Hash Bash: Genetic Algorithms for Hash Function Generation

Vaibhav Mallya

December 3, 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, having applications in encryption, authentication protocols, and integrity verification schemes (http://wiki.theory.org/BitTorrentSpecification). Today's most widely-used hash functions are inspired by Rivest's MD2 (http://tools.ietf.org/html/rfc1319) and MD4, the designs of which are characterized by input chunking, substitution, and the repeated application of overlapping bitwise operations.

Although they are important to the integrity of many security schemes, most modern commonly-used hash functions, including SHA-1 and MD5 (hereafter referred to as Rivest functions), 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. (http://eprint.iacr.org/2005/067) As a result, an NIST competition has been initiated for the creation of a new cryptopgraphically secure function (http://csrc.nist.gov/groups/ST/hash/index.html). Unfortunately, hash function creation is an extremely difficult task, and vetting and verification takes a great deal of careful analysis. However, we show that it is possible to approximate a good hash function through the use of genetic algorithm.

The genetic algorithm is inspired by biological natural selection. John Holland of the University of Michigan pioneered its use in the 1970s (Holland, Genetic algorithms and the optimal allocation of trials. SIAM J. Comput., 2.88-105); however, it wasn't until the 1980s and 1990s that the computational capacity for practical implementations of the algorithm became readily available. The algorithm as we use it consists of a series of functions - a randomized generator, a fitness evaluator, and a breeder - that operate on chromomsomes, which are strings formed from a genetic description alphabet. The algorithm proceeds as following:

1. Create a random starting population 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. Add random or old members until we have the next generation is once more of size S.
5. Go to step 2 if we're not done.

With proper parameter selection and a sufficient amount of time, the genetic algorithm will converge to some local optimum. Holland's Schemata Theorem provides a more rigid explanation as to why this occurs (Holland, Adaptation in Natural and Artificial Systems, 1975).

3 Implementation Description

Since we are creating, breeding, and interpreting programs, we first choose a representation 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

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.
    - Clearly impractical to implement in hardware, but this is not a design goal.

- 2^{15} words, each 68 bits large.
- 15 operators, each allowing four 16-bit operands.
  - Every instruction does not utilize every operand.
  - The MSB 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.
- A fixed section of memory treated as an output buffer.

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] add 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: Simply join subsets of the parent chromsomes.
- Mutate: AND some random 68-bit number(s) with some random word(s).

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

The genetic algorithm and custom simulator are implemented completely in Python, with the exception of the Diehard tests (http://www.stat.fsu.edu/pub/diehard/) which are written in C. There are four primary utility scripts: generate.py, fitness.py, simulate.py, 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 parameters file.

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. There are numerous additional opportunities for algorithmic parallelization that aren't utilized by our implementation.

Special mention must be given to Psyco (http://psyco.sourceforge.net/) and the processing module (http://pypi.python.org/pypi/processing/). Psyco improves individual script execution by utilizing a JIT compiler, while the processing module allows for quick and efficient parallelization of the algorithm itself.

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. However, we believe this machine learning approach holds promise, and given more time, we feel we can produce more substantial and interesting results. There are two primary reasons. First, the randomized generator function is weak. There are in fact four structured generator functions provided, but only one of them is actually used, 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 (http://burtleburtle.net/bob/hash/perfect.html), and obviously produces collisions for longer inputs.

Indeed, on first glance, it would seem like a poor candidate for an initial function. However, we conducted several runs (50 generations) with the initial population consisting entirely of this function, which is 5 lines long. 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 allow the algorithm to produce substantially more interesting input.

Secondly, the resources available for the purposes of this paper was limited. We had full-time access to a single dual-core machine that was frequently used for other tasks, and only one time-strapped developer. If the algorithm could be adapted to run on a cluster or in the cloud with the backing of additional human resources, it would allow much more rapid convergence on interesting behavior upon execution.