BeepBeep: Embedded Real-Time Encryption

Kevin Driscoll

Honeywell Laboratories, 3660 Technology Drive, Minneapolis, MN 55418, USA
kevin.driscoll@Honeywell.com

Abstract. The BeepBeep algorithm is designed to supply secrecy and integrity for embedded real-time systems. These systems must achieve their required timing performance under all conditions, while operating in a multi-tasking environment with tightly constrained CPU, memory, and bandwidth resources. BeepBeep was designed to be implemented as software on the processors most commonly used for embedded controllers. It uses little program memory, no data memory (its state fits into most processors' register sets), and no inherent message padding (ciphertext is a 1:1 replacement for plaintext). It is significantly faster than existing algorithms (e.g. AES) in this environment and includes mechanisms to support integrity as part of its basic secrecy operation.

1 Motivation and Requirements

Examples of embedded real-time applications requiring security include wireless communications (cell phones, pagers, aircraft, Bluetooth, IEEE 802.11), remote management of control systems for chemical and power plants, distributed management of distribution networks (pipelines and electrical grids, remote meter readers), access and control of remote sites (physical security management, electrical load shedding, medical equipment). Typical real-time cryptography requirements differ significantly from conventional cryptography in a number of ways.

- 1:1 message size: Varying-length byte streams must be encrypted with minimal message expansion, particularly in retrofit applications.
- varying integrity requirements: Detection of message corruption is essential, particularly for actions with serious consequences.
- key agility: Each message can have a different key for each unicast/multicast address, requiring rapid key change after a message header is processed.
- low latency: Input to output delay is more important than throughput.
- low jitter: Processing time for each message packet should be the same. (There is little or no time for per message key scheduling.)
- small memory footprint: Many high-volume cost-sensitive applications use single-chip microcomputers with a total RAM of 128 or 256 bytes.
- security time horizon: "tactical" rather than a "strategic"

– compatibility: Embedded systems tend to be closed communities, with little need to be compatible with the rest of the world

A search of existing cryptographic algorithms failed to find any that met these requirements. The algorithms' problems include large latency and slow speed (particularly for the small messages typical of real-time systems), large data structures, expansion of messages, and significant cost to switch keys in RAM-constrained implementations.

1.1 Deadlines

Real-time cryptography must live within deadlines, typically that repeat with fixed periodicity. Overstepping the deadline frequently is not possible. Only worst-case execution times factor are important; average performance better than worst-case has limited utility. Missed deadlines can cause catastrophic failures in safety critical systems. Most real-time systems are heavily multi-tasked and the time slices allocated to a cryptographic task may be very small.

1.2 Context Switching and State Size

The heavily multi-tasked and interrupt-driven nature of real-time systems, coupled with their tight latency requirements, means that such systems do a lot of context switching. Frequent switching reduces the utility of modern processor cache technology, and can even lead to counter-productive cache thrashing. Indeed, partly for this reason, microprocessors used in real-time systems often have no data cache.

A real-time cryptographic algorithm should be designed to minimize memory access penalties. Ideally, the crypto state of the algorithm should fit within the register set of the target CPUs. But, small 8-bit and 16-bit microcomputers often do not have enough register space to hold the minimum crypto state needed to be secure.

Given the general trend of real-time controllers increasing their word size to 32 bits and with most 32-bit controllers' register sets having sufficient size, it makes sense to size an encryption algorithm's state to fit into as many of the 32-bit microcomputers' register sets as possible and be resigned to the fact that smaller processors will have store part of its state in RAM.

A survey of 32-bit CPUs found that most have at least seven 32-bit registers available to hold crypto state data, when leaving enough other registers to hold the rest of algorithm's state and temporary values. The non-ignorable exception is the Intel x86 family. For the x86 family, use of MMX registers, or of a single on-chip cache line, can provide the same storage. Overall, then, an algorithm's crypto state should not exceed the magical number seven[1] 32-bit words.

[1] With apologies to Miller[8].

1.3 Message Size

Most real-time communication products are designed to minimize energy use, size, weight and cost, while providing an acceptable bit error or message loss rate. When cryptographically-based communications security is included in these products, either as a new design or a retrofit, there is seldom an available budget for cryptographic overhead. Instead, every increase in message size leads to a decrease in the functionality for which the product was purchased.

In some retrofit applications, where correctness of system communications behavior has previously been certified, changes in system timing due to cryptographic expansion of messages is not tolerable. In other networks, where users pay by the byte (e.g., with aircraft or LEO satellites), cryptographic expansion impacts profitability.

All of these situations create a need to minimize message growth due to cryptography. This need is amplified by the very short message sizes of many real-time systems, frequently on the order of 10 bytes, where even small per-message overhead due to cryptography can be a large burden.

For all of the just-stated reasons, a major goal of a real-time cryptographic algorithm design is to minimize or eliminate message expansion. This requirement eliminates the use of block cipher modes such as ECB, which round messages up to the next block size; the chosen cipher must either be a stream cipher, or a block cipher used in a stream cipher mode. The former is typically more efficient.

Stream ciphers also have communications overhead, in the form of an initialization vector (IV). Luckily, most real time communication messages are individually identified through extrinsic or intrinsic means which can be used as the IV without creating any additional overhead.

1.4 Security

Secrecy Most real-time communications have a need for short-term secrecy, to deny an attacker knowledge of current control system state. The need for long-term secrecy in such systems is infrequent, but it does exist. Information of major economic value, such as trade secret process parameters or inventory levels of arbitrage-able commodities, requires long-term secrecy. Such secrecy can always be provided by super-encryption of the information at risk, though this is undesirable. Ideally, a single cipher should meet both short-term and long-term secrecy needs.

Authentication and Integrity Real-time systems usually need to prevent message forgeries and unauthorized message modification. Corrupt control messages can cause disasters directly. Corrupt reports of current state can lead to disasters indirectly. Authentication and integrity can be supported by including predictable values in the (extended) plaintext message. The classical way of doing this is by appending a cryptographic hash of the plaintext to the message. A less computationally costly alternative is possible when the cipher provides suitable feedback of the plaintext (or a derivative text) into subsequent ciphertext, eventually affecting an expected value at the end of the message. In many real-time systems, particularly those involving retrofit or rollover, existing frame check data can be included in the encryption as the predictable postfix integrity value. This can reduce or eliminate message size expansion.

Existing real-time systems often can add cryptography only as new "lump in the cable" hardware. Coupled with latency restrictions, this requires on-the-fly cryptography – secrecy and integrity have to be done in one pass.

Where on-the-fly cryptography isn't needed, a second encryption pass over the message with a different starting point and/or direction, can distribute an integrity check over the entire message. This permits all predictable values within the plaintext to be used for integrity without regard to their location, including data that can do double duty as an IV as well integrity checking.

Real-time systems typically are autonomous and do not accept for encryption and transmission messages from untrusted sources. This precludes many "oracle" and related types of attack.

1.5 Asset Exploitation

Embedded real-time cryptography is a struggle of economics, in which the goal is to make an adversary incur more cost than the effort is worth while not imposing prohibitive cost on authorized users. A design should attempt to include assets which are already available to authorized users in a way that prevents an adversary from exploiting alternate technologies to gain an advantage. The greatest perceived threat is the conversion of a weakness in an algorithm into a workable break by using custom integrated circuit (ICs) or field programmable gate arrays (FPGAs) to greatly speed up trial decodes.[4][1] This suggests a design goal for the algorithm to include elements which are cheap or free when implemented in software on real-time controllers but are expensive to implement in ICs or FPGAs. The latter expense is primarily "silicon area" in terms of gates and routing. This goal can be accomplished by incorporating in the algorithm the use of hardware elements which exist in most real-time control CPUs and require a large number of gates and/or routing. The most obvious big consumer of "silicon" in a CPU is memory (including caches). For reasons given above, memory assets for data storage cannot be cheaply exploited. The next largest silicon element is a fast multiplier. CPUs in real-time controllers have 32-bit arithmetic units, including a multiplier. The multiply speed has been getting faster in each generation of CPUs. This trend is expected to continue.

2 BeepBeep Description

BeepBeep is a new algorithm designed specifically to meet the above requirements. Its main elements (described in detail below) are a 127-bit primitive Linear Feedback Shift Register (LFSR), clock control, non-linear filter, and two stage combiner. The LFSR provides a stream of pseudo-random values. The clock

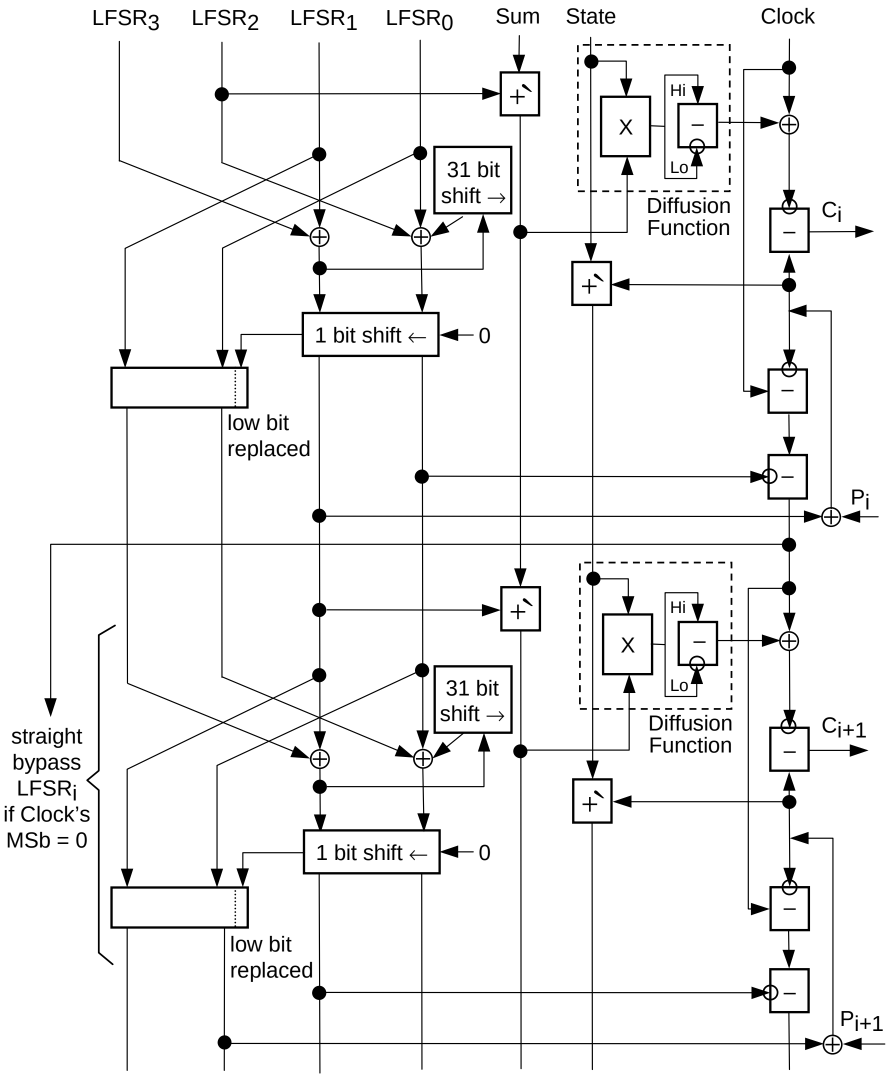

Fig. 1. Encryption Block Diagram

```
#define +'(a, b)   // unsigned ones complement sum
  ( (a + b) < a ? a + b + 1 : a + b )

#define step0_32(ctl)  // if ctl = 1, advance LFSR by 32 bits
  i       = lfsr[3] ^ lfsr[1];  // 32 new LFSR bits
  lfsr[3] = lfsr[3 - ctl];  // shift LFSR by 0 or 32 bits
  lfsr[2] = lfsr[2 - ctl];                     // "
  lfsr[1] = ctl ? lfsr[0] | i >> 31 : lfsr[1];  // "
  lfsr[0] = ctl ?           i <<  1 : lfsr[0];  // "

#define crypt(src, ctl)  // process one word, update variables
  sum = +'(sum, lfsr[src]);
  step0_32(ctl); step0_32(ctl); // if ctl, advance LFSR by 64 bits
  m.f = sum * state;  // entwine and nonlinearize LFSR output bits
  i = (m.h.u - m.h.l) ^ clock;  // " and whiten
  if (encrypt) { *c++ = (j = (*p++       ^ lfsr[3 - src])) - i; }
  else         { *p++ = (j = (*c++ + i)) ^ lfsr[3 - src]; }
  state  = +'(j, state);
  clock -= j + (src == 2 ? lfsr[0] : lfsr[2]);

void BeepBeep(lfsr, clock, sum, state, bytes, p, c, encrypt)
  uint32 lfsr[4]; // left-justified 127-bit LFSR; 128 key bits
  uint32 clock;   // LFSR clocking control; 5th word of key
  uint32 sum;     // running sum of LFSR ouput; 6th word of key
  uint32 state;   // filter state feedback; 7th word of key
  sint32 bytes;   // number of bytes in message
  uint32 *p, *c;  // plaintext and ciphertext word pointers
  char   encrypt;
{
  uint32 i, j;   // short term temporaries
  union { struct {uint32 u, l;} h;  // (u)pper and (l)ower (h)alves
                 uint64 f; }   m;  // of (f)ull 64-bit (m)ultiply

  // Optional initialization goes here

  if (big_endian)
     (encrypt ? *p : *c)[(bytes+3)/4] >>>= (4 - bytes % 4) * 8;
  for (; bytes > 4; bytes -= 8) { crypt(2, 1);  // 8 bytes per loop
                                  crypt(1, clock >> 31); }
  if (bytes > 0)                  crypt(2, 1);  // 1 to 4 bytes left
  if (big_endian)
     (encrypt ? *p : *c)[-1] <<<= (4 - bytes % 4) * 8;
}
```

Fig. 2. BeepBeep Pseudo-C Code

control and non-linear filter protect against known LFSR attacks. The two stage combiner mixes the algorithm's "text" input with the nonlinear filter output and a word from the LFSR to produce the algorithm's final output. A block diagram of BeepBeep's main loop is shown in Fig. 1 and pseudo C-code for BeepBeep is shown in Fig. 2. For simplicity, this pseudo C-code assumes the output buffer is a multiple of 8 bytes and BeepBeep's execution will change bytes in this buffer following the last byte of the message for messages sizes not a multiple of 8.

The symbol +' in the block diagram and the +' routine in the pseudo-C code means unsigned ones complement addition. Most CPUs today use twos complement addition. Conversion to ones complement is done by wrapping the carry out of the most significant bit of a twos complement sum back into its least significant bit. The carry wrap is done using a "with carry" variant of add (which exists on all twos complement CPUs) on the immediately following add to the sum. If the next operation isn't an add to the sum, a instruction must be inserted which adds a (twos complement) zero to the sum. Thus, the +' subroutine uses only one or two machine instructions. Ones complement format has two zeros (+0 and -0). As used here, +0 is not possible if either input is non-zero. A ones complement -0 is a -1 when used in twos complement operations.

2.1 Crypto State

BeepBeep uses seven 32-bit words as its crypto state, which is held in three 32-bit variables (*clock*, *sum*, and *state*) and one array of four 32-bit elements (*lfsr*). BeepBeep's key is simply its entire initial crypto state. The size of the key is be larger than the security of the algorithm. The "excess" key size is used to speed up the initialization of the key generator when key changes are made.

IV methods are discussed in the Initialization section following the description of BeepBeep's main processing loop. In the following description, the term "processed" means that all operations required to produce an output (ciphertext for encryption and plaintext for description) have been performed.

2.2 Clock Controlled Linear Feedback Shift Register

The polynomial for BeepBeep's LFSR is $x^{127}+x^{63}+1$. The LFSR is left-justified in four 32-bit words (*lfsr[3]* down to *lfsr[0]*) and is left shifted. It operates in Tausworthe (full words) fashion, producing three 32-bit output words (the right-most three words). The right-most bit of the right-most word contains one bit of key before the first LFSR shift and is zero thereafter. The LFSR advances 64 bits iff the most significant bit (MSb) of the *clock* variable is a one or an even number of text words have been processed.

LFSR outputs are used for inputs to three functions: non-linear filter, clocking, and two stage combiner. Which outputs are used for which function were chosen to avoid the reuse of any output value for the same function when clock control does not advance the LFSR, to avoid the use of LFSR[0] where knowledge of it LSb being zero would be a weakness, and to allow efficient implementation on 64 bit CPUs.

After each text word is processed, the autokey feedback (ciphertext + i) is subtracted from *clock*. If an even number of words have been processed, *lfsr[0]* is subtracted from *clock*, else *lfsr[2]* is subtracted from *clock*. This provides an autokey influence on LFSR clocking.

2.3 Nonlinear Filter with State and Diffusion Function

Before each text word is processed, one of the LFSR outputs is added to *sum* using ones complement addition. If an even number of words have been processed, *lfsr[2]* is added, else *lfsr[1]* is added. Then *sum* and *state* are multiplied together. After the multiplication, the lower half of the product is subtracted from the upper half. This multiply and subtract function provides most of BeepBeep's diffusion. The diffusion function output is then XORed with *clock* to create *i*.

After each text word is processed, the autokey feedback (ciphertext + i) is added to *state* using ones complement addition.

2.4 Two Stage Combiner

The output of the filter (i) goes to the final section of this algorithm, which is a two stage combiner.

For encryption, the plaintext is first combined with an LFSR word using XOR and then the result is combined with the nonlinear filter's output (i) using twos complement subtraction. The result of this operation is the ciphertext. If an even number of words have been processed, *lfsr[1]* is used, else *lfsr[2]* is used for the XOR.

For decryption, the order of the combiner's two operations is reversed and addition is used instead of subtraction.

2.5 Initialization

BeepBeep currently has five options for performing the IV function, depending on on system requirements and available resources. The first option is the traditional explicit IV value prepended to each message. The second option uses an implicit value available in most real-time systems, such as frame sequence number or system time. For these two options, the *state* variable is initialized with the IV value instead of key. (Such systems use only 192 bits of key.) Fig. 3 shows the method for incorporating this IV into the key.

The IV (held in *state*) is added with *sum* to form a value x, which is transformed via the bijective polynomial $2x^2 + x$. The lower 32 bits of the resulting value ($m.f$) is then added into *clock* and subtracted from *lfsr[2]* both using twos complement arithmetic. The $m.f$ value is then circularly rotated left by 16 bits. The result is added with the message size (in *bytes*) to create a new *state* value. This *state* value is added to *lfsr[1]* using ones complement addition and ORed into sum.

Initialization or key distribution must ensure a non-zero starting value for at least one word of the LFSR and preferably for *sum* and *state*. At the beginning

```
if (bytes < 1) {          // used only for new key or IV
  bytes    = - bytes;
  m.f      = (state + lfsr[3]) * (2 * (state + lfsr[3]) + 1);
  clock   += m.h.l;   // mix in IV changes
  lfsr[2] -= m.h.l;   // mix in IV changes
  m.h.l <<<= 16;      // circularly left shift by 16 bits
  state    = +‘(m.h.l, bytes);   // +‘ ensures state is not +0
  lfsr[1]  = +‘(lfsr[1], state); // ensure LFSR is not +0, vary with IV
  sum     |= state; }            // ensure sum  is not +0, vary with IV
```

Fig. 3. Initialization example

of the algorithm, the only variable that is known to not be zero is *bytes* (the count of bytes remaining to be processed). This fact is exploited in the last three lines of this initialization option to create the desired non-zero values. The OR function is used for *sum* because it is a single instruction where the ones complement addition requires two instructions. The downside of the OR function is that its result is too rich in one bits. This bias is not a problem because the next operation on the variable *sum* will always be a ones complement addition with one of the LFSR words. The IV is added into *lfsr[1]* because its change diffuses the fastest into the other LFSR words.

The third IV option is to just prepend truly random data to the plaintext before encryption and discarded it after decryption. Because BeepBeep uses an autokey, this data performs the role of an IV (without revealing its actual value). This option creates only a small variation in the LFSR's state, which could lead to related key attacks. Thus, this option should only be used where code space is extremely tight on a device that does decryption only and the random data can be made large relative the actual message data.

The fourth IV option is to derive an IV from the message itself. For applications that can't use either of the above IV options, a "hash IV" can be used. To use this, a one-way hash of the message's tail is done. The resulting hash is added to the message's head just for encryption (discarded before the message is transmitted). The combination of hashing and BeepBeep's forward diffusion causes all bits of the message's tail to be diffused throughout the whole message. The "tail" could be the whole message, if the operation starts at the middle of the message and wraps around, ending back at the middle. Because BeepBeep is faster than known hash algorithms, it can be used to create the hash and pre-encrypt the message at the same time. This effectively converts BeepBeep into a variable sized block cipher (no padding). This is similar to the old idea of using an autokeyed cipher to make two passes over a message (one pass in each direction), which also could be used. Of course, using an IV derived from the message body means that a repetition of a message is detectable by an adversary.

The fifth IV option is to use crypto-state carry over between messages. This can be used when BeepBeep is implemented on a reliable message delivery service, which guarantees in-order and error-free reception (as is the case in many real-time systems). This option can be viewed as all messages being just packets of one large virtual message. The initial key is never reused; so, no IV is needed.

3 Design Rational for Performance

3.1 Linear Feedback Shift Register

An LFSR may seem like an odd choice because LFSRs are notoriously slow in software. This problem is solved by using a trinomial with the taps spaced exactly a multiple of a word-width apart. Using word-wide operations creates a variant of a Tausworthe generator that can produce a word of new bits with fewer instructions than is used to produce one bit in typical software LFSRs. This trick increases the LFSR's speed by over 60 times.

BeepBeep's loop encrypts two text words per iteration, each word using different LFSR outputs. The selection can be virtual, with no software execution cost, by the loop being a 2x unroll of basic 32-bit encryption.

3.2 Clock Control

The minimum possible clock control scheme is self-decimation using one bit from the LFSR as the clock control. But, this scheme has been successfully attacked for even bit-wise clocking. BeepBeep's 64-bit step size would be even weaker. By adding just one instruction, a stronger clocking mechanism can be built. BeepBeep subtracts one of the LFSR's words from a running difference (*clock*). On top of this, BeepBeep includes an autokey feedback into the clock control using just one more instruction.

The most significant bit of *clock* is used as the clocking control to allow efficient implementation on most CPU types. Implementing clock control using branching instructions is very slow on most modern high performance CPUs. This is because the direction taken at each branch will be unpredictable. Unpredicted branches usually cause pipeline flushes and refills. Many CPU types (such as ARM, MIPS, and Pentium) have conditional instructions. For these CPUs, the new value of an LFSR word can be conditionally moved into the register that holds the old value, based on the sign of *clock*. Of all the bits in a register, the sign bit is the one that is universally the easiest to test. For CPUs without conditional instructions, the following trick can be used. First, convert the sign bit to a full boolean word by doing an arithmetic right shift by 31 bits. Then, replace the conditional expression "`ctl ? new : old`" with the logic expression "`(ctl & (new XOR old)) XOR old`" for each LFSR word.

To further reduce the performance cost of clock control and to increase the "decimation rate" (if that concept is even applicable to a Tausworthe generator where three of the four LFSR words are used for something in the remainder of the algorithm), the clock control is applied only to the second text word of each loop; the LFSR is always advanced for the first word.

The LFSR words used for the *clock* subtrahend are *lfsr[0]* and *lfsr[2]*. Because the least significant bit (LSb) of *lfsr[0]* is always 0 (required by the fast LFSR trick), it cannot be used as the LFSR value used for the ciphertext XOR. To balance LFSR word usage, *lfsr[0]* is used here where knowledge that the LSb is 0 does not create a possible weakness. The other subtrahend variant is *lfsr[2]* because it is easy to access on a hybrid 32/64 bit CPU.

4 Design Rationale for Security

The best known attacks against BeepBeep currently take on the order of 2^{96} work. However, this is the result of limited analysis. The remainder of this section describes some of BeepBeep's security considerations.

4.1 Clock Controlled Linear Feedback Shift Register

This LFSR was chosen to get a keystream sequence which is long enough to effectively never repeat and has known good statistics while using a minimum of storage resources. But, the Tausworthe speed-up trick's use of a trinomial exacerbates the LFSR's vulnerability to well known attacks. BeepBeep's defense against these attacks include clock control and a nonlinear filter with state. Other defenses (such as those using multiple LFSRs) were found to be too expensive to implement in real-time software. The LFSR's clock control is anemic. It adds only 1/2 bit of uncertainty for each 32 bits of text. This could lead to an attack by exhaustive enumeration. But, this mechanism is designed only to cover any weaknesses that may be found in the nonlinear filter which require large amounts of text to be successful. If an attack against the nonlinear filter gains less than 1/2 bit of information per word encrypted, this clocking mechanism may defeat that attack.

The clock control is of the stop-and-go type, which has known attacks. The LFSR output selection covers this by not reusing the same value for the same function whenever the LFSR is stopped.

Both *clock* and the *sum* variables act as integrators of the LFSR outputs. This stops the majority of attacks against clock controlled LFSRs, which assume the current crypto state is a function just of the number of deletions that have occurred. With the integrators, the current crypto state is dependent not only on the number of deletions, but on their specific history as well.

The author is unaware of any attacks against an LFSR of this size and has both clock control and a nonlinear filter with state.

4.2 Nonlinear Filter with State

One of desired characteristics of a nonlinear filter is a high nonlinear order. The nonlinearity (in GF_2) of this filter begins with the carry chain in the ones complement additions. Each bit of each sum is 33rd order (as opposed to a twos complement sum which would be 32nd order in its most significant bit and decreasing one order for each lesser significant bit). Each bit of each partial product in the multiply is 66th order. The carry chain for the multiply partial product additions and the borrow chain in the subtraction of the full product halves, complete the 128th order of the filter. This high nonlinear order makes higher order differential analysis infeasible (viewing the filter with feedback as a round in a 32-bit block cipher, ignoring the fact that the “key” changes for every block).[7]

The nonlinear filter with state has three design goals beyond those normally seen in a filter generator. The first is to exploit the use of the 32-bit multiplier that exists on almost all real-time controller CPUs. The second is to maintain real-time performance by working on full 32-bit words. The third is to provide forward text diffusion as part of the autokey mechanism.

The autokey feedback to the filter is first added into *state*. Because of the wrap-around carry, any bit change in the feedback can affect any bit in the result, although with decreasing probably along the carry propagation chains. Each bit of *state* is paired with each bit of *sum* in the multiply’s partial products. Given that *sum* is nearly uniform (except that it can’t be zero), each bit you flip in the ciphertext has about a 50% chance of affecting any bit in the filter output, and thus also any succeeding bit in the remainder of the message. This is one of BeepBeep’s mechanisms for providing integrity.

Because the multiply filter is nonlinear and one-way, including its output makes recovery of the previous states difficult.

The use of full words means each LFSR input to the filter is decimated by a factor of 32, even if there weren’t the additional decimation from clock controlled LFSR stepping. That is, without clock control, each bit position of an LFSR output word would “see” an LFSR sequence decimated by 32. This means that current nonlinear filter analyses (such as [3]) don’t hold here.

As with any LFSR filter, the output should be uniformly distributed. Without the ones complement addition on the inputs to the multiply, a zero output would be much more frequent than the mean output frequency. To solve this problem, the ones complement running sums are used. These sums are never zero if initialized to a non-zero value. Even with this “correction”, there is still some bias. This “multiplication followed by subtracting the product halves” has a distribution which is closely related to the factorization of $2^{32} + 1$ (641 and 6700417). That is, values which are multiples of these factors will occur more often than the average. The XOR of *clock* into the feedback path around the multiplier “whitens” the output and prevents any possible short cycles in the multiply’s feedback path.

DIEHARD (http://stat.fsu.edu/~geo/diehard.html) showed no problems, even when parts of BeepBeep disabled (e.g. keying the LFSR to zero, forcing *state* to zero).

Ignoring carry and borrow, each of the filter’s 32 output bits is a perfect nonlinear function of all its 64 input bits. The effect of carry and borrow on nonlinearity and correlation has not been analyzed.

4.3 Two Stage Combiner

This two stage combiner is used for the following reasons: (1) Addition and subtraction provide some lateral plaintext diffusion. (2) Using non-associative operations provides some integrity protection. With a simple XOR combiner (or any linear combiner), an adversary knowing a plaintext can manipulate the ciphertext bits to make the plaintext resulting from decryption be anything the adversary wants. But, not with a two stage combiner having non-associative functions; the most significant bit of each word is the only bit vulnerable. This is the lessor of BeepBeep's mechanisms for providing integrity protection.

4.4 Keying and IV

BeepBeep uses seven words (224 bits) for keying, or 192 bits if *state* is used as an IV. Some of these keys are so weak as to be illegal to use. These are the keys which make the LFSR all zero. The IV initialization described above prevents this. Therefore, any bit pattern can be used for keying BeepBeep with this initialization.

An autokey function is included to compensate for the very small amount of crypto state that can be held and to provide forward diffusion as part of the integrity protection. As with any autokey, adaptive chosen plaintext attacks are a concern. Such attacks are not possible for most applications in the intended domain. For those applications where such attacks are possible, the implementation should not allow messages to be sent such that they are associated with an IV that has known properties.

While BeepBeep has several IV options, ignoring the requirement for an IV is not one of them. Without an IV, BeepBeep can be the subject of "walking one" chosen plaintext attack. With just 32 one-word messages, the lower 31 bits of i and *lfsr[1]* and be found. The attack can proceed to find all of the LFSR and some bits of *clock*. An adaptive chosen attack needs only 32 messages minus the Hamming weight of the lower 30 bits of i.

4.5 Integrity

During both encryption and decryption, the middle value of the two-stage combiner (ciphertext + i) is fed back into *state* and *clock*. This autokey mechanism propagates errors forward to provide integrity. The *state* and *clock* variables provide only 64 bits of change propagation through the message. But, the most significant bit of *clock* controls the state of the LFSR and *sum*, which adds another 127 + 32 bits to the error propagation state. Information is accumulated into these latter 159 bits at a rate of 0.5 bits per word encrypted.

Integrity loss is detected by checking a known value (check data) at the end of a message after decryption. For most real-time communication, check data is already used in messages to detect naturally occurring errors. For messages without existing check data or if the size of the check data is too small for integrity checking, additional check data has to be appended to a message prior to encryption. Given the high diffusion in the feedback loop, any change in the ciphertext will have a 50% change of affecting each bit of the check data.

The two-stage combiner is another integrity mechanism. Because the operations are not associative, this is not equivalent to a simple additive combiner and the typical integrity attacks do not work. The "lateral diffusion" of the 32-bit twos complement addition is hidden by the 32-bit XOR super-encipherment. This leaves only the most significant bit vulnerable to attack. Given complete knowledge of a message's plaintext and ciphertext, an attacker still cannot manipulate the other decrypted plaintext bits in a word to be a value of his choosing, even if the autokey mechanism weren't used. Thus, BeepBeep has double integrity coverage for most of a message's bits.

Via the autokey feedbacks to *state* and *clock*, any change in a message will eventually affects the entire crypto state. Because most modern communications systems and virtually all communications in real-time control have error detection or correction schemes, which accept messages only if they are error free, the historic concern of plain-fed autokey propagating errors is rarely applicable today.

The autokey eventually has the enciphered text affecting all bits of the crypto state, which includes the LFSR's 127 bits, the filter's state of 64 bits and the clock control's 32 bits. All of these are interconnected with multiple paths to prevent divide and conquer attacks.

5 Applications and Performance

BeepBeep is being included in several product developments. One is to encrypt radio communications with commercial aircraft. Another is the remote control of buildings' safety, security and other automation functions, including meter reading, load shedding, and other gas and electric network management functions.

BeepBeep was first implemented on a Pentium II in assembly. Encryption took about 6.5 clocks per byte. A hand analysis of the assembly code showed it should have taken about 4.4 clocks. The reason for the difference is not known, but the most likely suspect is costly cache misses while reading in the plaintext on Windows NT. Start up time was under 100 clock cycles. The code space was 460 bytes each for encryption and decryption (they were coded separately), with the main loop being 184 bytes. The algorithm's entire data state was held in the Pentium's registers (including MMX).

One of the remote control application is interesting because its CPU is only an 8/16-bit hybrid. But, it is a typical heavily multi-tasked embedded control system. The requirements were that in the residual approximately 50 bytes of RAM and 1,638 bytes of ROM, the "security layer" of the protocol stack had to provide secrecy, integrity, authentication, and key management while consuming minimal communication bandwidth (including idle/turnaround time). Rijndael[2] exceeded the memory limits just trying to do secrecy;[6] XTEA (tean)[10] exceeded the limits when simple integrity was added; Skipjack[9] was better than

XTEA for RAM but not ROM. Surprisingly, a BeepBeep based solution fit into 28 bytes of RAM and 1,628 bytes of ROM (954 of it for BeepBeep) even though BeepBeep's 32 bit operations had to be synthesized out of 8 and 16 bit instructions. This application also had the constraints of minimizing bandwidth and execution time because the communication rate can be slow (2400 baud) and some customers are charged for communication time.

6 Conclusion

The need for an encryption algorithm designed specifically embedded real-time systems has been identified. An algorithm to meet the unique requirements for these systems has been designed and is being fielded. This algorithm exceeds the performance of other algorithms in several areas including memory, time (particularly latency and jitter), and message size.

References

1. Bond M., Clayton R.: Extracting a 3DES key from an IBM 4758 http://www.cl.cam.ac.uk/~rnc1/descrack/
2. Daemen, J., Rijmen, V.: AES Proposal: Rijndael. AES Submission. (June 1998)
3. Dichtl, M.: On Non-linear Filter Generators. FSE '97, Lecture Notes in Computer Science, Vol. 1267. Springer-Verlag, Berlin Heidelberg New York (1997) 103-106
4. Electronic Frontier Foundation: Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design. 1st Edition O'Reilly & Associates, Sebastopol CA (July 1998)
5. Gollmann, D., Chambers W.: Clock-Controlled Shift Registers: A Review. IEEE Journal on Selected Areas in Communications. (1989) 7: 525-533.
6. Keating, G.: Performance analysis of AES candidates on the 6805 CPU core. Proceedings of The Second AES Candidate Conference. (1999) 109-114. http://www.ozemail.com.au/~geoffk/aes-6805/paper.pdf
7. Knudsen, L. R.: The interpolation attack on block ciphers. FSE '95, Lecture Notes in Computer Science, Vol. 1008. Springer-Verlag, Berlin Heidelberg New York (1995) 196-211
8. Miller, G. A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychology Review, American Psychological Association Inc. **63** No 2. (1956)
9. United States National Security Agency: Skipjack and KEA algorithm specifications, Version 2.0. (29 May 1998) http://csrc.nist.gov/encryption/skipjack/skipjack.pdf
10. Needham, R., Wheeler, D.: Tea Extensions. Draft technical report, Computer Laboratory, University of Cambridge. (October 1997) http://www.ftp.cl.cam.ac.uk/ftp/users/djw3/xtea.ps