A Genetic Programming Approach for Hardware-Oriented Hash Functions for Network Security Applications

Mujtaba Hassan[a,*], Arish Sateesan[a], Jo Vliegen[a], Stjepan Picek[b], Nele Mentens[a,c]

[a] *ES&S and COSIC, ESAT, KU Leuven, Wetenschapspark 27, 3590, Diepenbeek, Belgium*
[b] *Digital Security group, Radboud University, Toernooiveld 212, 6525 EC, Nijmegen, The Netherlands*
[c] *LIACS, Leiden University, Einsteinweg 55, 2333 CC, Leiden, The Netherlands*

Abstract

Non-cryptographic (NC) hash functions are generally designed for speed and efficiency, which enables their use in many network security applications that require efficient lookup and counting, such as Bloom Filters and Count-Min (CM) Sketch structures. The performance of these structures heavily relies on underlying hash functions. Therefore, any advancement in the hash function design significantly impacts the overall performance of these structures. This paper presents a novel family of 32-bit NC hash functions (NCGPH-32) evolved using Genetic Programming (GP) and their corresponding implementation on Field Programmable Gate Arrays (FPGAs). This family of NC hash functions generates smaller hash values concatenated to produce larger hash outputs. Inspired by related work on 96-bit NC hash functions with GP, this work optimizes the performance of 32-bit NC hash functions on FPGA while achieving high scores on specific avalanche metrics (avalanche dependence, avalanche weight, and entropy) when considering concatenated 96-bit outputs. This optimization is of utmost importance to address the escalating demand for Terabit Ethernet networks, specifically

*Corresponding author

Email addresses: mujtaba.hassan@kuleuven.be (Mujtaba Hassan), arish.sateesan@kuleuven.be (Arish Sateesan), jo.vliegen@kuleuven.be (Jo Vliegen), stjepan.picek@ru.nl (Stjepan Picek), nele.mentens@kuleuven.be (Nele Mentens)

in processing real-time network flow IDs (identification and monitoring) at line rate. The throughput, latency, operating frequency, and resource utilization are evaluated on an FPGA and compared against 17 state-of-the-art NC hash functions. The results show that the proposed 96-bit concatenated hash function surpasses prior GP-based and other state-of-the-art NC hash functions by at least 36% in operating frequency, 30% in throughput and reduces latency by 27%. The demonstrated improvements in the hash design not only cater to the present demands of Terabit networks but also meet the expected near-future demands. Additionally, we integrate these hash functions into the Standard Bloom Filter (SBF) architecture and demonstrate comparable false positive rates (FPR) to state-of-the-art NC hash functions, affirming their effectiveness and applicability. We have also conducted several statistical tests on hash outputs of NCGPH-32 to demonstrate the high random nature and uniform distribution.

Keywords: Hash functions, Genetic Programming, Field Programmable Gate Arrays, Bloom Filters, Evolutionary Computation

---

1. Introduction

The increasing demand for processing large volumes of data at high speeds has necessitated the development of computer networks capable of supporting bandwidths as high as 46 Terabits/sec, with backbone links ranging from 400 Gbps to 1000 Gbps [1]. However, the benefits of these high-speed networks are not without drawbacks, as cybercriminals can now leverage such high-speed networks to launch large-scale flow attacks. For example, Cloudflare [2] recently reported a distributed denial-of-service attack (DDoS) with 71 million requests per second. Consequently, network packet monitoring and filtering systems must be able to process network packets at line rate and be future-proof to deal with even larger data flows. To mitigate such attacks, hardware accelerators, such as FPGAs, can be crucial in achieving this objective because they provide high speed, reconfigurability, and parallelism. Many large data centres, which form the foundation of cloud computing, are increasingly utilizing FPGAs for hardware acceleration [3]. This trend of using FPGAs for network applications has also been highlighted in [4]. A more recent endeavour employed the Intel Agilex FPGA to implement Network Intrusion Detection, achieving an impressive throughput of 400 Gbps [5].

Network monitoring tools frequently use network flow IDs to detect and

count large flows. For IPv4 networks, this flow ID can be modelled as a 96-bit vector [6] consisting of five fields, namely source and destination IP address, source and destination ports, and transmission protocol, although other combinations are also possible [7]. Network flow monitoring must be performed at line rate to meet the demanding bandwidth requirements of Terabit Ethernet networks (networks having a bandwidth greater than 100 Gbps). To detect and measure real-time large flows of data in such networks, probabilistic data structures (PDS) like CM sketches to count the frequency of items in a list) and Bloom filters (to detect the presence of an item in a list) are commonly utilized [7, 8, 9, 10, 11]. They outshine their exact counterparts due to their ability to fast query and update times while consuming less memory, albeit with a marginal trade-off in accuracy. The results show that these structures' performance highly depends upon the execution time of the hash functions used. Therefore, any optimization of the hash function execution time can significantly impact overall performance.

However, designing hash functions poses a formidable challenge due to their inherent complexity, highly nonlinear and counterintuitive properties and elusive relation between input and output variables. Moreover, hash functions should have uniform distributions and possess excellent avalanche properties; otherwise, theoretical and simulated false positive rates (FPR) of PDS such as Bloom filters will exhibit significant discrepancies [12, 10, 13, 14]. Failure to satisfy avalanche properties and uniformity will lead to certain bits within the Bloom filter having a greater likelihood of being set to one, primarily due to the elevated occurrence of hash collisions, consequently resulting in a high FPR (Bloom filters are explained in the next section).

Evolutionary Computation (EC) techniques excel in such scenarios as they prove highly effective when dealing with intricate and convoluted relationships between input and output variables [15]. In particular, when designing hardware-oriented hash functions, the methods such as tree-based Genetic Programming (GP) [15, 16], Cartesian Genetic Programming (CGP) [17], Linear Genetic Programming (LGP) [18, 19] and Gene Expression Programming (GEP) [20, 21] are adept at crafting and refining digital circuits. LGP, CGP, and tree-based GP are branches of GP that primarily differ in their representation of candidate solutions, while GEP combines the advantages of both tree-based GP and Genetic Algorithms (GA). All these methods leverage the same logical operations fundamental to digital circuit design, making them easily and efficiently deployable on hardware platforms [22].

To our surprise, there is very limited work available to design hardware-

friendly hash functions for PDS such as Bloom filters using EC. This motivates us to introduce a 32-bit family of uniform and independent hash functions for 96-bit inputs utilizing GP to process 96-bit network flow IDs on FPGA. A Bloom filter requires multiple independent hash functions to reduce the FPR as explained in [23]. Instead of using multiple hash functions, the hash designs in [7, 8, 10, 13] have suggested an alternative approach. They generated a single large hash value of 96-bit or higher and split it into smaller chunks to work as independent multiple hash functions. For instance, in the implementation of Bloom-1 architecture (a fast Bloom filter that requires a single memory read-out for each request) [10], initially, a certain number of bits, calculated as $log_2(l)$ are allocated to identify the memory locations (l) within the Bloom filter. Subsequently, the remaining bits are utilized to designate the membership words at each location. Typically, a membership word of 32 or 64 bits is used, corresponding to the data bus width.

In contrast, our study suggests a different approach. Rather than producing a single large 96-bit hash output, we propose concatenating the three most optimal hash functions from our 32-bit hash family to form a single 96-bit output. This strategy aims to enhance the FPGA's latency performance while maintaining a reasonable utilization of hardware resources. Moreover, our proposed concatenated hash function demonstrates similar performance in terms of the FPR when implemented in an SBF, as compared to the state-of-the-art FNV-1a and the 96-bit GP-based hash functions.

The contributions of our research can be summarized as follows.

- We propose a novel 32-bit NC hash family for 96-bit inputs using GP that is highly efficient on FPGA while achieving high scores on the avalanche metrics.

- We are the first to design a 96-bit hash output by concatenating the three best hash outputs from the 32-bit hash family using GP. The resulting 96-bit hash function performs better than the state-of-the-art 96-bit hash functions (that satisfy the avalanche criteria) on FPGA regarding operating frequency, throughput, and latency.

- We demonstrate the efficacy of our proposed design by implementing it in an SBF and showing that the FPR is comparable to state-of-the-art NC hash functions.

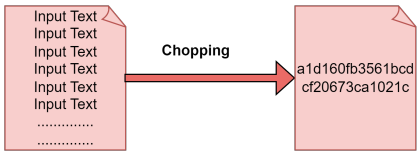

Figure 1: A hash function producing a fixed length code.

- Through conducting rigorous statistical tests, we demonstrate that the proposed NC hash functions exhibit favourable uniformity characteristics.

- Finally, our design produces a family of ten 32-bit hash functions that can be concatenated to produce hash values of length 32-320 bits. This is important, especially when PDS like Bloom filters require even large hash value sizes.

The remainder of this paper is organized as follows. In Section 2, we present some preliminary concepts necessary for understanding the context of this study. In Section 3, we provide an overview of existing NC hash functions. We discuss the design criteria of NC hash functions in Section 4. Section 5 presents the architecture and implementation of our proposed NCGPH-32 family. Section 6 discusses the results and provides a comparative analysis of our proposed design with state-of-the-art NC hash functions. In Section 7, we discuss the limitations and challenges. Finally, we conclude our work in Section 8.

2. Preliminaries

2.1. Hash Functions

Hash functions transform inputs of arbitrary length into outputs of fixed length, as illustrated in Fig. 1. There are two categories of hash functions: cryptographic hash functions and non-cryptographic (NC) hash functions. All hash functions should conform to the following properties:

- **Determinism:** means identical inputs invariably yield identical outputs.

- **Uniform distribution:** implies every possible outcome within a defined range should have an equal probability of occurring. This uniform distribution of potential outputs helps prevent clustering, thereby mitigating the risk of a large number of collisions.
- **Avalanche effect:** means a minor alteration in the input results in a significant number of alterations in the output. It also provides a high degree of unpredictability and randomness in the output.

In addition, a cryptographic hash function H must also provide the following strong security guarantees:

- **Pre-image resistance**: it must be computationally infeasible to find input m when H(m) = h, and h and H are given.
- **Second pre-image resistance**: it must be computationally infeasible to find another input n with $m \neq n$ such that H(m) = H(n) when m is given.
- **Collision resistance**: it must be computationally infeasible to find a pair of inputs (m,n) with $m \neq n$ such that H(m) = H(n).

These additional properties require a large number of rounds to satisfy, making cryptographic hash functions computationally expensive and rendering them unsuitable for applications that demand fast searching. Conversely, NC hash functions provide faster speeds at the cost of slightly weaker security guarantees. Therefore, NC hash functions are commonly used in applications that require fast lookups, such as Bloom filters, hash tables, and finding duplicate records.

2.2. Bloom Filters

A Bloom filter is an approximate data structure that uses fast and uniform hash functions with good avalanche properties to support membership queries in a set. Unlike conventional data structures, it does not store actual data, allowing it to represent large numbers of elements in a compact way. Inserting and searching for a component in the set is extremely fast, taking, on average, a constant time of $\mathcal{O}(k)$ to complete the process, where k is the number of hash operations. In hardware, these searches can be done in parallel because hash functions are independent. While the Bloom filter has several advantages, there is a caveat: it is probabilistic in nature. If an element is

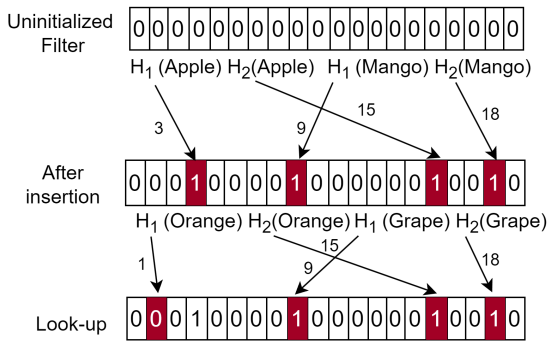

Figure 2: The Bloom filter insertion and query process.

not present in the set, it may still indicate that it is present with a small probability of error, known as a false positive. However, there is no possibility of false negatives, and 'not found' results are guaranteed to be true. This phenomenon is advantageous when the result is expected to be not found most of the time, thus saving memory access. False positives occur because the hash functions the filter uses might produce the same digest for different inputs, causing a collision that sets the bit on the same index multiple times. To reduce the probability of error, the filter load factor can be decreased, filter size can be increased, or multiple k hash functions can be used [23, 7, 9].

The principle working of the Standard Bloom Filter (SBF) for insertion and querying is straightforward and is demonstrated in Fig. 2. It uses a bit vector of size m, which is initialized to zero in the beginning. To insert a value n, the filter calculates multiple k hash values and uses the modulo m of these values as an index in the bit array. It goes to the corresponding indices and sets the bit to 1 at these locations. The process of finding an element is similar. Again, it calculates the hash values, and if all respective indices are set, it indicates that the value might be in the list. If any of the indexes are still zero, it indicates that the element is not present with a 100% guarantee.

Eq. (1) is widely used to calculate an FPR for the SBF, assuming that the probabilities of each bit being set are independent. Eq. (2) represents the optimal number of hash functions for the specific load factor.

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

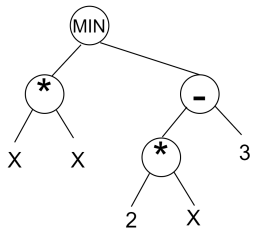

Figure 3: The tree representation of mathematical expression $min(x^2, 2 * x - 3)$.

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

2.3. Genetic Programming (GP)

In this section, we briefly explain GP along with its associated nomenclature. GP is one of many bio-inspired mechanisms defined in the field of evolutionary computation (EC), which aims to find improved solutions for a given problem by representing them as computer programs and evolving them iteratively, without prior knowledge of their exact structure [15]. In GP, solutions, referred to as individuals, are typically represented as syntax trees. Fig. 3 shows one possible representation of the computer program $min(x^2, 2 * x - 3)$. The tree structure consists of two types of nodes: leaf nodes, also known as terminals (such as x, 2, and 3), and internal nodes (such as multiply, subtract, and minimum), referred to as functions. The collective set of these functions and terminals is known as the primitive set within the GP framework.

Algorithm 1 GP Basic Steps

1: **BEGIN**
2: Generate a random population of individuals
   $G = \{g_0, g_1, ..., g_n\}$
3: **REPEAT**
4: Measure the fitness of each individual
5: Compare the fitness to the desired value
6: Select individuals to become parents
7: Create new individuals using genetic operators of crossover and mutation
8: Check for the stopping conditions
9: **Return the best individuals**
10: **STOP**

The basic steps of the GP process are summarized in Alg. 1. The algorithm initiates by generating a (large) set of potential solutions randomly, forming generation 0. These solutions constitute the initial population for the GP process. Subsequently, a series of steps (4-8) are executed iteratively to generate following generations, with the objective of yielding improved individuals. In step 4, each solution is evaluated based on predefined criteria, and a numerical fitness score is assigned to quantify its quality. This evaluation process assesses each solution's performance about the desired output or problem-specific objectives. Following the evaluation, step 5 involves comparing the fitness values among the solutions. Solutions with higher fitness scores are granted a greater likelihood of contributing their genetic material to the next generations.

The genetic operators of selection, crossover, and mutation are employed in steps 6-7 to generate new offspring individuals for subsequent generations. Selection entails identifying suitable parents from the current generation to produce offspring. Crossover involves combining selected parents' genetic material by exchanging parts of their solutions, thereby producing offspring solutions. Mutation introduces random modifications to the genetic material of selected individuals, thereby enhancing the exploration of the solution space and helping to maintain population diversity.

The steps (4-8) are repeated until a termination condition is met, such as finding the best individual solution or reaching a specified number of generations. Upon termination, the algorithm concludes in step 9 by returning the best-found individuals with the highest fitness scores achieved throughout

the evolution process.

3. Related Work

3.1. Non-EC Approaches

Many good NC hash functions have excellent avalanche and collision properties [16]. In a study by Estebanez et al. [24], the authors have provided a detailed review of commonly used NC hash functions. Despite possessing good hashing properties, they are not designed and optimized to perform well for specific hardware platforms like FPGA. For instance, authors in [10, 8] have reported the hardware latency of some of the widely used NC hash functions such as Murmur3, Fnv1-a, SipHash, and XORHash to be 3, 16, 4, and 7 clock cycles for 96-bit input blocks, respectively. This number of clock cycles, along with their low throughputs, renders those NC hash functions impractical to process network packets at line rate for Terabit Ethernet networks. To avoid complicated pipelining structures, in our work, we concentrate on hash functions that can be executed in one clock cycle.

A similar problem arises if we consider hash functions based on lightweight symmetric-key cryptography because their execution speed poses a significant bottleneck in their use in PDS. This slowdown is primarily attributed to the need for a considerable number of iterations, known as rounds, to fulfil stringent cryptographic security standards. As an example, the FPGA implementation of Photon and Sponge lightweight hash functions revealed a minimum cycle count of 12 and 45, respectively [25, 26, 27, 28], rendering them unsuitable for high-speed Terabit Ethernet networks due to high latency and low throughput. The 100G Ethernet subsystem (for example in the AMD Xilinx UltraScale+ platform) transmits and receives data at a rate of 512 bits per clock cycle. If we consider the smallest packet size to be 64 bytes, it is necessary to handle one packet within each clock cycle.

A cyclic redundancy check (CRC), commonly employed for error detection on communication links, can also serve as a non-cryptographic hash function thanks to its speed and reputation for achieving uniform distribution, particularly when the input size closely matches the output size. CRCs are capable of generating outputs of up to 128 bits. However, recent studies [29, 30] have underscored their limitations, such as high resource usage and latency on FPGAs, making them less ideal for large flow detection algorithms.

In [7], Sateesan et al. proposed the hardware-friendly NC hash function called Xoodo-NC to process 96-bit network flow IDs in one clock cycle. The

authors derived this function by reducing the number of rounds of its cryptographic counterpart while still maintaining the avalanche criteria. They used this hash function in the design of the Bloom filter for fast lookups by splitting the large output into smaller chunks, which they used as separate independent hash functions. The overall design outperformed all the other existing Bloom filter implementations on hardware in terms of lookup delay and occupied hardware resources while still maintaining a very good FPR. However, the study did not provide insights into the Xoodo-NC performance. Later, Claesen et al. [8] proposed five more hardware-specific NC hash functions by again reducing the internal state and number of rounds of famous cryptographic algorithms, namely AES, Pyjamask, GIFT, SKINNY, and SPECK to process 96-bit network flow IDs. All these hash functions processed the network flow IDs in one clock cycle. They compared their performance with Xoodoo-NC and other state-of-the-art, and the results showed that overall, Xoodoo-NC remained outstanding and could process 96 bits in just 2.75 ns. It was slightly bettered by their proposed NC hash called GIFT-NC, which had a latency of 2.70 ns at the cost of more hardware resources such as lookup tables (LUTs).

In the quest to match the increasing demand of present and future high-speed networks, Sateesan et al. [10] proposed three more NC hash functions by reducing the number of rounds of cryptographic ciphers TinyJAMBU-NC, PHOTON-NC, and Sparkle-NC. They compared the performance on FPGA, and one of their algorithms, PHOTON-NC, reduced the latency to 2.6 ns for the same 96-bit input blocks in one clock cycle. However, Xoodo-NC remained by far the most efficient regarding hardware resources and was only marginally behind in execution speed.

The main aim of our study is to further improve the operating frequency, latency, and throughput of NC hash functions on FPGA for PDS while still maintaining good avalanche properties to process 96-bit inputs. We propose to achieve this objective with the help of the EC approach.

3.2. EC-based Approaches

Grochol et al. [31] have utilized linear genetic programming (LGP) to construct a family of 16-bit NC hash functions, with hash collisions as a fitness metric. The proposed hash functions are specifically tailored for 96-bit network flow identifiers exhibiting slightly improved software execution speeds while delivering comparable collision rates against the state-of-the-art hash functions. Building upon their initial findings, they further refined their

approach by introducing a multiobjective optimization strategy [32], with execution time now considered alongside collision rates. They demonstrated a slight improvement over their previous design in terms of execution speed. Subsequently, they implemented these hash functions on FPGA platforms [6]. Comparative analyses against two hash functions, SipHash and XORHash, revealed that their proposed family consumed fewer FPGA resources while achieving comparable operating frequencies. Furthermore, in another study by them [33], they explored the application of Cartesian Genetic Programming (CGP) to tackle the processing of flow identifiers from both IPv4 and IPv6 networks. This investigation aimed to develop NC hash functions capable of accommodating 96-bit and 288-bit flow identifiers. Their findings highlighted that both CGP and LGP yielded solutions of similar quality and their solutions demonstrated parity with state-of-the-art approaches.

Estebanez et al. [24] introduced a novel approach by using genetic programming (GP) and leveraging the avalanche effect in crafting a family of 32-bit NC hash functions. The use of avalanche properties enabled the proposed hash functions to process inputs of diverse or random characteristics, as their effectiveness was not contingent upon the input nature. Subsequently, in a follow-up study [34], they employed collision as a fitness criterion to tailor hash functions specifically suited to distinct datasets.

In another very interesting work, Roland et al. [35] developed a pair of hash functions designed for FPGA-based network applications, specifically optimized for the cuckoo filters. Their approach involved the representation of hash functions using both linear and non-linear feedback shift registers (LFSRs, NLFSRs) alongside CGP, facilitating easier implementation on FPGA platforms. Their solution outperformed state-of-the-art alternatives in improving table load ratios for specific datasets. Building upon their initial work, they further refined their approach in a subsequent study [36], evolving hash functions based on NLFSRs within an FPGA-based network router. This evolution led to a remarkable 42% improvement in execution speed and a reduction of hardware resources by a factor of 15 compared to the benchmarked hash functions. Drawing inspiration from their study, Marek et al. [37] employed Genetic Programming (GP) to evolve hash functions tailored specifically for cuckoo hash tables, thereby enhancing table utilization even further.

However, it is important to note that the previously proposed approaches are not without their limitations. First, the majority of the EC-based approaches [31, 32, 33, 6, 35, 36, 37, 34] rely on the collision rate as a fitness

function. NC hash functions evolved solely based on this criterion may not possess the desired randomness properties required for PDS like Bloom filters. For instance, the entropy of the NSGAHash hash family, as proposed by Grochol et al. [6, 32], is evolved solely using collision criteria and is reported to be close to zero [13]. Additionally, this criterion is only suitable for specific datasets, as it exhibits a high degree of dependency on the data itself. This limits its adaptability to various datasets [24, 38].

Second, a number of techniques have employed multiplication operators during the evolution of NC hash functions [24, 35, 6, 36, 34, 37]. However, it should be noted that this operator is not recommended in the hardware design of NC hash functions [39, 37, 31] due to its complexity compared to other logical operations, resulting in increased hardware resource consumption. Moreover, the multiplication operator is also unsuitable when the objective is to achieve high operating frequencies, as it introduces a relatively longer critical path. The critical path refers to the longest path from input to output in a digital circuit that determines the maximum operating frequency of the digital circuit. As an illustration, when two 32-bit numbers were multiplied, and the design was implemented on an ultra-scale Virtex FPGA, it occupied 588 LUTs without Digital Signal Processors (DSPs), and with DSPs, it required 47 LUTs and 4 DSP blocks. In a study by Sateesan et al. [9], the authors have shown that the FPGA implementation of the NC hash function Murmur3, which uses 32-bit multiplications, consumes a larger number of hardware resources and has significantly lower operating frequency compared to their proposed NC hash function which uses no multiplication operator. Evidently, if this operator is utilized multiple times in the design of the NC hash function, this would have a negative impact on resources and implementation speed.

Lastly, certain methods have employed the Merkle-Damgård construction [36, 24, 34], evolving solely the mixing function to process input data block by block. The mixing function takes the present hash value and the next block as input and produces the new hash value as output. This approach is suitable for a universal class of hash functions that can process inputs of variable lengths. In cases where the input size is fixed, it is more efficient to process the entire input block together to achieve additional speed. This is especially critical for high-speed networks.

There has been a scarcity of research focusing on developing hardware-friendly and exceptionally fast NC hash functions using EC that simultaneously fulfil avalanche metrics. Consequently, Hassan et al. [13] put forward

a GP-based NC hash function, aiming to process 96-bit network flow IDs and produce a 96-bit output to be used in PDS-like Bloom filters. They aimed to divide the large output into smaller segments capable of functioning as independent hash functions within PDS. The resulting design achieved a commendable 19% reduction in hardware latency when compared to state-of-the-art hash functions while still preserving excellent avalanche properties. Remarkably, each hash was computed within only 1.68 ns, requiring a single clock cycle at an impressive clock speed of 595.5 MHz.

Our present study proposes an alternative approach where the same 96-bit output can be attained by concatenating three evolved 32-bit hash functions while maintaining good avalanche scores. By reducing the output size from 96 to 32 bits, we show that the number of nodes in the design can be significantly reduced, leading to an even lower latency.

4. NC Hash Function Design Criteria

The performance of NC hash functions is usually evaluated in terms of the following four performance metrics: collision rate (two different inputs producing the same output), avalanche effect (a minor alteration in the input of a hash function results in significant changes to its output), distribution of outputs (output should have uniform distribution), and speed (must be fast). It is worth noting that the avalanche criteria are independent of the data involved and, if satisfied, ensure that the resulting output possesses sufficient randomness, thereby diffusing all the statistical patterns present in the input. To assess the avalanche effect, the three metrics, namely avalanche dependence (D_{av}), weight (W_{av}), and entropy (E_{av}), as defined in the work of Daemen et al. [40], are typically used to evaluate hash functions. The avalanche entropy generalizes Strict Avalanche Criteria (SAC) [41] as well, which specifies that a change of one bit in the input should result in each output bit having a probability of 0.5. In a study of NC hash functions [24], it has been demonstrated that if the hash function satisfies SAC criteria, then it automatically leads to excellent uniform distribution and collision rate as a consequence. Hence, in this study, we have chosen avalanche metrics, as defined below, as a fitness measure and demonstrate its usability in implementing the SBF.

Avalanche dependence: This property describes the number of output bits

that may change when a single input bit is flipped. It is expressed as:

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

It is defined using a probability vector p, where each element p[i] represents the probability of the i-th output bit being flipped. The total number of output bits is denoted by t. The function $h(p)$ is equal to 1 if p is equal to 0, and 0 otherwise. This property is only satisfied when D_{av} is approximately equal to t for all possible outputs resulting from a single-bit change in the input. This property generalizes the full diffusion.

Avalanche weight: This refers to the predicted number of bits, on average, that will differ between two output values resulting from a single-bit change in the input. Mathematically it is expressed as follows:

$$W_{av} = \sum_{1}^{t} p[i]. \tag{4}$$

To satisfy the condition, 50% of the output bits should be changed for a single bit change, i.e., $W_{av} \simeq t/2$ for Hamming weight = 1. The avalanche weight generalizes the avalanche criteria.

Avalanche entropy: This measurement can be employed to assess the level of randomness in the distribution:

$$E_{av} = \sum_{1}^{t} (-p[i] * log_2 p[i] - (1 - p[i]) * log_2(1 - p[i])). \tag{5}$$

Full entropy refers to the condition where the resultant output generated by the hash function exhibits the highest degree of randomness, characterized by the absence of detectable patterns that hold practical relevance within the data.

5. Architecture and Implementation

To effectively encode any computational problem using GP, it is imperative to establish and specify the fundamental constituents, namely fitness functions, primitive set, stopping criteria, and parameters.

5.1. Fitness Function

As stated in Section 4, we have chosen avalanche metrics as a fitness function. Consequently, the evaluation of fitness is conducted in the following manner:

$$fitnessfunction = \sum(D_{\text{av}}, W_{av}, E_{\text{av}}). \tag{6}$$

As explained in Section 1, the intended output size of the proposed NC hash family is 32 bits. Therefore, the ideal values for D_{av} and E_{av} are 32, while for W_{av}, it is 16 as a consequence. When all three metrics are combined, they give a maximum possible fitness score of 80. Therefore, the objective of the proposed NC hash function is to achieve a value as close as possible to this score.

It is important to mention that evaluating these metrics for all potential 96-bit inputs for every individual in each generation is computationally infeasible due to the vast number of possible input combinations. For example, for a single individual, the number of possible combinations of input bit strings amounts to 96×2^{96}. Therefore, repeating this step for a population of 300 individuals over 3000 generations would require tremendous computing resources. Consequently, we decided to use the Monte Carlo simulation method [24, 13] and measure the values for these metrics in two steps. Initially, we evaluate all the individuals for all the generations for a smaller subset of input strings. Through various experiments, we have determined that 2400 bit flips are sufficient in terms of both available computing resources and validating the design on a large set of values in the second stage. Therefore, we calculate the output difference one by one while keeping the rest of the input constant resulting in $96 \times 25 = 2400$ bit flips, and store only the worst-case values. Examining worst-case scenarios is strongly recommended and helps ensure that the avalanche metrics consistently deliver satisfactory performance under any conceivable condition [7, 13]. Subsequently, we extract the best individuals and repeat the process with a very large number (24 million) of bit flips as recommended in the design of hash functions [40, 7, 13]. This ensures that a sufficiently large number of input patterns are covered. The reported results correspond to the outcomes obtained from the second step.

5.2. Stopping Criteria

The evolution will stop if either of the following two conditions are met:

- The ideal fitness value of 80 is achieved.
- The number of generations reaches to 3000 (justification is provided in Section 6).

5.3. Primitive Set

The present study draws inspiration from the work of Hassan et al. [13], which provides insights into identifying optimal operator combinations. Consequently, we have utilized the same set of operators (Rotation, SWAP, XOR, AND, OR, NOT) for the internal nodes of our proposed NC hash function. However, as the output size of individual hash functions is different compared to their work, we arbitrarily select the swap operators that encompass three distinct functions: SWAP16, SWAP8, and SWAP4. Each function interchanges the specified most significant bits (MSB) with the least significant bits (LSB) of 32-bit inputs. We experimentally identified that the absence of a SWAP operator slightly deteriorates the results and increasing their quantity beyond 3 also leads to unsatisfactory results. This implies that the inclusion of this operator leads to improved fitness values, however, its quantity should be kept to a minimum as it can replace other important operators such as XOR and Rotation.

We also conducted experiments involving various circular rotation operators containing 10, 20, and 30 rotations, respectively. Through experimentation, we discovered that employing only 10 rotations yielded suboptimal fitness values, while 30 rotations tended to overshadow other critical genetic components like XOR, OR and AND operations. We found the sweet spot at the inclusion of 20 circular rotations resulted in the attainment of high fitness scores. Therefore, we have employed a range of circular shifts, including 10 left shifts and 10 right shifts spanning from 1 to 10 bits. The outcomes obtained from performing left and right circular rotations are essentially identical ($m \lll n = m \ggg (32 - n)$). It is important to note that one complete GP run takes 2-4 days to complete as per available computing resources (see Sec.5.5), which limits our ability to perform exhaustive tests for the best possible combination.

For the leaf nodes, we have three 32-bit inputs alongside five 32-bit ephemeral random constants (ERC). To generate high randomness in the

output, we initially experimented with the inclusion of only one constant in the design and gradually increased the number of constants during various experiments, and the hash function produced the desired fitness scores at 5 constants. This is because constants also serve as secondary inputs and including constants in the design enhances the transformation process from input to output by introducing complexity and non-linearity, thereby amplifying the avalanche effect. Consequently, their strategic incorporation proves crucial in achieving high fitness scores. These ERCs possess a unique property: once randomly generated, their values remain unchanged throughout evolution.

5.4. Parameters

Table 1: General control parameters of GP run

No.	Parameter
1	Population size
2	Tree Size or Max nodes
3	Mutation Rate
4	Crossover Rate
5	Mutation Type
6	Crossover Type
7	Initialization method
8	Selection method
9	Tree height limit
10	Number of Generations
11	Elitism

Table 5.4 presents the control parameters for a typical GP run. It is difficult to provide general recommendations for optimal parameter settings due to their dependence on the specifics of the application. However, GP is generally robust, and general settings often work in most cases. For specific problems, these settings need to be experimentally evaluated [15]. Consequently, we investigated various settings to fine-tune the parameters to achieve the best possible results while maintaining acceptable avalanche metrics. Given the vast array of possibilities and the resource-intensive nature of testing and validating each one with multiple repetitions, conducting exhaustive exploration would demand significant computational resources. Thus, we opted to explore a limited set of possibilities, halting experimentation when fitness

values either markedly declined or ceased to improve further. We tested the maximum number of nodes in a tree (100-500), population size (200-500), tournament size (3-7), initialization method (Gen full, Gen grow, and Gen (half and half)), and generations (500-5000). In the Gen full method, a tree is created in which every leaf node has the same depth between min and max height, while in Gen grow, it varies. On the other hand, the Gen (half and half) method uses Gen grow half the time, while for the other half, it uses Gen full. The Gen half and half is the most common method to use, nevertheless, evolution will swiftly alter the population from its original state [15]. In our case, we did not notice any advantage of one initialization method over the other.

The hashing design demonstrates optimal fitness values when exceeding 300 nodes, indicating that smaller tree sizes struggle to sufficiently shuffle bits, resulting in low entropy and diminished randomness. While larger trees yield favourable results, their evaluation incurs heightened computational costs without additional benefits.

The population size plays a pivotal role in solution exploration, albeit at the expense of increased computational demands. Our findings underscore the necessity of scaling the population size to at least 300 nodes to attain satisfactory fitness levels, emphasizing the significance of extensive exploration for achieving optimal solutions.

Furthermore, the tournament size, crossover rate, and mutation rate are critical parameters for balancing selection pressure, exploration, and exploitation. While we did not explore the joint influences of all these three parameters, our observations (conducted univariate) indicate a slight decline in fitness values with larger tournament sizes while keeping the population size constant, a concern that can be mitigated by upscaling the population size. However, resource constraints precluded further exploration.

After extensive experimentation, we chose population size (300), total generations (3000), tournament size (3), a single point crossover with rate (0.9) and mutation rate (0.3). The crossover and mutation rates are selected based on recommendations provided in [13]. For the initialization of the population, we selected Gen full and limited the tree height to 75 because the DEAP framework [42] relies on a Python parser, which restricts the expression to just 91 basic elements.

5.5. Design Realization

The Python DEAP framework [42] is employed as a rapid prototyping platform for EC to facilitate the design and evaluation of our settings. Using this framework, all the simulations are executed on a 32-core Intel Xeon 4208 CPU operating at a clock speed of 2.1 GHz. Since GP is intrinsically a computationally expensive technique, we have calculated the fitness values of individuals in parallel using multiple available cores to speed up the evolutionary process. Later, the evolved hash functions are implemented on FPGA. More specifically, the Virtex Ultrascale+ xcvu7p-flvb2104-2-i chip is selected as the target platform, considering that the majority of comparable work presented in Section 6 also utilized this chip for design implementation. Hence, the hardware performance comparison is fair. The evaluation encompassed assessing the operational speed, delay, throughput, and utilization of hardware resources accomplished by applying the Vivado 2022.1 synthesis tool.

6. Results and Analysis

6.1. Fitness of NC Hash Functions/Individuals

Utilizing the parameters as discussed in Sections 5.3 and 5.4, a comprehensive evaluation encompassing approximately 9 million iterations is conducted to attain an optimal fitness score of 80. The experiment is repeated ten times, and during the third iteration, the most favourable fitness values are achieved. During the evolution process, we used elitism to keep the ten best-performing individuals of each generation intact and passed them unchanged to the subsequent generations. Consequently, we received the ten best individuals based on their fitness values at the end of evolution for each independent run. From the pool of ten individuals, we extracted the best three individuals representing NC hash functions based on their fitness scores, and their respective outcomes are detailed in Table 2. All individuals demonstrated high fitness scores based on avalanche metrics, closely approaching the desired values.

The population fitness and corresponding standard deviation (S.D.) across generations for this specific run are depicted in Fig. 4. It is evident that fitness values experience rapid improvement in the initial generations, conforming to the typical behaviour exhibited by the GP. However, the rate of variation between the 1000th and 2000th generations remains relatively stable. Subsequently, negligible enhancements are observed after the 2500th generation,

Table 2: Avalanche scores of selected individuals.

NC hash	Hash block size	Fitness	D_{av}	W_{av}	E_{av}
NCGPH32-I	96/32	78	32	15.36	30.59
NCGPH32-II	96/32	78	32	15.61	30.4
NCGPH32-III	96/32	78.14	32	15.58	30.56

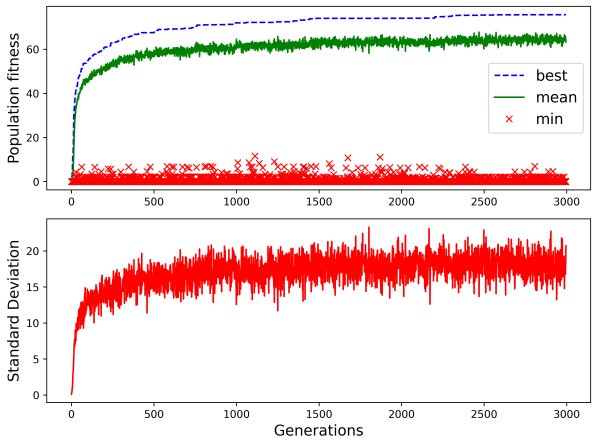

Figure 4: Population fitness and standard deviation over generations.

leading us to the conclusion that conducting further experiments beyond 3000 generations would yield minimal benefits. The S.D. ranges from 15 to 22, indicating an acceptable level of diversity within the population across generations. Over the ten experiments, the highest fitness scores achieved ranged from 63 to 78.14, with an average value of around 74. These findings demonstrate the efficacy of our tuned parameter settings.

6.2. Tree Branches and Height Analysis

Within this subsection, we present a comprehensive analysis of the different branches within the top three individuals obtained from Experiment 3 (NCGPH32-I-III), as well as the leading individual from Experiment 10 (NCGPH32-IV) and the 96-bit GP-based hash function (GPNCH-V) [13]. The statistical results are presented in Table 3. The inclusion of NCGPH32-IV is justified by its high fitness score of 77.73 despite possessing a relatively low tree height of only 32.

Table 3: Tree branches and height analysis.

Individuals	Total Nodes (TN)	Total Logic Gates (LG)	XOR	OR	AND	NOT	Constants	Tree Height	LG in Largest Branch	LG in any Branch
NCGPH32-I (Exp. 3)	300	100	79	21	0	0	0	67	28	28
NCGPH32-II (Exp. 3)	300	100	79	21	0	0	0	67	28	28
NCGPH32-III (Exp. 3)	300	100	79	21	0	0	0	67	28	28
NCGPH32-IV (Exp. 10)	300	111	97	14	0	0	02	32	17	18
GPNCH-V [13]	550	125	109	02	04	10	12	74	24	26

Initially, our design hypothesis posited that the shorter branches in the tree may lead to high operating frequencies and low logical depth when the design is implemented on FPGA. Since the operating frequency is dependent upon the longest path in the design and the shorter branches will have fewer logical gates, it could possibly lead to low logical depth in hardware as well. Consequently, we conducted an extensive analysis of all branches within the tree structure, considering the branch with the greatest number of logical gates as the largest because the presence of rotation and swap nodes incur negligible hardware implementation costs. As revealed in Table 3, the longest branch may not necessarily possess the highest logical gates. Based on this analysis, it was anticipated that (NCGPH32-IV) would exhibit the significantly highest operating frequency and lowest latency, given its inclusion of only 18 logic gates in the largest branch, compared to 28 and 26 logic gates, respectively, in the compared individuals. Nevertheless, this assumption was proven incorrect, and its explanation is provided in the subsequent section.

It can also be observed from Table 3 that the best three individuals derived from Experiment 3 do not incorporate AND or NOT logic gates in their final designs, nor do they include constants. However, our experimental findings reveal that excluding these elements from the primitive set impedes the attainment of satisfactory avalanche scores, thus underscoring their crucial role in the evolution process.

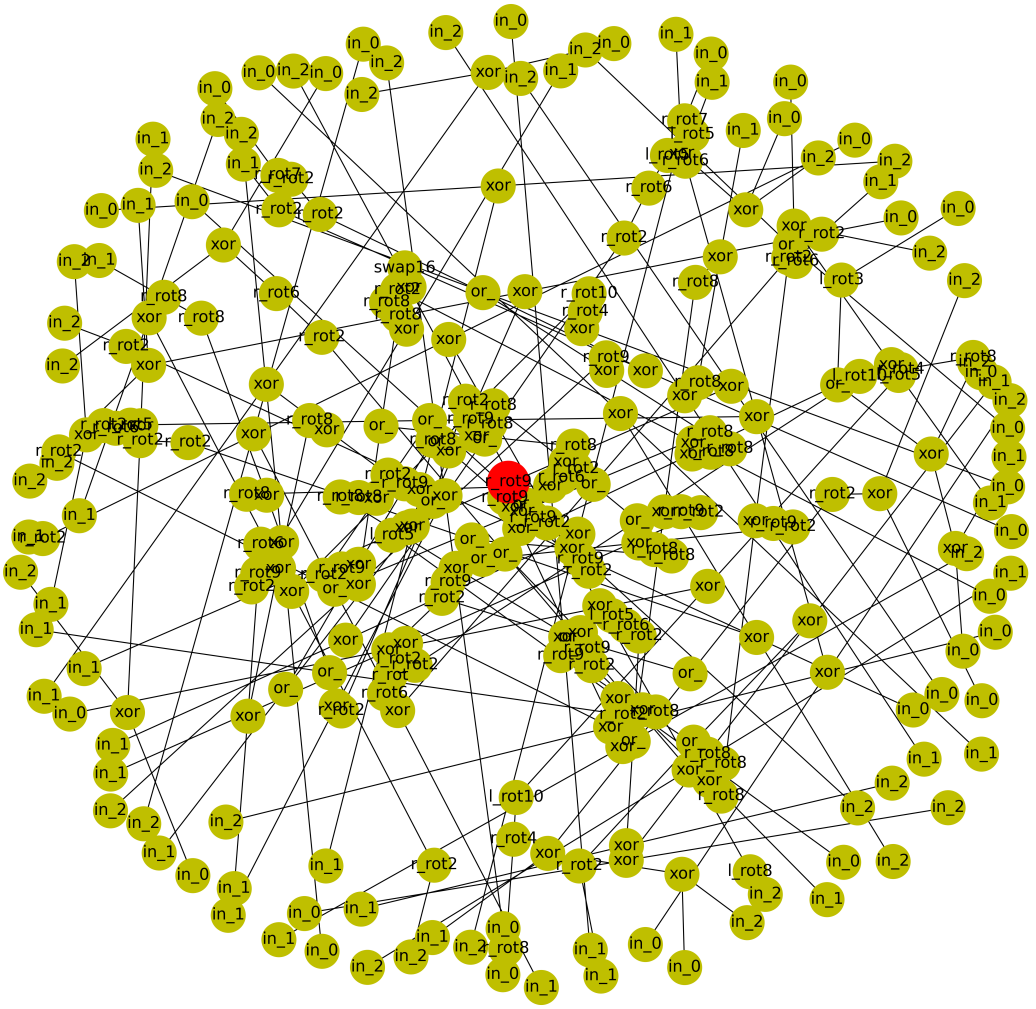

Figure 5: Best individual representing the NC hash function.

Fig. 5 illustrates the syntax tree representation of the NC hash function NCGPH32-I. Numerous parallel paths towards the Root node can be

observed, and a significant proportion of nodes rely on rotation and swaps, which have minimal impact on hardware implementation costs. Consequently, the execution of this function is fast on hardware platforms.

Table 4: Hardware performance analysis of NC hash functions.

Individuals	Total Nodes(TN)	Max. operating frequency(MHz)	Overall Latency(ns)	Total LUT's	Fan out (Mean)	Fan out (S.D)	Logic Delay (Mean)	Logic Delay (S.D)	Routing Delay (Mean)	Routing Delay (S.D)
NCGPH32-I (Exp. 3)	300	847	1.18	567	20	1.94	0.41	0.08	0.71	0.08
NCGPH32-II (Exp. 3)	300	840	1.19	561	22	1.72	0.44	0.05	0.69	0.05
NCGPH32-III (Exp. 3)	300	813	1.23	555	21	2.04	0.41	0.05	0.78	0.06
NCGPH32 -IV (Exp. 10)	300	847	1.18	492	15	1.47	0.38	0.06	0.73	0.05
GPNCH-V[13]	550	595	1.68	1446	45	4.10	0.46	0.06	1.16	0.07

6.3. Performance of NC Hash Functions/Individuals on FPGA

To assess the FPGA implementations of the best hash functions generated from Experiment 3 and the leading hash function from Experiment 10, we conduct a series of experiments. The obtained results are presented in Table 4. The design utilizes a combinatorial path with no registers between the terminals and the root node, resulting in an efficient NC hash function generation. All the NC hash functions derived from Experiment 3 and Experiment 10 exhibit high operating frequencies, low latency, and minimal resource utilization in terms of LUTs. Moreover, these designs can generate hash values within a single clock cycle. Among these implementations, NCGPH32-I achieves the highest clock speed of 847 MHz, accompanied by a latency of 1.18 ns. Conversely, NCGPH32-III demonstrates the slowest clock speed, operating at 813 MHz with a latency of 1.23 ns. Interestingly, NCGPH32-IV achieves the same operating speed and latency as NCGPH32-I despite having a significantly smaller tree height of only 32 compared to the 67 of NCGPH32-I. To gain further insights into the underlying factors contributing to the high operating speed and low latency of these implementations, we conduct an analysis of their FPGA designs and compare these implementations with the FPGA design of the 96-bit GP-based hash function (GPNCH-V) proposed in [13].

Operating frequency depends on the propagation delay, comprising logical delay and routing delay (net delay). Logical delay is influenced by the computational elements utilized (in our case, logical gates), while routing delay refers to the delay incurred as the signal traverses through various wiring paths. Consequently, we compute the mean and standard deviation (S.D) of the ten worst paths for all the NC hash functions. Notably, there is no substantial difference in the mean values of logical delay among the five NC

hash functions, and the small variations align with the number of logical gates employed in their designs, as indicated in Table 3. However, in terms of routing delay, a significant disparity is observed, particularly in the case of GPNCH-V, which exhibits approximately 64% higher routing delay. A correlation between routing delay and the number of nodes used becomes evident upon closer examination. GPNCH-V utilizes 550 nodes, while the NCGPH32 I-IV designs employ only 300 nodes, resulting in a higher mean fan-out of 45 for the ten worst paths in GPNCH-V. In comparison, there is no substantial difference in the mean fan-out values for NCGPH32 I-IV, as they all utilize 300 nodes. Fan-out refers to the number of logical elements driven by the same input signal. These findings indicate that routing delay can be reduced by minimizing the number of input nodes in the software design, which should be considered more significant than the tree's height. The high fan-out problem can also be addressed during the FPGA implementation. For example, by duplicating the input value into multiple registers in the design. However, this approach increases the number of flip-flops.

6.4. Statistical Tests for Uniformity

Table 5: Chi-Square Goodness of Fit Test statistics

NC hash	Total Chi Square	P-Value	Mean of Ones	Mean of Zeros
NCGPH32-I	10030.75	0.41	16.0082	15.9918
NCGPH32-II	10092.62	0.25	16.0149	15.9851
NCGPH32-III	9987.12	0.53	15.9981	16.0019

The best three performing individuals or NC hash functions derived from Experiment 3 exhibited a high level of entropy, indicating that the binary sequences they produce have a random nature and a significant degree of non-predictability. To further substantiate this claim, we conducted the Chi-Square Goodness of Fit Test [43], Frequency Test and the Wald-Wolfowitz Runs Test to examine whether any evidence of non-uniformity exists in the binary sequences generated by the three NC hash functions. For all the tests, we generated 10000 binary outputs from each of the three best-performing hash functions.

6.4.1. Chi-Square Goodness of Fit Test

The results of this test are presented in Table 5. We formulated the Null hypothesis (h_0), proposing that the binary sequences produced by the

three best individuals conform to a uniform distribution. Since each binary sequence has a length of 32 bits with two possible outcomes per bit, the degree of freedom (d.o.f) for each binary sequence is 1 in this case. Consequently, the overall degree of freedom for the 10000 binary sequences is calculated as $1 * 10000$.

With a chosen significance level of $\alpha = 0.01$ and the given d.o.f, the corresponding theoretical value of the χ^2 statistic is determined as 10331.93. $\alpha = 0.01$ means that there is only a 1% chance of a false positive, and this criterion is considered to be very strict [44]. Subsequently, we computed the Chi-Square values for occurrences of both zeros and ones across all 10000 binary sequences, summing them together. Additionally, we assessed the associated P-values, which serve as an indicator of the strength of evidence against the Null hypothesis. Notably, the measured χ^2 values for all three NC hash functions are found to be lower than their respective theoretical reference values, and the calculated P-values are greater than 0.01. Hence, the results are determined to lack statistical significance, leading to the inability to reject the Null hypothesis. This outcome supports the indication that the binary sequences are random in nature. Furthermore, we computed the mean of occurrences for both zeros and ones across all 10000 binary sequences for all three NC hash functions, revealing an equitable distribution of these numbers.

6.4.2. Frequency Test for Uniformity

The results of this test can be seen in Fig. 6. In a uniform distribution, the occurrence of zeros and ones at each bit location should ideally approach 1/2. Observation of the values across all bit positions reveals alignment with the expected value of 5000, indicating that the output of the hash functions exhibits randomness.

6.4.3. Frequency (Monobit) Test for Uniformity

The purpose of this test is to count the total number of ones and zeros in each binary sequence and compare it with the number of ones and zeros that is expected for truly random sequences. We used the χ^2 with $\alpha = 0.01$ and for NCGPH32-I to III, 99.40%, 99.33% and 99.47% sequences, respectively passed the test. For acceptable results, the proportion should lie above 96.85% for 10000 sequences [44].

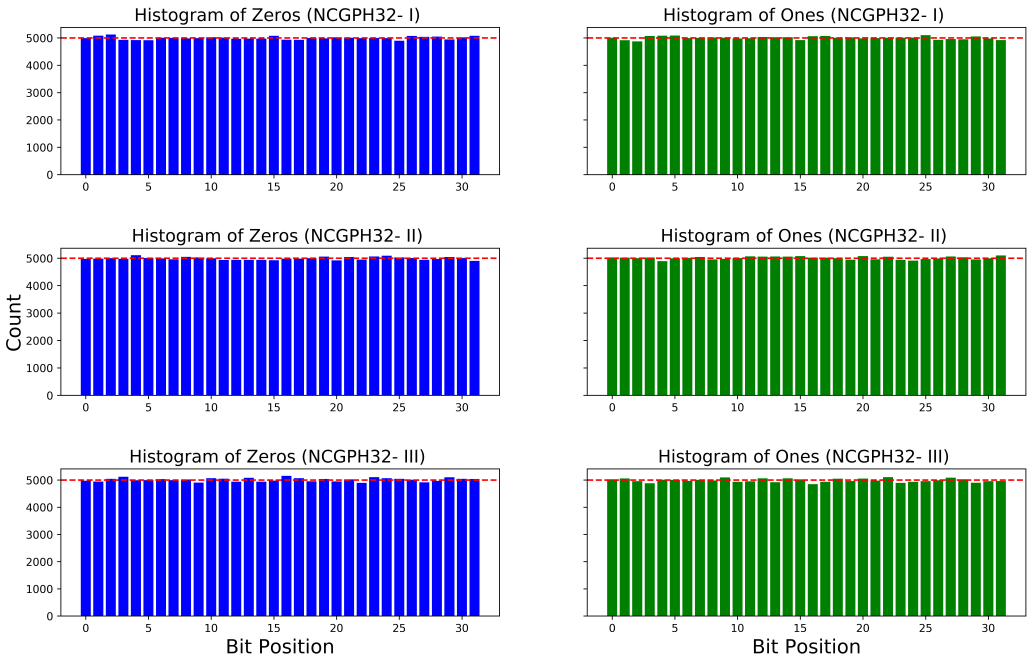

Figure 6: Frequency distribution of zeros and ones for each bit position.

6.4.4. Wald-Wolfowitz Runs Test for Uniformity

The Runs test aims to ascertain if the number of runs of ones and zeros of different lengths aligns with what would be anticipated in a random sequence. A run refers to a continuous sequence of identical binary digits without interruption. For $\alpha = 0.01$, all binary sequences for all the three hash functions passed the test indicating a random nature of generated sequences.

6.5. Comparison with the Related Work

The primary objective of this study revolves around the design of an NC hash function capable of processing a fixed input size of 96-bit and generating a corresponding 96-bit output. To achieve this, we propose the combination of the 32-bit outputs from the three most promising NC hash functions, resulting in a concatenated 96-bit output. Given that these NC hash functions are separate and independent entities, they can operate in parallel on an FPGA platform with an overall computational complexity of 1.23 ns.

To assess the performance of our proposed NCGPH96-concat hash function, we conducted a comparative analysis of its execution speed and resource

Table 6: Comparison with the related work.

No.	NC hash function	Block Size	Max. operating frequency (MHz)	Throughput (Gbps)	No. of LUTs	Latency (ns)
1	Murmur3 [10]	96/64	120.6	2.57	567	24.87
2	Xoodoo-NC (2.5 rounds) [13]	96/96	500	48	480	2.00
3	Xoodoo-NC [10]	96/192	363.6	34.9	309	2.75
4	SPECK-NC [8]	96/96	171.6	16.47	432	5.82
5	Pyjamask-NC [8]	96/96	279.4	26.82	832	3.58
6	TinyJAMBU-NC [10]	96/96	138.9	13.33	1520	7.2
7	GPNCH-V [13]	96/96	595.23	59.95	1446	1.68
8	SipHash [10, 6]	96/16	182.8	1.463	1061	21.88
9	XORHash [10, 6]	96/16	627.3	2.868	291	11.13
10	NSGAHash-VII [10, 6]	96/16	184.1	1.473	80	21.72
11	FNV-1a [10]	96/128	122.9	0.92	567	130.08
12	GIFT-NC [8]	96/128	369.7	35.49	546	2.70
13	AES-NC [8]	96/128	255.9	24.56	2225	3.91
14	SKINNY-NC [8]	96/128	202.4	19.44	2176	4.93
15	Sparkle-NC [10]	96/128	133.3	12.79	811	7.2
16	PHOTON-NC [10]	96/144	384.6	36.92	1338	2.6
17	CRC-128 [29]	128/32	312.5	40	3511	3.2
18	**NCGPH96-concat (Ours)**	96/96	**813**	**78.04**	1683	**1.23**

utilization against a set of state-of-the-art hardware-oriented NC hash functions and results are presented in Table 6. The proposed NCGPH96-concat function surpasses all the compared NC hash functions regarding operating speed, latency, and throughput while maintaining a reasonable number of LUTs. Specifically, when compared to the previously reported GPNCH-V function, known to have the best computational speed and throughput for 96-bit inputs, the NCGPH96-concat hash function demonstrates improvements of at least 36% in operating speed and 30% in throughput, as well as a 27% decrease in latency. It is important to acknowledge that a trade-off between speed and logical resources is inherent, and as a consequence, the proposed NC hash function employs slightly more LUTs. Nevertheless, this slight increase in resource utilization represents a reasonable compromise considering the substantial gains in computational speed. Moreover, modern FPGA platforms have a significant number of LUTs at their disposal, rendering the marginal increase easily justifiable.

While the Xoodoo-NC hash function remains more efficient in terms of throughput to LUT ratio, our proposed NC hash function surpasses it by approximately 63% in terms of operating frequency and throughput while decreasing latency by 39%. It is also worth mentioning that Sateesan et al. in [9] reported the operating frequency, latency and throughput of Xoodo-

NC(96/96) to be 666.67 MHz, 1.5 ns and 64 Gbps respectively. Though the results are very promising, the evaluations are performed only for 2 rounds [45] when the avalanche properties were not satisfied. Therefore, we did not consider these results.

The two widely adopted NC hash functions, namely Murmur3 and Fnv-1a, exhibit considerable latency values of 23.87 ns and 130.08 ns, respectively. This latency can be primarily attributed to their reliance on multiplication operators, rendering them unsuitable for high-speed Terabit Ethernet networks, as we already discussed in Section 3. In particular, the reasonably low number of LUTs employed by these hash functions is due to the utilization of 29 and 38 DSP slices in their designs, whereas our proposed NC hash functions do not rely on DSPs. In Table 6, DSP slices are not reported, but they do contribute to a total number of resources utilized on the FPGA.

The XORHash demonstrates a very high operating frequency of 627.3 MHz, but its throughput is almost 27 times less than our proposed hash function. In a series of works performed in [7, 9, 8], the PHOTON-NC is their best-performing NC hash function in terms of execution speed and throughput and our proposed hash function surpasses this by 160% in operating frequency, 111% in throughput and 53% in latency.

It is also prudent to note that the FPGA implementations of SipHash, XORhash and NSGAHash-VII utilize Virtex 7 boards, while CRC-128 employs a Virtex 6 board. Although these boards are comparatively less advanced than the platform utilized for our NCGPH96-concat and others listed in Table 6, the observed differences in latency and throughput remain considerable.

6.6. Case Study: Integration of NC Hash in SBF

To assess the effectiveness of our proposed NCGPH96-concat function, we conducted an evaluation by integrating it into the SBF and examining its performance in terms of the FPR across different load factors ($L = m/n$), where m represents the filter size and n denotes the number of insertions. We selected two alternative NC hash functions for comparison: GPNCH-V, which exhibits perfect avalanche scores and uniformity, and FNV-1a (32-bit), a widely employed NC hash function known for its excellent uniformity properties [7, 9].

To ensure a fair comparison, we maintained a constant value for m at 4096 bits. For GPNCH-V and NCGPH96-concat, we partitioned the 96-bit output into 12-bit chunks to accommodate a maximum of 8 hash functions,

Table 7: FPR comparison of SBF employing various hash functions.

Filter Size (m)	Insertions (n)	Optimal Hash Functions (k)	Theoretical FPR	Observed FPR		
				FNV-1a	GPNCH-V	NCGPH96-concat
4096	400	7	0.0073 (L=10)	0.0081	0.0077	0.0077
4096	500	6	0.0196 (L=8)	0.0224	0.0174	0.0194
4096	1000	3	0.1400 (L=4)	0.1444	0.1376	0.1406

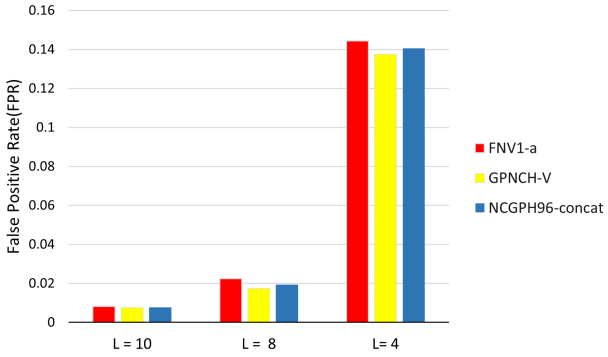

Figure 7: FPR comparison of SBF employing various hash functions.

which proved sufficient for the tested load ratios. By employing Eq. (2), we determined the optimal number of hash functions and subsequently calculated the theoretical FPR using Eq. (1). To obtain the observed FPR, we repeated the process 20 times for each load factor with different inputs, and the comprehensive results are presented in Fig. 7 and Table 7.

We tested all three hash functions by filling the SBF capacity to approximately 10%, 12%, and 25% as represented by load factors (L) of 10, 8, and 4, respectively. Upon analysis, it is evident that for varying load factors, the FPR values of all the hash functions are consistently similar and closely align with the theoretical FPR for the same load factors. This observation signifies that our proposed NCGPH96-concat hash function exhibits good dispersion properties.

7. Limitations and Challenges

One of the major challenges in our adapted approach utilizing GP to design hash functions is the requirement for large computation resources. In our case, one complete evolutionary run takes around 2 to 4 days to complete, which hinders our ability to test all possible parameter settings; therefore, we believe the performance of the proposed hash function can be

further increased if more resources are available. It would also be interesting to evaluate the fitness of each individual on FPGA by keeping it in a loop during the training phase of hash functions in software or implementing the whole evolution process on FPGA itself. This would be highly challenging but holds the potential to significantly decrease the performance evaluation times and provide the opportunity to test a wide range of parameter possibilities.

8. Conclusions and Future Work

This work presents a novel contribution in the form of a family of 32-bit NC hash functions generated through the utilization of GP. These NC hash functions are specifically designed to process 96-bit flow IDs for IPv4 networks. The selection process involved approximately 9 million evaluations conducted in ten independent runs, leading to the identification of three NC hash functions based on their performance in terms of avalanche metrics. To evaluate the quality of the selected NC hash functions, we conducted statistical tests on their outputs, revealing a favourable uniform distribution. Furthermore, we implemented these hash functions on an FPGA platform and conducted extensive measurements encompassing operating frequency, throughput, latency, and resource utilization.

Subsequently, we combined the three selected NC hash functions to generate an output of 96 bits, enabling their integration into the design of PDS. To assess the performance of our concatenated NC hash function, we compared its hardware performance against 17 other state-of-the-art hardware-friendly NC hash functions with similar input block sizes. The results demonstrate that the NCGPH96-concat hash function exhibits a very high operating speed and reduced latency. It improves the operating speed by at least 36%, latency by 27%, and also achieves a 30% increase in throughput, albeit with a moderate increase in LUTs. These improvements are vital for the real-time packet monitoring and filtering systems, that use FPGAs and PDS to detect malicious flows at line rate for high-speed Terabit Ethernet networks.

As a practical evaluation, we integrated the NCGPH96-concat hash function into the design of an SBF and compared its FPR with that of two other uniform NC hash functions across various load factors. The results indicate that our proposed NC hash function performs on par with the alternatives in terms of FPR. Furthermore, the proposed NC hash family comprises ten nearly equally efficient hash functions. Thus, in scenarios where PDS like the Bloom filter necessitates an augmented number of NC hash functions,

the output size can be expanded beyond 96 bits. For instance, if the SBF size is increased from 4096 to 8192 while preserving identical load factors, it potentially requires the utilization of up to 14 hash functions to optimize FPR. Consequently, accommodating all memory locations may necessitate the allocation of a 192-bit output, partitioned into 13-bit hash functions.

In our future work, we aim to reduce the number of nodes in the software design while maintaining good avalanche scores. A smaller number of nodes will not only decrease the evaluation time in software but will help decrease propagation delay on the FPGA and improve the occupied resources in terms of LUTs. It would be interesting to compare and contrast the performance of CGP and GEP-based implementations with the proposed tree-based GP method. CGP allows nodes to be used multiple times, thereby potentially reducing the number of nodes needed, which could lead to higher throughput and lower latency on FPGA. In the case of GEP, unlike GP, any type of genetic operator can be applied, allowing for a more thorough exploration of the solution space and increasing the likelihood of finding better solutions. Additionally, GEP benefits from smaller population sizes and fixed-size chromosomes, enabling the testing of various parameter settings that can result in smaller and fitter solutions in less time than GP. Another challenging aspect could be the development of hardware-friendly hash functions for IPv6 packets with a flow ID of 288 bits.

CRediT authorship contribution statement

Mujtaba Hassan: Conceptualization, Methodology, Formal Analysis, Software, Writing – original draft, Validation. Arish Sateesan: Conceptualization, Software. Jo Vliegen: Validation, Writing – review & editing. Stjepan Picek: Methodology, Writing – review & editing. Nele Mentens: Supervision, Writing – review & editing

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

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

References

[1] M. Irving, World's fastest internet network upgraded to staggering 46 Terabit/s, last accessed 26 April 2023.
URL `https://newatlas.com/telecommunications/esnet6-worlds-fastest-internet-46-terabit-second`

[2] Y. Omer, D. Julien, F. Alex, Cloudflare mitigates record-breaking 71 million request-per-second DDoS attack, last accessed 26 April 2023.
URL `https://blog.cloudflare.com/cloudflare-mitigates-record-breaking-71-million-request-per-second-ddos-attack`

[3] Data Centers are Short on Compute Capacity, last accessed 26 April 2023.
URL `https://www.intel.com/content/www/us/en/products/docs/storage/programmable/overview.html`

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

[5] V. Košař, L. Šišmiš, J. Matoušek, J. Kořenek, Accelerating ids using tls pre-filter in fpga, in: 2023 IEEE Symposium on Computers and Communications (ISCC), 2023, pp. 436–442. doi:10.1109/ISCC58397.2023.10218049.

[6] D. Grochol, L. Sekanina, Fast reconfigurable hash functions for network flow hashing in FPGAs, in: 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), IEEE, 2018, pp. 257–263.

[7] 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), IEEE, 2020, pp. 262–269.

[8] T. Claesen, A. Sateesan, J. Vliegen, N. 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.

[9] A. Sateesan, J. Vliegen, J. Daemen, N. Mentens, Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications, Microprocessors and Microsystems 93 (2022) 104619.

[10] A. Sateesan, J. Biesmans, T. Claesen, J. Vliegen, N. Mentens, Optimized algorithms and architectures for fast non-cryptographic hash functions in hardware, Microprocessors and Microsystems 98 (2023) 104782.

[11] A. Sateesan, J. Vliegen, S. Scherrer, H.-C. Hsiao, A. Perrig, N. Mentens, Speed records in network flow measurement on fpga, in: 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), 2021, pp. 219–224. doi:10.1109/FPL53798.2021.00043.

[12] M. Ramakrishna, Practical performance of bloom filters and parallel free-text searching, Communications of the ACM 32 (10) (1989) 1237–1239.

[13] M. Hassan, A. Sateesan, J. Vliegen, S. Picek, N. Mentens, Evolving Non-cryptographic Hash Functions Using Genetic Programming for High-speed Lookups in Network Security Applications, in: Applications of Evolutionary Computation, Springer Nature Switzerland, Cham, 2023, pp. 302–318.

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

[15] M. O'Neill, Riccardo Poli, William B. Langdon, Nicholas F. McPhee: A Field Guide to Genetic Programming: Lulu. com, 2008, 250 pp, ISBN 978-1-4092-0073-4 (2009).

[16] Y. Saez, C. Estebanez, D. Quintana, P. Isasi, Evolutionary hash functions for specific domains, Applied Soft Computing 78 (2019) 58–69.

[17] J. Miller, A. Turner, Cartesian genetic programming, in: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, 2015, pp. 179–198.

[18] D. Grochol, L. Sekanina, Comparison of parallel linear genetic programming implementations, in: Recent Advances in Soft Computing: Proceedings of the 22nd International Conference on Soft Computing (MENDEL 2016) held in Brno, Czech Republic, at June 8-10, 2016 22, Springer, 2017, pp. 64–76.

[19] D. Grochol, L. Sekanina, Multi-objective evolution of ultra-fast general-purpose hash functions, in: Genetic Programming: 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings 21, Springer, 2018, pp. 187–202.

[20] J. Zhong, L. Feng, Y.-S. Ong, Gene expression programming: A survey, IEEE Computational Intelligence Magazine 12 (3) (2017) 54–72.

[21] S. Varrette, J. Muszyński, P. Bouvry, Hash function generation by means of gene expression programming, Annales Universitatis Mariae Curie-Skłodowska. Sectio AI, Informatica 12 (3) (2012) 37–53.

[22] S. Picek, B. Yang, V. Rozic, J. Vliegen, J. Winderickx, T. De Cnudde, 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, Springer, 2017, pp. 209–227.

[23] Y. Qiao, T. Li, S. Chen, Fast Bloom Filters and Their Generalization, IEEE Transactions on Parallel and Distributed Systems 25 (1) (2014) 93–103. doi:10.1109/TPDS.2013.46.

[24] C. Estébanez, Y. Saez, G. Recio, P. Isasi, Automatic design of noncryptographic hash functions using genetic programming, Computational Intelligence 30 (4) (2014) 798–831.

[25] M. O. A. Al-Shatari, F. A. Hussin, A. Abd Aziz, G. Witjaksono, X.-T. Tran, Fpga-based lightweight hardware architecture of the photon hash function for iot edge devices, IEEE Access 8 (2020) 207610–207618.

[26] C. A. Lara-Nino, M. Morales-Sandoval, A. Diaz-Perez, Small lightweight hash functions in fpga, in: 2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS), IEEE, 2018, pp. 1–4.

[27] N. Nalla Anandakumar, T. Peyrin, A. Poschmann, A very compact fpga implementation of led and photon, in: Progress in Cryptology–INDOCRYPT 2014: 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings 15, Springer, 2014, pp. 304–321.

[28] B. Jungk, L. R. Lima, M. Hiller, A systematic study of lightweight hash functions on fpgas, in: 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), IEEE, 2014, pp. 1–6.

[29] Bajarangbali, P. A. Anand, Design of high speed crc algorithm for ethernet on fpga using reduced lookup table algorithm, in: 2016 IEEE Annual India Conference (INDICON), 2016, pp. 1–6. doi:10.1109/INDICON.2016.7839009.

[30] Q. Clark Shen, J. C. Vega, P. Chow, Parallel crc on an fpga at terabit speeds, in: 2022 International Conference on Field-Programmable Technology (ICFPT), 2022, pp. 1–6. doi:10.1109/ICFPT56656.2022.9974233.

[31] D. Grochol, L. Sekanina, Evolutionary design of fast high-quality hash functions for network applications, in: Proceedings of the Genetic and Evolutionary Computation Conference 2016, 2016, pp. 901–908.

[32] D. Grochol, L. Sekanina, Multi-objective evolution of hash functions for high speed networks, in: 2017 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2017, pp. 1533–1540.

[33] D. Grochol, L. Sekanina, Evolutionary Design of Hash Functions for IPv6 Network Flow Hashing, in: 2020 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2020, pp. 1–8.

[34] Y. Saez, C. Estebanez, D. Quintana, P. Isasi, Evolutionary hash functions for specific domains, Applied Soft Computing 78 (2019) 58–69.

[35] R. Dobai, J. Korenek, Evolution of non-cryptographic hash function pairs for FPGA-based network applications, in: 2015 IEEE Symposium Series on Computational Intelligence, IEEE, 2015, pp. 1214–1219.

[36] R. Dobai, J. Korenek, L. Sekanina, Adaptive development of hash functions in fpga-based network routers, in: 2016 IEEE Symposium Series on Computational Intelligence (SSCI), 2016, pp. 1–8. doi:10.1109/SSCI.2016.7850171.

[37] M. Kidoň, R. Dobai, Evolutionary design of hash functions for IP address hashing using genetic programming, in: 2017 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2017, pp. 1720–1727.

[38] A. Valloud, Hashing in smallTalk: Theory and Practice, na, 2008.

[39] N. Hua, E. Norige, S. Kumar, B. Lynch, Non-crypto Hardware Hash Functions for High Performance Networking ASICs, in: 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, 2011, pp. 156–166. doi:10.1109/ANCS.2011.32.

[40] J. Daemen, S. Hoffert, G. Van Assche, R. Van Keer, The design of Xoodoo and Xoofff, IACR Transactions on Symmetric Cryptology (2018) 1–38.

[41] A. F. Webster, S. E. Tavares, On the design of S-boxes, in: Advances in Cryptology—CRYPTO'85 Proceedings, Springer, 2000, pp. 523–534.

[42] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy, The Journal of Machine Learning Research 13 (1) (2012) 2171–2175.

[43] C. R. Kothari, Research methodology: Methods and techniques, New Age International, 2004.

[44] L. E. Bassham III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, et al., Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications, National Institute of Standards & Technology, 2010.

[45] A. Sateesan, Fpga design for large flow detection in high-speed networks (2024-01-26).