# Hash Bash: Genetic Algorithms for Hash Function Generation

Vaibhav Mallya

December 4, 2008

## 1 Abstract

This paper explores the use of genetic algorithms for the generation of hash functions. We define a custom Turing-complete assembly language to represent our generated functions. We evaluate fitness primarily from collision-resistance. We can test uniform output distribution only intermittently due to the length and expense of the randomness tests. Speed is guaranteed by a hard maximum cycle limit. Our results are limited, but we believe the approach is promising.

## 2 Introduction

Hash functions are key building blocks of modern cryptography. They have applications in encryption algorithms, authentication protocols, and integrity verification schemes [4]. Today's most widely-used hash functions are inspired by Rivest's MD2 [13] and MD4, the designs of which are characterized by input chunking, substitution, and the repeated application of overlapping bitwise operations. Although there numerous other hashing schemes, they are either not as widely-used, or are not as effective, and therefore, not a focus of this paper. For the purposes of this paper, we will refer to all functions structurally related to the MDs and the SHAs as Rivest functions.

Although they are important to the soundness of many security schemes, most common modern hash functions, including SHA-1 and MD5, have either had serious attacks mounted against them or are in danger of being vulnerable in the near future. MD5 in particular can now be broken in under a minute on a commodity machine. [8] As a result, an NIST competition has been initiated for the creation of a new cryptographically secure function [2]. Unfortunately, hash function creation is an extremely difficult task, and vetting and verification takes a great deal of careful analysis. Regardless, we show that it may be possible to approximate a good hash function through the use of the genetic algorithm.

The genetic algorithm is inspired by biological natural selection. John Holland of the University of Michigan pioneered its use in the 1970s [6]. However, it wasn't until the 1980s and 1990s that the computational capacity for practical implementations of the algorithm became readily available [5]. The algorithm as we use it consists of a series of functions - a randomized generator, a fitness evaluator, and a breeder - that operate on chromosomes, which are strings formed from a genetic description alphabet. The algorithm proceeds as follows:

1. Create a random starting generation of size S.
2. Evaluate the fitness of every member of the current generation.
3. Breed the fittest members to form the next generation.
4. Insert new or old chromosomes into the next generation until it is of size S.
5. Randomly mutate some member(s) of the generation.
6. Go to step 2 if we're not done.

With proper parameter selection and a sufficient amount of time, the genetic algorithm will tend to converge to some local optimum. Since we are combining parts of the best solutions to form new solutions, it seems reasonable that this should happen. Holland's Schemata Theorem provides a more rigid explanation as to how this occurs [7].

# 3 Language Description

Since we are creating, breeding, and interpreting programs, we first need a language that meets the following criteria:

1. Readable enough to manually program and interpret.
2. Powerful enough to allow a wide range of hash functions to be efficiently and compactly represented.
3. Compact enough that there are few-to-none overlapping or redundant operators.
4. Flexible enough that syntax errors are impossible.
5. Ability to implement all Rivest functions.
6. Represented in a way that allows the breeding of two functions to yield a better function with high probability.

An assembly language is an ideal candidate here. It's straightforward how machine code represents instructions, bitstrings are easy to manipulate, and disassembly is easy to interpret. Additionally, interpreters are easy to find or even write. Although there are a number of special-purpose assembly languages that fulfill one or two of these criteria, we were unable to find any that fulfilled all of them. We created our own assembly language with the following characteristics:

- No registers - all operators work directly on memory.
  - Somewhat impractical to implement in hardware, but this is not a design goal.
- $2^{15}$ words of memory, each 68 bits large. By providing more space, we hope to provide more effectiveness.
- 15 operators, each with four 16-bit operands.
  - Includes N/AND, X/OR, NOT, conditional sets, and conditional adds (to be used for branching)
  - Every instruction does not utilize every operand.
  - Leading bit of every operand indicates if the remaining 15 bits refer to an immediate or an address.
- Program counter stored in a fixed location in memory, allowing the program to modify it (i.e. branching).
- A special operator **iterinput** that fetches the next input chunk and places it in a fixed memory location.
- Another fixed range of memory that is treated as an output buffer. When the program halts, the buffer is flushed and the bits in the output buffer at that time are viewed as the program's output.

Although the algorithm itself views this language in terms of its machine code, the human-readable disassembly is straightforward, since instructions take the form **[dest] [op] [op0] [op2]**. The machine code itself is just a series of binary strings. This makes programs trivial to:

- Generate: Given a template with gaps, randomly generate bitstrings of length 68 to fill the gaps.
- Breed: Concatenate the parent chromosomes.
  - Hash functions concatenated should yield better hash functions, so logically, this is a good strategy.
  - If our child function is too large, we remove words from the middle.

- Mutate: AND some random bitstring(s) of length 68 with some random word(s).
  - Allow a small number of additional words to be created with some small probability.

Finally, it's obvious that we can duplicate all Rivest functions in a straightforward way - we have access to all the bitwise operators used, as well as simple chunking and branching.

# 4 Implementation Details and the Fitness Function

The genetic algorithm and custom language simulator are implemented completely in Python, with the exception of the Diehard tests [10] which are written in C. There are four primary utility scripts: generate.py, fitness.py, simulate.py, and breed.py. Each has a corresponding parameters file for simple fine-tuning. The script that ties all of these together to compute current and successive generations is master.py, which has its own parameter file. Additionally, we add initial entropy to all hash functions through the use of a Munroe random number [11]. We initialize certain fixed ranges in memory to these numbers for potential use by a hash function.

We make extensive use of Psyco [12] and the processing module [1]. Psyco improves individual script execution by utilizing a JIT compiler, while the processing module allows for quick and efficient parallelization of the algorithm itself. The primary performance bottleneck, the fitness testing, is completely parallelized, with a pool of processes that greedily grab functions to simulate and evaluate. Breeding is less of a performance concern, but it is also parallelized due to the ease with which it is possible [9]. There are numerous additional opportunities for algorithmic parallelization that aren't utilized by our implementation.

The fitness function consists of two main parts. First, it attempts to filter out trivially bad functions by hashing the elements in [-10, 10]. If the function fails this trivial test by colliding at all, it is flagged. No further tests are conducted on it. Next, it iterates over many random numbers, some of them very close to each other, to determine collision rates. The fitness function writes the ratio (collisions $\div$ total inputs), and then terminates. It is important to note that the Diehard [10] randomness and uniformity tests are not conducted programmatically due to their cost. Over 68 million bytes are required for a single hash function's test. The intended procedure for randomness testing is as follows:

1. Manually create the named lockfile in the computational results directory to pause the algorithm.

2. Initiate the randomness tests by hand.

3. Manually append the results of the test to the fitness file of the relevant function.

4. The results will then be accounted for in the following generation's selection and breeding process.

As a result of the randomness testing expense, we base our results primarily on the collision rates of the generated functions.

# 5 Computational Results

Unfortunately, the hash functions the algorithm currently generates are very weak, and would at best serve as CRCs or checksums of some sort. In all runs, we detected only minor differences between hash values for lexicographically adjacent inputs. There were also a distressingly high number of overall functions ($\geq 15\%$) that produced only constants. Upon reexamination, we found several design decisions that seem to have contributed to the exceptionally poor quality of generated functions.

## 5.1 Too many conditional operators

Out of 16 instructions, there are 3 conditional sets and 3 conditional adds. These were intended to be used for branching, so one of the generators actually creates branches with an overly high probability. This obviously excludes bitwise operators that are supposed to do the bulk of the computation from appearing as much as would be ideal.

## 5.2 Poor generator function

There are in fact four structured generator functions provided, but only one of them is actually used most of the time, since the others never produce anything but constants. The single usable generator is itself a manually-created and very trivial hash function. It simply returns some number of bits from the first block of the input. For small inputs, it is a kind of perfect hash function, and obviously produces collisions for longer inputs. Originally, we had intended to create a plethora of generators and randomly choose one when generating a new cell, but this was scrapped due to time constraints.

## 5.3 Too-large word size; too many words

These are related problems; the 68-bit word size is a by-product of wanting to address $2^{15}$ words of memory with three operands. In retrospect, this was a terrible decision, since there is never a need to utilize anything beyond the first 2000 words or so for the program and all temporary variables it could reasonably

use. 2000 itself is a very generous upper bound. By providing so much space, we provide the algorithm with a lot of room to do a lot of nothing.

### 5.4 Turing-completeness

Turing completeness grants the power to solve more difficult problems. But in the context of the genetic algorithm, it also leaves significant opportunities to solve even simple problems incorrectly. More power requires more responsibility, and the genetic algorithm is not responsible at all. As implemented in most languages, the Rivest functions' operators include looping and conditionals. However, when we look at diagrams of the functions themselves, we notice the information stream always "flows down". There is no backwards flow [3]. We can therefore implement Rivest functions with a conditional relative branch whose offset is always positive. There is no need to have a negative branch value. There will be an obscene amount of code repetition with this design, since we'll have to duplicate the code for every round over and over, but it's a computationally valid way of approaching the problem. By eliminating language features that allow Turing completeness, we can simply fitness testing and generation without sacrificing functionality of the hash function. Overall, it should eliminate unfit members much faster. To take advantage of this, we would have to make the following changes to our language:

1. Remove the program counter from the unified memory, so it is not open to arbitrary modification.
2. Add a new branch-on-equality operator with three operands.
3. Make this operator increment the program counter by $|\mathbf{op3}|$ if ($\mathbf{op1} == \mathbf{op2}$)

We feel the question of not needing a Turing-complete language to implement a Rivest function is an interesting one that merits further analysis.

## 6 Further Thoughts and Conclusions

Despite all of this, we believe this approach holds promise, and given more time, we feel we can produce more substantial and interesting results. The justification for our optimism goes back to our generator function. As mentioned, it's a trivial hash function that just moves some subset of the input buffer into the output buffer. Indeed, on first glance, it would seem like a poor candidate for an initial function. However, we conducted several runs ($\approx 50$ generations) with the initial population consisting entirely of this 5-line function. At the end of each of these runs, we noticed that there were two or three long functions (>300 lines) whose exact hash outputs varied slightly from their initial forms. The variation wasn't significant, but the fact that phenotypic variation existed at all is a hopeful sign. It stands to reason that if provided additional trivial hash functions that collectively exercised the full power of the assembly language, the generator would help the algorithm to produce substantially more interesting input.

Also of note is the fact that the resources available for the purposes of this paper were limited. We only had part-time access to a single dual-core machine that was frequently used for other tasks, and only one (very time-strapped) developer. With the injection of additional human resources, the algorithm could be adapted to run on a cluster, the cloud, etc with the modifications as outlined above implemented. Although the runs conducted with this iteration produced limited results, fixing the deficiencies as outlined above would definitely allow for more rapid convergence on interesting behavior.

# References


[1] http://pypi.python.org/pypi/processing.

[2] Cryptographic hash project, April 2005. http://csrc.nist.gov/groups/ST/hash/index.html.

[3] Nathaniel Alderson. Md5 demo. http://nsfsecurity.pr.erau.edu/crypto/md5.html.

[4] Bram Cohen. Bittorrent specification. http://wiki.theory.org/BitTorrentSpecification.

[5] David E. Goldberg. Addison-Wesley Professional, title = Genetic Algorithms in Search, Optimization, and Machine Learning, note = http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0201157675, year = 1989,, January.

[6] J. H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM J. Comput.*, 2:88–105, 1973.

[7] John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992. http://portal.acm.org/citation.cfm?id=129194.

[8] Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding x.509 certificates. Cryptology ePrint Archive, Report 2005/067, 2005. http://eprint.iacr.org/.

[9] Vaibhav Mallya. Github hashbash repository, December 2008. http://github.com/mallyvai/hashbash/.

[10] George Marsaglia, 1995. http://www.stat.fsu.edu/pub/diehard/.

[11] Randall Munroe. Random number. http://xkcd.org/221/.

[12] Armin Rigo. http://psyco.sourceforge.net/.

[13] B. Kaliski Ron Rivest, John Linn. The md2 message-digest algorithm. http://www.faqs.org/rfcs/rfc1319.html.