Cryptogr. Commun. (2018) 10:123–137
DOI 10.1007/s12095-017-0225-x

KISS: A bit too simple

Gregory G. Rose[1]

Received: 5 November 2016 / Accepted: 10 April 2017 / Published online: 2 May 2017

Abstract KISS ('Keep it Simple Stupid') is an efficient pseudo-random number generator originally specified by G. Marsaglia and A. Zaman in 1993. G. Marsaglia in 1998 posted a C version to various USENET newsgroups, including `sci.crypt`. Marsaglia himself has never claimed cryptographic security for the KISS generator, but others have made the intellectual leap and claimed that it is of cryptographic quality. In this paper we show a number of reasons why the generator does not meet some of the KISS authors' claims, why it is not suitable for use as a stream cipher, and that it is not cryptographically secure. Our best attack requires about 70 words of generated output and a few hours of computation to recover the initial state. In early 2011, G. Marsaglia posted a new version of KISS, which falls to a simple divide-and-conquer attack.

Keywords Stream cipher · PRNG · Cryptanalysis

Mathematics Subject Classification (2010) 94A60

1 Introduction

KISS ('Keep it Simple Stupid') is an efficient pseudo-random number generator specified by G. Marsaglia and A. Zaman in 1993 [1]. G. Marsaglia in 1998 posted a C version to various USENET newsgroups, including `sci.crypt`, culminating in a final version [3]. While Marsaglia himself has never claimed cryptographic security for the KISS generator, he does hint about it, and others have made the intellectual leap and claimed that it is of

This article is part of the Topical Collection on Recent Trends in Cryptography.

✉ Gregory G. Rose
ggr@seer-grog.net
www.Allocrypt.com

[1] Allocrypt, Inc., San Diego, CA, USA

cryptographic quality. In this paper we show a number of reasons why the generator does not meet some of the KISS authors' claims, why it is not suitable for use as a stream cipher, and that it is not cryptographically secure as a pseudo-random number generator. Indeed, this paper is written as a cautionary tale; stream ciphers and cryptographically secure random number generators are **hard to get right**, even for some of the world's preeminent experts on the subject of randomness. We present a number of standard attacks that damage KISS to some extent, and our best attack requires only 70 words of generated output and on average $2^{29.1}$ operations of a simple algorithm to recover the initial state. We attempt to present the reasoning that leads to the attacks.

Since 1998 Marsaglia has posted a number of variants of KISS (without version numbers), including [4] in January 2011. In the sequel, we refer to this as *KISS11* for disambiguation. These variants have similar design elements but ever increasing state sizes, making them impractical for most cryptographic uses anyway. We present a straightforward divide-and-conquer attack on this (currently) latest version of KISS, requiring 2^{41} operations equivalent to key initializations.

Marsaglia's generators are primarily intended for scientific applications such as Monte Carlo simulations. There are many other generators in use, often with (ridiculously) long periods, for example the Mersenne Twister [6], and later versions of KISS. We choose to examine the 1999 version of KISS rather than some of these later proposals, because:

- It is proposed by well known and respected authors
- It has a reasonably long but not excessive (claimed) period
- It has a compact state
- The output 'looks random' immediately after initialization.

Other proposals often involve large state variables, often calling on simpler RNGs to initialize these large states. For cryptographic purposes, the lack of a well-specified initialization or keying phase, the large state, and the time required for initialization are all undesirable. It is our opinion that most of these other proposals are also cryptographically weak, but that doesn't matter because they are also impractical in the context of a stream cipher. In any case, one recent proposal is examined below.

In Section 2 we describe KISS and the components that make it up. Section 4 discusses some simple structural attacks. Mathematical properties of the components are described in Section 3, and in Section 5 these properties are combined to form a cryptanalytic key recovery attack in the known-keystream model. A variation of the attack would even apply to unknown plaintext, if the input language is sufficiently biased and with a significant slowdown. Finally Section 6 examines the 2011 version of KISS (KISS11) and presents a simple attack.

Quotations in this document are from Marsaglia's USENET posting [3] unless mentioned otherwise.

2 Description of KISS

KISS consists of a combination of four sub-generators each with 32 bits of state, of three kinds:

- one linear congruential generator modulo 2^{32}
- one general binary linear generator over the vector space $GF(2)^{32}$
- two multiply-with-carry generators modulo 2^{16}, with different parameters

The four generators are updated independently, and their states are combined to form a stream of 32-bit output words. The four state variables are treated as unsigned 32-bit words. Figure 1 shows C code from [3], while Fig. 2 shows the structure pictorially. We refer to the state registers using the names of the C macros.

```
The KISS generator, (Keep It Simple Stupid), is
designed to combine the two multiply-with-carry
generators in MWC with the 3-shift register SHR3 and
the congruential generator CONG, using addition and
exclusive-or. Period about 2^123.
```

Note that only half of the state of *znew* contributes directly to the output of KISS. Its placement in the diagram is intended to show this, and is not an error.

3 Properties of components

The four registers, of three kinds, that make up KISS are independently updated, then combined using shifts, adds, and one XOR operation. As is often the case, the registers that use arithmetic operations are combined with XOR, while the results that are produced bitwise are combined with addition modulo 2^{32}.

3.1 Multiply with carry registers

Two of the registers, *wnew* and *znew*, are Multiply With Carry generators. At each step, the low-order 16 bits is multiplied with a constant, and the high order 16 bits from the previous step are added. The low order 16 bits are random looking, and with a period determined by the entire construct. The most significant bits of the register are not very random; in particular the MSB will be almost always zero. But these most significant bits contribute to the long period. The low 16 bits of *znew* are shifted left before being added to the whole register *wnew*, presumably to cover up this nonrandomness.

With a multiplier C, this generator calculates $x_n = Cx_{n-1} \bmod 2^{16}C - 1$. With the stated constants, the modulus is prime, so the generator is readily inverted.

```
The MWC generator concatenates two 16-bit multiply-
with-carry generators, x(n)=36969x(n-1)+carry,
y(n)=18000y(n-1)+carry mod 2^16, has period about
2^60 and seems to pass all tests of randomness. A
favorite stand-alone generator---faster than KISS,
which contains it. [sic]
```

The word ‘concatenates’ is somewhat misleading, as the high order bits of *wnew* contribute to the output, even though Marsaglia appears to consider it to be a 16-bit generator, but this does not affect either the period of the *MWC* construction, or our later cryptanalysis.

For cryptographic purposes, initialization of these registers is somewhat important. It is easy to see that the value zero will repeat forever. Large values will, at the first update, become smaller values that collide with the results of updating some smaller value, and the generators will then follow the same path as if they were initialized with those smaller

```
#define znew   (z=36969*(z&65535)+(z>>16))
#define wnew   (w=18000*(w&65535)+(w>>16))
#define MWC    ((znew<<16)+wnew )
#define SHR3  (jsr^=(jsr<<17), jsr^=(jsr>>13), jsr^=(jsr<<5))
#define CONG  (jcong=69069*jcong+1234567)
#define KISS  ((MWC^CONG)+SHR3)
```

Fig. 1 C code of KISS

values. For each register, there are two distinct cycles of equal prime period. (We were not aware of any analytical result, but the small state space allowed us to enumerate the cycles in a few minutes of CPU time.) There is also another fixed point (besides zero) for each register. Table 1 summarizes these details. In the table, a cycle length of 'n/a' denotes a value that collides and enters an existing cycle.

The values shown in the table are the least values that occur in the cycle, or the values that lead into the existing cycles.

For cryptographic purposes there are thus two bad values for each register, and approximately half of random values enter the same cycles as other values, which might be seen as nearly equivalent keys: only the first word generated would be different, but the same thereafter.

Since the periods of the two registers are prime, *if* the initial values are not bad, the period of *MWC* (that is, the combination of the two subgenerators) will be about $2^{59.3}$. The period of the least significant 16 bits will only be that of *wnew*, which is about $2^{29.1}$.

Two consecutive 16-bit outputs from either generator suffice to recover the unknown upper 16 bit carry value, once the generator is in a cycle. It is only slightly more complicated to go backwards in the cycle. Given the least significant 16 bits of an initial (out of cycle) value, and the next (in cycle) state, it is easy to recover the most significant 16 bits of the initial value.

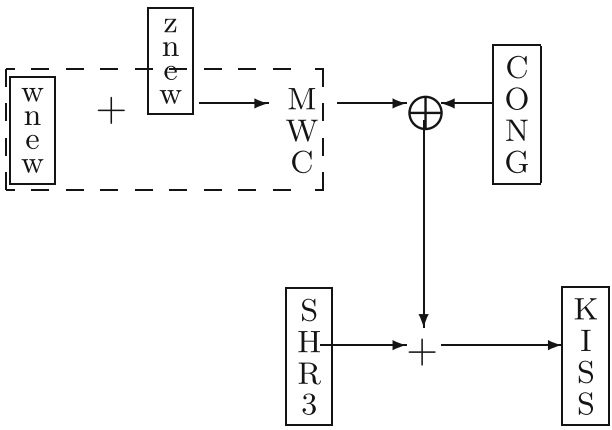

Fig. 2 Diagram of KISS

Table 1 Properties of the Multiply-With-Carry registers

Register	Constant	No. cycles	Cycle length	Value
wnew	18,000	2	1	0 and 0x464fffff
		2	589,823,999	1 and 31
		3,115,319,296	n/a	$\geq 0x46500000$
znew	36,969	2	1	0 and 0x9068ffff
		2	1,211,400,191	1 and 5
		1,872,166,912	n/a	$\geq 0x90690000$

3.2 Properties of the linear congruential generator

The linear congruential generator *CONG* is well studied in the literature. With the specified parameters it has period 2^{32}.

```
CONG is a congruential generator with the widely used 69069
multiplier: x(n)=69069x(n-1)+1234567. It has period
2^32. The leading half of its 32 bits seem to pass
tests, but bits in the last half are too regular.
```

Observe that if the value is odd, the next value will be even, and vice versa. That is, the least significant bit toggles ...010101.... Other bits exhibit patterns with longer periods, but the attack we describe only needs the least significant bit.

Observation For each $n < 32$, the least significant n bits of *CONG* form a smaller Linear Congruential Generator modulo 2^n, with recurrence

$$x_i = (69069 \bmod 2^n)x_{i-1} + (1234567 \bmod 2^n)(\text{modulo}2^n)$$

where *mod* represents a remainder operation as in C. The behaviour of the least significant bit is just the simplest application of this observation.

Plumstead [7] gives a method of inferring the state of a linear congruential generator from information about its *most* significant bits, which appear more random, but the above observation implies that information from the least significant bits cannot be extended upward; consider the analogy with knowing the least significant bits of a simple counter.

3.3 The 3-shift register generator

The generator *SHR3* is updated using shifts and XOR operations that are efficient. However for analysis purposes this is most easily understood as multiplication modulo 2 of a 32 by 32 bit update matrix with the 32-bit register state. The system is by definition linear and amenable to standard techniques.

```
SHR3 is a 3-shift-register generator with period
2^32-1. It uses y(n)=y(n-1)(I+L^17)(I+R^13)(I+L^5),
with the y's viewed as binary vectors, L the 32x32
binary matrix that shifts a vector left 1, and R its
transpose. SHR3 seems to pass all except those
related to the binary rank test, since 32 successive
values, as binary vectors, must be linearly
independent, while 32 successive truly random 32-bit
integers, viewed as binary vectors, will be linearly
independent only about 29% of the time.
```

The sequence of values generated by *SHR3* is not maximal length. Appendix lists the possible cycles. The bit sequence generated by each bit position of the register is the output of some Linear Feedback Shift Register of degree $D \leq 32$. There are 16 different polynomials, with 64 disjoint cycles, with lengths ranging from 1 to 306,706,140 (approximately $2^{28.2}$). The expected cycle length (although no such cycle exists) is approximately $2^{27.7}$; more than half of initial values lie on one of the longest cycles. The values 0 and 0xaea21b8f are self-perpetuating.

McQueen [5] pointed out that later versions of KISS exchange the values 13 and 17 in the formula, and do indeed lead to a single cycle with maximum length. In particular KISS11, analyzed below, uses the correct version of this generator.

Any sequence of 32 output bits (in particular, we use the least significant bit of consecutive output words) suffices to recover the initial state of the SHR3 register, via a simple matrix multiplication.

4 Cryptographic deficiencies and simple structural attacks

Keying Since the authors of KISS did not envision the generator being used for cryptographic purposes, they did not think about issues such as setting up 'keys', or reinitializing with nonces. Since the state of KISS is only 128 bits, it makes sense to simply use a 128-bit key by dividing it into 4 32-bit words, and defining that words (both key and output) should be treated as Little- or Big- Endian. None of the components show obvious output relationships with related keys after a few words of output; if a nonce or initialization vector is to be used, it could be simply XORed or added to the key, with a few rounds of output being discarded to allow any similarities to dissipate. We equate key recovery with recovering the exact initial state.

Time-Memory Tradeoff Since the keyspace is the same as the state space, standard time-memory tradeoff attacks could be easily applied. See [8] for an overview of the past and current application of such attacks. We do not explore them further here.

Weak keys If the first output word is ignored, only about 2^{126} output sequences can be generated, due to the colliding behaviour of the Multiply-With-Carry generators. *CONG* is the only generator for which zero is not a bad value. Any key in which one of the words

initializing another register is zero will effectively take that register out of the combined generator, leading to a shortened cycle and possibly exposing the generator to simpler attacks. In particular, the all-zero key results in the generator being exactly the output from *CONG*, a simple and easily recognized linear congruential generator.

Cycle length The maximum cycle length is about $2^{32+59.3+28.2} = 2^{119.5}$. This contradicts the KISS authors' claim of 'about 2^{123}'. We hypothesize that they expected *SHR3* to generate a maximum length sequence, as in the quotation above. Significantly shorter cycles are possible but correspondingly less likely to happen. The shortest cycle is 2^{32} (from *CONG*) occurring with probability 2^{-93}, since there are two minimum-cycle-length values for each of the other three generators.

Divide and Conquer KISS updates the four sub-generators independently, and only combines them for the output. This structure invites attacks where one or more of the components can somehow be ignored or cancelled out to enable an attack on the remainder. Indeed the attack below uses this technique to some extent. Another way to apply divide-and-conquer attacks would be to take output words from a long keystream, and combine words separated by the period of one of the subgenerators, in such a way as to cancel it out. For example, if the output of *SHR3* really had period $2^{32} - 1$, words that far apart in the keystream could be subtracted to remove the effect of that sub-generator. This technique was used successfully in the cryptanalysis of the SSC2 stream cipher [9].

5 Key recovery attack in the known keystream model

Marsaglia, in a different article [2], writes:

```
"A random number generator is like sex:
When it's good, its wonderful;
And when it's bad, it's still pretty good."
Add to that, in line with my recommendations
on combination generators;
"And if it's bad, try a twosome or threesome."
```

It is a common theme in both amateur and professional cryptography to start with a design, and as defects are discovered, to try to cover them with another feature or another generator. In the case of KISS, this design is apparent with the use of simple generators with known flaws to cover each others' deficiencies. The job of the cryptanalyst, conversely, is to find ways to expose the flaws and defeat them in detail. In this section we not only show a key recovery attack against KISS, but we attempt to reveal the thought process, and experiments conducted, to arrive at this attack. This paper hopes to teach a little bit of cryptanalysis technique.

Our goal is to take some amount of known keystream (a standard assumption for cryptanalysis of stream ciphers) and recover the initial state of the four registers.

We first began by analyzing the components, to better understand them, as Marsaglia's USENET posting was short on details. We did not try to obtain the original technical report [1], which probably contains more analysis, in favor of developing our own understanding. *CONG* was already well understood. The Multiply With Carry generators *wnew* and *znew*

were new to us, so we wrote code to enumerate the cycle structure, and looked at how easy it was to run the generators backwards.

When it came time to analyze the *SHR3* generator, at first we accepted the implication that it had the maximum possible period of $2^{32}-1$, and wanted to find the equivalent LFSR, as we were more familiar with cryptanalyzing such structures. Using the default initial value from the KISS paper, we generated 128 output words, and selected the least significant bits for input to the Berlekamp-Massey algorithm. As expected, a degree-32 polynomial was the output, which did not contradict our belief that it might have a maximum length cycle. We continued on that assumption.

Sorting out the combination of four generators seems already to be difficult, but one of the generators, *znew*, only contributes to the upper half of the output words. Concentrating on the lower half reduces the number of generators to three. This is already an application of the *Divide and Conquer* strategy. The generators are combined with both integer addition and XOR. Either the carries from the addition must be accomodated, or we could focus on the least significant bit, in which addition and XOR are equivalent since there is no carry in. This approach looked promising, as the behaviour of the least significant bits of two of the generators were easily analyzed.

The least significant bit of *wnew* did not exhibit any linearity or other non-random properties we could detect, as was to be expected from Marsaglia's description. Further, it depended on the entire 32-bit state of the register. But only the very first output depended on the full state; the second and all subsequent outputs were in one of the known cycles. By skipping the first output word, at least for now, only 1,179,648,000 values (about $2^{30.1}$) for the state needed to be considered.

This suggests the first part of the attack. Assume that the cryptanalyst knows some amount of output keystream words $Z_i, 0 \le i \le N$ for some as yet undetermined number N. We want to apply some sort of *Guess and Determine* attack. In such an attack, we *guess* a value for some unknown quantity, *determine* from the known and guessed values some other values, and then attempt to verify whether or not the guess was correct, that is, consistent with known values or properties. If the verification fails, we try another guess, thus eventually enumerating some subset of the unknown state.

Step 1. **Ignore Z_0. Ignore all but the least significant bits of Z_i.** We will worry about the actual initial state later; for now we try to recover the state after the first output word. We now have a sequence of N bits of known keystream.

Step 2. **Guess** $wnew_1$ (that is, the state of the register *wnew* immediately after the first output). We only have to enumerate the values in cycles. From this guess, generate the stream of least significant bits $w_i = wnew_i \& 1, 1 \le i \le N$.

Step 3. **Guess the LSB of $CONG_1$.** As discussed above, the stream of LSBs is alternating zeros and ones, but we don't know which it starts with. We need to test both. Let $c_i = 010101010\ldots$. If this guess doesn't work out, invert it; in fact, we can just invert the entire sequence used in steps 4 and 5.

Step 4. **Determine the possible LSBs of the *SHR3* output.** Set $s_i = Z_i \oplus w_i \oplus c_i, 1 \le i \le N$.

Step 5. **Verify that s_i is the output of an LFSR.** When we first implemented this step, we used the discovered polynomial $1 + x^3 + x^5 + x^8 + x^{18} + x^{22} + x^{30} + x^{32}$ to check the parity of the corresponding output bits, with $N = 100$. If our guesses for the *wnew* and *CONG* registers were correct, the sums of the bits should all be zero, otherwise with overwhelming probability $(1-2^{-67})$ at least one of the parity equations would be one. This worked fine for the set of initial values used by default in KISS, but our second trial did not find a solution! After a bit of thought, the only reason we could think of was that there

was not a single LFSR that described the system. We already had a very simple program to enumerate cycles in a 32-bit function, so with one minute of editing and three minutes of computation on an Apple laptop, we were able to find the details in Appendix. We could have proceeded with the verification by checking all 16 of the distinct sets of LFSR taps, but instead chose the simpler approach of just using the Berlekamp-Massey algorithm again. 64 bits of input are necessary to discover the longest LFSR, and only surprisingly few more bits are needed to detect if the bit stream is not linear. With high probability, an extra 5 bits will show that. It is faster to let a small number of false positive results through this stage and eliminate them later than to run extra bits through the Berlekamp-Massey algorithm, which has $O(n^2)$ behaviour in the number of bits. This turns out to be the limiting step in the attack, and could perhaps be optimized further.

Step 6. **Recover $SHR3_1$.** A simple matrix multiplication turns the bits s_i, $1 \leq i \leq 32$ into the initial state $SHR3_1$. At this point, with high probability, we know all 32 bits of *SHR3*.

Step 7. **Recover the least 16 bits of $CONG_1$.** Given the known output, and full (assumed correct for now) values for *SHR3* and *wnew*, the least significant 16 bits of *CONG* are recovered. At first we assumed that the well known result [7] would reveal the full initial state, but we had misremembered the result. At this point we know the full state of two of the registers, half the state of *CONG*, and nothing about *znew*.

Step 8. **Guess the high 16 bits of $CONG_1$.** We have to apply the Guess and Determine attack again, guessing one of the registers and determining the other. It is obviously most efficient to guess the remaining contents of the register that is partially known, rather than the whole 32 bits of the other unknown register. We then generate 3 output words from the three known or guessed registers.

Step 9. **Determine the least 16 bits of $znew$.** From the known keystream and the other registers, we recover the low 16 bits of $znew_i$, $1 \leq i \leq 3$.

Step 10. **Verify consistency of the guess of $CONG_1$.** From the low 16 bits of the first two values of $znew$, we derive the full state of $znew_1$. From this we generate $znew_3$ and verify that the least 16 bits agree with those generated in Step 9. If not, we continue enumerating the high order bits of $CONG$.

Step 11. **Verify the entire state**. We now have a candidate for the state of all four registers after the first output. We generate 8 words of output from KISS with that state, and check that they agree with Z_i, $1 \leq i \leq 8$. Checking 8 values is almost certainly more than necessary.

Step 12. **Recover the true initial state.** Remember that we skipped the first output Z_0, because the multiply-with-carry registers might not have started with values in their corresponding cycles. For each of the registers there are two (or for $wnew$ sometimes three) possible initial states. We simply need to check which of the (up to 6) possibilities are consistent with the known output Z_0.

The algorithm above is completely dominated by the time taken in Step 5; at most a handful of candidates get through to steps 7-12. This step runs the Berlekamp-Massey algorithm on 69 bits; in the worst possible case it will be run on about $2^{31.1}$ inputs (the possible cycles of the $wnew$ register, times two for the LSB of the $CONG$ register). It takes on average about 2 hours on a single CPU of an Apple Macbook Pro (2.66 GHz Intel Core i7), with negligible memory usage. We did not bother optimizing or parallelizing the code. By the nature of the Guess and Determine steps, there are no obstacles to parallelization. Possible optimizations of this step include performing the parity checks instead; we think this would result in a significant speed increase but we didn't actually try it.

We note in passing that Step 5 could be done by applying a Fast Correlation Attack to unknown plaintext, if the least significant bits of the plaintext language are sufficiently biased and enough output is available. Of course this would take significantly longer to run the attack.

6 KISS11

In January 2011, just after we completed the above analysis, Marsaglia posted an article to `sci.crypt` and other Usenet newsgroups with a new version of KISS, which we refer to as KISS11. In this article he presented two generators, one based on 32-bit integer variables, and the other on 64-bit integer variables. Here we examine only the 32-bit version, but the attack applies equally to the larger version with correspondingly larger complexity.

KISS11 (shown in Fig. 3, with one declaration moved for readability) has three components. The congruential generator and the 3-shift generator (CNG and XS respectively) have new names but are otherwise the same as described above (with XS corrected). The real change is that the two small Multiply With Carry generators have been replaced with a single, huge Complementary Multiply With Carry generator, implemented by the function `b32MWC`. This generator has $2^{22} + 1$ words of state (the array `Q` and integer `carry` above). We don't try to explain how it works; for that see [4]. Marsaglia does mention its cryptographic weakness, and attempting to hide it:

> `Of course, if you can observe [4194304] successive values, then, after a little modular arithmetic to determine the carry, you are OK.`
> `[...] Even though the MWC RNGs perform very well on tests of randomness, combining with Congruential (CNG) and Xorshift (XS) probably provides extra insurance at the cost of a few simple instructions, (and making the resulting KISS even harder to crack).`

Really, the only cryptographic strength of this subgenerator is its huge (just over 2^{27} bits) state. For the first 16 megabytes of output, it is behaving like a Vernam cipher. For simplicity

```
static unsigned long Q[4194304],carry=0;
unsigned long int i,x,cng=123456789,xs=362436069;

unsigned long b32MWC(void)
{unsigned long t,x; static int j=4194303;
 j=(j+1)&4194303;
 x=Q[j]; t=(x<<28)+carry;
 carry=(x>>4)-(t<x);
 return (Q[j]=t-x);
}

#define CNG  ( cng=69069*cng+13579 )
#define XS ( xs^=(xs<<13), xs^=(xs>>17), xs^=(xs<<5) )
#define KISS ( b32MWC()+CNG+XS )
```

Fig. 3 C code of KISS11

in the discussion below, we refer to $R = 2^{22}$ as the number of outputs in a "round", that is, a full pass through the array Q.

We assume that the contents of Q, cng and xs are initialized completely randomly and independently of each other. Note that the initial value of carry is explicitly set to zero. We assume that the output stream from KISS is known to the attacker (a standard cryptographic assumption); call this output stream $Z_i, i = 1...$

Some observations:

1) at every stage, carry will usually be only 28 bits in length. The exception is when the high order 28 bits of the input Q value are all zero, carry will (almost never, but depending on the input carry value) become 0xFFFFFFFF. This is rare enough not to worry about in the attack below.

1a) if you know the input value $Q[t]$, the output carry value is almost entirely defined by the most significant 28 bits. 15/16ths of the time, whether the optional subtraction of 1 happens is determined by comparing the least significant and most significant 4 bits of the input $Q[t]$, independent of the input carry.

2) at every stage t, the output of b32MWC will be the input value of $Q[t \bmod R]$ to the calculation of b32MWC R steps later.

3) CNG and XS are easily invertible (see above).

4) b32MWC is also invertible; if you know both the input $Q[j]$ and the output $Q[j]$ not only can you derive the output carry, you can also derive the input carry.

5) given (3) and (4), you can run the generator backwards if you need to.

The simple attack we present is a trivial application of the divide-and-conquer method, splitting the generator into two parts.

1. Start by guessing (enumerating) the 2^{64} possible pairs (xs_0 and cng_0).

2. Run the subgenerators CNG and XS forward R+3 rounds, and calculate $M_{i-1} = Z_i - cng_i - xs_i$, $(1 <= i <= R+3)$. These are possible candidate values for the $Q[i-1 \bmod R]$ values.

3. If we guessed the initial values correctly in the first step, then M_0 should be the output $Q[0]$ (hence also the input to the calculation of $Q[R]$), and M_R should be the output $Q[0]$ after R steps. From these, we can calculate $carry_R$, and from $carry_R$ and M_{R+1} (which we hope equals output $Q[1]$) we can check whether, in fact, the computation given input and output $Q[1]$, and what we just calculated, agree with each other. If they don't, go back to step 1. If they do (which might happen randomly), do the same for $R+2$, and $R+3$. At this point, if all checks worked, with high probability we must have guessed CNG and XS correctly.

4. Run b32MWC backwards to recover the initial settings of Q[i].

The expected work to do this is about 2^{85} times the work to generate output words, which is a pretty normal measure. That is $2^{64}/2$ (since you expect to search half the values) times about 2^{22} words generated (because of the huge state space). The attack requires a little more than 2^{22} words of known keystream, again because of the size of the state. Another way to look at this is that it's about 2^{63} times the cost of doing a key setup (independent of how you actually set up that huge key!).

However this attack can be significantly optimized. First we note that most of the generated values of from CNG and XS are not used in the important validation step; of the $R + 3$ pairs of values generated, only the first three and last three are used. Secondly, we note that both generators can be jumped forward a known amount (in our case R

steps) with very much less complexity than simply evaluating the functions R times. In the case of XS, we can precompute the update matrix, and compute xs_R by multiplying xs_0 by it. Similarly, CNG can be jumped forward using a modular "square and multiply" exponentiation calculation.

When enumerating the possible initial values of XS and CNG, there is no reason to enumerate them as counters. Since the values all fall on cycles (a single cycle in both cases, given the corrected values in XS) the values can be enumerated in the sequence produced by the cycle. For example, we start with $cng_0 = 0$, and quickly calculate cng_R from that. By iterating CNG we easily calculate $cng_1..cng_3$ and $cng_{R+1}..cng_{R+3}$ as required above. After testing this value (and presumably failing), we can test the next value by discarding cng_0, using the old cng_1 as the new cng_0, similarly rolling over cng_R and so on. The cycle of XS can be handled in the same way. Only once the correct values for cng_0 and xs_0 have been identified is it necessary to run the generators R steps to recover the initial values of the Q array. Parallelization of this attack is a bit more tricky, but still easy. With this optimization, about 2^{63} operations on average can be expected to recover the key. Since the Q array is so large, this corresponds to about 2^{41} key setup operations.

7 Conclusion

We conclude that the KISS generator, unsurprisingly, should not be used in any context where cryptographic security is important. Its period is more than sufficient for tasks like simulations, so long as it is correctly initialized, but care should be taken with this initialization step. Its period (maximum, expected, and minimum) is not as long as its authors claim.

As a design rule, updating the component generators independently leads to easy analysis, which is both a good and a bad thing. Exchanging values between the various state registers might defeat the kind of attack we have applied, but it would also make any kind of rigorous analysis impossible, and might make the generator weaker in ways that are difficult to deduce but damaging in practice.

We would like to thank our colleague David Jacobson for help with the linear algebra, and the anonymous reviewers who provided suggestions for improvement.

Appendix: Cycles of the *SHR3* Generator

The register *SHR3* in the original KISS has a number of disjoint cycles, as shown in the following table. For each cycle, the hexadecimal value of the smallest value in the cycle is shown, along with the cycle length, and the characteristic polynomial of the shortest LFSR that generates the same bits as the least significant bit position in the word. The LFSR is found using the Berlekamp-Massey algorithm on some sample output. There are 12 distinct cycle lengths, and 16 distinct polynomials (LFSRs).

Cycle length	Value	LFSR Polynomial
1	0	0
1	0xaea21b8f	$1+x$
2	0x4655eac4	$1+x^2$
4	0x3aca807f	$1+x+x^2+x^3$
585	0x86c8	$1+x^2+x^5+x^6+x^9+x^{10}+x^{12}$
585	0x163de9	$1+x^2+x^5+x^6+x^9+x^{10}+x^{12}$
585	0x16bb21	$1+x+x^2+x^3+x^5+x^7+x^9+x^{11}+x^{12}+x^{13}$
585	0x2479f9	$1+x+x^2+x^3+x^5+x^7+x^9+x^{11}+x^{12}+x^{13}$
585	0x32c2d8	$1+x+x^2+x^3+x^5+x^7+x^9+x^{11}+x^{12}+x^{13}$
585	0x4305a3	$1+x+x^2+x^3+x^5+x^7+x^9+x^{11}+x^{12}+x^{13}$
585	0x43836b	$1+x^2+x^5+x^6+x^9+x^{10}+x^{12}$
585	0x55384a	$1+x^2+x^5+x^6+x^9+x^{10}+x^{12}$
585	0x677c5a	$1+x+x^2+x^3+x^5+x^7+x^9+x^{11}+x^{12}+x^{13}$
585	0x67fa92	$1+x^2+x^5+x^6+x^9+x^{10}+x^{12}$
585	0x835009	$1+x^2+x^5+x^6+x^9+x^{10}+x^{12}$
585	0x956de0	$1+x+x^2+x^3+x^5+x^7+x^9+x^{11}+x^{12}+x^{13}$
585	0xa729f0	$1+x^2+x^5+x^6+x^9+x^{10}+x^{12}$
585	0xf211ba	$1+x+x^2+x^3+x^5+x^7+x^9+x^{11}+x^{12}+x^{13}$
1170	0xc4e50	$1+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{14}$
1170	0xcc898	$1+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{14}$
1170	0x1a73b9	$1+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{14}$
1170	0x1af571	$1+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{14}$
1170	0x28b161	$1+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{14}$
1170	0x7d0fe3	$1+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{14}$
1170	0x7d892b	$1+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{14}$
2340	0x1883	$1+x+x^4+x^{12}+x^{14}+x^{15}$
2340	0xc56d3	$1+x+x^4+x^{12}+x^{14}+x^{15}$
2340	0xcd01b	$1+x+x^4+x^{12}+x^{14}+x^{15}$
2340	0x16a3a2	$1+x+x^4+x^{12}+x^{14}+x^{15}$
2340	0x1aedf2	$1+x+x^4+x^{12}+x^{14}+x^{15}$
2340	0x24617a	$1+x+x^4+x^{12}+x^{14}+x^{15}$
2340	0x3e12c3	$1+x+x^4+x^{12}+x^{14}+x^{15}$
131071	0x2114a	$1+x+x^2+x^4+x^5+x^9+x^{10}+x^{11}+x^{14}+x^{16}+x^{17}$
131071	0xdfa	$1+x^3+x^4+x^6+x^9+x^{12}+x^{14}+x^{15}+x^{16}+x^{18}$
262142	0xc7de	$1+x+x^3+x^5+x^6+x^7+x^9+x^{10}+x^{12}$ $+x^{13}+x^{14}+x^{17}+x^{18}+x^{19}$
524284	0x13f0	$1+x^2+x^3+x^4+x^5+x^8+x^9+x^{11}+x^{12}$ $+x^{15}+x^{17}+x^{20}$
76676535	0x1	$1+x+x^3+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}$ $+x^{13}+x^{14}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{22}+x^{23}$ $+x^{24}+x^{25}+x^{26}+x^{27}+x^{28}+x^{29}$
76676535	0xd	$1+x^2+x^3+x^4+x^6+x^{18}+x^{20}+x^{30}$
76676535	0x12	$1+x+x^3+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}$ $+x^{13}+x^{14}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{22}+x^{23}$ $+x^{24}+x^{25}+x^{26}+x^{27}+x^{28}+x^{29}$

(continued)

Cycle length	Value	LFSR Polynomial
76676535	0x13	$1+x^2+x^3+x^4+x^6+x^{18}+x^{20}+x^{30}$
76676535	0x1e	$1+x^2+x^3+x^4+x^6+x^{18}+x^{20}+x^{30}$
76676535	0x1f	$1+x+x^3+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}$ $+x^{13}+x^{14}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{22}+x^{23}$ $+x^{24}+x^{25}+x^{26}+x^{27}+x^{28}+x^{29}$
76676535	0x26	$1+x^2+x^3+x^4+x^6+x^{18}+x^{20}+x^{30}$
76676535	0x27	$1+x+x^3+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}$ $+x^{13}+x^{14}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{22}+x^{23}$ $+x^{24}+x^{25}+x^{26}+x^{27}+x^{28}+x^{29}$
76676535	0x38	$1+x^2+x^3+x^4+x^6+x^{18}+x^{20}+x^{30}$
76676535	0x39	$1+x+x^3+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}$ $+x^{13}+x^{14}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{22}+x^{23}$ $+x^{24}+x^{25}+x^{26}+x^{27}+x^{28}+x^{29}$
76676535	0x43	$1+x+x^3+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}$ $+x^{13}+x^{14}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{22}+x^{23}$ $+x^{24}+x^{25}+x^{26}+x^{27}+x^{28}+x^{29}$
76676535	0x50	$1+x+x^3+x^6+x^7+x^8+x^9+x^{10}+x^{11}+x^{12}$ $+x^{13}+x^{14}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{22}+x^{23}$ $+x^{24}+x^{25}+x^{26}+x^{27}+x^{28}+x^{29}$
76676535	0x77	$1+x^2+x^3+x^4+x^6+x^{18}+x^{20}+x^{30}$
76676535	0xae	$1+x^2+x^3+x^4+x^6+x^{18}+x^{20}+x^{30}$
153353070	0x11	$1+x+x^2+x^5+x^6+x^7+x^{18}+x^{19}+x^{20}+x^{21}+x^{30}+x^{31}$
153353070	0x2	$1+x+x^2+x^5+x^6+x^7+x^{18}+x^{19}+x^{20}+x^{21}+x^{30}+x^{31}$
153353070	0x3	$1+x+x^2+x^5+x^6+x^7+x^{18}+x^{19}+x^{20}+x^{21}+x^{30}+x^{31}$
153353070	0xe	$1+x+x^2+x^5+x^6+x^7+x^{18}+x^{19}+x^{20}+x^{21}+x^{30}+x^{31}$
153353070	0xf	$1+x+x^2+x^5+x^6+x^7+x^{18}+x^{19}+x^{20}+x^{21}+x^{30}+x^{31}$
153353070	0x1c	$1+x+x^2+x^5+x^6+x^7+x^{18}+x^{19}+x^{20}+x^{21}+x^{30}+x^{31}$
153353070	0x24	$1+x+x^2+x^5+x^6+x^7+x^{18}+x^{19}+x^{20}+x^{21}+x^{30}+x^{31}$
306706140	0x4	$1+x^3+x^5+x^8+x^{18}+x^{22}+x^{30}+x^{32}$
306706140	0x6	$1+x^3+x^5+x^8+x^{18}+x^{22}+x^{30}+x^{32}$
306706140	0x8	$1+x^3+x^5+x^8+x^{18}+x^{22}+x^{30}+x^{32}$
306706140	0x9	$1+x^3+x^5+x^8+x^{18}+x^{22}+x^{30}+x^{32}$
306706140	0xa	$1+x^3+x^5+x^8+x^{18}+x^{22}+x^{30}+x^{32}$
306706140	0xb	$1+x^3+x^5+x^8+x^{18}+x^{22}+x^{30}+x^{32}$
306706140	0x2c	$1+x^3+x^5+x^8+x^{18}+x^{22}+x^{30}+x^{32}$

References

1. Marsaglia, G., Zaman, A.: The KISS generator. Technical report, Department of Statistics, Florida State University, FL, USA (1993)
2. Marsaglia, G.: Re: Random numbers in C: Some suggestions, Message-ID 369B9AE9.52C98810@stat.fsu.edu, various newsgroups including sci.crypt (1999)
3. Marsaglia, G.: Random numbers for C: The END?, Message-ID 36A5FC62.17C9CC33@stat.fsu.edu in newsgroups sci.math and sci.stat.math (1999)

4. Marsaglia, G.: RNGs with periods exceeding 10(40million), Message-ID 603ebe15-a32f-4fbb-ba44-6c73f7919a33@t35g2000yqj.googlegroups.com in newsgroups sci.math, comp.lang.c and sci.crypt (2011)
5. McQueen, C.: Private communication
6. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998)
7. Plumstead, J.: Inferring a sequence generated by a linear congruence. In: Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science, pp. 153–159. IEEE, New York (1982)
8. Hong, J., Sarkar, P.: Rediscovery of time memory tradeoffs, IACR Eprint Report 2005/090. http://eprint.iacr.org/2005/090 (2005)
9. Hawkes, P., Rose, G.: A Practical Cryptanalysis of SSC2, Selected Areas in Cryptography 2001, LNCS