Microprocessors and Microsystems 93 (2022) 104619

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications

Arish Sateesan [a,*], Jo Vliegen [a], Joan Daemen [b], Nele Mentens [a,c]

[a] imec-COSIC/ES&S, ESAT, KU Leuven, Belgium
[b] Digital Security Group, Radboud University, The Netherlands
[c] LIACS, Leiden University, The Netherlands

A R T I C L E I N F O

Keywords:
Bloom filter
FPGA
Network security
Approximate membership query
Non-cryptographic hash

A B S T R A C T

This paper optimizes Bloom filter algorithms and hardware architectures for high-speed implementation on FPGAs. A Bloom filter is a data structure that is used to check whether input search data are present in a table of stored data. Bloom filters are extensively used for monitoring network traffic. Improving the speed of Bloom filters can therefore have a significant impact on the speed of many network applications. The most important components determining the speed of Bloom filters are hash functions. While hash functions in Bloom filters do not require strong cryptographic properties, they do need a minimized computational delay. In this work, we evaluate and compare the performance and resource utilization of different Bloom filter algorithms and architectures as well as non-cryptographic hash functions on an FPGA. We also propose a new non-cryptographic hash function, called Xoodoo-NC, derived from the cryptographic permutation Xoodoo. Xoodoo-NC is a reduced-round, reduced-state version of Xoodoo, inheriting Xoodoo's desired avalanche properties and low logical depth, resulting in an ultra-low-latency non-cryptographic hash function. We evaluate the performance of Bloom filter architectures based on Xoodoo-NC on a Xilinx UltraScale+ FPGA and we compare the performance and resource occupation to existing Bloom filter implementations. We additionally compare our results to memories that use the built-in CAM cores in Xilinx UltraScale+ FPGAs. Our proposed algorithmic and architectural advances lead to Bloom filters that, to the best of our knowledge, outperform all other FPGA-based solutions.

1. Introduction

Network applications require efficient lookup algorithms for, e.g., packet forwarding, traffic flow monitoring and security, and deep packet inspection. The most popular data structures for high-speed lookups are Content-addressable Memories (CAMs) and Bloom filters. Whereas a CAM searches for a match between input data and stored data, and returns the address of the matching data, a Bloom filter can only detect a match without returning the address. Another difference is that false positives are possible in a Bloom filter, while a CAM has no false positives nor false negatives. Nevertheless, CAMs perform worse than Bloom filters when it comes to chip area, energy consumption and operating speed, as presented by George Varghese in [1]. Since these are critical implementation properties for network routers [2], Bloom filters are the preferred lookup mechanism when a match address is not needed, and when (a limited number of) false positives are acceptable. This is the case in various network (security) applications, such as deep packet inspection [3], network intrusion detection [4], distributed caching, resource routing, and network measurement infrastructures [5,6]. Additionally, Bloom filters also provide the benefit of constant-time queries, access refinement, and content anonymization [6]. Applications like flow measurement or traffic mapping [7–9] check whether or not the identifier (ID) of a network flow has been recorded earlier. The ID could, for example, be a combination of the host and destination addresses and ports.

In order to adhere to the strong bandwidth and energy requirements in Terabit networks (which are defined as networks with a bandwidth higher than 100 Gbps), hardware implementation platforms, such as FPGAs, are increasingly used in network applications [10]. This motivates our work which studies the design and implementation of high-speed Bloom filters on FPGA. Since the speed of a Bloom filter heavily depends on the speed of the hash functions that process the incoming data, we first concentrate on the design and implementation

* Corresponding author.
E-mail addresses: arish.sateesan@kuleuven.be (A. Sateesan), jo.vliegen@esat.kuleuven.be (J. Vliegen), joan@cs.ru.nl (J. Daemen), nele.mentens@kuleuven.be (N. Mentens).

https://doi.org/10.1016/j.micpro.2022.104619
Received 14 January 2021; Received in revised form 21 December 2021; Accepted 15 April 2022
Available online 28 July 2022

of a high-speed, hardware-friendly hash function. The desired properties of the non-cryptographic hash functions in Bloom filters are more relaxed than the properties of cryptographic hash functions. Both cryptographic and non-cryptographic hash functions map an input of arbitrary length to an output of fixed length. Good hash functions, cryptographic as well as non-cryptographic: (1) map the inputs as uniformly as possible over the entire set of outputs, (2) are easy to compute, and (3) exhibit the avalanche effect, which says that many output bits change, even when there is only a small change in the input. Cryptographic hash functions additionally require preimage resistance, second-preimage resistance and collision resistance [11]. We propose to start from a cryptographic hash function that possesses all the aforementioned properties, and to simplify it such that the computational delay is minimized, while the non-cryptographic properties are maintained. This leads to the Xoodoo-NC (Xoodoo-non-cryptographic) algorithm, derived from the cryptographic permutation Xoodoo [12]. We use Xoodoo-NC in the Bloom-1 architecture, proposed by Qiao et al. in [13], and the Parallel Bloom Filter architecture, proposed by Dharmapurikar et al. in [4]. We show that our Bloom filter implementations based on Xoodoo-NC are more efficient with respect to occupied FPGA resources and computational delay in comparison to Bloom filters based on the previously proposed hash function FNV-1a [14], and CAM-based lookup architectures. This paper extends our previous work presented in [15] by giving more details on the hardware architectures of the implemented hash functions and by adding Parallel Bloom Filters to the comparison.

This paper is organized as follows: Section 2 consists of background information on lookup algorithms and architectures in network applications and explains the Bloom filter architectures that we use. Section 3 gives an overview of related work. Section 4 presents our novel algorithms and architectures as well as the CAM architectures that we compare with. The hardware architectures of the implemented Bloom filters and hash functions are presented in Section 5. The results are discussed in Section 6. Finally, conclusions are drawn in Section 7.

2. Lookup algorithms and architectures

2.1. Content-addressable memory

Content-addressable memory (CAM) compares input data with data that are stored in the memory. If the same data are present in the memory, the CAM returns the address of the matching data. In a conventional memory, such as a Random-Access Memory (RAM), the address is used as an input and the data that are stored at that address are returned by the RAM. CAMs are, for example, used in network switches, which do a lookup of the destination MAC address of incoming network traffic to determine the port on which the traffic needs to be sent out. A CAM stores each bit explicitly in a memory cell, just like a RAM does. CAMs contain additional hardware circuits that facilitate the look-up. In recent FPGAs, IP cores are available to build fast and resource-efficient CAM structures based on the embedded RAM in the FPGA [16].

2.2. Bloom filters

In 1970, Howard Bloom presented space/time trade-offs for lookup algorithms in data structures based on hash coding with allowable errors [2]. The author's name has been linked to the presented algorithms ever since. The Bloom filter is an approximate membership query data structure that is used to assist in determining whether an element is in a set of elements. The filter has two possible outcomes: (1) the element might be in the set, or (2) the element is not in the set. Although the first option can trigger false positives, the second option rules out false negatives.

A standard Bloom filter (SBF) that looks for an input x in a set S, consists of a bit vector of length m, which is a compacted representation of the elements that are present in S, and k hash functions, which are used to map n inputs to the bit vector. An example is shown in Fig. 1, where m is 14 and k is 3. Before the SBF can be used, the bit vector is first loaded with zeroes. Then, all k hash functions are applied to each element s in the set S, i.e. $H_{k_i}(s)$ is calculated, with $i \in 1, \dots, k$. For each hash function, the resulting hash value is used as an index in the bit vector and the corresponding bit is set to 1. This process is repeated for each hash function, and subsequently for each possible $s \in S$. In Fig. 1, this means that 3 bits out of 14 are set to 1 for each loaded element s. In the figure, this process is illustrated for two elements s_1 and s_2. After these steps, the set S has been successfully loaded into the Bloom filter.

To query the SBF using input x, all hash digests $H_{k_i}(x)$ on x are calculated. Analogous to the initialization phase, the hash values are used as indices in the bit vector. If all the corresponding bits in the bit vector are set to 1, the SBF returns $x \in S$. Otherwise, $x \notin S$ is returned. As stated above, a membership query can have a false positive result. This happens when, during the initialization of the Bloom filter, the hash values of different elements in the set lead to a 1 being written to the same position in the bit vector. The probability of false positive rate of a standard bloom filter is approximated as in Eq. (1).

$$fpr = (1 - e^{-kn/m})^k \tag{1}$$

Unlike CAMs, Bloom filters do not explicitly store each element in a memory cell, but rather store the bit vector that represents all elements that are present in the data structure. This leads to more economical memory resource utilization.

2.2.1. Bloom-1 architecture

When the SBF structure in Fig. 1 is used, k locations in the bit vector need to be read out. In order to reduce the number of read-outs, a more efficient structure is proposed by Qiao et al. [13]. They propose Bloom-1, a fast Bloom filter architecture that only requires one read-out for each query. Bloom-1 uses hash functions to map an input element s to $log_2(l) + k \times log_2(w)$ bits. The first $log_2(l)$ bits are the output of the hash function $H_l(s)$ and are used as an address in a data structure that has l memory locations, as shown in Fig. 2. We call each word in the data structure a membership word. The next k values (which are $log_2(w)$ bits each) are the outputs of the hash functions $H_{k_i}(s)$, with $i \in 1, \dots, k$. They are used as addresses that point to bit locations in the membership word. The size of a membership word is w and a 1 is written at the bit locations that correspond to the addresses $H_{k_i}(s)$. To query an input x, a similar procedure is followed to read out k bits (of which the locations are determined by $H_{k_i}(x)$, with $i \in 1, \dots, k$) from the membership word at the address determined by $H_l(x)$. If all k bits are 1, the Bloom-1 filter returns $x \in S$. Otherwise, $x \notin S$ is returned.

Typically, w corresponds to the data bus width in a computer system, i.e., $w = 32$ or 64. The size of the entire Bloom-1 filter is denoted as $m = l \times w$. The advantage of Bloom-1 is that the number of required hash bits is significantly lower when k is large, compared to other Bloom filters [13]. The total number of hash bits for Bloom-1 is $log_2(l) + k \times log_2(w)$, compared to $k \times log_2(m)$ for SBF. Additionally, the w-bit membership word can be read out at once, while the k 1-bit values in SBF require k read-outs. This significantly increases the speed of Bloom-1 compared to SBF.

Qiao et al. give an approximation of the false positive rate (fpr) for Bloom-1 in [13]. A comment published by Reviriego et al. in [17] refines the approximation, resulting in Eq. (2). Besides the Bloom-1 parameters, l, w and k, the number of entries, n, has an influence on the false positive rate.

$$fpr = \sum_{x=0}^{n} \left(\binom{n}{k} \cdot \left(\frac{1}{l}\right)^x \cdot \left(1 - \frac{1}{l}\right)^{n-x} \cdot \left(\frac{w!}{w^{k(x+1)}} \sum_{i=1}^{w} \sum_{j=1}^{i} (-1)^{i-j} \frac{j^{kx} i^k}{(w-i)!\,j!\,(i-j)!} \right) \right) \tag{2}$$

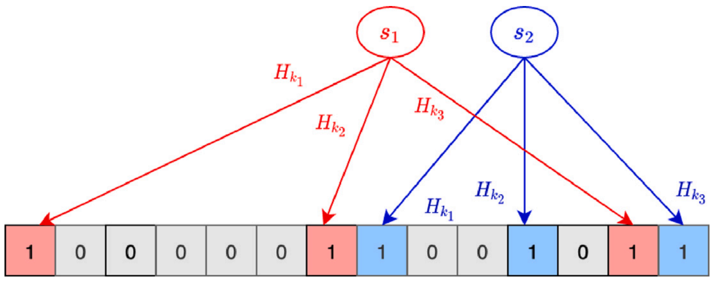

Fig. 1. Standard Bloom filter (SBF) representation.

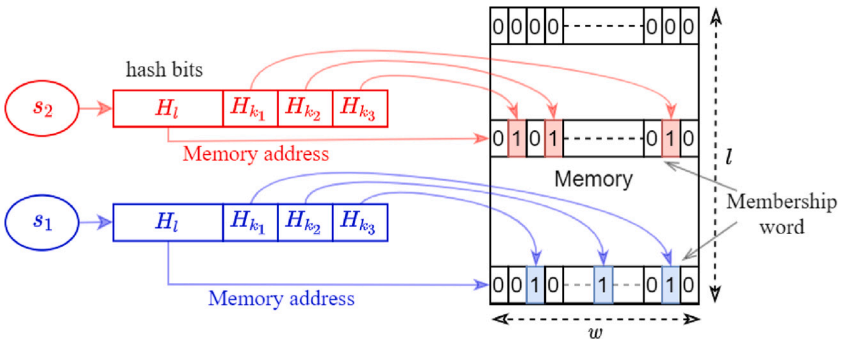

Fig. 2. Bloom-1 representation.

2.2.2. Parallel Bloom Filter (PBF) architecture

We propose a modified architecture named as Parallel Bloom Filter to keep the overall latency of Bloom filter to a single cycle. The parallel bloom filter (PBF) implementation approach is inspired from the work by Dharmapurikar [4]. Our approach follows the Standard Bloom Filter (SBF) model, but uses multiple smaller memory blocks to form SBFs in parallel to increase the lookup speed. When the number of hash functions k is set to 2 for each SBF, a Bi-SBF is formed, which requires two memory accesses per update/query. The Bi-SBF is termed as mini-bloom filter in the original literature and the authors of [4] suggest that a dual-port BRAM can be employed to perform 2 read/write operations at once, hence keeping the latency at one cycle. Moreover, to avoid write conflicts, it is better to assign one port for reading and the other port for writing, especially if the memory is being shared.

The Bi-SBFs can be arranged in parallel for two reasons, either to reduce the false positive rate, or to increase the string storage capacity. In order to reduce the false positive rate, the value of k and m has to be increased keeping the value of n constant, which is achieved by stacking the Bi-SBF blocks in parallel. Such Bi-SBFs stacked together in parallel form a parallel Bloom filter (PBF), as shown in Fig. 3(a). To form a parallel Bloom filter with k hash functions, $k/2$ SBFs are stacked together. The authors of [4] indicate that the false positive rate of a Bi-SBF is $(\frac{1}{2})^2$, considering a constant n/m ratio of 1419/4096, based on Eq. (1). The overall false positive rate of the PBF is then $(fpr(\text{Bi-SBF}))^{k/2}$.

In order to increase the storage capacity, Dharmapurikar et al. [4] suggest to stack the PBFs together to form a Large Bloom Filter (LBF) to achieve the required storage capacity. The storage capacity of a LBF formed by arranging p PBFs together is $n \times p$ while keeping the same false positive rate, where n is the string storage capacity of a single PBF. If the overall memory size of a PBF is m, then p such filters in parallel constitute a Bloom filter of memory size $p\times m$. An element will be stored in any one of the PBFs and one of the PBFs will be selected using a hash value. However, this architecture requires additional operational logic and adversely affects the resource requirements as well as speed.

Hence we prefer to change the memory size of the Bi-SBFs accordingly for the required false positive rate and storage capacity. To increase the storage capacity, m is increased while keeping the same number of SBFs in parallel. The query/configure operations are similar to standard Bloom filters, with every block operating in parallel. An element e is said to be present in the set only if all the parallel filters show a match. In our implementation, the false positive rate of the Bi-SBF is determined by Eq. (1) and the n/m ratio changes depending on the false positive rate or string storage requirements. The false positive rate of a Bi-SBF is $(1 - e^{-2n/m})^2$ and the equation $(fpr(\text{Bi-SBF}))^{k/2}$ still holds for the false positive rate of PBF. The total memory size of each Bi-SBF is $2 \times m/k$, where m is the total memory size of the Bloom filter.

To make the PBFs faster, the number of hash functions for each SBF can be set to 1, resulting in a Uni-SBF. The memory size of each Uni-SBF is m/k. The false positive rate of each such Uni-SBF adheres to Eq. (1) and is $(1 - e^{-n/m})$. The overall false positive rate of the PBF is given by $(fpr(\text{Uni-SBF}))^k$. The PBF based on the Uni-SBF architecture shown in Fig. 3(b) has the same false positive rate as PBF using Bi-SBF, but requires less hash bits. The PBF based on Uni-SBF can employ a dual port BRAM with dedicated ports for read and write operations, which can reduce the look up latency to only one cycle with all SBFs operating in parallel.

3. Related work

A wide range of related work is available, elaborating on the design and optimization of Bloom filters for different applications. Various aspects are to be considered when designing hardware architectures of Bloom filters, such as the lookup delay, the false positive rate and the operating frequency.

One of the main factors affecting the lookup delay of a Bloom filter is the execution delay of the hash functions. Cryptographic hash functions would have been the best choice if speed is not a prime concern. A number of fast non-cryptographic hash functions with good avalanche properties have been proposed, such as Murmur3 and FNV-1a. Murmur3 is the result of a general study of non-cryptographic hash functions by Estebanez et al. [18]. FNV-1a was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee by Fowler and Vo and later improved by Landon Curt Noll [14]. It is considered to have the smallest computational delay. However, on a high-speed hardware perspective, FNV-1a and Murmur3 are not

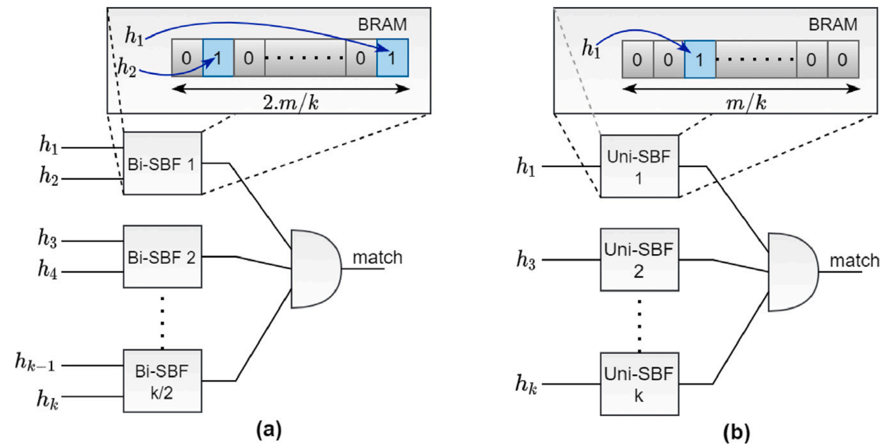

Fig. 3. Parallel Bloom filter logic (a) using Bi-SBF and (b) using Uni-SBF.

very efficient. In this work, we propose Xoodoo-NC, a fast hardware-oriented non-cryptographic hash function with good avalanche properties, which outperforms all other solutions in terms of execution delay in hardware. Xoodoo-NC is derived from the cryptographic permutation Xoodoo [12]. It uses a reduced number of rounds and a reduced size of the internal state, which leads to a lower logical depth than cryptographic hash functions based on Xoodoo, while maintaining the desired avalanche properties.

Besides optimizing the hash functions, there are other methods that contribute to the fast generation of the lookup addresses in Bloom filters. Examples are the one-hashing Bloom filter (OHBF) [19], the less hashing Bloom filter (LHBF) [20], and the ultra-fast Bloom filter (UFBF) [21]. OHBF and UHBF follow similar techniques, where a single base hash function is used, and k modulo operations are performed to obtain k addresses. LHBF employs two base hash functions, and a linear equation to generate additional addresses from the base hash functions. Although these techniques can reduce the generation delay of the lookup addresses, the overall delay of a query will still be dominated by the delay of the memory accesses when a standard Bloom filter (SBF) array [2] is used as depicted in Fig. 1. This is because each read/write operation will require one clock cycle when the embedded RAM in an FPGA is used for the storage of the Bloom filter. Hence, it is important to not only reduce the delay for the generation of the lookup addresses, but to also consider Bloom filter architectures that reduce the number of memory accesses.

In an SBF, the number of memory accesses is directly proportional to the number of hash values (k). In order to reduce the false positive rate, the number of hash functions and the size of the data structure (m) must be increased, but this has a negative effect on the execution speed. Parallel Bloom filters, as proposed by Dharmapurikar et al. [3], are a classical solution to overcome this issue. There are other approaches like multi-partitioning counting Bloom filters (MPCBF) [22], OMASS [23] and One-memory-access Bloom filters (Bloom-1) [13], where it takes only one memory access per query. MPCBF, OMASS and Bloom-1 employ similar approaches based on the partitioning of memory blocks to limit the number of memory accesses to one, as explained for Bloom-1 in Section 2. In this work, we do not focus on counting Bloom filters which store a count value for each memory cell in order to enable the deletion of elements. That is why MPCBF is not considered for our implementation. In OMASS, each element is represented in multiple sets with different hash mappings, which decreases the false positive rate but at the cost of extra memory. In this work, we concentrate on Bloom-1 [13], because it requires less memory storage than OMASS. For applications in which the false positive rate of Bloom-1 is not sufficiently low, the possibility of moving to an OMASS architecture can be considered.

A number of works are available on the implementation of Bloom filters in hardware. Dharmapurikar et al. [4] present an implementation of parallel Bloom filters on FPGA for string matching. The implementation is fast in terms of number of memory accesses with acceptable resource utilization, but has a very low operating frequency of 73.5 MHz, which is not suitable for high speed networks. However, an improved version of PBF is implemented in this work. In [24], a rolling hash function based Bloom filter is presented and implemented on a high-end FPGA for fast streaming data. However, the architecture assumes a byte-based interface which is only suitable for Gigabit Ethernet networks. When moving towards Terabit Ethernet, the incoming data need to be processed in larger parallel blocks. The work of Kaya et al. [25] and of Lyons et al. [26] focus mainly on the power reduction of Bloom filter implementations. A counting Bloom filter implementation proposed by [27] for network intrusion detection seems to provide operating frequencies above 200 MHz for an input data size up to 8 bytes. Although all these publications present very good implementations, an even higher parallelization level and/or operating frequency is needed in Terabit Ethernet networks.

In this work, we propose to use the existing Bloom-1 architecture and the modified parallel Bloom filter architecture, both of which requires only one memory access per query. We also propose Xoodoo-NC, an extremely fast non-cryptographic hash function that outputs a hash value large enough to be split into chunks that can directly be used in the Bloom filter architecture. We implement speed-optimized hardware architectures that outperforms all other FPGA-based Bloom filter implementations as well as architectures based on the CAM IP cores in Xilinx FPGAs.

4. Novel algorithms and architectures

4.1. Experimental setting and basis of comparison

We concentrate on Terabit Ethernet (defined as Ethernet with speeds above 100 Gbps). An example of an FPGA platform that supports Terabit Ethernet is the 200 Gbps NFB-200G2QL board [28], which processes network packets at a line rate of 512 bits per clock cycle. It is clear that implementing network security applications on Terabit Ethernet FPGA platforms requires the processing of many data bits in parallel. Parallelization is also suggested by the authors of the large flow detection algorithm EARDET [7]. An example of a network encryption implementation that adheres to the strong operating frequency and parallel processing requirements of Terabit FPGA platforms is given in [29].

Typically in network applications, flow identifiers need to be queried in a data structure. Examples of such applications are given in Section 1. The considered flow ID in this work contains the source IP

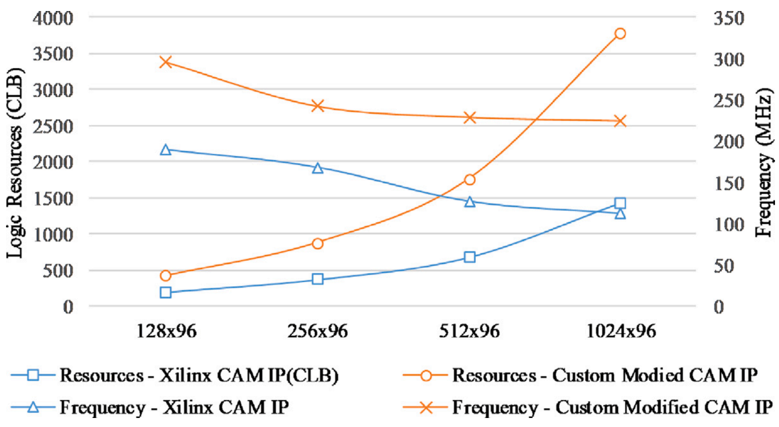

Fig. 4. Operating frequency and resource utilization of the CAM directly implemented with the Xilinx tools (referred to as Xilinx CAM IP) and the speed-optimized CAM architecture (Custom Modified CAM IP).

address, destination IP address, and both the source and the destination ports. The size of the flow ID is 96 bits, which is the case in an IPv4 network. To adhere to the requirements of Terabit Ethernet FPGA platforms, these 96 bits need to be processed in parallel at an operating frequency of at least 200 MHz. We compare six different lookup data structures in terms of cycle count, clock frequency, occupied FPGA resources and false positive rate:

1. CAM block generated by the design tools of Xilinx,
2. speed-optimized custom CAM consisting of manually combined smaller CAM blocks,
3. Bloom filter based on the PBF architecture and the FNV-1a hash function,
4. Bloom filter based on the PBF architecture and the newly proposed Xoodoo-NC hash function.
5. Bloom filter based on the Bloom-1 architecture and the FNV-1a hash function,
6. Bloom filter based on the Bloom-1 architecture and the newly proposed Xoodoo-NC hash function.

All six architectures store 1024 data words of 96 bits each. For the first two architectures, which are CAM-based, this means that exactly 1024×96 memory cells are occupied. For the last four architectures, which are based on Bloom filters, we explore different options for the size of the Bloom filter, false positive rate, and the number and output size of the hash functions. The number of stored entries in the Bloom filters is equal to 1024 to make a fair comparison with the CAM-based implementations.

The first CAM architecture is directly generated by the design tools of Xilinx, resulting in the utilization of embedded RAM (Block RAM or BRAM) with size 1024×96 bits, in combination with a limited amount of logic in the form of FPGA LUTs (lookup tables) and FFs (flip-flops). A lookup in the directly generated CAM takes 2 clock cycles. The second CAM architecture consists of 128 CAM blocks of size 24×32 bits. These smaller CAM blocks are again generated by the Xilinx design tools, and the LUTs and FFs for combining and addressing these blocks are added manually. This leads to an architecture with a higher operating frequency, but with a larger resource occupation compared to the directly generated CAM architecture. A lookup in the manually modified CAM takes 3 clock cycles. The operating frequency and resource utilization of the CAM-based architectures that are implemented as a reference for comparison, are given in Fig. 4. The third, fourth, fifth and sixth architectures that we evaluate are Bloom filters, which differ in the utilized hash function. We compare FNV-1a, the fastest hash function in hardware according to our knowledge, to Xoodoo-NC, the proposed hash function in this paper.

For all experiments, the Vivado 2017.4 design tool of Xilinx is used and the UltraScale+ XCVU7P-FLVC2104-1-E FPGA is targeted. Only for the generation of the CAM blocks, the ISE 14.7 design tool of Xilinx is used, because CAM generation is no longer supported in Vivado. The resulting code is imported in Vivado to be used in the UltraScale+ FPGA (which is not supported in ISE 14.7).

4.2. Design of a high-speed non-cryptographic hash function

The optimization goals in the design of a non-cryptographic hash function for Bloom filters are low logical depth or critical path (i.e., high operating frequency), low cycle count, high avalanche score and acceptable resource utilization. FNV-1a [14] and Murmur3 [18] are non-cryptographic hash functions with excellent avalanche properties. However, both of these algorithms use multiplication operations in which the width of the multiplier is proportional to the output hash size. FNV-1a and Murmur3 process 8 bits and 32 bits per cycle, respectively, which means the execution time is proportional to the number of 8-bit or 32-bit input blocks. That is why FNV-1a and Murmur3 perform very well on an 8-bit or 32-bit microprocessor. When implemented in hardware, however, the high number of flip-flops to store the intermediate values and the large multipliers have a negative effect on the resource utilization, the operating frequency and the execution delay, especially when the required output hash size is large.

In our effort to find a non-cryptographic hash function that has a low logical depth and a low resource utilization in hardware, we start from the hardware-friendly cryptographic permutation Xoodoo, presented by Daemen et al. in [12]. A Xoodoo round employs only shift, AND and XOR operations, and does not need flip-flops inside the round. While Xoodoo uses a 384-bit state, Xoodoo-NC operates on a 96-bit state, which perfectly fits our needs to process incoming network flow IDs of 96 bits. Output values can be taken as multiples of 96-bits. A more detailed explanation of Xoodoo-NC is given in the following paragraphs.

The original Xoodoo permutation is parameterized by the number of rounds n_r, and it iteratively applies the round function R_i to the state A. The state A is depicted in Fig. 5 and has a size of 48 bytes, divided into three 2-dimensional planes, which are indexed by y. Each plane has four 1-dimensional lanes which are indexed by x. Each lane consists of 32 bits in size, indexed by z. A collection of three parallel stacked lanes in A is called a sheet, and any three parallel stacked bits in A are called a column. Each byte in the state can thus be referred to with the coordinates (x, y, z).

The round function R_i of Xoodoo consists of 5 sequential steps: a mixing step θ, a plane shifting step ρ_{west}, a step for the addition of round constants ι, a non-linear layer χ, and a second plane shifting step ρ_{east}. For more in-depth details on these steps and the round constants, we refer to [12].

In this work, we require a hash function with an input size of 96 bits or 12 bytes. Therefore, we reduce the state A of the Xoodoo

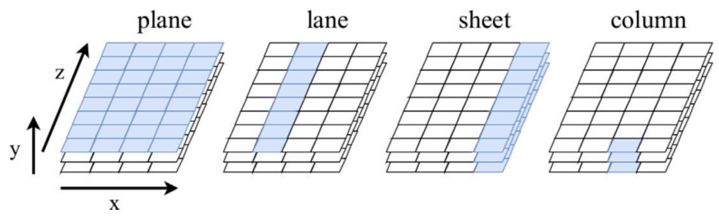

Fig. 5. Graphical representation of the Xoodoo state.

permutation to 12 bytes, corresponding to one sheet. This fixes the x coordinate and allows the y coordinate to vary from 0 to 2 and the z coordinate from 0 to 31. The original definitions of lanes and columns still hold. Because of the reduction of the size of the state, the Xoodoo round function is slightly adapted. The cyclic shift operations involving the x and z coordinates are replaced by operations involving only z. The other operations and the round constant C_i remain the same as in the original algorithm. The resulting Xoodoo-NC permutation is specified in Alg. 1, and the notational conventions of Xoodoo-NC is specified in Table 1.

Algorithm 1: Xoodoo-NC[n_r]

```
Parameters: Number of rounds $n_r$
for Round index $i$ from $1 - n_r$ to 0 do
    $A = R_i(A)$ ;        // $A$ is 96-bit input state (1 sheet)
end
    Round $R_i$:
        $\theta$:
                $P \leftarrow A_0 \bigoplus A_1 \bigoplus A_2$
                $E \leftarrow P \lll 5 \bigoplus P \lll 14$
                $A_y \leftarrow A_y \bigoplus E$ for $y \in \{0,1,2\}$
        $\rho_{west}$:
                $A_2 \leftarrow A_2 \lll 11$
        $\iota$:
                $A_0 \leftarrow A_0 \bigoplus C_i$
        $\chi$:
                $B_0 \leftarrow \overline{A_1}.A_2$
                $B_1 \leftarrow \overline{A_2}.A_0$
                $B_2 \leftarrow \overline{A_0}.A_1$
                $A_y \leftarrow A_y \bigoplus B_y$ for $y \in \{0,1,2\}$
        $\rho_{east}$:
                $A_1 \leftarrow A_1 \lll 1$
                $A_2 \leftarrow A_2 \lll 8$
```

For the Xoodoo permutation, Table 8 in [12] summarizes the avalanche properties. The avalanche metrics mentioned in [12] are used here also to determine the avalanche behavior of Xoodoo-Nc. The metrics, namely avalanche dependence (D_{av}), avalanche weight ($\overline{w}_{av}$), and avalanche entropy (H_{av}), are calculated as in [12] for single-bit differences at the input. Only the worst-case values are reported, which are the minimum values taken over all individual input differences of a given type. The results are presented in Table 2. Xoodoo-NC provides full bit-dependence and quasi-strict avalanche weight after 2.5 rounds, and $\approx$ 90% dependence after round 2. We assume that it is sufficient for our intended application and generate a 96-bit Xoodoo-NC hash output after 2.5 rounds. To obtain an output of 192 bits, one additional round is executed and the output of the next round is concatenated with the output of the previous round. In a similar way, an output size of any multiple of 96 bits can be obtained by increasing the number of rounds and concatenating the results.

Xoodoo-NC proves to be a better compared to FNV-1a and Murmur3 when it comes to high-speed applications and if the required output bits are high. Table 3 summarizes the hardware implementation results of FNV-1a, Murmur3 and Xoodoo-NC on a Xilinx Virtex Ultrascale+ FPGA (XCVU7P-FLVB2104-2-i). The throughput is calculated using the

Table 1
Notational conventions of Xoodoo-NC.

A_y	Lane y of state A. In Xoodoo-NC, A is one sheet.
$A_y \lll v$	Cyclic shift of A_y moving bit in z to position $(z+v)$
$A_y \bigoplus B_y$	Bitwise sum (XOR) of planes A_y and B_y
$A_y.B_y$	Bitwise product (AND) of planes A_y and B_y
$\overline{A_y}$	Bitwise complement of lane A_y

Table 2
Avalanche Scores for Xoodoo-NC.

Rounds	D_{av}	$\overline{w}_{av}$	H_{av}
2	84	35.408	80.332
2.5	96	47.324	95.864
3	96	47.309	95.867
3.5	96	47.922	95.996

equation $\frac{Input\ block\ size}{latency\ in\ cycles} \times f_{max}$, where f_{max} is the maximum operating frequency. As seen from the table, Xoodoo-NC has improved the frequency of operation and latency significantly compared to both Murmur3 and FNV-1a despite having a higher output block size. A work by Sateesan et al. [30] on the CM sketch implementation on FPGA has shown the effectiveness of Xoodoo-NC on hardware. Following the design technique of Xoodoo-NC, Claesen et al. [31] constructed fast non-cryptographic hash functions from lightweight cryptographic block ciphers chosen from NIST lightweight competition. These non-cryptographic hashes derived from lightweight block ciphers significantly outperforms any other non-cryptographic hashes and Xoodoo-NC exhibits the best performance in terms of throughput per area.

5. Hardware implementation

We discuss the hardware implementation of the Bloom-1 architectures that are evaluated in this work. We employ a single non-cryptographic hash function of which the output is split into $k+1$ parts, i.e., one part of size $log_2(l)$ and k parts of size $log_2(w)$, as explained in Section 2.2.1. Two non-cryptographic hash functions with an input size of 96 bits are evaluated, namely FNV-1a and Xoodoo-NC. In the following paragraphs, we elaborate on the FNV-1a implementation, the Xoodoo-NC implementation and the Bloom-1 implementation that uses one of these hash functions.

5.1. Hash functions

5.1.1. FNV-1a

FNV-1a can provide output hash sizes ranging from 32 to 1024 bits. It uses only two parameters — a non-zero FNV Offset basis, and an FNV Prime. Both parameters depend on the hash output size. FNV-1a processes one byte at a time, and the algorithm consists of iterations of two sequential operations — XOR and multiplication.

The hardware block diagram of FNV-1a is shown in Fig. 6. The input can be of any size, but only 8 bits are processed per cycle. The received input is stored in an octet shift register, which shifts a single byte per cycle and has a total size equal to the size of the incoming message, namely q bytes. In this work, the size of the flow ID is taken as 96 bits, which corresponds to $q = 12$. Every clock cycle, a byte is passed to

Table 3
Comparison of hash implementations on a Xilinx Virtex Ultrascale+.

Function	Block size (In/Out bits)	LUT	FF	BRAM	DSP	Frequency	Latency (ns)	Throughput (Gbps)
FNV-1a	96/32	38	70	0	2	211.01 MHz	61.6	1.56
	96/64	76	161	0	7	160.23 MHz	81.1	1.18
Murmur3	96/32	574	277	0	24	153.87 MHz	65.0	1.48
	96/64	566	341	0	38	120.58 MHz	82.9	1.16
Xoodoo-NC (2.5 rounds)	96/96	256	194	0	0	666.67 MHz	1.5	64
	96/192	323	290	0	0	416.67 MHz	2.4	40

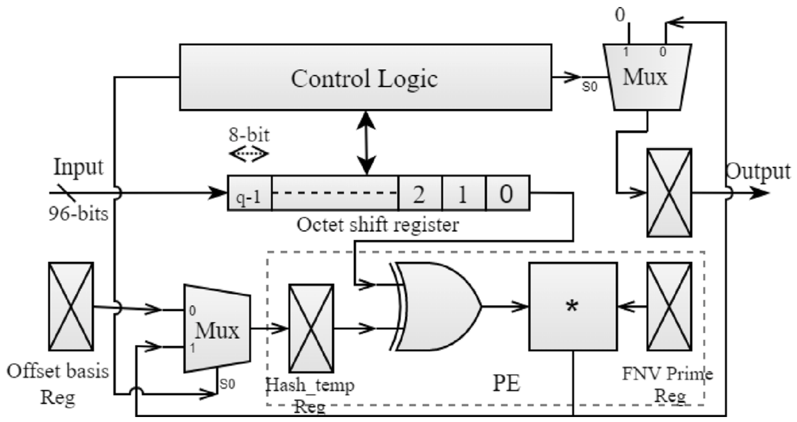

Fig. 6. FNV-1a — serialized hardware architecture block diagram.

the processing element (PE), which performs the XOR operation with the intermediate hash value, or with the FNV Offset basis for the first byte. Subsequently, a multiplication with the FNV Prime is calculated. The hash_temp register stores the intermediate hash value. The output is obtained after processing the final byte of the input. Before the hash value is valid, zeroes are sent to the output.

The drawback of a straightforward FNV-1a architecture is that it processes only one byte per clock cycle. The input needs to be buffered until the processing is completed. This is a major concern for high-speed networks like Terabit Ethernet networks, in which multiple bytes need to be processed in each clock cycle. For the applications that we consider in this paper, 12 bytes need to be processed every clock cycle, but the FNV-1a architecture requires 12 clock cycles to process 12 bytes. Another concern regarding the hardware implementation is that one of the core operations in the PE is a multiplication. If we want to increase the input size and the output hash size, the multiplier width also needs to be increased, which further increases the hardware complexity and the critical path. Pipelining somewhat helps to mitigate this issue and to improve the overall throughput (after an initial delay equal to q clock cycles. The hardware architecture of the pipelined version is shown in Fig. 7.

The core of the design is a free-running FSM, which processes q bytes in each clock cycle. The input buffers store the incoming q-byte inputs. In total, q input buffers are implemented that each store a q-bit input word. In each clock cycle, the bytes from the input buffers (starting from register 0) are shifted to the octet register of size q bytes. The octet register is an array of 8-bit registers. For every input, the first byte is stored in the 0th register, the second byte in the 1st register and so on. Each PE takes two inputs, one byte from the octet register and the output from the previous PE (except for PE_0). PE_0 takes the FNV Offset basis and the 0th byte of the octet register as inputs. The output of each PE is connected to an output buffer that stores the intermediate hash value and is connected to the input of the next PE in the sequence. The pipelined version of FNV-1a can process a 96-bit input in every clock cycle and has an output latency of 12 clock cycles.

5.1.2. Xoodoo-NC

The architectural features of Xoodoo-NC are depicted in Section 4.2. The hardware block diagram of Xoodoo-NC is shown in Fig. 8. The heart of the Xoodoo permutation is the round function, which consists of the 5 steps shown in Alg. 1: θ, ρ_{west}, ι, χ, and ρ_{east}. In general, the round steps are applied sequentially n_r times on the input state function A. In Xoodoo-NC, the round is executed twice when a 96-bit hash output is needed and once more to generate a second 96-bit output, to be concatenated with the first output when a 192-bit hash output is needed. This way, the output of Xoodoo-NC can be any multiple of 96 bits, but we stick to two different output sizes in our evaluation: 96 and 192 bits.

The rounds are implemented in a fully unrolled architecture without registers in between the rounds, i.e., the 2 or 3 rounds of Xoodoo-NC are completely implemented in combinational logic. The detailed hardware block diagram of the round function is shown in Fig. 9. The 96-bit input A consists of 3 lanes of size 32 bits, namely A_0, A_1, and A_2 (cfr. Alg. 1). The round constant C_i used in ι is a single 32-bit lane, indexed from $1-n_r$ to 0. The 3-lane output of the round goes to the input of the next round.

5.2. Bloom-1

The hardware architecture of the Bloom-1 filter follows the concept depicted in Section 2.2.1 and is shown in Fig. 10. Bloom-1 implementation employs only BRAMs for storage. The hash bits are generated with a single hash function, Xoodoo-NC, of which the output is split into $k+1$ parts. The first part, H_l, is used to determine the location of the membership word in the BRAM. The k remaining parts are used to address the bits inside the membership word. Depending on the $query/insert$ instruction, the memory is read and written. The execution delay is determined by the BRAM read–write delay, which is 1 clock cycle, and the delay of the hash generation. Xoodoo-NC is executed in one clock cycle, while the non-pipelined (serialized) and pipelined versions of FNV-1a take 14 and 2 clock cycles, respectively.

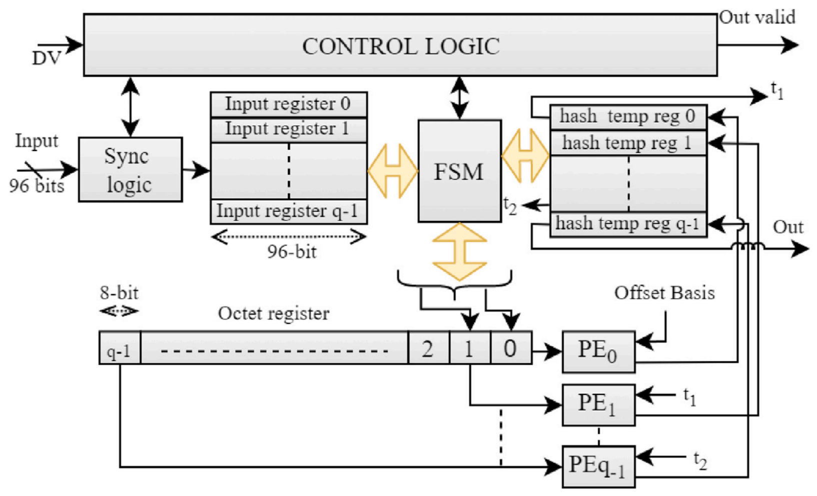

Fig. 7. FNV-1a - Pipelined hardware architecture block diagram.

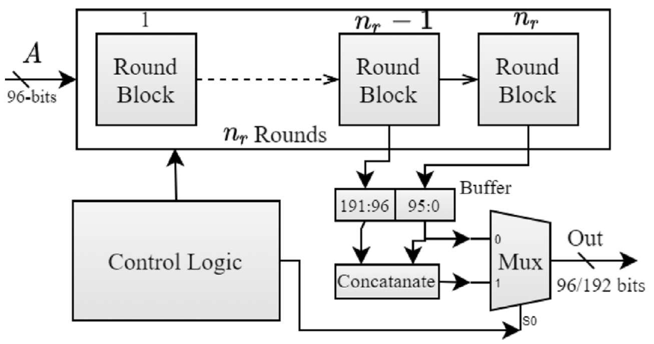

Fig. 8. Xoodoo-NC block diagram.

5.3. Parallel Bloom filter

Parallel bloom filters (PBF) are constructed by stacking Bi-SBFs or Uni-SBFs in parallel as depicted in Section 2.2.2 and the hardware architecture of PFF using Bi-SBFs is shown in Fig. 11. Only on-chip BRAMs are employed to implement the memory. Dual port BRAMs are employed with dedicated ports for read and write operations. Xoodoo-NC is used to generate the hash values. The SBFs are operated in parallel such that the read/write latency is always equal to the latency of a single SBF. Using Bi-SBF, the latency is 3 cycles whereas using Uni-SBF, the latency is only 2 cycles. The memory sizes of Bi-SBF and Uni-SBF are $2m/k$ and m/k, respectively, where m is the overall size of the Bloom filter and k is the number of hash functions. Dividing the memory block into multiple smaller blocks in PBF helps to minimize the routing delays associated with larger BRAM blocks.

6. Results & analysis

In this section, we present the results of the experiments conducted on a Xilinx Virtex Ultrascale+ FPGA (XCVU7P-FLVC2104-1-E) using the Vivado 2017.4 design tool. For the Bloom filter implementations, we employ FNV-1a with 64-bit and 128-bit hash outputs, and Xoodoo-NC with 96-bit and 192-bit hash outputs. The input size is always 96 bits. For FNV-1a, both the non-pipelined and the pipelined architectures are evaluated. The Bloom-1 parameters l, w, and $m = l * m$ are taken as 4096, 64 and 262144, respectively. For the PBF implementations, the values of m and k are varied to match the fpr of Bloom-1. The size of the CAMs that we compare with is 1024 × 96.

6.1. Effect of hash functions on Bloom-1

The usage of FPGA logic and DSP units in the Bloom-1 architectures while using different hash functions is depicted in Fig. 12. The Xoodoo-NC based Bloom filters outperform the FNV-1a based implementations, especially for larger values of k. The FNV-1a implementations make use of the multipliers inside the DSP units, whereas no DSP units are employed in Xoodoo-NC. The pipelined version of FNV-1a has a much higher consumption of logic resources and DSP units in comparison to the serial FNV-1a and the Xoodoo-NC based Bloom filters. The operating frequency and execution delay of the Bloom-1 architectures are shown in Fig. 13. The Xoodoo-NC based architectures clearly feature a much higher operating frequency and a much lower execution delay than the FNV-1a based architectures.

The results of the Bloom filter implementations are listed in Table 4, and in Figs. 12, and 13. Two versions of the Bloom-1 architectures are considered in the table: those with $k = 2$ and $k = 12$. In order to calculate the false positive rate (fpr) using Eq. (2), the number of entries in the Bloom filter is assumed to be 1024, such that a fair comparison can be made with the CAMs whose depth is taken as 1024. Going from $k = 2$ to $k = 12$, leads to a resource occupation that is more than doubled for the Xoodoo-NC based implementation, and at the same time decreases the false positive rate drastically. Nevertheless, the resource occupation of the Xoodoo-NC based Bloom filter is much lower than all other considered architectures, even when $k = 12$.

6.2. Bloom filter versus CAM

Table 4 shows the comparison of the bloom filter variants with CAM based lookup architectures. The number of entries to be stored is taken as 1024. The Xilinx CAM blocks mentioned in [16,32] require 40x-50x more BRAMs than the Bloom-1 architecture, and requires ≈20 times more LUTs than the PBF architecture. The CAM block having a size of 1024 × 96 generated by the Xilinx design tools has an operating frequency of only 112.75 MHz. The search delay of CAM increases as the number of entries to be stored increases, but the number of entries to be stored barely affects the search delay of Bloom filters as the number of locations to be searched remains a constant and is equal to the number of hash functions. The custom modified CAM architecture operates at a higher frequency of 225 MHz, but consumes more FPGA resources as depicted in Fig. 4 and Table 4. Although the latency is increased by 1 cycle to query the custom modified CAM, the total query time is lower compared to the directly generated CAM. The lookup delay of the Xoodoo-NC based architectures is significantly lower than the lookup delay of all other implemented architectures. This is mainly because of the high operating frequency, and thanks to the cycle count of only 3 cycles.

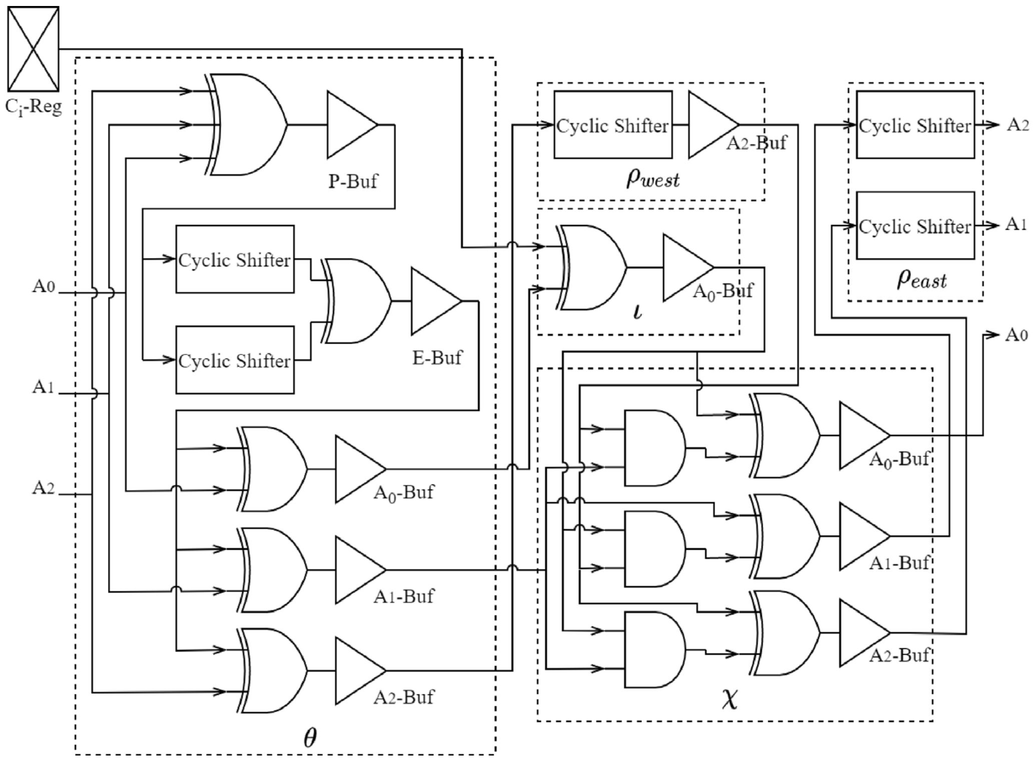

Fig. 9. Xoodoo-NC Round, which is implemented 2 or 3 times in combinational logic for a 96-bit or 192-bit hash output, respectively.

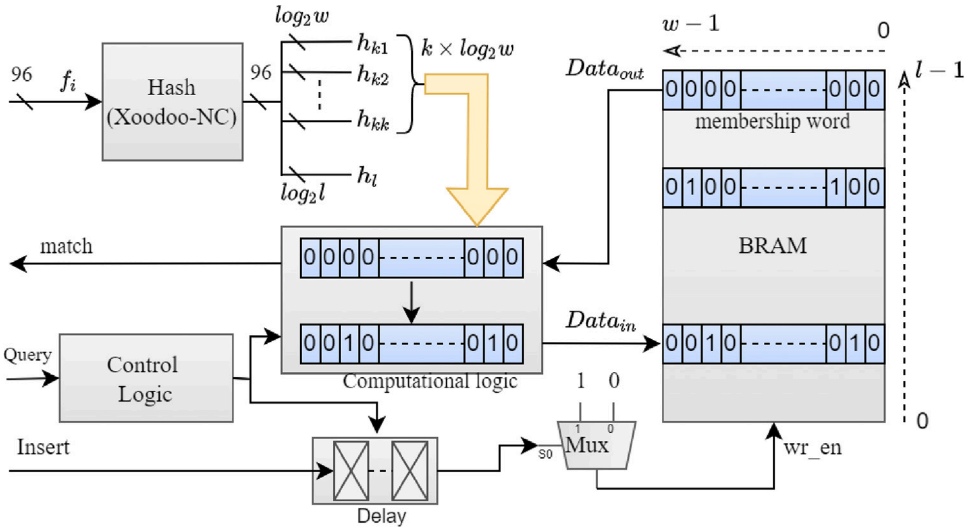

Fig. 10. Bloom-1 hardware architecture.

Table 4
Comparison of Bloom-1, PBF, and CAM implementations on a Xilinx Virtex Ultrascale+.

Bloom filter	k	fpr	LUTs	FFs	BRAM	LUTRAM	DSP	Max freq [MHz]	No. of cycles for lookup	Lookup Delay [ns]
Bloom-1 (Xoodoo-NC)	2	0.0002976	263	34	7.5	0	0	490.92	2	4.074
Bloom-1 (FNV-1a)	2	0.0002976	261	228	7.5	0	3	173.52	16	92.21
Bloom-1 (FNV-1a) Pipelined	2	0.0002976	649	1507	7.5	6	36	151.52	4	24.76
Bloom-1 (Xoodoo-NC)	12	2.61×10^{-7}	643	94	7.5	0	0	462.32	2	**4.124**
Bloom-1 (FNV-1a)	12	2.61×10^{-7}	1366	339	7.5	0	15	121.18	16	132.03
Bloom-1 (FNV-1a Pipelined)	12	2.61×10^{-7}	2706	1562	7.5	6	180	104.87	4	38.14
PBF (Uni-SBF,Xoodoo-NC)	12	0.14×10^{-7}	338	220	6	0	0	434.21	2	4.606
PBF (Bi-SBF,Xoodoo-NC)	12	0.14×10^{-7}	324	148	6	0	0	389.10	3	7.71
Direct CAM		0.0	6731	1145	320	1536	0	112.75	2	17.74
Custom Modified CAM		0.0	20’797	988	384	1536	0	225.07	3	13.33

Table 5
Comparison with related work.

Design	m	k	Input size	LUTs	FFs	BRAMs	Frequency	fpr	No. of memory accesses	FPGA
PBF [4]	4096	10	10 byte	1495	1297	5	73.51 MHz	$8.89x10^{-5}$	2	Virtex 2000E
[27](SBF)	16384	8	8 byte	1058	1058	4	200.6 MHz	$5.74x10^{-4}$	4	Virtex-4
[27](CBF)	16384	8	8 byte	1188	1188	4	201.6 MHz	$5.74x10^{-4}$	4	Virtex-4
Ours, PBF (Bi-SBF)	49152	12	12 byte	338	220	6	389.10 MHz	$0.14x10^{-7}$	2	Virtex UltraScale+
Ours, PBF (Uni-SBF)	49152	12	12 byte	324	148	6	434.21 MHz	$0.14x10^{-7}$	1	Virtex UltraScale+
Ours, Bloom-1(xoodoo-NC)	262144	12	12 byte	675	158	7.5	462.32 MHz	$2.61x10^{-7}$	1	Virtex UltraScale+

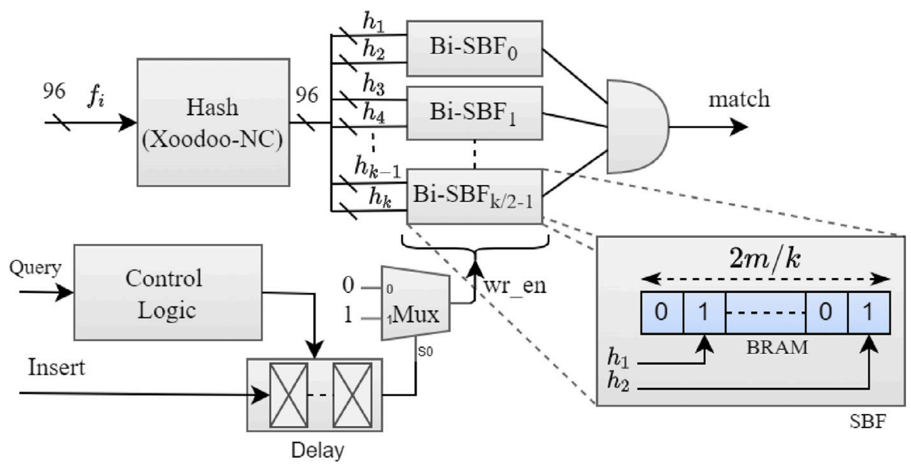

Fig. 11. Parallel Bloom Filter hardware architecture.

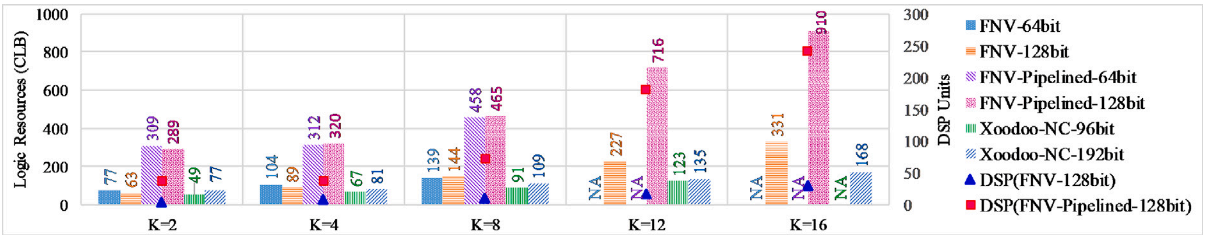

Fig. 12. Resource utilization of Bloom-1: Logic resources and DSP units.

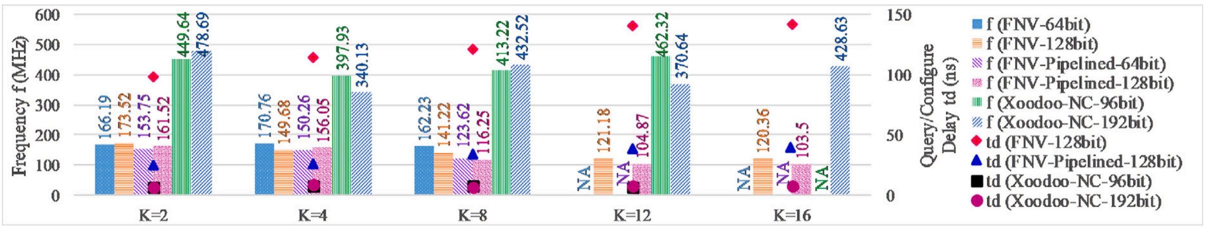

Fig. 13. Maximum operating frequency and delay of Bloom-1.

6.3. Analysis of Bloom filter variants

Analyzing the different bloom filter implementations, we can observe that there is always a trade-off between the false positive rate and memory utilization. To analyze this, we consider the Bloom-1 architecture and the parallel Bloom filter architectures, as proposed in Section 2.2.2. Table 6 shows the comparison between Bloom-1, parallel Bloom filter (PBF), and standard Bloom filter (SBF). When comparing Bloom-1 with PBF, it can be noticed from Tables 4 and 6 that PBF can achieve a better false positive rate with less memory, but at the cost of a higher number of hash bits. The Bloom-1 architecture with 12 hash functions requires only 84 hash bits, whereas PBF using Uni-SBF requires 144 bits to achieve a better *fpr*. However, Bloom-1 requires ≈ 5 times more memory than PBF. Comparing standard Bloom filter (SBF) and PBF, SBF requires more memory while keeping the number of hash bits the same. PBF using Uni-SBF shows less hash bit requirements and higher frequency compared to PBF using Bi-SBF. The number of cycles for lookup given in Tables 4 and 6 includes the number of clock cycles required for both hashing and memory accesses.

6.4. Power analysis

Power consumption is another important factor to be addressed. Our implementation results from Vivado 2017.4 underline the statement that CAM architectures are power hungry, when comparing with Bloom filters. The static power consumption of the FPGA occupies the lion's

Table 6
Comparison of the properties of the Bloom filter architectures considered in this paper.

Bloom filter	*f pr*	k	n	m	Hash bits	No. of cycles for lookup
Bloom-1	$2.61x10^{-7}$	12	1024	262144	84	2
Keeping k constant while achieving similar *fpr* as Bloom-1						
PBF (Bi-SBF)	$0.14x10^{-7}$	12	1024	49152	156	3
PBF (Uni-SBF)	$0.14x10^{-7}$	12	1024	49152	144	2
SBF	$8.74x10^{-7}$	12	1024	32768	192	13
Keeping hash bits constant while achieving similar *fpr* as Bloom-1						
PBF (Bi-SBF)	$0.49x10^{-7}$	6	1024	98304	90	3
PBF (Uni-SBF)	$0.49x10^{-7}$	6	1024	98304	84	2
SBF	$0.82x10^{-7}$	5	1024	131072	85	6

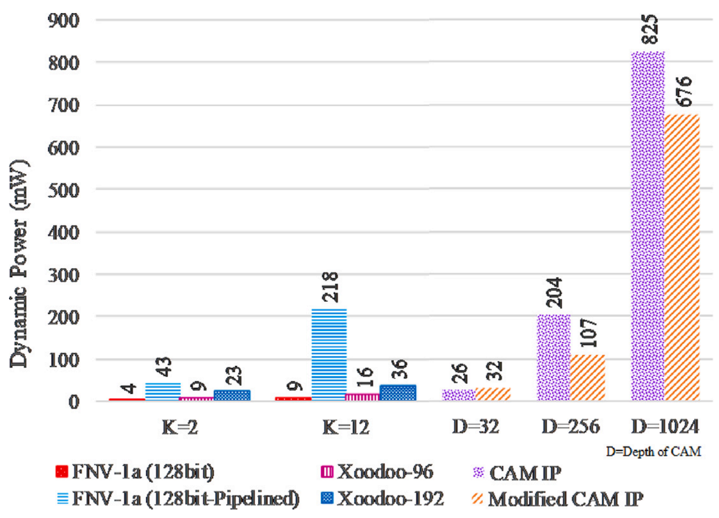

Fig. 14. Dynamic power consumption of Bloom-1 (Using FNV & Xoodoo-NC), and CAM.

share of the total power consumption for all implementations. Static power consumption is almost equal for all the Bloom-1 and CAM implementations and there is only a maximum increase of 0.6% in static power consumption for CAM compared to Bloom-1. The dynamic power consumption of Bloom-1 is only less than 1% of the total power consumption for FNV-1a and Xoodoo-NC based implementations. However, dynamic power consumption of Bloom-1 employing the pipelined version of FNV-1a can rise up to 12% of the total power consumption. The dynamic power consumption of Bloom-1 based on FNV-1a and Xoodoo-NC is negligibly small compared to CAMs as shown in Fig. 14. Although, the dynamic power consumption of pipelined FNV-1a based Bloom-1 increases with increasing value of k, it is still significantly lower than the power consumption of a CAM. As the CAM lookup operations are performed in parallel, the dynamic power consumption of CAM having a depth of 1024 is 33% of the total power consumption, which is significantly larger when compared to <1% of Xoodoo-NC based Bloom-1 implementation. Another noticeable thing to be mentioned is that our modified CAM implementation consumes less power compared to the Xilinx IP CAM.

6.5. Comparison with related work

A quantitative comparison with related work is shown in Table 5 for n=1024. It is clear that our Xoodoo-NC based Bloom-1 architecture with a 96-bit input and $m = 4096 * 64$ significantly outperforms previously proposed FPGA architectures that have a smaller input size, a smaller m and a larger false positive rate. Our design drastically improves the resource occupation and the speed. Although the total lookup delay is not reported for the considered related architectures, the table shows that the delay of the memory accesses only is longer than the delay of one query in our Xoodoo-NC based implementation. Note, however, that the considered related work uses older FPGAs with 4-input LUTs, while our FPGA has 6-input LUTs. Nevertheless, our reported number of LUTs is only around half of the LUTs reported in related work and the number of FFs is roughly one fifth to one tenth of the FFs reported in related work. The difference in operating frequency can partially be attributed to the difference in silicon technology nodes between older and newer FPGAs. Nevertheless, the frequency of 462 MHz is extremely high thanks to the low logical depth of the Xoodoo-NC hash function.

6.6. Future-proofing

The present work targeted IPv4 as network-layer protocol. However, the presented architectures are also adaptable for IPv6. The Bloom filter algorithm is based on the hashed fingerprint and the algorithm is not affected by the size of the input (flow ID). The false positive rate of the Bloom filter is only a function of the number of hash functions, number of elements to be inserted and the total memory size. Therefor, it is not affected by the size of the input or hash collisions, irrespective of the network-layer protocol.

To handle IPv6 flows, the only change to be made is to adapt the hash function to handle IPv6 flow ids, which can be done by increasing the input size of Xoodoo-NC. The original Xoodoo permutation has a 384-bit input state and in Xoodoo-NC we used only 96-bit input state as our requirement was only 96-bits. We can modify the Xoodoo-NC hash to accept either 192-bit (2 sheets of the Xoodoo state) or 288-bit (3 sheets) as input state, and then re-compute the number of rounds required to meet the avalanche criteria. The latency can still be a single clock cycle and the computational resources will be slightly higher compared to Xoodoo-NC with a 96-bit input state. This would still be more advantageous than any other existing non-cryptographic hashes because as the input size becomes large, the latency and resource utilization of hashes like FNV-1a and Murmurhash increase drastically.

7. Conclusion

In this work, we targeted novel algorithms and architectures for high-speed Bloom filters on FPGA, used for fast lookups in network applications. We proposed a new high-speed hardware-oriented non-cryptographic hash function called Xoodoo-NC. The hash function was integrated into Bloom-1 and parallel Bloom filter architectures and evaluated on a Virtex UltraScale+ FPGA of Xilinx. The resulting resource occupation, power consumption, operating frequency, lookup delay and false positive rate were compared to previously reported Bloom filter architectures and Content-addressable Memories on FPGA. To our knowledge, our work significantly outperforms all other solutions on all these comparison criteria. Finally, we discussed the presented work that it can comply with another protocol stack.

Acknowledgment

This work is supported by the ESCALATE project, funded by FWO, Belgium (G0E0719N) and SNSF, Switzerland (200021L_182005). This work is also supported by CyberSecurity Research Flanders with reference number VR20192203. Joan Daemen is supported by the European Research Council under the ERC advanced grant agreement under grant ERC-2017-ADG Nr. 788980 ESCADA. We thank Seth Hoffert and the co-designers of Xoodoo, Gilles Van Assche and Ronny Van Keer, for their valuable assistance.

References

[1] G. Varghese, Network algorithmics: An interdisciplinary approach to designing fast networked devices, 2004.
[2] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[3] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, J.W. Lockwood, Deep packet inspection using parallel bloom filters, IEEE Micro 24 (1) (2004) 52–61.
[4] S. Dharmapurikar, M. Attig, J. Lockwood, Design and implementation of a string matching system for network intrusion detection using FPGA-based bloom filters, in: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '04, 2004.
[5] A.Z. Broder, M. Mitzenmacher, Network applications of bloom filters: A survey, Internet Math. (2004).
[6] L. Luo, D. Guo, Richard T.B. Ma, O. Rottenstreich, X. Luo, Optimizing bloom filter: Challenges, solutions, and comparisons, 2018, arXiv:1804.04777.
[7] C. Estan, G. Varghese, New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice, ACM Trans. Comput. Syst. (2003).
[8] A. Kumar, J. Xu, J. Wang, O. Spatschek, L. Li, Space-code bloom filter for efficient per-flow traffic measurement, in: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, 2003, pp. 167–172.
[9] H. Wu, H.C. Hsiao, Y.C. Hu, Efficient large flow detection over arbitrary windows: An algorithm exact outside an ambiguity region, in: Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC, 2014, pp. 209–222.
[10] Simon Stanley, FPGAS & ASICs for telecom, 2015, Heavy Reading, http://www.heavyreading.com/details.asp?sku_id=3418&skuitem_itemid=1657.
[11] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, 2001.
[12] J. Daemen, S. Hoffert, G. Van Assche, R. Van Keer, The design of xoodoo and xoofff, IACR Trans. Symmetric Cryptol. (2018) 1–38.
[13] Y. Qiao, T. Li, S. Chen, Fast bloom filters and their generalization, IEEE TPDS 25 (1) (2014) 93–103.
[14] G. Fowler, L.C. Noll, K.-P. Vo, The FNV non-cryptographic hash algorithm, 2012, https://tools.ietf.org/html/draft-eastlake-fnv-03.
[15] A. Sateesan, J. Vliegen, J. Daemen, N. Mentens, Novel bloom filter algorithms and architectures for ultra-high-speed network security applications, in: 2020 23rd Euromicro Conference on Digital System Design, DSD, 2020, pp. 262–269.
[16] K. Locke, Parameterizable Content-Addressable Memory, https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf.
[17] P. Reviriego, K. Christensen, J.A. Maestro, A comment on fast bloom filters and their generalization, IEEE Trans. Parallel Distrib. Syst. 27 (1) (2016) 303–304.
[18] C. Estébanez, Y. Sáez, G. Recio, P. Isasi, Performance of the most common non-cryptographic hash functions, Softw. - Pract. Exp. 44 (2014).
[19] J. Lu, T. Yang, Y. Wang, H. Dai, L. Jin, H. Song, B. Liu, One-hashing bloom filter, in: Proc. IEEE/ACM IWQoS, 2015.
[20] A. Kirsch, M. Mitzenmacher, Less hashing, same performance: Building a better bloom filter, Random Struct. Algorithms 33 (2) (2006) 187–218.
[21] J. Lu, Y. Wan, Y. Li, C. Zhang, H. Dai, Y. Wang, G. Zhang, B. Liu, Ultra-fast bloom filters using SIMD techniques, in: Proc. IEEE/ACM IWQoS, 2017.
[22] K. Huang, et al., A multi-partitioning approach to building fast and accurate counting bloom filters, in: IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 1159–1170.
[23] M. Mitzenmacher, P. Reviriego, S. Pontarelli, OMASS: One memory access set separation, IEEE Trans. Knowl. Data Eng. 28 (7) (2016) 1940–1943.
[24] T. Wada, N. Matsumura, K. Nakano, Y. Ito, Efficient byte stream pattern test using bloom filter with rolling hash functions on the FPGA, in: 2018 Sixth International Symposium on Computing and Networking, 2018, pp. 66–75.
[25] I. Kaya, T. Kocak, A low power lookup technique for multi-hashing network applications, in: IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006.
[26] M.J. Lyons, D. Brooks, The design of a bloom filter hardware accelerator for ultra low power systems, in: Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design, 2009, pp. 371–376.
[27] J. Harwayne-Gidansky, D. Stefan, I. Dalal, FPGA-based soc for real-time network intrusion detection using counting bloom filters, in: IEEE Southeastcon 2009, Atlanta, GA, 2009, pp. 452–458.
[28] Netcope, www.netcope.com/getattachment/bb2b8efa-9925-438d-b895-897d7c1e4745/NFB-200G2QL-product-brief.aspx.
[29] Z. Martinasek, J. Hajny, D. Smekal, L. Malina, D. Matousek, M. Kekely, N. Mentens, 200 Gbps Hardware accelerated encryption system for FPGA network cards, in: Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security, 2018, pp. 11–17.
[30] Arish Sateesan, Jo Vliegen, Simon Scherrer, Hsu-Chun Hsiao, Adrian Perrig, Nele Mentens, Speed records in network flow measurement on FPGA, in: 2021 31st International Conference on Field-Programmable Logic and Applications, FPL, IEEE, pp. 219–224.
[31] Thomas Claesen, Arish Sateesan, Jo Vliegen, Nele Mentens, Novel non-cryptographic hash functions for networking and security applications on FPGA, in: 2021 24th Euromicro Conference on Digital System Design, DSD, IEEE, 2021, pp. 347–354.
[32] Xilinx User Guide, Ternary content addressable memory (TCAM) search IP for SDNet, https://www.xilinx.com/support/documentation/ip_documentation/tcam/pg190-tcam.pdf.

Arish Sateesan is a Ph.D. student at Embedded Systems and Security Group, COSIC, KU Leuven under the supervision of Prof. Nele Mentens. His research interests and publications lie in the fields of FPGA based system design, Network Security, and Convolutional Neural Networks on resource-constrained devices. He obtained his Bachelor's degree in Electronics & Communication Engineering from Cochin University of Science and Technology, India in 2009. After his bachelors, he worked in academia as well as in the industry before commencing his Master's degree in 2013. He obtained his Master's degree in Embedded System Design from the National Institute of Technology Kurukshetra, India. After the completion of his Master's, he joined IBM as Business Intelligence Application Developer in 2015. Later he moved into academic research at Nanyang Technological University Singapore as Research Associate (Deep Learning and Embedded Systems) in 2018 prior to joining KU Leuven in 2019.

Jo Vliegen got his master degree in Engineering Science in 2005 from the KHLim University College. After a little over three years, he returned to campus Diepenbeek and started working on FPGAs under the supervision of Nele Mentens. He obtained his Ph.D. from KU Leuven in 2014 with a dissertation entitled: "Partial and dynamic FPGA reconfiguration for security applications" under the promotorship of Ingrid Verbauwhede and Nele Mentens. Jo was a post-doc in the COSIC and ES&S research groups between 2014 and 2020. From 2020 onwards he works as a research expert in both COSIC and ES&S. His research still focuses on FPGAs and its usage in security applications.

Joan Daemen After graduating in electromechanical engineering Joan Daemen was awarded his Ph.D. in symmetric cryptography in 1995 from KU Leuven. After his contract ended at COSIC, he privately continued his crypto research and contacted Vincent Rijmen to continue their collaboration that would lead to the Rijndael block cipher, and this was selected by NIST as the new Advanced Encryption Standard in 2000. After over 20 years of security industry experience, including work as a security architect and cryptographer for STMicroelectronics, he is now a professor in the Digital Security Group at Radboud University Nijmegen. He co-designed the Keccak cryptographic hash function that was selected as the SHA-3 hash standard by NIST in 2012 and is one of the founders of the permutation-based cryptography movement and coinventor of the sponge, duplex and farfalle constructions. In 2017 he won the Levchin Prize for Real World Cryptography "for the development of AES and SHA3". In 2018 he was awarded an ERC advanced grant for research on the foundations of security in symmetric cryptography called ESCADA and an NWO TOP grant for the design of symmetric crypto in the presence of efficient multipliers called SCALAR.

Nele Mentens received her master and Ph.D. degree from KU Leuven in 2003 and 2007, respectively. Currently, Nele is a professor at Leiden University and KU Leuven. Her research interests are in the domains of configurable computing for security, hardware acceleration of network security applications and security in constrained environments. Nele was a visiting researcher at Ruhr University Bochum in 2013 and at EPFL in 2017. She was/is the PI in around 20 finished and ongoing research projects with national and international funding. She serves as a program committee member of renowned international conferences on security and hardware design, such as NDSS, Usenix Security Symposium, CHES, DAC, DATE, FPL and ESWEEK. She was the general co-chair of FPL in 2017, the program chair of EWME and PROOFS in 2018, the program chair of FPL and CARDIS in 2020, and the program chair of RAW in 2021. Nele is (co-)author in over 100 publications in international journals, conferences and books. She serves as an associate editor for IEEE Transactions on Information Forensics and Security and IEEE Circuits and Systems Magazine.