1 Cover Sheet

Name of the submission: Bleep64

Name of the submitter:

Kevin R. Driscoll

Kevin.Driscoll@Honeywell.com

+1 763-954-2198

Honeywell International, Inc.

12001 State Highway 55

Mail Stop MN14-4C40

Plymouth

MN

55441

USA

2 Bleep64 Description

Bleep64 is a slight improvement over its predecessor (BeepBeep) which was published in FSE2002. Bleep64 retains the basic structure of its predecessor and all of its innovations. The improvements include being slightly smaller, slightly faster, and having better avalanche. The native key size also has been reduced from 223 bits to 128 (optionally and natively 192) bits. Nonces can be 32, 64, or 96 bits. Tag lengths can be 32 to 192 bits. For this Bleep64 submission, key length is 128 bits, nonce length is 96 bits, and tag length is 64 bits.

These algorithms are product ciphers with two components. One component is a clock-controlled[3] nonlinear-filtered[4] linear feedback shift register (LFSR) used as a pseudo-random number generator (PRNG) in an additive stream cipher. The other component is an autokey stream cipher. This PRNG plus autokey product cipher construct provides secrecy and integrity in a single pass, making it a very light weight authenticated encryption with associated data (AEAD) algorithm.

The autokey component is included to support integrity and the PRNG component is included to cover autokey weaknesses (e.g., adaptive chosen plaintext and ciphertext attacks). Autokey ciphers can be divided into two classes: cipher-fed and plain-fed. (The term “key autokey” is a misnomer and is ambiguous. Sometimes its use is synonymous with cipher-fed autokey and sometimes it is just a misleading alias for a PRNG stream cipher.) Cipher-fed autokey algorithms are self-synchronizing and are not useful for supporting integrity. Bleep64’s autokey is technically plain-fed. However, it isn’t a pure plain-fed autokey. It could be described as being halfway between a plain-fed and a cipher-fed autokey. This can be seen in the Bleep64 block diagram shown in Figure 1. The autokey feedback is at the points labeled **u** on the right side of the figure. A pure plain-fed autokey feedback would be taken directly from the plaintext signals (**P**) and a cipher-fed autokey feedback would be taken from the ciphertext signals (**C**). The point **u** is midway between **C** and **P**.

For descriptive purposes only, the Bleep64 crypto state is divided into two 64-bit values (L0 and L1) that hold state of a 127-bit LFSR and two 32-bit values (**x** and **y**) that are used in nonlinear filtering of the LFSR and in holding the feedback state for the autokey component of these product ciphers. These crypto variables are shown across the top and bottom of Figure 1.

The left side of Figure 1 (shown in **blue**) updates the state of the LFSR (L0 and L1). This center section (shown in **black**) updates **x** and **y**, which performs the nonlinear filter of the LFSR and implements the autokey component of the product cipher. The right side (shown in **red**) does encryption combining or decryption decombining.

This figure shows two 32-bit words of plaintext being processed (ciphertext processing differences are shown as insets). Bleep64’s main software loop does two words per iteration because the LFSR clock control is used only every other word processed and the roles of **x** and **y** are swapped for every other word.

Bleep64 uses five different algebras. The idea is that any cryptanalytic technique that uses an approximation to Bleep64 in one algebra would result in poor approximations in the other algebras. This leaves complex, highly nonlinear equations in GF2 as the only way to represent Bleep64 in a single algebra. One focus of the Bleep64 design was to make these GF2 equations as complex and nonlinear as possible, while meeting all of the CPS constraints.

2.1 Ones’-Complement Addition

In the block diagram, the symbol +‘ means ones’-complement addition. Most CPUs today use two’s-complement arithmetic. The conversion from two’s-complement to ones’-complement addition is done by first using a two’s-complement addition instruction followed by adding the carry from the first addition into the resulting sum using a “with carry” variant of the add instruction. Ones’-complement arithmetic has two forms of zero (+0 and -0). The -0 value is represented in binary by having all the bits set to one. With the form of conversion from two’s-complement to ones’-complement addition described here, a sum of +0 is not possible as long as at least one of the inputs to the addition is not +0, which is always the case in Bleep64. The values **x** and **y** are initialized to values that are not +0. Thereafter, all additions to these values will never produce +0.

Using ones’-complement addition creates more nonlinearity in the bits of its sum then does two’s-complement addition. The first row of Figure 2 shows the degree for each carry bit in a 32-bit two’s-complement sum. The second row shows the degree for every carry bit in the sum for the first instruction of a ones’-complement addition. The third row shows the degree for each carry bit after the second instruction in a ones’-complement addition. The GF2 equation for each carry bit has 62 terms.

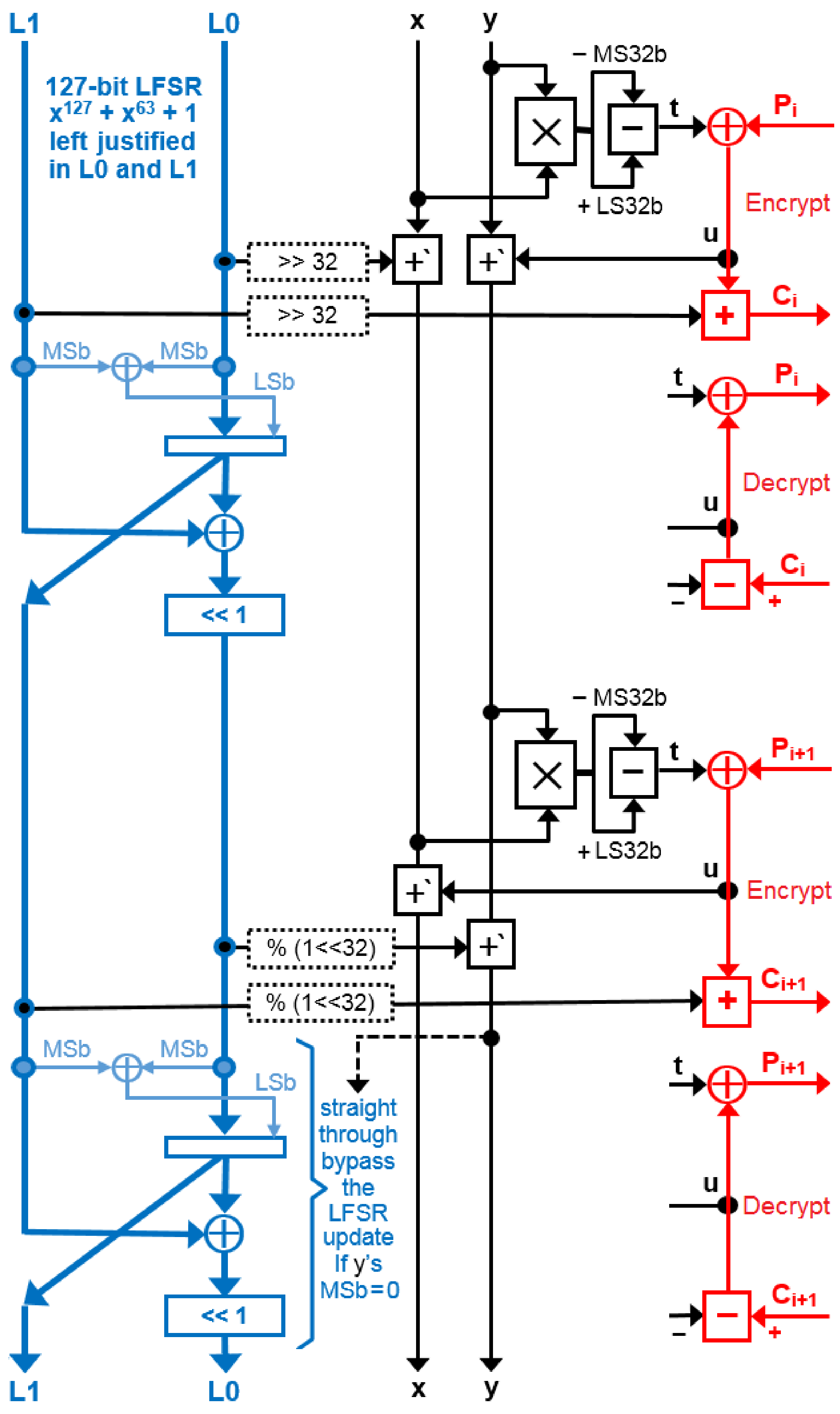


Figure 1: Bleep64 Block Diagram

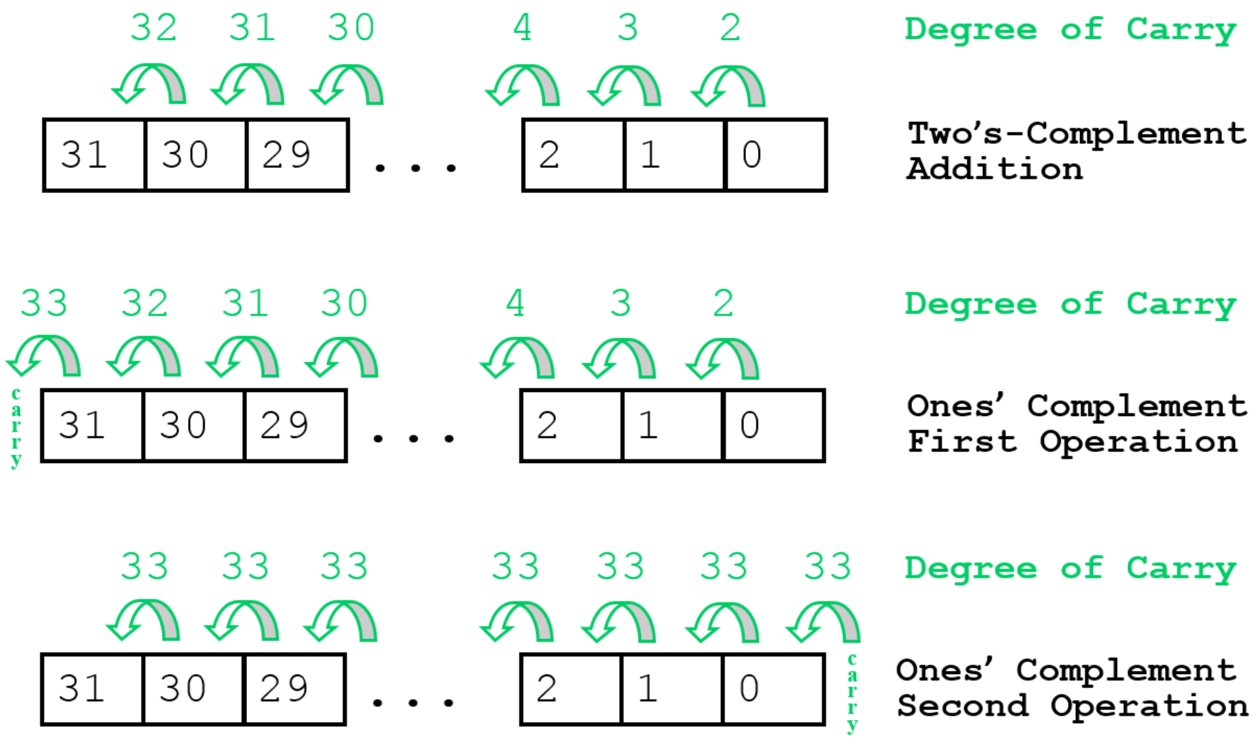


Figure 2: Nonlinearity via Addition Carry

2.2 Linear Feedback Shift Register

The base of Bleep64's PRNG is an LFSR with the primitive characteristic polynomial $x^{127} + x^{63} + 1$. This LFSR was chosen to get a keystream sequence which is long enough to never repeat (the period is $2^{127} - 1$ bits) and has known good statistics while using a minimum amount of storage resources. The LFSR's output satisfies Golomb's randomness postulates[5]. In particular, the 32-bit LFSR values used in the rest of the algorithm are uniformly distributed. And, all transitions from one 32-bit value to another are equally probable. Figure 1 shows the LFSR outputs going through functions in dashed boxes. These are virtual functions; actual implementations would do this via register selection.

Given the CPS design constraints, using an LFSR may seem like an odd choice because of two main LFSR shortcomings. The first is that 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 a word-wide XOR operation produces a word of new bits with just one instruction. Whereas, traditional LFSR software needs multiple instructions to create just one new bit. Because this LFSR's taps are 64 bits apart, word-wide updates can be accomplished on 64-bit CPUs or any CPU with a word size that is a factor of 64. Bleep64 exploits this by doing LFSR updates 64 bits at a time. Given that the LFSR's length of 127 bits is not a multiple of word size (it is known that there are no maximal-length trinomial LFSRs with a size that is a multiple of eight), adjustments need to be made. The first adjustment is to XOR the two tap bits to create a new least significant bit (LSb) prior to doing the 64-bit XOR operation. The second adjustment is a left shift to keep the LFSR left justified within the CPU registers. This LFSR is over 6,000% faster than traditional software LFSRs. The LFSR's second problem is that it's linearity makes it vulnerable to some well-known attacks (e.g., Berlekamp-Massey[6]). Bleep64'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.

2.3 Clock Control

To be conservative, Bleep64 uses two methods to nonlinearize the LFSR: clock-control and nonlinear filtering with state. Because clock control is expensive in software, it is used sparingly. For every loop iteration, the LFSR is unconditionally stepped forward by 64 bits and conditionally stepped forward by an additional 64 bits. The most significant bit (MSb) of $\mathbf{y}$ is used as the clocking control. Because $\mathbf{y}$ is affected by plaintext,

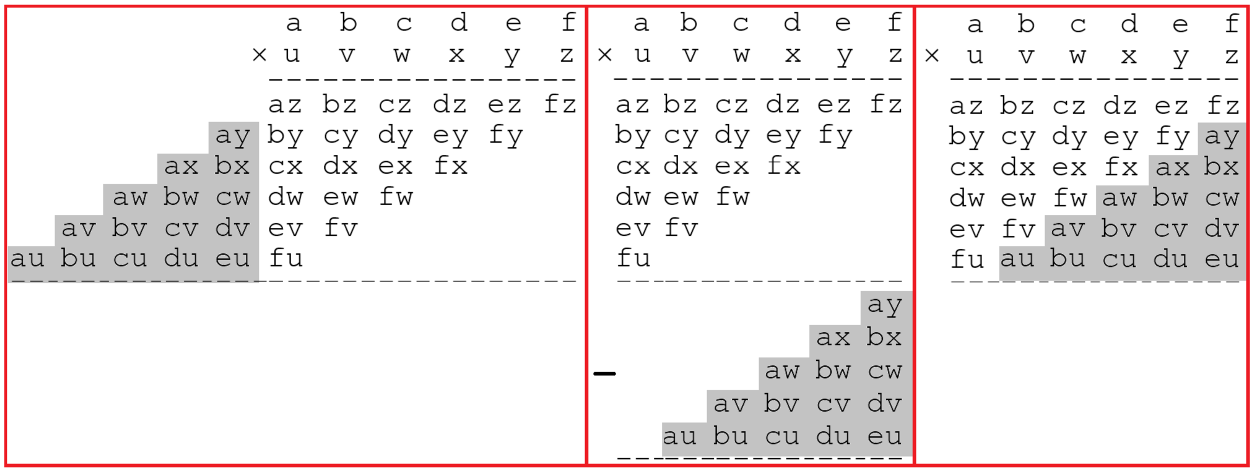


Figure 3: Multiply Operation

using it for clock control includes the LFSR as part of the autokey state carried forward for integrity.

Using branch instructions for clock control is slow on most high-performance CPUs and can leak timing information. An unpredicted branch usually causes a pipeline flush and refill. Many CPU types have conditional instructions that can be used to implement a conditional LFSR update in constant time. Another option is to use this "multiplexer" idea: First, convert the clock control MSb to a full word by doing an arithmetic right shift by 31 bits. Then, replace any conditional expressions of the form

```
IF control THEN state = new_value
```

with the following logic expression using bitwise operations

```
state = (control AND (new_value XOR state)) XOR state
```

Thus, there are three CPU-dependent implementation options for the conditional LFSR stepping: balanced branching, conditional instructions, or multiplexer.

Having a nonlinear filter with state improves the effectiveness of clock control. Clock control by itself results in a crypto state difference which is just a normally-distributed phase shift of the LFSR sequence, dependent only on the total number of zeros versus ones in the sequence of clock control bits. However, the filter state is dependent not only on the total number of zeros versus ones, but also on the specific history of these bits.

Because the LFSR advances either 64 or 128 bits per loop iteration and always uses 128 bits, some bits are reused. Bits are never reused for the same thing (i.e., the combiner, $\mathbf{x}$, or $\mathbf{y}$) and are separated in time as much as possible.

2.4 Multiply Filter

Bleep64's main nonlinearity comes from a multiply operation where two 32-bit values are multiplied together (unsigned) and the upper half of the 64-bit product (MS32b in Figure 1) is subtracted from the lower half (LS32b in Figure 1). With just two instructions, this operation is equivalent to a 32-round block cipher, by itself. Figure 3 illustrates this in three steps for a hypothetical 6-bit multiply where each letter is a bit and juxtaposition indicates an AND operation (for simplicity, carry and borrow are not shown in this figure). The left section of Figure 3 shows the partial products of the multiply and shades the upper half of the product. The center section aligns the upper half of the product underneath the lower half, as is done by the subtraction operation. The right section shifts the upper half of the product upwards to form a square with the lower half of the product. It can be seen that each column has a structure similar to an inner product of the two inputs to the multiply, when viewed as bit vectors. This provides perfect diffusion of the input bits. Of course, including the carry and borrow operations makes this more complicated, with even more nonlinearity and terms in a GF2 polynomial expression of this operation.

This multiply operation has characteristics of an operation in a ring of size $2^{32}+1$. Because this number is semi-prime (versus prime), the output of the multiply operation is not as uniform as desired for an LFSR filter. The major outlying value is zero, which occurs much more frequently than other values. By using inputs to the multiply that are ones'-complement sums (which are never zero), this main outlier is eliminated. Other values are anticipated to be less than one percent of the mean and are distributed over a large range. While not completely uniform, this value distribution is good enough given the "whitening" operation performed in the two-stage combiner.

One desired characteristic of a nonlinear filter is a high nonlinear order. The ones'-complement sums are 33rd order for all of their bits. Because each partial product bit includes the AND of two independent 33rd order results, they are at least 66th order. The carry chains in the multiply and the borrow chain in the subtraction make this a maximal 128th order filter (with 64 bits coming from the LFSR and **u**; and 64 bits coming from the previous **x** and **y** state). In addition to being highly nonlinear, the GF2 equation for each of the output bits has a very large number of terms.

The inclusion of the autokey feedback (**u**) into **x** and **y** very tightly couples the PRNG and autokey components of this product cipher, which protects against divide-and-conquer attacks. Because of the diffusion of the filter, any bit of the autokey feedback has about a 50% chance of affecting any succeeding bit in the message, providing good plaintext diffusion.

Some multipliers leak timing information via early termination. The most common case, and the only one that reveals **t**, is when an input is zero. These cases are precluded by using inputs that are ones'-complement sums, which never result in +0. Some more complex early termination schemes could reveal some information about **x** and **y** (but not their full values). However, any such leakage provides no information about **t** and does not directly affect the relationship between ciphertext and plaintext. Furthermore, these values are overwritten by ones'-complement additions before any subsequent use.

2.5 Two-Stage Combiner

The output of the multiply filter goes to the final section of these algorithms, which is a two-stage combiner. For encryption, the plaintext is first combined with the filter's output using 32-bit XOR. Then, the XOR result is added to a 32-bit LFSR value using two's-complement addition. Because the LFSR value is uniformly distributed, the ciphertext will be uniformly distributed, which "whitens" the not quite uniform output of the filter. Decryption is the obvious reverse of encryption, as shown in the two Figure 1 insets.

This two-stage combiner is used for the following reasons: (1) Addition and subtraction provide some lateral plaintext diffusion via the carry/borrow chains. (2) Using non-associative operations provides some integrity protection. With a simple XOR combiner (or any linear combiner), an adversary knowing a plaintext/ciphertext pair can manipulate the ciphertext bits to make the plaintext resulting from decryption be anything the adversary wants (malleability). A two-stage non-associative combiner precludes this attack. The most significant bit of each word is the only vulnerable bit, but that bit is still covered by the autokey feedback. (3) More generally, knowing a plaintext/ciphertext pair doesn't reveal the PRNG (LFSR) value going to the combiner. (4) It hides the autokey feedback value. (5) It provides the connection between the two parts (PRNG and autokey) of the product cipher.

While this two-stage combiner design means that the algorithm is not an involution and would need separate encryption and decryption code, the increase in code size is negligible. With Bleep64's loop typically being less than 50 instructions, separate copies of the loop can be made for encryption and decryption. If code space is very tight and the utmost performance is not needed, just the two operations of the combiner could be instantiated separately for encryption and decryption, with a conditional jump selecting encryption or decryption for every word.

2.6 Nonce Initialization Vector (IV) Processing

The design goal for Bleep64's initialization is to diffuse nonce bits into Bleep64's key to produce Bleep64's initial state as quickly as possible. Design constraints include that the resulting values for **x**, **y**, and one of either **L0** or **L1** must be nonzero. The nonce processing code is shown below.

```
t  = key[0] * (iv[0] + key[4]);
u  = key[1] * (iv[1] + t);
L0 = key[2] * (iv[2] + u);
L1 = key[3] * (iv[0] + (L0 & 0xFFFF));
x  = key[0] * (iv[1] + (L1 & 0xFFFF));
y  = key[1] * (iv[2] + x);
L0 = ((uint64_t) t << 32) | (L0 & 0xFFFF);   OCS(x, 1);
L1 = ((uint64_t) u << 32) | x;               OCS(y, x);
```

2.7 AEAD Operation

Bleep64 processes authenticated unencrypted data using the same processing as for encrypting data, except that the resulting ciphertext is not written to memory or otherwise sent to an output. The resulting internal crypto state of Bleep64 is carried over to be the initial crypto state for encrypting data, without redoing IV processing. At the end of encrypting data, Bleep64's crypto state is a function of the authenticated unencrypted data and the encrypted data. This crypto state is then the initial state for encrypting the authentication tag, again without be doing IV processing. The decryption process is just the inverse of the encryption process. verification is performed by checking that the decrypted tag matches the expected known value. This is shown in the source code in the following section. This allows single-pass Uninterrupted processing over the typical message structure stored in memory and allows "bump in the wire" processing with minimal latency. The processing described in this subsection can be seen in the source code in the following subsection.

2.8 C Source Code

This section contains contains the C source code for Bleep64. This code is included only as a pedagogic definition of Bleep64. Given all its constraints, lightweight encryption for cyber physical control systems should be written in assembly. Even the smallest production run of CPS devices would have negligible amortized cost for writing the equivalent of less than one page of C code in assembly. In addition, the Bleep64 code shown below was substantially modified from its original form (and the form that would be used in actual operation) in order to conform to the supplied software "test harness", which looks nothing like how encryption would be used in a CPS.

```
#include "api.h"
#include <stdint.h>
#define OCS(a, b) a += (b) + (a + (b) < a ? 1 : 0)  // 1s complement sum
#define Swap(a, b) (a ^= b, b ^= a, a ^= b)
//---------------------------------------------------------------------- Bleep64
inline int Bleep64(B_iv, B_key, B_pt, B_ct, B_ad, adlen, inlen, outlen, encrypt)
  unsigned char *B_iv;        // pointer to initialization vector
  unsigned char *B_key;       // pointer to key
  unsigned char *B_pt, *B_ct; // plaintext and ciphertext pointers respectively
  unsigned char *B_ad;        // pointer to associated data
  int32_t       adlen, inlen; // number of bytes in associated date, and input
  int32_t       *outlen;      // number of bytes in output
  char          encrypt;      // 0 means decrypt, else encrypt
  // local variables
{ register uint32_t t, u;    // short-term temporaries
  register uint32_t x, y;    // auto-key and LFSR filter state
  register uint64_t L0, L1;  // LFSR x^127 + x^63 + 1, left-justified
  register int32_t n;        // byte to process
  uint32_t *iv, *key, *pt, *ct, *ad;    // word casts of byte pointers
  unsigned char *bpt, *bct, *bad; // byte sized pointers for remainders
  unsigned char valid[CRYPTO_ABYTES + 8] = {0}; // buffer for extra validation data

  inline void crypt(char odd, char assoc) // do one word & update crypto state
  { t = (((uint64_t)x * y) >> 32) - (x * y);  // filter LFSR, diffuse autokey
    if (assoc)        u = *ad++ ^ t;
    else if (encrypt) u = *pt++ ^ t, *ct++ = u + (L0 >> 32*odd);
    else              u = *ct++ - (L0 >> 32*odd), *pt++ = u ^ t;
    if (odd) OCS(y, u), OCS(x, L1 >> 32);      // feedback from 1st half of loop
    else     OCS(x, u), OCS(y, L1 & 0xFFFF);   // feedback from 2nd half of loop
    if (odd || y >> 31) L1 |= (L0 ^ L1) >> 63, L0 = (L0 ^ L1) << 1, Swap(L0,L1); }

  // casting kludges
  iv  = (uint32_t*)B_iv;  // word pointer to initialization vector
  key = (uint32_t*)B_key; // word pointer to key
  pt  = (uint32_t*)B_pt;  // plaintext word pointer
  ct  = (uint32_t*)B_ct;  // ciphertext word pointer
  ad  = (uint32_t*)B_ad;  // word pointer to associated data
  // IV processing
  t  = key[0] * (iv[0] + key[4]);
  u  = key[1] * (iv[1] + t);
  L0 = key[2] * (iv[2] + u);
  L1 = key[3] * (iv[0] + (L0 & 0xFFFF));
  x  = key[0] * (iv[1] + (L1 & 0xFFFF));
  y  = key[1] * (iv[2] + x);
  L0 = ((uint64_t) t << 32) | (L0 & 0xFFFF);   OCS(x, 1);   // x != 0
  L1 = ((uint64_t) u << 32) | x;               OCS(y, x);   // L1 != 0, y != 0
  // process associated data
  n = adlen;
  while ( (n -= 8) >= 0 ) { crypt(1, 1); crypt(0, 1); } // do 8 bytes per loop
  if (n > -5) crypt(1, 1);  // if 4 to 7 bytes left, do another full word
  for (n &= 3, bad = (unsigned char *) ad; n--; OCS(x, *bad), bad++);   // do last 1 to 3 bytes
  // do en/de-cryption; decryption needs to end sooner due to validation data
  n = inlen - (encrypt? 0 : CRYPTO_ABYTES);
  while ( (n -= 8) >= 0 ) { crypt(1, 0); crypt(0, 0); } // do 8 bytes per loop
  if (encrypt) {
    // Copy last n bytes to buffer and encrypt from there instead
    bpt = (unsigned char *) pt;
    for (n = 0; n < (inlen % 8); n++) valid[n] = bpt[n];
    pt = (uint32_t *) valid; // use the buffer for pt instead
    n = (inlen % 8) + CRYPTO_ABYTES;
    while ( (n -= 8) >= 0 ) { crypt(1, 0); crypt(0, 0); } // do 8 bytes per loop
  } else {
    // Switch to decrypting into our buffer, B_pt is about to run out of space
    pt = (uint32_t *) valid;
    n = (inlen % 8) + CRYPTO_ABYTES;
    while ( (n -= 8) >= 0 ) { crypt(1, 0); crypt(0, 0); } // do 8 bytes per loop
  }
  if (n > -5) crypt(1, 0);  // if 4 to 7 bytes left, do another full word
  t = (((uint64_t)x * y) >> 32) - (x * y);  // filter LFSR and diffuse autokey
  bpt = (unsigned char *) pt; bct = (unsigned char *) ct; // need char sized pointers
  for (n &= 3; n--;)   // do last 1 to 3 bytes little-endian
    if (encrypt) *bct++ = (char) (*bpt++ ^ (t >> 8*(3-n))) + (L0 >> 8*(3-n));
    else         *bpt++ = (char) (*bct++ - (L0 >> 8*(3-n))) ^ (t >> 8*(3-n));
  if (encrypt) *outlen = inlen + CRYPTO_ABYTES;
  else {
    *outlen = inlen - CRYPTO_ABYTES;
    // Copy first few bytes from the temp buffer to the end of the real one
    bpt = B_pt + (((inlen-CRYPTO_ABYTES)/8) * 8);
    for (n = 0; n < (inlen % 8); n++) bpt[n] = valid[n];
    // And the rest should be zeros
    for (n = 0; n < CRYPTO_ABYTES; n++) if (valid[n+(inlen%8)]) return -1;
  }
  return 0;}
//------------------------------------------------------------------ encrypt
int crypto_aead_encrypt(unsigned char *c,       unsigned long long *clen,
                        const unsigned char *m,  unsigned long long mlen,
                        const unsigned char *ad, unsigned long long adlen,
                        const unsigned char *nsec,
                        const unsigned char *npub,
                        const unsigned char *k)
{ return Bleep64(npub, k, m, c, ad, adlen, mlen, clen, 1); }
```

```
//---------------------------------------------------------------------- decrypt
int crypto_aead_decrypt(unsigned char *m,        unsigned long long *mlen,
                        unsigned char *nsec,
                        const unsigned char *c,  unsigned long long clen,
                        const unsigned char *ad, unsigned long long adlen,
                        const unsigned char *npub,
                        const unsigned char *k)
{ return Bleep64(npub, k, m, c, ad, adlen, clen, mlen, 0); }
```

Bleep64's loop processes two 32-bit words per iteration. When there is less than two words left, the loop exits and the rest of the processing is handled, down to byte granularity, so as to not add any "padding" overhead on CPS communication networks (which typically have byte granularity and can ill afford any additional communication overhead).

3 Security

There is no know attack against Bleep64 that's any more efficient than brute force key search over 128-bit keys. Given the structure of the multiply operation, standard linear and differential cryptanalysis techniques are not feasible. Mod N (with N= $2^{32}+1$) analysis could be applicable against the multiply operation. However, with the ones'-complement sums feeding the multiply inputs being mod N operations with N= $2^{32}-1$, this line of attack is precluded. Independent analsys by David Wagner confirmed these observations. DIEHARD showed no biases. Big Crush has not been tried yet.

4 Bleep64 Performance

Except for Bleep64, the table in Figure 4 was taken from the Triathlon results for the ARM Cortex M3[7]. This type of processor is typical of the kind found in CPSs. Bleep64 was adapted to the Triathlon rules by eliminating remnant processing and conforming to the Triathlon's calling conventions.

AES[8] is highlighted because it is the current standard. Note that, compared to AES, Bleep64 has about 9% of the code size, about 15% the RAM size, and is about 38 times faster. However, only Bleep64 provides the authentication and integrity required for the CPS environment. If authentication and integrity were added to these block ciphers, e.g., using Galois counter mode (GCM)[11] instead of CBC mode, the relative performance of Bleep64 compared to these block ciphers would be much larger. For example, a 128-bit AES-GCM is about two and half times slower than AES-CBC on ARM Cortex processors[9]. Another M3 AES test implementation with GCM[10] took 44,306 cycles, where Bleep64 would have taken less than 400 cycles.

Some processors have hardware acceleration for AES. Typical performance on an M3 with such hardware is 76 clocks per block (using a 128-bit key), including moving data to/from the accelerator[12]. However, even if this acceleration was infinitely fast, AES-GCM would still be many times slower than Bleep64. There are several impediments to efficient implementation of GCM on CPS processors. Most of these processors don't have any form of GF2 polynomial multiply instruction and CPS SoCs don't have enough memory to implement the tables needed to improve GHASH speed[13]. Even if there were enough memory, the lack of data cache would make these operations slow.

Previously, BeepBeep was found to be the fastest and smallest of 19 ciphers tested on a 16-bit MSP430 processor[14] and the only cipher small and fast enough to be usable on an 8/16-bit hybrid processor[2]. Given that Bleep64 is a smaller and faster version of BeepBeep, it would perform even better. It is serendipitous that these algorithms are the fastest known encryption algorithms on 16-bit processors, as this was not a specific design goal.

5 Intellectual Property Statements

Intellectual property statements have been mailed and scanned copies are included in submission email.

6 Conclusions

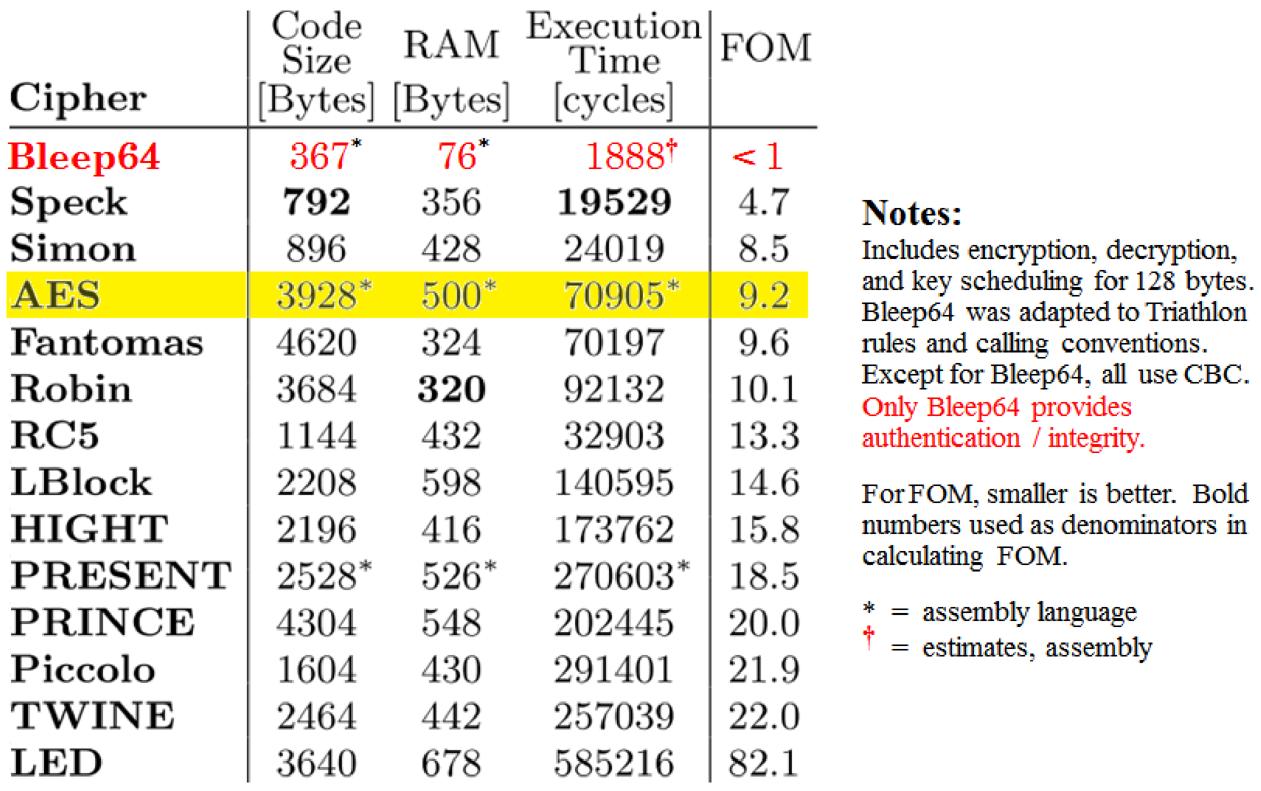
The cyber physical control system environment (including much of IoT) has some unique characteristics that lead to specific requirements and design goals for cryptography, which include:

- great computational efficiency (CPU cycles and memory) when implemented as software on the 32-bit CPUs most likely to be used in CPSs and IoT
- require no special encryption hardware
- internal state size small enough to fit into a CPU's register set
- exploit "free" processor assets that make cryptanalysis costly
- low power consumption and low variation in power consumption
- negligible increase in required communications bandwidth
- negligible added latency and variation in latency (jitter)
- integrity and secrecy processing in one pass
- very low side-channel attack susceptibility (use a native stream cipher)
- very good key agility

The BeepBeep and Bleep64 algorithms were designed to meet all the items in this list. They are better suited to CPS encryption than any other known algorithms. They have been designed to be resistant to various forms of cryptanalysis, including: linear, differential, mod N, and algebraic. They've also been designed to withstand side-channel attacks: DPA, TEMPEST, timing. They have characteristics allowing some special uses, including: as an arbitrarily large block cipher, as an AEAD cipher with an arbitrary inter-mix of integrity and secrecy segments.

The following observations should be given serious consideration for an LWC standardization effort:

- Meeting current LWC requirements is insufficient for the CPS/IoT environment
- CPS ciphers should be designed for software implementation
- CPS cipher designs should not be degraded to meet 8- or 16-bit processor constraints
- CPS cipher designs should not be hindered by requiring them to be written in C
- Ciphers intentionally designed with a key size larger than its desired security level (e.g., for performance reasons) should not be considered to be "broken"



Cipher	Code Size [Bytes]	RAM [Bytes]	Execution Time [cycles]	FOM
Bleep64	367*	76*	1888†	< 1
Speck	**792**	356	**19529**	4.7
Simon	896	428	24019	8.5
AES	3928*	500*	70905*	9.2
Fantomas	4620	324	70197	9.6
Robin	3684	**320**	92132	10.1
RC5	1144	432	32903	13.3
LBlock	2208	598	140595	14.6
HIGHT	2196	416	173762	15.8
PRESENT	2528*	526*	270603*	18.5
PRINCE	4304	548	202445	20.0
Piccolo	1604	430	291401	21.9
TWINE	2464	442	257039	22.0
LED	3640	678	585216	82.1

Notes:
Includes encryption, decryption, and key scheduling for 128 bytes. Bleep64 was adapted to Triathlon rules and calling conventions. Except for Bleep64, all use CBC. Only Bleep64 provides authentication / integrity.

For FOM, smaller is better. Bold numbers used as denominators in calculating FOM.

* = assembly language
† = estimates, assembly

Figure 4: Performance on an ARM Cortex M3

- Crypto algorithm performance should be stated for both best and worst case cache-state scenarios (including all caches flushed and “cooled” replacement history)

References

[1] National Security Agency: NACSIM 5000, February 1982 ~~(Confidential)~~, Tempest Fundamentals: Redacted FOIA version at cryptome.org/jya/nacsim-5000/nacsim-5000.htm

[2] Driscoll, K.: BeepBeep: Embedded Real-Time Encryption. FSE 2002, LNCS 2365, pp. 164–178, 2002.

[3] Gollmann, D., Chambers W.: Clock-Controlled Shift Registers: A Review. IEEE Journal on Selected Areas in Communications. (1989) 7: 525-533.

[4] 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

[5] Golomb, S. W.: Shift Register Sequences. Aegean Park Press, Laguna Hills, CA, 1982, revised ed.

[6] Berlekamp E.R.,: Algebraic Coding Theory. McGraw-Hill, 1968

[7] Dinu, D., Le Corre Y„ Khovratovich D., Perrin L., Großschädl J., and Biryukov A.: Triathlon of Lightweight Block Ciphers for the Internet of Things University of Luxembourg

[8] Daernen, J., Rijmen, V.: AES Proposal: Rijndael. AES Submission. (June 1998)

[9] Vincent H., Tschofenig H., Pegourie-Gonnard M.: Performance of State-of-the-Art Cryptography on ARM-based Microprocessors. NIST Lightweight Cryptography Workshop 2015

[10] Birr-Pixton J.: Cifra. github.com/ctz/cifra

[11] Dworkin M.: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, November, 2007

[12] Silicon Labs: EFM32TG Reference Manual, Section 27. www.silabs.com

[13] McGrew D., Viega J.: The Galois/Counter Mode of Operation (GCM). May 31, 2005

[14] Malina L., Hajny J., Mlynek P., Machacek J., Svoboda R.: Towards Efficient Application of Cryptographic Schemes on Constrained Microcontrollers. Journal of Circuits, Systems, and Computers Vol. 25, No. 10 (2016) 1650129
DOI: 10.1142/S0218126616501292