Evolutionary Design of Hash Functions for IPv6 Network Flow Hashing

David Grochol and Lukas Sekanina
Brno University of Technology, Faculty of Information Technology
IT4Innovations Centre of Excellence
Brno, Czech Republic
Email: igrochol@fit.vutbr.cz, sekanina@fit.vutbr.cz

***Abstract*—Fast and high-quality network flow hashing is an essential operation in many high-speed network systems such as network monitoring probes. We propose a multi-objective evolutionary design method capable of evolving hash functions for IPv4 and IPv6 flow hashing. Our approach combines Cartesian genetic programming (CGP) with Non-dominated sorting genetic algorithm II (NSGA-II) and aims to optimize not only the quality of hashing, but also the execution time of the hash function. The evolved hash functions are evaluated on real data sets collected in computer network and compared against other evolved and conventionally created hash functions.**

***Index Terms*—Cartesian genetic programming, linear genetic programming, hash function, network flow, Internet protocol.**

I. Introduction

Network monitoring is an important collection of tasks that has to be performed in any non-trivial computer network. It is based on probing of device states, collecting of traffic information and traffic analysis. Results of monitoring are useful for administrators to improve network security, performance and functionality. Current high-speed networks (with 40 Gbps and higher throughputs) require novel solutions to traffic monitoring because the monitoring systems developed for previous standards (10 Gbps and lower) do not have enough throughput. As software solutions are often insufficient in terms of performance, hardware implementations of monitoring systems, based on field-programmable gate arrays (FPGA) or application-specific integrated circuits, are currently employed to ensure a sufficient throughput. The monitoring is conducted at the level of *network flows*, where a network flow is a collection of packets having some common features. One of the recent approaches is called *Software Defined Monitoring* (SDM). In SDM most operations are conducted in fast programmable hardware and the remaining (more complex) operations are left for software processing [1].

Network *flow hashing* is one of the most frequently executed operations in network monitoring systems. Hash functions are typically used for searching in rule tables, for distributing the flow data to process units and for storing the flow data to a database. For example, in the distribution unit, a hash function must be called for each incoming packet. Less than 7 ns can be spent on a common processor to process this packet in a 100 Gbps network [1]. In order to maximize the performance of network monitoring systems, hashing has to be not only of a high quality, but also fast. For hashing of network flows, the so-called XOR folding has been proposed [2]. In addition to the human designs of hash functions, an automated approach has been developed. It is based on linear genetic programming (LGP) and single- as well as multi-objective LGP versions were proposed [3], [4]. However, these methods consider IPv4 flow hashing only, which is a serious limitation for modern computer networks utilizing Internet Protocol version 6 (IPv6).

In this paper, we propose a multi-objective evolutionary design system capable of evolving fast and high quality hash functions for IPv6 flow hashing. The system is based on Cartesian genetic programming (CGP) which is combined with Non-dominated sorting genetic algorithm II (NSGA-II) and the aim is to optimize not only the quality of hashing, but also the execution time of hash functions on a common processor.

Another challenge is that the CGP setup needed for IPv6 flow hashing increases the search space and the task becomes more complex for genetic programming. In the IPv4 flow hashing with LGP, the input vector dimension is 96 bits [3]. The proposed CGP-based solution works with a 320 bit input vector as both the source and destination addresses are stored on 128 bits and other bits are needed to store the source and destination ports and transport protocol. This additional complexity has to be appropriately handled during the design of our CGP-based method.

The evolved hash functions are evaluated on real data sets collected in IPv4 and IPv6 networks and compared against hash functions evolved with LGP and other 11 commonly used hash functions.

The rest of the paper is organized as follows. Section II introduces the principles of hash functions and their design, surveys the hash functions evolved in the past and discusses evolutionary design of hash functions for network flow hashing. In Section III, CGP is presented as a method suitable for the evolutionary design of hash functions and, in connection with NSGA-II, as a method capable of optimizing not only the quality of hashing, but also the time of hashing in the context of IPv4 and IPv6 networks. Results are presented in Section IV, in which we first compare LGP and CGP in the design of IPv4 flow hash functions and then present evolved hash functions for IPv6 flows. Conclusions are given in Section V.

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

II. Relevant Work

A. Hash Functions

For the scope of this paper, let us define the hash function as a mathematical functions h that accepts an n-bit string and produces another m-bit string, where $n \gg m$. The resulting bit string is usually interpreted as a natural number and called a *hash value* (or a *hash*) [5]. In the case of network flow hashing, the n bits are extracted from key features of each flow. The number of output bits (m) determines the number of items (i.e. 2^m) that one needs to distinguish. For example, it is the number of slots in the *hash table*. The use of a hash function thus enables to quickly identify the flow in the incoming packet, assuming that the hash is easy to compute. However, as $n \gg m$, many input vectors are inevitably mapped to the same output value (hash). This situation is called a *collision*.

In the case of hash tables, colliding input vectors are usually handled by means of a set of linear lists containing the flow identifiers having the same hash. The flow is then recognized either in constant time (no collision) or in linear time (with respect to the number of flows in the list that are assigned to a given slot). Other approaches to collision resolving are the open addressing or cuckoo hashing [6].

A good hash function produces a low number of collisions. It is also required that two similar input vectors produce very different output vectors. That property is called an avalanche effect and can be formalized in various ways. For cryptographic applications, some additional requirements on hash functions are specified. However, the network flow hashing is usually implemented with non-cryptographic hash functions.

Literature offers many good hash functions created by experts, for example, DJBHash [7], DEKHash [5], FVN (Fowler-Noll-Vo) [8], One At Time, Lookup3 [9], MurmurHash2, MurmurHash3 [10] and CityHash [11]. Specifically for the IPv4 network flow hashing the so-called xor folding (XORHash function) was proposed. In this function, the items of the flow identifier are shifted by a predetermined number of bits and then aggregated with the xor operation [2]. Details of various common hash functions are summarized in [12].

B. Evolved Hash Functions

Evolutionary algorithms (EA) have been applied to the design of universal as well as application-specific non-cryptographic hash functions for more than two decades. EAs are popular in this domain as they excel in creating complex non-linear functions and producing high-quality functions optimized for a given application. Table I surveys the most relevant EA-based methods available in the literature on the subject. Because hash functions are usually constructed using elementary arithmetic and logic operators, most EA branches can easily be adopted for the hash function design, including LGP, CGP, tree GP (TGP), grammatical evolution (GE) and even GAs optimizing parameters of various FPGA-based reconfigurable circuits implementing candidate hash functions. The fitness function is usually defined by means of collision counting for some training data; however; other methods (such as formalizing and measuring the avalanche effects) have also been proposed. In most cases, the problem is formulated as a single-objective optimization problem, but papers [4], [13] deal with a multi-objective approach in which the number of collisions is minimized together with the hash function execution time on a processor.

TABLE I
Methods for EA-based design of non-cryptographic hash functions

Paper	EA method	Fitness	Application
Damiani, 1998 [15]	GA-FPGA	collisions	universal
Berarducci, 2004 [16]	GE	collisions	universal
Estébanez, 2006 [17]	TGP	avalanche	universal
Widiger, 2006 [18]	GA-FPGA	collisions	packet classif.
Safdari, 2009 [19]	GA	collisions	universal
Kaufmann, 2009 [14]	CGP	CPU runtime	cache maps
Karasek, 2011 [20]	GE	collisions	universal
Estébanez, 2014 [21]	TGP	avalanche	universal
Grochol, 2016 [3]	LGP	collisions	IPv4 flow hash
Dobai, 2017 [22]	GA-FPGA	collisions	Cuckoo IP hash
Kidon, 2017 [23]	TGP	collisions	Cuckoo IP hash
Grochol, 2017 [4]	LGP+NSGA2	1. collisions 2. exec. time	IPv4 flow hash
Grochol, 2018 [13]	LGP+NSGA2	1. collisions 2. exec. time	universal
Saez, 2019 [12]	TGP	collisions	4 applications

CGP was only used in one of the applications listed in Table I, in particular, to evolve an application-specific hash mapping for a processor cache [14]. CGP, in fact, evolved a combinational gate-level circuit with a 27 bit input and a 15 bit output. However, the quality of hashing was not directly evaluated (for example, by means of the collision counting) as in the other case studies from Table I. The objective was to minimized the execution time for a given application running on the processor utilizing a given candidate hash function in its cache. In our work, CGP is employed to evolve hash functions that are directly evaluated, i.e. they can directly be compared with other hash functions and reused.

C. Evolutionary network flow hashing

In our previous work, single- as well as multi-objective LGP was used to evolve hash functions for IPv4 network flow hashing [3], [4]. Collisions in the hash table were resolved by introducing a linked list for each slot. The input to the hash function was a 5-tuple identifying a network flow by the source and destinations IP addresses (2×32 bits), source and destination ports (2×16 bits) and transport protocol (8 bits). This 104 bit input was read in one step. The hash function was implemented as a sequence of several arithmetic/logic operations without any loop in its body. LGP employed eight 32 bit registers (R0 – R7), 32 bit operators (logical XOR, addition, multiplication and right rotation) over the registers and returned a 16 bit hash (in low half of R0). In order to fit the problem into this LGP setup, the 104 bit input vector was reduced to 3×32 bits only. The source and destination IP addresses remained in the original format, but the source and destination ports (SP, DP) and transport protocol (tp) were compressed into a new 32 bit vector as shown in Fig. 1(A).

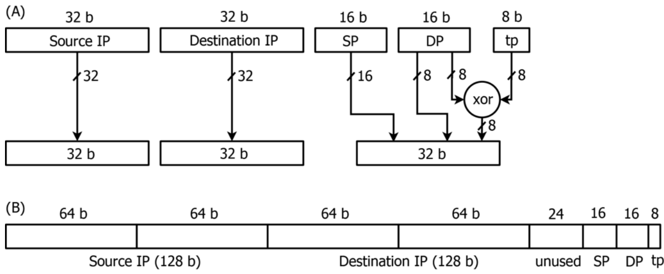


Fig. 1. Creating the input vectors for the hash function from the source IP address, destination IP address, source port (SP), destination port (DP) and transport protocol (tp) in the case of IPv4 flow hashing (A) and IPv6 flow hashing (B).

The three input vectors were stored to R0, R1 and R2. The remaining registers were initialized to 0. The fitness function was based on the weighted number of collisions produced for a training data set. In the multi-objective scenario, the second objective was to minimize the execution time which was estimated as the weighted number of instructions. Detailed comparison of evolved hash functions with state of the art hash functions is given in [3], [4]

III. Multi-objective CGP for Flow Hashing

In order to evolve hash functions for network flow hashing, we propose to combine CGP with NSGA-II.

A. Cartesian Genetic Programming

CGP is typically used to design or optimize digital circuits [24], [14] or solve symbolic regression problems [25].

In CGP, a candidate program is represented in a two-dimensional grid of nodes consisting of n_r rows and n_c columns. A candidate program accepts n_i primary inputs and returns n_o outputs. Every node is configured to perform one of n_a-input functions defined in the function set Γ. As each of the inputs of a node can be connected either to a primary input or to a node placed in previous up to L columns, no feedbacks are allowed in the resulting program, i.e. in the phenotype. A unique address is assigned to every primary input and every node output. By means of these addresses, program's encoding (genotype) is established as seen in Fig. 2. One node is represented in the genotype using n_a+1 integers, where n_a integers specify the connection addresses and one integer encodes the node's function. Finally, n_o integers are employed to define the nodes serving as primary outputs of the program. The genotype size is $n_r \cdot n_c \cdot (n_a+1)+n_o$ integers. This encoding is redundant because some nodes, some of their inputs or some primary inputs need not be used in the phenotype.

The standard single-objective CGP utilizes a search method known as $1+\lambda$ in which a new population consisting of $\lambda+1$ individuals is generated by applying the mutation operator on the best individual of the previous population. Various mutation operators have been proposed in the context of CGP [26]. One of the most popular ones modifies each node

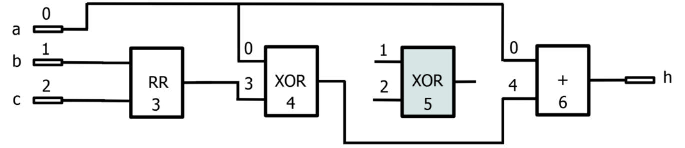


Fig. 2. Hash function $h = (a\ xor\ RR(b)) + a$ represented in CGP with $n_i = 3, n_o = 1, n_r = 1, n_c = 4, n_a = 2$. Node 5 is not used. Genotype: 1, 2, 0; 0, 3, 1; 1, 2, 1; 0, 4, 2; 6. $\Gamma = \{RRotate(0), XOR(1), +(2)\}$

with the probability p_m. If a node has to be mutated then either one of input connections or its function is randomly modified.

B. Evolutionary design of hash functions

As one of our objectives is to compare CGP with LGP in this task, the CGP setup is reflecting the approach described in Section II-C, including the use of NSGA-II as an efficient multi-objective EA for two objectives. CGP is used with one row ($n_r = 1$), $n_c > 1$ and $L = n_c$ which enables all possible connections among the nodes in the resulting directed acyclic graphs (phenotypes). We will consider IPv4 flow hashing (in the same setup as in previous studies [3], [4]) and, as a new contribution of this paper, IPv6 flow hashing.

In the case of IPv4 flow hashing, a candidate hash function will accept three 32 bit vectors and produce a 16 bit vector by xor-ing low 16 bits with high 16 bits of the 32 bit output vector, i.e. $n_i = 3, n_o = 1$ in the CGP context. All primary inputs, internal signals and functions of Γ will be defined on 32 bits. Γ consists of logical XOR, OR, addition, multiplication and 1 bit right rotation. In CGP we will not employ any set of constants because the hash functions obtained using LGP do not usually contain any constants even if a set of constants is defined for LGP.

NSGA-II needs a larger population than the standard CGP (in which λ is usually 4 according to [25]). Hence, the population size α will be at least 10 individuals. We replaced the original (1+λ) mechanism of CGP in such a way that the offspring population consists of α individuals if the population size is α. Each individual from the parental population generates just one offspring by mutation. NSGA-II then creates the new population from α parents and its α offspring. The remaining operations such as the non-dominated sorting and crowding distance calculations are as implemented in the original NSGA-II [27].

The quality of candidate hash function h is measured exactly as in [3], i.e. fitness function f_1 is the weighted number of collisions (with a quadratic penalty for collisions):

$$f_1(h) = \sum_{i=1}^{s} g_i, \text{ where} \quad (1)$$

$$g_i = \begin{cases} 0 & \text{if } K_i \leq 1 \\ \sum_{j=2}^{K_i} j^2 & \text{if } K_i \geq 2, \end{cases} \quad (2)$$

g is an auxiliary function, s is the number of slots and K_i is the number of inputs (keys) mapped by h into i-th memory slot.

The second fitness function f_2 deals with the execution time of a candidate hash function. It is defined as the number of operations along the longest path from the input to the output in the CGP grid. The rationale behind this definition is that the hash function will be executed on a modern processor, where several arithmetic operations are performed in parallel in the so-called Single Instruction Multiple Data (SIMD) scenario, for example, by means of the SSE and AVX instruction set extensions. This definition provides more reliable estimate of the real execution time than another metric – the number of used nodes. It has to be emphasized that the real execution time will be measured for the best evolved hash functions on various data sets (at the end of evolution) and the proposed estimate is used during the evolution only. In the proposed method, the objective is to minimize f_1 as well as f_2.

In the case of IPv6 flow hashing, the input to the hash function is a 5-tuple identifying an IPv6 network flow by the source and destination IP addresses (2×128 bits), source and destination ports (2×16 bits) and transport protocol (8 bits). As Fig. 1(B) shows, this information is provided to CGP via five 64-bit primary inputs and CGP operates with 64 bit vectors. The function set is the same as for IPv4, but all the functions operate over 64 bit operands. The 16 bit hash is then obtained by xor folding of the 64 bit output vector (an example will be given in Section IV-C). Two fitness functions f_1 and f_2 are defined identically to IPv4 hashing, but different training data sets are employed to obtain the fitness score(s).

C. Implementation Details

The implementation was created in C. The CGP-based approach employed the CGP-library [28] and used 32 and 64 bit unsigned integers as the basic data types for all the operations in the phenotype. The implementations of LGP and NSGA-II were developed by the authors.

IV. Results

After introducing the data sets used in our experiments (Section IV-A), the experimental setup and the results obtained from the comparison of LGP and CGP in the design of IPv4 flow hash functions will be presented in Section IV-B. Section IV-C is then devoted to the results obtained from the evolutionary design of IPv6 hash functions using CGP.

A. Data Sets

The data sets utilized for the experiments were collected by a network monitoring device installed in the CESNET network in the Czech Republic.

For IPv4 flow hashing, we use the same data sets as in [3], i.e. 20,000 (DataSet1), 50,000 (DataSet2) and 100,000 (DataSet3) identifiers of network flows. Note that these identifiers of network flows are unique. While DataSet1 is employed in the fitness function, remaining data sets are devoted to testing of evolved hash functions.

For IPv6 hashing, the training data set (ipv6f0) used in the fitness function contains 20,000 flow records with unique source addresses. The remaining six data sets (ipv6f1 – ipv6f6) contain 65536 flow records each. The data sets were collected on the backbone CESNET-ACONET network in January 2018.

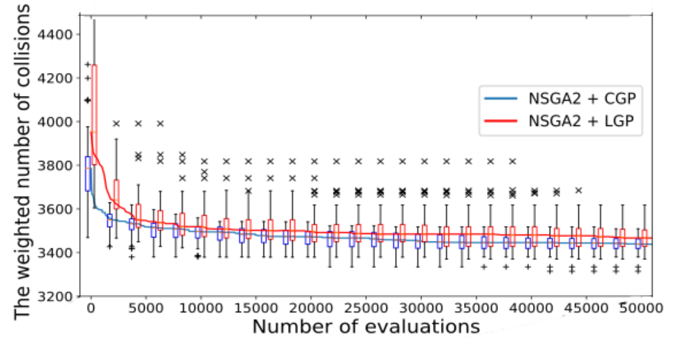


Fig. 3. Box plots showing the weighted number of collisions (f_1) during the course of evolution. They were constructed from 100 independent runs of LGP and CGP in the IPv4 hash function design.

B. Comparison of Multi-objective LGP and CGP for IPv4 Flow Hashing

The baseline setup of LGP (with NSGA-II) is described in Section II-C. LGP employs a 200 member population, produces 1000 generations and new individuals are created using one-point crossover (with probability 0.9), mutation (with probability 0.15) and a 4-member tournament selection. The maximum program length is 20 instructions. This parameter setup was identified as the most useful in [3], [4].

The baseline setup of CGP (with NSGA-II) is as proposed in Section III-B, i.e. $n_i = 3$, $n_o = 1$ $n_r = 1$, $n_c = 20$, $L = n_c$, $p_m = 0.8$ and $\alpha = 10$. The number of generations is 10000 to evaluate the same number of candidate solutions as LGP. All experiments reported in this paper start with a randomly generated initial population.

Fig 3 shows box plots with the weighted number of collisions (f_1) during the course of 100 independent LGP and CGP runs (the best solution of a given generation is considered). On our training data set, the quality of hashing provided by CGP is better than LGP. All obtained trade-offs are depicted in Fig. 4 in which the size of circles represents the number of solutions with particular values (f_1, f_2). It has to be emphasized that the number of computational steps is only estimated and the real execution time can be slightly different.

In order to validate the hash functions evolved with CGP and LGP, we selected some of them from the Pareto front, implemented them in C and compared the exact number of collisions and the execution time with conventional hash functions. For 7 hash functions evolved with LGP (NSGALGPHash1-7), 7 hash functions evolved with CGP (NSGACGPHash1-7) and 11 other hash functions, Tables II and III give the number of collisions and the average execution time needed to process all data sets 20 times on the Intel XEON E5-2620v3 processor. These numbers are translated to three two-dimensional plots constructed for our three data sets (Fig. 5). Clearly weak hash functions are not displayed in the plots. Both CGP and LGP provided many significantly faster hash functions

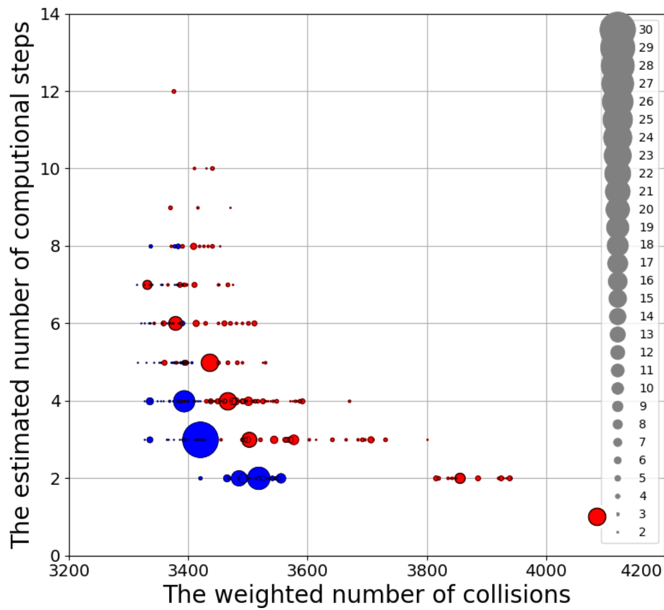


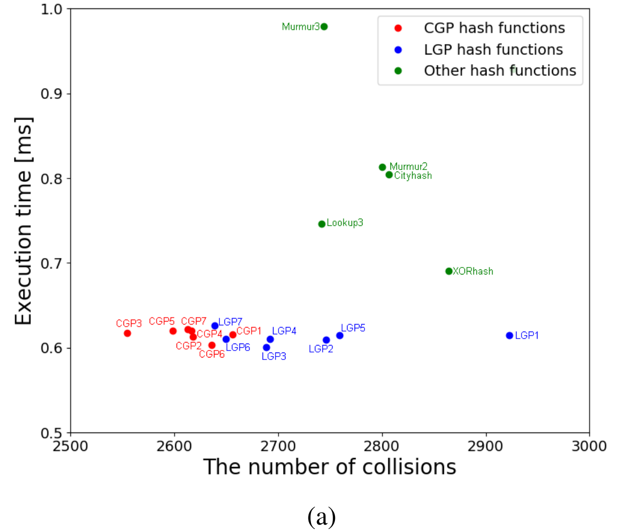
Fig. 4. Fitness values of all hash functions evolved for IPv4 flow hashing and their frequency (shown as the circle size) in 100 CGP (blue) and 100 LGP (red) runs for IPv4 flow hashing.

TABLE II
THE NUMBER OF COLLISIONS FOR COMMON HASH FUNCTIONS AND HASH FUNCTIONS EVOLVED WITH LGP AND CGP (IPV4 FLOW HASHING).

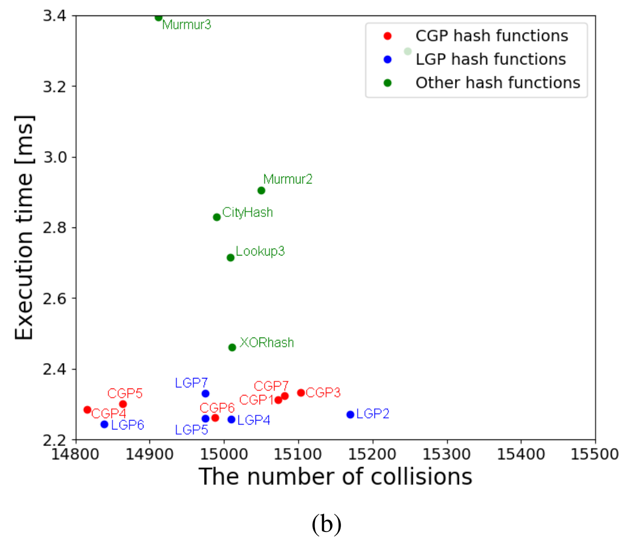
Hash function	The number of collisions		
	DataSet1	DataSet2	DataSet3
DJBHash	2835	15113	48925
DEKHash	2926	15247	49017
FVNHash	2756	14957	48780
One At Time	2821	14988	48636
lookup3	2742	15009	48737
Murmur2	2800	15050	48749
Murmur3	2744	14911	48763
CityHash	2807	14990	48647
GPHash	2777	15052	48750
EFHash	5317	25266	63175
XORHash	2864	15011	48575
NSGALGPHash1	2923	15677	49336
NSGALGPHash2	2746	15170	48835
NSGALGPHash3	2689	15575	49292
NSGALGPHash4	2692	15010	48715
NSGALGPHash5	2759	14975	48749
NSGALGPHash6	2650	14839	48680
NSGALGPHash7	2639	14975	48650
NSGACGPHash1	2656	15073	48607
NSGACGPHash2	2618	15592	49130
NSGACGPHash3	2555	15103	48889
NSGACGPHash4	2617	14816	48624
NSGACGPHash5	2599	14864	48692
NSGACGPHash6	2636	14988	48703
NSGACGPHash7	2613	15082	48642

in comparison with common hash functions. In terms of the quality of hashing, there is only one conventional solution – the XORHash – which is (slightly) better than evolved hash functions on DataSet3.

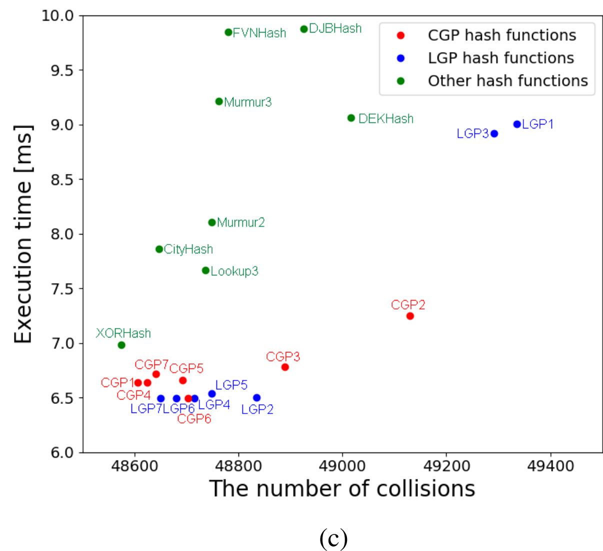
Regarding the parameter setup of LGP and CGP, we were interested in the correct population sizing as the NSGA-II algorithm usually requires larger populations. We fixed the



(a)



(b)



(c)

Fig. 5. Comparison of hash functions for IPv4 flow hashing on (a) DataSet1, (b) DataSet2 and (c) DataSet3.

TABLE III
THE AVERAGE EXECUTION TIME FOR COMMON HASH FUNCTIONS AND HASH FUNCTIONS EVOLVED WITH LGP AND CGP (IPV4 FLOW HASHING).

Hash function	Time $[ms]$		
	DataSet1	DataSet2	DataSet3
DJBHash	1.091	3.661	9.874
DEKHash	0.929	3.298	9.063
FVNHash	1.055	3.631	9.845
One At Time	1.396	4.687	12.494
lookup3	0.746	2.715	7.665
Murmur2	0.813	2.905	8.104
Murmur3	0.979	3.395	9.215
CityHash	0.804	2.829	7.862
GPHash	1.472	4.836	12.784
EFHash	1.927	13.461	50.690
XORHash	0.691	2.462	6.980
NSGALGPHash1	0.615	2.929	9.001
NSGALGPHash2	0.609	2.271	6.500
NSGALGPHash3	0.601	2.938	8.919
NSGALGPHash4	0.610	2.257	6.492
NSGALGPHash5	0.615	2.259	6.537
NSGALGPHash6	0.610	2.243	6.491
NSGALGPHash7	0.626	2.330	6.491
NSGACGPHash1	0.616	2.312	6.636
NSGACGPHash2	0.613	2.537	7.246
NSGACGPHash3	0.617	2.332	6.778
NSGACGPHash4	0.620	2.286	6.639
NSGACGPHash5	0.620	2.302	6.656
NSGACGPHash6	0.603	2.261	6.490
NSGACGPHash7	0.622	2.323	6.713

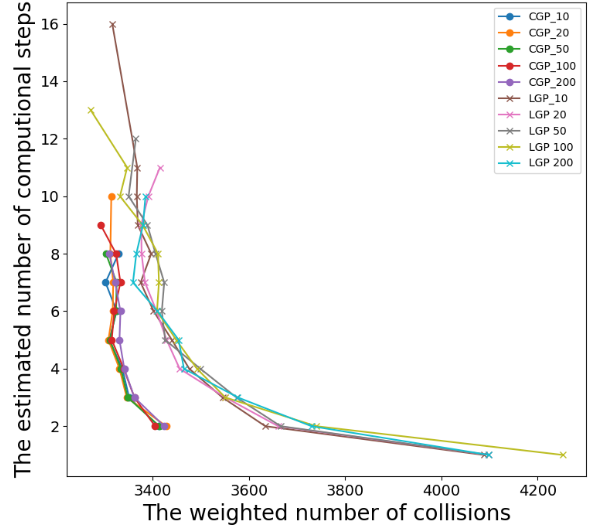


Fig. 6. Trade-offs obtained using CGP and LGP for different population sizes that are shown in the legend. We plot the median of the weighted number of collisions obtained for a given number of computational steps. 200,000 evaluations were performed in each of 100 independent runs.

number of evaluations to 200 thousands and observed the impact of the population sizing on obtained trade-offs. Figure 6 illustrates that the results of CGP as well as LGP are almost insensitive to the population size. Our explanation is that the number of computational steps (f_2) in evolved hash functions is usually very small (between 2 and 10) and, hence, even a small population can lead to good results. This is an important outcome, because the execution time of NSGA-II largely depends on the population size and its good performance even with a small population is a clear advantage.

C. IPv6 Flow Hashing

We have shown that CGP provides very competitive solutions in comparison with LGP for IPv4 flow hashing. Hence, the evolutionary design of hash functions for IPv6 flow hashing is performed with CGP only. In Section III-B we proposed that the hash function for IPv6 hashing will accept five 64 bit vectors and execute 64 bit operations over these inputs. Because the design problem is more complex than in the case of IPv4, we increased the number of nodes to 30 and the population size to 20, i.e. $n_i = 5$, $n_o = 1$ $n_r = 1$, $n_c = 30$, $L = n_c$, $p_m = 0.8$ and $\alpha = 20$. The number of generations is 10000.

One additional constraint has been introduced to CGP. A candidate hash function is accepted only if it uses all five inputs. This constraint should promote more general hash functions over those well-tuned for a particular data set. A side effect of introducing this constraint is that many candidate hash functions are not evaluated (because the constraint is violated) and the total execution time is reduced about 40% on average in comparison with CGP imposing no constraint of this type.

Figure 7 shows the parameters – the number of weighted collisions (f_1) and the estimated number of computational steps (f_2) of all hash functions that we obtained from 100 independent CGP runs. The circle size corresponds with the number of occurrences of the hash function exhibiting particular (f_1, f_2) pair. The solution showing the best trade-off requires 4 computational steps, i.e. there are four layers of nodes between the input and output. However, several interesting solutions need only 3 layers.

We selected four interesting evolved hash functions for a detailed analysis. One of them, IPv6Hash1, is depicted in Fig. 8 and the corresponding C code is given in Fig. 9. In this function, the addition operation is used three times in the same way, i.e. as addition of two identical numbers, which is in fact a 1 bit shift operation. This is probably a good strategy to improve the non-linearity of hashing. Detailed explanation of this phenomenon is left for future research.

Selected conventional hash functions and four evolved hash functions were implemented in C and compiled under the same setup. In Tables IV and V we report the number of collisions and the average execution time needed to process all data sets 20 times. These results are then graphically compared in Fig 10, where the number of collisions is averaged across all data sets. Note that hash functions with highly non-competitive execution times or average number of collisions are omitted from the plot. Evolved hash functions are much faster than conventional hash functions. In terms of the quality of hashing on the test data, the highest-scored function is IPv6Hash1, which is closely followed by Lookup3. Fig. 10 also demonstrates the importance of evaluation on test data as IPv6Hash1 scored worst on the training data.

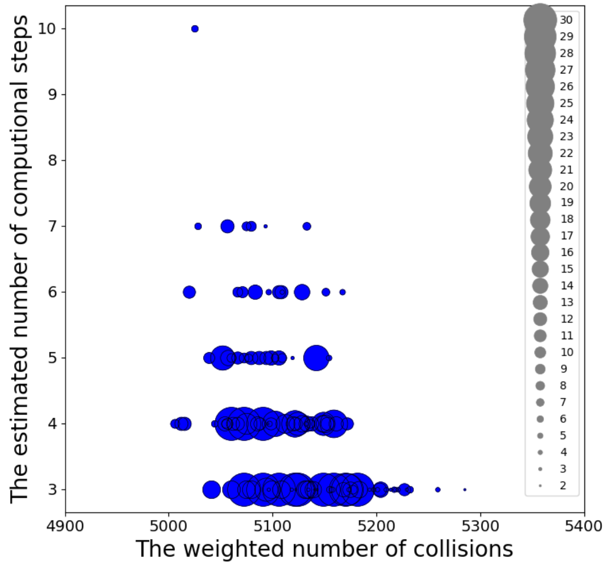


Fig. 7. Fitness values of all evolved hash functions for IPv6 flow hashing and their frequency (shown as the circle size) in 100 CGP runs.

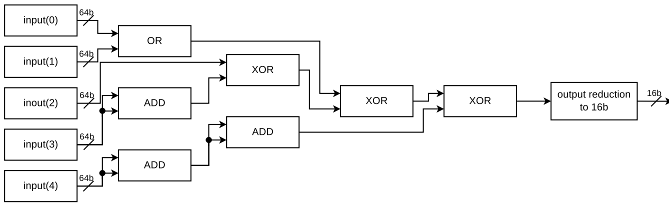
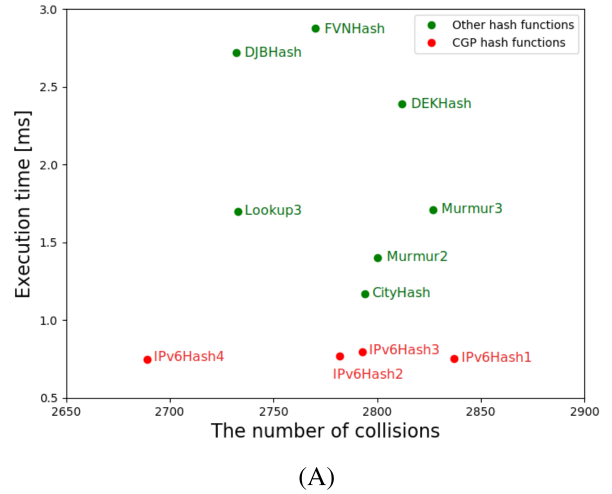


Fig. 8. Example evolved CGP phenotype – IPv6Hash1 hash function

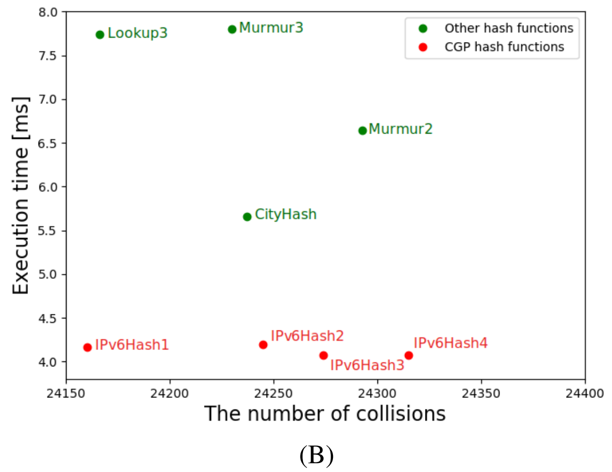
```
unsigned int IPV6Hash1(const char * key, size_t len){
  uint64_t *text = (uint64_t *)key;
  uint64_t v0 = text[0];
  uint64_t v1 = text[1];
  uint64_t v2 = text[2];
  uint64_t v3 = text[3];
  uint64_t v4 = text[4];

  uint64_t v5 = v3 + v3;
  uint64_t v6 = v2 ^ v5;
  uint64_t v7 = v4 + v4;
  uint64_t v8 = v7 + v7;
  uint64_t v9 = v1 | v0;
  uint64_t v10 = v9 ^ v6;
  uint64_t v11 = v8 ^ v10;
  return (uint16_t)(v11 >> 48) ^ (v11 >> 32) ^ (v11 >> 16) ^ v11;
}
```

Fig. 9. C program for evolved hash function IPV6Hash1



(A)



(B)

Fig. 10. The average execution time and the average number of collisions for conventional and evolved hash functions for IPv6 flow hashing on training data (A) and test data (B).

TABLE IV
THE NUMBER OF COLLISIONS FOR CONVENTIONAL AND EVOLVED HASH FUNCTIONS ON 7 DATA SETS (IPV6 FLOW HASHING).

Function	The number of collisions						
	ipv6f0	ipv6f1	ipv6f2	ipv6f3	ipv6f4	ipv6f5	ipv6f6
DJBHash	2732	24859	24247	24472	24634	24205	24198
DEKHash	2812	25574	24476	25618	26496	25133	24468
FVNHash	2770	24896	23968	24215	24320	24068	23973
One At Time	2759	24774	24047	24203	24202	24094	24116
lookup3	2733	24820	24002	24058	24080	24034	24004
Murmur2	2800	24885	24183	24183	24208	24142	24157
Murmur3	2827	24982	24093	23979	24146	24051	24127
CityHash	2794	24923	24150	24025	24118	24129	24074
GPHash	2777	24923	24085	23995	24088	23941	24081
IPV6Hash1	2837	24709	24026	24062	24095	24012	24058
IPV6Hash2	2782	24779	24054	24028	24296	24181	24133
IPV6Hash3	2793	24721	24239	24081	24062	24259	24284
IPV6Hash4	2689	24966	24265	24110	24105	24365	24081

TABLE V
THE AVERAGE EXECUTION TIME ON INTEL XEON E5-2620V3 FOR CONVENTIONAL AND EVOLVED HASH FUNCTIONS (IPV6 FLOW HASHING).

Hash function	Time $[ms]$	
	test sets	training set
DJBHash	10.983	2.719
DEKHash	9.639	2.389
FVNHash	11.825	2.877
One At Time	14.107	3.948
lookup3	7.743	1.699
Murmur2	6.643	1.401
Murmur3	7.803	1.711
CityHash	5.655	1.167
GPHash	18.646	5.408
IPV6Hash1	4.166	0.756
IPV6Hash2	4.199	0.767
IPV6Hash3	4.072	0.795
IPV6Hash4	4.069	0.746

V. CONCLUSIONS

We proposed an evolutionary design system based on CGP and NSGA-II which is capable of providing high quality hash functions in terms of the quality of network IPv4 and IPv6 flow hashing and the execution time of the resulting hash function. We showed that our CGP-based method provides competitive results in comparison with the former approach based on LGP. Evolved hash functions are the first hash functions solely devoted to IPv6 flow hashing. They provide similar quality of hashing as the state of the art hash functions, but reduce the execution time of the hashing operation which is essential in high speed computer networks.

Our future work will be devoted to detailed analysis of evolved IPv6 hash functions and improving the proposed search algorithm.

ACKNOWLEDGMENTS

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science – LQ1602.

REFERENCES

[1] L. Kekely, J. Kucera, V. Pus, J. Korenek, and A. Vasilakos, "Software defined monitoring of application protocols," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 615–626, 2016.

[2] Z. Cao and Z. Wang, "Flow identification for supporting per-flow queueing," in *Proc. of the 9th Int. Conf. on Computer Communications and Networks*. IEEE, 2000, pp. 88–93.

[3] D. Grochol and L. Sekanina, "Evolutionary design of fast high-quality hash functions for network applications," in *Proc. of the 2016 Genetic and Evolutionary Computation Conference*. ACM, 2016, pp. 901–908.

[4] ——, "Multiobjective evolution of hash functions for high speed networks," in *IEEE Congress on Evolutionary Computation*. IEEE, 2017, pp. 1533–1540.

[5] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching, 2nd ed.* Upper Saddle River : Addison-Wesley, 1998.

[6] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms - ESA 2001*, ser. LNCS 2161. Springer, 2001, pp. 121–133.

[7] D. J. Bernstein, "Mathematics and computer science," https://cr.yp.to/djb.html.

[8] G. Fowler, P. Vo, and L. C. Noll, "FVN Hash," http://www.isthe.com/chongo/tech/comp/fnv/.

[9] B. Jenkins, "A hash function for hash table lookup," http://www.burtleburtle.net/bob/hash/doobs.html.

[10] A. Appleby, "Murmur hash functions," https://github.com/aappleby/smhasher.

[11] G. Pike and J. Alakuijala, "Introducing CityHash," 2011.

[12] Y. Saez, C. Estebanez, D. Quintana, and P. Isasi, "Evolutionary hash functions for specific domains," *Applied Soft Computing*, vol. 78, pp. 58 – 69, 2019.

[13] D. Grochol and L. Sekanina, "Multi-objective evolution of ultra-fast general-purpose hash functions," in *European Conference on Genetic Programming*, ser. LNCS, vol. 10781. Springer, 2018, pp. 187–202.

[14] P. Kaufmann, C. Plessl, and M. Platzner, "EvoCaches: Application-specific Adaptation of Cache Mappings," in *Adaptive Hardware and Systems (AHS)*. IEEE CS, 2009, pp. 11–18.

[15] E. Damiani, V. Liberali, and A. Tettamanzi, "Evolutionary design of hashing function circuits using an FPGA," in *Evolvable Systems: From Biology to Hardware*, ser. LNCS, vol. 1478. Springer, 1998, pp. 36–46.

[16] P. Berarducci, D. Jordan, D. Martin, and J. Seitzer, "GEVOSH: Using grammatical evolution to generate hashing functions," in *MAICS*. Omnipress, 2004, pp. 31–39.

[17] C. Estébanez, J. C. Hernández-Castro, A. Ribagorda, and P. Isasi, "Finding state-of-the-art non-cryptographic hashes with genetic programming," in *Parallel Problem Solving from Nature – PPSN IX*. Springer, 2006, pp. 818–827.

[18] H. Widiger, R. Salomon, and D. Timmermann, "Packet classification with evolvable hardware hash functions–an intrinsic approach," in *International Workshop on Biologically Inspired Approaches to Advanced Information Technology*. Springer, 2006, pp. 64–79.

[19] M. Safdari and R. Joshi, "Evolving universal hash functions using genetic algorithms," in *Proc. of the 2009 Int. Conf. on Future Computer and Communication*. IEEE, 2009, pp. 84–87.

[20] J. Karasek, R. Burget, and O. Morsky, "Towards an automatic design of non-cryptographic hash function," in *Telecommunications and Signal Processing (TSP), 2011 34th International Conference on*. IEEE, 2011, pp. 19–23.

[21] C. Estebanez, Y. Saez, G. Recio, and P. Isasi, "Automatic design of noncryptographic hash functions using genetic programming," *Computational Intelligence*, vol. 30, no. 4, pp. 798–831, 2014.

[22] R. Dobai, J. Korenek, and L. Sekanina, "Evolutionary design of hash function pairs for network filters," *Applied Soft Computing*, vol. 56, no. 7, pp. 173–181, 2017.

[23] M. Kidon and R. Dobai, "Evolutionary design of hash functions for IP address hashing using genetic programming," in *IEEE Congress on Evolutionary Computation*. IEEE, 2017, pp. 1720–1727.

[24] L. Sekanina and Z. Vasicek, "Approximate circuit design by means of evolvable hardware," in *2013 IEEE International Conference on Evolvable Systems (ICES-SSCI)*. IEEE, 2013, pp. 21–28.

[25] J. F. Miller, *Cartesian Genetic Programming*. Springer-Verlag, 2011.

[26] B. W. Goldman and W. F. Punch, "Analysis of cartesian genetic programmings evolutionary mechanisms," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 359–373, 2015.

[27] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[28] A. J. Turner and J. F. Miller, "Introducing a cross platform open source cartesian genetic programming library," *Genetic Programming and Evolvable Machines*, vol. 16, no. 1, p. 8391, 2015.