Novel Non-cryptographic Hash Functions for Networking and Security Applications on FPGA

DOI: 10.1109/DSD53832.2021.00059
Handle: http://hdl.handle.net/1942/36536

Novel Non-cryptographic Hash Functions for Networking and Security Applications on FPGA

Thomas Claesen
KU Leuven & UHasselt
Belgium
thomasclaesen@live.be

Arish Sateesan
imec-COSIC/ES&S
ESAT, KU Leuven, Belgium
arish.sateesan@kuleuven.be

Jo Vliegen
imec-COSIC/ES&S, ESAT
KU Leuven, Belgium
jo.vliegen@kuleuven.be

Nele Mentens
imec-COSIC/ES&S, ESAT
KU Leuven, Belgium
LIACS, Leiden University
The Netherlands
nele.mentens@kuleuven.be

***Abstract*—This paper proposes the design and FPGA implementation of five novel non-cryptographic hash functions, that are suitable to be used in networking and security applications that require fast lookup and/or counting architectures. Our approach is inspired by the design of the existing non-cryptographic hash function Xoodoo-NC, which is constructed through the concatenation of several Xoodoo permutations. We similarly construct non-cryptographic hash functions based on the concatenation of several rounds of symmetric-key ciphers. The goal is to achieve high performance in combination with good avalanche properties, which are required in order to have a significant change in the output value as a result of a limited change in the input value. We simulate how many rounds are needed to achieve satisfactory avalanche scores and we implement the corresponding non-cryptographic hash functions on an FPGA to evaluate the occupied resources and the performance. One of the proposed non-cryptographic hash functions, namely GIFT-NC, outperforms all previously proposed non-cryptographic hash functions in terms of throughput and latency, in exchange for an acceptable increase in FPGA resources.**

***Index Terms*—non-cryptographic hash functions, NIST lightweight standardization competition, FPGA, avalanche metrics**

I. Introduction

Hashing is the practice of taking a message of arbitrary length and converting it into a digest of fixed length. A hash function is a one-way function, which means recovering the original message from the digest is not feasible in practice. Hash functions make an important building block in security applications and are part of data structures like dictionaries and associative arrays, that are employed in many networking applications. In network applications, a network flow is represented by a flow identifier which is extracted from the packet header and consists of source and destination addresses and ports. These flow identifiers are required to be stored in or queried from a data structure. Hash-based lookup architectures provide fast lookup and low memory overhead. Probabilistic data structures in network security applications, such as Bloom filters [1] for fast lookups and sketches [2] for fast counting, also use hash functions as important components. Even though cryptographic hash functions offer high levels of security, not all of their security properties, such as preimage resistance, second preimage resistance and collision resistance [3], are necessary in probabilistic data structures employed in network security applications. The throughput of these data structures also depends on the performance of the hash computations, which makes the speed of the hash functions a crucial factor. If cryptographic hash functions would be used in these data structures, processing at line rate in high-speed networks would be impossible. Therefore, this paper concentrates on the design of novel non-cryptographic hash functions.

Non-cryptographic hashes for applications such as Bloom filters, hash tables, and sketches must be fast, uniformly distributed and must have excellent avalanche properties [4], [5]. The latter indicates how well a hash function succeeds in minimizing the correlation between the input text and the generated digest. Security properties such as pre-image resistance are of minor importance. Universal classes of hash functions [6] are most commonly employed in most of the network applications, but at the cost of increased latency. One way to reduce the latency in network applications is to employ a single hash function having a large enough output size and then split the output to generate multiple hash values. Sateesan et al. [7] have shown that this technique can significantly improve the speed to match Terabit networks without any adverse effects on false positive rates. Yet, choosing a suitable non-cryptographic hash function which can serve the purpose is a cumbersome task. The FNV and Murmur series hash functions [8], [9] show excellent avalanche properties, but exhibit high latency with larger inputs. Also, the hardware suitability of FNV and Murmur hashes, like many other non-cryptographic hashes, is limited, because they target efficient execution in software [7]. The latency of these hashes is dependant on the size of the input key. Some of the recent work has proven that non-cryptographic hashes constitute a significant proportion of the overall resource requirements of data structures [10].

In this work, we target high-speed implementations of non-cryptographic hash functions on FPGA, in order to serve lookup and counting applications that rely on the reconfigurability, the parallelism, the ability to work as a standalone unit, and the low time to market offered by FPGAs. The trend of moving security and networking applications to FPGAs in order to achieve a higher throughput is motivated in [11]. We propose to use a concatenation of the rounds in symmetric-key ciphers as hashes. We reduce the number of rounds in order to optimize the performance in hardware, while preserving the avalanche properties. Inspired by Xoodoo-NC [7], we consider

96-bit hash functions because network (security) applications typically process source and destination addresses and ports that constitute a total of 96 bits of the header of incoming network packets. We evaluate the resource utilization and performance of the resulting non-cryptographic hash functions to finally conclude that we outperform all previously proposed work in terms of speed.

II. Background and related work

Non-cryptographic hashes are fast and resource-efficient compared to cryptographic hashes. However, popular non-cryptographic hash functions such as FNV-1a [8] and Murmur3 [9] are optimized for implementation in software and do not perform well in hardware. FNV-1a and Murmur3 employ multiplication as the core operation, which is a complex operation in hardware. Moreover, FNV-1a processes 8 bits per cycle and Murmur processes 32 bits per cycle, which also causes the execution delay to be dependent on the message size. Some of the non-cryptographic hash implementations on FPGA presented by Grochol and Sekanina [12] prove to be faster than FNV-1a and Murmur3 in terms of operating frequency, but the overall execution delay is not fast enough for high-speed applications such as Terabit Ethernet network applications.

Cryptographic hash functions have been proposed based on lightweight symmetric-key algorithms, which provide both area-efficiency and security, but are typically slower than non-cryptographic hash functions because they need a relatively large number of rounds to satisfy all security requirements. A number of papers target FPGAs as the platform for running the lightweight cryptographic algorithms [13]–[15], [15], [16]. Even though most of the implementations are resource and energy efficient, high latency is still a drawback. The implementation of the Photon lightweight hash function presented in [13], [15], [15], [16] shows a minimum cycle count of 12 and a maximum cycle count of 1680. Similarly for Spongent hash [14]–[16], the minimum cycle count is 45 and the maximum cycle count is 1980. Most lightweight hash functions have in common that the latency is large. This is not acceptable in high-speed network security applications where processing at line-rate is a matter of utmost concern. Reduced-round versions of these functions result in deteriorated security properties while maintaining the avalanche properties. For non-cryptographic hash functions, this is sufficient. Xoodoo-NC [7] is the result of such an effort. It uses the cryptographic permutation Xoodoo [17] and is proven to be very efficient. In this work, we evaluate a number of non-cryptographic hash functions that are based on reduced-round symmetric-key algorithms with respect to avalanche properties, performance and FPGA resource utilization.

III. Lightweight ciphers

We arbitrarily choose five symmetric-key ciphers for our analysis: Pyjamask [18], GIFT [19], SKINNY [20], AES [21] and SPECK [22]. These ciphers are briefly described in this section. It is pointed out that any special modifications to the first or last round in these algorithms are not discussed here nor are they implemented. Only the regular round function of the ciphers is considered. For a more in-depth and complete description, we forward the interested reader to the specifications. Some of the ciphers have a 96-bit block size, which matches our goal to design 96-bit non-cryptographic hash functions. Some ciphers have a 128-bit block size. For those, the avalanche properties shown in Sect. IV are calculated for the entire 128-bit output, but the hardware implementation results are generated for only 96 bits, while fixing 32 bits to 0.

A. SPECK

The SPECK block cipher family holds 10 different variations of the SPECK algorithm, having block sizes of 32, 48, 64, 96, and 128 with varying key sizes ranging from 64 to 256. This paper focuses only on the cipher with a block size and key size of 96 bits. SPECK is an add-rotate-xor cipher, which means it only uses these three operations to calculate the ciphertext. SPECK with a block size and key size of 96-bits consists of 28 rounds. During the rounds, the key and the plaintext are split into two parts of 48 bits. The most significant bits of the plaintext (PT) and the key (K) are stored in PT_1 and K_1 respectively, while the least significant bits are stored in PT_2 and K_2.

Round function: In SPECK, each round performs three operations: Rotation, Addition, and XOR. First, PT_1 is rotated 8 bits to the right, this is followed by an addition with PT_2. Then, PT_1 is XORed with K_2. For PT_2, the operations start after the addition with PT_1, with rotating 3 bits to the left followed by an XOR with PT_1. Fig. 1 shows how this is done for a single round. After each round, the output is connected to the input of the next round.

Key schedule: The key changes every round for SPECK, this round function works similarly to the encryption round for the plaintext. The only difference is the XOR, where the second operand is the round number, starting from 0 (the first round) and incrementing by 1 in every round.

B. Pyjamask

The Pyjamask block cipher family [18] contains two algorithms which support both 96-bit and 128-bit block sizes. Both algorithms use a 128-bit key, perform 14 rounds and rely on a Substitution-Permutation Network (SPN) structure to transform the plaintext into the ciphertext. The plaintext is structured in a bit-by-bit left-right, top-down structure, where each row consists of 32 bits. Each row R_i starts from $32 \times i$ and increments by 1, where $i \in 0, 1, 2, 3$. Depending on the block size of the chosen algorithm, 3 or 4 rows are used. The cell which holds the lowest index represents the most significant bit of the plaintext and the cell with the highest index represents the least significant bit. The key is represented in the same way as the plaintext. Only this time, there are 4 rows used of which each contain 32 bits. Once the plaintext enters the algorithm, it will be referred to as the internal state.

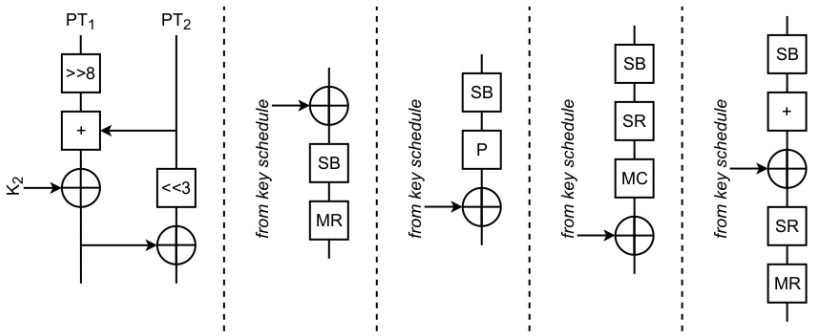

Fig. 1. Functional representation of a single round of the 5 chosen block ciphers: SPECK, Pyjamask, GIFT, AES, and SKINNY.

Round function: Each round is composed of three operations, which happen in sequential order: AddRoundKey, SubBytes and MixRows. AddRoundKey first XORs n bits of the key with the internal state. n is 128 for both Pyjamask-128 and Pyjamask-96, however, in case of the latter only the 96 most significant bits are addressed. The result of the XOR is passed through an SBOX. Finally, the internal state is multiplied with a constant matrix.

Key schedule: Throughout the algorithm, different keys are used in each AddRoundKey. These keys are called subkeys, and all originate from the original secret key. To receive these subkeys, three operations are executed in each round on the current round key: MixColumns, MixRows and Constant Addition. Because Pyjamask-96 and Pyjamask-128 both use 128-bit keys, the same operations are used to create the subkeys.

C. GIFT

The GIFT block cipher [19] family contains two algorithms: GIFT-64 and GIFT-128. Both use a key size of 128 bits, while the former uses a block size of 64 bits and the latter of 128 bits. In this paper, GIFT-128 is used. Like Pyjamask, GIFT is an SPN cipher containing 40 rounds.

Round function: Each round of GIFT is based on three operations, namely SubCells, PermBits, and AddRoundKey. Similar to SubBytes in Pyjamask, each 4-bit column is compared to an Sbox and replaced by the appropriate value. The PermBits operation shuffles all the bits in the internal state according to a specific permutation. Finally, the AddRoundKey operation adds the roundkey to the internal state through an XOR.

Key Schedule: GIFT follows a simple key schedule operation to minimize the area in hardware. The key is split in to blocks K_i of 32 bits each as in the AddRoundKey operation, where $i \in \{0, 1, 2, 3\}$. K_0, K_1 and K_2 are then shifted towards the least significant bits of the key state, while K_3 now holds the most significant bit. K_3 is then split again into two 16-bit blocks, $K_{3,1}$ and $K_{3,2}$. These two blocks are then right-rotated over 2 and 12 positions, respectively.

D. AES

AES is probably the most popular and well-known cipher. It has a block size of 128 bits and the key size can be either 128, 192 or 256 bits. Depending on the key size, the number of rounds is 10, 12 and 14, respectively. This paper focuses on the algorithm having a key size of 128 bits. Similar to GIFT, the input plaintext is structured in a top-down,left-right manner. The key is also structured in a similar fashion.

Round function: Each round consists of 4 operations: SubBytes, ShiftRows, MixColumns and AddRoundKey. AddRoundKey performs a simple bitwise XOR of the key with the internal State. Each round uses a different round key. These round keys are calculated in the key schedule. In the next step, each byte in the internal state is replaced through an SBOX lookup. After the SBOX, the rows of the state are shifted cyclically in a specific manner. Finally, in the MixColumns step, a matrix multiplication is done. The state is updated with the resulting product.

Key schedule: The key schedule generates a new round key, starting from the initial key. Every key is split in four equally sized parts. One of these parts is rotated and subsequently passes another SBOX. The result of the SBOX is then XORed with a round constant and another part of the incoming key. The final two parts are finally XORed in turn with the previous result.

E. SKINNY

The SKINNY block cipher family holds six algorithms, namely SKINNY-64-64, SKINNY-64-128, SKINNY-64-196, SKINNY-128-128, SKINNY-128-256, and SKINNY-128-384. The first numerical value in the name is the block size used in the algorithm, and the second value is the key size. In this paper, we focus only on SKINNY-128-128. The plaintext and key are initialized similar to AES, but in a left-right top-down manner. The number of rounds of the SKINNY cipher varies with the algorithm used. For SKINNY-128-128, the number of rounds required are 40. Each round consists of 5 different operations: SubBytes, AddConstants, AddRoundKey, ShiftRows and MixColumns.

Round function: The SubBytes operation follows the exact operation of SubBytes in AES, the only difference being the values in the SBOX.In the AddConstants operation, the round constants are XORed to the first column of the internal State. The round constant is a 6-bit number and is generated using an LFSR, which changes every round. The key state is added

to the internal state in the AddRoundKey operation. Unlike the AddRoundKey function in AES, only the 8 most significant bytes of the key are XORed to the 8 most significant bytes of the internal state. The ShiftRows function works the same way as the ShiftRows function in AES, the only difference is that the rows are shifted in the opposite direction. The MixColumns function is the final round operation. Similar to MixColumns in Pyjamask and AES, each column of the internal state is multiplied by a constant Matrix M.

Key schedule: Just like other ciphers, the key is updated in every round into different subkeys. In SKINNY, the key updates are similar to the key schedule in GIFT.

IV. Analysis of non-cryptographic hash functions

The ciphers mentioned in the previous section have proven to be secure, but not fast enough for applications such as Bloom filters and sketching architectures. Our goal is to use these ciphers as hash functions by reducing the number of rounds and to analyze the avalanche properties [23] to make sure that they are still satisfactory. The use of keys does not cause any difference in avalanche properties, but allows us to easily change the hash function by changing the key value. The avalanche performance can be assessed by analyzing to which extent the output value changes for a slight change in the input of the hash function. If the hash function does not show a good avalanche performance, the randomization is poor for the hash function, which leaves the function vulnerable to attackers. The avalanche properties are determined here using three avalanche metrics put forward by Daemen et al. [24]: Avalanche dependence, Avalanche weight and Avalanche entropy, which are defined below.

Avalanche dependence: For a single-bit change in the input, the number of bits in the output that may flip defines the avalanche dependence. The avalanche dependence is defined as:

$$D_{av} = n - \sum_i g(p[i]) \tag{1}$$

Here n is the number of bits in the output, and p is the probability vector and $p[i]$ represents the probability that the bit i of the output flips for a single bit change in the input. $g(p) = 1$ if $p = 0$ and $g(p) = 0$ otherwise.

The Avalanche dependence is satisfied when $D_{av} = n$ for all outputs as a result of a single bit change at the input.

Avalanche weight: The avalanche weight generalizes the avalanche criterion and is defined as the expected Hamming weight of the output difference for a single bit change in the input. It is defined as:

$$w_{av} = \sum_i p[i] \tag{2}$$

The condition is satisfied when the avalanche weight is equal to 50% of the number of output bits i.e., $w_{av} \approx \frac{n}{2}$, for all inputs with a single bit change (Hamming weight = 1).

Avalanche entropy: The avalanche entropy visualizes the uncertainty of whether the bits in output flip or not for a single bit change in the input. It is defined as:

$$H_{av} = \sum_i (-p[i].log_2(p[i]) - (1 - p[i]).log_2(1 - p[i])) \tag{3}$$

The H_{av} generalizes the strict avalanche criterion. The avalanche entropy is said to be satisfactory if it fulfills the condition $H_{av} \approx n$ for all inputs with a single bit change.

In our analysis, to make sure that the avalanche metrics are satisfactory in any given condition, the worst case values are considered. To have high precision for the avalanche metrics, we set the number of iterations M as large as 250'000, and $\frac{1}{\sqrt{M}}$ is the expected standard deviation of each element in the probability vector p [24]. In each iteration, all n bits are flipped one by one and the avalanche metrics are calculated. This ensures that the avalanche metrics obtained are stable and precise.

The plot of the avalanche metrics of each cipher as a function of the number of rounds is shown in Fig. 2. We add NC (Non-Cryptographic) to the name of the cipher to indicate that we are considering the reduced-round version in order to construct the non-cryptographic hash function. While SPECK has 28 rounds, SPECK-NC only requires 7 rounds to achieve satisfactory avalanche properties. Pyjamask normally requires 14 rounds, but the avalanche metrics plot shows that only 2 or 3 rounds are required for Pyjamask-NC to meet the avalanche properties. GIFT-NC requires only 6 to 7 rounds to achieve the avalanche property requirements as shown in the figure, in contrast to the actual 40 rounds in GIFT. The number of rounds to have sufficient avalanche properties for AES-128 can also be reduced from 10 rounds to 3 rounds in AES-NC. Similar to GIFT, SKINNY-128-128 also uses 40 rounds, but requires only 6 rounds to meet the avalanche criteria in SKINNY-NC. This analysis shows that we can use reduced-round versions of cryptographic ciphers as non-cryptographic hash functions. The number of rounds can be reduced 70-85% while maintaining excellent avalanche properties. With a reduced number of rounds, the latency can be improved keeping the hardware resource requirement to a minimum. A summary of the avalanche metrics analysis as a function of the number of rounds is shown in Table I.

TABLE I
The number of rounds needed for meeting the avalanche criteria

Cipher	D_{av}	w_{av}	H_{av}	r	r_{av}
SPECK-NC	96	47.45	95.68	28	7
Pyjamask-NC	96	47.73	95.96	14	3
GIFT-NC	128	63.39	127.95	40	7
AES-NC	128	63.0	127.43	10	3
SKINNY-NC	128	63.72	127.95	40	7

r: Number of rounds in the original cipher
r_{av}: Number of rounds to meet avalanche criteria

V. Hardware Evaluation

A. Experimental setup

The non-cryptographic hash functions are implemented on an FPGA to evaluate the resource requirements and operating

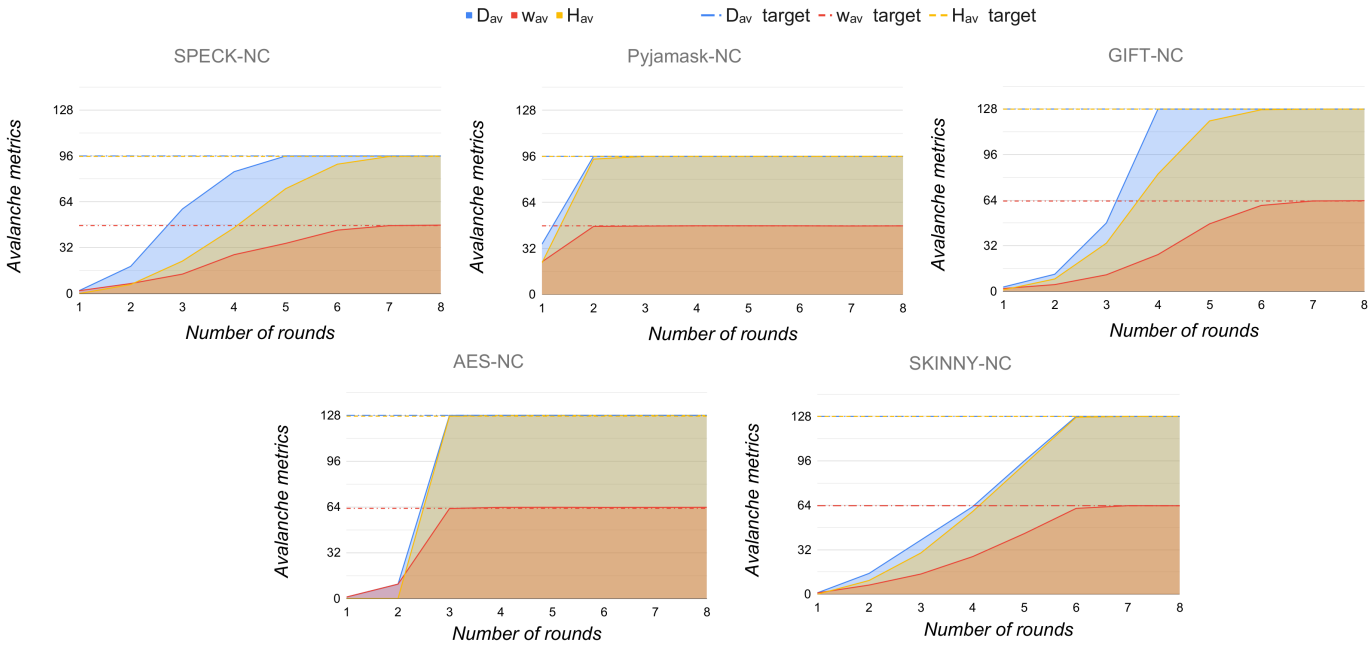

Fig. 2. The avalanche metrics for SPECK-NC, Pyjamask-NC, GIFT-NC, AES-NC, and SKINNY-NC, respectively. The areas show the obtained metric while the horizontal lines indicate the target. The legend indicates which colors are the Avalanche Dependence (D_{av}), the Avalanche Weight (w_{av}), and the Avalanche Entropy (H_{av}).

speed. The hash functions are evaluated on both Zynq and Virtex Ultrascale+ FPGAs. All the values presented in the paper are the implementation results on Virtex Ultrascale+ XCVU7P-FLVB2104-2-i device. The general block diagram of the hardware implementation of the reduced-round ciphers as hash functions is shown in Fig. 3. This figure shows that the implementations are unrolled with only a register before and after the rounds. Therefore, the cycle count is fixed to a single clock cycle.

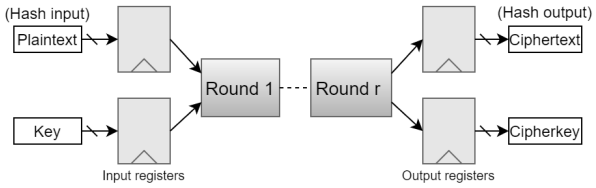

Fig. 3. General hardware setup of hash implementation

B. Evaluation

Latency and Throughput: The number of rounds r for each cipher has been chosen corresponding to the number of rounds required to meet the sufficient avalanche properties (see Table I). The plaintext is the input to the hash function and the resulting ciphertext is the output of the hash. Adding a key is not really necessary and the addition of key does not affect the avalanche properties. In textbook hash functions, the digest should always be the same, given the same input. In some applications, however, it is useful to have multiple, independent hash values for the same input, like in Bloom filters. Adding a key to the block cipher allows to generate multiple independent hash values using the same hash function but with different keys.

To analyze the timing, the worst negative slack is plotted against varying clock periods and is shown in figures Figs. 4 to 8. For each cipher, the timing results with and without adding the key are plotted. In general, it can be seen from the figures that the implementation with key is slightly slower compared to the implementation without key. This is because of the fact that all the calculations involving the key are completely eliminated in the implementation without key. It is also evident from the plots that for larger FPGAs, the latency is lower. The maximum possible frequency for each implementation on hardware is given in Table II. Comparing to each other, GIFT-NC exhibits the best operating frequency and proves to be faster than the other ciphers. With and without using the key, the variations in latency of GIFT-NC, Pyjamask-NC, and SKINNY-NC are negligible. However, adding the key in SPECK-NC and AES-NC adds around 17% latency, as computational steps and hence the delay involving the key calculations are added.

The throughput of the hash function is a function of the latency and input block size. The throughput here is calculated using the equation $\frac{Block\ size}{latency\ in\ cycles} \times f_{max}$, where f_{max} is the maximum operating frequency. The throughput is of the order of ten thousands as the latency is only a single clock cycle thanks to the unrolling.

Resource requirements: The resource utilization of the implementations for the best achievable latency is shown in Table III. GIFT-NC proves to be the most resource-efficient of all the compared ciphers when the key is included. SPECK-NC

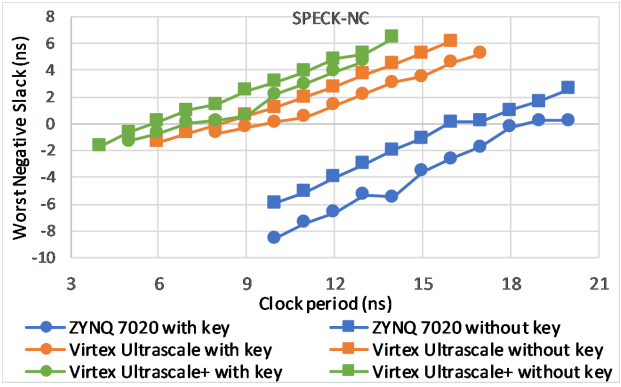

Fig. 4. Timing analysis of SPECK-NC

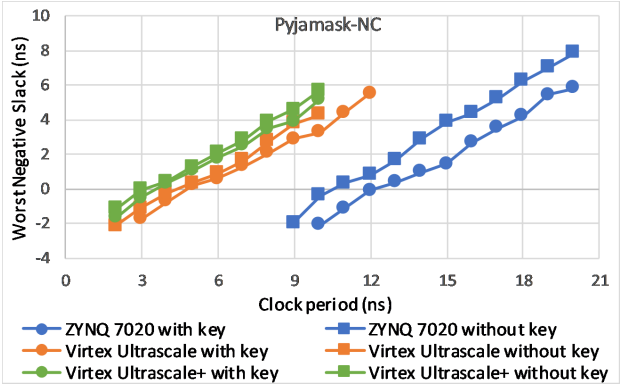

Fig. 5. Timing analysis of Pyjamask-NC

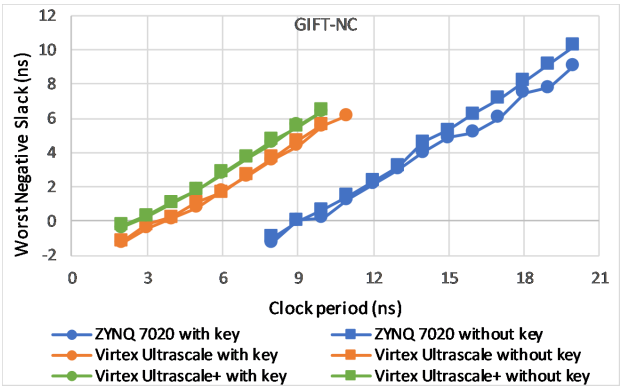

Fig. 6. Timing analysis of GIFT-NC

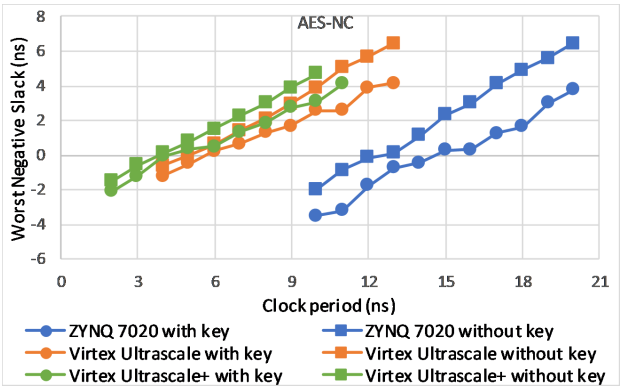

Fig. 7. Timing analysis of AES-NC

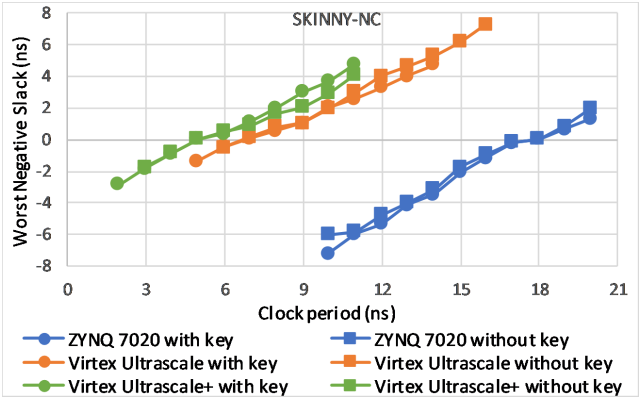

Fig. 8. Timing analysis of SKINNY-NC

TABLE II
Maximum operating frequency of hash implementations

Hash Function	Max. Frequency (without key)	Max. Frequency (with key)
SPECK-NC	171.6 MHz	144.8 MHz
Pyjamask-NC	279.4 MHz	271.6 MHz
GIFT-NC	369.7 MHz	365.8 MHz
AES-NC	255.9 MHz	214.8 MHz
SKINNY-NC	202.4 MHz	202.5 MHz

TABLE III
Resource utilization

Hash function	LUTs (without key)	Flip Flops (without key)	LUTs (with key)	Flip Flops (with key)
SPECK-NC	432	192	1273	384
Pyjamask-NC	811	448	1615	448
GIFT-NC	546	512	665	512
AES-NC	2225	512	3402	512
SKINNY-NC	2176	512	2348	512

uses the lowest resources when the key is not added. AES-NC and SKINNY-NC are the most expensive in terms of resources, while showing moderate operating frequency. SPECK-NC, even though showing efficiency in resource utilization without the key, requires ≈3× more resources when the key is added. GIFT shows the least variation in resources whether the key is added or not.

Comparison with related work: In this work, we focus on reduced-round, reduced-logic implementations of non-cryptographic hash functions based on existing ciphers. Hence, a straightforward comparison with the existing full-fledged implementations of the same ciphers on FPGA is not fair. For example, the throughput of the FPGA implementation of AES-128 and SPECK-96 given by Diehl et al. [25] are 119 Mbps and 1622 Mbps, respectively, which is 206× and 10× lower than our reduced-round implementations. Similarly, the high-

throughput implementation of SKINNY-128 on FPGA [20] has a throughput/LUT of 3.06 Mbps, which is 3 times lower than our reduced-round implementation. To have a fair comparison, we have also implemented FNV-1a and Murmur3 hash on FPGA and compared with the reduced-round ciphers. The comparison of some of the fast non-cryptographic hashes with our implementations is shown in Table IV.

TABLE IV
Comparison of maximum frequencies, throughput (Tp), throughput per LUT (Tp/LUT), and delay with related work

Design	Block Size (In/Out)	Maximum Frequency	Tp (Mbps)	Tp / LUT (Mbps/LUT)	Latency (ns)
Murmur3	96/64	120.6 MHz	2573	4.54	24.87
FNV-1a	96/128	122.9 MHz	925	1.63	130.08
SipHash [12]	96/16	182.8 MHz	1463	1.38	21.88
XORHash [12]	96/16	**627.3 MHz**	2868	9.86	11.13
NSGAHash7 [12]	96/16	184.1 MHz	1473	18.41	21.72
Xoodoo-NC [7]	96/96	363.6 MHz	34'906	**112.96**	2.75
SPECK-NC (ours)	96/96	171.6 MHz	16'473	38.13	5.82
Pyjamask-NC (ours)	96/96	279.4 MHz	26,822	32.70	3.58
GIFT-NC (ours)	96/128	369.7 MHz	**35'491**	65.00	**2.70**
AES-NC (ours)	96/128	255.9 MHz	24'566	11.04	3.91
SKINNY-NC (ours)	96/128	202.4 MHz	19'440	8.93	4.93

As shown in the table, the throughput of the previously proposed non-cryptographic hash functions is significantly lower compared to our implementations. Only the throughput of Xoodoo-NC is somewhat comparable. XORHash [12] shows the highest operating frequency of all related work, but the throughput compared to SPECK-NC is still 5.74× lower, where SPECK-NC has the lowest throughput among our implementations. GIFT-NC shows the best throughput, which is 12.4× higher than that of XORHash and slightly higher than Xoodoo-NC. Besides the throughput per LUT, which was already given in Table IV, we calculate another metric that takes into account both the latency and the FPGA resources. It is calculated as $1/(latency * LUTs)$ and shown in Fig. 9 together with the $throughput/LUT$. The higher these two parameters, the better the performance of the hash function.

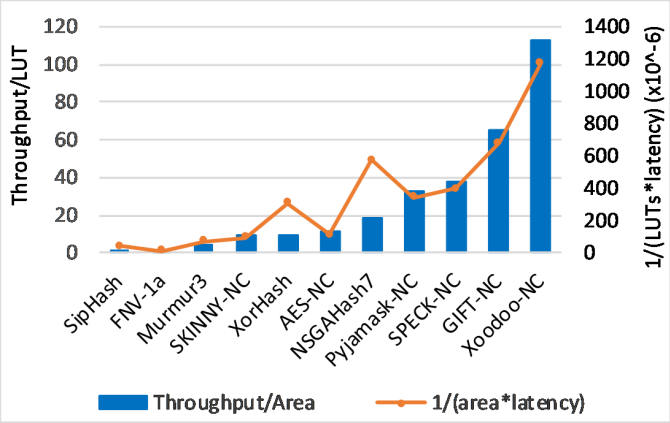

Fig. 9. Throughput and latency with respect to resource utilization

Xoodoo-NC [7] exhibits the best throughput per LUT, which is 1.7× higher than GIFT-NC, so for applications where area plays an important role, Xoodoo-NC should be preferred over GIFT-NC. Another observation when evaluating throughput per area is that Murmur3, FNV-1a, and NSGAHash employ 29, 38, and 3 DSP blocks, respectively, in their implementations, whereas our reduced-round ciphers do not use any DSP slices. GIFT-NC also has the lowest execution delay among all the implementations. The area-latency performance of hashes shows similar characteristics as throughput-area for all hashes except for the NSGAHash7 and XorHash. Both NSGAHash7 and XorHash shows increased area-latency performance, thanks to the relatively lower area to latency ratio, while all other hashes' area-latency performance closely follows their throughput-area characteristics.

VI. Conclusion

In this paper, we investigate the use of reduced-round versions of existing lightweight ciphers to serve as non-cryptographic hash functions. In order to do so, we simulate the avalanche properties of each cipher as a function of the number of rounds. Based on the minimum number of rounds required to satisfy the avalanche requirements, we evaluate the resource utilization and latency on an FPGA, assuming completely unrolled and non-pipelined architectures. We show that the newly proposed GIFT-NC outperforms all previously proposed hash functions in terms of throughput and latency. Also the other non-cryptographic hash functions that we evaluate, are valuable alternatives for GIFT-NC, which allows to have a pool of different non-cryptographic hash functions with desirable throughput and latency. This is important for applications that need more than one different non-cryptographic hash function in high-speed lookup or counting architectures.

Acknowledgement

This work is supported by the ESCALATE project, funded by FWO and SNSF (G0E0719N), and by Cybersecurity Initiative Flanders (VR20192203).

References

[1] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006.

[2] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[3] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[4] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[6] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[7] Arish Sateesan, Jo Vliegen, Joan Daemen, and Nele Mentens. Novel Bloom filter algorithms and architectures for ultra-high-speed network security applications. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 262–269. IEEE, 2020.

[8] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald Eastlake, and Tony Hansen. The FNV non-cryptographic hash algorithm. *Ietf-draft*, 2011.

[9] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*, 44(6):681–698, 2014.

[10] Amit Kulkarni, Monica Chiosa, Thomas B Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. Hyperloglog sketch acceleration on fpga. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 47–56. IEEE, 2020.

[11] Stephen M Trimberger and Jason J Moore. FPGA security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

[12] David Grochol and Lukas Sekanina. Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 257–263. IEEE, 2018.

[13] Mohammed Omar Awadh Al-Shatari, Fawnizu Azmadi Hussin, Azrina Abd Aziz, Gunawan Witjaksono, and Xuan-Tu Tran. FPGA-Based Lightweight Hardware Architecture of the PHOTON Hash Function for IoT Edge Devices. *IEEE Access*, 8:207610–207618, 2020.

[14] Carlos Andres Lara-Nino, Miguel Morales-Sandoval, and Arturo Diaz-Perez. Small lightweight hash functions in FPGA. In *2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 1–4. IEEE, 2018.

[15] N Nalla Anandakumar, Thomas Peyrin, and Axel Poschmann. A very compact FPGA implementation of LED and PHOTON. In *International Conference on Cryptology in India*, pages 304–321. Springer, 2014.

[16] Bernhard Jungk, Leandro Rodrigues Lima, and Matthias Hiller. A systematic study of lightweight hash functions on FPGAs. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6. IEEE, 2014.

[17] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. *IACR Cryptol. ePrint Arch.*, 2018:767, 2018.

[18] Dahmun Goudarzi, Jérémy Jean, Stefan Kölbl, Thomas Peyrin, Matthieu Rivain, Yu Sasaki, and Siang Meng Sim. Pyjamask: Block Cipher and Authenticated Encryption with Highly Efficient Masked Implementation. *IACR Transactions on Symmetric Cryptology*, pages 31–59, 2020.

[19] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.

[20] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. Skinny-aead and skinny-hash. *IACR Transactions on Symmetric Cryptology*, pages 88–131, 2020.

[21] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced Encryption Standard (AES), 2001-11-26 2001.

[22] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[23] Réjane Forrié. The strict avalanche criterion: spectral properties of boolean functions and an extended definition. In *Conference on the Theory and Application of Cryptography*, pages 450–468. Springer, 1988.

[24] Joan Daemen, Seth Hoffert, G Van Assche, and R Van Keer. The design of xoodoo and xoofff. 2018.

[25] William Diehl, Farnoud Farahmand, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. Comparison of hardware and software implementations of selected lightweight block ciphers. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.