# Evolving Non-cryptographic Hash Functions using Genetic Programming for High-speed Lookups in Network Security Applications

Mujtaba Hassan[1][0000−0001−9590−2469], Arish Sateesan[1][0000−0002−8197−0097], Jo Vliegen[1][0000−0003−4258−2208], Stjepan Picek[3][0000−0001−7509−4337], and Nele Mentens[1,2][0000−0001−8753−7895]

[1] imec-COSIC/ES&S, ESAT, KU Leuven, Belgium
[2] LIACS, Leiden University, The Netherlands
{mujtaba.hassan,arish.sateesan,jo.vliegen,nele.mentens}@kuleuven.be
[3] Digital Security Group, Radboud University, The Netherlands
stjepan.picek@ru.nl

**Abstract.** Non-cryptographic (NC) hash functions are the core part of many networking and security applications such as traffic flow monitoring and deep packet inspection. For these applications, speed is more important than strong cryptographic properties. In Terabit Ethernet networks, the speed of the hash functions can have a significant impact on the overall performance of the system when it is required to process the packets at a line rate. Hence, improving the speed of hash functions can have a significant impact on the overall performance of such architectures. Designing a good hash function is a challenging task because of the highly non-linear and complex relationship between input and output variables. Techniques based on Evolutionary Computation (EC) excel in addressing such challenges. In this paper, we propose novel fast non-cryptographic hash functions using genetic programming, and we call the resulting hash functions the GPNCH (Genetic Programming-based Non-Cryptographic Hash) family. We choose to employ avalanche metrics as a fitness function because the networking and security applications we consider require hash functions to be uniform and independent. We evaluate the performance of GPNCH functions on FPGA and compare the delay, throughput, and resource occupation with the state-of-the-art NC hash functions that satisfy the avalanche criteria. We show that GPNCH functions outperform the other algorithms in terms of latency, operating frequency, and throughput at the modest cost of hardware resources.

**Keywords:** Evolutionary Computation · Genetic Programming · Non-cryptographic Hash Functions · FPGA · Avalanche Metrics

## 1 Introduction

All the hash functions (cryptographic and non-cryptographic) map messages of arbitrary length into fixed-length codes called hashes or digests. The output of

any good hash function for a large set of inputs must be uniformly distributed and exhibit good randomness. Moreover, the hash code should be changed significantly by altering any single bit in the input, which is known as the avalanche effect. Cryptographic hash functions offer better security guarantees by holding properties such as pre-image, second pre-image, and collision resistance. On the other hand, for NC hash functions, such properties are relatively relaxed in exchange for fast execution time. Name any application where there is a need to do a fast lookup, NC hash functions can be used, e.g., hash tables, checksums and correction codes, finding duplicate records, caches, data compression, Bloom filters, transposition tables, etc. Probabilistic data structures such as Bloom filters [22], Hyperloglog [12], and Count-Min Sketches [7] depend upon NC hash functions to do probabilistic membership queries or counting. In network security applications where a few false positives are acceptable, and the address of stored data is not required, Bloom filters are the preferred choice because they provide constant time queries along with better space utilization [23].

Network traffic monitoring and filtering allow network managers to keep the network safe from malicious traffic and ensure that network bandwidth is utilized effectively. These tools often check for the source and destination IP addresses, sender and receiver ports, and protocols by extracting information from the packet header. A combination of these fields makes up a network flow ID. These flow IDs can be used as an index in a table to look for potential matches to redirect, count or block the packets. On average, hash-based search algorithms offer constant time complexity of $O(1)$, independent of input data size. In the worst case, where every hash value matches with the input key, they can behave like linear search $O(n)$, but this will only happen if they are poorly designed. This time complexity, together with low memory requirements, makes them an ideal choice for high-speed lookups in databases.

For high-speed networks like Terabit Ethernet networks (i.e., networks having a bandwidth of more than 100 Gbps), high-speed flow ID lookups need to be accomplished at a line rate. Therefore, implementing probabilistic architectures such as Bloom filters and Sketches on FPGA can lead to low latency, high throughput, and high operating frequencies [1]. Many large data centers that form the backbone of cloud computing are also using FPGAs to achieve high efficiency at a low-cost [4]. To meet the present and future bandwidth requirements, one way is to increase the computing capabilities of such architectures by improving the design of NC hash functions. This has led Sateesan et al. [29] to develop a hardware-oriented NC hash function called Xoodoo-NC for Bloom filter to process 96-bit network flow IDs. The authors significantly improved the performance of the Bloom filter compared to the FNV-1a [14] based implementation, which is known to have the smallest computational delay [29]. They partitioned a single hash output of 96 bits into several sets instead of using many hash functions. Claesen et al. [6], inspired by this work [29], proposed five more NC hash functions for FPGA-based high-speed lookups. One of their proposed algorithms, GIFT-NC, managed to slightly improve the latency and throughput of Xoodoo-NC at the cost of more hardware resources.

The main motivation behind our work is to build upon the work of [6] and [29] by improving the latency and throughput of hash functions on FPGAs since networks having a speed of 46 Terabits have become a reality, and in the future bandwidths, will continue to increase [5]. Therefore, network monitoring and security applications that process data at a line rate should be ready for these bandwidths. Designing hash functions is extremely challenging as the interconnection between input and output variables is highly nonlinear, convoluted, and abstruse, which makes them an ideal candidate for evolutionary computation (EC) techniques [11]. EC approaches [10] such as Cartesian Genetic Programming (CGP) and Genetic Programming (GP), which are inspired by Darwin's theory of natural selection [3], make an excellent choice to design and optimize digital circuits because both can easily be modeled in the hardware. They use the same logical operations that are the main building blocks in the modeling of digital circuits [26]. Surprisingly, there has been very little research done to use EC in the design of NC hash functions targeting efficient FPGA implementations. This drives our motivation to use GP to evolve digital circuits to design high-speed NC hash functions. GP is an example of the EC algorithms [10] in which the solution to a given problem is represented as a computer program with the help of syntax trees, as is shown in Figure 1. All the solutions start from the embryonic stage and undergo the evolution process of selection, crossover, and mutation in search of finding the optimized individual representing the final solution.

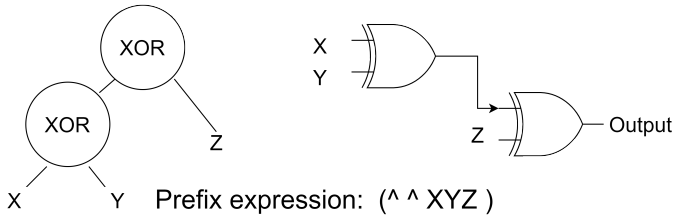



Fig. 1: Individual represented with GP (left), equivalent 3-bit parity checker (right), and prefix expression (bottom)

In this research, we show that our proposed family of NC hash functions outperforms the compared NC hash functions in terms of latency, operating frequency, and throughput on FPGA. Our contributions to the research domain are as follows:

- We are the first to design NC hash functions using EC for a 96-bit input/output block while fully satisfying all three avalanche metrics (entropy, bit dependence, and avalanche weight).
- Our proposed method outperforms the current state-of-the-art NC hash functions (which satisfy the avalanche criteria) on FPGA in terms of delay, operating frequency, and throughput.

– The proposed solution generates a family of independent NC hash functions. This could cater to different applications that require multiple, independent hash algorithms, like the Bloom filter.

The rest of the paper is organized as follows: Section 2 presents related work on the design of NC hash functions using various techniques. Section 3 discusses the design criteria of NC hash functions. Section 4 introduces the design and implementation of the GPNCH family. Results are discussed and compared with the state of the art in Section 5. Conclusions are drawn in Section 6.

## 2 Related Work

NC hash functions provide fast speed at the cost of less stringent security properties. As discussed in Section 1, this enables their use in many network security applications that require fast searching. On the other hand, hardware platforms such as FPGAs are increasingly being used to implement probabilistic architectures like Bloom filters, sketch data structures, and other network security applications because they offer high performance, low cost, reconfigurability, scalability, parallelism, and fast market availability [1].

There is not much work in literature where the performance of NC hash functions is evaluated on FPGA. As already discussed in Section 1, Sateesan et al. [29] proposed a novel NC hash function for Bloom filter called Xoodoo-NC that satisfies the avalanche criteria. The resulting architecture performed better than the other state-of-the-art Bloom filter implementations in terms of hardware operating frequency, delay, and occupied resources. Still, the authors did not report specifically on the performance and occupied resources of their used NC hash function called Xoodoo-NC. Extending the work of [29], Claesen et al. [6] chose five famous ciphers from literature (AES, GIFT, SKINNY, SPECK, and Pyjamask), reduced their internal states and rounds to derive NC hash functions while maintaining the same avalanche properties. They compared the performance on FPGA with state-of-the-art NC hash functions such as FNV1a, Xoodoo-NC, Murmur3, etc. One of their proposed algorithms, GIFT-NC, performed better than all in terms of computational delay though it consumed more Lookup Tables (LUTs) on FPGA.

Table 1: NC hash Functions using EC approaches

| Author | EC Method | Block Size | Fitness Function | Implementation |
|---|---|---|---|---|
| Estebanez,2014 [11] | GP | 32/32 | Avalanche Score | software |
| Dobai,2015 [9] | EA | 32/16 | Collision rate | software |
| Grochol,2016 [15] | LGP | 96/16 | Collision rate | software |
| Kidon,2017 [21] | GP | 32/16 | Collision rate | software |
| Grochol,2017 [16] | LGP,NSGA-II | 96/16 | Collision rate, execution time | software |
| Grochol,2018 [17] | LGP,NSGA-II | 96/16 | Collision rate, execution time | FPGA |
| Saez,2019 [28] | GP | 96/16 | Collision rate | software |
| Grochol,2020 [18] | CGP | 96/16 | Collision rate, execution time | software |
| Hu,2020 [20] | GP | 96/16 | Active flow estimation, Uniformity | software |

The last decade has seen reasonable work on using EC techniques for designing NC hash functions. Table 1 summarizes some of the prominent work in EC to evolve NC hash functions using various fitness functions. Estebanez et al. [11] used strict avalanche criteria as a fitness function to evolve 32-bit NC hash functions using GP. They produced remarkable results on different key sets with only 25 nodes (tree size), but their work has certain limitations. First, they used a multiplication operator to produce a good entropy in the output, but using this operator in the design is not recommended as it takes up a lot of FPGA resources. Second, the hash output size is small. Finally, the authors used Monte Carlo simulation to calculate the avalanche probability matrix to save computation time that considers a small subset of a large set. This is necessary as it is not feasible to check for all $32*2^{32}$ combinations. But at the same time, it is also paramount that the avalanche probability matrix should be based on the worst-case values for all possible input/output differences. This is important as the resulting hash function will provide assurance against millions of different kinds of keys.

In literature, many people use collision rate as a fitness function to evolve NC hash functions, although it has been proven that this only works for specific datasets [11]. In [21], Kidon et al. have used GP to evolve 32-bit domain-specific NC hash functions for hash tables. Their used fitness function measured the successful hashes until the first collision is detected. The resulting hash function performed 2.7 to 7 times faster and inserted 1.6% more keys than the state-of-the-art for specific datasets. Saez et al. [28] used the same GP algorithm as in their earlier work [11] but only changed the fitness function to collision rate to evolve a NC hash function called GP-Hash-AdHoc. They compared their results to the state-of-the-art, and for some datasets, they managed to improve upon collision rates. Another very interesting work is done by Dobai et al. [9], they have used an evolutionary algorithm to evolve a pair of NC hash functions for cuckoo filtering to be implemented on FPGA. The use of LFSRs and NLFSRs in the design of 32-bit hash functions is expected to be efficient on FPGA but they did not evaluate the performance on FPGA. In [20], Hu et al. used multi-objective GP to evolve the NC hash function to process IPv6 flow IDs of 288-bit length. Their main objective is to achieve better packet flow estimation while having a smaller number of collisions. The proposed method used three rounds to produce 16-bit hash outputs, and the results were comparable to the state-of-the-art on specific datasets in terms of active flow estimation and uniformity.

Grochol et al. [15] proposed the use of Linear Genetic Programming (LGP) to evolve NC hash functions to process 96-bit Network flow IDs. Further improving their own work, Grochol et al. [16] proposed a family of 16-bit hash functions using LGP and NSGA-II. They have used multi-objective fitness functions (collision rate and execution time). Additionally, they implemented their algorithms on FPGA in [17]. One of their designs clocked 739 MHz but this work is not comparable to our work because they have optimized the design with a very different fitness function that does not produce uniform hash functions. We verified this by testing their NC hash functions on the avalanche criteria explained

in Section 3. On changing the most significant bit 200k times, the entropy is close to zero, which clearly indicates it has poor randomness and is highly vulnerable to network attacks. Grochol et al. also experimented with CGP [18] to evolve NC hash functions to process network flow IDs of IPv4 and IPv6 packets. They reported similar performance for IPv4 flow ID processing as their previous work [15] and achieved good collision results for 320-bit IPv6 flow IDs as well.

All the above-described EC approaches except [11] are designed for specific datasets, hence they have mostly used collision resistance as the fitness function. They can only work on specific datasets and fail to generalize. They also produce either 16-bit or 32-bit hash outputs. The majority of the techniques use multiplication operators as well, which is a costly operator in terms of resources and computation time in a hardware implementation. Probabilistic architectures like Bloom filters require hash functions to be independent and have uniform output distribution, otherwise, the theoretical and simulation results for false positive rate (FPR) will have large discrepancy [27]. To ensure the above condition, one way is to evolve NC hash functions using avalanche metrics as a fitness measure to have a uniform distribution.

## 3 Design Criteria

The quality of NC hash functions can be analyzed based on the collision resistance, output distribution, avalanche effect, and speed of execution [11].

**Collision resistance:** When two different input values produce the same hash output, a hash collision occurs. Since collisions are inevitable in hash functions as the range of input values is larger than the output, it is important to reduce the collision rate for many applications. This property is highly data dependent [11] and works well with specific datasets only. For this reason, many EC techniques rely on this criterion for designing hash functions but are not suitable for Bloom filters for the above reasons.

**Output distribution:** Uneven output distribution will lead to clustering issues in the hash outputs and will have poor randomness. So a good NC hash function should have uniform output distribution [19].

**Speed:** The NC hash functions are mostly used in search operations, therefore they should be fast. This criterion highly depends upon the type of architecture used [11]. In the case of FPGAs, the choice and number of used operators matter the most in terms of hardware resources.

**Avalanche effect:** The Avalanche effect illustrates how significantly the output change for a small change in the input of the hash function. Satisfying the avalanche criteria ensures that there is sufficient randomness in the output, thus all the input statistical patterns are properly diffused. Avalanche criteria are not data-dependent. To measure the avalanche effect, we have used the three avalanche metrics that are typically used to evaluate hash functions and defined in the work of Daemen et al. [8]: avalanche dependence($D_{av}$), weight($W_{av}$), and entropy($E_{av}$).

*Avalanche dependence:* This tells us how many bits in the output may flip in response to a single input bit flip. The property is defined as:

$$D_{av} = t - \sum_{1}^{t} h(p[i]). \tag{1}$$

Here, $p$ is the probability vector, where $p$[i] represents the probability of each bit $i$ of output and $t$ is the total number of bits in the output. Further, *h(p)* = 1 if $p = 0$, and otherwise it is zero. This property generalizes full diffusion and is only satisfied when $D_{av} \simeq t$ for all outputs subject to a single-bit change in the input.
*Avalanche weight:* This is the expected Hamming weight of the difference in output for a single-bit change in the input ($\Delta$) and is defined as:

$$W_{av} = \sum_{1}^{t} p[i]. \tag{2}$$

To satisfy the condition, 50% of the output bits should be changed for $\Delta$, i.e., $W_{av} \simeq t/2$ for Hamming weight = 1. The avalanche weight generalizes the avalanche criteria.
*Avalanche entropy:* This metric can be used to measure the randomness of the distribution, and it is defined as:

$$E_{av} = \sum_{1}^{t} (-p[i] * log_2 p[i] - (1 - p[i]) * log_2(1 - p[i])). \tag{3}$$

Full entropy means that the output of the hash function is completely random and there are no useful patterns left in the data. This uncertainty makes it very difficult for the intruder to guess the sequence. Assuming an output bit size of $t$, an entropy of value $t$ has the same degree of uncertainty as a uniformly distributed $t$-bit random value [25]. This metric generalizes the Strict Avalanche Criteria (SAC) [31], which state that the probability of each output bit should be 1/2 on a single bit change in the input. This only happens when $E_{av} \simeq t$ is satisfied.

## 4 Design and Implementation

The primary objective of this research is to design NC hash functions for probabilistic data structures such as Bloom filters and Sketch data structures. Thus it is very important that the output of the hash function should be uniformly distributed [27]. Therefore, we seek to achieve the avalanche criterion as described in Section 3. This section delineates all the choices and decisions made during the design of the 96-bit GPNCH function.

### 4.1 Fitness function

The following two scenarios are considered for a fitness function.

**Scenario 1:** In this scenario, we use the sum of Bit dependence, Avalanche weight, and Entropy as a fitness function.

$$fitnessfunction = \sum(D_{\text{av}}, W_{\text{av}}, E_{\text{av}}). \tag{4}$$

The proposed GPNCH function input/output block size is 96-bit, so the perfect values are 96 for $D_{av}$ and $E_{av}$ and 48 for $W_{av}$. The sum of all these three metrics is 240, therefore, we have used this value as a fitness objective and evolved the individuals seeking to maximize this sum and get as close as possible to the perfect value of 240. That being said, it is not practical to calculate the hash values for all $96 * 2^{96}$ possible input bit strings for all the individuals in each generation, hence we decided to calculate the output difference one by one while keeping the rest of the input constant. This results in $96 * 25$ bit flips (M=25) in the first stage. We keep only the worst-case values for the three avalanche metrics($D_{av}$,$W_{av}$, and $E_{av}$). In the second stage, we extract the best individual and again calculate the scores for $96 * 2.5$ million bit flips (M=2.5 million) to ensure many input patterns are covered. The reported results are of the second stage.

**Scenario 2:** In the second scenario, we use Entropy as a fitness function.

$$fitnessfunction = E_{\text{av}}. \tag{5}$$

We hypothesized (later confirmed by experiments) that if the candidate solution achieves the maximum possible entropy (96 bits) then it should also have perfect values for $D_{av}$ and $W_{av}$ consequently.

Table 2: Elementary operators used in state-of-the-art NC hash functions

| Hash functions | Rotation | Shift | XOR | AND | OR | NOT | + | - | / | * |
|---|---|---|---|---|---|---|---|---|---|---|
| Xoodo NC | ✓ | | ✓ | ✓ | | | | | | |
| MurmurHash2 | | ✓ | ✓ | | | | | | | ✓ |
| MurmurHash3 | ✓ | | ✓ | | | | ✓ | | | ✓ |
| FNV | | | ✓ | | | | | | | ✓ |
| SuperFastHash | | ✓ | ✓ | | | | ✓ | | | |
| Lookup3 | ✓ | ✓ | ✓ | | | | ✓ | ✓ | | |
| FarmHash | ✓ | ✓ | ✓ | ✓ | | | ✓ | | | ✓ |
| CityHash | ✓ | ✓ | ✓ | | | | ✓ | | | ✓ |
| SipHash | ✓ | | ✓ | | | | | | | |
| DEK | | | ✓ | | | | | | | ✓ |
| GP-Hash-Adhoc/Generic | ✓ | | ✓ | | | | | | | ✓ |
| NSGAHash6 | ✓ | | ✓ | | | | ✓ | | | |

### 4.2 Terminal and Non-terminal set

For the non-terminal set, we gathered the elementary operators used by state-of-the-art NC hash functions as depicted in Table 2. The majority of the techniques use the multiplication operator as it produces significant entropy in the output, but it also consumes a lot of hardware resources on FPGA as is shown in [30], thus

we did not use this operator. The use of a division operator is also avoided for the same reasons [24, 2]. [11] has provided useful insights regarding the selection of operators for GP but our extensive experiments show that excluding any of the operators (Rotation, Swaps, XOR, AND, OR, NOT) degrades the avalanche results for 96-bit hash outputs. We have used 20 different rotation functions (10 left and 10 right) with circular shifts from 1 to 10 bits. The left and right circular rotations effectually produce the same results ($m \lll n = m \lll 96 - n$).

The swap operator also produces significant disorder in the output. Hence, we have used three swap operators of 48, 24, and 12 bits respectively. For the terminal nodes, we have used 96-bit inputs and ephemeral random constants (ERC). The ERC has the unique property that once randomly generated, it keeps its value throughout the evolution process.

### 4.3 Parameters Setting

The values of the hyperparameters control the overall run and help the algorithm fine-tune the results. All these values are tuned by doing extensive empirical analysis of various sets of combinations. Table 3 summarizes the settings under which the best results are obtained. Our experiments show that for a 96-bit hash output, the number of nodes should be large, otherwise avalanche metrics are not satisfied. This trial and error process suggested a required value of 550 for the number of nodes for the given primitive set. These seemingly large numbers of nodes are not the cause of concern since the final design contains a large proportion of nodes comprising rotations and swaps. These operations add negligible cost to FPGA resources. The selection of parents for the next generation of individuals has been performed through tournament selection. The size of the tournament is kept small to avoid premature convergence. Sometimes, the best solutions can be lost during evolution, hence elitism is used to preserve the best ones.

Table 3: Basic configuration of GPNCH functions

| Parameter | Value |
|---|---|
| Input/output block size | 96 bits |
| Basic function set | {XOR, AND, OR, NOT, ROT, Swaps, Constant} |
| Max generations | 1500 |
| Max nodes | 550 |
| Population size | 300 |
| Crossover Rate and type | 0.9 & one point |
| Mutation Rate | 0.3 |
| Mutation operator | uniform |
| Tree height(Min & Max) | 5 &10 |
| Tree height limit | 75 |
| Mutated tree height(Min & Max) | 1 & 3 |
| Mutated Tree type | Gen grow |
| Selection | Tournament |
| Tournament size | 3 |
| Elitism | 10 |
| Initialization | Gen full |

### 4.4 Stopping Criteria

To stop the evolution process, either of the following two conditions should be met:

1. The perfect fitness score, i.e., 240, is achieved.
2. The total number of generations is completed.

### 4.5 Software and Hardware Implementation

The Python DEAP framework [13], which is a fast prototyping evolutionary computing platform to design and test ideas, is used to code the GPNCH functions. It also supports multiprocessing to speed up the evaluations. The simulations are performed on a 32-core Intel Xeon 4208 CPU with a clock speed of 2.1 GHz.

The best individuals representing GPNCH functions are later implemented on the FPGA targeting Virtex Ultrascale+ xcvu7p-flvb2104-2-i chip using the Vivado 2022.1 to evaluate the operating speed, delay, throughput, and hardware resources. The general block diagram (Figure 2) shows that only a single round is needed to generate hash outputs.

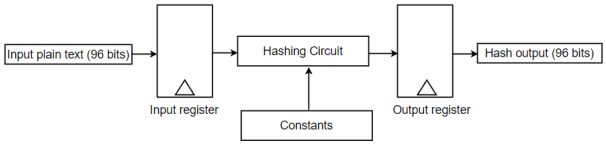



Fig. 2: GPNCH hash function hardware implementation

## 5 Results and Discussion

### 5.1 Software Evaluation

The first objective of this research work is to design the NC hash function that satisfies the avalanche metrics as described in Section 3 for a 96-bit block size. To achieve the said objective, around 450000 evaluations are performed across 1500 generations and a population size of 300 in each independent run. The fitness values are measured for both scenarios detailed in Section 4. They are plotted in Figure 3. The plots show the typical GP trait of the rapid increase in earlier generations. Once the basic structure of the design is set, the algorithm fine-tunes the values and gradually moves toward the objective.

For scenario 1 (fitness = $\sum D_{av}, W_{av}, E_{av}$), the objective is to achieve the fitness value of 240. As can be seen in Figure 3, the best fitness value is achieved in around 600 generations in one of the runs, and it does not improve further in later generations. On average, the individuals achieve the mean fitness value of approx. 220 in around 500 generations. In the second scenario (fitness = $E_{av}$), the goal is to get to the value of 96 to achieve the maximum possible entropy while also observing values for $D_{av}$ and $W_{av}$. In this case, the best fitness is achieved in approximately 1400 generations. This shows the stochastic nature of the GP algorithm.

The complete results of avalanche metrics for both scenarios are compiled and compared against the selected state-of-the-art NC hash functions in Table 4. As can be seen from the avalanche scores, both variants produced almost perfect

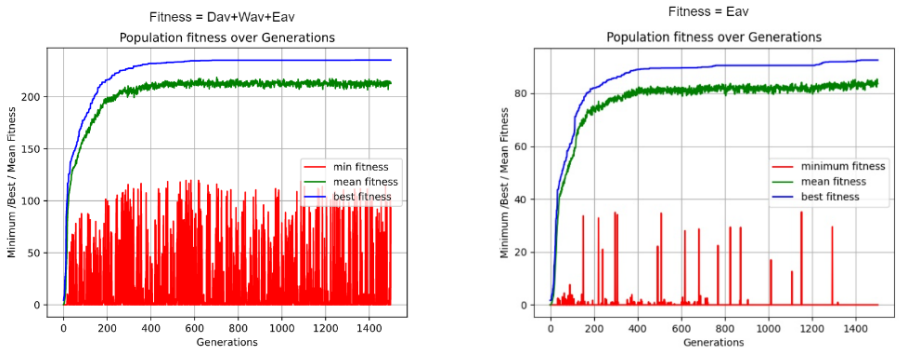


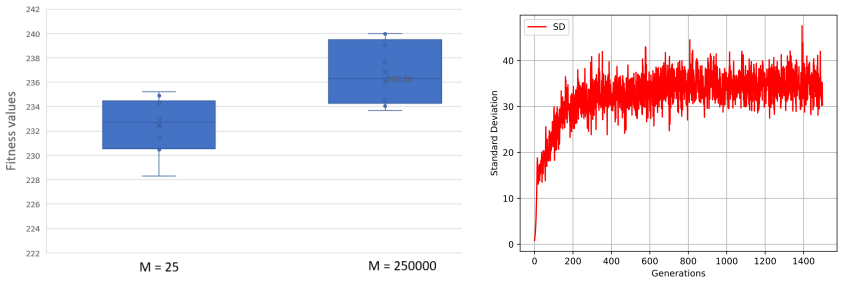

Fig. 3: Fitness function scores for both scenarios over the generations

fitness values. In comparison with other algorithms, the GPNCH achieved better avalanche scores, and that too in only a single round. Scenario 2 also confirmed our hypothesis that if the perfect entropy of 96 bits is achieved then $D_{av}$ and $W_{av}$ are also satisfied. Perfect entropy is only possible when all the bits in the probability vector, as described by Equation (3) have a probability of 0.5. This means for each input bit change; every output bit will have a 50% chance to change. Consequently, every output bit could be affected by a single bit change in each input which automatically satisfies the $D_{av}$. Moreover, SAC criteria, as described in Section 3 is also fully satisfied due to this perfect entropy. Further, the summation of all the bits in the probability vector leads to half of the bits being changed on average, which satisfies the $W_{av}$ as well. We have also tried to experiment with using $D_{av}$ and $W_{av}$ as fitness functions separately but the $E_{av}$ values were not up to the mark in all the runs. Due to the probabilistic nature of

Table 4: Comparison with the avalanche metrics of selected NC hash functions

| NC hash | Hash block size | $D_{av}$ | $W_{av}$ | $E_{av}$ | No.of rounds |
|---|---|---|---|---|---|
| Xoodo-NC | 96/96 | 96 | 47.324 | 95.864 | 3 |
| SPECK-NC | 96/96 | 96 | 47.45 | 95.68 | 7 |
| Pyjamask-NC | 96/96 | 96 | 47.73 | 95.96 | 3 |
| GIFT-NC | 96/96 | 128 | 63.39 | 127.95 | 7 |
| AES-NC | 96/96 | 128 | 63.0 | 127.43 | 3 |
| SKINNY-NC | 96/96 | 128 | 63.72 | 127.95 | 7 |
| GPNCH (scenario 1) | 96/96 | 96 | 48 | 95.99 | 1 |
| GPNCH (scenario 2) | 96/96 | 96 | 48 | 95.97 | 1 |

the GP algorithm, it is necessary to run the experiment enough times to ensure that results are consistent over several runs. As both fitness functions lead to equally good results, we chose the fitness function from scenario 1 to complete multiple runs. During the 10 independent runs, the average of the best fitness is approx. 237 when M = 250000. The complete results are shown in Figure 4(a). This figure shows that the results are very consistent, and the algorithm is extremely robust. Moreover, for all the 10 runs, the best individual is always found between 600-1500 generations, so there is no need to go beyond that.




(a) (b)

Fig. 4: (a) Box plot of fitness values for all the 10 independent runs; (b) Standard deviation of the population across generations for a single run

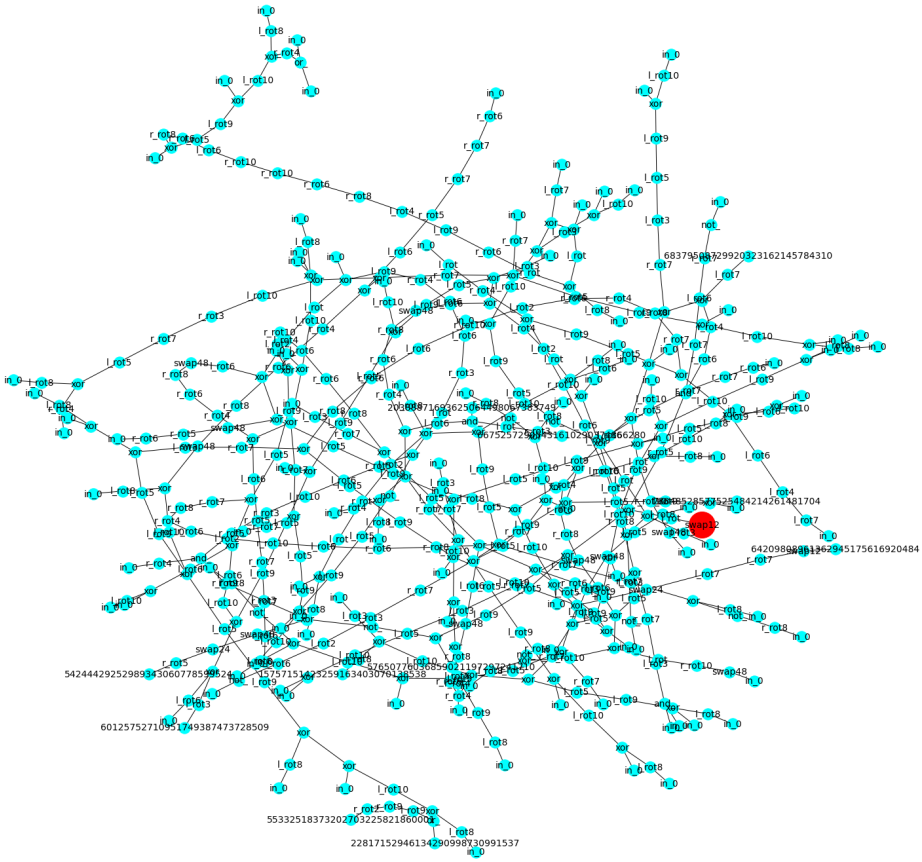


Fig. 5: Best individual representing the GPNCH function

The best fitness value, i.e., 239.99 is found during the $7^{th}$run of an algorithm. Figure 4(b) shows the standard deviation (SD) of the population for this run. The SD varies between 20 to 40, which is expected as the population size is high (300) and the mutation rate is 0.3. It indicates that the population has good variance at any given time, which is a good indicator of population diversity. Figure 5 shows the complete tree representing the one GPNCH function. As we said earlier, it can be observed that the majority of nodes are rotations and swaps, which barely add any resources when implemented on FPGA.

### 5.2 Hardware Evaluation

Obtaining high performance in hardware is the main focus of this research work. To carry out the evaluation, we selected the best run ($7th$) among the 10 completely independent runs. We extracted the top 10 individuals representing a family of 10 different GPNCH functions. All these NC functions are implemented on FPGA and the operating frequency, latency, and occupied hardware resources are measured and shown in Table 5. All the individuals achieved the fitness values of 239.99 which means they are equally good candidates to represent the NC hash function. This is a very important result as some networking search applications use separate independent uniform hash functions. The best individual is GPNCH-V, which has an operating frequency of 595.23 MHz, a throughput of 57.142 Gbps, and a latency of 1.68 ns. The complete design of all hash functions takes only 1 clock cycle to generate the result. When it comes to hardware resources, the design itself does not use any flip-flops and all individuals consume between 1446-1673 LUTs.

Table 5: Hardware evaluation of the top 10 best individuals

| Best individuals | Fitness | Max. operating frequency(MHz) | Throughput(Gbps) | No. of LUTs | Flip-flops | Latency(ns) |
|---|---|---|---|---|---|---|
| GPNCH-I | 239.99 | 595.23 | 57.142 | 1490 | 0 | 1.68 |
| GPNCH-II | 239.99 | 595.23 | 57.142 | 1471 | 0 | 1.68 |
| GPNCH-III | 239.99 | 595.23 | 57.142 | 1614 | 0 | 1.68 |
| GPNCH-IV | 239.99 | 591.71 | 56.804 | 1673 | 0 | 1.69 |
| **GPNCH-V** | **239.99** | **595.23** | **57.142** | **1446** | **0** | **1.68** |
| GPNCH-VI | 239.99 | 595.23 | 57.142 | 1475 | 0 | 1.68 |
| GPNCH-VII | 239.99 | 595.23 | 57.142 | 1462 | 0 | 1.68 |
| GPNCH-VIII | 239.99 | 595.23 | 57.142 | 1463 | 0 | 1.68 |
| GPNCH-IX | 239.99 | 595.23 | 57.142 | 1495 | 0 | 1.68 |
| GPNCH-X | 239.99 | 591.71 | 56.804 | 1483 | 0 | 1.69 |

**Comparison with related work** As we mentioned in Section 1, the NC hash functions designed by [6] and [29] are by far the best among all the known state-of-the-art NC hash functions when it comes to performance on FPGA for a 96-bit block size while satisfying the avalanche properties. Therefore, to have a fair comparison, we first implemented Xoodoo-NC on the same FPGA (Virtex Ultrascale+ xcvu7p-flvb2104-2-i) used in our work and by the authors of [6]. We chose the number of rounds for Xoodoo-NC to be 2.5 as the avalanche properties are satisfied at this stage [30]. We compared the latency, throughput, maximal operating frequency, and hardware resources. The complete results are

summarised in Table 6. It should be noted that in the implementation of GIFT-NC, AES-NC, and SKINNY-NC, the authors have kept the last 32 bits of the output block to zero, so effectively they have measured the hardware performance on the 96-bit block. As evident from Table 6, GPNCH-V outperforms Xoodoo-NC by 19% in throughput, frequency, and latency while an improvement of 62% is achieved compared to the GIFT-NC [6]. The speed-up we achieve comes at the cost of increased FPGA resources. In high-speed network monitoring and security applications, this increase in resources is negligible compared to the overall resources of the considered FPGA. We can therefore conclude that GPNCH offers a better alternative than state-of-the-art NC hash functions.

Finally, it is pointed out that the proposed solution, using Genetic Programming, is capable of generating multiple NC hash functions. Some approximate membership query data structures require multiple, independent NC hash functions, for example, Bloom filters, Cuckoo filters, and sketches. In contrast to the single NC hash functions that are reported in the literature, our solution could supply these data structures with the number of NC hash functions they require.

Table 6: Comparison with the related work

| NC Hash function | Block Size | Max. operating frequency(MHz) | Throughput (Gbps) | No. of LUTs | Latency (ns) |
|---|---|---|---|---|---|
| Xoodoo-NC (2.5 rounds)[1] | 96/96 | 500 | 48 | 480 | 2.00 |
| SPECK-NC | 96/96 | 171.6 | 16.47 | 432 | 5.82 |
| Pyjamask-NC | 96/96 | 279.4 | 26.82 | 832 | 3.58 |
| GIFT-NC | 96/128 | 369.7 | 35.49 | 546 | 2.70 |
| AES-NC | 96/128 | 255.9 | 24.56 | 2225 | 3.91 |
| SKINNY-NC | 96/128 | 202.4 | 19.44 | 2176 | 4.93 |
| **GPNCH-V** | **96/96** | **595.23** | **59.95** | **1446** | **1.68** |

[1]: these are the results of our implementation

## 6 Conclusion

This paper presents a novel non-cryptographic hash function family (GPNCH) through the use of Genetic Programming. We evolve non-cryptographic hash functions by satisfying the avalanche metrics for a 96-bit input/output block size. During the design phase, we propose two different fitness functions that can be used to satisfy the avalanche properties. We have simulated approx. 4.5 million evaluations to generate a family of the best possible uniform and independent NC hash functions. In order to evaluate the performance on the hardware setup, we have used an FPGA platform on which we compared the speed, the delay, the throughput, and the hardware resources with the state-of-art non-cryptographic hash functions. We show that our proposed method outperforms all the other non-cryptographic hash functions in terms of execution delay and throughput by at least 19%. This improvement is very important as the speed of probabilistic lookup and counting architectures depends heavily on the execution speed of hash functions. We next plan to use this GPNCH function family in the implementation of Bloom filters and report on the efficiency.

## Acknowledgments

This work is supported by the ESCALATE project, funded by FWO (G0E0719N) and SNSF (200021L_182005). This work is also supported by Cybersecurity Research Flanders (VR20192203) and the Higher Education Commission, Pakistan.

## References

1. The advantages of using FPGAs. https://www.af-inventions.de/en/services/fpga-development/fpga-advantages.html. Last accessed 15 Sept 2022.
2. FPGA optimization guide for intel® oneapi toolkits:developer guide. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/resource-use/data-types-and-operations/avoid-expensive-functions.html. Last accessed 20 Jan 2023.
3. Natural selection. https://education.nationalgeographic.org/resource/natural-selection/. Last accessed 15 Oct 2022.
4. Shifting to an FPGA data center future: How are FPGAs a potential solution? https://www.allaboutcircuits.com/news/shifting-to-a-field-programable-gate-array-data-center-future/. Last accessed 15 Sept 2022.
5. World's fastest internet network upgraded to staggering 46 terabit/s. https://newatlas.com/telecommunications/esnet6-worlds-fastest-internet-46-terabit-second/. Last accessed 01 Nov 2022.
6. T. Claesen, A. Sateesan, J. Vliegen, and N. Mentens. Novel non-cryptographic hash functions for networking and security applications on FPGA. In *2021 24th Euromicro conference on digital system design (DSD)*, pages 347–354. IEEE, 2021.
7. G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
8. J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. The design of xoodoo and xoofff. *IACR Transactions on Symmetric Cryptology*, pages 1–38, 2018.
9. R. Dobai and J. Korenek. Evolution of non-cryptographic hash function pairs for FPGA-based network applications. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1214–1219. IEEE, 2015.
10. A. E. Eiben, J. E. Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.
11. C. Estébanez, Y. Saez, G. Recio, and P. Isasi. Automatic design of noncryptographic hash functions using genetic programming. *Computational Intelligence*, 30(4):798–831, 2014.
12. P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.
13. F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné. Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012.
14. G. Fowler, K.-P. Vo, D. Eastlake, and T. Hansen. The FNV non-cryptographic hash algorithm. *Ietf-draft*, 2012.
15. D. Grochol and L. Sekanina. Evolutionary design of fast high-quality hash functions for network applications. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 901–908, 2016.

16. D. Grochol and L. Sekanina. Multi-objective evolution of hash functions for high speed networks. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 1533–1540. IEEE, 2017.
17. D. Grochol and L. 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.
18. D. Grochol and L. Sekanina. Evolutionary design of hash functions for IPv6 network flow hashing. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.
19. C. Henke, C. Schmoll, and T. Zseby. Empirical evaluation of hash functions for multipoint measurements. *ACM SIGCOMM Computer Communication Review*, 38(3):39–50, 2008.
20. Y. Hu, G. Cheng, Y. Tang, and F. Wang. A practical design of hash functions for ipv6 using multi-objective genetic programming. *Computer Communications*, 162:160–168, 2020.
21. M. Kidoň and R. Dobai. Evolutionary design of hash functions for IP address hashing using genetic programming. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 1720–1727. IEEE, 2017.
22. A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006.
23. L. Luo, D. Guo, R. T. Ma, O. Rottenstreich, and X. Luo. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2018.
24. K. Mannatunga and M. Perera. Performance evaluation of division algorithms in fpga. 2016.
25. NIST. Digital identity guidelines. Technical Report NIST Special Publication 800-63-3,INCLUDES UPDATES AS OF 03-02-2020, U.S. Department of Commerce, Washington, D.C., 2017.
26. S. Picek, B. Yang, V. Rozic, J. Vliegen, J. Winderickx, T. De Cnudde, and N. Mentens. Prngs for masking applications and their mapping to evolvable hardware. In *Smart Card Research and Advanced Applications: 15th International Conference, CARDIS 2016, Cannes, France, November 7–9, 2016, Revised Selected Papers 15*, pages 209–227. Springer, 2017.
27. M. Ramakrishna. Practical performance of bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237–1239, 1989.
28. Y. Saez, C. Estebanez, D. Quintana, and P. Isasi. Evolutionary hash functions for specific domains. *Applied Soft Computing*, 78:58–69, 2019.
29. A. Sateesan, J. Vliegen, J. Daemen, and 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)*, pages 262–269. IEEE, 2020.
30. A. Sateesan, J. Vliegen, J. Daemen, and N. Mentens. Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. *Microprocessors and Microsystems*, 93:104619, 2022.
31. A. Webster and S. E. Tavares. On the design of s-boxes. In *Conference on the theory and application of cryptographic techniques*, pages 523–534. Springer, 1985.