The Glowworm hash: Increased Speed and Security for BBC Unkeyed Jam Resistance

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

Eric Smith
Email: eric@brouhaha.com

Abstract—Jam resistance for omnidirectional wireless networks is an important problem. Existing jam-resistant systems use a secret spreading sequence or secret hop sequence, or some other information that must be kept secret from the jammer. There is currently only one system for achieving such jam resistance without any shared secret: BBC coding. BBC requires the use of a hash function that is fast and secure, but "secure" in a different sense than for standard cryptographic hashes. At MILCOM-2010, the Inchworm hash was presented [1], which is 300 times faster than SHA-1 for this application, and had no security flaws yet known. A variant of it, Inchworm-S, was also presented, that was intended to be more secure. We present an *academic* break of both Inchworm and Inchworm-S, and a *practical* break of the former. The advantage to the attacker is very small, but it is definitely detectable. We also present a new hash algorithm: Glowworm. Glowworm is simpler than Inchworm, and the same speed as Inchworm-S, while being more secure. We mathematically prove that it is immune to the theoretical attack we show for Inchworm and Inchworm-S. We also show empirically that it is immune to our empirical attack on Inchworm, even when the attack algorithm is run for much longer periods. In fact, we show that for our best attack software, it is indistinguishable from SHA-1, MD5, and all five SHA-3 candidates. We also mathematically prove that it has avalanche properties that prevent many other types of internal-state collisions and related attacks. We give a public domain, optimized, portable, C implementation of Glowworm. For incremental hashes as used in BBC codes, it can hash a string of arbitrary length in 11 clock cycles. That is not 11 cycles per bit or 11 cycles per byte. That is 11 cycles to hash the entire string, given that the current string being hashed differs from the last in only an addition or deletion of its last bit. Finally, we discuss our implementation of Glowworm in a Field Programmable Gate Array (FPGA), where it runs in just one clock cycle per string, using only a modest amount of resources. [1]

I. INTRODUCTION

Jam resistance is an important problem for omnidirectional radios. In most systems, this requires some form of shared secret, such as a chip sequence or hop sequence that is kept secret from the jammer. For large radio networks, especially ad hoc networks, this can cause a difficult key management problem. Only one system currently achieves jam resistance without a shared secret: the BBC (Baird, Bahn, Collins) concurrent code [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. The BBC signal is transmitted with no shared secret. The receiver then decodes it using the process in Fig. 1.

The radio receives the packet shown at the top of the figure, which in this example has 18 positions, of which 5 are *marks* (shown as gray 1 bits). Each mark might be a pulse of broadband radio noise at a given point in time, where there are 18 possible times shown. Or each mark might be an entire chip sequence, shifted by some amount in time, where there are 18 possible shifts shown. Or each mark might represent energy being broadcast at a single frequency, where there are 18 possible frequencies shown. In every case, the send expends energy to create a mark, and the receiver can then detect the mark.

This packet is decoded by a depth-first search on the tree at the bottom. Each vertex contains a string, to which a 0 is appended for the left child and a 1 for right. A vertex is gray if the hash of its string (shown in the middle) yields an index into the packet at a position with a mark. It is white if there is no mark there.

The decoder performs a depth-first search of the tree, hashing each vertex and checking for a mark in that position in the packet, to see whether to continue down or to backtrack. The received "message" is any gray string found on the 1000th level of the tree (possibly with some additional filtering).

The jammer must pay an energy cost to broadcast each mark, and the legitimate receiver must pay a computational cost for each vertex in the decode tree. So a successful jamming attack would have few marks in the packet, but

1	0	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

H(101)=0   H(00)=7   H(0)=12
            H(1)=5   H(10)=12

H(1011)=3   H(100)=6   H(1010)=9   H(11)=14
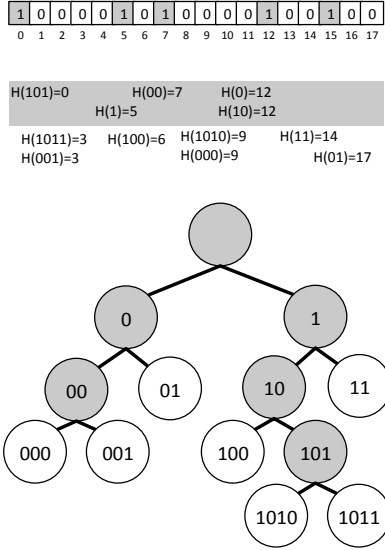H(001)=3   H(000)=9   H(01)=17

Fig. 1. Use of a hash in BBC decoding. The receiver detects the packet at the top, and decodes it by a depth-first search of the tree at the bottom, using the hash values shown in the middle. A string in the tree is defined to be a non-leaf (gray) if its hash returns the index of a position in the packet containing a mark (a gray 1).

many vertices in the tree. Thus the jammer wants many hash collisions. This example shows only 3 collisions, at positions 3, 9, and 12.

The BBC algorithm can use any hash function, but the security of BBC depends on the choice of hash. A *random function* would be good. That is a function chosen at random from the space of all possible functions. It is commonly considered in theory, but impossible to achieve in practice: a computer creating such a random function would require more atoms than exists in the known universe.

A cryptographic hash such as SHA-1 [20] would presumably also be good, because it is designed to be indistinguishable from a random function with any reasonable amount of analysis. But cryptographic hashes are designed to have a different type of security, and so are not optimized for this problem. A hash like SHA-1 has many bits of output, and is designed to make it difficult to find two strings that yield the same output, even when searching for long periods on a supercomputer. The hash for BBC will have a small output of at most 20 or 30 bits, so it is easy to find such collisions. The security for BBC will depend on whether it has the property that it is difficult to find a packet with few marks that results in a large decode tree. So a few collisions are acceptable (and inevitable), but it is bad if there are many collisions that are adjacent in the tree, and so form a rooted subtree of colliding strings.

In addition, BBC decoding consists of doing a depth-first search of the string tree and hashing all the strings in that order. That means that each string to be hashed differs from the last by either the addition or deletion of a bit from the end

of it. It is therefore useful to design a hash to be incremental in its last bit, in order to be a fast hash for BBC.

II. THE INCHWORM HASH

The Inchworm algorithm is shown in Figure 2, and was first published in MILCOM-2010 [1]. It hashes a string bit by bit. To hash the next bit, it retrieves the next word from the buffer (a shift register of 31 words of 64 bits each), and XORs it with one of the two registers R and S. It XORs a constant with the other register. The bit being hashed determines which of the two buffers will be modified.

This approach was used to guarantee that calculating $H(x0)$ and $H(x1)$ could not result in the same values for both R and S, no matter what string x is. Clearly, $H(x0)$ and $H(x1)$ can both result in the same value for R only if the word read from the buffer is equal to C. But C and D are not equal to each other. So it must not be equal to D. So S must not be the same after both $H(x0)$ and $H(x1)$. Similarly, if they cause S to collide, then R won't. So Inchworm effectively makes one kind of internal state collision impossible.

Unfortunately, other forms of collision may still be possible. It might be possible for the calculation of $H(x01)$ and $H(x10)$ to result in the same R and S register values. That could happen if two consecutive words in the buffer happened to have just the right relationship to achieve that. If two machines are hashing in parallel, their R and S registers would become unequal after hashing $H(x0)$ and $H(x1)$, but then would become equal again after $H(x01)$ and $H(x10)$. The singe iteration where they are different would force one word of the buffer to differ between the two machines. But after R and S become the same again, the two machines could remain in sync for the next 30 iterations, before they wrapped back around to that spot in the buffer. Thus we would have $H(x01y) = H(x10y)$ for any string y that is of length 30 or less. This would mean that in the decode tree, an entire subtree of 2^{30} vertices would have the same hash values as another subtree of 2^{30} vertices. So the attacker could force the receiver to do twice as much computation for a given amount of energy expenditure on the attacker's part.

That is a problem. A BBC hash needs to avoid such collisions of its internal state. The Glowworm hash was designed to make such collisions impossible.

III. THE GLOWWORM HASH

The Glowworm hash is designed for BBC decoding, so it needs to be fast for incremental hashing. It is used to hash each string in a long sequence of strings, where each string is identical to the previous except for a single bit that is added to or deleted from the end of the string. So, as the string is growing, the system is fed one bit at a time, using each one to update its internal state, and outputting a hash after each bit. When the string is shrinking, it needs to undo those growth operations. So we need an invertible operation.

For invertibility, an obvious candidate would be an unbalanced Feistel operating on the buffer. The following is such a system, where n is the length of the last string hashed, and s is
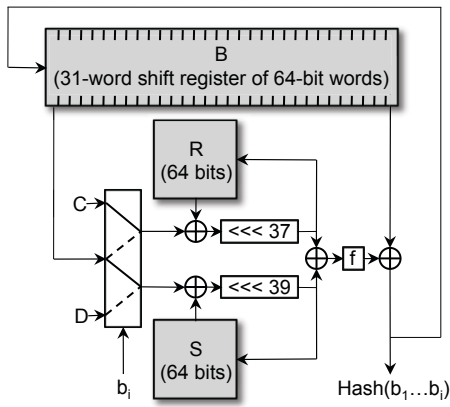
Fig. 2. The Inchworm hash. The $+$ in a circle is an XOR. All arrows carry 64-bit words (except the one from b_i, which is a single bit). The switch connects along either solid or dotted lines, depending on the bit being hashed. The bits of the string are consumed one by one, and each time a word is modified in the big shift register. Also, two small registers R and S are updated, one by XORing with a word from the buffer, and one by XORing with a constant. The choice of register is controlled by the bit being hashed.

Fig. 3. The Glowworm hash. The $+$ in a circle is XOR, and the v in a circle is OR. All arrows carry 64-bit values. The $>>$ symbol is a right shift, and $<<$ is left.

the ring buffer used for the internal state. An element $s[i]$ is a 64-bit word, which is the ith element of that buffer, wrapping around if necessary, when the index i becomes too large for the buffer. The buffer is 32 words, for a total of 2096 bits. This is good, because the Small Internal State Theorem [21] says that the size of the buffer must be a large fraction of the length of the longest string that will be hashed, and should be twice that to avoid birthday paradox attacks. Since BBC generally hashes strings up to about 1000 bits, the internal state should be on the order of 2000 bits. So 2096 is a good size. In addition, the mathematical analysis in the next section shows that having more than 32 elements in the buffer could be a problem, because certain types of collisions are only guaranteed to be avoided for 32 iterations. So a buffer size of 32 elements is ideal on multiple grounds.

The Glowworm algorithm is shown in Figure 3. It is simpler than Inchworm, using only the buffer, without the two extra registers. In fact, it is so much simpler, that the figure actually shows the entire contents of the f function, which is just shown as a single block in the Inchworm diagram.

The nonlinear f function for Glowworm is similar to that in Inchworm-S, but uses shifts rather than rotates. This makes it easier to port, since compilers such as GCC sometime fail to compile rotations to a single rotation instruction, depending on the surrounding code. Shifts always compile correctly. More importantly, the use of shifts allows the propagation of differences to be better controlled, which is necessary to achieve the properties that are mathematically proved in the next section.

The algorithm for hashing the next bit will be very simple. It will grab the current element from the shift register, apply an arbitrary f function to it and the bit to hash, and XOR the result into the next buffer element, and return that same result as the hash. The return statement at the end returns a 64-bit value, but only the 32 least significant bits should be used as
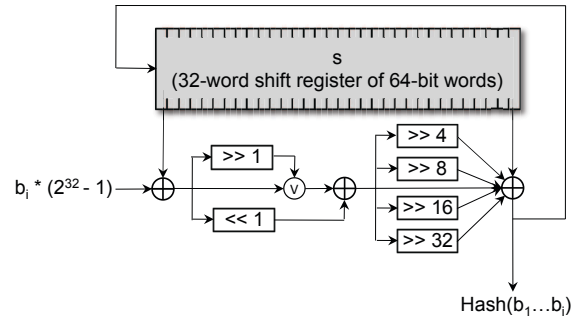
the hash value. If fewer than 32 bits are needed, then only the least significant bits should be taken. If it is desired for the hash to output an integer in the range from 0 to $N-1$ for some $N < 2^{32}$, then the hash is defined to be the 64-bit returned value, taken modulo N.

```
//If the last hash was of a string of length n,
//then concatenate bit b to the end,
//and return the hash of the resulting string.
GlowwormAdd(b):
    t = s[n]
    L++
    s[n] ^= f(b, t)
    Return s[n]
```

This pseudocode uses C notation, so $L++$ increments L, and $x \,\hat{=}\, y$ changes variable x to contain the bitwise XOR of x and y. All variables and buffer elements are 64-bit unsigned integers, except for b, which is a single bit. As the string grows, this walks through the buffer, changing one word after each bit is received. A given word is changed by XORing it with a function f of both the previous word and of the current bit being added to the string. Note that this is reversible given the bit, even if the f function is not itself reversible. So the following will reverse it, regardless of the choice of f.

```
//If the last hash was of a string of length n,
//ending with bit b, then delete that bit,
//and return the hash of the resulting string.
GlowwormDel(b):
    n--
    GlowwormAdd(b)
    n--
    Return s[n]
```

This works because GlowwormAdd changes one word by XORing it with the output of f, and GlowwormDel simply XORs the same word with that same value, thus canceling out the original XOR. Note that the caller must store the string being hashed, in order to know what bit to pass to GlowwormDel. Glowworm itself stores only the length, n.

This is much simpler than the Inchworm algorithm. In Inchworm, there were two registers in addition to the big buffer. Each register was constantly rotating at a different rate, and data from the buffer was constantly being transferred to the registers. In Glowworm, all of that is gone. There's simply a single buffer, and the algorithm merely walks around the buffer, changing each word as it goes.

The Feistel structure shown above can be very strong or very weak, depending on the choice of f function. For Glowworm, we define that function as follows.

```
f(b, t):
    t ^= b * 0xffffffff
    t = (t | (t>>1)) ^ (t<<1)
    t ^= (t>>4) ^ (t>>8) ^ (t>>16) ^ (t>>32)
    return t;
```

Given the bit b, the f function is almost invertible. The first line is its own inverse. Executing it a second time would undo the first execution. The third line is almost its own inverse. Its execution can be undone by executing a line that is identical, except the shifts $\{8, 16, 32\}$ are replaced with $\{12, 28, 60\}$, respectively.

The second line is almost invertible. Given the new value of t after the line is executed, it is impossible to know the most-significant bit of t before it was executed. But given that bit, the other 63 bits can be recovered. So it loses only one bit of information on each execution, and always has exactly two preimages. It would have been easy to make this line completely invertible, if Glowworm had been defined with this as its second line, instead:

```
    t = (((t & (t>>1)) ^ (t<<1)) | 1) ^ (t>>63)
```

That would have been invertible, and doesn't look any less secure. But it would have been slightly slower. And there is no obvious advantage to making f itself invertible, since Glowworm as a whole is invertible regardless of whether f is invertible. So Glowworm was actually defined with the simpler, almost-invertible version of this line.

Glowworm is very simple. It was defined above in just a few lines of pseudocode, and it has no internal state other than a simple buffer. And it is almost entirely linear. The "|" symbol in the second line of f is a bitwise OR, and is the only nonlinear operation in the entire Glowworm algorithm. The multiplication in the first line is linear, because it is actually just a way to make 32 copies of the bit b. The incrementing and decrementing of n are actually linear, because they are just ways to implement rotations in a shift register. So that single OR remains the only nonlinear operation in the entire hash algorithm. This aids in analyzing it, and makes it possible to prove the theorem in the next section.

A fast, portable C implementation of Glowworm is given in figure 4.

IV. MATHEMATICAL ANALYSIS OF GLOWWORM

Glowworm has security properties that can be mathematically proved, that prevent the attacks that were possible on Inchworm. The following theorem shows both avalanche and

```
// Glowworm - A hash for use with BBC codes
// April 2012, Version 1.0 - this is public domain
// Leemon Baird, Leemon@Leemon.com
//
// Call glowwormInit once, which returns the hash of the
// empty string, which should equal CHECKVALUE. Repeatedly
// call AddBit to add a new bit to the end, and return the
// hash of the resulting string.  DelBit deletes the last
// bit, and must be passed the last bit of the most recent
// string hashed. The macros should be passed these:
//     uint64 s[32];    //buffer
//     static uint64 n; //current string length
//     uint64 t, i, h;  //temporary
//     const uint64 CHECKVALUE = 0xCCA4220FC78D45E0;

typedef unsigned long long uint64; //64-bit unsigned int

#define glowwormAddBit(b,s,n,t) (                    \
    t  = s[n % 32] ^ ((b) ? 0xffffffff : 0),         \
    t  = (t|(t>>1)) ^ (t<<1),                        \
    t ^= (t>>4) ^ (t>>8) ^ (t>>16) ^ (t>>32),        \
    n++,                                             \
    s[n % 32] ^= t                                   \
)

#define glowwormDelBit(b,s,n,t) (                    \
    n--,                                             \
    glowwormAddBit(b,s,n,t),                         \
    n--,                                             \
    s[n % 32]                                        \
)

#define glowwormInit(s,n,t,i,h) {                    \
    h = 1;                                           \
    n = 0;                                           \
    for (i=0; i<32; i++)                             \
        s[i]=0;                                      \
    for (i=0; i<4096; i++)                           \
        h=glowwormAddBit(h & 1L,s,n,t);             \
    n = 0;                                           \
}
```

Fig. 4. C implementation of the Glowworm hash.

anti-collision properties. It refers to the *internal state*, which is all 2096 bits in the buffer. It also refers to the *fast internal state*, which is the 64-bit word that changes on each iteration, which is also the output of the hash on each iteration.

Theorem 1: If two machines run Glowworm in parallel on two strings, their internal states will be identical until they reach the first bit where the two strings differ. Their internal states will then continue to differ for 64 iterations, regardless of what the two strings contain after that point. The fast internal states will differ for at least the first half of that (32 iterations), at the end of which, each of the 32 words in the buffer of one machine will differ from the corresponding word in the buffer of the other machine.

Proof: First, consider the second line of f, run on two machines which at that point have different values for t. Suppose the two t registers differ in one or more bits, the most significant of which is in position p. Then after executing the second line, they are now guaranteed to have a most significant differing bit in position $p + 1$ (where the least significant bit is position 0, and the most significant is position 63). In other words, differences ripple to the left (toward the more significant bits), moving one bit position per iteration.

This rippling effect is because all of the shifts in f are right shifts, except for the single left shift in the second line. In that

case, t is shifted left by only a single bit position, and it is then XORed into t. So as long as the two machines have identical words coming in from the buffer, the f function will ensure that the leftmost difference will move left at a speed of one bit per iteration.

This ensures that as new bits are hashed, their effects slowly ripple leftward into more significant positions, and the effect of any given bit that is hashed lasts for at least 32 steps. This guarantees that if two machines are hashing in parallel, as soon as they get to a differing bit in their respective strings, their fast state will start to differ. And they will continue to differ for at least 32 steps. But the buffer is only 32 words long, so that means that after 32 iterations, every single word in one machine's buffer will differ from the corresponding word in the other machine's buffer. And it will take at least another 32 steps to work its way around and bring all 32 words back into sync. Thus the theorem is proved.

Thus, two short strings, when hashed, cannot cause a state collision. Two long strings could, but only by first causing every word in the buffer to differ, then bringing them back into matching. If the overall effect on the buffer were a random function, then a collision on a 2048-bit buffer would take 1024-bit strings to be hashed. But BBC was designed with the assumption that the strings being hashed would never be more than about 1000 bits long. So such a collision is not a problem for BBC. Other attacks may still be possible on Glowworm. But none are yet known. And the guarantees presented here are much stronger than were available for Inchworm.

V. Empirical analysis of Glowworm

We now propose an algorithm for generating attack packets. We give empirical results that show that this algorithm is able to find attack packets that cause more work for the receiver when using Inchworm than when using Inchworm-S, Glowworm, and standard cryptographic hashes. Therefore, this constitutes a break of Inchworm. Glowworm is immune to this particular attack. We were unable to find an empirical attack that was successful against Glowworm.

The algorithm starts with a packet of all zeros. Then, whenever the packet is less than or equal to one third full of marks, it adds a mark to it, in the location which causes the most growth in the decode tree. Whenever the packet is more than one third full of marks, it removes a mark, choosing the mark that will result in the least shrinkage of the decode tree. The greedy algorithm continues until it reaches a cycle of constantly adding and removing the same mark. At that point, it stops, and outputs the size of the decode tree.

We compared two hashes by comparing the distribution of trees sizes generated by each. When we compared Glowworm and SHA-1 on a million runs each, the Cramer-von Mises statistical test found no statistically-significant difference in their distributions. We found that to be true when comparing any pair of algorithms from the set of Glowworm, SHA-1, Inchworm-S, MD5, and all 5 of the SHA-3 candidates. We therefore conclude that all 9 of these hashes are indistinguishable from a random function, at least as far as could be

determined by this greedy attacker with one million runs each. Although hashes like MD5 are known to have weaknesses for other purposes, none of these have known weaknesses for use in BBC, and this particular test was unable to find any such weaknesses with any of these hashes.

However, the Cramer-von Mises test found that Inchworm was different. It generated packets that required more computation for the receiver to decode. The amount was small, less than one percent, but the difference was statistically significant. Therefore, this is definitely a deviation from the behavior of a random function, and is certainly a break in Inchworm, even though it is only a small break so far. It would not be surprising if it could be converted to a large break.

We also ran the test on 64 variants of Glowworm. Glowworm uses a set of shifts of $\{4,8,16,32\}$ in its linear stage. A variant of Glowworm could be formed by using the shift set $\{1,2,4,8,16,32\}$, or using any subset of that, including the empty set. That gives 64 different algorithms, all of which were tested before choosing one to become Glowworm.

We found that any set of shifts would pass our test, as long as it included the shift by 32, and as long as it included at least two different shifts. In those cases, it was indistinguishable from random. In the other cases, it was bad, and appeared to get worse for fewer shifts and for smaller shifts. So it seems that somewhere between 2 and 6 (inclusive) shifts are needed, and that large shifts are better. Glowworm is therefore defined to be the variant with 4 shifts (midway between 2 and 6), and where the 4 shifts are the 4 largest. In other words, Glowworm was defined to be $\{4,8,16,32\}$, as shown above, as a result of running the attacker on all 64 variants. Shifts other than powers of 2 would likely be good as well, but our current test is unlikely to detect any advantages for them.

VI. Hardware implementation of Glowworm

The simplicity of the structure of the Glowworm hash makes it ideally suited to efficient implementation in a Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC). The resource utilization in an inexpensive FPGA is quite small, and the performance exceeds that of any external memory that would be used to store the BBC packet, thus the hasher will not limit the performance of a hardware BBC decoder.

Glowworm has been implemented in VHDL and synthesized for the Xilinx Spartan 3 and Spartan 6 family FPGAs. The implementation follows directly from the Glowworm block diagram, with only a few details being of particular note.

The implementation consists of a control state machine, a 32 word by 64 bit dual-port RAM for the s shift register, a 32 word by 64 bit ROM for the initialization values for the s shift register, three registers to point to entries in the s array, and combinatorial logic consisting primarily of multiplexers, XOR gates, and OR gates. While the source code uses the multiple instances of shift left and shift right operators, these are all shifts by a fixed bit count, and thus synthesize to direct wiring rather than multiplexers or barrel shifters.

Rather than use the 4096-step initialization process defined in the Glowworm specification, the ROM is used to load the s shift register with the results of the initialization process in only 32 clock cycles. The same technique is expected to be used in efficient software implementations. With some added complexity, the initialization of the hardware implementation could be reduced to a single clock cycle.

The AddBit and DelBit operations each take a single clock cycle, and function identically with the exception of the addresses used to index s. For efficiency, three separate registers are used to index s, containing the values n-1, n, and n+1, with two multiplexers to select which of these counters are used for each of the two address ports of the dual-port RAM.

In order to avoid the incrementer and decrementer for the n registers being in the critical path, they are used in parallel to compute n-2 and n+2, with multiplexers to select the new values to be stored into the n-1, n, and n+1 registers.

VII. CONCLUSION

We have presented a new hash function, the *Glowworm* hash. It is simpler and more portable than the Inchworm that was published in MILCOM-2010, yet it is more secure. We showed both theoretical and empirical breaks for Inchworm that give the attacker a real (though very small) advantage over an ideal hash. Glowworm was designed to eliminate those flaws. And in fact, we mathematically proved that Glowworm is not susceptible to the flaws we showed in Inchworm. If two machines hash two strings, the first differing bit will cause their internal states to differ for at least the next 64 steps, and will eventually cause every word in the buffer of one machine to differ from the other.

In addition to these theoretical results, we showed empirical results with attack software designed to search for good attack packets. It tested Glowworm, SHA-1, MD5, and all 5 of the SHA-3 candidates, as well as 63 variants of Glowworm. Flaws were found in Inchworm and the weaker variants of Glowworm, but no flaws were found in Inchworm-S or Glowworm.

Finally, we implemented Glowworm in VHDL for execution on an FPGA. We found in simulation that it can hash each string in a single clock cycle. Therefore, Glowworm is extremely efficient in both software and hardware. And the bottleneck in speeding up BBC decoding is no longer in the speed of the hash, it is simply in the time to access memory to see whether a mark is present at a given location or not. There is therefore no need to work on speeding up Glowworm, because it now takes such a tiny fraction of the entire run time.

This work used only simple, greedy attack algorithms. It would be good to look for more sophisticated ways of finding attack packets. This would be a good area for future research.

REFERENCES

[1] L. C. Baird III, M. C. Carlisle, and W. L. Bahn, "Unkeyed jam resistance 300 times faster: The inchworm hash," in *MILCOM 2010 - Military Communications Conference*, Oct 2010. [Online]. Available: http://leemon.com/papers/2010bcb.pdf

[2] 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.

[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] 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.