PSIpress
perfscipress.com
15 nov 09: A

Cryptographic hash functions based on ALife

Mark A. Bedau[1]*, Richard Crandall[2], and Michael J. Raven[3]

[1] Department of Philosophy, Reed College, Portland OR USA,
[2] Center for Advanced Computation, Reed College, Portland OR USA,
[3] Department of Philosophy, New York University, New York NY USA
* Corresponding author (Email: mab@reed.edu).

Abstract. There is a long history of cryptographic hash functions, i.e. functions mapping variable-length strings to fixed-length strings, and such functions are also expected to enjoy certain security properties. Hash functions can be effected via modular arithmetic, permutation-based schemes, chaotic mixing, and so on. Herein we introduce the notion of an artificial-life (ALife) hash function (ALHF), whereby the requisite mixing action of a good hash function is accomplished via ALife rules that give rise to complex evolution of a given system. Various security tests have been run, and the results reported for examples of ALHFs.

1 Brief history of hash function design

A hash function H maps arbitrary messages (bitstrings) called *keys* or *pre-images* into fixed-length bitstrings called *hash values* (the definitive treatment of hash functions is [Knuth 1998]).

By the nomenclature

$$H(\kappa) = \nu$$

we mean that κ has some number of bits, say m bits, and ν has n bits. Usually, $m > n$, so that hash values are "compressions" of the corresponding keys. We, however, do not assume that $m > n$. The notion of arbitrary message bitlength for κ is—if one so desires—easily reduced to the more convenient notion of m, n, via the simple observation that a long message may be split up into blocks of m bits each, with one block possibly zero-padded, and so on. References for descriptions of hash function characteristics, thorough hash function nomenclature, and analysis are [Merkle 1979], [Preneel 1993], and [Preneel 1994].

Various modern hash functions in actual use have somewhat arbitrary foundations, with rigorous security analysis almost always nontrivial. In cryptography, a hash function might *appear* to be cleverly constructed to "mix up" the m bits to render a smaller number n of bits, and yet there are often various security weaknesses of which a good hash function should be devoid. For example, the hash function defined

$$H(\kappa) = \left(\prod_j p_j(\kappa) \right) \bmod 65536$$

Mark A. Bedau, Richard Crandall, and Michael J. Raven, "Cryptographic hash functions based on ALife," PSIpress, 15 nov 09: A, http://www.perfscipress.com/papers/crypto9_psipress.pdf

where $p_j(x)$ is the position of the j-th "1" in x, is always a 16-bit value because of the mod, yet this H is terribly insecure. Imagine a password file having one 16-bit hash value for each of some 60000 users (each user's typed password is some κ, and gets mapped), whence many of the hash values "collide"—in fact, a long enough typed password is likely to have enough even positions of its 1's to render the hash value 0! For an informal overview of the main kinds of modern hash functions, see [Schneier 1996]; a more rigorous overview can be found in [Menezes *et al.* 1997].

We authors are of the belief that statistical tests, such as the avalanche test (see [Feistel 1973] and [Menezes *et al.* 1997]) and the collision test (see [Menezes *et al.* 1997], [Schneier 1996], and [Stinson 1995]), are the best approach for assessing hash function security. In spite of this, we do appreciate the conceptual and aesthetic approaches to hash functions. In the present treatment, we attempt to spring from the intuitive understanding that evolving systems can be rife with complexity, at many levels, and use such intuition to create hash functions. To this end, we adapt evolving computational systems as studied in Artificial Life to create an artificial-life hash function (ALHF), whereby the requisite mixing action of a good hash function is accomplished via the sort of rules that give rise to complex evolution in standard Artificial Life systems.

2 Cryptographic security tests for hash functions

There are several typical ways to test the security of cryptographic hash functions. We tested the security of the ALHF against the so-called *Avalanche Test* and *Collision Test*. *Avalanche Test.* Consider a hash function H operating on m-bit input keys, $\kappa_1, \kappa_2, ...$, and producing n-bit hash values, $\nu_1, \nu_2, ...$ Let $H(\kappa_1) = \nu_1$ and $H(\kappa_2) = \nu_2$, where $\kappa_1 \neq \kappa_2$ and $\nu_1 \neq \nu_2$. One way in which H can be insecure is if small changes in κ_1 and κ_2 result in predictable changes in ν_1 and ν_2. First, let $\mid \nu \mid$ be the bit-length of hash value ν. Let

$$\delta_{input}(\kappa_1, \kappa_2) = \frac{\kappa_1 \oplus \kappa_2}{\mid H(\kappa_1) \mid}$$

and let

$$\delta_{output}(\nu_1, \nu_2) = \frac{\nu_1 \oplus \nu_2}{\mid H(\nu_1) \mid}.$$

(We assume that $\mid H(\kappa_1) \mid = \mid H(\kappa_2) \mid$ and that $\mid H(\nu_1) \mid = \mid H(\nu_2) \mid$.) One could predict ν_1 and ν_2 from κ_1 and κ_2 if both $\delta_{input}(\kappa_1, \kappa_2)$ and $\delta_{output}(\nu_1, \nu_2)$ are small. The hardest case to avoid is where $\delta_{input}(\kappa_1, \kappa_2) = \frac{1}{m}$. In this case, one wants $\delta_{output}(\nu_1, \nu_2) = \frac{1}{2}$ (this roughly amounts to satisfying the *strict avalanche criterion* introduced and discussed in [Webster 1985] and [Webster and Tavares 1986], according to which if a single bit in the key is complemented, then there is a one half probability that the hash value is complemented). If this can be achieved, the cryptographic hash function achieves *avalanche* and the Avalanche Test is passed. See [Feistel 1973] for an early description of avalanches in cryptography. For recent developments see [Seberry *et al.* 1994], [Seberry *et al.* 1995],

and [Zhang and Zheng 1995]. *Collision Test.* A *collision* is said to occur when two distinct input keys κ_1 and κ_2 are such that $H(\kappa_1) = H(\kappa_2)$. Any secure cryptographic hash function should rarely produce collisions. One good way of testing how well a given cryptographic hash function performs with respect to collisions is by means of the so-called *Birthday Attack.* 253 people must be in the same room as you if the probability that someone in the room shares your birthday is greater than chance. What is prima facie puzzling is that there must only be 23 people in the same room for the probability that any two of them share birthdays to be greater than chance. The cryptographic application of the Birthday Attack is clear: it is much easier to find two m-bit random input keys κ_1 and κ_2 such that $H(\kappa_1) = H(\kappa_2)$ (where $H(x)$ is n-bits long) than it is to find a κ_2 such that $H(\kappa_1) = H(\kappa_2)$ *given* a κ_1. Given a κ_1, computing hash values for 2^n random κ_2's is necessary to get a match. However, if one only wishes to find any two matching random κ_1's and κ_2's, then computing hash values for only $2^{n/2}$ random κ_1's and κ_2's are necessary. See [Schneier 1996] and [Stinson 1995] for more on the Birthday Attack; [van Oorschot and Wiener 1994] and [Yuval 1979] discuss how to carry out a Birthday Attack. A secure cryptographic hash function should perform accordingly. The Birthday Attack Test is passed if the cryptographic hash function does not yield any matching hash values for less than $2^{n/2}$ random input keys, and after then, yields them as statistically expected. This is generally achieved if the bit length of the ciphertext is sufficiently long; see [Beth *et al.* 1992] for estimates on appropriate lengths for various tasks.

3 Artificial life—background

Artificial life (also known as "ALife) is an interdisciplinary study of life and life-like processes that uses a synthetic methodology. Artificial life has three broad branches, corresponding to three different synthetic methods. "Soft artificial life creates simulations or other computational systems that exhibit life-like behavior, "hard artificial life produces hardware implementations of life-like systems, and "wet artificial life synthesized living systems in biochemical media. The general goals of artificial life include understanding and creating life and life-like systems, and developing practical devices inspired by living systems. The main open questions in artificial life involve determining how life arises from non-life, determining the potentials and limits of living systems, and determining how life is connected to mind, machines, and culture [Bedau et al. 2000].

The American computer scientist Christopher Langton coined the phrase "artificial life in 1987, when he organized the first scientific conference explicitly devoted to this field [Langton 1989]. Before there were artificial life conferences, the simulation and synthesis of life-like systems occurred in isolated pockets scattered across a variety of disciplines. The Hungarian-born mathematician and physicist John von Neumann created the first artificial life model (without referring to it as such) in the 1940s when he produced a self-reproducing, computation-universal entity using cellular automata [von Neumann 1966]. Rather than modeling some existing living system, many artificial life systems are in-

tended to generate wholly newand typically extremely simpleinstances of life-like phenomena. The simplest example of such a system is the so-called “Game of Life devised by the British mathematician John Conway [Berlekamp et al. 1982] in the 1960s, before the field of artificial life was conceived.

Perhaps the most famous recent artificial life system Tierra, designed by the American biologist Tom Ray [Ray 1992]. Tierra consists of a population of self-replicating computer programs populating computer memory and consuming CPU time. The system is initialized when a single (human-designed) self-replicating program, the ancestor, is placed in computer memory and left alone to self- replicate. The ancestor and its descendants repeatedly replicate until memory is teeming with self-replicating programs. Errors (mutations) sometimes occur, so the population of Tierra program evolves by natural selection. If a mutation allows a program to replicate faster, that type of program tends to spread through the population. Over time, the ecology of Tierran programs becomes remarkably diverse. Quickly reproducing parasites that exploit a hosts genetic code evolve, and this spurs the evolution of new programs that resist the parasites. After millions of CPU cycles, Tierra typically contains many kinds of programs exhibiting a variety of competitive and cooperative ecological relationships.

Artificial life is similar to artificial intelligence (AI), both because study natural phenomena through computational models and because natural intelligent and living systems tend to coincide. Despite these similarities, AI and artificial life typically employ different modeling strategies. In most traditional artificial intelligence systems, events occur one by one (serially). A complicated, centralized controller makes decisions based on global information about all aspects of the system, and the controllers decisions have the potential to affect directly any aspect of the whole system. This centralized, top-down architecture is quite unlike the structure of many natural living systems that exhibit complex autonomous behavior. Such systems are often parallel, distributed networks of relatively simple low-level “agents, and they all simultaneously interact with each other. Each agents decisions are based on information about only its own local situation, and its decisions directly affect only its own local situation. In similar fashion, artificial life characteristically constructs massively parallel, bottom-up-specified systems of simple local agents. One repeatedly iterates the simultaneous low-level interactions among the agents, and then observes what aggregate behavior emerges. The working hypothesis of artificial life is that this kind of bottom-up architecture with a population of autonomous agents that follow simple local rules is the only plausible method to synthesize the complex adaptive behavior characteristic of living systems. A specific example of an ALife model is the Generic Neutral Model (GNM) devised by Bedau for normalizing evolutionary activity statistics [Rechtsteiner and Bedau 1999a,b]. The GNM is a generic model of neutral genotype evolution. It consists of a population of individuals that reproduce and die in a fixed genotype space. The genotype space is defined by some number of loci at each of which some number of alleles are segregating. Parameters that need to be specified in the GNM are N, the size of

the population of individuals, r, the reproduction rate (the number of individuals that die and reproduce per time step), l, the number of loci, a, the number of possible alleles per locus, m_l, the probability that the allele at a given locus will be mutated when an individual is born. (The probability that an offspring will have mutation somewhere in its genome, i.e., the mutation rate per individual is $m_i = 1 - (1 - m_l)^l$.) The parameters together determine the model's generic behavior. The genotype space is a hypercube of dimension l and size a^l (number of possible genotypes), with each location in this space corresponding to a given genotype. The current state of the model is given by the distribution of N individuals in genotype space. The population wanders through the space stochastically, spreading and clustering at random.

The individuals in the initial population are assigned genotypes at random. Time is discrete, and moves forward each time step by iterating the following two-step algorithm:

(1) r individuals (selected at random, with replacement) each produce a child that is genetically identical to itself except for mutations. Mutant alleles are chosen at random from the set of possible alleles.

(2) r individuals (selected at random, without replacement) die and are removed from the population and are replaced by the r children produced at step (1).

See [Rechsteiner and Bedau 1999] for a more detailed treatment of the GNM as well as its application to ALife research.

4 Design of an ALife hash function (ALHF)

A simulation of an ALife model run for a finite number of timesteps can be thought of as a computable function from given initial conditions to some sort of resulting value. The analogy with cryptographc hash functions, as mappings from keys to hash values, is clear. To use an ALife model as a cryptographic hash function, one must (i) specify how the keys are related to the ALife model's initial conditions; and (ii) define what the resulting values are of an ALife model and specify how the hash values are related to these resulting values.

We use the GNM as a cryptographic hash function. The idea is to extract parameters for the GNM from the input key, let the GNM evolve, and use the resulting genome as the hash value.

Recall that the GNM has five parameters, N, r, l, a, and m_l. For the purpose of the ALHF, we introduce two new parameters: t, the number of time steps the GNM should evolve before terminating, and s, the seed for a pseudorandom number generator.

Since the ALHF was designed for application with a user-login system, the input key actually consists of two variables. One represents the login name of the user, call it κ_n. The other represents the password of the user, call it κ_p. Both κ_n and κ_p are 64-bit quantities, though our method can be extended to quantities of arbitrary bit length.

The values of parameters m_l, t, and s are functions of either κ_n or κ_p alone, or both. Although in principle there are a number of ways to extract the m_l, t, and s parameters from κ_n and κ_p, we chose the following:

$$m_l = M_L(\kappa_p) = \frac{\kappa_p \vee 2^{63}}{2^{64} - 1}$$

First, the most significant bit in κ_p is set. This ensures that $m_l > 0.5$. (Relatively high mutation rates, like $m_l > 0.5$, provide for security. Recall that the ciphertext is the genetic makeup of the population after t timesteps. With a relatively high m_l, the chance that the genetic makeup of the population at t is similar to the initial genetic makeup of the population is slim.) The resulting value is then bounded so that $0.5 < m_l \leq 1.0$:

$$t = T(\kappa_n, \kappa_p) = \frac{\sum_{i=1}^{8} (\kappa_{n_i} \vee (2^7)) \wedge (\kappa_{p_i})}{2}$$

For each consecutive 8 bit group (indexed by i) in both κ_n and κ_p, we set the most significant bit in κ_{n_i}. Next, this quantity is bitwise AND'ed with κ_{p_i}. (Here is where setting the most significant bit of κ_{n_i} comes into play. If $\kappa_{n_i} = \kappa_{p_i}$, then $\kappa_{n_i} \vee \kappa_{p_i} = 0$. If this happened for all i, then t would be 0, which is obviously undesirable. Setting the most significant bit in κ_{n_i} guarantees that this will never happen.) Finally, the resulting quantity is divided by 2 so as to scale down t to a computationally-feasible value.

$$s = S(\kappa_n, \kappa_p) = \neg\kappa_n \oplus \kappa_p$$

Here we simply flip all bits in κ_n and bitwise XOR this with κ_p.

These value were chosen so as to make use of the limited number of ASCII characters capable of being used in either a login name or a password. Little research was done to see if our choice for M_L, T, and S were optimally secure. More research must be done on this matter.

In essence, the ALHF is a function of seven variables to a hash value, such that:

$$ALHF(N, r, l, a, M_L(\kappa_p), T(\kappa_n, \kappa_p), S(\kappa_n, \kappa_p)) : N \times N \times N \times N \times N \times N \times N \rightarrow N$$

We let

$$\nu = ALHF(N, r, l, a, M_L(\kappa_p), T(\kappa_n, \kappa_p), S(\kappa_n, \kappa_p))$$

Of the seven parameters of the GNM, the parameters N, r, l, and a do not depend upon the input key for their values. The bit-length of the hash value is a function of the three parameters N, l, and a. Specifically, let $\mid \nu \mid$ be the bit-length of hash value ν, so that $\mid \nu \mid = N \cdot l \cdot \log_2 a$. To keep hash values with the same bit-length, either each of the N, l, and a parameters must be held constant for variable κ_n and κ_p or else some appropriate values for these parameters must be chosen. We held N, l, and a constant in a given run. (We let $a = 2^{16}$ in all cases. The reason was to make use of all 16 bits in a 2 byte addressable data

type. While the values of N and l, like a, were held fixed in a given run, their values varied across runs. See the following sections for the specific choices of the N and l parameters.) Although the r need not be constant (indeed, it could depend upon either κ_n or κ_p), we let r stay constant. The reason is that the r parameter plays a significant role in the number of computations needed; lower values of r result in fewer computations and, as we shall see in Section 4, do not render the ALHF any less secure (our tests produced no evidence to suggest that the security strength of the ALHF depended in any way upon r).

The seven parameters of the GNM thus set, the GNM is evolved. After t time steps, the evolution is halted. Next, for each existing agent (their order is arbitrarily given by the GNM) at t, its genotype is recorded, where an agent's genotype is just the ordered sequence of its alleles at each of its loci (alleles are indexed by natural numbers). The sequence of genotypes of all existing agents in the GNM is thus a string of natural numbers. This string, at t, just is the hash value of the ALHF.

Another way to think about the ALHF is as follows. The GNM is heavily based upon a pseudorandom number generator. Specifically, we used Wu's 61-bit pseudorandom number generator (see [Crandall and Pomerance 2001], though any cryptographically-appropriate pseudorandom number generator will suffice (see [Menezes *et al.* 1997] for a brief overview). The parameters N, r, l, a, m_l can be thought of as determining a selection function from a sequence of pseudorandom numbers to the genetic state of the GNM at a given time step. Distinct numbers are selected as this function is iterated along the sequence of pseudorandom numbers.

The ALHF offers a good deal of practical flexibility. The bit length of the ciphertext can be easily customized by appropriately choosing values for N, l, and a. Similarly, the functions T, S, and M can easily be altered. If CPU cycles are a concern, the value of r and the functions T and M can be customized so as to result in speedier performance with the potential trade-off of decreased security. Alternatively, they can be customized so as to result in slower performance with the potential trade-off of increased security. The ALHF is scalable with respect to how many computations are needed to calculate a hash value from an given input key. The bit length, run time, and (to some degree) the security strength of the ALHF can all be customized to suit individual needs.

5 Experimental results

Avalanche Test. The Avalanche Test for the ALHF went as follows. We chose eight different ciphertext bit lengths (2^4 up to 2^{11}) by choosing appropriate values for a and l. For each such choice of the ciphertext bit length given by a and l, we randomly selected 1000 pairs of input keys, $(\kappa_{n_1}, \kappa_{p_1}), ..., (\kappa_{n_{1000}}, \kappa_{p_{1000}})$. For each $(\kappa_{n_i}, \kappa_{p_i})$, we calcualted every possible $(\kappa'_{n_i}, \kappa'_{p_i})$ composed of ASCII printable characters differing only by a single bit from $(\kappa_{n_i}, \kappa_{p_i})$. Then,

$$H = ALHF(N, r, l, a, M_L(\kappa_{p_i}), T(\kappa_{n_i}, \kappa_{p_i}), S(\kappa_{n_i}, \kappa_{p_i}))$$

and

$$H' = ALHF(N, r, l, a, M_L(\kappa'_{p_i}), T(\kappa'_{n_i}, \kappa'_{p_i}), S(\kappa'_{n_i}, \kappa'_{p_i}))$$

are computed. Now, let

$$\delta((\kappa_{n_i}, \kappa_{p_i}), (\kappa'_{n_i}, \kappa'_{p_i})) = \frac{(H \oplus H')}{\mid H \mid}$$

The mean δ was recorded, along with the least and greatest δ values from the 1000 pairs of input keys. There were approximately 50 distinct pairs $(\kappa'_{n_i}, \kappa'_{p_i})$ for each $(\kappa_{n_i}, \kappa_{p_i})$. Also, the difference (i.e. range) between the greatest δ and least δ was recorded. The results follow.

The following graph plots the mean δ, least δ and greatest δ as a function of the ciphertext length in bits, each averaged over a number of permutations for the N, l, and r parameters. The standard deviation of δ is also plotted.

Collision Test. To test the collision-resistance of the ALHF, we focused upon the Birthday Attack. The Birthday Attack is typically avoided by secure cryptographic hash functions if the bit length of the ciphertext hash values are appropriately long. The ciphertext bit length of the ALHF is scalable, depending on one's choice of the N and l parameters. So the ALHF can easily be customized so as to have ciphertext bit lengths of an appropriate size to avoid the Birthday Attack.

Nevertheless, to verify that the ALHF did not face any peculiar problems with respect to the Birthday Attack, we opted to test the ALHF's response to the Birthday Attack. Testing the security of the ALHF against the Birthday Attack is computationally feasible only for small ciphertext bit lengths. We limited ourselves to ciphertext bit lengths of 16 and 32. Thus, for each of approximately $2^{32/2} = 2^{16}$ iterations (78644, to be exact), we randomly selected two pairs of input keys (κ_n, κ_p) and (κ'_n, κ'_p). Next,

$$ALHF(N, r, l, a, M_L(\kappa_{p_i}), T(\kappa_{n_i}, \kappa_{p_i}), S(\kappa_{n_i}, \kappa_{p_i}))$$

and

$$ALHF(N, r, l, a, M_L(\kappa'_{p_i}), T(\kappa'_{n_i}, \kappa'_{p_i}), S(\kappa'_{n_i}, \kappa'_{p_i}))$$

were calculated and recorded. Finally, we examined the list of hash values to see if there were any duplicate entries. Only one collision was found when the ciphertext bit length was 16 (i.e. $N = 1$ and $l = 1$). No collisions were discovered with ciphertexts of bit length 32.

Comparison to pseudorandom number hash functions. As suggested in Section 4, the ALHF uses the evolutionary processes of an ALife model to select elements from a sequence of pseudorandom numbers; the elements so selected comprise the ciphertext. It is thus natural to wonder whether the use of an ALife model for selecting these pseudorandom numbers offers any improvement over some other method which does not make use of an ALife model for selection. To this end, we show that the ALHF's performance with respect to the Avalanche and Birthday Tests is statistically equivalent to the performance of one such

different kind of hash function on the same tests. However, we conjecture that the ALHF nevertheless is still advantageous in at least one respect.

The different kind of hash function we used to test against the performance of the ALHF on the Avalanche and Birthday tests is a pseudorandom-number hash function (PRNHF). Roughly, the PRNHF behaves similarly to the ALHF, except no ALife model is used. So, the PRNHF, like the ALHF, is a function of κ_n and κ_p. We define t_{prn} as:

$$t_{prn} = T_{prn}(\kappa_n, \kappa_p) = 8 \cdot \sum_{i=1}^{8} (\kappa_{n_i} \vee (2^7)) \wedge (\kappa_{p_i})$$

and s_{prn} as:

$$s_{prn} = S_{prn}(\kappa_n, \kappa_p) = \neg \kappa_n \oplus \kappa_p$$

We define the PRNHF as follows:

$$\nu = PRNHF(N, l, a, T_{prn}(\kappa_n, \kappa_p), S_{prn}(\kappa_n, \kappa_p)) : N \times N \times N \times N \times N \rightarrow N$$

We let

$$\nu = PRNHF(N, l, a, T_{prn}(\kappa_n, \kappa_p), S_{prn}(\kappa_n, \kappa_p))$$

The PRNHF uses the parameters N, l, and a to set the bit length of the ciphertext analogously to the ALHF. Unlike the ALHF, the PRNHF simply extracts a random seed s_{prn} and number of iterations t_{prn} from κ_n and κ_p and records the last consecutive $N \cdot l \cdot \log_2 a$ bits as the hash value (ciphertext). Therefore, the PRNHF does not make use of any evolutionary processes to select just which elements of the sequence of pseudorandom numbers generated from s_{prn} are used as the ciphertext.

Here we plot graphs and tables pertaining to the PRNHF's performance on the Avalanche Test analogous to those presented for the ALHF. We plot the mean δ, least δ, and greatest δ as a function of the ciphertext length in bits, each averaged over a number of permutations for the N, l, and r parameters. The standard deviation of δ is also plotted.

As is evident, the security of the PRNHF with respect to the Avalanche Test is virtually identical to that of the ALHF. Also, the security of the PRNHF as witnessed in the Birthday Test was nearly identical to that of the ALHF.

The point of looking at the PRNHF was to see if the ALHF performed any differently on the various tests. It does not. Indeed, PRNHF achieves the same security with respect to the Avalanche and Birthday Tests as the ALHF while being less CPU-intensive. However, this should not suggest that the PRNHF is to be straightaway preferred. The PRNHF rather simplistically takes the last $N \cdot l \cdot \log_2 a$ bits of a sequence of pseudorandomly generated numbers as the ciphertext. The ALHF does not. The $N \cdot l \cdot \log_2 a$ bits of the ALHF's values are "scattered", as it were, across the sequence of pseudorandomly generated numbers as a direct result of the GNM's behavior. This is because just which of the pseudorandomly generated numbers are present in the final sequence (i.e. the

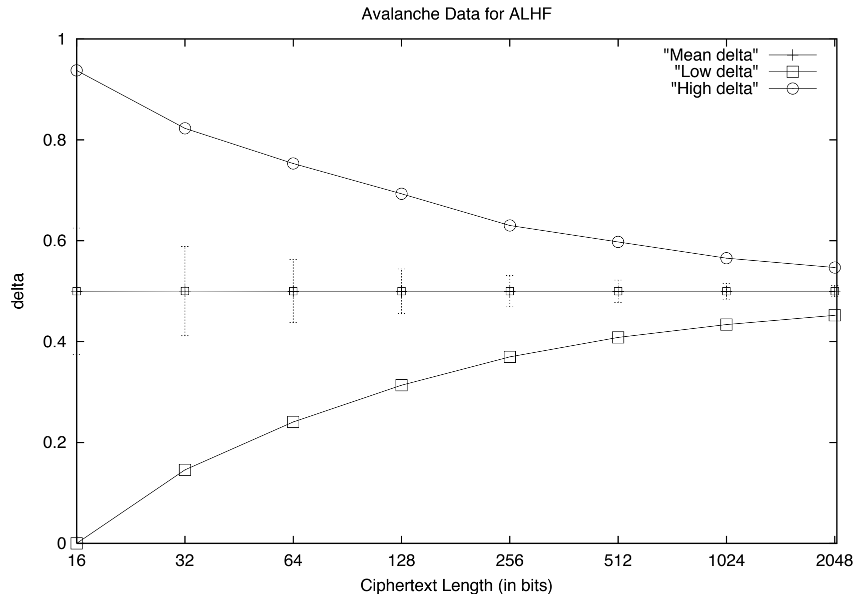

Fig. 1. Avalanche data for the artificial life hash function (ALHF).

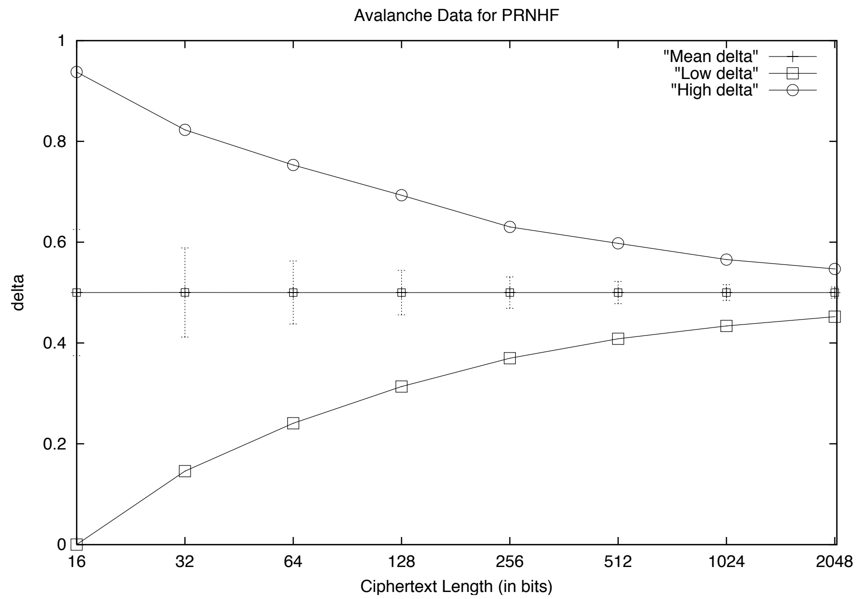

Fig. 2. Avalanche data for the pseudorandom-number hash function (PRNHF). The plot is virtually identicial, as hoped and expected, to that of Figure 1.

final state of the GNM's genome) depends upon the evolution of that particular GNM simulation. As the GNM's parameters vary corresponding to variations in the login name and password, so too does the evolution of the GNM. Just which agents survive, and thus just which agents contribute their genes to the final genome used as the ciphertext, varies across simulations of the GNM. The elements of the pseudorandomly generated sequence comprising the final genome, therefore, fluctuates as a function of the login name and password. In this sense, then, the bits of the ALHF's values are "scattered".

While this "scattering" offers no advantage in the statistical tests we provide here, it is nevertheless suggested that this "scattering" is beneficial in that it makes it more difficult to backtrack from a known ciphertext to the sequence of pseudo-randomly generated numbers from which it is a result. The bits comprising the ciphertext hash value of the PRNHF are not thus "scattered", and thus it is potentially easier to backtrack from known ciphertexts with the PRNHF to the sequence of pseudo-randomly generated numbers from which they result. By making this backtracking "harder", the ALHF makes finding the input key "harder". In this respect, we believe the ALHF is more secure than the PRNHF.

6 Conclusion

By inspecting the tables presented for the Avalanche Test, we can see that the r does not significantly affect the performance of the ALHF on the Avalanche Test. Instead, the crucial parameters seem to be a, l, and N, since they together determine the ciphertext bit length. As can be seen, the performance of the ALHF improves on the Avalanche Test as the ciphertext bit length increases. The mean δ gets closer to 0.5, δ's standard deviation decreases, as does the δ range. This suggests ciphertext bit length (and thus the parameters a, l, and N) are the most significant parameters for the Avalanche Test.

Acknowledgements The authors are indebted to J. P. Buhler for insights.

7. References

Bedau, Mark A., et al. "Open Problems in Artificial Life." Artificial Life 6 (2000): 363-376.

M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," *Advances in Cryptology—Crypto 96 Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag 1996.

Berlekamp, Elwyn R., Conway, John H., and Guy, Richard K. *Winning Ways for your Mathematical Plays, Vol. 2: Games in Particular.* New York: Academic Press, 1982.

T. Beth, M. Frisch, and G.J. Simmons, eds., *Lecture Notes in Computer Science 578; Public Key Cryptography: State of the Art and Future Directions*, Springer-Verlag, 1992.

R. Crandall and C. Pomerance, *Prime numbers: a computational perspective*, Springer-Verlag, New York, 2001.

H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, vol. 228, no. 5, pp. 15-23.

L. Knudsen, X. Lai, and B. Preneel, "Attacks on Fast Double Block Length Hash Functions," *J. Cryptology*, 11: 59-72, 1998.

D. Knuth, "The Art of Computer Programming," third edition, Addison Wesley, 1998.

J. Lagarias, "Pseudorandom Number Generators in Cryptography and Number Theory," in *Cryptology and Computational Number Theory*, ed. C. Pomerance, Amer. Math. Soc., Providence, RI, 1990.

Langton, Christopher G., ed. *Artificial Life: The Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems held September, 1987 in Los Alamos, New Mexico*. Redwood City: Addison- Wesley, 1989.

A.J. Menezes, P. van Oorschot, and S.A. Vanstone, "Handbook of Applied Cryptography," CRC Press, Inc., Boca Raton, FL, 1997.

R.C. Merkle, "Secrecy, Authentication, and Public Key Systems," Ph.D. dissertation, Stanford University, 1979.

R.C. Merkle, "One Way Hash Functions and DES," *Advances in Cryptology—CRYPTO '89 Proceedings*, Springer-Verlag, 1990, pp. 428-446.

P. van Oorschot and M.Wiener, "Parallel collision search with application to hash functions and discrete logarithms," in *Proceedings of* 2nd *ACM Conference on Computer and Communication Security*, 1994.

B. Preneel, "Analysis and Design of Cryptographic Hash Functions," Ph.D. Thesis, Kath-olieke Universiteit Leuven, 1993.

B. Preneel, "Cryptographic Hash Functions," *European Transactions on Telecommunications*, vol. 5, n. 4, Jul/Aug 1994, pp. 431-448.

A. Rechsteiner and M. A. Bedau, "A Generic Model for Measuring Excess Evolutionary Activity," in *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida*, vol. 2, ed. Morgan Kaufmann, pp. 1366-1373. San Franciso, CA, 1999.

A. Rechtsteiner and M. A. Bedau. 1999. "A Generic Model for Quantitative Comparison of Genotypic Evolutionary Activity," in Dario Floreano, Jean-Daniel Nicoud, Francesco Mondada, eds., *Advances in Artificial Life, Fifth European Conferenc, ECAL99*, pp. 109-118. Heidelberg: Springer-Verlag. Lecture Notes in Artificial Intelligence 1674.

T. Satoh, M. Haga, and K. Kurosawa, "Towards Secure and Fast Hash Functions," IEICE Trans. Fund., vol. E-82A, 1, Jan. 1999.

B. Schneier, “Applied Cryptography: Protocols, Algorithms, and Source Code in C,” second edition, John Wiley & Sons, Inc., New York, NY, 1996.

J. Seberry, X. Zhang, and Y. Zheng, “Improving the strict avalanche characteristics of cryptographic functions,” Information Processing Letters, vol. 50, pp. 37-41, 1994.

J. Seberry, X. Zhang, and Y. Zheng, “Structures of cryptographic functions with strong avalanche characteristics,” Advances in Cryptology – AsiaCrypt’94, Lecture Notes in Computer Science, vol. 917, pp. 119-132, Springer-Verlag, 1995.

D.R. Stinson, “Cryptography: Theory and Practice,” CRC Press, Inc., Boca Raton, FL, 2000.

Von Neumann, J. (1966) *Theory of self-reproducing automata*, University of Illinois Press.

A.F. Webster, “Plaintext/ciphertext bit dependencies in cryptographic systems,” Master’s Thesis, Department of Electrical Engineering, Queen’s University, Ontario, Canada, 1985.

A.F. Webster and S.E. Tavares, “On the design of S-boxes,” in *Advances in Cryptology – CRYPTO’85*, vol. 219, Lecture Notes in Computer Science, pp. 523-534. Springer-Verlag, Berlin, 1986.

G. Yuval, “How to swindle Rabin,” *Cryptologia*, July 1979.

X. Zhang and Y. Zheng, “GAC – the criterion for global avalanche characteristics of cryptographic hash functions,” Journal of Universal Computer Science, vol. 1, no. 5, pp. 316-333, 1995.