Unkeyed Jam Resistance 300 Times Faster: The Inchworm Hash

Leemon C. Baird III
and Martin C. Carlisle
and William L. Bahn
Academy Center for Cyberspace Research
Computer Science Department
US Air Force Academy
USAFA, Colorado 80840
Email: Leemon.Baird@USAFA.edu

***Abstract*—An important problem is achieving jam resistance in omnidirectional radio communication without any shared secret or shared key. The only known algorithm that solves this problem is the BBC (Baird, Bahn, Collins) concurrent code [1]. However, BBC requires the choice of a hash function. The choice of hash determines both the speed and security of BBC. Cryptographic hashes such as the standard SHA-1 hash are not well suited for this application. We propose the *Inchworm hash*, a new algorithm specifically designed for use in BBC. We show that this avoids a theoretical weakness for this application that is present in SHA-1 due to the Small Internal State Theorem [2], and that it passes a simple battery of empirical tests. When used in BBC, Inchworm is over 300 times faster than SHA-1. This speeds up encoding and decoding by orders of magnitude, with great benefits for practical implementations of unkeyed jam resistance, especially on small, cheap radios.** [1]

I. Introduction

Jam resistance has typically been achieved in omnidirectional radios using some kind of secret shared by the sender and receiver. In frequency hopping, the sequence of frequencies must be secret. In Direct Sequence Spread Spectrum (DSSS), the chip sequence must be kept secret. In pulse-based Ultra Wideband (UWB) systems, a secret must be incorporated into the timing of the pulses. However, it is desirable to achieve jam resistance without keys. This makes key management easier, since there is no key. It allows the manufacture of large numbers of identical radios that never have to be re-keyed, and that don't have to be protected from adversary reverse engineering.

There is currently only one system for achieving such unkeyed jam resistance: the BBC (Baird, Bahn, Collins) concurrent code [3], [1], [4], [5], [6], [7], [2], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. The algorithm will not be given here; see the above references for detailed descriptions of it. It is important to note that one part of this algorithm requires the choice of a hash function. The hash function can be any function that takes in a binary string of up to about 1000 bits, and returns a hash of at least 20 to 30 bits. Ideally, the mapping from inputs to outputs should look random, though any given input should always give the same output.

When BBC decodes information that the receiver's radio receives, the received information is represented as a *packet* of bits, with about one third of them set to 1 and the rest 0. The BBC decoder then converts this packet into a set of received messages. During decoding, most of the time is spent in calls to the hash function.

The BBC decoder starts by calling the hash function on an empty string. Then it repeatedly adds one bit to the end of the string or deletes one bit from the end of the string, and calculates the hash of the resulting string. The string is therefore constantly growing and shrinking at the right end, as the decoder performs a depth-first search through a space of possible strings. This search is expensive for a standard hash function such as SHA-1 because it must start over from scratch when hashing each of the strings. It would be better to have an incremental hash function that can quickly calculate a hash of a new string that differs from the last string in just a single bit addition/deletion at the end. That is the motivation for creating the Inchworm algorithm, which turns out to be more than 300 times faster than SHA-1 for this specific application.

In a BBC system, the attacker may try to create an *attack packet* which is a sequence of bits with one third 1 bits and the rest 0 bits. A good attack packet is designed to require as many calls as possible to the hash function during decoding. A successful attack will require so many calls that the receiver is overwhelmed and is unable to decode all of the packet, thus jamming the communication. The *security* of a hash function for BBC is the degree to which it prevents this attack.

The Small Internal State Theorem [1] proves that for any hash with a small amount of changeable memory (a small internal state), there will exist an attack packet that increases the decoding time exponentially. Therefore, it would be preferable to implement BBC with hashes that have as many bits of internal state as there are bits in the longest string that will be hashed. In fact, they should be twice as long, if doing average-case rather than worst-case analysis. For a typical BBC implementation, this means it needs at least 2000 bits of

[1]This work was sponsored in part by the 688th Information Operations Wing (IOW), Lackland AFB, TX, and was performed at the Academy Center for Cyberspace Resaerch (ACCR) at the United States Air Force Academy.

internal state, which is more than any of the cryptographic hash functions currently in use. Of course, this is only an existence proof. It proves that effective attacks *exist* for a hash like SHA-1. But it may be difficult for an attacker to actually find how to do such an attack with a practical amount of computation. So the flaw in SHA-1 is only theoretical. And it is only a "flaw" when SHA-1 is used in a BBC code, not when it is used for its originally-intended purpose. Still, it is possible to avoid even this theoretical flaw by developing a hash function with a larger internal state. So for both speed and security reasons, there is a need for a better hash to be developed specifically for use in BBC. Inchworm is such a hash.

II. Derivation of Inchworm

The proposed Inchworm hash is shown as a block diagram in Fig. 1 and as a C implementation in Fig. 2. The C code is the official definition of the algorithm, taking precedence in case of conflict with the diagram or text. It is designed to be very fast when used incrementally, to have an internal state of more than 2000 bits, and to at least pass the currently-known theoretical and empirical security tests (though few such tests exist yet for testing BBC security). This section will go through the derivation of the algorithm, explaining the reasons for each choice made along the way.

Hashes are typically designed by dividing the string to be hashed into blocks $(b_1, b_2, b_3, ...b_n)$, initializing an internal state buffer B to all zeros, and then repeatedly consuming each block b_i and updating the internal state B with some nonlinear function F:

$$B \leftarrow F(B,\, b_i) \tag{1}$$

The arrow represents a variable assignment, which is first done for b_1, the first block of the string to hash, then for b_2, and so on for all the blocks of the string. After the final block is hashed, the final value of B is the hash of the string.

The F function is typically not invertible. However, BBC decoding benefits from an *incremental* hash that can hash a string then quickly recalculate the hash when a bit is added to the end or deleted from the end. To support efficient deletions, F must be invertible. So we will choose F to be an "encryption" of B using B_i as a "key", to ensure it can be inverted. There are many ways to build such a cipher, but one common design is a *Feistel* design. In a *balanced* Feistel, the buffer B is divided into two halves, one of which is updated on each step. This would be fast on modern processors if the internal state of the hash were small enough to fit in a few registers. But the Small Internal State Theorem requires the internal state to be on the order of twice the message length, as mentioned above, so the lengths should sum to around 2000 bits. Therefore, a better approach is to use an *unbalanced* Feistel structure. The internal buffer B will consist of 31 different blocks of 64 bits each, $B_0, B_1, B_2, ...B_{30}$. Then, as the ith bit of the string, b_i, is consumed, only one block is updated:

$$B_{i \bmod 31} \leftarrow B_{i \bmod 31}\ XOR\ f(B_{i-1 \bmod 31},\ b_i) \tag{2}$$

where f is an arbitrary function, that is not necessarily invertible. Note that as each bit b_i is consumed, it immediately affects the single block $B_{i \bmod 31}$ directly, and then affects all future blocks of B indirectly, because each time an element of B is modified, the new value is then fed back into the f function on the next round. So at any given time, the latest block of B acts as a fast-changing internal state, while the rest of B is a slowly-changing internal state.

Note that the above assignment cancels itself if performed twice, so it is easy to run this algorithm backward, to return the internal state to an earlier value. So once a string has been hashed, bits can be deleted from the end of it, and the algorithm run backward to return to the appropriate internal state. Then new bits can be added to the end, and the hash of the new string can be found quickly.

The function f is shown as being a function of only B and b. But there would be no problem if it were a function of additional internal state variables, as long as they, too, could be run backward. For example, there could be a 64-bit variable S that changes over time in an invertible way, if it is updated by the bit b_i using some constant D this way:

if (b_i=0)
    $S \leftarrow S$ XOR D
$S \leftarrow S <<< 39$

This updates S by XORing with the constant, if and only if the bit is 0. And then the $<<<$ operator rotates S left by 39 positions, regardless of the value of the bit. The conditional XOR allows the bit to affect about half the bits of S (assuming the constant D has about half its bits set to 1). The rotation ensures successive XORs with D won't cancel each other. The number 39 is coprime to 64 (i.e. they have no factors in common), so it takes a full 64 steps before S rotates back around to where it started. A rotation by 1 would also work, but rotating by 39 ensures that each bit moves fairly far each time, and takes as long as possible before it is even near its original position. A rotate by 39 has that effect on a 64-bit variable because 39 is the Golden Ratio times 64, rounded to the nearest number coprime to 64.

This update to S ensures good avalanche, in the sense that each bit b_i affects about half the bits of S, assuming D is a typical random number, and so has about half its bits set to 1. It also ensures that any given bit b_i will continue to have an effect on S for a fairly long time. However, it is purely linear. If you look at S after hashing a string of bits, and then redo it with just one bit different, the two S values will differ by the XORs of just two rotations of D.

To make this nonlinear, we first need to add a second 64-bit internal state variable, R, with slightly different parameters. It can be updated with a new constant C, and rotated by a slightly-different amount 37 (which is close to 39 and coprime

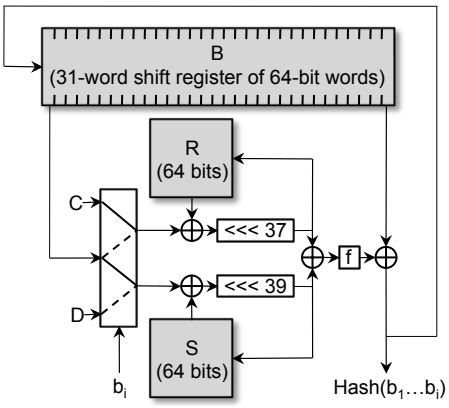

Fig. 1. Block diagram of Inchworm (f is the identity function) and Inchworm-S (f is more complex), showing the update for b_i, the ith bit from the string to be hashed, where $<<<$ is left rotation, $\oplus$ is XOR, and the switch in the lower left routes signals along dotted lines when $b_i = 0$, and solid when $b_i = 1$. Every arrow carries a 64-bit signal (except b_i).

to both 39 and 64). The bits can interact nonlinearly if the XOR of S and R with the current block of B is returned as the hash, and if the hash then feeds back into S or R on the next step.

if (b_i=1)

$$R \leftarrow (R \text{ XOR } C) <<< 37$$

$$S \leftarrow (S \text{ XOR } B_{i-1 \bmod 31}) <<< 39$$

else

$$R \leftarrow (R \text{ XOR } B_{i-1 \bmod 31}) <<< 37$$

$$S \leftarrow (S \text{ XOR } D) <<< 39$$

$$B_{i \bmod 31} \leftarrow B_{i \bmod 31} \text{ XOR } R \text{ XOR } S$$

The result calculated on the last line is returned as the hash of the string $(b_1, b_2, ..., b_i)$. So if the bits of a string are fed to the algorithm in order, it will calculate the hash of all prefixes of the string, as is needed in BBC decoding.

This is almost the complete Inchworm algorithm for hashing the next bit. All that remains is to define the initialization done at the start of the string, and the values of the C and D constants. The complete algorithm is shown as a block diagram in Fig. 1 and as a C implementation in Fig. 2.

It would seem that most arbitrary constants C and D would be acceptable for this algorithm. Therefore, the given C code uses Inchworm itself as a pseudo-random number generator to create the constants through a bootstrapping process. The constants are first set to 1, then the hash is run 512 times, feeding back the least significant bit of each hash output as the next input bit to hash. After 512 rounds, the final output is captured to define C. Then another 512 iterations are done using that C to get D. And another 512 rounds are run using those C and D values to initialize the system at the start of each string. The example code recalculates C and D this way every time a new string is hashed. Of course, any practical implementation would just hard code the C and D constants rather than recalculating them every time.

```
//  Inchworm - A hash for use with BBC codes
//  25 Aug 2010,  Version 1.0
//  Leemon Baird

// The Inchworm internal state.
typedef struct  {
    uint32_t p;
    uint64_t R, S, C, D, B[31];
} inchworm_t;

//You should check that your compiler compiles each
//of the following as a single x86-64 rotate instruction
inline uint64_t rotL (int n, uint64_t x)
    {return (((x)<<(n)) ^ ((x)>>(64-n)));}
inline uint64_t rotR (int n, uint64_t x)
    {return (((x)>>(n)) ^ ((x)<<(64-n)));}

//uncomment the 1st line for Inchworm, 2nd for Inchworm-S
#define f(x) x
//#define f(x) g(1,g(3,g(9,g(27,(x)))))

inline uint64_t g(int r, uint64_t x) {
    return rotL(r,x) ^ (x | rotR(r,x));
}

//Set the current string to the empty string,
//and return its hash. This must be called
//before inchwormAdd or inchwormDel.
uint64_t inchwormReset(inchworm_t *s) {
    s->C = s->D = 1;
    s->C = inchwormReset2(s);
    s->D = inchwormReset2(s);
    return inchwormReset2(s);
}

uint64_t inchwormReset2(inchworm_t *s) {
    int i;
    uint64_t b;
    b = s->R = s->S = s->p = 0;
    for (i=0; i<31; i++)
        s->B[i] = 0;
    for (i=0; i<512; i++)
        b = inchwormAdd(s,b&1);
    return b;
}

// Concatenate a bit (0 or 1) to the end of the
// current string, and return its hash.
//  inchworm_t *s  is the inchworm state
//  uint64_t   bit is string's new last bit (0 or 1)
#define inchwormAdd(s,bit) (                                    \
  (bit) ? ((s)->R = rotL(37,(s)->R ^ (s)->C ),                  \
           (s)->S = rotL(39,(s)->S ^ (s)->B[(s)->p % 31])) \
        : ((s)->R = rotL(37,(s)->R ^ (s)->B[(s)->p % 31]), \
           (s)->S = rotL(39,(s)->S ^ (s)->D )),                 \
  ((s)->p)=(((s)->p)+1),                                        \
  (s)->B[(s)->p % 31] ^= f((s)->R ^ (s)->S)                     \
)

//Delete the last bit, return the resulting string's hash.
#define inchwormDel(s,bit) (                                    \
  (s)->B[(s)->p % 31] ^= f((s)->R ^ (s)->S),                    \
  ((s)->p) = (((s)->p)-1),                                      \
  (bit) ? ((s)->S = rotR(39,(s)->S) ^ (s)->B[(s)->p % 31], \
           (s)->R = rotR(37,(s)->R) ^ (s)->C)                   \
        : ((s)->S = rotR(39,(s)->S) ^ (s)->D,                   \
           (s)->R = rotR(37,(s)->R) ^ (s)->B[(s)->p % 31]),\
  (s)->B[(s)->p % 31]                                           \
)
```

Fig. 2. C implementation of the Inchworm algorithm.

Inchworm is nonlinear. The first bit affects both R and S in a large way, flipping about half the bits of each. So the first bit b_1 affects about half the bits in the returned hash value (i.e. it has good avalanche properties, as described above). When

the second bit b_2 is processed, it controls whether that result will be XORed into R or S, which in turn controls whether it is rotated by 37 or 39 positions. So many of the bits of the second hash are affected by both b_1 and b_2, interacting nonlinearly. This continues for the rest of the bits of the string being hashed, so that all of them interact nonlinearly with each other.

Inchworm is also surprisingly simple, using only 64-bit XORs and rotations, plus a simple decision based on the incoming bit. There are no binary multiplications or additions, no table lookups or S-boxes, and no complex bit transpositions. There aren't even any bitwise AND or OR operations. All operations are fast in both hardware and software. The B buffer is a simple shift register, with only a single entry being read and written on each step. All of the operations are independent of the endianness of the machine used to implement it. But if it is desired to interpret the hash as a sequence of bytes, such as when writing it to a file, it is hereby defined to output in little endian order (i.e. least significant byte first).

Though there is an enormous literature on analyzing cryptographic hash functions, it is not clear how much of that carries over to hash functions designed for use in BBC codes. Traditional hash functions are intended to make it difficult to find collisions. But the Inchworm output is only 64 bits, so it is easy to find collisions. On the other hand, traditional hashes typically have small internal state, but large internal state is required by the theory of BBC codes. So both Inchworm and traditional hashes are "weak" by the standards applied to the other. Traditional hash functions are often analyzed with some form of linear or differential cryptanalysis. It is not at all clear that such analysis has any relevance for a hash used for BBC codes. Although Inchworm currently avoids all known theoretical attacks when used in BBC, that is not saying much, since the theory of how to attack BBC is still in its infancy.

Given these uncertainties, it might be desirable to add additional nonlinearities to Inchworm, using operations that appear to be different from the current ones, in order to possibly strengthen it against unknown attacks. That is the purpose of the block labelled "f" in the block diagram. In normal Inchworm, that block is the identity function, $f(x) = x$. But we will now propose a variant algorithm called *Inchworm-S* (where "S" stands for both "Secure" and "Slow"), that puts additional operations in the f box. It is clear that Inchworm-S is slower than Inchworm. Whether it is also more secure remains to be seen. The new f function is defined in terms of a g function:

$$f(x) = g(1, g(3, g(9, g(27, x)))) \tag{3}$$

$$g(r, x) = (x <<< r)\ XOR\ (x\ OR\ (x >>> r)) \tag{4}$$

These operations work on different principles from the rest of Inchworm. The g function combines the 64-bit x with copies of itself rotated left and right by r bits. The OR ensures nonlinearity, and the XOR ensures 1 and 0 bits are about equally common. When $r = 1$, this g operation is one that has been proposed before for cryptographic purposes [21]. Although it has received little cryptanalysis, and so its cryptographic security is largely unknown, it forms only a small part of Inchworm-S, and so may be good enough for the present purposes. In the past, it has only been suggested with $r = 1$, which has terrible avalanche properties, requiring 32 steps for a single bit to affect all others. That is why consecutive powers of 3 are used for r here, to achieve full avalanche in just 4 rounds, as shown.

Inchworm uses 64-bit registers for R, S, the elements of B, and the final hash output. The registers could be increased to larger numbers of bits if it were being run on a processor optimized for larger sizes. In that case, the algorithm would remain largely unchanged. The C and D constants would still be bootstrapped the same way. The rotation distance of 39 would become a rotation by the Golden Ratio times the new register size, rounded to the nearest number coprime to the register size. And the 37 would become the nearest number that is coprime to both that number and the size. Inchworm-S would continue to use powers of 3 (assuming the new register size is not a power of 3). It would use all powers of 3 up to the largest power of 3 that is less than the new register size. There is no apparent reason to change the size of the shift register (31 elements) or the number of rounds in the initializer (512 iterations), regardless of the size chosen for the registers. It might be expected that larger registers give a more secure hash for BBC, though they would certainly slow down the algorithm when running on a processor optimized for 64-bit registers.

To aid in implementing the algorithm, it is useful to know the constants C and D, rather than deriving them through the bootstrap method. It is also useful to have good test vectors. Because of the feedback used in the initialization algorithm, the value returned by the initialization (which is the hash of the empty string) is a sufficient test vector. It gives the result of hashing 512 bits, about half of which are 0 and half 1, which tests the algorithm thoroughly. The constants and test vectors for both algorithms are:

```
Inchworm:
    C      = 0xd489ebd61e8e3ea1
    D      = 0x2d236ed1707ecf2c
    hash() = 0x0c29b196ec9c4ef5
Inchworm-S:
    C      = 0x808ae1ad9290478c
    D      = 0x09f598887c4c10fc
    hash() = 0x093aa5618c96e5a9
```

Inchworm is a 64-bit hash. As with any other 64-bit hash, it is easy to modify the output as needed. If an n-bit hash is need, for $n < 64$, then use the n least significant bits of Inchworm's output. If a hash value is needed in the range 0 to n-1, then interpret the output of Inchworm as an unsigned integer, and take the output modulo n. Both of those approaches are useful for most BBC implementations, where the output of the hash will be interpreted as choosing a point in time. However, for a

frequency-hopping BBC implementation, it may be necessary for it to choose both a point in time and a frequency, (i.e., a 2-dimensional hash). If there are n points in time and m frequencies from which to choose, then the 32 least significant bits of the output can be taken modulo n and the upper 32 bits can be taken modulo m to choose both a time and a frequency.

In one such implementation, there would be n points in time and m frequencies, and whenever the system is given an n-bit string, that string will have n possible prefix strings (from 1 bit to n bits in length), and the hash must choose a pseudo-random frequency for each prefix string, and a *distinct* pseudo-random time for each prefix string. So the frequency hashes can still be chosen by simply taking the upper 32 bits modulo m, but the time hashes must be chosen in a slightly different way, so that no two prefixes have the same time hash. This is easy to accomplish. Create an n-element array initialized with the numbers from 0 to n-1. When the first prefix (which is just the first bit of the string) is hashed, take the least significant 32 bits modulo n to choose an element of the array. Swap that element with the nth element of the array, and return it as the time hash of the first prefix. Then when the second prefix is hashed (i.e. the first two bits of the string), take the least significant 32 bits modulo $n-1$, swap the first element with the element in position $n-1$, and return that as the time hash for the second prefix. The third time hash works modulo $n-2$, and so on.

Of course, this entire discussion of modifying the output applies to any hash, not just Inchworm. but it is included here for completeness.

III. Empirical Results

In addition to mathematical analysis of a hash, it is useful to get empirical data on how well a computer search can attack it. If it survives that attack, that provides some (weak) evidence that the hash is secure. But if such an attack is successful, then the hash is definitely insecure, at least for that application. It is also useful to time it and compare it to SHA-1 for the case of incremental hashing, as needed in BBC decoding.

Inchworm was timed at 2.56 clock cycles per hash, on a 2.8 GHz MacBook Pro (one core of an Intel Core 2 Duo), and SHA-1 was timed at 1665 cycles, so Inchworm was 650 times faster. These are cycles per string hashed, not cycles per byte or cycles per bit. This was non-optimized Inchworm (in C without any embedded assembly language) compared to the optimized SHA-1 (written in C with embedded assembly) that is part of the OpenSSL package that comes with the MacOSX operating system.

During the SHA-1 test, all prefixes of a 1000-bit string were hashed equally often. During the Inchworm test, half of the hashes were found by adding a bit to the last string hashed, and half were found by deleting a bit, with 0 and 1 bits occurring equally often. It was also timed with only bit additions instead of mixed additions and deletions, and the time was the same. This was timed with code carefully written to avoid letting the compiler cheat. For example, the random number generator was used to either set x=0 and y=1 or to set them as the reverse (x=1 and y=0). Then each call to add or delete a bit passed in either x or y, which prevented the compiler from knowing whether it was a 0 or a 1, and so prevented it from hard coding the if statements. And the entire Inchworm internal state was printed at the end to ensure there were no unused variables for the compiler to eliminate. The timing loop was manually unrolled, and macros were used for implementing the hash function, in order to increase speed. The C code given in this paper is the exact code that was timed. Although Inchworm would be slower for hashing an arbitrary string, it is clearly very fast for the incremental problem of hashing a string that changes by a single bit at the end, as occurs in BBC decoding. Inchworm was specifically designed for this single purpose, and appears to be well suited for it.

Note that Inchworm averages so few cycles that it was very sensitive to subtle cache effects, making consistent timing difficult. Changing irrelevant parts of the code (even code after the timing loop) had large effects on the timing. In an actual program that used Inchworm, the time per hash could be longer or shorter, or even zero (with superscalar pipelining).

In an actual BBC implementation, during decoding it would have to both add and delete one bit for each hash of interest, so in that sense Inchworm should be considered to be only half as fast. So it might be best to describe Inchworm as 325 times as fast as SHA-1, which is why the more conservative ratio of 300 is claimed in the title of this paper. But regardless, it is clear that Inchworm is orders of magnitude faster than existing hashes for this particular application, and that most BBC programs that use Inchworm will end up spending the vast majority of their time in operations other than the hash. Further speedups of Inchworm are unlikely to have much of an effect on total running time of any real implementation of BBC that uses Inchworm.

Inchworm-S was timed at 10.8 clock cycles per hash, using the same timing procedures as for Inchworm, so it is almost 5 times slower. Even so, 10.8 clock cycles is fairly fast for a hash function, and is still orders of magnitude faster than the 1665 cycles required by SHA-1. For applications where security is more important than raw speed, Inchworm-S might be a useful algorithm. None of our current theoretical or empirical tests are able to distinguish the security of Inchworm and Inchworm-S. It is theoretically possible that they are equal, or even that Inchworm is actually more secure than Inchworm-S. But at this point, Inchworm-S would seem to be the more conservative choice for security.

A simple empirical attack was used to compare the security of Inchworm, Inchworm-S, and SHA-1 for use in BBC. The attacker was a randomized algorithm that created a BBC *attack packet* designed to maximize the number of calls to the hash during its decoding, while having only one third of its bits set to 1. In each run, a 2048-bit packet was initialized to the encoding of a 40-bit random message. Then, it repeatedly flipped the 0 bit in the packet that would most increase the number of calls to the hash function needed to decode it. When it reached a one third density of one bits, it would alternate that operation with the flipping of the 1 bit that caused the

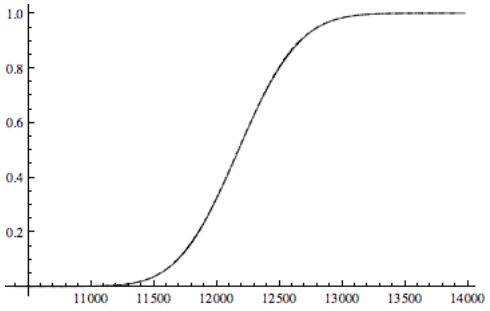

Fig. 3. Cumulative Distribution Function (CDF) for results of the attack optimizer running against Inchworm (solid), Inchworm-S (dashed) and SHA-1 (dotted) for a quarter million iterations each (a third that for SHA-1). The curves are practically identical.

smallest decrease in the number of calls. When a pair of such operations failed to change the packet, it returned the number of calls needed to decode that packet, and started over. This was repeated 250,000 times for Inchworm, the same for Inchworm-S, and a third of that for SHA-1 (because it is so much slower).

Figure 3 shows the resulting probability distribution for each of the three algorithms. The curves are nearly identical, and seem to lie directly on top of each other. They also have nearly identical mean, median, mode, min, and max. These three algorithms are therefore equally resistant to attack from this simple attacker. This is good, because this attacker defeated dozens of earlier versions of Inchworm during its development. This doesn't prove true security, of course. It is just serves as a sanity check to guard against certain blatant flaws. And it serves as a first step that can be expanded in future research.

IV. Conclusion

A new hash function was proposed, the *Inchworm* hash. This hash was designed specifically for use in BBC, which is a very different application from the typical uses of cryptographic hashes. For this application, The new hash was shown to be theoretically more secure than SHA-1 for BBC codes, while being over 300 times faster than SHA-1. In addition, empirical computer searches were run to find good attacks on the new hash, and it survived those attacks without problems.

The 300-fold speedup makes Inchworm an important improvement for real-world implementations of BBC for unkeyed jam resistance, especially on small, cheap radios with less computational power.

Inchworm has not yet been broken by any of the current analytical or empirical attacks considered here. The Inchworm-S variant might be more secure, though it is slower. The use of hashes for BBC codes is very different from traditional cryptographic applications of hashes, so there is currently little known about how to analyze its security. It would clearly be useful to develop new cryptanalytic methods for analyzing and attacking BBC hashes, and applying them to Inchworm. Neither Inchworm nor any other hash should be trusted for use in BBC until further analysis has been done. This appears to be a fruitful area for further research.

References

[1] L. C. Baird III, W. L. Bahn, and M. D. Collins, "Jam-resistant communication without shared secrets through the use of concurrent codes," U. S. Air Force Academy, Tech. Rep. USAFA-TR-2007-01, Feb 14 2007.

[2] L. C. Baird III and W. L. Bahn, "Security analysis of bbc coding," U. S. Air Force Academy, Academy Center for Cyberspace Research, Tech. Rep. USAFA-TR-2008-ACCR-01, Dec 8 2008.

[3] W. L. Bahn, L. C. Baird III, and M. D. Collins, "The use of concurrent codes in computer programming and digital signal processing education," *Journal of Computing Sciences in College*, vol. 23, no. 1, pp. 174–180, Oct 2007, also in the Proceedings of the 16th Annual Rocky Mountain Conference of the Consortium for Computing Sciences in Colleges (RMCCSC), Orem Utah.

[4] W. L. Bahn and L. C. Baird III, "Impediments to systems thinking: Communities separated by a common language," in *Proceedings of the 4th International Conference on Cybernetics and Information (CITSA)*, July 12-15 2007, pp. 122–127.

[5] L. C. Baird III, W. L. Bahn, M. D. Collins, M. C. Carlisle, and S. Butler, "Keyless jam resistance," in *Proceedings of the 8th Annual IEEE SMC Information Assurance Workshop (IAW)*, June 20-22 2007, pp. 143–150.

[6] L. C. Baird III and D. H. Kraft, "A new approach for boolean query processing in text information retreival," in *Proceedings of the International Fuzzy Systems Association (IFSA) 2007 World Congress*, June 18-21 2007.

[7] D. Schweitzer, L. C. Baird III, and W. Bahn, "Visually understanding jam resistant communication," in *Proceedings of the 3rd International Workshop on Visualization for Computer Security*, Oct 29 2007, pp. 175–186.

[8] W. L. Bahn and L. C. Baird III, "Extending critical mark densities in concurrent codecs through the use of interstitial checksum bits," U. S. Air Force Academy, Academy Center for Cyberspace Research, Tech. Rep. USAFA-TR-2008-ACCR-02, Dec 8 2008.

[9] ——, "Hardware-centric implementation considerations for bbc-based concurrent codecs," U. S. Air Force Academy, Academy Center for Cyberspace Research, Tech. Rep. USAFA-TR-2008-ACCR-03, Dec 8 2008.

[10] W. L. Bahn, L. C. Baird III, and M. D. Collins, "Jam resistant communications without shared secrets," in *Proceedings of the 3rd International Conference on Information Warfare and Security (ICIW08)*, April 24-25 2008.

[11] W. L. Bahn, L. C. Baird III, and D. Collins, Michael, "Oscillator mismatch and jitter compensation in concurrent codecs," in *IEEE Military Communication Conference (MILCOM08)*, Nov 17-19 2008.

[12] R. Thurimella and L. C. Baird III, *Cryptography for Cyber Security and Defense: Information Encryption and Cyphering*. IGI Global, 2009, chapter title: "Network Security".

[13] L. C. Baird III and W. L. Bahn, "Parallel bbc decoding with little interprocess communication," U. S. Air Force Academy, Academy Center for Cyberspace Research, Tech. Rep. USAFA-TR-2009-ACCR-01, Nov 2009.

[14] ——, "An efficient correlator for implementations of bbc jam resistance," U. S. Air Force Academy, Academy Center for Cyberspace Research, Tech. Rep. USAFA-TR-2009-ACCR-02, Nov 2009.

[15] ——, "An o(log n) running median or running statistic method, for use with bbc jam resistance," U. S. Air Force Academy, Academy Center for Cyberspace Research, Tech. Rep. USAFA-TR-2009-ACCR-03, Nov 2009.

[16] S. Hamilton, "Secure jam resistant key transfer," Masters thesis, Auburn Univeristy, Tech. Rep., May 2008.

[17] M. Kuhr, "An adaptive jam-resistant cross-layer protocol for mobile ad-hoc networks in noisy environments," PhD thesis, Auburn Univeristy, Tech. Rep., May 2009.

[18] D. Sanders, "A single-hop medium access control layer for noisy channels," PhD thesis, Auburn Univeristy, Tech. Rep., August 2009.

[19] S. S. Hamilton and J. A. Hamilton Jr., "A secure jam resistant key transfer : Using the dod cac card to secure a radio link by employing the bbc jam resistant algorithm," in *IEEE Military Communication Conference (MILCOM08)*, Nov 17-19 2008.

[20] *FIPS PUB 180-1 Secure Hash Standard*. National Institute of Standards and Tchnology, 1995.

[21] S. Wolfram, *A New Kind of Science*. Wolfram Media, 2002.