implementation

We will modify hash functions so that the signature $f(t_{j-l+1} t_{j-l+2} \cdots t_j)$ is computed in every clock cycle. We will show how $f(x)$ is computed, where $x = x_0 x_1 \cdots x_{l-1}$ is a l-byte sequence. We assume that l is divisible by four. We make four sequences of length $\frac{l}{4}$ by picking every four bytes in x as follows:

- $\mathcal{X}_0 = x_0 x_4 x_8 \cdots x_{l-4}$,
- $\mathcal{X}_1 = x_1 x_5 x_9 \cdots x_{l-3}$,
- $\mathcal{X}_2 = x_2 x_6 x_{10} \cdots x_{l-2}$, and
- $\mathcal{X}_3 = x_3 x_7 x_{10} \cdots x_{l-1}$.

We define two hash function a and b for x using a rolling hash function f for $\mathcal{X}_0$, $\mathcal{X}_1$, $\mathcal{X}_2$, and $\mathcal{X}_3$ as follows:

$$a(x) = a'(x) \bmod 16\text{K}$$
$$b(x) = \begin{cases} 8 & \text{if } b'(x) \bmod 16\text{K} \le 1820 \\ b'(x) \bmod 8 & \text{otherwise,} \end{cases}$$

where

$$a'(x) = f(\mathcal{X}_0) + 31 \cdot f(\mathcal{X}_1) + 127 \cdot f(\mathcal{X}_3)$$
$$b'(x) = f(\mathcal{X}_1) + 127 \cdot f(\mathcal{X}_2) + 31 \cdot f(\mathcal{X}_3).$$

Note that $\bmod$ 16K (i.e. $\bmod$ 16,384) can be computed by taking least significant 14 bits. If rolling hash function f for $q = 1021$ is used, then the return values of f are in $[0, 1020]$. The values of $a(x)$ and $b(x)$ are in $[0, 16K-1]$ and $[0, 8]$, respectively. Constant numbers 31, and 127 in the definition are selected from prime numbers, which are powers of two minus 1. Thus, multiplication is not necessary. For example, $31 \cdot f(\mathcal{X}_1)$ can be computed by evaluating $(f(\mathcal{X}_1) << 5) - f(\mathcal{X}_1)$. Also, a constant number 1820 is used in the definition of b, because $\frac{16\text{K}}{9} \approx 1820.4$. Hence, $b'(x) \bmod 16\text{K} \le 1820$ is satisfied with probability $\frac{1}{9}$ and thus the resulting value of $b(x)$ is 8 with probability $\frac{1}{9}$. If $b'(x) \bmod 16\text{K} > 1820$ then $b' \bmod 8$ takes 0, 1, …, 7 with equal probability $\frac{1}{8}$. Since $b'(x) \bmod 16\text{K} > 1820$ with probability $\frac{8}{9}$, the value of $b(x)$ takes 0, 1, …, 7 with equal probability $\frac{8}{9} \cdot \frac{1}{8} = \frac{1}{9}$. Let $a_{13} a_{12} \cdots a_0$ and $b_3 b_2 b_1 b_0$ denote the binary representations of the resulting values of $a(x)$ and $b(x)$. We use $a_{10} a_9 \cdots a_0$, which takes value in $[0, 2K-1]$, to specify an address of the 2K $\times$ 72 2-dimensional bit array. Also, $b_3 b_2 b_1 b_0 a_{13} a_{12} a_{11}$, which is in $[0, 71]$, is used to specify a bit of a 72-bit word.

To clarify that the resulting values of a and b are almost uniform, we have evaluated the number of occurrences of each resulting values of a and b. More specifically, the number of occurrences of 144K integers $b_3b_2b_1b_0a_{13}a_{12}\cdots a_0$ for all possible values of $f(\mathcal{X}_0), f(\mathcal{X}_1), f(\mathcal{X}_2)$ and $f(\mathcal{X}_3)$. Since each number takes $q = 1021$ integers, the total number of all possible combinations is $q^4 = 1021^4 \approx 10^{12}$, which is too large to evaluate them by a conventional CPU. Thus, we have used NVIDIA Tesla V100 GPU for this task. Table III shows the minimum/average/maximum numbers of occurrences of 144K integers for $q = 1009, 1013, 1019$, and 1021. For example, the average number of occurrences of each integer if $\frac{1021^4}{144\text{K}} \approx 7369542.4$, when $q = 1021$. Also, the numbers of occurrences of all integers are in the range $[7354614, 7400244]$. Thus, the bias ration is $\frac{7400244-7354614}{7369542.4} = 0.0062$. We can see that the bias ratios for all qs are less than 1%, so the resulting values of a and b are almost uniformly distributed.

TABLE III
THE RANGE OF THE OCCURRENCES OF 144K NUMBERS USING OUR HASH FUNCTIONS

q	minimum	average	maximum	bias ratio
1009	7004135	7029140.4	7069022	0.0092
1013	7119695	7141268.0	7179593	0.0084
1019	7295475	7311968.1	7345400	0.0068
1021	7354614	7369542.4	7400244	0.0062

D. BF engine to compute hash functions a and b

We will modify a circuit shown in Figure 4. Figure 5 illustrates a modified circuit to compute $a(t_{j-l+1}t_{j-l+2}\cdots t_j)$ and $b(t_{j-l+1}t_{j-l+2}\cdots t_j)$ from the values of t_j and t_{j-l}. The register v is used to store the previous signature f. Two pipeline stages with pipeline registers are inserted to decrease the critical path. Thus, the path from the output of v to the input of v, has 3 registers including the output register of the block RAM for g and registers in two pipeline stages.

Let $v_j = f(t_{j-l}t_{j-l+4}t_{j-l+8}, \cdots t_{j-4})$ and $u_j = g(v_j) + h(t_{j-l}) + t_j$. Figure 6 illustrates a timing chart of the circuit. We can see that, from the value of v_j stored in register v, the value of v_{j+4} is computed and stored in register v in 4 clock cycles. Three registers are used to store past three values of v. Using the four values of v, we can compute the resulting values of a and b by combinational circuits. We should insert pipeline stages to these combinational circuits to maximize the clock frequency. For later reference, *BF engine* $E(q, d)$ denote this circuit in Figure 5 with parameters q and d.

E. Bloom filter circuit using multiple BF Engines

We will design a circuit for the Bloom filter with multiple bit arrays using multiple BF engines $E(q, d)$s and multiple Ultra RAMs. Since a BF engine uses a 144K-bit bit array, two BF engines can share a 288K-bit Ultra RAM, which stores two 144K-bit bit arrays.

Figure 7 illustrate our Bloom filter with multiple bit arrays, which corresponds to $\mathrm{BF}(1, h, 1)$ in Table I. We use BF engines $E(q_0, d_0)$, $E(q_1, d_1)$, …, $E(q_{h-1}, d_{h-1})$, each of which computes the values of $a(t_{j-l+1}t_{j-l+2}\cdots t_j)$ and $b(t_{j-l+1}t_{j-l+2}\cdots t_j)$ with parameters q_i and d_i. This circuit can perform byte stream pattern test for 1 byte stream and

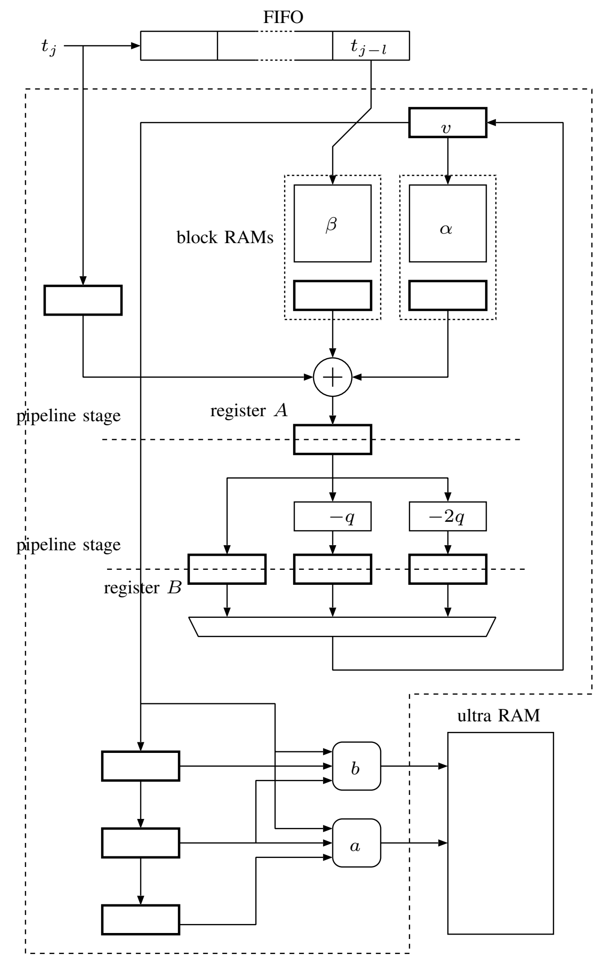

Fig. 5. A BF engine $E(q, d)$ to compute $a(t_{j-l+1}t_{j-l+2}\cdots t_j)$ and $b(t_{j-l+1}t_{j-l+2}\cdots t_j)$

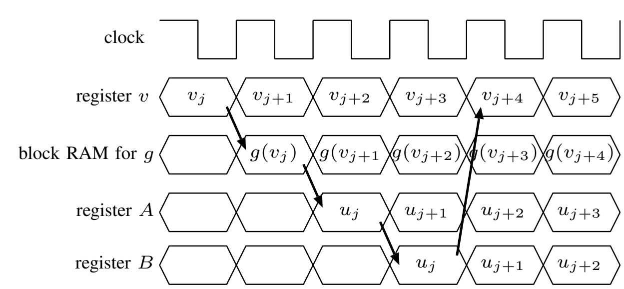

Fig. 6. A timing chart of the BF engine in Figure 5

100K patterns. We should select every pair q_i and d_i such that

- all pairs are distinct,
- q_i is 1009, 1013, 1019, or 1021, and these four values are used equally for h BF engines,
- $\gamma(q_i, d_i) = \max(\gamma(q_i)) = q_i - 1$.

Recall that $d_i^0, d_i^1, \cdots, d_i^{q_i-2} \pmod{q_i}$ take distinct integers in $[1, q_i - 1]$, and $d_i^0 \bmod q_i = d_i^{q_i-1} \bmod q_i = 1$. Thus, a sequence d_i^0, d_i^1, d_i^2, …, $\pmod{q_i}$ is a repeat of a sequence

$d_i^0, d_i^1, \cdots, d_i^{q_i-2} \pmod{q_i}$ of length $q_i - 1$. Since we use 4 prime numbers 1009, 1013, 1019, and 1021, we can think that 4 sequences using these 4 prime numbers, one for each, involve iterations of length $1009 \cdot 1013 \cdot 1019 \cdot 1021 \approx 10^{12}$, which is quite large.

We should use multiple BF circuits to reduce the false positive probability. Figure 7 illustrates $\mathrm{BF}(1, h, 1)$ with $h = 6$ in Table I. Note that a bit array is generated for each pair of parameters q_i and d_i and written in a half of the Ultra RAM before hand. Each BF engine $E(q_i, d_i)$ compute functions a and b for the input byte stream.

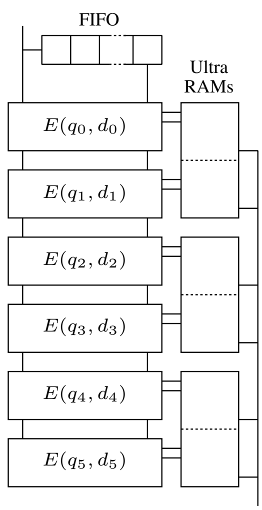

Fig. 7. Illustrating Bloom filter circuit $\mathrm{BF}(1, 6, 1)$.

Recall that an BF engine for $q = 1021$ computes two functions $\alpha : [0, 1020] \rightarrow [0, 1020]$ and $\beta : [0, 255] \rightarrow [0, 1020]$ using Block RAMs. Since uses 256 words in a dual-port Block RAMs, two different hs can be computed at the same time. Hence, two BF engines can share one Block RAM to compute h. Thus, two BF engines can be implemented using 3 Block RAMs, two for α and one for β.

F. Bloom filter circuits for multiple pattern sets

Figure 8 illustrate our Bloom filter circuit, which corresponds to $\mathrm{BF}(1, h, p)$ with $h = 6$ and $p = 5$ in Table I. Each of $p = 5$ rows of Ultra RAMs uses h bit arrays used for one of $p = 5$ pattern sets $P_1, P_2, \ldots, P_{p-1}$. Thus, the signatures of patterns in each P_i are written in Ultra RAMs in i-th row. Using $\frac{h}{2}$ Ultra RAMs in each i-th row, we can perform the byte stream pattern test for P_i. Hence, we have,

Lemma 4: $\mathrm{BF}(1, h, p)$ can perform the byte stream pattern test for $P_1, P_2, \ldots, P_{p-1}$ with 100K patterns each in parallel with false positive probability 2^{-h}.

We can simply arrange multiple $\mathrm{BF}(1, h, p)$s to perform the byte stream pattern test for multiple byte streams. Thus, we have,

Theorem 5: $\mathrm{BF}(w, h, p)$ can perform the byte stream pattern test for w byte streams and $P_1, P_2, \ldots, P_{p-1}$ with 100K patterns each in parallel with false positive probability 2^{-h}.

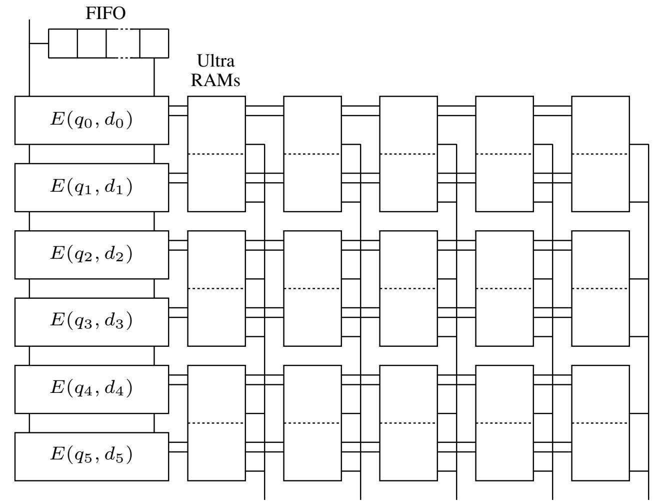

Fig. 8. Illustrating Bloom filter circuit $\mathrm{BF}(1, 6, 5)$.

IV. EXPERIMENTAL RESULTS

A. The false positive probability

We have evaluated the false positive probability using (1) randomly generated byte streams and (2) Wikipedia text. We have used Mersenne twister random number generator [26] to generate 1G-byte streams, in which 8-bit numbers are selected uniformly at random. Also, randomly selected 100K 1024-byte data in it to use them as patterns. Thus, each pattern matches at least one position in 1G-byte streams. Further, we merged Wikipedia text appropriately to have 1G-byte streams to see the false positive probability for biased byte streams. Similarly, we randomly selected 100K 1024-byte sequence in it to use them as patterns. Table IV shows the resulting the false positive probability for these cases. We can see that the false positive probability is close to $2^{-h} \approx 10^{3h/10}$, and so we can say that the theoretical analysis of the false positive probability is correct. The false positive probability is a little larger than 10^{-12}, because we use 102,400 patterns, which is a little larger than 102,208 in formula (2).

TABLE IV
THE FALSE POSITIVE PROBABILITY

h	random	Wikipedia
1	0.5007	0.5007
10	0.989×10^{-3}	0.990×10^{-3}
20	0.971×10^{-6}	0.959×10^{-6}
30	1.061×10^{-9}	1.024×10^{-9}
40	1.500×10^{-12}	1.250×10^{-12}

B. The performance of our Bloom filter circuits

We have implemented our Bloom filter circuits and evaluated the used hardware resources of the FPGA and the performance. The used hardware resources are the number of CLBs, LUTs, FFs, Block RAMs, and Ultra RAMs. CLBs (Configurable Logic Blocks) in the FPGA are used for implementing combinational and sequential logic [13]. One CLB contains 8 LUTs (Look-Up-Tables) and 16 FFs (Flip-Flops). An LUT is a 64-bit memory, which can be configured as a 6-to-1 LUT or a 5-to-2 LUT. A VU9P FPGA has 147,780 CLBs with 1,182,240 LUTs and 2,364,480 FFs totally. We also evaluated the number of 4,320 18K-bit Block RAMs and 960 288K-bit Ultra RAMs used in our Bloom filter circuits. Table V shows the performance of our Bloom filter circuits $\mathrm{BF}(1, 40, p)$ for $p = 1, 2, 4, 8, 16, 32, 48$. In these circuits, the false positive probability for testing one sequence is $2^{-40} \approx 10^{-12}$. Our Bloom filter circuit $\mathrm{BF}(1, 40, 1)$ runs in 456MHz and so the throughput of a byte stream is 456MHz$\times 8 = 3.65$Gbps. We can think that the circuit outputs one false positive result in every $\frac{1}{456 \cdot 10^6 \cdot 10^{-12}} \approx 2193$ seconds on average, which is quite large. Also, $\mathrm{BF}(1, 40, 48)$, which fully utilizes 960 Ultra RAMs, runs in 143MHz. Due to wire routing and resource mapping overhead, the clock frequency is lower than that of $\mathrm{BF}(1, 40, 1)$. In this implementation, only 33.5% CLBs and 1.4% Block RAMs are used. Also, the false positive frequency is one false positive result in $\frac{48}{143 \cdot 10^6 \cdot 10^{-12}} = 146$ seconds on average, which is still quite large.

Table VI shows the performance of our Bloom filter circuits $\mathrm{BF}(w, 40, 1)$ for $w = 1, 2, 4, 8, 16, 32, 48$. The false positive probability is also $2^{-40} \approx 10^{-12}$. Note that, $\mathrm{BF}(w, 40, 1)$ and $\mathrm{BF}(1, 40, p)$ are the same when $w = p = 1$. We can see that $\mathrm{BF}(48, 40, 1)$ fully utilized 960 Ultra RAMs, and runs in 129MHz. Since it uses $40 \times 48 = 1920$ BF engines, more CLBs and Block RAMs are used than $\mathrm{BF}(1, 40, 48)$. Further, since it reads $w = 48$ byte streams, the throughput is $129MHz \times 8 \times 48 = 49.5$Gpbs. The false positive frequency is one false positive result in $\frac{48}{129 \cdot 10^6 \cdot 10^{-12}} = 161$ seconds on average.

C. Comparison with a sequential algorithm

Although the performance of our Bloom filter circuit is quite high, it is not easy to see how high it is. So, we have implemented a sequential Bloom filter algorithm and evaluated the performance using a latest Intel CPU. Note that, this is just a reference to see the performance of our Bloom filter circuit relative to a sequential algorithm on an Intel CPU for the same byte stream pattern test. For fair comparison, we optimize a sequential byte stream pattern test to obtain almost the same result as $\mathrm{BF}(48, 40, 1)$. Recall that our BF engine computes four hash functions and combine them to obtain a signature due to the limitation of circuits. However, since modulo of a large prime can be computed by a CPU very easily, we select $q = 147,451$ which is the largest prime number less than 144K. Also, we select d's such that $\max(\gamma(q, d)) = q - 1$. For such q and d's, we simply perform $v \leftarrow (v \cdot d - t_{j-l} \cdot d^l + t_j) \bmod q$ in the Rolling Hash Function Algorithm. We can accelerate this sequential algorithm by omitting the computation of hash functions as follows. Recall that, in the Bloom filter, h signatures $f_0(x)$, $f_1(x)$, …, $f_{h-1}(x)$ are computed for all sequences x in the input byte stream, and it returns positive (that is, $x \in P$) if all of $B[f_0(x)]$, $B[f_1(x)]$, …, $B[f_{h-1}(x)]$ are 1. If one of them is 0, it returns negative. Thus, the following algorithm works correctly as a Bloom filter.

for $i \leftarrow 0$ to $h - 1$
    if($B[f_i(x)] = 0$) then return negative;
return positive;

Since each $B[f_i(x)] = 1$ with probability 2^{-1}, the value of $B[f_i(x)]$ is read only if $B[f_0(x)]$, $B[f_1(x)]$, …, $B[f_{i-1}(x)]$ are 1. Hence, the probability that $B[f_i(x)]$ ($0 \le i \le h-1$) is read is 2^{-i} So, it makes sense to evaluate the value of $f_i(x)$ from scratch using formula (1) for large i, because $B[f_i(x)]$ is not read with probability $1 - 2^{-i}$. Thus, to accelerate the sequential algorithm we modify it so that

- each $f_i(x)$ ($0 \le i \le T - 1$) is evaluated for every sequence $x = t_{j-l+1} t_{j-l+2} \cdots t_j$ by evaluating $v \leftarrow (v \cdot d - t_{j-l} \cdot d^l + t_j) \bmod q$, and
- each $f_i(x)$ ($T \le i \le h - 1$) is evaluated from the scratch using formula (1) only if the value of $B[f_i(x)]$ is necessary.

Roughly speaking, the expected running time of the modified sequential algorithm is

$$O(1) \times T + \sum_{i=T}^{h-1} (O(l) \times 2^{-i}) \quad = \quad O(T + l 2^{-T})$$

time per byte of the input byte stream, where l is the length of the patterns. We can select threshold value T so that the running time is minimized. From above theoretical analysis of the running time, the value of T that minimizes the running time satisfies $T = O(\log l - \log\log l)$. Table VII shows experimental results of this modified sequential algorithm for $l = 1024$ and $h = 40$ with each T in $[8, 14]$. From the table, we can see that the throughput is maximized when $T = 11$. In this case, the throughput is 0.218Gbps, and that of $\mathrm{BF}(48, 40, 1)$ is $\frac{49.5\mathrm{Gbps}}{0.218\mathrm{Gbps}} = 227$ times better.

Since Core i7-6700K has 4 cores with eight hyperthreads, it may be possible to accelerate the computation. It needs parallel computing techniques, and so it is out of scope of this paper. However, we can say that the throughput can not be improved more than 8 times using 8 hyperthreads. Thus, we can say that our Bloom filter circuit implemented in the FPGA is fast enough.

V. Conclusion

In this paper, we have presented Bloom filter circuits for byte stream input optimized for the Xilinx Ultrascale+ VU9P FPGA. It computes rolling hash functions using Block RAMs in the FPGA, and uses Ultra RAMs to store signatures of patterns. The experimental results show that, the throughput of our Bloom filter circuit for 48 byte streams and 100K patterns is 49.5Gbps. On the other hand, the throughput of the optimized sequential algorithm is 0.218Gbps on Intel Core i7-6700K. Thus, our Bloom filter circuit running on the FPGA is 227 times faster than a sequential algorithm on an Intel Core i7 CPU for the same task.

TABLE V
THE PERFORMANCE OF $\mathrm{BF}(1, 40, p)$

p	CLB	LUT	FF	Block RAM	Ultra RAM	Clock (MHz)	Throughput (Gbps)	Patterns (K)	false positive frequency (sec)
VU9P	147,780	1,182,240	2,364,480	4,320	960	-	-	-	-
1	1,512 (1.0%)	7,358 (0.6%)	8,516 (0.4%)	61 (1.4%)	20 (2.1%)	456	3.65	100	2193
2	2,477 (1.7%)	12,524 (1.1%)	15,618 (0.7%)	61 (1.4%)	40 (4.2%)	453	3.62	200	1104
4	4,394 (3.0%)	22,832 (1.9%)	29,823 (1.3%)	61 (1.4%)	80 (8.3%)	452	3.62	400	553
8	8,949 (6.1%)	41,917 (3.5%)	58,232 (2.5%)	61 (1.4%)	160 (16.7%)	389	3.11	800	321
16	17,633 (11.9%)	81,241 (6.9%)	115,049 (4.9%)	61 (1.4%)	320 (33.3%)	274	2.19	1,600	228
32	33,553 (22.7%)	160,699 (13.6%)	228,682 (9.7%)	61 (1.4%)	640 (66.7%)	167	1.34	3,200	187
48	49,541 (33.5%)	240,200 (20.3%)	342,315 (14.5%)	61 (1.4%)	960 (100%)	143	1.14	4,800	146

TABLE VI
THE PERFORMANCE OF $\mathrm{BF}(w, 40, 1)$

w	CLB	LUT	FF	Block RAM	Ultra RAM	Clock (MHz)	Throughput (Gbps)	Patterns (K)	positive false frequency (sec)
VU9P	147,780	1,182,240	2,364,480	4,320	960	-	-	-	-
1	1,512 (1.0%)	7,358 (0.6%)	8,516 (0.4%)	61 (1.4%)	20 (2.1%)	456	3.65	100	2193
2	2,851 (1.9%)	14,286 (1.2%)	16,919 (0.7%)	122 (2.8%)	40 (4.2%)	425	6.80	100	1176
4	5,585 (3.8%)	28,984 (2.5%)	33,734 (1.4%)	244 (5.6%)	80 (8.3%)	409	13.1	100	611
8	11,100 (7.5%)	57,983 (4.9%)	67,363 (2.8%)	488 (11.3%)	160 (16.7%)	388	24.9	100	322
16	22,532 (15.2%)	108,746 (9.2%)	134,624 (5.7%)	976 (22.6%)	320 (33.3%)	298	38.1	100	210
32	46,476 (31.4%)	211,047 (17.9%)	269,133 (11.4%)	1952 (45.2%)	640 (66.7%)	175	44.8	100	179
48	68,783 (46.5%)	316,558 (26.8%)	403,646 (17.1%)	2928 (67.8%)	960 (100%)	129	49.5	100	161

TABLE VII
THE THROUGHPUT OF THE MODIFIED SEQUENTIAL ALGORITHM FOR EACH THRESHOLD T

T	8	9	10	11	12	13	14
Gps	0.148	0.190	0.213	0.218	0.216	0.204	0.193

REFERENCES

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communication of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Mathematics*, vol. 1, pp. 485–509, Jan. 2004.

[3] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Computer Networks*, vol. 57, no. 18, pp. 4047–4064, Dec 2013.

[4] E. H. Spafford, "OPUS: Preventing weak password choices," *Computers & Security*, vol. 11, no. 3, pp. 273–278, May 1992.

[5] N. S. Artan, K. Sinkar, and J. Patel, "Aggregated Bloom filters for intrusion detection and prevention hardware," in *Proc. of IEEE Global Telecommunications Conference*, Nov. 2007.

[6] S. Parthasarathy and D. Kundur, "Bloom filter based intrusion detection for smart grid SCADA," in *Proc. IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, Oct. 2012.

[7] M. Aldwairi, K. Al-Khamaiseh, F. Alharbi, and B. Shah, "Bloom filters optimized wu-manber for intrusion detection," *The Journal of Digital Forensics, Security and Law (JDFSL)*, vol. 11, no. 4, 2016.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[9] M. Ahmadi and S. Wong, "A cache architecture for counting Bloom filters," in *Proc. of IEEE International Conference on Networks*, Nov. 2007, pp. 218–223.

[10] K. Nakano and E. Takamichi, "An image retrieval system using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, May 2003.

[11] K. Nakano and Y. Yamagishi, "Hardware n choose k counters with applications to the partial exhaustive search," *IEICE Trans. on Information & Systems*, vol. E88-D, no. 7, pp. 1350–1359, 2005.

[12] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an FPGA," in *Proc. of International Conference on Parallel and Distributed Systems*, Dec. 2008, pp. 723–728.

[13] Xilinx Inc., "Ultrascale architecture configurable logic block: User guide," Feb 2017.

[14] ——, "Ultrascale architecture memory resources," May 2017.

[15] ——, "Ultrascale architecture DSP slice," June 2017.

[16] ——, "Ultrascale architecture and product data sheet: Overview," Feb. 2017.

[17] R. Woods, J. McAllister, and G. L. amnd Ying Yi, *FPGA-based Implementation of Signal Processing Systems:2nd Edition*. Wiley, May 2017.

[18] J. L. Bordim, Y. Ito, and K. Nakano, "Instance-specific solutions to accelerate the CKY parsing for large context-free grammars," *International Journal on Foundations of Computer Science*, vol. 15, no. 2, pp. 403–416, 2004.

[19] Y. Ito, K. Nakano, and S. Bo, "The parallel FDFM processor core approach for CRT-based RSA decryption," *International Journal of Networking and Computing*, vol. 2, no. 1, pp. 79–96, Jan. 2012.

[20] T. Kawamoto, X. Zhou, J. L. Bordim, Y. Ito, and K. Nakano, "An FPGA implementation for a flexible-length-arithmetic processor employing the FDFM processor core approach," *IEICE Transactions on Information and Systems*, vol. E99-D, no. 12, pp. 2901–2910, Dec. 2016.

[21] A. Manoharan, A. Krishnan, and P. Periasamy, "Design and implementation of a string matching system for network intrusion detection using FPGA-based low power multiple-hashing bloom filters," *International Journal of Computer Science and Applications*, vol. 1, no. 3, pp. 186–189, Jan. 2008.

[22] J. Harwayne-Gidansky, D. Stefan, and I. Dalal, "FPGA-based SoC for real-time network intrusion detection using counting Bloom filters," in *Proc. of IEEE SOUTHEASTCON*, 2009.

[23] J. M. Cho and K. Choi, "An FPGA implementation of high-throughput key-value store using Bloom filter," in *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, Jun. 2014.

[24] Sireesha and M. Roopa, "An FPGA implementation of hashed key-value store using bloom filter," *International Journal of Computer Science and Mobile Computing*, vol. 4, no. 5, pp. 1094–1100, May 2014.

[25] H. Vandierendonck and K. D. Bosschere, "XOR-based hash functions," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 800–812, 2005.

[26] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998.